Advanced Programming

Inheritance, Polymorphism

Adriana Paluszny

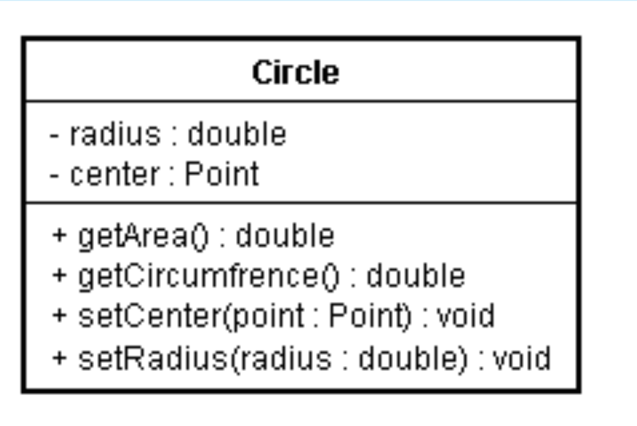
[Many of the slides are adapted from the original Bjarne Stroustrup slides on C++]

Imperial College London

Imperial College London

A simple class hierarchy (UML)

UML Class (single) diagram



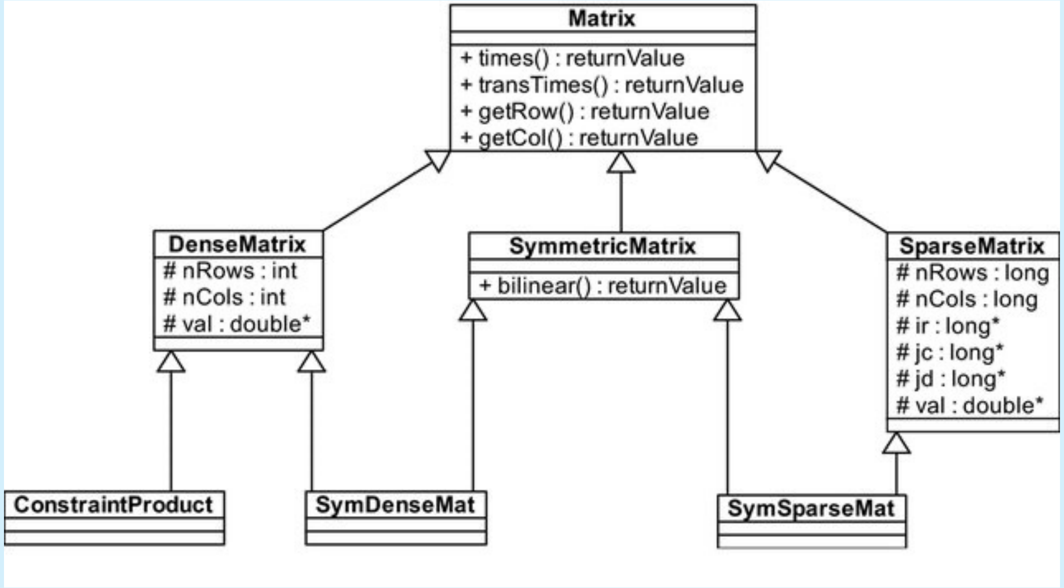

```
class Circle {
private:
double radius;
Point center;
public:
setRadius(double radius);
setCenter(Point center);
double getArea();
double getCircumfrence();
};
```

UML, short for Unified Modeling Language, is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/

Imperial College London

A simple class hierarchy (UML)

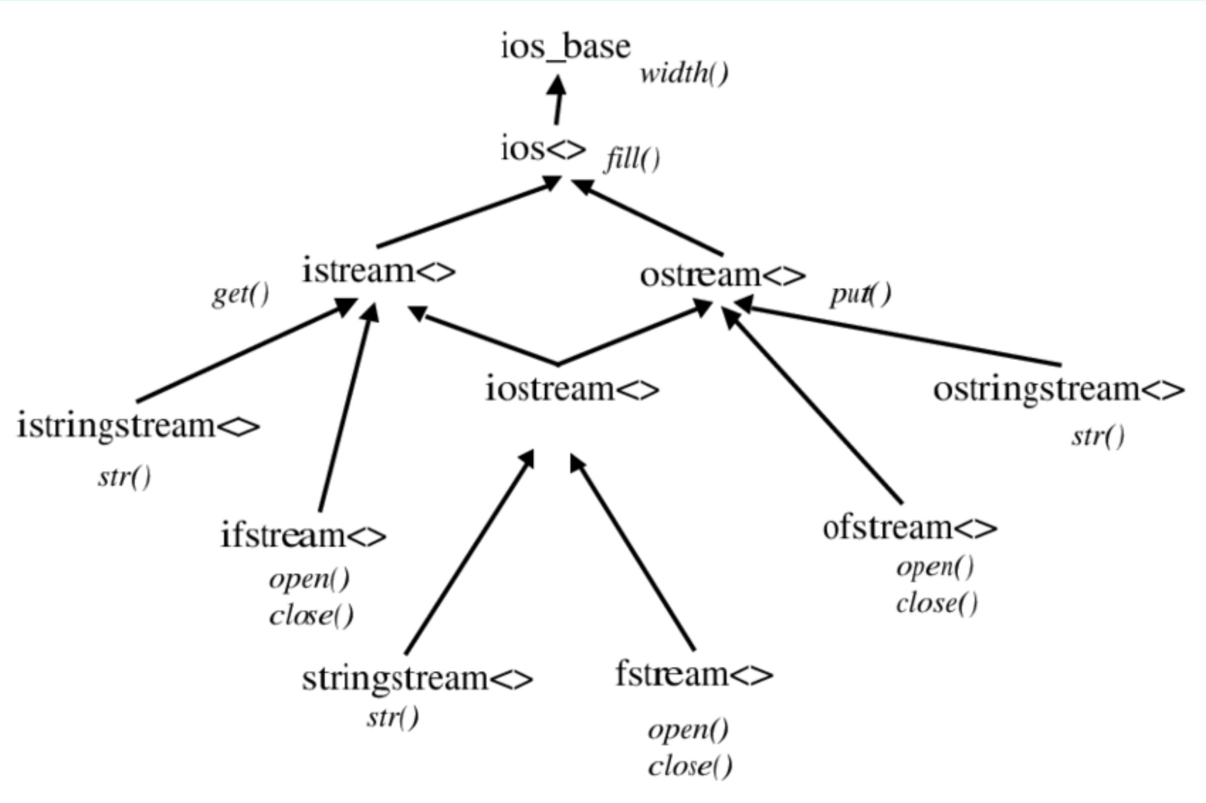
UML Class diagram




```
class Matrix {…};
class DenseMatrix : public Matrix {…};
class SparseMatrix : public Matrix {…};
class SymMatrix : public Matrix {…};
class SymSparseMatrix : public SparseMatrix, public SymMatrix {…};
class SymDenseMat : public DenseMatrix, public SymMatrix {…};
class ConstraintProduct : public DenseMatrix {…};
```

Imperial College London

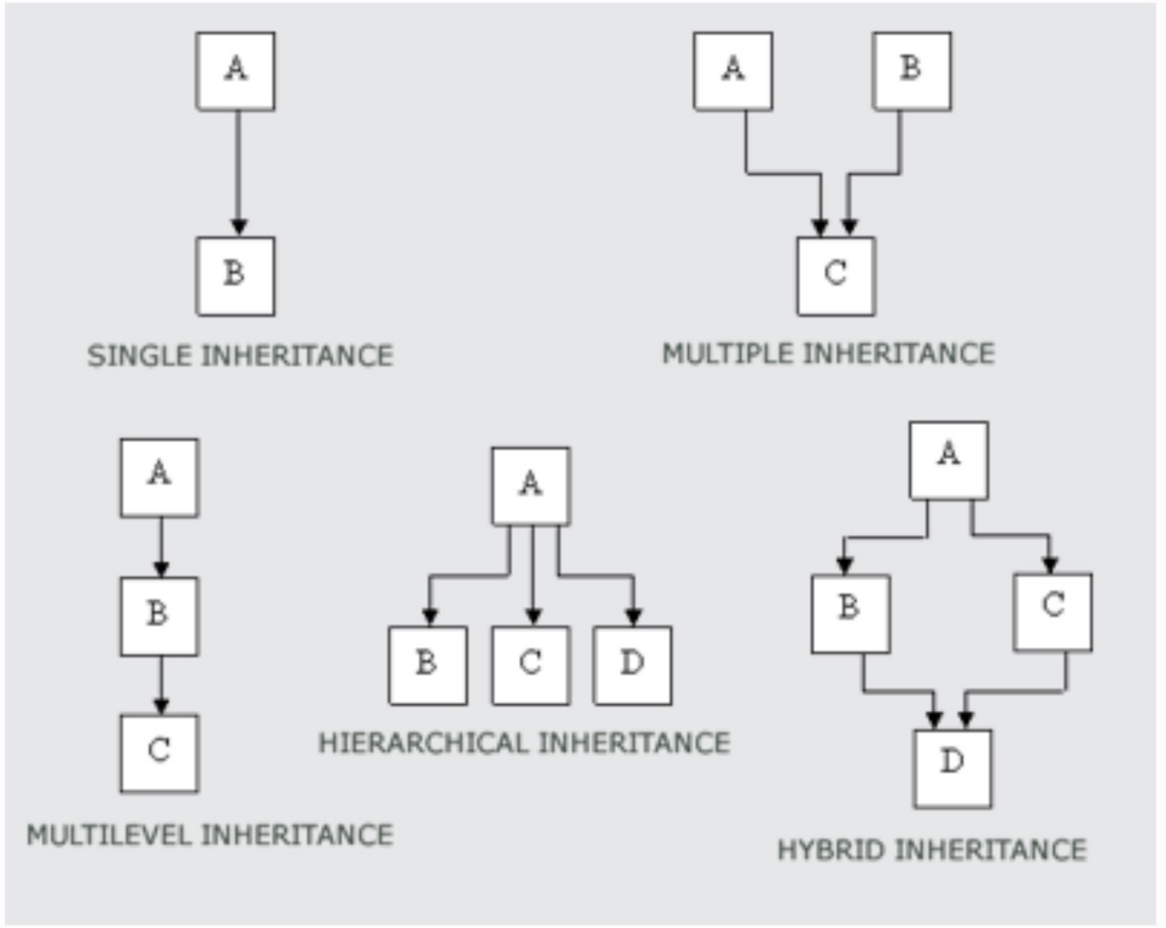
The C++ stream class hierarchy



Types of inheritance

Inheritance can also be:
Private
Protected
Public

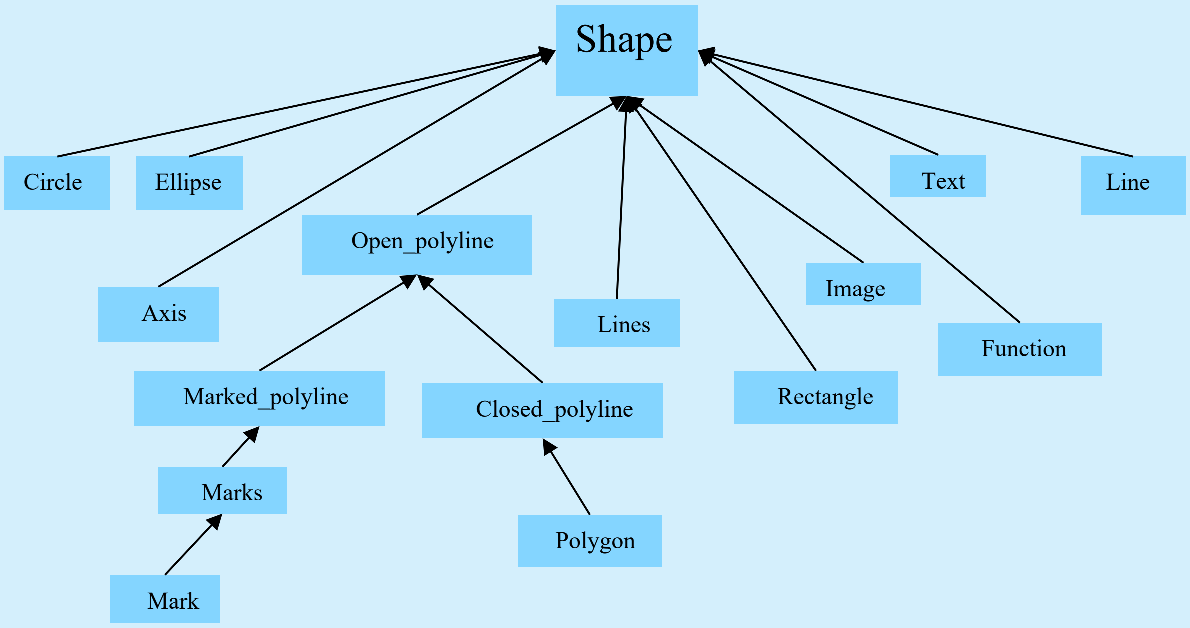
Both members and functions can be inherited



Imperial College London

Consider a Stroustrup example: Class Shape

- All our shapes are “based on” the Shape class
  - E.g., a **Polygon** is a kind of **Shape**



Imperial College London

Class Shape

- **Shape** ties our graphics objects to "the screen"
  - **Window** "knows about" **Shapes**
  - All our graphics objects are kinds of **Shapes**
- **Shape** is the class that deals with color and style
  - It has **Color** and **Line_style** members
- **Shape** can hold **Points**
- **Shape** has a basic notion of how to draw lines
  - It just connects its **Points**

The "Shape" example was developed by Stroustrup, it links C++ and advanced programming to computer graphics, one of the main drivers behind C++.

Class Shape

- Shape deals with color and style
  - It keeps its data private and provides access functions

```
    void set_color(Color col);
    Color color() const;
    void set_style(Line_style sty);
    Line_style style() const;
    // …
private:
    // …
    Color line_color;
    Line_style ls;
```

Imperial College London

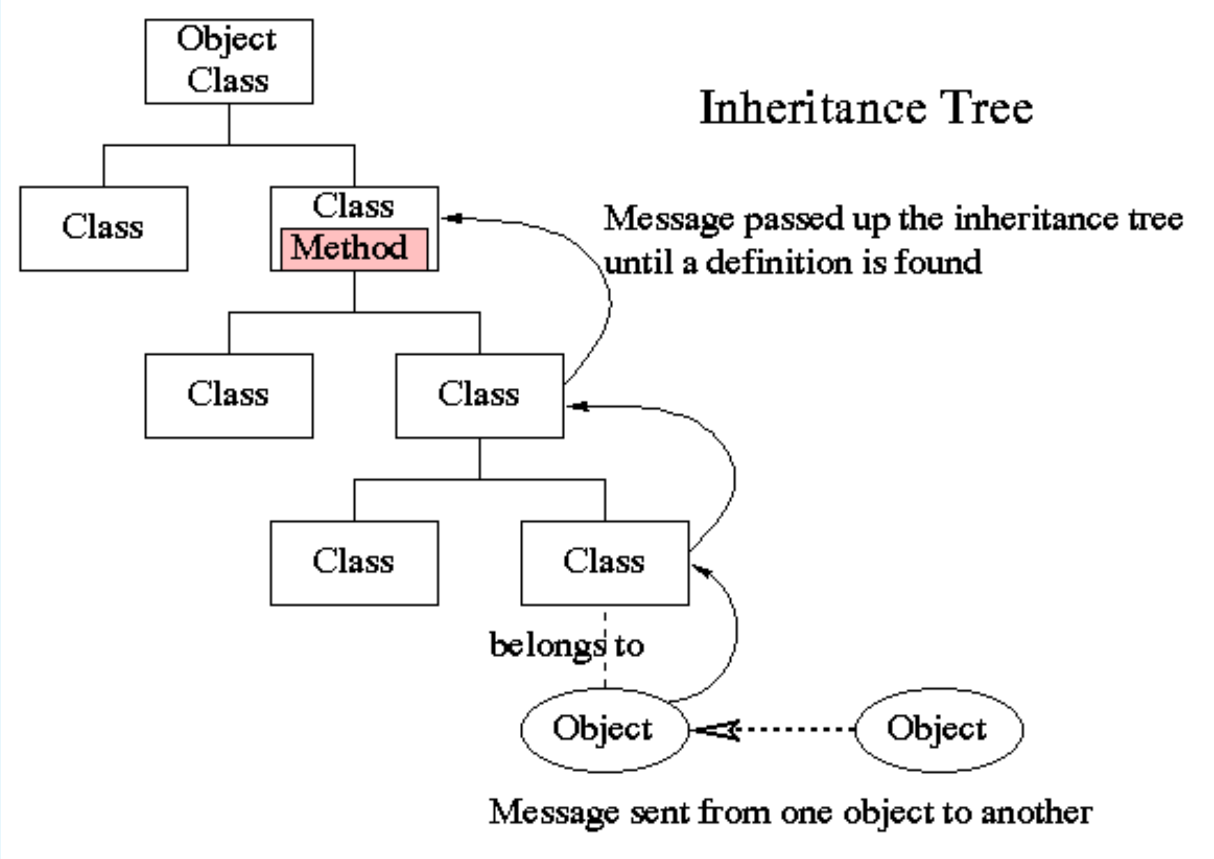
Class Shape

- **Shape** stores **Points**
  - It keeps its data private and provides access functions

```
Point point(int i) const;  // read-only access to points
int number_of_points() const;
// …
protected:
      void add(Point p);         // add p to points
      // …
private:
vector<Point> points; // not used by all shapes
```

Imperial College London

What happens when you use inheritance?



Language mechanisms

- Most popular definition of object-oriented programming:

  OOP == inheritance + polymorphism + encapsulation

- Base and derived classes
  - `struct Circle : Shape { … };`
  - Also called “inheritance”

  inheritance

- Virtual functions
  - `virtual void draw_lines() const;`
  - Also called “run-time polymorphism” or “dynamic dispatch”

  polymorphism

- Private and protected
  - `protected: Shape();`
  - `private: vector<Point> points;`

  encapsulation

Benefits of inheritance

- Interface inheritance
  - A function expecting a shape (a `Shape&`) can accept any object of a class derived from Shape.
  - Simplifies use
    - sometimes dramatically
  - We can add classes derived from Shape to a program without rewriting user code
    - Adding without touching old code is one of the “holy grails” of programming
- Implementation inheritance
  - Simplifies implementation of derived classes
    - Common functionality can be provided in one place
    - Changes can be done in one place and have universal effect
      - Another “holy grail”

Class Shape

- **Shape** itself can access points directly:

```
void Shape::draw_lines() const // draw connecting lines
{
   if (color().visible() && 1<points.size())
     for (int i=1; i<points.size(); ++i)
          fl_line(points[i-1].x,points[i-1].y,points[i].x,points[i].y);
}
```

- Others (incl. derived classes) use **point()** and **number_of_points()**
  - why?

```
void Lines::draw_lines() const // draw a line for each pair of points
{
   for (int i=1; i<number_of_points(); i+=2)
          fl_line(point(i-1).x, point(i-1).y, point(i).x, point(i).y);
}
```

Imperial College London

Class Shape (basic idea of drawing)

```
void Shape::draw() const
   // The real heart of class Shape (and of our graphics interface system)
   // called by Window (only)
{
   // … save old color and style …
   // … set color and style for this shape…

   // … draw what is specific for this particular shape …
   // … Note: this varies dramatically depending on the type of shape …
   // … e.g. Text, Circle, Closed_polyline

   // … reset the color and style to their old values …
}
```

Imperial College London

Class Shape (implementation of drawing)

```
void Shape::draw() const
   // The real heart of class Shape (and of our graphics interface system)
   // called by Window (only)
{
   Fl_Color oldc = fl_color();// save old color
   // there is no good portable way of retrieving the current style (sigh!)
   fl_color(line_color.as_int());  // set color and style
   fl_line_style(ls.style(),ls.width());

   draw_lines(); // call the appropriate draw_lines()
        // a "virtual call"
        // here is what is specific for a "derived class" is done

   fl_color(oldc);    // reset color to previous
   fl_line_style(0);  // (re)set style to default
}
```

Note!

Imperial College London

Class shape

- In class **Shape**

```
virtual void draw_lines() const;     // draw the appropriate lines
```

- In class **Circle**

```
void draw_lines() const { /* draw the Circle */ }
```

- In class **Text**

```
void draw_lines() const { /* draw the Text */ }
```

- **Circle**, **Text**, and other classes
  - "Derive from" **Shape**
  - May "override" **draw_lines()**

Imperial College London

Object layout

- The data members of a derived class are simply added at the end of its base class (a Circle is a Shape with a radius)

Shape:

```
points
line_color
ls
```

Circle:

```
points
line_color
ls
------------
r
```

Ideals

- Our ideal of program design is to represent the concepts of the application domain directly in code.
  - If you understand the application domain, you understand the code, and *vice versa*. For example:
    - **Window** – a window as presented by the operating system
    - **Line** – a line as you see it on the screen
    - **Point** – a coordinate point
    - **Color** – as you see it on the screen
    - **Shape** – what's common for all shapes in our Graph/GUI view of the world
- The last example, **Shape**, is different from the rest in that it is a generalization.
  - You can't make an object that's "just a Shape"

Imperial College London

Operations may have the same name

- For every class,
  - `draw_lines()` does the drawing
  - `move(dx,dy)` does the moving
  - `s.add(x)` adds some `x` (*e.g.*, a point) to a shape `s`.
- For every property `x` of a Shape,
  - `x()` gives its current value and
  - `set_x()` gives it a new value
  - e.g.,

```
Color c = s.color();
s.set_color(Color::blue);
```

Imperial College London

Expose uniformly

- Data should be private
  - Data hiding – so it will not be changed inadvertently
  - Use **private** data, and pairs of public access functions to get and set the data

```
c.set_radius(12);             // set radius to 12
c.set_radius(c.radius()*2);   // double the radius (fine)
c.set_radius(-9);             // set_radius() could check for negative,
                              // but doesn't yet
double r = c.radius();        // returns value of radius
c.radius = -9;                // error: radius is a function (good!)
c.r = -9;                     // error: radius is private (good!)
```

- Private and public functions
  - Public for interface
  - Private for functions used only internally to a class

Imperial College London

Imperial College London

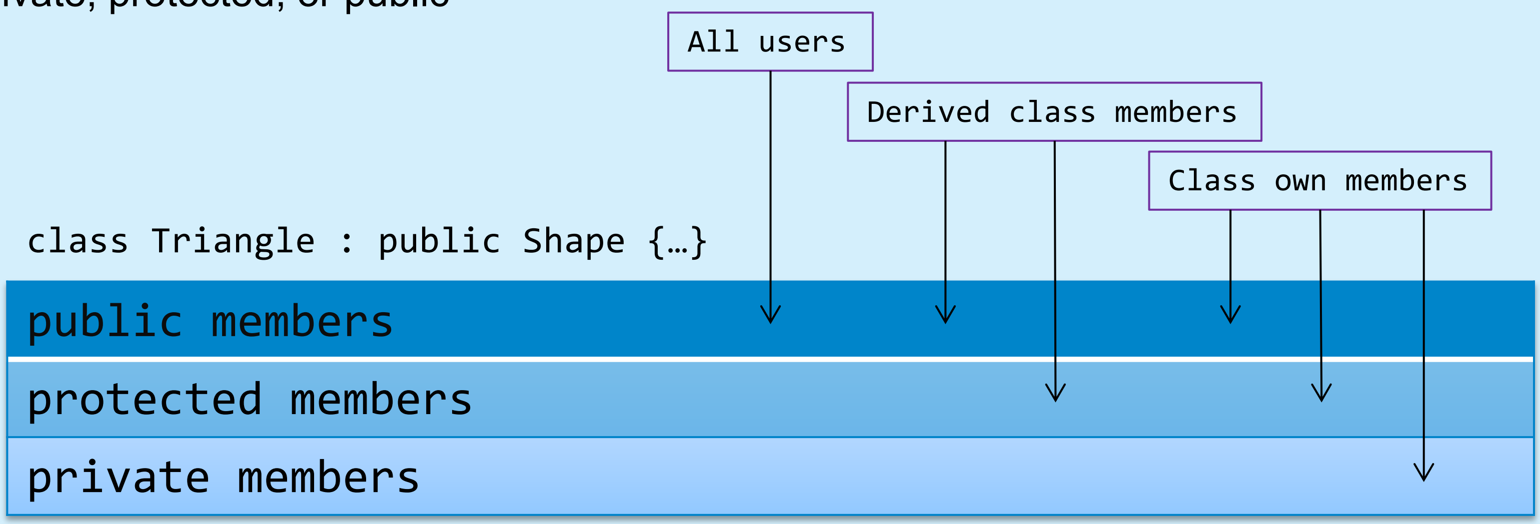
Why create "private" members and functions?

- We can change our implementation after release
- We could provide checking in access functions
  – But we haven't done so systematically (later?)
- Functional interfaces can be nicer to read and use
  – E.g., `s.add(x)` rather than `s.points.push_back(x)`
- We enforce immutability of shape
  – Only color and style change; not the relative position of points
  – **const** member functions
- The value of this "encapsulation" varies with application domains
  – Is often most valuable
  – Is the ideal
    - i.e., hide representation unless you have a good reason not to

Imperial College London

Access model

- A member (data, function, or type member) or a base can be
  - Private, protected, or public



Three ways of inheriting

```
class Triangle : public Shape {…}
class Triangle : protected Shape {…}
class Triangle : private Shape {…}
```

Type of inheritance

Base class member access specifier	Type of inheritance		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	Not accessible (hidden)	Not accessible (hidden)	Not accessible (hidden)

EXERCISE

1. Create a base class "Animal" with functions "make_sound()" and "move()". Create two derived classes "Dog" and "Bird". Define the functions "make_sound()" and "move()" in the derived classes as well.

Imperial College London

```
class Shape  {  // deals with color and style, and holds a sequence of lines
public:
    void draw() const;           // deal with color and call draw_lines()
    virtual void move(int dx, int dy);      // move the shape +=dx and +=dy

    void set_color(Color col);   // color access
    int color() const;
    // … style and fill_color access functions …

    Point point(int i) const;    // (read-only) access to points
    int number_of_points() const;
protected:
    Shape();                     // protected to make class Shape abstract
    void add(Point p);           // add p to points
    virtual void draw_lines() const;  // simply draw the appropriate lines
private:
    vector<Point> points;        // not used by all shapes
    Color  lcolor;               // line color
    Line_style  ls;              // line style
    Color  fcolor;               // fill color

    // … prevent copying …
};
```

Pure virtual functions

- Often, a function in an interface can't be implemented
  - E.g. the data needed is "hidden" in the derived class
  - We must ensure that a derived class implements that function
  - Make it a "pure virtual function" (**=0**)

- This is how we define truly abstract interfaces ("pure interfaces")

```
struct Shape {  // interface to electric motors
    // no data
    // (usually) no constructor
    virtual double Draw() = 0;   // must be defined in a derived class
    // …
    virtual ~Shape(); // (usually) a virtual destructor
};
Shape s;   // error: Collection is an abstract class
```

Pure virtual functions

- A pure interface can then be used as a base class
  - Constructors and destructors are defined in the derived class, but can also exist in the base class

```
class Triangle : public Shape {  // a specific shape
    // representation
public:
    Triangle();  // constructor:  initialization, acquire resources
    double Draw() { /* … */ }    // overrides Shape ::draw
    // …
    ~Triangle(); // destructor: cleanup, release resources
};

Triangle t;     // OK
```

Imperial College London

Technicality: Copying

- If you don’t know how to copy an object, prevent copying
  - Abstract classes typically should not be copied

```
class Shape {
    // …
    Shape(const Shape&) = delete;                   // don’t “copy construct”
    Shape& operator=(const Shape&) = delete;        // don’t “copy assign”
};

void f(Shape& a)
{
    Shape s2 = a;       // error: no Shape “copy constructor” (it’s deleted)
    a = s2;             // error: no Shape “copy assignment” (it’s deleted)
}
```

Prevent copying C++98 style

- If you don't know how to copy an object, prevent copying
  - Abstract classes typically should not be copied

```
class Shape {
    // …
private:
    Shape(const Shape&);               // don't "copy construct"
    Shape& operator=(const Shape&);    // don't "copy assign"
};

void f(Shape& a)
{
    Shape s2 = a;      // error: no Shape "copy constructor" (it's private)
    a = s2;            // error: no Shape "copy assignment" (it's private)
}
```

Imperial College London

Technicality: Overriding

- To override a virtual function, you need
  - A virtual function
  - Exactly the same name
  - Exactly the same type

```
struct B {
    void f1();   // not virtual
    virtual void f2(char);
    virtual void f3(char) const;
    virtual void f4(int);
};
```

```
struct D : B {
    void f1();          // doesn't override
    void f2(int);       // doesn't override
    void f3(char);      // doesn't override
    void f4(int);       // overrides
};
```

Technicality: Overriding

- To override a virtual function, you need
  - A virtual function
  - Exactly the same name
  - Exactly the same type

```
struct B {
    void f1();   // not virtual
    virtual void f2(char);
    virtual void f3(char) const;
    virtual void f4(int);
};
```

```
struct D : B {
    void f1() override;          // error
    void f2(int) override;       // error
    void f3(char) override;      // error
    void f4(int) override;       // OK
};
```

Technicality: Overriding

- To invoke a virtual function, you need
  - A reference, or
  - A pointer

```
D d1;
B& bref = d1;          // d1 is a D, and a D is a B, so d1 is a B
bref.f4(2);            // calls D::f4(2) on d1 since bref names a D

// pointers (we will cover this soon)
B *bptr = &d1;         // d1 is a D, and a D is a B, so d1 is a B
bptr->f4(2);           // calls D::f4(2) on d1 since bptr points to a D
```

This is important, as polymorphism relies on pointers to function.

Imperial College London

Effect of ‘virtual’ when casting

```
#include <iostream>
using namespace std;
class Person
{
public:
  virtual void talk()
  {
    cout << "I’m a person " << endl;
  }
};

class Student: public Person
{
public:
  virtual void talk()
  {
    cout << "I’m a student " << endl;
  }
  void study()
  {
    cout<< "study" << endl;
  }
};
```

```
class ESEStudent : public Student
{
public:
  virtual void talk()
  {
    cout << "I’m an ESE student" << endl;
  }
  void writeCode()
  {
    cout << "write Code" << endl;
  }
};
int main()
{
  ESEStudent ese_st;
  ese_st.talk(); //"I’m an ESE student"
  Person& asPerson = ese_st;
  asPerson.talk(); //"I’m an ESE student"
  return 0;
}
```

Imperial College London

Effect of ‘virtual’ when casting

```
#include <iostream>
using namespace std;
class Person
{
public:
  void talk()
  {
    cout << "I’m a person " << endl;
  }
};

class Student: public Person
{
public:
  void talk()
  {
    cout << "I’m a student " << endl;
  }
  void study()
  {
    cout<< "study" << endl;
  }
};
```

```
class ESEStudent : public Student
{
public:
  void talk()
  {
    cout << "I’m an ESE student" << endl;
  }
  void writeCode()
  {
    cout << "write Code" << endl;
  }
};
int main()
{
  ESEStudent ese_st;
  ese_st.talk(); //"I’m an ESE student"
  Person& asPerson = ese_st;
  asPerson.talk(); //"I’m a person"
  return 0;
}
```

Imperial College London

Two ways of doing the same

```
class Animal
{
public:
  AnimalType type;
  virtual string talk()
  {
    switch(type) {
    case CAT: return "Meow!";
    case DOG: return "Woof!";
    case DUCK: return "Quack!";
    case PIG: return "Oink!";
    default:
     assert(0);
    }
    return string();
  }
}
```

```
class Animal
{
  public:
    virtual string talk() = 0;
};
class Cat : public Animal
{
  public:
    virtual string talk() { return "Meow!"; }
};
class Dog : public Animal
{
  public:
    virtual string talk() { return "Woof!"; }
};
class Duck : public Animal
{
  public:
    virtual string talk() { return "Quack!"; }
};
class Pig : public Animal
{
  public:
    virtual string talk() { return "Oink!"; }
};
```

Imperial College London

An abstract class needs a constructor

```
class Animal
{
private:
  string name;
public:
  Animal(const string& name_):
     name(name_) {}
  virtual string talk() = 0;
  virtual int getNumLegs() = 0;
  virtual void walk() = 0;
};
```

```
class Cat : public Animal
{
public:
  Cat(const string& name_) : Animal(name_) {}
  virtual string talk() { return "Meow!"; }
  virtual intgetNumLegs() = { return 4; }
  virtual void walk() {...};
};

class Dog : public Animal
{
public:
  Dog(const string& name_) : Animal(name_) {}
  virtual string talk() { return "Woof!"; }
  virtual int getNumLegs() = { return 4; }
  virtual void walk() {...};
};
```

Imperial College London

An abstract class needs a virtual destructor

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

class Shape
{
public:
  Shape() {}
  virtual~Shape() {}
  virtual void draw() = 0;
};
```

```
class Rectangle : public Shape
{
public:
  Rectangle()
  {
      width = new int;
      height = new int;
  }
  virtual ~Rectangle()
  {
    delete width;
    delete height;
  }
  virtual void draw()
  { ... }

private:
  int* width;
  int* height;
};
```

```
int main()
{
  Shape* shape1 = new Rectangle;
  shape1->draw();
  delete shape1;
  return 0;
}
```

An abstract class should have a virtual destructor even if it does nothing.

A destructor of a *base* class **should be** `virtual` if

- its descendant class instance is deleted by the base class pointer. (or)
- any of member function is **virtual** (which means it's a polymorphic base class).

Type-Casting

Change the type of an object (force it)

- `static_cast<T>(var)`
- `dynamic_cast<T*>(ptr)`
- `const_cast<T*>(ptr)`
- `reinterpret_cast<T*>(ptr)` (legacy)

Each operator is designed to be used for specific purpose

static_cast

- `static_cast` performs type checking at compile time.
- Safe for upcast(derived → base)
- Unsafe for downcast (base → derived)
- It’s the programmer's responsibility to make sure that base class pointer is actually pointing to a specified derived class object.
- Can be used for casting between primitive types

```
int i= static_cast<int>(2.0);
```

Imperial College London

static_cast

```
class B{};
class D: public B
{
  public:
    int member_D;
    void test_D() { member_D = 10; }
};
class X{};

int main() {
     B b; D d; char ch; inti=65;
     B* pb= &b; D* pd= &d;
     D* pd2 = static_cast<D*>(pb);    // Unsafe. If you access pd2's members not
                                           // in B, you get a run time error.
     pd2->test_D();                        // Runtime error!
     B* pb2 = static_cast<B*>(pd);    // Safe, D always contains all of B.
     X* px= static_cast<X*>(pd);           // Compile error!
     ch= static_cast<char>(i);             // int to char
}
```

Imperial College London

`dynamic_cast`

- `dynamic_cast` performs type checking at run time.
- Safe for downcast
- If base class pointer is not pointing to a specified derived class object, `dynamic_cast` of base to derived pointer returns null pointer (0).
- Note that `dynamic_cast` can only downcast polymorphic types (base class should have at least one virtual function).

Imperial College London

dynamic_cast

```
#include <iostream>
class B
{
  public:
    virtual ~B() {}
};

class D: public B
{
  public:
  void test_D()
  {
    std::cout<< "test_D()" << std::endl;
  }
};
```

```
int main()
{
  B b; D d;
  B* pb= &b;
  //B* pb= &d;
  D* pd2 = dynamic_cast<D*>(pb);
  if(pd2)
    pd2->test_D();
}
```

const_cast, reinterpret_cast

`const_cast` removes 'const' from `const T* ptr`

`reinterpret_cast` is just like C-style cast; avoid using it.

```
class B {};
class X {};
int main() {
B b;
B* pb= &b;
const B* cpb= pb;
B* pb2 = const_cast<B*>(cpb);
X* px= reinterpret_cast<X*>(pb);
}
```

Imperial College London

Casting vs. Polymorphism

- Casting is different to polymorphism.
- Casting is a solution to what is usually a design issue.
- Avoid casting as far as possible. Prefer polymorphism.

EXERCISE

1. Part I: Create a base class "Animal" with virtual functions "make_sound()" and "move()". Create two derived classes "Dog" and "Bird". Override the virtual functions in the derived classes with appropriate functionality. Create instances of both the derived classes and call their virtual functions.

2. Part II: Create a base class "Animal" with pure virtual functions "make_sound()" and "move()". Create three derived classes "Dog", "Cat", and "Bird". Override the virtual functions in each of the derived classes with appropriate functionality. Create a vector of "Animal" pointers that contains several "Dog" objects, several "Cat" objects, and several "Bird" objects. Randomly shuffle the vector and then loop through the vector and call the "make_sound()" and "move()" functions on each object.


TIME FOR
A BREAK!

Templates

- Another way of controlling type morphism
- Templating is a mechanism in C++ to create classes in which one or more types are *parameterised*
- Allows for *generic programming*
- Compile-time typing

This is an example of a template function

```
template <typename T>
T minimum(const T& lhs, const T& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

This is a ternary operator equivalent to:

```
if(lhs < rhs)
  return lhs;
else
  return rhs;
```

Imperial College London

Example of use

```
template <typename T>
T minimum(const T& lhs, const T& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

Needs to be defined in the object used

```
int a = 1;
int b = 200;
int i = minimum<int>(a, b); //i=1

double a = 3.5;
double b = 7.2;
double i = minimum<double>(a, b); //i=3.5

std::string a = “aa”;
std::string b = “zzz”;
std::string i = minimum<string>(a, b); //i=“aa”
```

These examples use in-built types

This is an example adapted from https://learn.microsoft.com/en-us/cpp/cpp/

Imperial College London

Example of “Issue”

```
template <typename T>
T minimum(const T& lhs, const T& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

```
class Greeting
{
public:
    int num;
    std::string description;
};

int main()
{
    Greeting mc1 {1, "hello"};
    Greeting mc2 {2, "goodbye"};
    auto result = minimum(mc1, mc2); // error: the operator is not defined
}
```

This newly created class cannot use our template “`minimum`” function, because it does not have the `<` operator defined!

Template

You can have multiple template parameters

```
template <typename T, typename U, typename V> class XY{};
```

The keyword class is equivalent to typename in this context.

```
template <class T, class U, class V> class XY{};
```

You can use the ellipsis operator (...) to define a template that takes an arbitrary number of zero or more type parameters [aka. ‘variadic template’]

```
template<typename... Arguments> class XYZ;
XYZ< > xyz_instance1;
XYZ<int> xyz_instance2;
XYZ<string, float> xyz_instance3;
```

Example

```
template<typename T, size_t L> //note that L is a size_t
class MyArray
{
    T arr[L];   //I can create a very flexible static array!
public:
    MyArray() { ... }
};
```

Note the syntax in the template declaration. The `size_t` value is passed in as a template argument at compile time and must be a `const` or a `constexpr` expression. You use it like this:

```
MyArray<MyClass*, 10> arr;
```

auto Templates

In Visual Studio 2017 and later, and in /std:c++17 mode or later, the compiler deduces the type of a non-type template argument that's declared with auto:

```
template <auto x> constexpr auto constant = x;
```

This is an ‘advanced’ type of template

```
auto v1 = constant<5>;      // v1 == 5, decltype(v1) is int
auto v2 = constant<true>;   // v2 == true, decltype(v2) is bool
auto v3 = constant<'a'>;    // v3 == 'a', decltype(v3) is char
```

Imperial College London

Templates as template parameters

A template can be a template parameter. In this example, MyClass2 has two template parameters: a typename parameter T and a template parameter Arr

```
template<typename T, template<typename, int> class Game>
class SundayActivity
{
    T t; //OK
    Game<T, 10> a;
    U u; //Error. U not in scope
};
```

Notice this is now specifically an int

Parameter names are not needed

Imperial College London

Default Template Arguments

```
template<typename A = int, typename B = double>  //defaults
class Bar
{
    //...
};

int main()
{
    Bar<> bar; // use all default type arguments
}
```

Imperial College London

Define ‘partial specialisation’

```
template <typename K, typename V>
class MyMap{/*...*/};

// partial specialization for string keys – define specifics
template<typename V>
class MyMap<string, V> {/*...*/};

MyMap<int, MyClass> classes; // uses original template

MyMap<string, MyClass> classes2;
// uses the partial specialization
```

Imperial College London

EXERCISE

Part 1. Consider the function

```
int fact(int c)
{
  int factorial = 1;
  for (int i = 1; i <= c; i++)
  {
    factorial *= i;
  }
  return factorial;
}
```

Create a main function that prints the first 100 factorials to screen.

Part 2. Generalise the function so that it has a template parameter

```
template<typename T>
T fact(T c)
```

Change your main function to call `fact` with the following parameters: `int`, `long long`, `double`, `long double`.

Print the first 100 factorials to screen.

What has changed?

Imperial College London

- Additional Slides

Imperial College London

Examples of C++ Libraries

Qt

TensorFlow

boost
C++ LIBRARIES

JSON

VTK

Intel® oneAPI DPC++ Library
Speed Up DPC++ (SYCL*) Kernels on CPUs, GPUs, and FPGAs

FFmpeg

OpenSSL
Cryptography and SSL/TLS Toolkit

Imperial College London

Linking a library

- Collection of pre-compiled functions and classes that can be used by a program.
- A collection of classes and functions meant to be used together
  - As building blocks for applications
  - To build more such "building blocks"
- A good library models some aspect of a domain
  - It doesn't try to do everything
  - Libraries aim at simplicity and small size
- A good library exhibits a uniform style ("regularity")
  - Uniform and related concepts
  - Threaded style and strategy
  - Common purpose

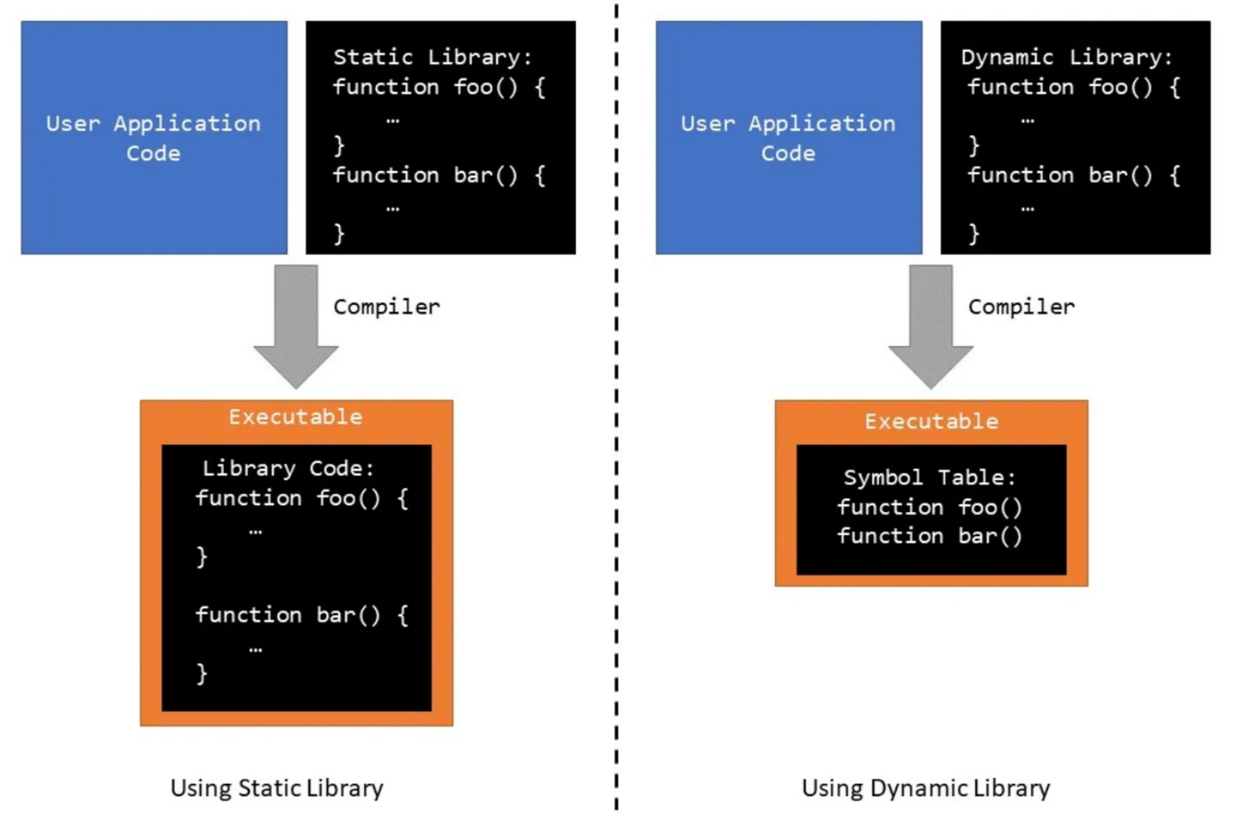
Imperial College London

Types of library: “static” and “dynamic”

- A static library is a library where the object code of the library is linked directly into the executable file at compile time. (*.lib (Windows), *.a(Unix/Mac))
  - Functions and classes in the library are copied into the executable file, and the executable file does not depend on the presence of the library at runtime.
  - Used when the library is small and does not change frequently.
- A dynamic library, also known as a "shared library", is a library where the object code of the library is linked at runtime. (*.dll (Windows), *.so(Unix), .dylib(Mac))
  - The executable file contains only references to the library functions and classes, and the library is loaded into memory when the program is run.
  - Typically used when the library is large or frequently updated, allows multiple programs to share the same library code in memory.

Imperial College London

Static vs. Dynamic Linking

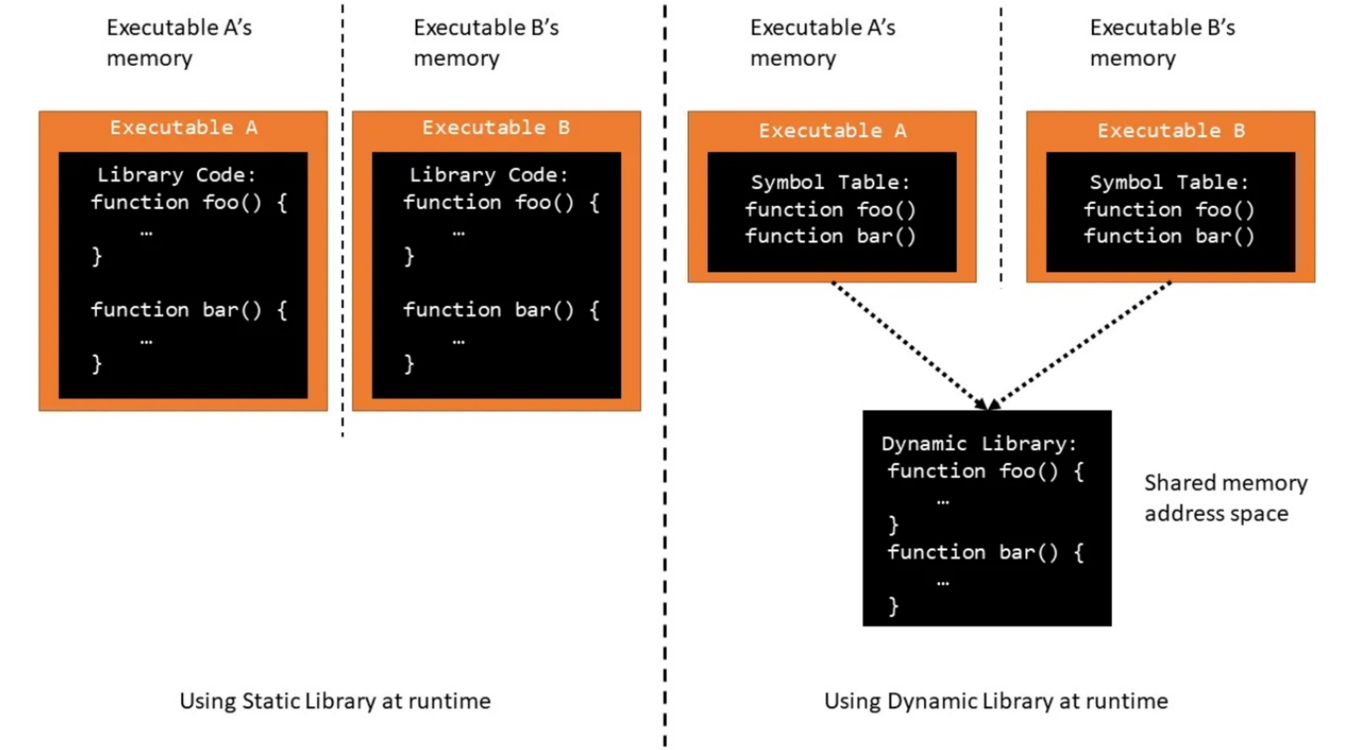


Static Library vs Dynamic Library at Compile Time

Imperial College London

Static vs. Dynamic Linking

dlls will be sharing memory space, unless they are "thread safe"! Warning!



Static Library vs Dynamic Library at Runtime

Imperial College London

Static vs. Dynamic Linking

STATIC Linking

Copies all libraries required directly into the executable

Simplifies process of distributing / deploying your app to multiple OS

Can be faster to startup

Large program files

Maintenance requires app update

DYNAMIC Linking

External / shared libraries copied into executable only by name

Lower maintenance cost / Separate tasks

Update routines without needing to relink

Smaller file sizes

Changes made to library may result in incompatibilities

(more code protection)