CSE 1325: Object-Oriented Programming

Lecture 8 – Chapter 11

Custom Input / Output, UML Activity Diagram, and the Decorator Pattern

Mr. George F. Rice

george.rice@uta.edu

Based on material by Bjarne Stroustrup
www.stroustrup.com/Programming

ERB 402
Office Hours:
Tuesday Thursday 11 - 12
Or by appointment

Quick Review

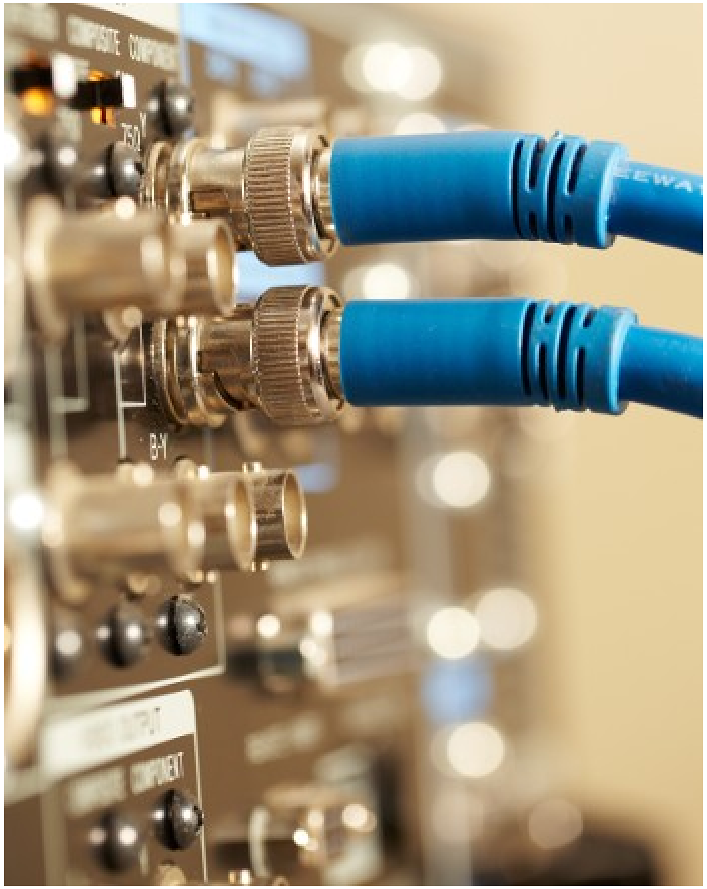
- What is multiple inheritance? **A derived class inheriting from two or more base classes**
- How does UML model multiple inheritance? **Simply draw an inheritance (“implements”) arrow from the derived class to each base class**
- How is multiple inheritance specified in C++? **Simply list the base classes in comma-separated sequence after the class declaration**
- Explain the “Diamond Problem”. **A common “grandparent” makes references from the derived class to certain inherited members ambiguous**
  - How does explicit base class method calls help? **If the ambiguity lies in which base class defines the reference, explicit specification resolves the ambiguity**
  - How does virtual inheritance help? **If the ambiguity lies in multiple parents inheriting the same member from their shared grandparent, virtual inheritance ensures only one block of memory is allocated for the shared grandparent’s data**
- True or **False**: The layout of C++ objects in memory is defined in the language standard.

Quick Review

- A named sequence of bytes available via the operating system is called a **file**.
- To parse a file, we must know its **name** and **data format**.
- Throwing **runtime_error** reports an generic error that occurred during the time the program was running. The parameter is a **string** that describes the error.
- When reading from a stream, what do these statuses mean?
  - good() - **success, data is available**
  - eof () - **no more data is available**
  - bad() - **a (likely) unrecoverable error occurred**
  - fail() - **a (likely) recoverable error occurred**
- The end of file keystroke is ^**Z** in Windows and ^**D** in Mac / Linux.
- The **exceptions()** method of the stream allows us to specify a mask enabling the throwing of exceptions for desired stream events.

Overview

- Text Formatting
  - Manipulators
  - String Streams
  - Characters
- Files
  - Open Modes
  - Text vs Binary
  - Random Access
- Decorator Pattern
- UML Activity Diagram



Types of (Data) I/O

- Individual values
  - See Chapters 4, 10
- Streams
  - See Chapters 10-11
- Graphics and GUI
  - See Chapters 12-16
- Text
  - Type driven, formatted
  - Line oriented
  - Individual characters
- Numeric
  - Integer
  - Floating point
  - User-defined types

Streams vs printf / scanf
(Adapted from the C++ FAQ)

- Compared to printf and scanf, streams are
  - **More type-safe**: The object type is known at compile time, while “%” fields are evaluated at runtime
  - **Less error prone**: Streams require no redundant “%” tokens that must align with the object types
  - **Extensible**: Streams are easily and uniquely defined for each new class. Imagine the chaos if every class defined it own incompatible “%” fields!
  - **Inheritable**: Streams belong to a class hierarchy, meaning anything can be treated as a stream
- Printf / scanf are
  - Significantly faster in some cases (but see premature optimization)

https://isocpp.org/wiki/faq/input-output#iostream-vs-stdio

A Stroustrup Observation

- As programmers we prefer regularity and simplicity
  - But, our job is to meet people’s expectations
- People are very fussy, and some very particular, and some downright *picky* about the way their output looks
  - They often have good reasons to be
  - Convention and tradition rules – domain-specific vocabularies
    - What does 110 mean?
    - What does 123,456 mean?
    - What does (123) mean?
  - The world of output formats is weirder than you could possibly imagine

Output formats

- Integer values
  - **1234** (decimal)
  - **2322** (octal)
  - **4d2** (hexadecimal)
- Floating point values
  - **1234.57** (general)
  - **1.2345678e+03** (scientific)
  - **1234.567890** (fixed)
- Precision (for floating-point values)
  - **1234.57** (precision 6)
  - **1234.6** (precision 5)
- Fields
  - **|12|** (default for **|** followed by **12** followed by **|**)
  - **| 12|** (**12** in a field of 4 characters)

Numerical Base Output

dec hex oct

- You can change “base”
  - Base 10 == decimal; digits: 0 1 2 3 4 5 6 7 8 9
  - Base 8 == octal; digits: 0 1 2 3 4 5 6 7
  - Base 16 == hexadecimal; digits: 0 1 2 3 4 5 6 7 8 9 a b c d e f

```
// simple test:
    cout << dec << 1234 << "\t(decimal)\n"
         << hex << 1234 << "\t(hexadecimal)\n"
         << oct << 1234 << "\t(octal)\n";
// The '\t' character is a “tab”
```

- Results

```
1234    (decimal)
4d2     (hexadecimal)
2322    (octal)
```

“Sticky” Manipulators

- You can change “base”
  - Base 10 == decimal; digits: 0 1 2 3 4 5 6 7 8 9
  - Base 8 == octal; digits: 0 1 2 3 4 5 6 7
  - Base 16 == hexadecimal; digits: 0 1 2 3 4 5 6 7 8 9 a b c d e f

```
// simple test:
    cout << 1234 << '\t'
         << hex << 1234 << '\t'
         << oct << 1234 << '\n';
    cout << 1234 << '\n'; // the octal base is still in effect
```

- Results

```
1234      4d2      2322
2322
```

Most manipulators are “sticky”, and remain in effect until changes. A few are transient, and only affect the next output. “A few” may mean “just setw”, though.

Other Manipulators

showbase noshowbase

- You can change “base”
  - Base 10 == decimal; digits: 0 1 2 3 4 5 6 7 8 9
  - Base 8 == octal; digits: 0 1 2 3 4 5 6 7
  - Base 16 == hexadecimal; digits: 0 1 2 3 4 5 6 7 8 9 a b c d e f

```
// simple test:
    cout << 1234 << '\t'
         << hex << 1234 << '\t'
         << oct << 1234 << endl;
    cout << showbase << dec;      // show bases via prefix
    cout << 1234 << '\t'
         << hex << 1234 << '\t'
         << oct << 1234 << '\n';
```

- The opposite of showbase is noshowbase

- Results

```
1234    4d2     2322
1234    0x4d2   02322
```

hex   octal

Floating-point Manipulators

defaultfloat scientific fixed

- You can change floating-point output format
  - **defaultfloat – iostream** chooses best format using **n** digits (default)
  - **scientific** – one digit before the decimal point plus exponent; **n** digits after **.**
  - **fixed** – no exponent; **n** digits after the decimal point

```
// simple test:
cout << 1234.56789 << "\t(defaultfloat)\n"
     << fixed << 1234.56789 << "\t(fixed)\n"
     << scientific << 1234.56789 << "\t(scientific)\n";
```

- Results

```
1234.57          (defaultfloat)
1234.567890      (fixed)
1.234568e+03     (scientific)
```

Precision Manipulator

setprecision(digits)

- Precision (the default is 6) from <iomanip>
  - **defaultfloat** – precision is the number of digits
  - **scientific** – precision is the number of digits after the . (dot)
  - **fixed** – precision is the number of digits after the . (dot)

```
// example:
    cout << 1234.56789 << '\t' << fixed << 1234.56789 << '\t'
         << scientific << 1234.56789 << '\n';
    cout << general << setprecision(5)
         << 1234.56789 << '\t' << fixed << 1234.56789 << '\t'
         << scientific << 1234.56789 << '\n';
    cout << general << setprecision(8)
         << 1234.56789 << '\t' << fixed << 1234.56789 << '\t'
         << scientific << 1234.56789 << '\n';
```

- Results (note the rounding):

```
1234.57        1234.567890     1.234568e+03
1234.6         1234.56789      1.23457e+03
1234.5679      1234.56789000   1.23456789e+03
```

Output field width

setw(min_width)

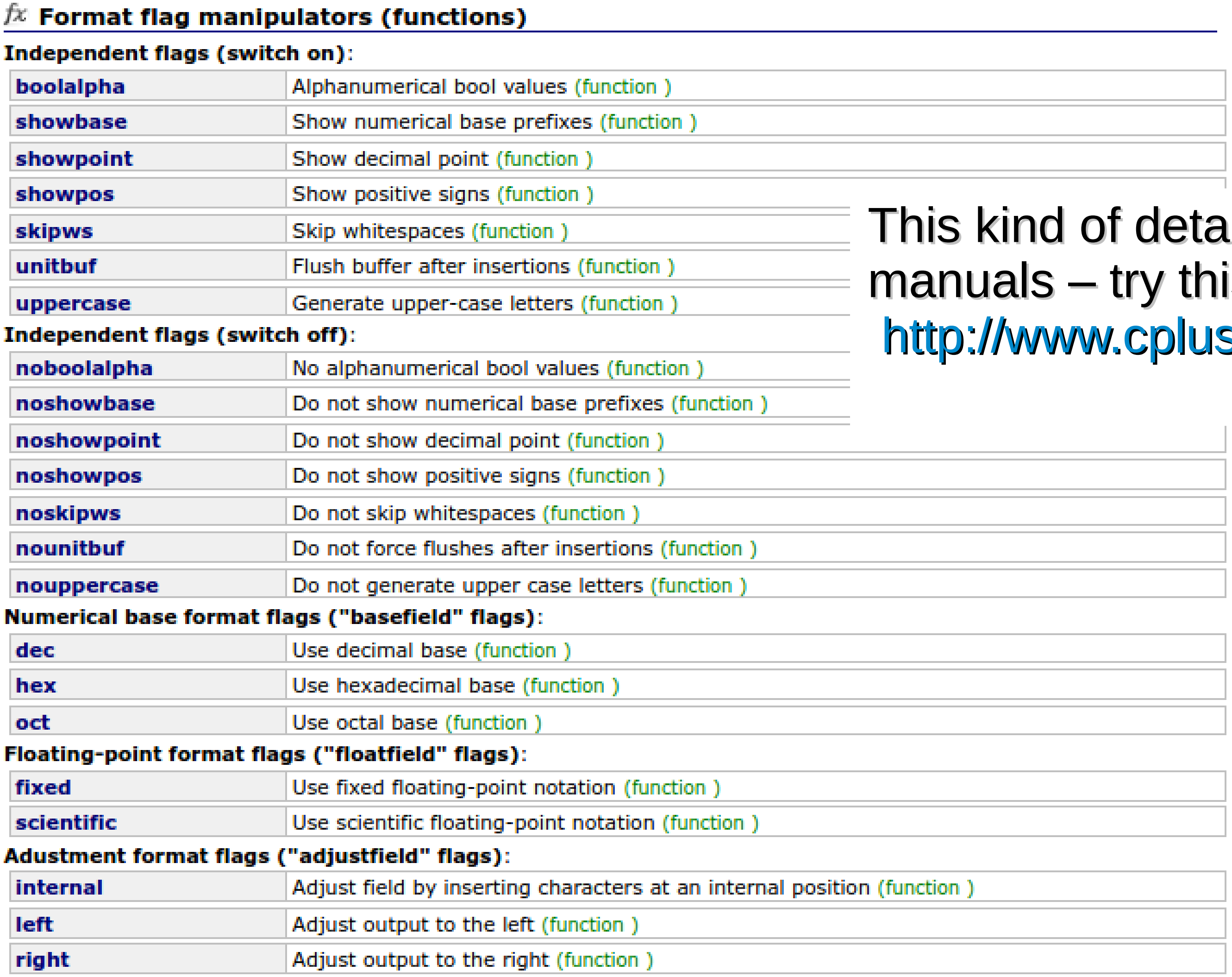
- Width is the number of characters to be used for the next output operation
  - **Beware:** width is transient and applies to next output only
    (it doesn't "stick" like precision, base, and floating-point format)
  - **Beware:** output is never truncated to fit into field
    - (better a bad format than a bad value)

```
#include <iomanip>
    cout << 123456 <<'|'<< setw(4) << 123456 << '|'
         << setw(8) << 123456 << '|' << 123456 << "|\n";
    cout << 1234.56 <<'|'<< setw(4) << 1234.56 << '|'
         << setw(8) << 1234.56 << '|' << 1234.56 << "|\n";
    cout << "asdfgh" <<'|'<< setw(4) << "asdfgh" << '|'
         << setw(8) << "asdfgh" << '|' << "asdfgh" << "|\n";
```

- Results

```
123456|123456|  123456|123456|
1234.56|1234.56| 1234.56|1234.56|
asdfgh|asdfgh|  asdfgh|asdfgh|
```

Observation



Format flag manipulators (functions)

Independent flags (switch on):

boolalpha	Alphanumerical bool values (function)
showbase	Show numerical base prefixes (function)
showpoint	Show decimal point (function)
showpos	Show positive signs (function)
skipws	Skip whitespaces (function)
unitbuf	Flush buffer after insertions (function)
uppercase	Generate upper-case letters (function)

Independent flags (switch off):

noboolalpha	No alphanumerical bool values (function)
noshowbase	Do not show numerical base prefixes (function)
noshowpoint	Do not show decimal point (function)
noshowpos	Do not show positive signs (function)
noskipws	Do not skip whitespaces (function)
nounitbuf	Do not force flushes after insertions (function)
nouppercase	Do not generate upper case letters (function)

Numerical base format flags ("basefield" flags):

dec	Use decimal base (function)
hex	Use hexadecimal base (function)
oct	Use octal base (function)

Floating-point format flags ("floatfield" flags):

fixed	Use fixed floating-point notation (function)
scientific	Use scientific floating-point notation (function)

Adustment format flags ("adjustfield" flags):

internal	Adjust field by inserting characters at an internal position (function)
left	Adjust output to the left (function)
right	Adjust output to the right (function)

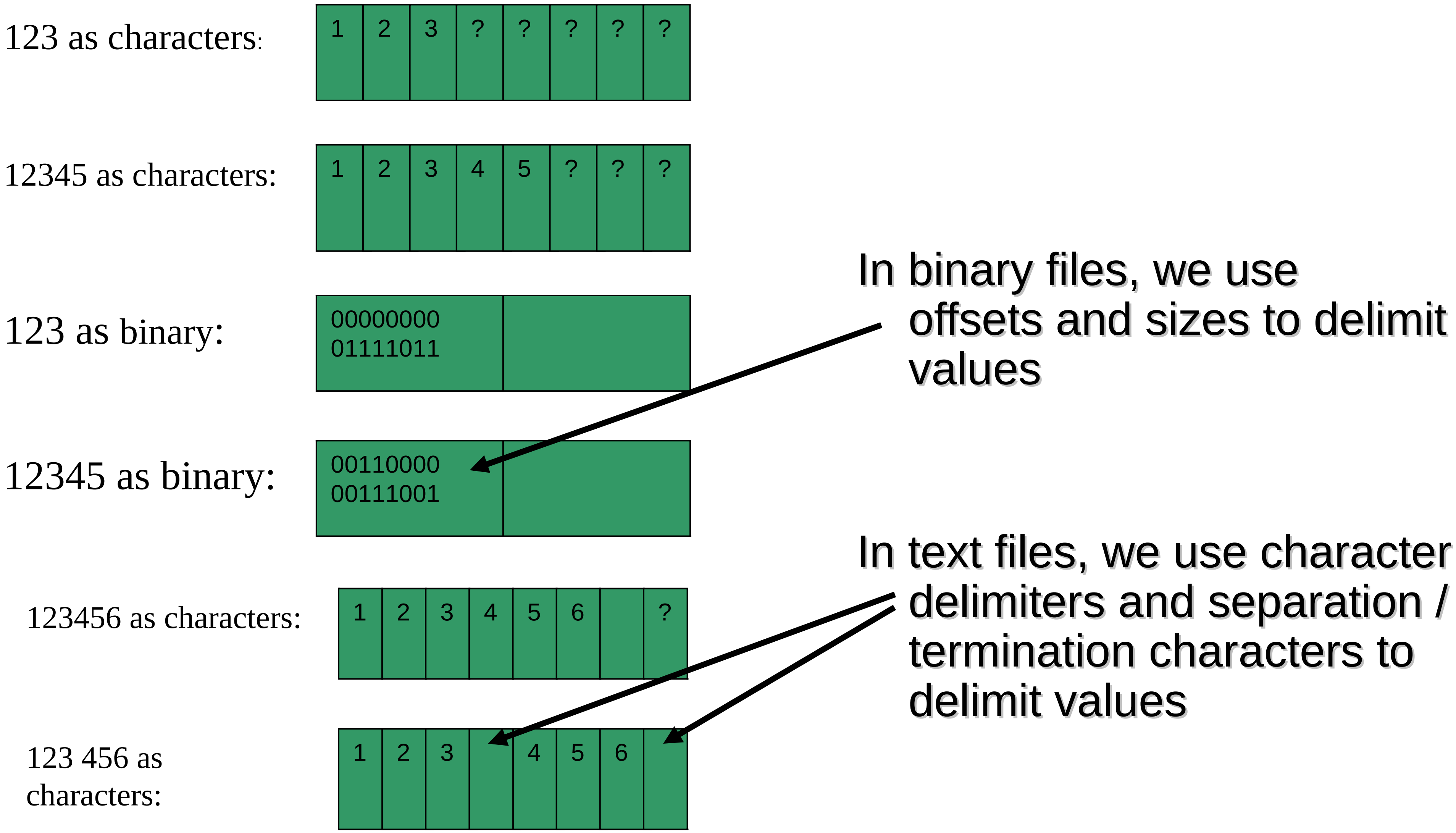
This kind of detail is why you need (online) manuals – try this one:

http://www.cplusplus.com/reference/ios/

File open modes

- By default, an **ifstream** opens its file for reading
- By default, an **ofstream** opens its file for writing
- Alternatives:
  - **ios_base::app** *// append (i.e., output adds to the end of the file)*
  - **ios_base::ate** *// "at end" (open and seek to end)*
  - **ios_base::binary** *// binary mode – beware of system specific behavior*
  - **ios_base::in** *// for reading*
  - **ios_base::out** *// for writing*
  - **ios_base::trunc** *// truncate file to 0-length*
- A file mode is optionally specified after the name of the file:
  - **ofstream of1 {name1};** *// defaults to ios_base::out*
  - **ifstream if1 {name2};** *// defaults to ios_base::in*
  - **ofstream ofs {name, ios_base::app};** *// append rather than overwrite*
  - **fstream fs {"myfile", ios_base::in | ios_base::out};** *// both in and out*

Text vs. binary files



123 as characters:

1	2	3	?	?	?	?	?

12345 as characters:

1	2	3	4	5	?	?	?

123 as binary:

00000000 01111011	

12345 as binary:

00110000 00111001	

123456 as characters:

1	2	3	4	5	6		?

123 456 as characters:

1	2	3		4	5	6	

In binary files, we use offsets and sizes to delimit values

In text files, we use character delimiters and separation / termination characters to delimit values

Text vs. binary

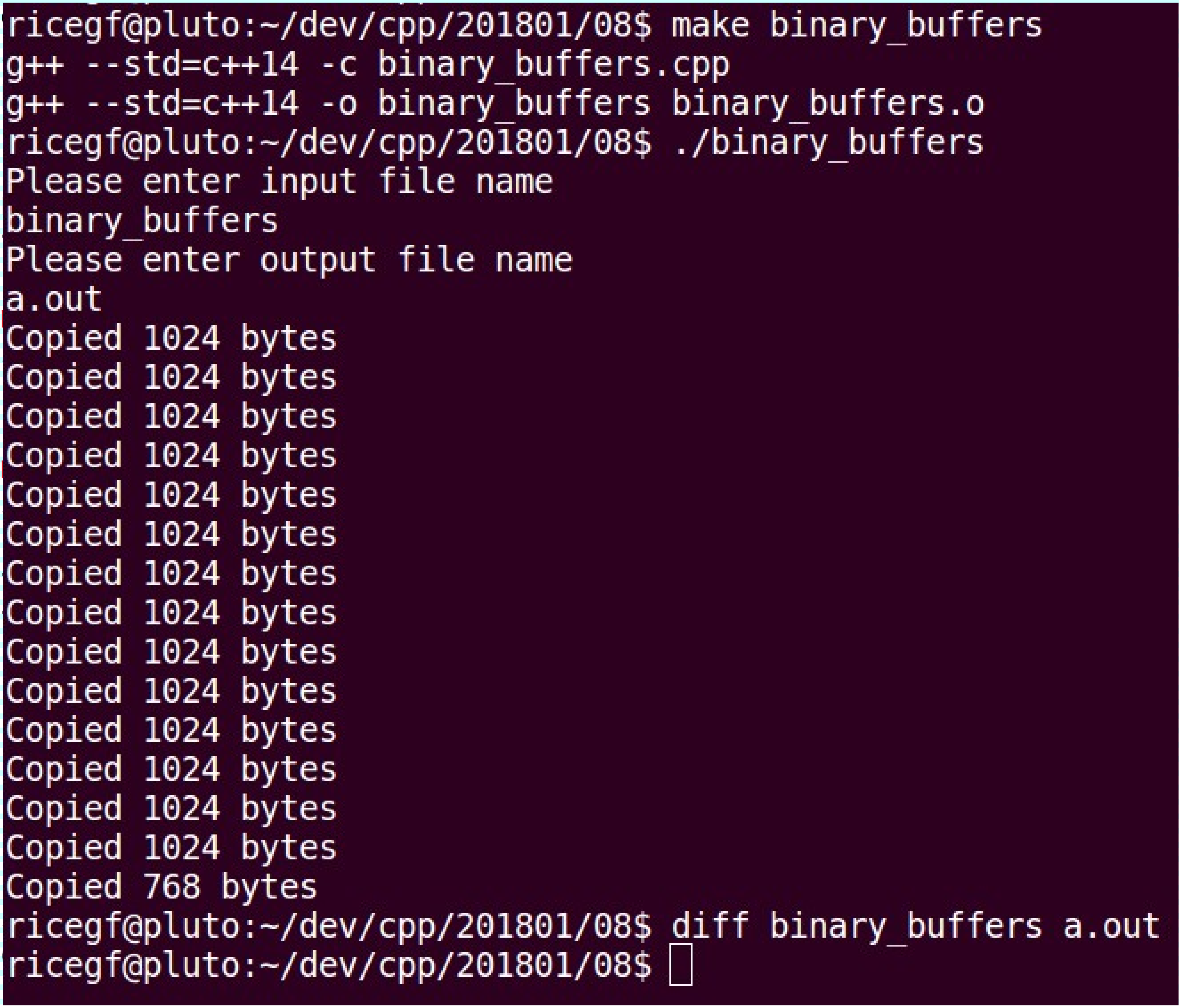
- Use text whenever possible
  - You can read it (without a fancy program)
  - You can debug your programs more easily
  - Text is portable across different systems
  - Size (compressed) is typically comparable
  - Most information can be represented reasonably as text
- Use binary when you must
  - E.g. image files, sound files for faster decoding
  - Compressed and / or encrypted files

Buffered Binary File I/O

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

int main() {
    const int BUFFER_SIZE = 1024;
    string filename;
    cout << "Please enter input file name\n"; getline(cin, filename);
    ifstream ifs {filename,ios_base::binary};  // note: binary
    if (!ifs) {cerr << "Can't open input file: aborted" << endl; return -1;}

    cout << "Please enter output file name\n"; getline(cin, filename);
    ofstream ofs {filename, ios_base::binary};  // note: binary
    if (!ofs) {cerr << "Can't open output file: aborted" << endl; return -2;}

    char buffer[BUFFER_SIZE];
    while(ifs) {
        ifs.read(buffer, BUFFER_SIZE);
        if (ifs.gcount()) {
            ofs.write(buffer, ifs.gcount());
            if (!ofs) {cerr << "File write error: aborted" << endl; return -4;}
        }
        cout << "Copied " << ifs.gcount() << " bytes" << endl;
    }
    if (!ifs.eof()) {cerr << "File read error: aborted" << endl; return -3;}
    return 0;
}
```

Buffered Binary File I/O

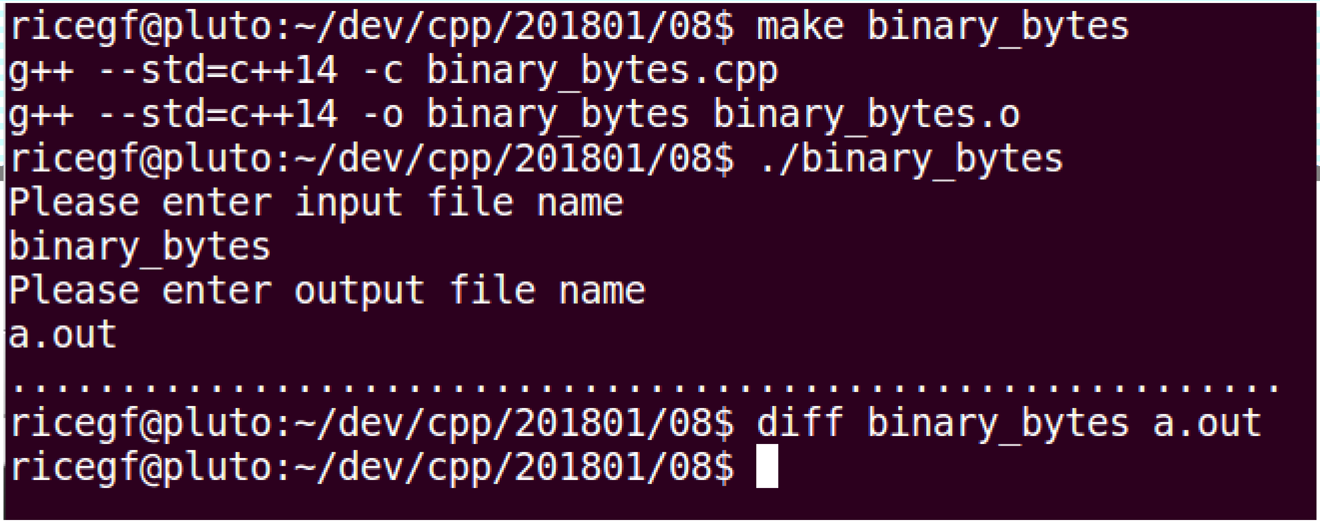
```
#include <i
#include <f
using names

int main()
    const i
    string
    cout <<
    ifstrea
    if (!if                                                    n -1;}

    cout <<
    ofstrea
    if (!of                                                    rn -2;}

    char bu
    while(i
        ifs
        if
                                                               turn -4;}
        }
        cou
    }
    if (!if                                                    n -3;}
    return
}
```



```
ricegf@pluto:~/dev/cpp/201801/08$ make binary_buffers
g++ --std=c++14 -c binary_buffers.cpp
g++ --std=c++14 -o binary_buffers binary_buffers.o
ricegf@pluto:~/dev/cpp/201801/08$ ./binary_buffers
Please enter input file name
binary_buffers
Please enter output file name
a.out
Copied 1024 bytes
Copied 1024 bytes
Copied 1024 bytes
Copied 1024 bytes
Copied 1024 bytes
Copied 1024 bytes
Copied 1024 bytes
Copied 1024 bytes
Copied 1024 bytes
Copied 1024 bytes
Copied 1024 bytes
Copied 1024 bytes
Copied 1024 bytes
Copied 1024 bytes
Copied 768 bytes
ricegf@pluto:~/dev/cpp/201801/08$ diff binary_buffers a.out
ricegf@pluto:~/dev/cpp/201801/08$ 
```

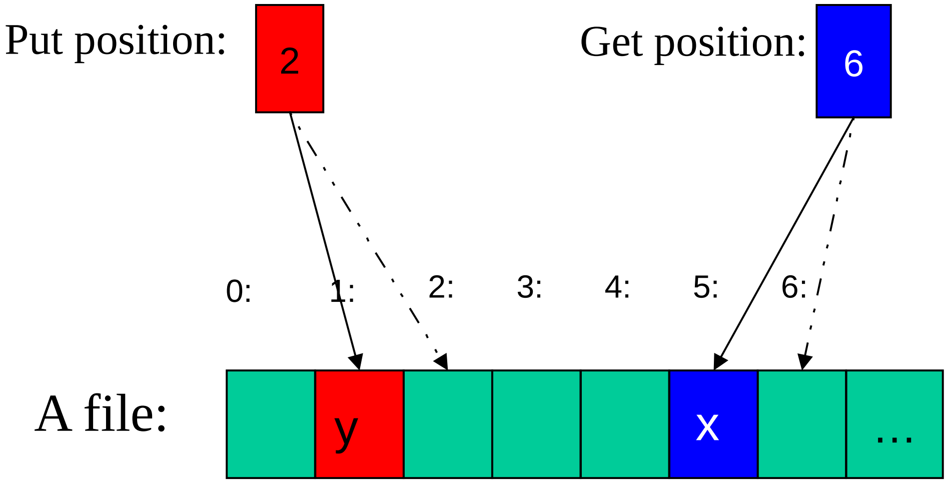
Binary File I/O by Bytes

```
// Same as before

    char byte;       // replaces buffer
    int counter = 0; // for status reporting
    while(ifs) {
        ifs.get(byte);
        if (ifs) {
            ofs.put(byte);
            if (!ofs) {cerr << "File write error: aborted" << endl; return -4;}
        }
        if (!(++counter % 256)) cout << ".";  // output '.' every 256 bytes
    }
    cout << endl;
    if (!ifs.eof()) {
        cerr << "File read error: aborted" << endl;
        return -3;
    }
    return 0;
}
```



```
ricegf@pluto:~/dev/cpp/201801/08$ make binary_bytes
g++ --std=c++14 -c binary_bytes.cpp
g++ --std=c++14 -o binary_bytes binary_bytes.o
ricegf@pluto:~/dev/cpp/201801/08$ ./binary_bytes
Please enter input file name
binary_bytes
Please enter output file name
a.out
..........................................................
ricegf@pluto:~/dev/cpp/201801/08$ diff binary_bytes a.out
ricegf@pluto:~/dev/cpp/201801/08$ 
```

Positioning in a filestream



```
fstream fs {name};   // open for input and output (C++ 11 and later)

fs.seekg(5); // move reading position (‘g’ for ‘get’) to 5 (the 6th character)
char ch;
fs.get(ch);      // read the x and increment the reading position to 6
cout << "sixth character is " << ch << '(' << int(ch) << ")\n";

fs.seekp(1); // move writing position (‘p’ for ‘put’) to 1 (the 2nd character)
fs.put('y');     // write and increment writing position to 2
```

Positioning

- Whenever you can
  - Use simple streaming
    - Streams/streaming is a very powerful metaphor
    - Write most of your code in terms of “plain” **istream** and **ostream**
    - Default backups for file modifications are fairly easy to implement, e.g., rename the old file with a trailing '~' and write the updated file to the original filename
  - Positioning is far more error-prone
    - Handling of the end of file position is system dependent and basically unchecked
    - A subtle bug can destroy the file being edited

String streams

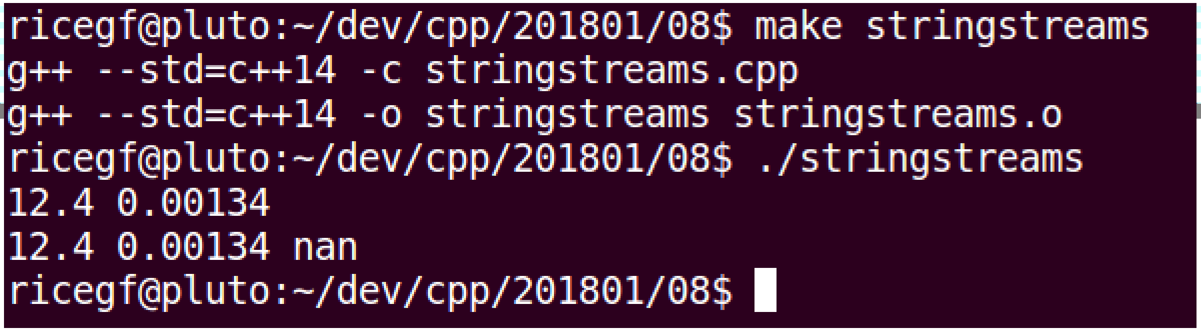
A **stringstream** (from <sstream>) reads/writes from/to a **string** rather than a file or a keyboard/screen.

This adds all stream capabilities to your string editing arsenal

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

double str_to_double(string s) {
    istringstream iss{s}; // make an input stream from s
    double d;
    iss >> d;             // stream a double from s
    if (!iss) throw runtime_error("double format error");
    return d;
}
string double_to_string(double d) {
    ostringstream oss;  // make a stream so that we can read from s
    oss << d;
    if (!oss) throw runtime_error("string format error");
    return oss.str();
}
```

String streams

```
int main() {
    double d1 = str_to_double("12.4");
    double d2 = str_to_double("1.34e-3");
    // double d3 = str_to_double("twelve point three"); // will throw exception

    string s1 = double_to_string(12.4);
    string s2 = double_to_string(1.34e-3);
    string s3 = double_to_string(NAN);

    cout << d1 << ' ' << d2 << endl;
    cout << s1 << ' ' << s2 << ' ' << s3 << endl;

    return 0;
}
```



```
ricegf@pluto:~/dev/cpp/201801/08$ make stringstreams
g++ --std=c++14 -c stringstreams.cpp
g++ --std=c++14 -o stringstreams stringstreams.o
ricegf@pluto:~/dev/cpp/201801/08$ ./stringstreams
12.4 0.00134
12.4 0.00134 nan
ricegf@pluto:~/dev/cpp/201801/08$ 
```

String streams

- String streams are very useful for
  - formatting into a fixed-sized space
    - Often useful for fields in a GUI dialog, e.g., converting a text entry field into a double
    - Any time you need to build a well-formatted string representation of an object
  - for extracting typed objects out of a string
    - Sometimes used with getline when you don’t know how many elements and what type is each in the input

http://www.cplusplus.com/reference/sstream/stringstream/

Type vs. line

- Read a whitespace-terminated string

```
string name;
cin >> name;     // input: Dennis Ritchie
cout << name << '\n';// output: Dennis
```

- Read a line

```
string name;
getline(cin, name);  // input: Dennis Ritchie
cout << name << '\n';// output: Dennis Ritchie

// now what? Maybe:

istringstream ss(name);
ss >> first_name;
ss >> second_name;
```

Reading Characters

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

int main() {
    char ch;
    cout << "(1) cin>> or (2) cin.get? ";
    cin >> ch;

    cin.ignore();

    if (ch == '1') for ( ; cin>>ch      && ch != 'x'; ) cout << ch;
    else           for ( ; cin.get(ch) && ch != 'x'; ) cout << ch;

    cout << endl;
    return 0;
}
```

For input “Hello there. How are you today?”...

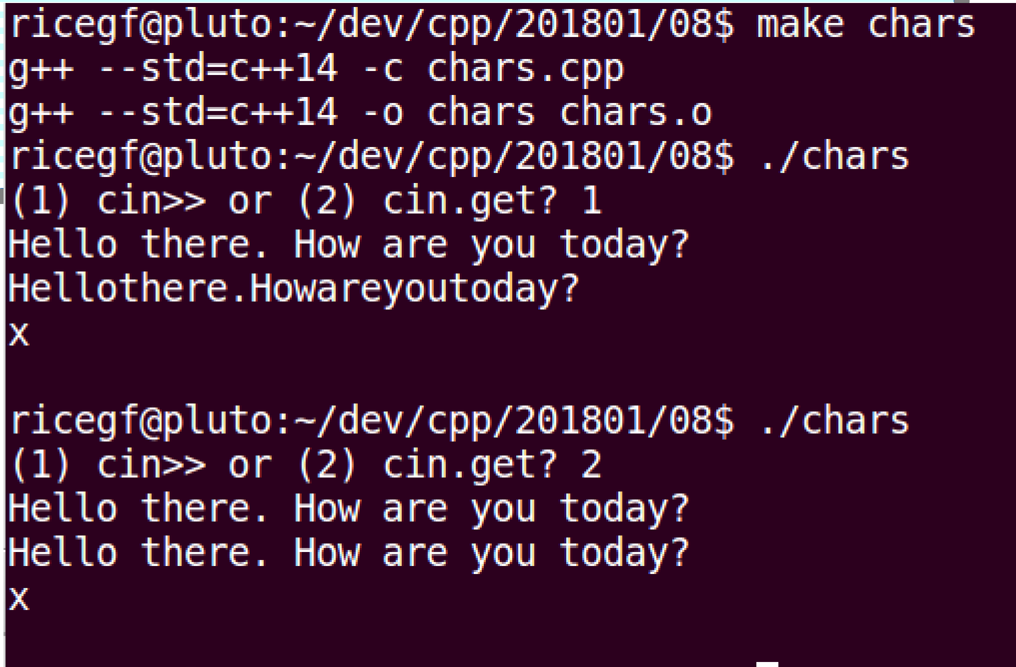
What is the output if “(1) cin>>ch” is selected?
What is the output if “(2) cin.get(ch)” is selected?

Reading Characters

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

int main() {
    char ch;
    cout << "(1) cin>> or (2) cin.get? ";
    cin >> ch;

    cin.ignore();

    if (ch == '1') for ( ; cin>>ch     && ch != 'x'; ) cout << ch;
    else           for ( ; cin.get(ch) && ch != 'x'; ) cout << ch;

    cout << endl;
    return 0;
}
```

For input “Hello there. How are you today?”...

What is the output if “(1) cin>>ch” is selected?
What is the output if “(2) cin.get(ch)” is selected?



```
ricegf@pluto:~/dev/cpp/201801/08$ make chars
g++ --std=c++14 -c chars.cpp
g++ --std=c++14 -o chars chars.o
ricegf@pluto:~/dev/cpp/201801/08$ ./chars
(1) cin>> or (2) cin.get? 1
Hello there. How are you today?
Hellothere.Howareyoutoday?
x

ricegf@pluto:~/dev/cpp/201801/08$ ./chars
(1) cin>> or (2) cin.get? 2
Hello there. How are you today?
Hello there. How are you today?
x
```

Character classification functions

- If you use character input, you often need one or more of these (from header **<cctype>**):
  - **isspace(c)** *// is **c** whitespace? (' ', '\t', '\n', etc.)*
  - **isalpha(c)** *// is **c** a letter? ('a'..'z', 'A'..'Z') note: not '_'*
  - **isdigit(c)** *// is **c** a decimal digit? ('0'..'9')*
  - **isupper(c)** *// is **c** an upper case letter?*
  - **islower(c)** *// is **c** a lower case letter?*
  - **isalnum(c)** *// is **c** a letter or a decimal digit?*

  etc.

http://cplusplus.com/reference/cctype/

Line-oriented input

- Prefer **>>** to **getline()**
  - i.e. avoid line-oriented input when you can

- People often use **getline()** because they see no alternative
  - But it easily gets messy
  - When trying to use **getline()**, you often end up
    - using **>>** to parse the line from a **stringstream**
    - using **get()** to read individual characters

```
int a, b;
while (infile >> a >> b)
{
    // process pair (a,b)
}
```

```
std::string line;
while (std::getline(infile, line))
{
    std::istringstream iss(line);
    int a, b;
    if (!(iss >> a >> b)) { break; } // error

    // process pair (a,b)
}
```

Code from http://stackoverflow.com/questions/7868936/read-file-line-by-line
Used under “non-commercial use” clause of the license agreement.

C++14 Literals

- Binary literals
  - **0b1010100100000011**
- Digit separators
  - **0b1010'1001'0000'0011**
  - Can also be used for for decimal, octal, and hexadecimal numbers
- User-Defined Literals (UDLs) in the standard library
  - Time: **2h+10m+12s+123ms+3456ns**
  - Complex: **2+4i**

Structural

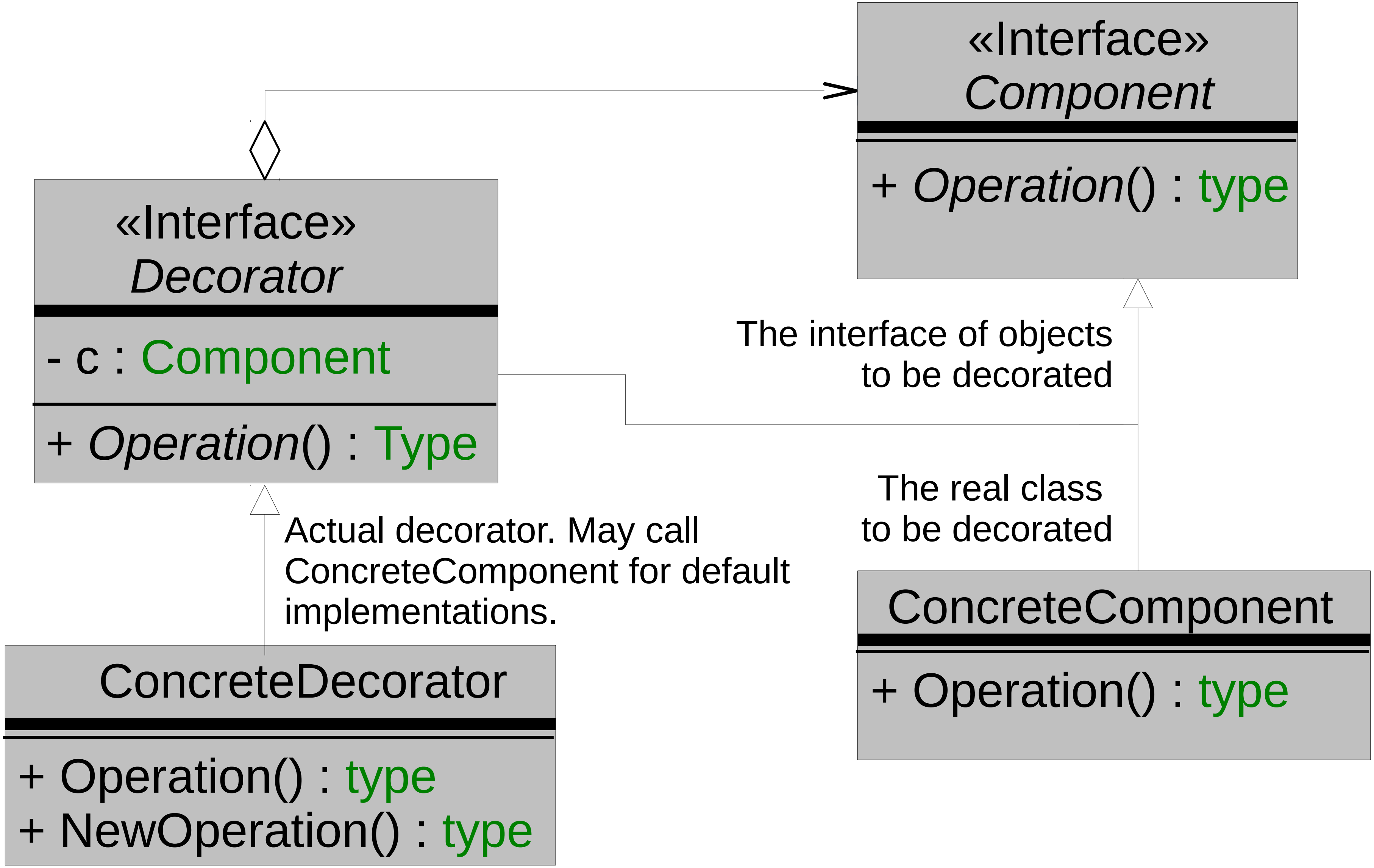
Decorator Pattern

- The **Decorator** pattern *dynamically* (at runtime) adds new functionality to an object without altering its structure
  - Distinct from inheritance, which is a *static* (at compile time) functionality addition
  - Decorator relies on composition to reuse the decorated class code, while adding additional code
  - Decorators are typically small, and overuse can impact supportability due to too many small similar classes

Structural

The Decorator Pattern

(Slightly Simplified)



«Interface»
Component

+ *Operation*() : type

«Interface»
Decorator

- c : Component

+ *Operation*() : Type

The interface of objects to be decorated

The real class to be decorated

Actual decorator. May call ConcreteComponent for default implementations.

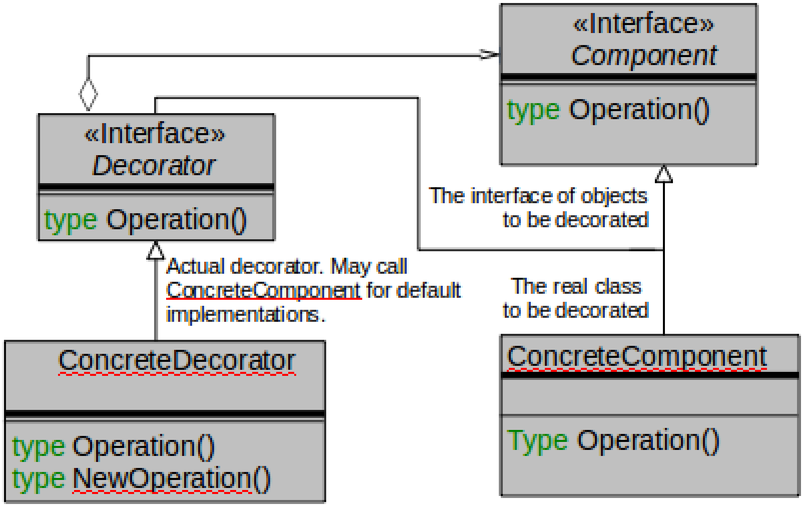
ConcreteDecorator

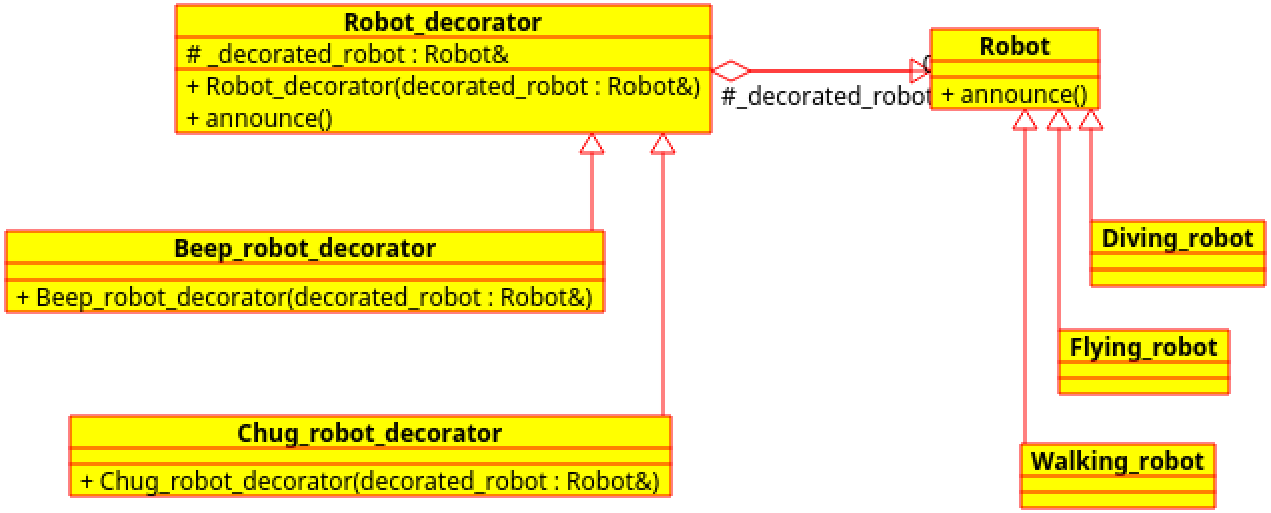
+ Operation() : type
+ NewOperation() : type

ConcreteComponent

+ Operation() : type

Decorator in C++



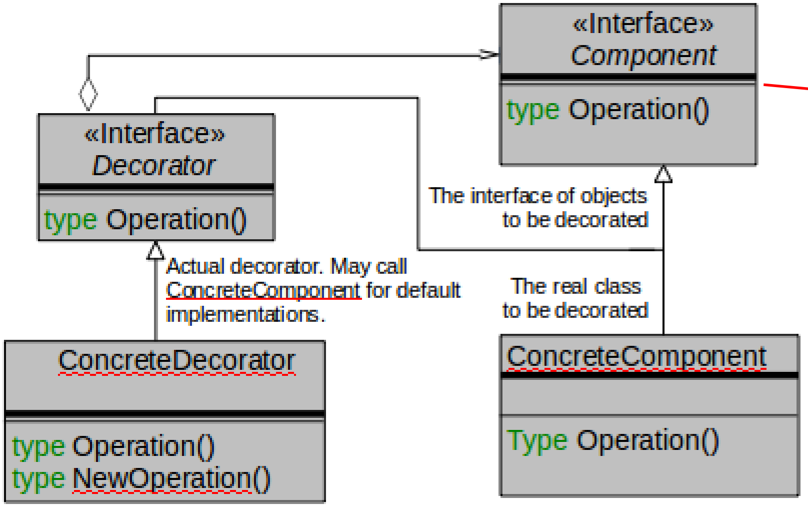

Robot_decorator
# _decorated_robot : Robot&
+ Robot_decorator(decorated_robot : Robot&)
+ announce()
#_decorated_robot
Robot
+ announce()
Beep_robot_decorator
+ Beep_robot_decorator(decorated_robot : Robot&)
Chug_robot_decorator
+ Chug_robot_decorator(decorated_robot : Robot&)
Diving_robot
Flying_robot
Walking_robot

Generated from C++ headers by Umbrello

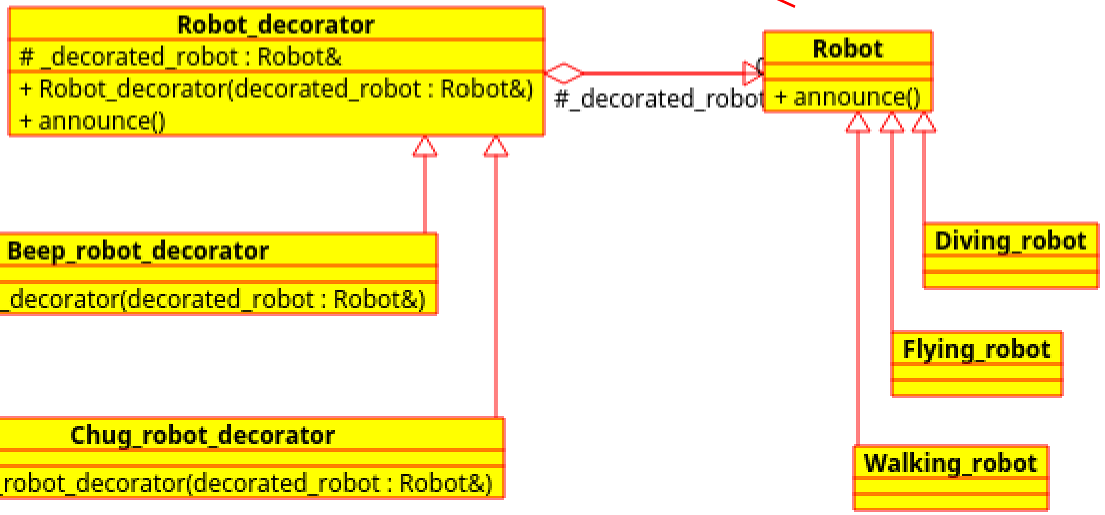
The Component Interface

(For the Classes to be Decorated)

Class Interface to be decorated



```
#ifndef __ROBOT_H
#define __ROBOT_H

#include <iostream>
using namespace std;

class Robot {
  public:
    virtual void announce() { }
};

#endif
```

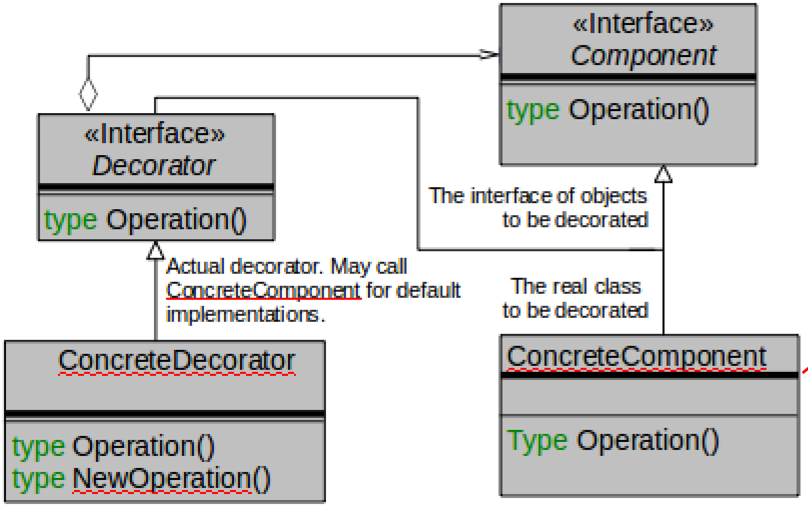

Robot_decorator
# _decorated_robot : Robot&
+ Robot_decorator(decorated_robot : Robot&)
+ announce()
#_decorated_robot
Robot
+ announce()
Beep_robot_decorator
+ Beep_robot_decorator(decorated_robot : Robot&)
Chug_robot_decorator
+ Chug_robot_decorator(decorated_robot : Robot&)
Diving_robot
Flying_robot
Walking_robot

A robot simply announces its presence with a phrase

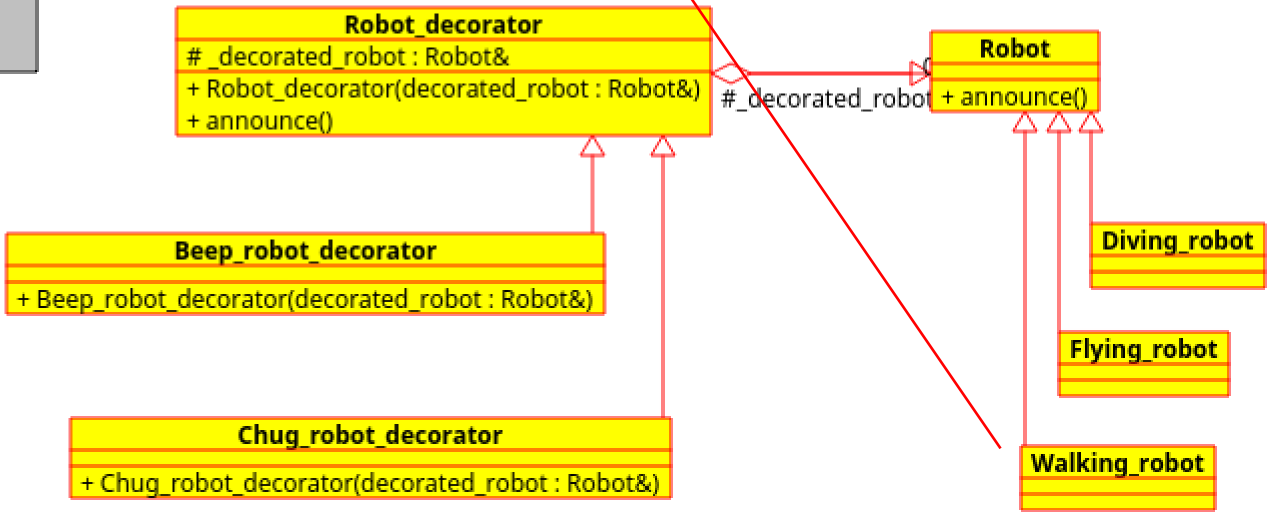
The Component Interface

(For the Classes to be Decorated)

Classes to be decorated



```
#include "robot.h"

class Walking_robot : public Robot {
  public:
    void announce() override;
};

void Walking_robot::announce() {
  cout << "Make way!" << endl;
}
```


Robot_decorator
# _decorated_robot : Robot&
+ Robot_decorator(decorated_robot : Robot&)
+ announce()
#_decorated_robot
Robot
+ announce()
Beep_robot_decorator
+ Beep_robot_decorator(decorated_robot : Robot&)
Chug_robot_decorator
+ Chug_robot_decorator(decorated_robot : Robot&)
Diving_robot
Flying_robot
Walking_robot

Walking robots say “Make way!”

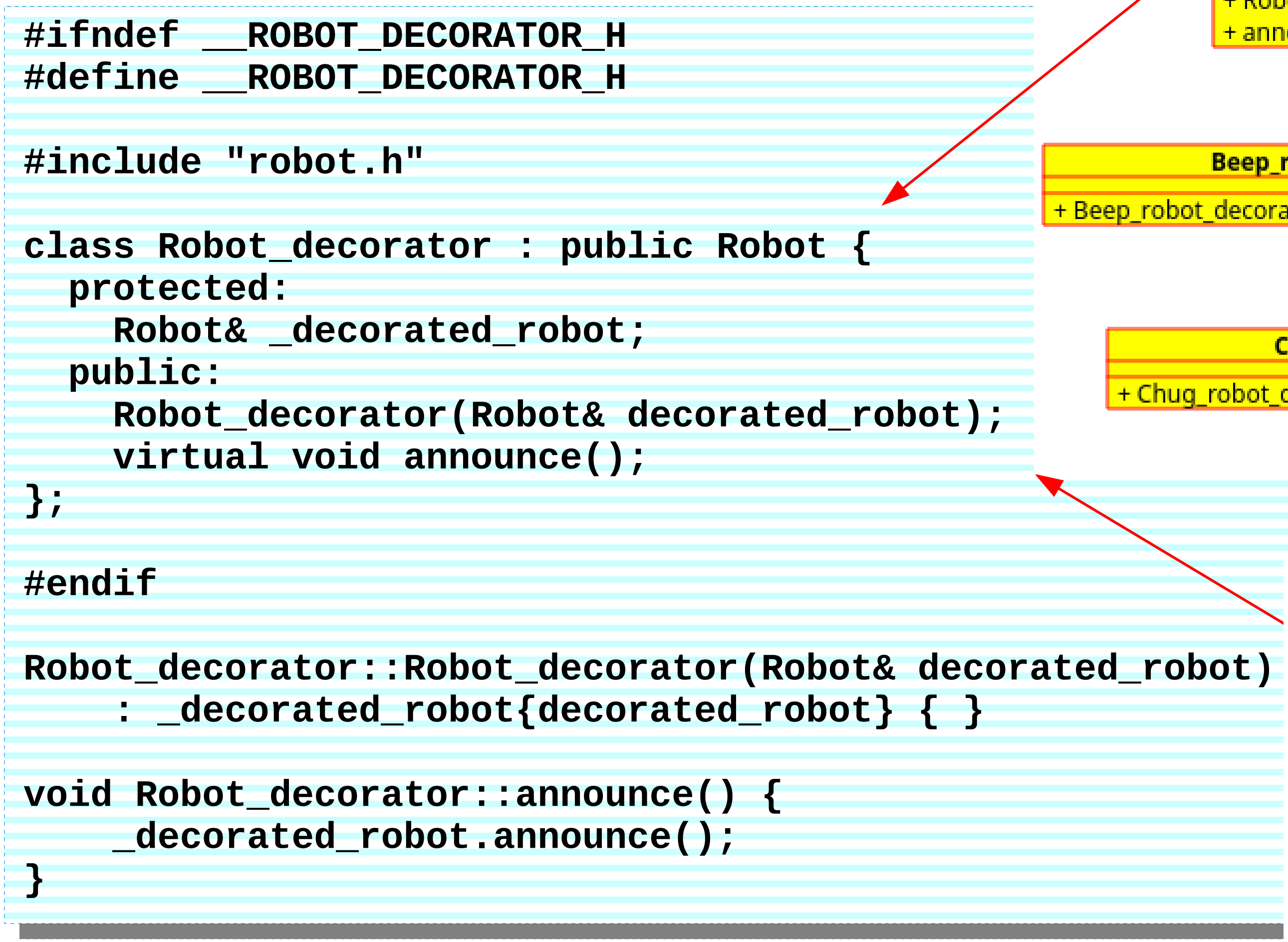
Flying robots say “Heads up!”

Diving robots say “Glub! Glub!”

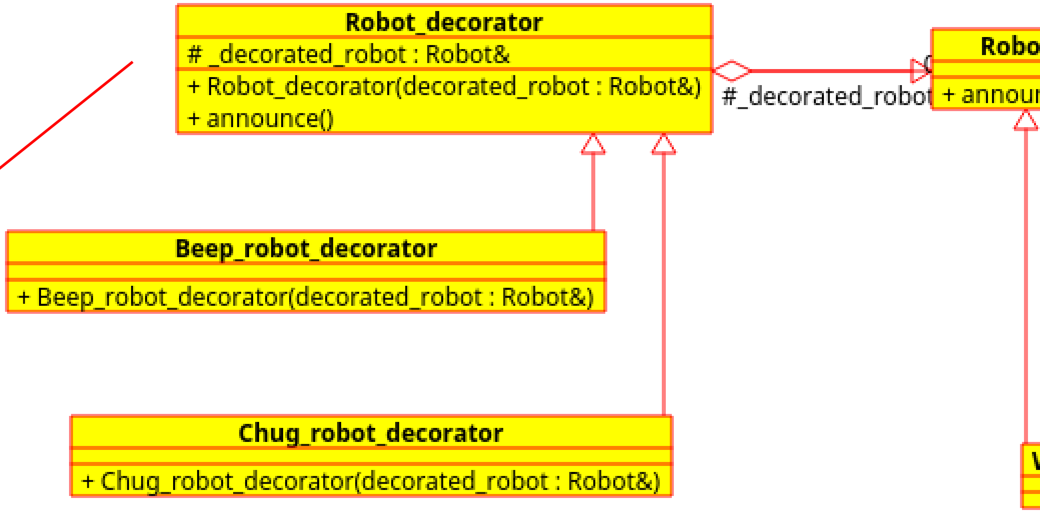
The Decorator Base Class

(from which many decorators may be derived)

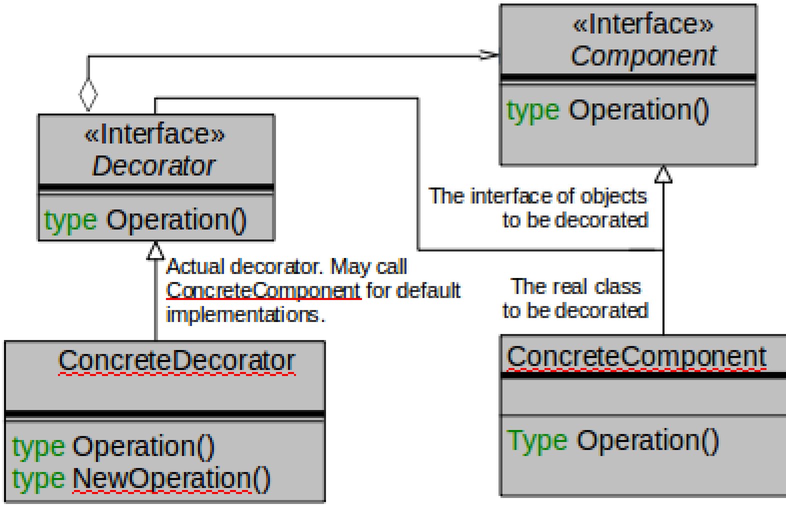
Decoration base class



```
#ifndef __ROBOT_DECORATOR_H
#define __ROBOT_DECORATOR_H

#include "robot.h"

class Robot_decorator : public Robot {
  protected:
    Robot& _decorated_robot;
  public:
    Robot_decorator(Robot& decorated_robot);
    virtual void announce();
};

#endif

Robot_decorator::Robot_decorator(Robot& decorated_robot)
    : _decorated_robot{decorated_robot} { }

void Robot_decorator::announce() {
    _decorated_robot.announce();
}
```



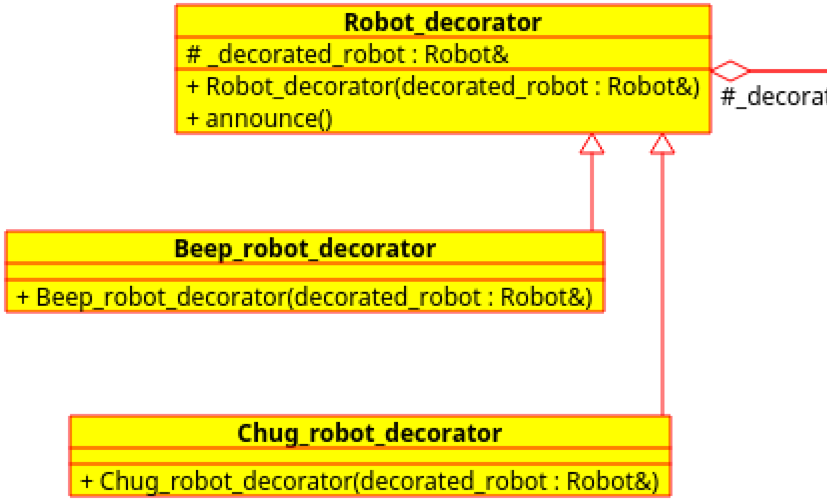
Robot_decorator
# _decorated_robot : Robot&
+ Robot_decorator(decorated_robot : Robot&)
+ announce()
#_decorated_robot
Robo
+ annour
Beep_robot_decorator
+ Beep_robot_decorator(decorated_robot : Robot&)
Chug_robot_decorator
+ Chug_robot_decorator(decorated_robot : Robot&)



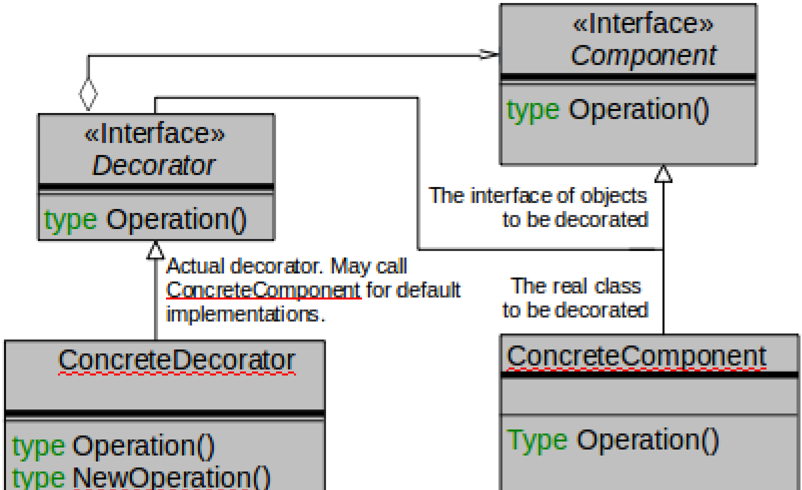
Decorator Classes

Decoration derived class

```
#ifndef __BEEP_ROBOT_DECORATOR
#define __BEEP_ROBOT_DECORATOR

#include "robot_decorator.h"

class Beep_robot_decorator : public Robot_decorator {
  public:
    Beep_robot_decorator(Robot& decorated_robot);
    void announce() override;
};

#endif

Beep_robot_decorator::Beep_robot_decorator(Robot& decorated_robot)
    : Robot_decorator(decorated_robot) { }

void Beep_robot_decorator::announce() {
    _decorated_robot.announce();
    cout << "Beep! Beep!" << endl;
}
```



Robot_decorator
# _decorated_robot : Robot&
+ Robot_decorator(decorated_robot : Robot&)
+ announce()
#_decora
Beep_robot_decorator
+ Beep_robot_decorator(decorated_robot : Robot&)
Chug_robot_decorator
+ Chug_robot_decorator(decorated_robot : Robot&)



The Beep decorator adds “Beep! Beep!” to the robot’s announcement.

The Chug decorator adds “Chug! Chug!”

Using the Decorated Classes

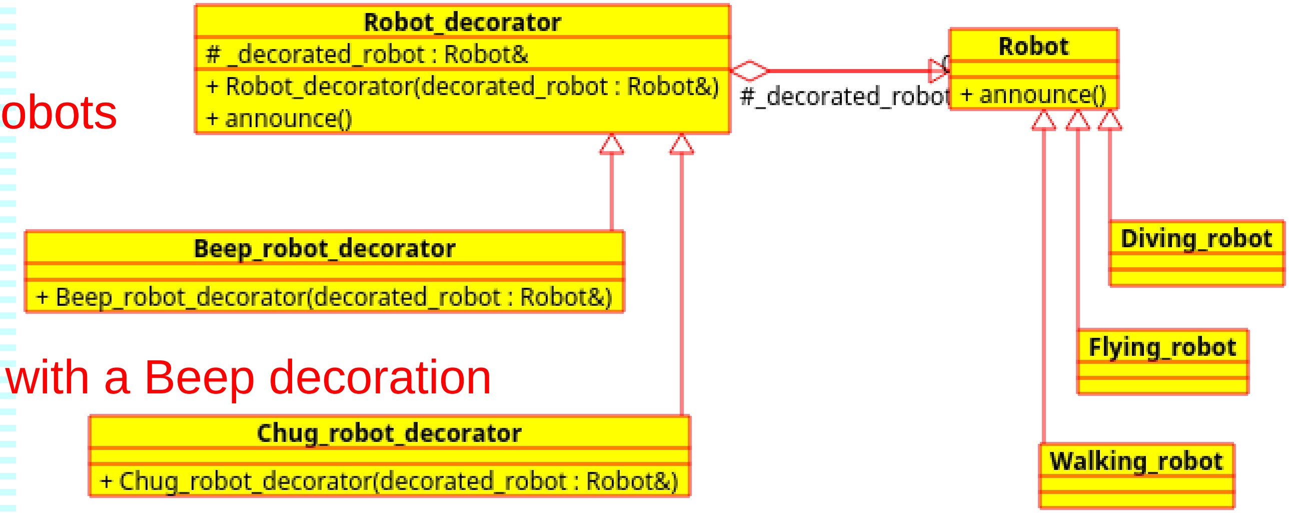
Decoration derived class

```
#include "walking_robot.h"
#include "diving_robot.h"
#include "flying_robot.h"
#include "beep_robot_decorator.h"
#include "chug_robot_decorator.h"

int main() {
    Walking_robot w;
    Diving_robot d;
    Flying_robot f;

    w.announce();
    d.announce();
    f.announce();

    cout << endl;
    Beep_robot_decorator b{w};
    b.announce();

    cout << endl;
    Chug_robot_decorator c{b};
    c.announce();
}
```

Undecorated robots

Instance b as w with a Beep decoration

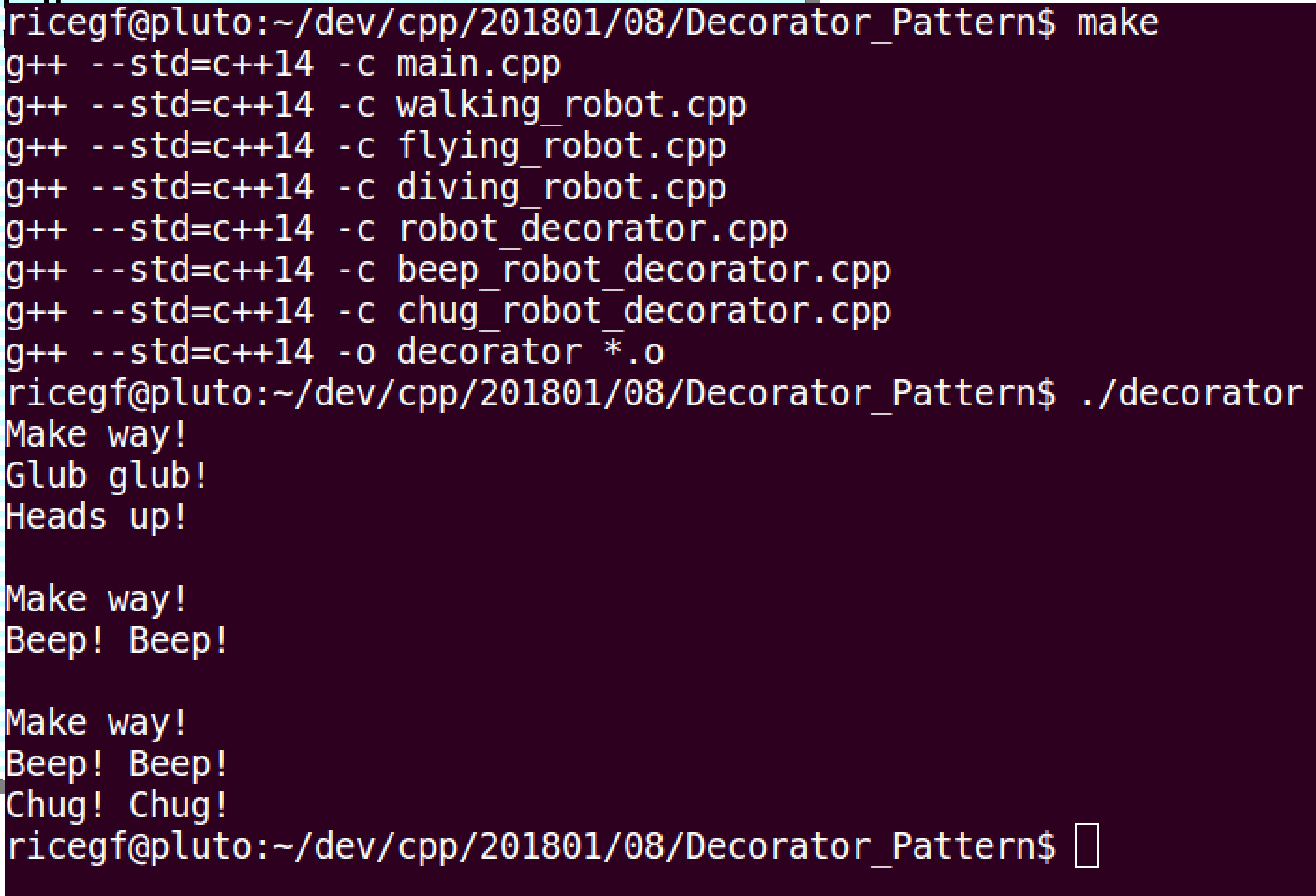
Instance c as b with a Chug decoration - c is now w decorated with Beep and Chug!


Robot_decorator
# _decorated_robot : Robot&
+ Robot_decorator(decorated_robot : Robot&)
+ announce()
#_decorated_robot
Robot
+ announce()
Beep_robot_decorator
+ Beep_robot_decorator(decorated_robot : Robot&)
Chug_robot_decorator
+ Chug_robot_decorator(decorated_robot : Robot&)
Diving_robot
Flying_robot
Walking_robot

Using the Decorated Classes

Decoration derived class

```
#include "walking_robot.h"
#include "diving_robot.h"
#include "flying_robot.h"
#include "beep_robot_decorator.h"
#include "chug_robot_decorato

int main() {
    Walking_robot w;
    Diving_robot d;
    Flying_robot f;

    w.announce();
    d.announce();
    f.announce();

    cout << endl;
    Beep_robot_decorator b{w}
    b.announce();

    cout << endl;
    Chug_robot_decorator c{b}
    c.announce();
}
```



```
ricegf@pluto:~/dev/cpp/201801/08/Decorator_Pattern$ make
g++ --std=c++14 -c main.cpp
g++ --std=c++14 -c walking_robot.cpp
g++ --std=c++14 -c flying_robot.cpp
g++ --std=c++14 -c diving_robot.cpp
g++ --std=c++14 -c robot_decorator.cpp
g++ --std=c++14 -c beep_robot_decorator.cpp
g++ --std=c++14 -c chug_robot_decorator.cpp
g++ --std=c++14 -o decorator *.o
ricegf@pluto:~/dev/cpp/201801/08/Decorator_Pattern$ ./decorator
Make way!
Glub glub!
Heads up!

Make way!
Beep! Beep!

Make way!
Beep! Beep!
Chug! Chug!
ricegf@pluto:~/dev/cpp/201801/08/Decorator_Pattern$
```

Decorator Pattern

- The **Decorator** pattern implements *compositional* inheritance rather than *structural* inheritance
  - Capability is added at runtime by instancing, rather than at compile time by overriding and extending
  - Each capability is stand-alone, and can be combined in an ad hoc fashion
  - However, decorations are brittle and do not scale as well as structural inheritance

A Practical Example (in Python)

Flask is a Decorator-based "Web Micro-Framework"

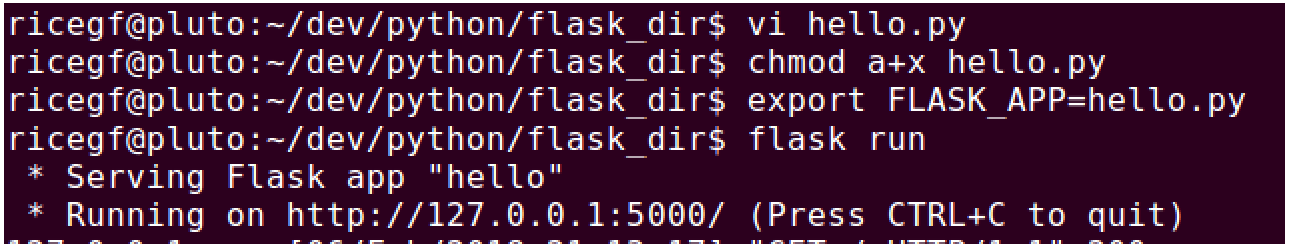
Python

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

Decorator!

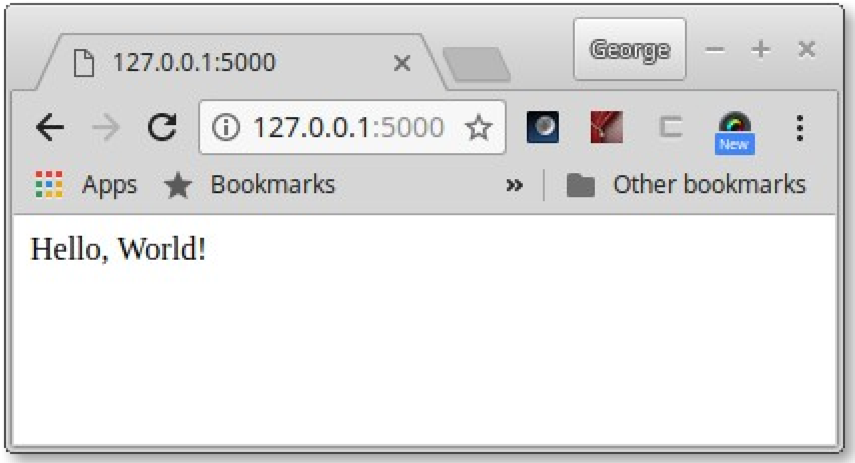
Roughly equivalent C++

```
#include <flask>
Flask app{name};

string hello_world() {
    return "Hello, World!";
}
app.route('/', hello_world);
```



```
ricegf@pluto:~/dev/python/flask_dir$ vi hello.py
ricegf@pluto:~/dev/python/flask_dir$ chmod a+x hello.py
ricegf@pluto:~/dev/python/flask_dir$ export FLASK_APP=hello.py
ricegf@pluto:~/dev/python/flask_dir$ flask run
 * Serving Flask app "hello"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Create the above program
Make it executable
Set an environment variable
Run flask



Hello, World!

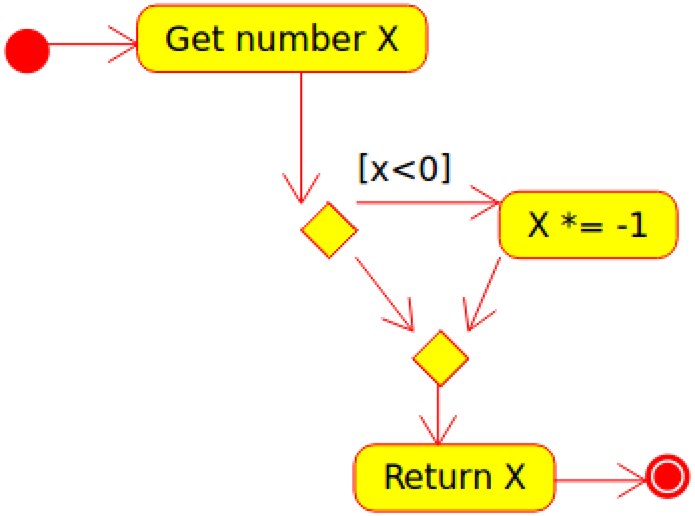
Instant web app!

UML Activity Diagram

- The UML Activity diagram displays a sequence of activities at the algorithm level
  - Similar to a classic “flow chart” or “data flow diagram”
- Represents decisions as well as concurrency
  - Supports decision points – take only one path
  - Supports forks and joins – take all paths, and later sync back up
- Supports hierarchies
  - An activity can contain another Activity Diagram

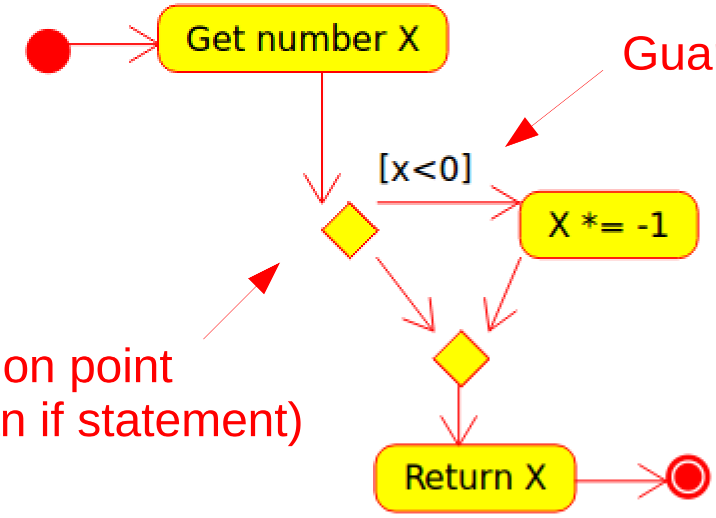
Trivial Example Activity Diagram

Absolute Value



http://www.uml-diagrams.org/activity-diagrams.html

Trivial Example Activity Diagram



Activity to perform

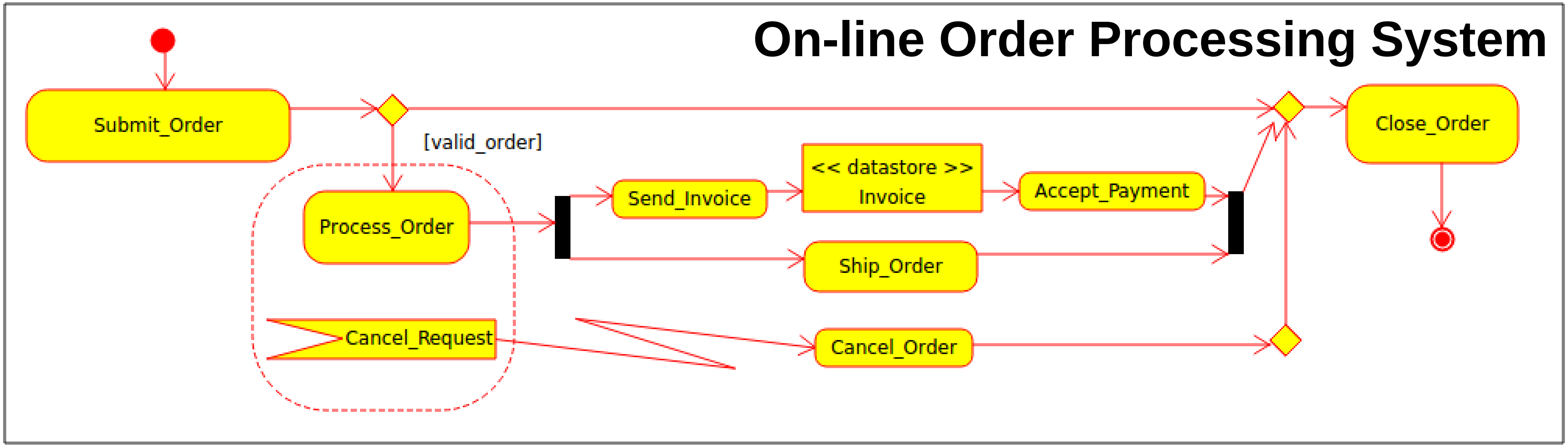
Always start here!
(Must be exactly 1)

Guard (conditional)

Decision point
(like an if statement)

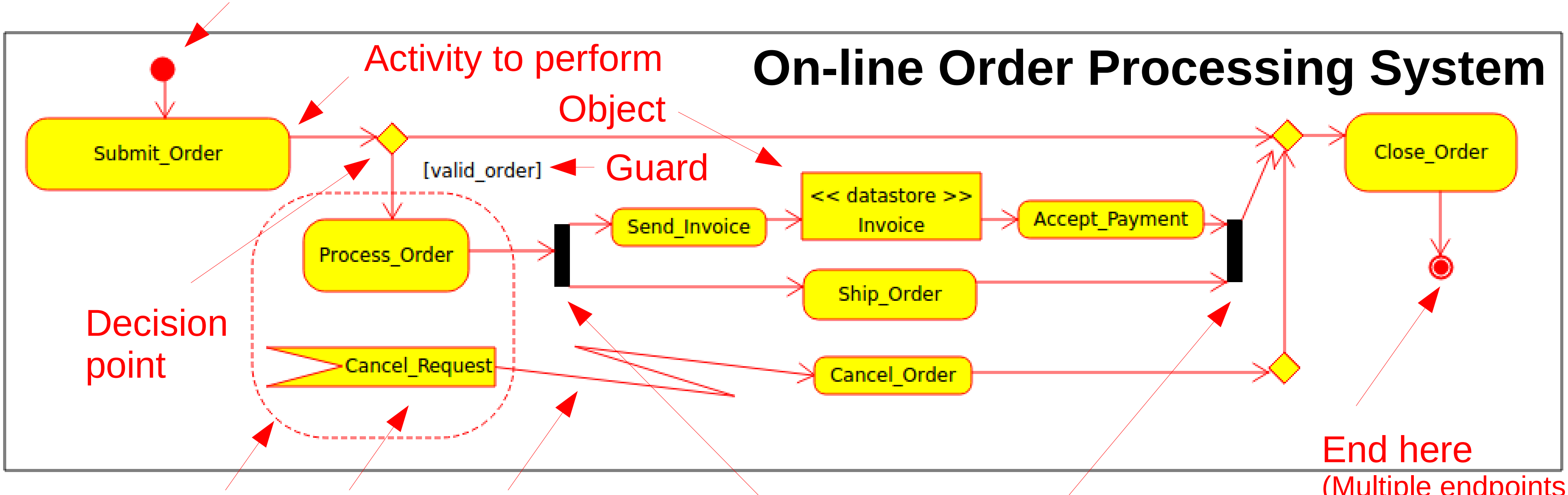
End here
(Multiple endpoints
are permitted)

Example Activity Diagram



Example Activity Diagram

Always start here!
(Must be exactly 1)



Activity to perform

Object

Guard

Decision point

Region

Signal

Interrupt

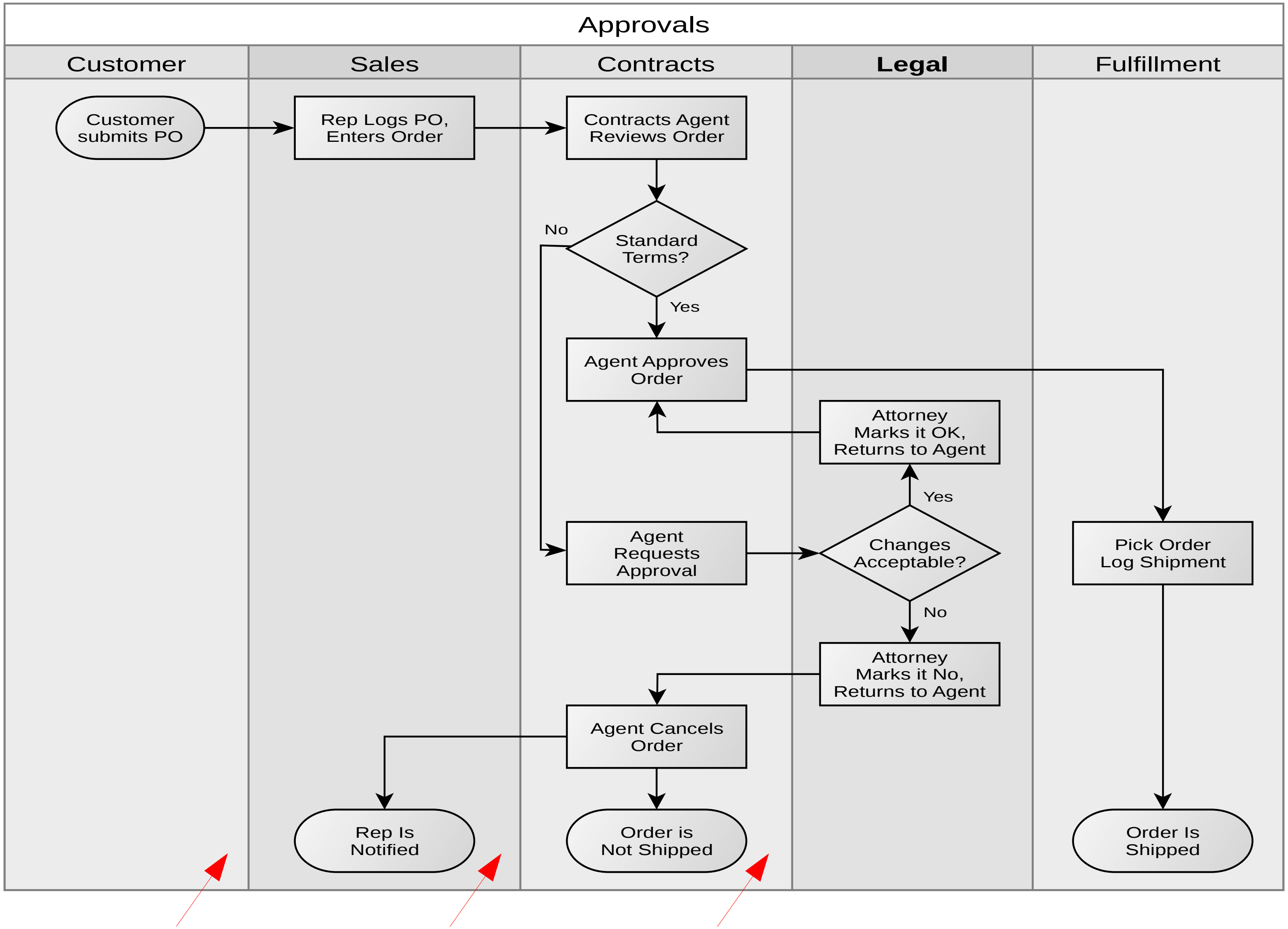
Split / join paths
(multiple threads)

End here
(Multiple endpoints are permitted)

http://www.uml-diagrams.org/activity-diagrams.html

Swimlanes

(Sometimes Called Partitions)

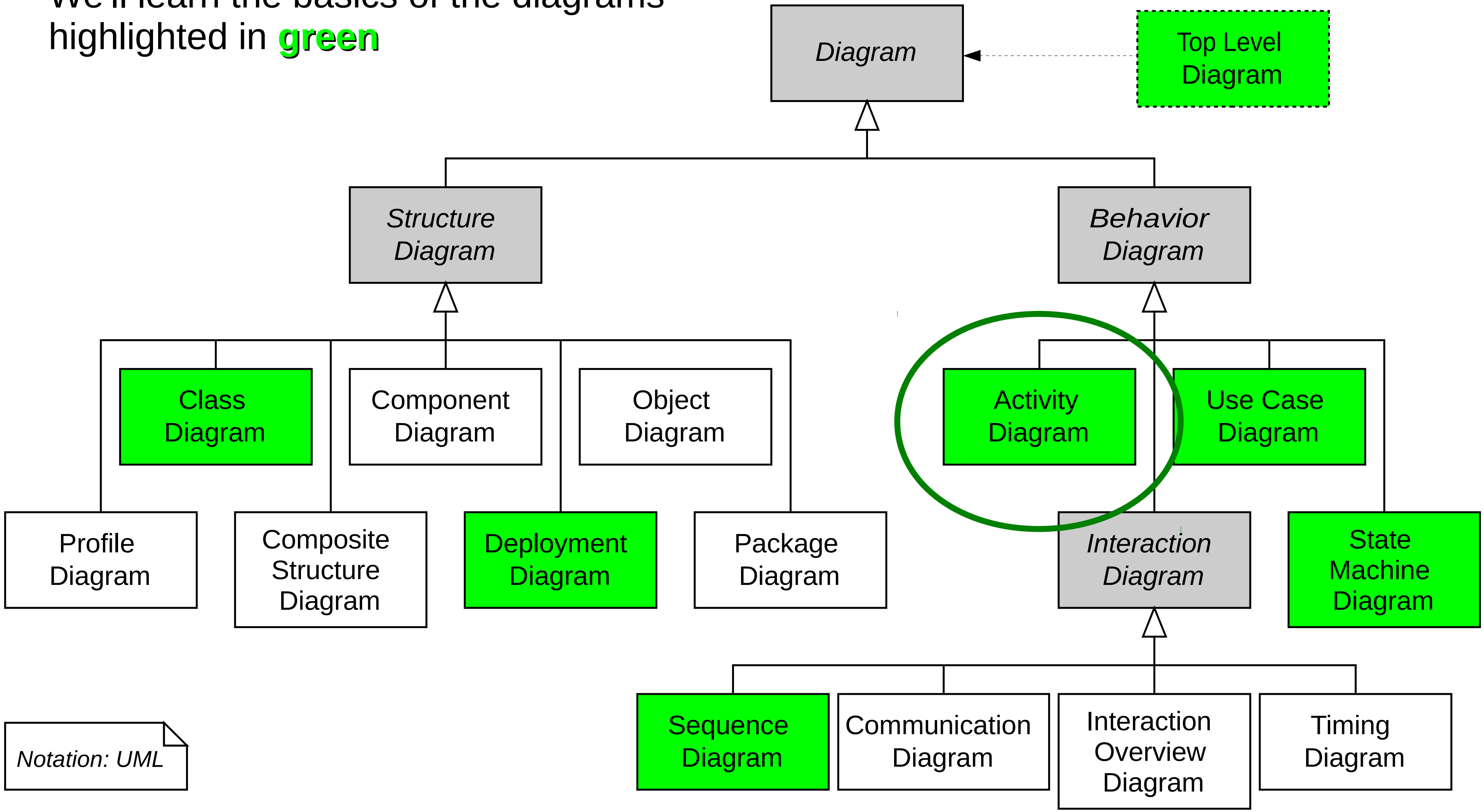


Swimlanes assign responsibility for activities

By Paul Kerr - http://commons.wikimedia.org/wiki/File:Approvals.jpg, CC0, https://commons.wikimedia.org/w/index.php?curid=21550293

The Activity Diagram in Context

We'll learn the basics of the diagrams highlighted in **green**



Original source: Wikipedia, Public Domain SVG

What We Learned Today

- Formatting streams
  - Sticky operators hex, dec, oct, showbase, scientific, fixed, defaultfloat
  - Non-sticky operator setw()
- File
  - Why text formats are usually superior to binary formats
  - Open modes: os_base::app, ate, binary, in, out, trunc
- Random access / access positioning
  - seekg(), seekp()
- String streams
- Decorator pattern
- UML Activity Diagram

Quick Review

- True or False: Most (but not all) stream operators are “sticky”. If False, which is it? If True, give an example of each.
- True or False: Stream operators are constants and thus cannot accept parameters.
- To output hexadecimal numbers via cout, include the ___ operator in the stream. To precede hexadecimal numbers with “0x”, include ________ in the stream.
- What happens if a value exceeds the specified stream output width?
- True or False: Binary file operations are inherently less portable than text file operations.
- Stream operations can target string variables by using the __________ class.
- The __________ pattern dynamically adds more functionality to an object without modifying its structure.
- The UML Activity diagram displays a sequence of activities at the ________ level, particularly useful for documenting Use Cases.
- Which of the following are supported by the UML Activity Diagram?
  (a) Friends (b) Interrupts (c) Swimlanes (d) Hierarchical Diagrams (e) Inheritance

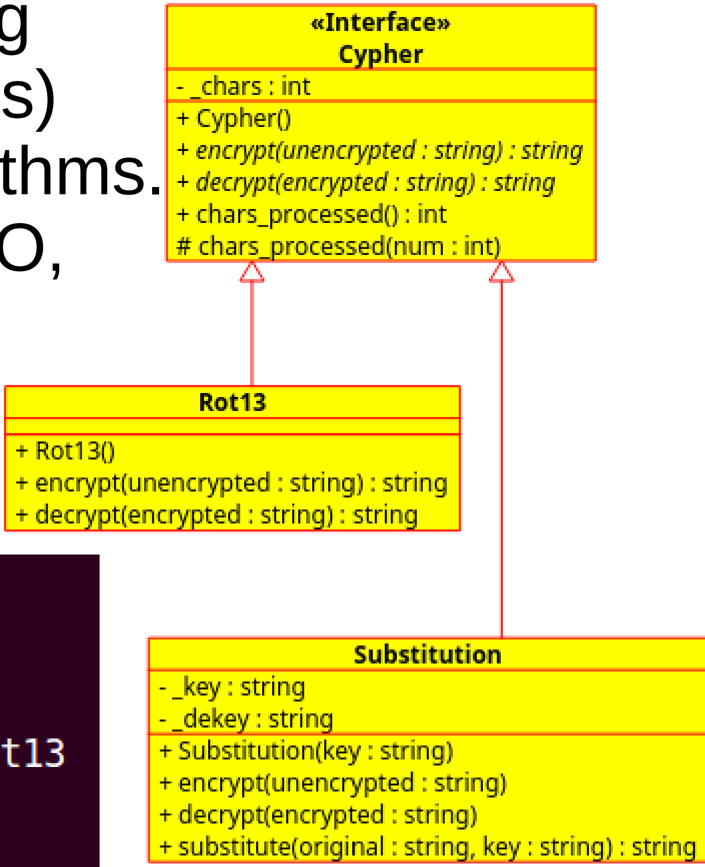
Next Week

- (Optional) Review chapter 11 in Stroustrop
  - Do the drills!
- Read chapters 6 and 7 for next week
  - We will NOT discuss the functional decomposition example in the book – though it's a good example of C++ non-OOP code
  - Instead, we will develop a fully object-oriented application including requirements analysis, design, implementation and test

Homework #4
Cypher

This assignment involves writing a cryptography tool implementing multiple algorithms. We'll utilize a pure virtual class (the interface), and from it derive several classes to encrypt or decrypt a user-specified file using the Rot13, Substitution, and (at bonus levels) Exclusive-OR and asynchronous key algorithms. In doing so, we'll practice inheritance, file I/O, and static class members, too!

Due Thursday, February 15 at 8 am.



«Interface»
Cypher

- _chars : int

+ Cypher()
+ *encrypt(unencrypted : string) : string*
+ *decrypt(encrypted : string) : string*
+ chars_processed() : int
# chars_processed(num : int)

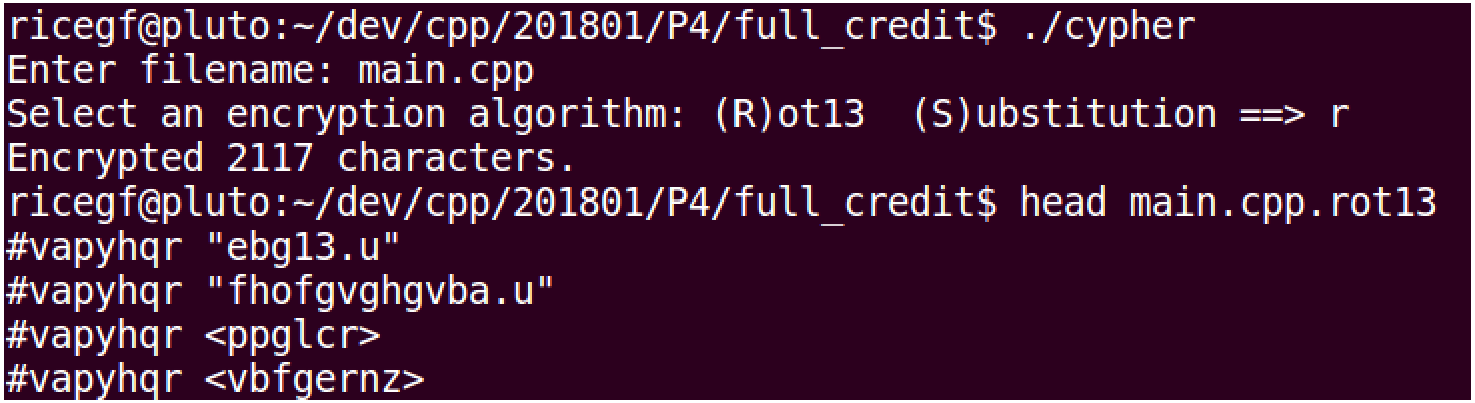
Rot13

+ Rot13()
+ encrypt(unencrypted : string) : string
+ decrypt(encrypted : string) : string

Substitution

- _key : string
- _dekey : string

+ Substitution(key : string)
+ encrypt(unencrypted : string)
+ decrypt(encrypted : string)
+ substitute(original : string, key : string) : string



```
ricegf@pluto:~/dev/cpp/201801/P4/full_credit$ ./cypher
Enter filename: main.cpp
Select an encryption algorithm: (R)ot13  (S)ubstitution ==> r
Encrypted 2117 characters.
ricegf@pluto:~/dev/cpp/201801/P4/full_credit$ head main.cpp.rot13
#vapyhqr "ebg13.u"
#vapyhqr "fhofgvghgvba.u"
#vapyhqr <ppglcr>
#vapyhqr <vbfgernz>
```