



**UNSW**  
SYDNEY

# **Review of Core Concepts of Object-Oriented Programming (OOP) in C++**

A/Prof Shiyang Tang

**MTRN3500**

Computing Applications in Mechatronics Systems

OOP in C++ is a programming paradigm that organises code into *objects*, which are instances of *classes*. Core concepts of OOP:

## 1. Class and Object

- A class is a blueprint for creating objects. It defines attributes (data) and methods (functions).
- An object is an instance of a class.

## 2. Encapsulation

- Bundles data and related functions into a single unit (class).
- Controls access using access specifiers: *private*, *protected*, and *public*.

## 3. Inheritance

- Allows a class to derive properties and behaviour from another class.
- Promotes code reuse.

## 4. Polymorphism

- Allows functions or objects to behave differently based on context.
- Achieved through function overloading, operator overloading, and virtual functions.

# Encapsulation

**Encapsulation** bundles *data* and *functions* into a single unit (class) and restricts direct access to internal data.

```
class BankAccount {  
private:  
    double balance; // hidden data  
  
public:  
    void deposit(double amount) {  
        if (amount > 0) balance += amount;  
    }  
  
    double getBalance() {  
        return balance;  
    }  
};
```

Access Level	Accessible from inside the class?	Accessible from derived classes?	Accessible from outside the class?
`public`	Yes	Yes	Yes
`protected`	Yes	Yes	No
`private`	Yes	No	No

# Inheritance

**Inheritance** allows a class to acquire *properties* and *functions* from another class.

```
class Animal {
public:
    void eat() {
        std::cout << "This animal eats food.\n";
    }
};

class Dog : public Animal {
public:
    void bark() {
        std::cout << "The dog barks.\n";
    }
};
```

```
Dog d;    // Creates an object d of type Dog,
d.eat();  // Inherited from Animal
d.bark(); // Defined in Dog
```

**Polymorphism** enables objects to be treated as instances of their base type while still invoking derived class behaviour.

## 1. Compile-time polymorphism (*via* function overloading)

```
class Math {
public:
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }
};
```

```
Math math;    // Create an object math of type Math
cout << "2 + 3 = " << math.add(2, 3) << endl;    // Call the version: int add(int a, int b)
cout << "2.5 + 3.1 = " << math.add(2.5, 3.1) << endl;    // Call the overloaded version: double add(double a, double b)
cout << "1 + 2 + 3 = " << math.add(1, 2, 3) << endl;    // Call another overloaded version
```

## 2. Run-time polymorphism (via *virtual* functions and inheritance)

- A base-class pointer or reference calls a *virtual* method.
- The method that gets executed depends on the actual (derived) object type, not the pointer type.

```
// Base class
class Animal {
public:
    // Virtual function enables runtime polymorphism
    virtual void Speak() {
        cout << "Animal makes a sound." << endl;
    }

    // Always define a virtual destructor in base classes
    virtual ~Animal() {}
};

// Derived class: Dog
class Dog : public Animal {
public:
    void Speak() override {
        cout << "Dog says: Woof!" << endl;
    }
};
```

```
Animal* pet1 = new Dog(); // base class pointer to Dog object
pet1->Speak(); // Call the Speak() function through a base class pointer
// Output: Dog says: Woof!
```

# Constructor and Destructor

**Constructor** and **Destructor** are special member functions used for object initialisation and cleanup.

## Constructor

- Purpose: Initialises an object when it is created.
- Name: Same as the class name, no return type.

```
class MyClass
{
public:
    MyClass(); //Default constructor
    MyClass(int x); //Parameterised constructor
};
```

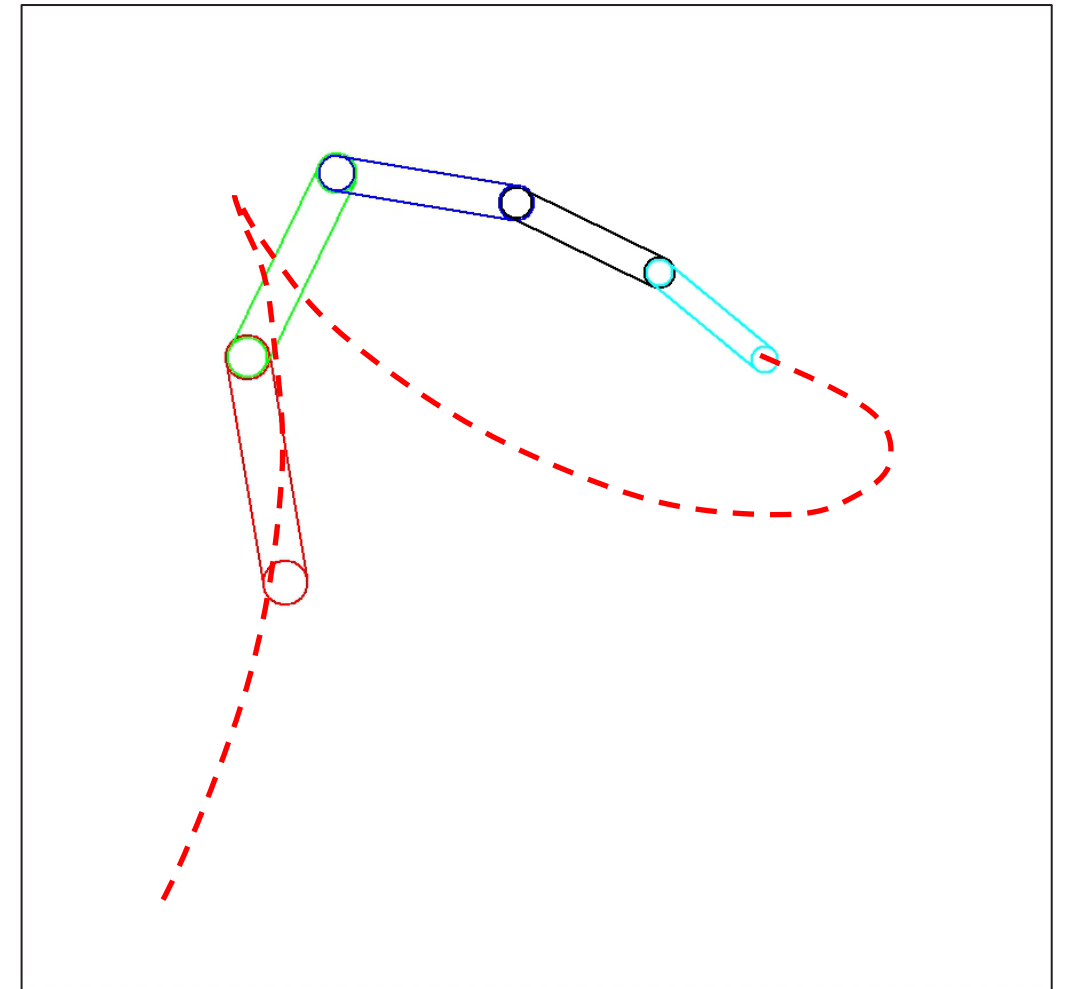
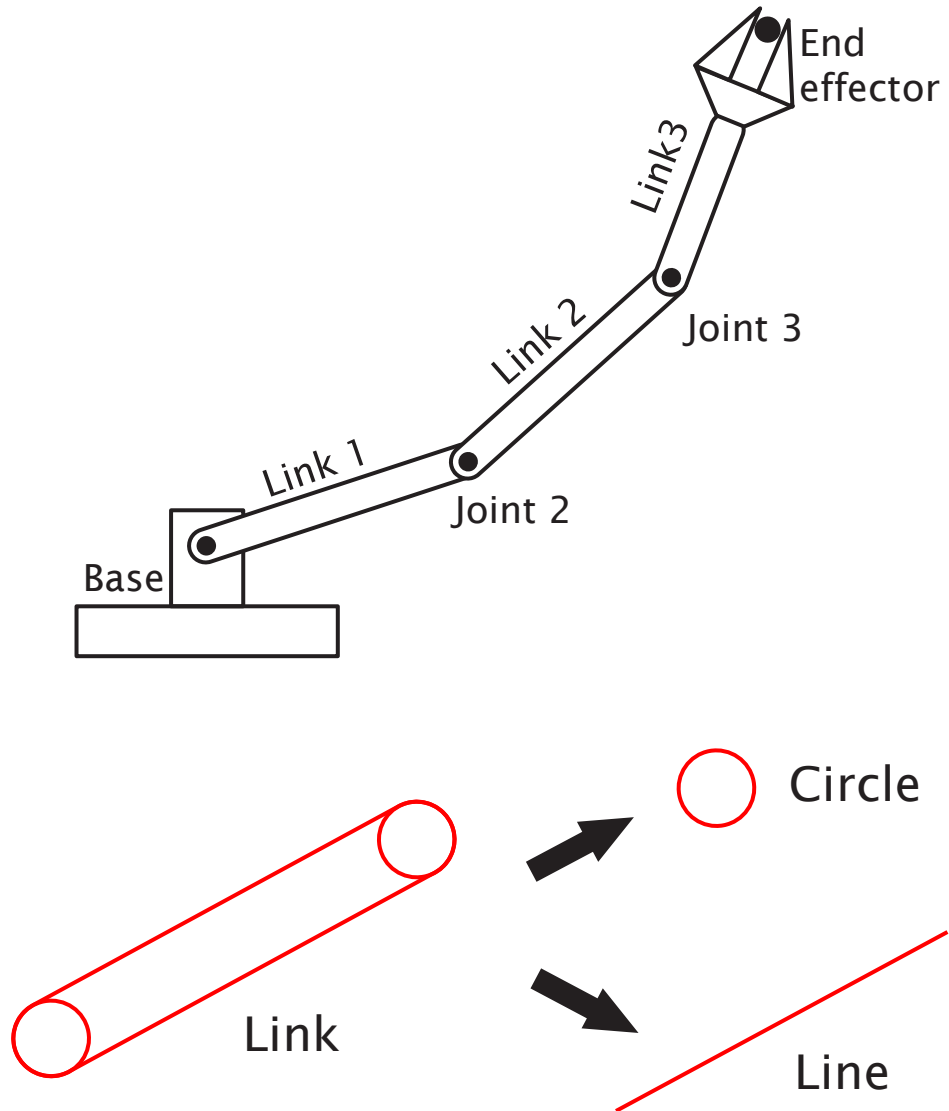
## Destructor

- Purpose: Cleans up resources when the object is destroyed.
- Name: Same as the class but prefixed with ~.
- No parameters and no return type.
- Called automatically when the object is **deleted**.

```
class MyClass {
public:
    ~MyClass(); // Destructor
};
```

Feature	Constructor	Destructor
Name	Same as class	Same as class with ~
Return type	None	None
Parameters	Yes (except default)	No
Overloadable	Yes	No
Called when	Object is created	Object is destroyed

# Learn from an Example



Track and export the coordinates of the robotic arm's endpoint.



# Appendix 1: Member Initialiser List

```
Shapes::Shapes(Point p) : P(p), Orientation(0.0), R(0), G(0), B(0)
{
}
```

The line after the colon “:” is called the **member initialiser list**, and it directly initialises the member variables.

“::” is the **scope resolution** to access members (constructors, functions, etc).

This constructor is roughly equivalent to:

```
Shapes::Shapes(Point p)
{
    P = p;
    Orientation = 0.0;
    R = 0;
    G = 0;
    B = 0;
}
```

```
Circle::Circle(Point c, double radius, int r, int g, int b) : Radius(radius), Shapes(c, 0.0, r, g, b)
{
}
```

You need to explicitly call the appropriate `Shapes(...)` constructor inside any `Circle(...)` constructor that takes arguments, because the compiler does not automatically know which *Shapes* constructor to use unless it's default.

```
Circle::Circle(Point c, double radius, int r, int g, int b) : P(c), Radius(radius), R(r), G(g), B(b)
```

`P`, `R`, `G`, and `B` are not members of `Circle`, they are protected members of *Shapes*. Although `Circle` inherits the parameters from the base class `Shapes`, in C++, you **cannot** directly initialise base-class members from a derived class constructor's initialiser list.

# Appendix 2: Arc Draw Function

```
void Draw(HDC h)
{
    int X = P.GetX(), Y = P.GetY();

    COLORREF NewColor = RGB(R, G, B);
    HPEN hNewPen = CreatePen(PS_SOLID, 2, NewColor);
    SelectObject(h, hNewPen);
    Arc(h, X - (int)Radius, Y - (int)Radius, X + (int)Radius, Y + (int)Radius, X - (int)Radius, Y, X - (int)Radius, Y);
    DeleteObject(hNewPen);
}
```

- The function takes a Windows GDI device context handle (HDC h) as an input.
- RGB(R, G, B) creates a COLORREF value from the red, green, and blue components.
- CreatePen(...) creates a solid colour pen with a thickness of 2 pixels.

```
Arc(
    HDC hdc,           // handle to device context
    int left,          // bounding box left
    int top,            // bounding box top
    int right,          // bounding box right
    int bottom,         // bounding box bottom
    int xStartArc,      // starting point x
    int yStartArc,      // starting point y
    int xEndArc,        // ending point x
    int yEndArc,        // ending point y
);
```

- Start and end points are both (X - Radius, Y). This tricks Arc() into drawing a full ellipse, which is a circle since width = height.
- Note that the data type is **int** as pixels are discrete.

# Appendix 3: Line Draw Function

```
void Line::Draw(HDC h)
{
    int X = P.GetX();
    int Y = P.GetY();
    COLORREF NewColor = RGB(R, G, B);
    HPEN hNewPen = CreatePen(PS_SOLID, 2, NewColor);
    SelectObject(h, hNewPen);
    MoveToEx(h, P.GetX(), P.GetY(), NULL);
    LineTo(h, P.GetX() + (int)(Length * cos(Orientation)), P.GetY() + (int)(Length * sin(Orientation)));
    DeleteObject(hNewPen);
}
```

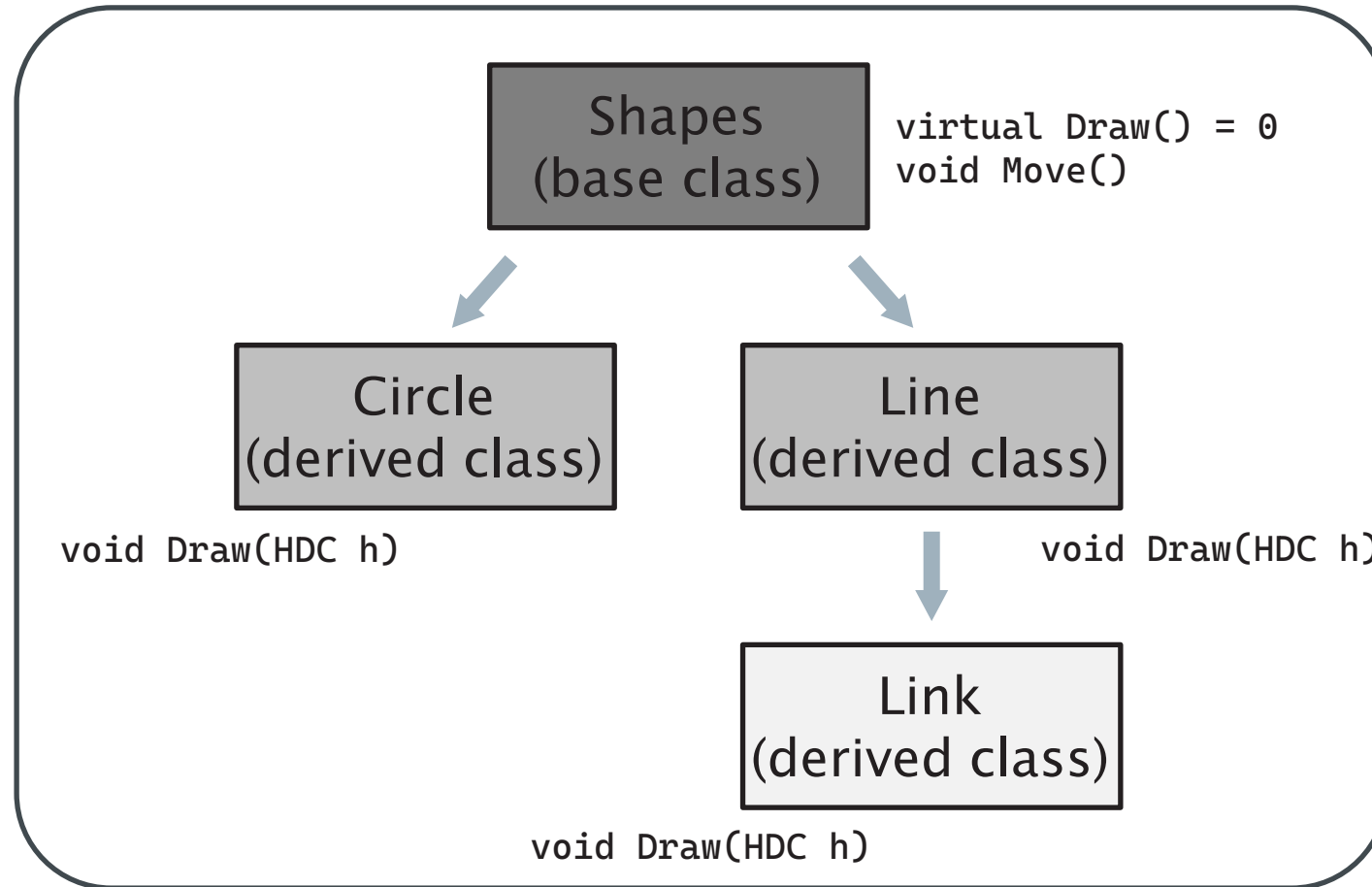
MoveToEx(h, P.GetX(), P.GetY(), NULL);

- This tells GDI: "Move the cursor (starting point of line drawing) to (X, Y)".
- The 4th parameter is a pointer to receive the old position (NULL if not needed).

```
BOOL LineTo(
    HDC hdc,    // Handle to the device context
    int x,      // X-coordinate of the endpoint
    int y       // Y-coordinate of the endpoint
);
```

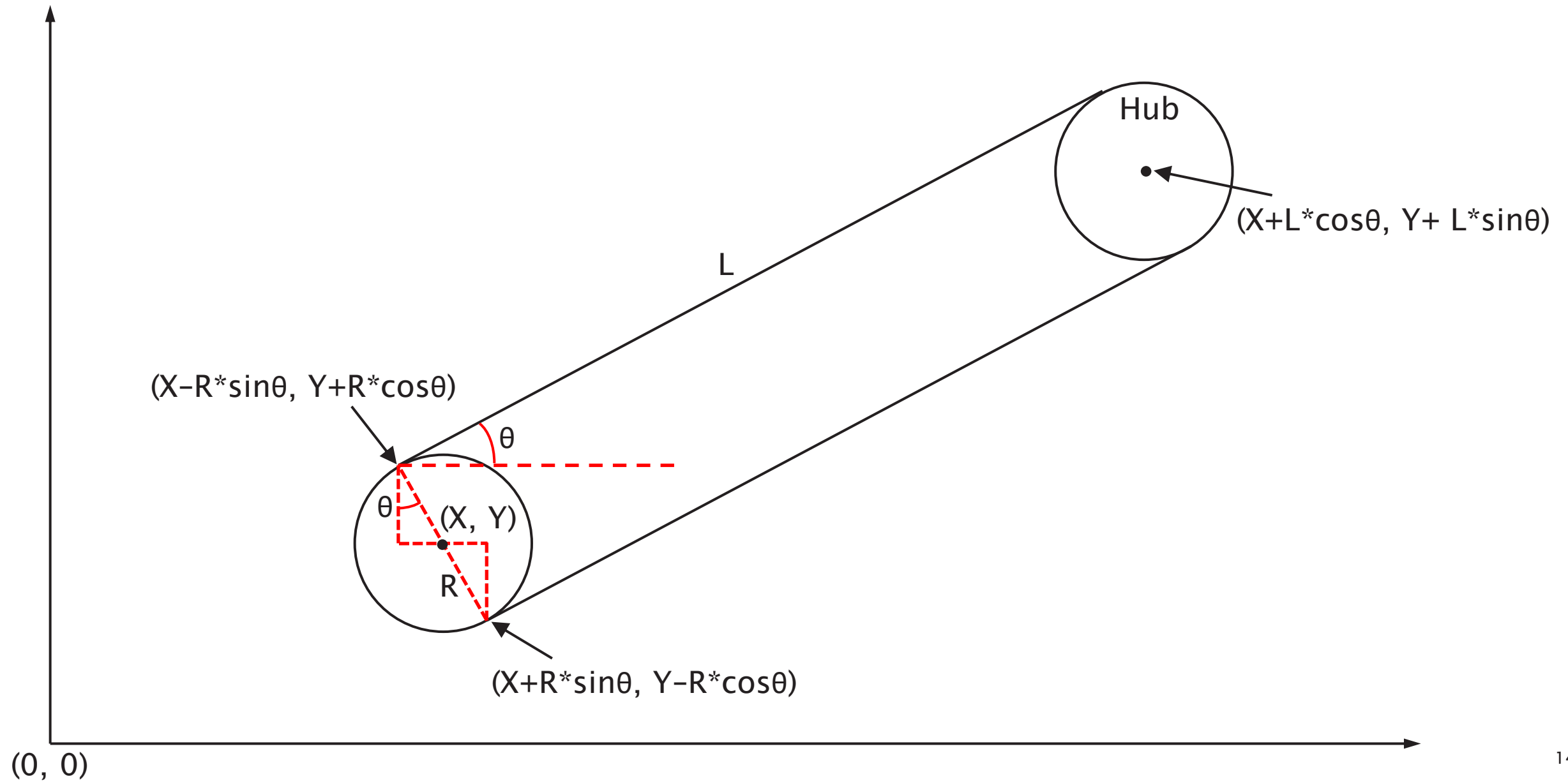
# Appendix 4: Class Structure

Point  
(helper)



Robot  
`void Draw(HDC h)`

# Appendix 5: Link Configuration



# Appendix 6: case WM\_PAINT

```
RECT ClientRect;
GetClientRect(hWnd, (LPRECT)&ClientRect);

PAINTSTRUCT ps;
HDC hdc = BeginPaint(hWnd, &ps);
ShapesPtr->Draw(hdc);
// TODO: Add any drawing code that uses hdc here...
EndPaint(hWnd, &ps);

Sleep(40);
ShapesPtr->Move(1, 0, PI/32);
InvalidateRect(hWnd, (const RECT*)&ClientRect, true);
```

- ClientRect is a RECT structure that stores the dimensions of the window's drawable area.
- GetClientRect fills ClientRect with the coordinates relative to the top-left corner (typically (0, 0) to (width, height)).
- Marks the entire ClientRect area as invalid, telling Windows: "this part needs to be redrawn".
- Triggers a new WM\_PAINT message in the next message loop cycle.
- The true argument means to erase the background, so the window is cleared before repainting.

# Appendix 7: struct vs class

A **struct** (short for structure) is a user-defined data type that groups related variables (called members) under a single name.

A **struct** can have constructors, functions, inheritance, operator overloading, etc. It is used for:


- Simple data containers
- When you want all members to be *public* by default

Feature	'struct'	'class'
Default access	'public'	'private'
Inheritance	'public' by default	'private' by default
Syntax style	Often used for plain data	Often used for full OOP



```
struct LinkProperties
{
    double LinkLength;
    double Angle;
    double HubRad;
    int Red;
    int Green;
    int Blue;
};
```

```
std::vector<LinkProperties> data
```



```
{200, -PI / 2.0, 20, 255, 0, 0},
{180, -PI / 4.0, 18, 0, 255, 0},
{160, PI / 8.0, 16, 0, 0, 255},
{140, PI / 4.0, 14, 255, 255, 0},
{120, PI / 6.0, 12, 0, 255, 255}
```

```
for (std::vector<LinkProperties>::iterator it = data.begin(); it != data.end(); it++)
```

- The for loop iterates through the data vector using an *iterator* (“it”, like a pointer), pointing to each LinkProperties struct in the vector, one by one.
- So “it” is an iterator object that “points to” elements inside a std::vector<LinkProperties>.

```
    LinkPtrs[i++] = new Link(Point(0, 0), it->LinkLength, it->Angle, it->HubRad, it->Red, it->Green, it->Blue);
```

- This line creates a new Link object (on the heap) using the data pointed to by “it”, and assigns it to LinkPtrs[i].
- it->LinkLength is equivalent to: (\*it).LinkLength

# Appendix 9: Operator Overloading

**Operator overloading** allows developers to redefine the meaning of standard operators (like +, -, \*, <<, etc.) for user-defined types such as classes and structs.

$$\begin{array}{l} \vec{v}_1 = (x_1, y_1) \\ \vec{v}_2 = (x_2, y_2) \end{array} \Rightarrow \vec{v}_1 + \vec{v}_2 = (x_1 + x_2, y_1 + y_2)$$

Without operator overloading

```
class Vector2D {
    double X, Y;
public:
    Vector2D(double x, double y) : X(x), Y(y) {}

    Vector2D add(const Vector2D& other) const
    {
        return Vector2D(X + other.X, Y + other.Y);
    }
};
```

```
Vector2D a(1.0, 2.0);
Vector2D b(3.0, 4.0);
Vector2D c = a.add(b);
```

a → “this” object

b → “other” object

The add() function takes a reference to another Vector2D object, named other, but won't modify it (const).

const after the function means this function also won't modify the current object (“this” object).

Using add() makes the code verbose and less intuitive, especially if you're dealing with multiple chained operations (e.g., a + b + c).

# Appendix 9: Operator Overloading

Using operator overloading

```
class Vector2D {  
private:  
    int X, Y;  
  
public:  
    Vector2D(double x, double y) : X(x), Y(y) {}  
  
    Vector2D operator+(const Vector2D& other) const  
    {  
        return Vector2D(X + other.X, Y + other.Y);  
    }  
};
```

This is a *member function* that *overloads* the “+” operator

The ampersand & means that the parameter **other** is passed by reference, not a copy. Copying can be slow, especially for large objects.

```
Vector2D a(1.0, 2.0);  
Vector2D b(3.0, 4.0);  
Vector2D c = a + b;
```

**a + b** calls the function on **a** (“this” object), passing **b** as “other” object.  
This supports chaining (e.g., **d = a + b + c**)

Operator	Overload Type	Notes
+	Member / Friend	Arithmetic addition
-	Member / Friend	Arithmetic subtraction
*	Member / Friend	Multiplication
/	Member / Friend	Division
<<	Friend	Stream output
>>	Friend	Stream input



# Appendix 9: Operator Overloading

How to print/export the vector?

`Vector2D v(1, 2);`  
`std::cout << v;` → The compiler doesn't know how to handle `<< v` because:

- `std::cout` only knows how to print basic types like *int*, *double*, *char*, etc.
- `Vector2D` is your own custom class.

The solution is to **overload** the “<<” operator so it knows how to turn a `Vector2D` into readable text.

```
class Vector2D {
private:
    int X, Y;

public:
    Vector2D(double x, double y) : X(x), Y(y) {}

    Vector2D operator+(const Vector2D& other) const
    {
        return Vector2D(X + other.X, Y + other.Y);
    }

    // Friend function: overload << to print the vector
    friend std::ostream& operator<<(std::ostream& os, const Vector2D& vec);
};

// Define the non-member << operator for printing
std::ostream& operator<<(std::ostream& os, const Vector2D& vec)
{
    os << "(" << vec.X << ", " << vec.Y << ")";
    return os;
}
```

friend means: “This function is allowed to access private members of this class.”

`vec.X` and `vec.Y` are private, so without friend, the function would not compile.

`std::ostream` is the C++ type for output streams. Examples of objects:

- `std::cout` → prints to console
- `std::ofstream` → prints to a file

→ Takes two Inputs:

- `os`: the output stream (like `std::cout`)
- `vec`: the `Vector2D` object to be printed

# Appendix 9: Operator Overloading

Full sample code

```
#include <iostream>    // For std::cout and std::ostream

class Vector2D {
private:
    int X, Y;
public:
    Vector2D(double x, double y) : X(x), Y(y) {}

    Vector2D operator+(const Vector2D& other) const
    {
        return Vector2D(X + other.X, Y + other.Y);
    }

    // Friend function: overload << to print the vector
    friend std::ostream& operator<<(std::ostream& os, const Vector2D& vec);
};

// Define the non-member << operator for printing
std::ostream& operator<<(std::ostream& os, const Vector2D& vec)
{
    os << "(" << vec.X << ", " << vec.Y << ")";
    return os;
}

int main() {
    Vector2D p1(3.0, 4.0);
    Vector2D p2(1.0, 2.0);
    Vector2D result = p1 + p2;
    std::cout << "p1 + p2 = " << result << std::endl; // Output: p1 + p2 = (4.0, 6.0)
}
```