Imperial College London

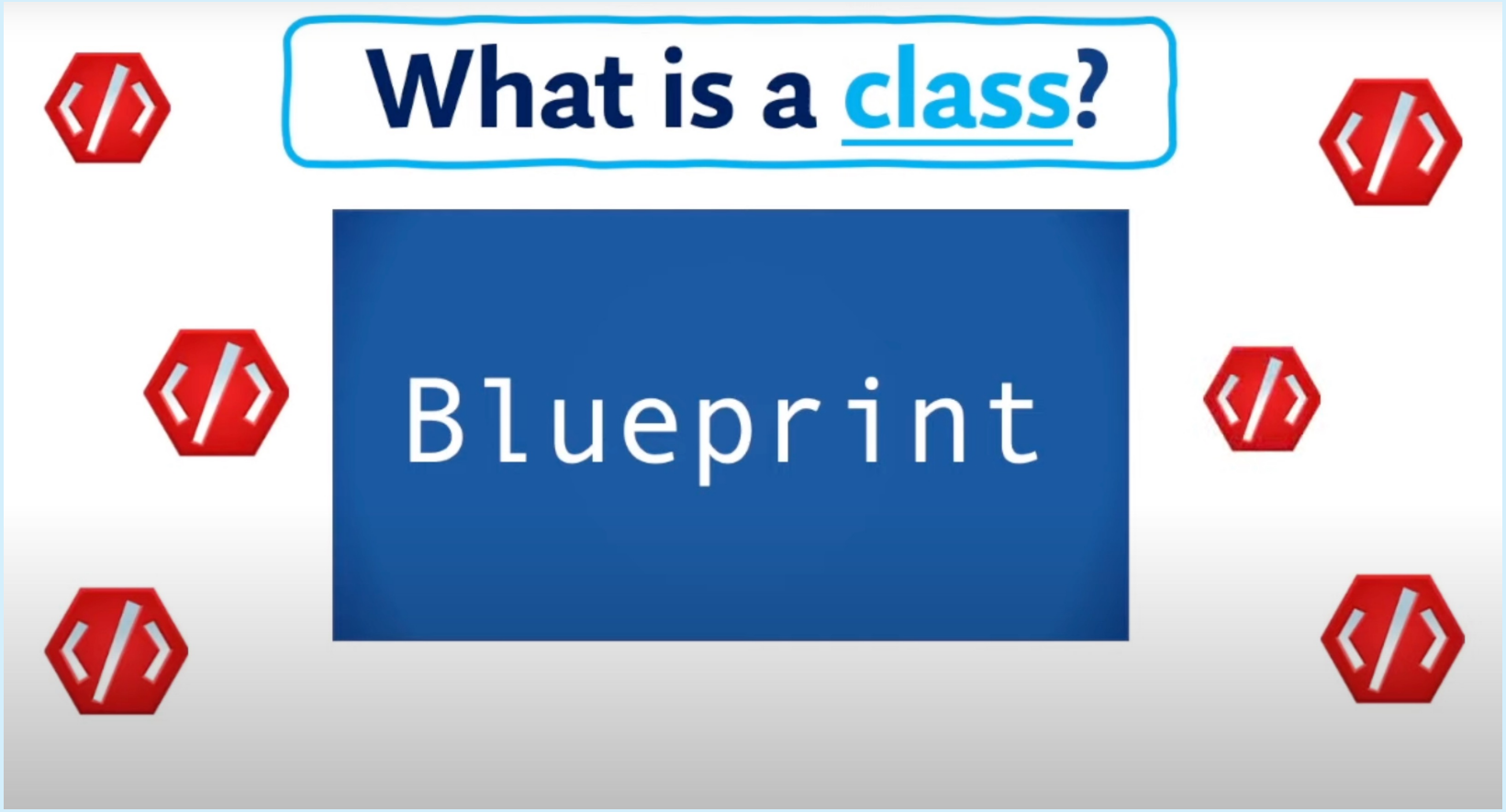
Advanced Programming

Classes, Structs, Enumerations

Adriana Paluszny

[Many of the slides are adapted from the original Bjarne Stroustrup slides on C++]

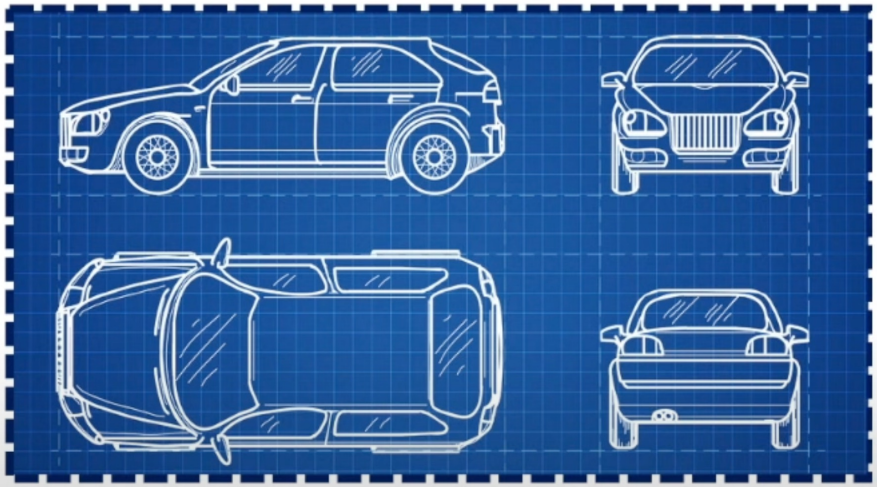
Imperial College London



What is Object-Oriented Programming? | Coding for Kids | Kodabl

Kodable
12.5K subscribers
Subscribe
915

Class = blueprint



body

wheels

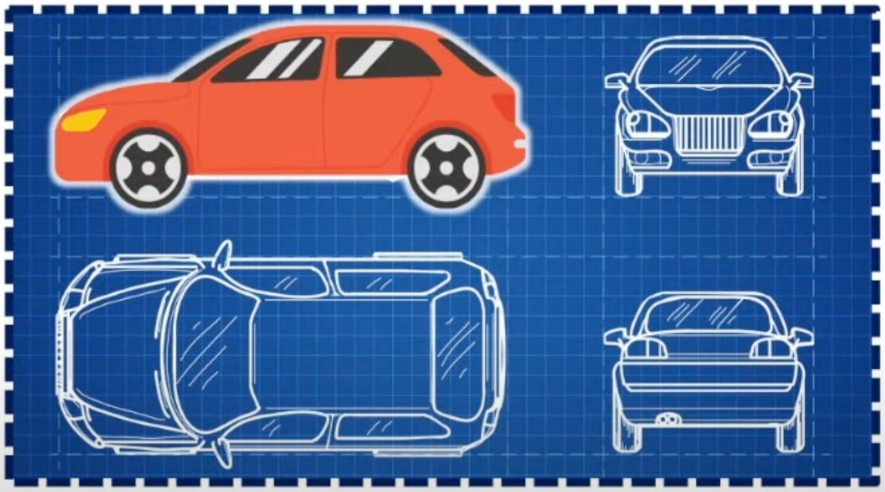
engine

drive( )

brake( )

turn( )

Imperial College
London



Imperial College
London

Classes

- The idea:
  - A class directly represents a concept in a program
    - If you can think of “it” as a separate entity, it is plausible that it could be a class or an object of a class
    - Examples: vector, matrix, input stream, string, FFT, valve controller, robot arm, device driver, picture on screen, dialog box, graph, window, temperature reading, clock
  - A class is a (user-defined) type that specifies how objects of its type can be created and used
  - In C++ (as in most modern languages), a class is the key building block for large programs
    - And very useful for small ones also
  - The concept was originally introduced in Simula67

Imperial College London

Types of classes

- Regular class: This is the most common type of class in C++. It is used to define objects that encapsulate data and operations that can be performed on that data.
- Abstract class: This is a class that cannot be instantiated, and is used as a base class for other classes. It contains pure virtual functions that must be implemented by the derived classes.
- Concrete class: This is a class that can be instantiated, and is not meant to be used as a base class for other classes.
- Friend class: This is a class that is granted access to the private and protected members of another class.
- Static class: This is a class that contains only static members, and cannot be instantiated. Its members are accessed using the class name, rather than through an object.
- Template class: This is a class that is defined using a template, which allows it to be instantiated with different data types.
- Singleton class: This is a class that can only have one instance throughout the entire program.
- Derived class: This is a class that inherits properties and methods from a base class, and can add its own properties and methods.

Imperial College London

Members and member access

- One way of looking at a class;

```
class X { // this class' name is X
   // data members (they store information)
   // function members (they do things, using the information)
};
```

- Example

```
class X {
public:
   int m;    // data member
   int mf(int v) { int old = m; m=v; return old; }   // function member
};

X var;              // var is a variable of type X
var.m = 7;          // access var's data member m
int x = var.mf(9);  // call var's member function mf()
```

Classes

- A class is a user-defined type

```
class X { // this class' name is X
public:   // public members -- that's the interface to users
          //   (accessible by all)
   // functions
   // types
   // data (often best kept private)
private:  // private members -- that's the implementation details
          //        (accessible by members of this class only)
   // functions
   // types
   // data
};
```

Struct and class

- Class members are private by default:

```
class X {
     int mf();
     // …
};
```

- Means

```
class X {
private:
     int mf();
     // …
};
```

- So

```
X x;                // variable x of type X
int y = x.mf();     // error: mf is private (i.e., inaccessible)
```

Imperial College London

Struct and class

- A struct is a class where members are public by default:

```
struct X {
     int m;
     // …
};
```

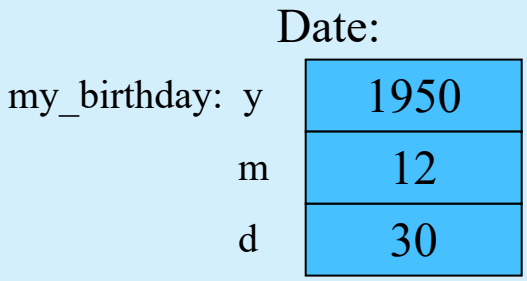
- Means

```
class X {
public:
     int m;
     // …
};
```

- **struct**s are primarily used for data structures where the members can take any value

Imperial College London

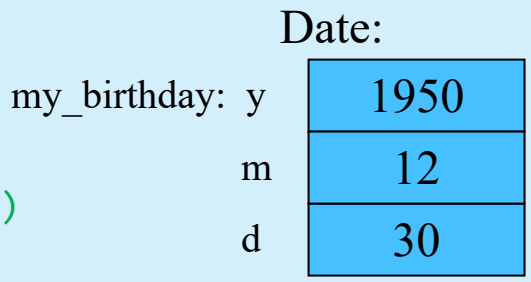
Structs




```
// simplest Date (just data)
struct Date {
    int y,m,d; // year, month, day
};

Date my_birthday;   // a Date variable (object)

my_birthday.y = 12;
my_birthday.m = 30;
my_birthday.d = 1950;    // oops! (no day 1950 in month 30)
                         // later in the program, we'll have a problem
```

Imperial College London

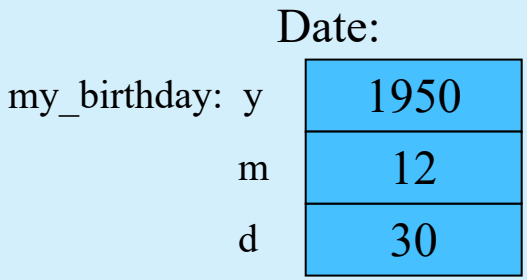
Structs




```
// simple Date (with a few helper functions for convenience)
struct Date {
    int y,m,d; // year, month, day
};

Date my_birthday;   // a Date variable (object)


// helper functions:

void init_day(Date& dd, int y, int m, int d); // check for valid date and initialize
                            // Note: this y, m, and d are local

void add_day(Date& dd, int n);    // increase the Date by n days
// …

init_day(my_birthday, 12, 30, 1950); // run time error: no day 1950 in month 30
```

Imperial College London

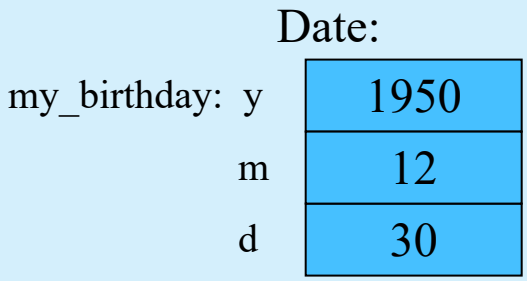
Structs




```
// simple Date
//   guarantee initialization with constructor
//   provide some notational convenience
struct Date {
    int y,m,d;           // year, month, day
    Date(int y, int m, int d);  // constructor: check for valid date and initialize
    void add_day(int n);      // increase the Date by n days
};

// …
Date my_birthday;        // error: my_birthday not initialized
Date my_birthday {12, 30, 1950};  // oops! Runtime error
Date my_day {1950, 12, 30};  // ok
my_day.add_day(2);       // January 1, 1951
my_day.m = 14;           // ouch! (now my_day is a bad date)
```

Imperial College London

Classes



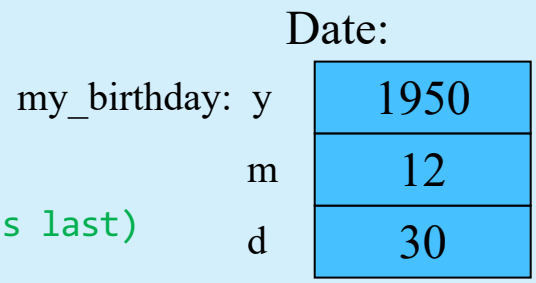

```
// simple Date (control access)
class Date {
    int y,m,d; // year, month, day
public:
    Date(int y, int m, int d);  // constructor: check for valid date and initialize

    // access functions:
    void add_day(int n);     // increase the Date by n days
    int month() { return m; }
    int day() { return d; }
    int year() { return y; }
};

// …
Date my_birthday {1950, 12, 30};        // ok
cout << my_birthday.month() << endl;    // we can read
my_birthday.m = 14;           // error: Date::m is private
```

Classes

- The notion of a “valid Date” is an important special case of the idea of a valid value
- We try to design our types so that values are guaranteed to be valid
  - Or we have to check for validity all the time
- A rule for what constitutes a valid value is called an “invariant”
  - The invariant for Date (“a Date must represent a date in the past, present, or future”) is unusually hard to state precisely
    - Remember February 28, leap years, etc.
- If we can’t think of a good invariant, we are probably dealing with plain data
  - If so, use a struct
  - Try hard to think of good invariants for your classes
    - that saves you from poor buggy code

Imperial College London

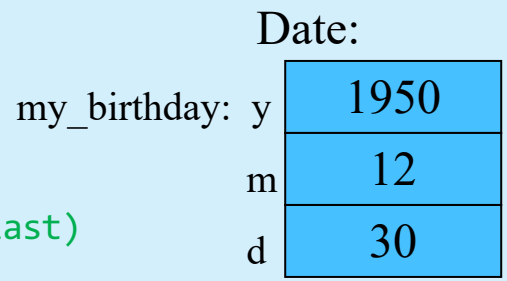
Classes




```
// simple Date (some people prefer implementation details last)
class Date {
public:
    Date(int yy, int mm, int dd);    // constructor: check for valid date and
                                     // initialize

    void add_day(int n);             // increase the Date by n days
    int month();
    // …
private:
    int y, m, d;    // year, month, day
};

Date::Date(int yy, int mm, int dd)     // definition; note :: "member of"
    :y{yy}, m{mm}, d{dd} { /* … */ };  // note: member initializers

void Date::add_day(int n) { /* … */ }; // definition
```

Imperial College London

Classes




```
// simple Date (some people prefer implementation details last)
class Date {
public:
    Date(int yy, int mm, int dd);  // constructor: check for valid date and initialize
    void add_day(int n);           // increase the Date by n days
    int month();
    // …
private:
    int y, m, d;    // year, month, day
};

int month() { return m; }     // error: forgot Date::
                              // this month() will be seen as a global function
                              // not the member function, so can't access members

int Date::season() { /* … */ }    // error: no member called season
```

Classes

```
// simple Date (what can we do in case of an invalid date?)
class Date {
public:
    class Invalid { };                  // to be used as exception
    Date(int y, int m, int d);          // check for valid date and initialize
    // …
private:
    int y, m, d;                        // year, month, day
    bool is_valid(int y, int m, int d); // is (y,m,d) a valid date?
};

Date::Date(int yy, int mm, int dd)
    : y{yy}, m{mm}, d{dd}               // initialize data members
{
    if (!is_valid (y,m,d)) throw Invalid(); // check for validity
}
```

Imperial College London

Classes

- Why bother with the public/private distinction?
- Why not make everything public?
  - To provide a clean interface
    - Data and messy functions can be made private
  - To maintain an invariant
    - Only a fixed set of functions can access the data
  - To ease debugging
    - Only a fixed set of functions can access the data
    - (known as the "round up the usual suspects" technique)
  - To allow a change of representation
    - You need only to change a fixed set of functions
    - You don't really know who is using a public member

Imperial College London

Imperial College London

EXERCISE

1. Create and compile the class Date. Implement the function `void add_day(int d).`

2. Create a class called Point with member variables x and y, and member functions setX(), setY(), getX(), getY(), and distance(), which calculates the distance between two points.

3. Create a class called Rectangle with member variables length and width, and member functions setLength(), setWidth(), getLength(), getWidth(), and area(), which calculates the area of the rectangle.

Imperial College London

Enumerations

- An **enum** (enumeration) is a simple user-defined type, specifying its set of values (its enumerators)
- For example:

```
enum class Month {
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

Month m = feb;
m = 7;                  // error: can't assign int to Month
int n = m;              // error: we can't get the numeric value of a Month
Month mm = Month(7);    // convert int to Month (unchecked)
```

Imperial College London

“Plain” Enumerations

- Simple list of constants:

```
enum { red, green };          // a “plain” enum { } doesn’t define a scope
int a = red;                  // red is available here
enum { red, blue, purple };   // error: red defined twice
```

- Type with a list of named constants

```
enum Color { red, green, blue, /* … */  };
enum Month { jan, feb, mar, /* … */ };

Month m1 = jan;
Month m2 = red;     // error: red isn’t a Month
Month m3 = 7;       // error: 7 isn’t a Month
int i = m1;         // ok: an enumerator is converted to its value, i==0
```

Imperial College London

Class Enumerations

- Type with a list of typed named constants

```
enum class Color { red, green, blue, /* … */  };
enum class Month { jan, feb, mar, /* … */ };
enum class Traffic_light { green, yellow, red }; // OK: scoped enumerators

Month m1 = jan;          // error: jan not in scope
Month m1 = Month::jan;   // OK
Month m2 = Month::red;   // error: red isn't a Month
Month m3 = 7;            // error: 7 isn't a Month
Color c1 = Color::red;   // OK
Color c2 = Traffic_light::red;    // error
int i = m1;              // error: an enumerator is not converted to int
```

Imperial College London

Enumerations – Values

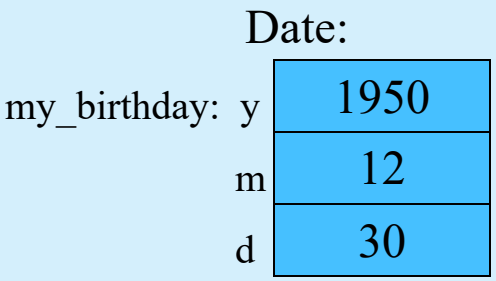
- By default

```
// the first enumerator  has the value  0,
// the next enumerator has the value “one plus the value of the
// enumerator before it”
enum { horse, pig, chicken };      // horse==0, pig==1, chicken==2
```

- You can control numbering

```
enum { jan=1, feb, march /* … */ };     // feb==2, march==3
enum stream_state { good=1, fail=2, bad=4, eof=8 };
int flags = fail+eof;                   // flags==10
stream_state s = flags; // error: can’t assign an int to a stream_state
stream_state s2 = stream_state(flags); // explicit conversion (be careful!)
```

Classes




```
// simple Date (use enum class Month)
class Date {
public:
     Date(int y, Month m, int d);  // check for valid date and initialize
    // …
private:
    int y;      // year
    Month m;
    int d;      // day
};

Date my_birthday(1950, 30, Month::dec);  // error: 2nd argument not a Month
Date my_birthday(1950, Month::dec, 30);  // OK
```

Imperial College London

Const

```
class Date {
public:
    // …
    int day() const { return d; } // const member: can’t modify
    void add_day(int n);      // non-const member: can modify
    // …
};

Date d {2000, Month::jan, 20};
const Date cd {2001, Month::feb, 21};

cout << d.day() << " – " << cd.day() << endl;    // ok
d.add_day(1);  // ok
cd.add_day(1); // error: cd is a const
```

Imperial College London

Const

```
Date d {2004, Month::jan, 7};           // a variable
const Date d2 {2004, Month::feb, 28};   // a constant
d2 = d;                                 // error: d2 is const
d2.add(1);                              // error d2 is const
d = d2;                                 // fine
d.add(1);                               // fine

d2.f();   // should work if and only if f() doesn't modify d2
          // how do we achieve that? (say that's what we want, of course)
```

Imperial College London

Const member functions

```
// Distinguish between functions that can modify (mutate) objects
// and those that cannot (“const member functions”)
class Date {
public:
    // …
    int day() const;     // get (a copy of) the day
    // …
    void add_day(int n); // move the date n days forward
    // …
};

const Date dx {2008, Month::nov, 4};
int d = dx.day();   // fine
dx.add_day(4); // error: can’t modify constant (immutable) date
```

Imperial College London

Classes

- What makes a good interface?
  - Minimal
    - As small as possible
  - Complete
    - And no smaller
  - Type safe
    - Beware of confusing argument orders
    - Beware of over-general types (e.g., int to represent a month)
  - Const correct
    - Everything that can be const should be const

Classes

- Essential operations
  - Default constructor (defaults to: nothing)
    - No default if any other constructor is declared
  - Copy constructor (defaults to: copy the member)
  - Assignment operator (defaults to: copy the members)
  - Destructor (defaults to: nothing)
- For example

```
Date d;   // error: no default constructor
Date d2(d);    // ok: copy constructor (copy the elements)
d = d2;   // ok: assignment operator (copy the elements)
```

Imperial College London

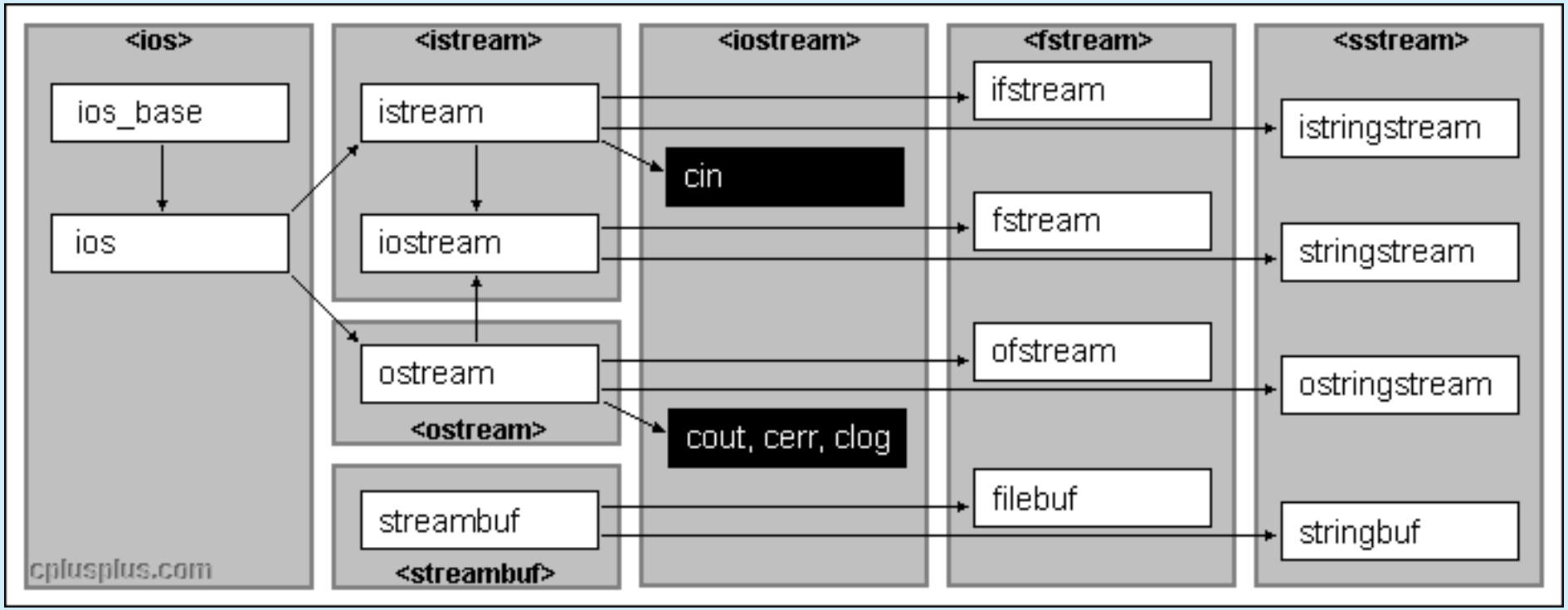
Advanced Programming

Streams and Data Files

Adriana Paluszny

Imperial College
London

Overview of standard library streams



Streams and Data Files

- We have used two streams already, namely `cin` and `cout`
- We can create our own streams to input and output data
- Streams are variables (strictly speaking they are objects, though more on that later in the course)
  - For files the variable type is `fstream` (need to include the header `<fstream>`)
- Stream variables are declared like any other. E.g. To create a stream called `myFile`:

```
fstream myFile;
```

Opening Data Files

- Simply creating a stream variable is not enough, we need to associate it with a file, which must either be created or opened
- This is done using `open`:
- e.g. `myFile.open("test.txt",fstream::out);`
  - As stated before, `fstream` is not a simple variable, but actually a class of which `open` is a member function (or method), but much more on this later when we look at classes in C++ in detail
- The first parameter in `open` is the name of the file to be opened
- The second parameter is a set of flags saying how the file is to be opened
  - `fstream::out` indicates that the file is being opened for writing. This also means that the file will be created or cleared if it already exists.

Opening Files

- There are a number of flags that can be set when opening a file stream:
- `fstream::in`: Open file as an input stream
- `fstream::out:` Open file as an out stream
- `fstream::binary:` Open the file for binary output (it is set to text output by default). **More on this later**.
- `fstream::app:` Append the output to the end of the file
- `fstream::trunc:` Clear the file on opening (this is done by default if `out` is set, but not if `in,` or `in` and `out` are set)

- These flags can be combined using the `|` symbol (`|` is a bitwise or). E.g. To open a file for reading and writing and to clear it before use:
- `myFile.open("Inout.txt", fstream::in| fstream::out| fstream::trunc);`

Checking and Closing Files

- After `open` has been called it is good practice to check if the file was opened successfully
  - A file might not open if, for instance, it is already open in another program or if you are trying to open for reading a file that does not exist
- After `open` has been called, the member function `fail` will return true if the file couldn't be opened
- After a file stream is finished with it should be closed using the `close` member function. This does 2 things:
  - It makes the file available for other programs to use
  - It flushes any unwritten output from the memory buffer into the file

EXERCISE

- Open a text file in notepad and input the data:
- “1,2,3,100,4,5,6,-1,0.7,16,18,21,17,15,12”
- Save it as “input_data.txt”
- Write some code that reads this file and stores the information in a 2D array
- Create a function that normalizes all values (L1 normalization)
- Write another file “output_data.txt” with the normalized values (same format)