

# C++ Introduction

---

HANG SU

# Recommended

---

Website:

<http://www.cplusplus.com/doc/tutorial/>

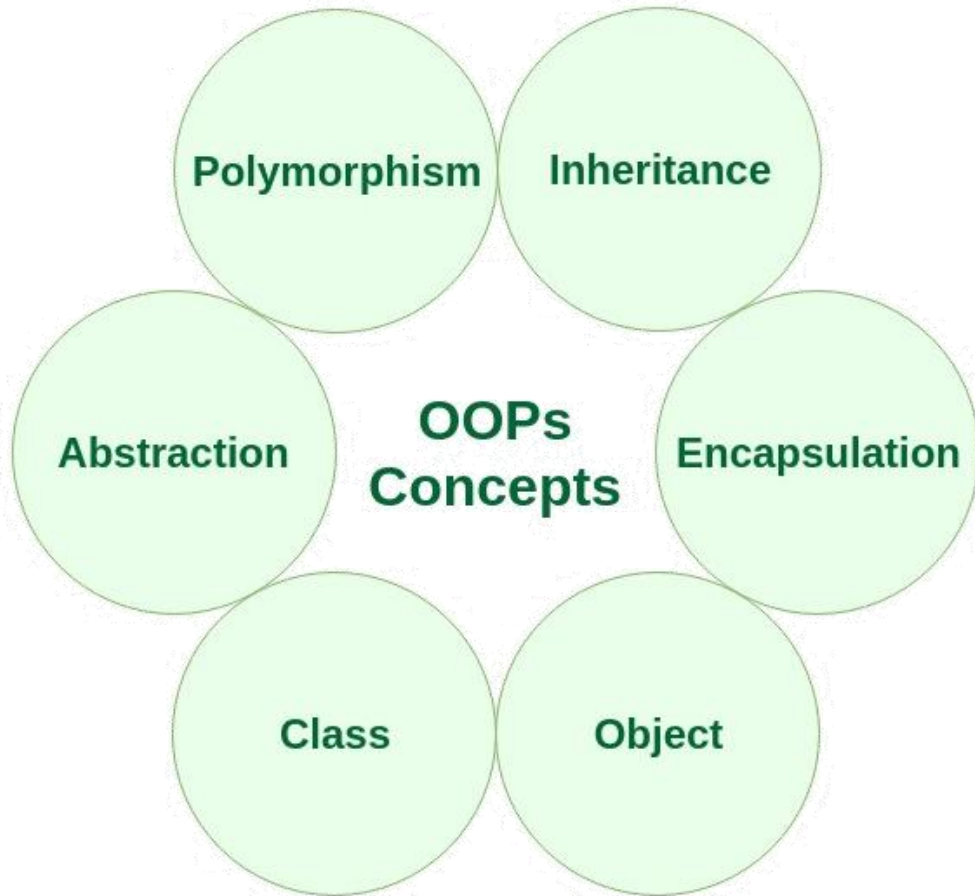
Books:

C++ Primer Fifth Edition

The C++ Programming Language, Fourth Edition

# From C to C++

---



**Abstraction:** Picking out features

**Class:** A blueprint for an entity with properties and functions

**Object:** An instance of a class

**Encapsulation:** A form of information hiding and abstraction

**Inheritance:** Inherit properties and functions from another class

**Polymorphism:** Process objects differently depending on their data type or class

# Class and Objects

---

```
1  class Square
2  {
3  public:
4      Square();
5      Square(int _side);
6      int GetPerimeter();
7      int GetArea();
8      void SetSide(int _side);
9  private:
10     int side;
11 };
```

Annotations:

- Line 1: `class Square` → Name of class
- Line 3: `public:` → Access specifier (public, private, protected)
- Lines 4-8: `Square();`, `Square(int _side);`, `int GetPerimeter();`, `int GetArea();`, `void SetSide(int _side);` → Member functions
- Line 10: `int side;` → Member variables

# Constructors

---

Special **member functions** used to construct new objects

Automatically called when an object is created

Implicit: `Square square;`

Explicit: `Square square(5);`

Initialize all data members

# Constructor Example

---

## Declaration

```
class Square
{
public:
    Square(int _side);
    int GetPerimeter();
    int GetArea();
private:
    int side;
};
```

## Implementation

```
Square::Square(int _side):
side(_side)
{}
```

# Copy Constructor

---

Declaration

```
class Square
{
public:
    Square();
    Square(int _side);
    Square(const Square &rhs);
    ...
private:
    ...
};
```

In main function

```
Square square1(5);
```

```
Square square2(square1);
```

```
Square square3 = square1;
```

} Copy  
Constructor

# Operators


---

## New

```
Square *square = new Square(5);
```

```
Square *squares = new Square[10];
```

The allocation  
occurs on the heap



## Delete

```
delete square;
```

```
delete []squares;
```

Only delete if memory was allocated by new

Don't use memory after deletion

Don't delete memory twice



# Destructor

---

```
~Square::Square()  
{  
    // Do Cleanups  
    ...  
}
```

One destructor for a class

Invokes when lifetime ends

If created by new operation,  
only invoked by delete

# C++ References

---

Valid types, just like pointers

Internally: just a pointer

Easier to manipulate  
No de-referencing needed

Safer  
Can only be initialized from  
a valid instance of an object

## Example

```
Square square1(5);  
Square &square2 = square1;  
square2.SetSide(6);  
std::cout<<square1.GetArea()<<std::endl;  
std::cout<<square2.GetArea()<<std::endl;
```

[Running]

36

36

# Inheritance (not covered in hw7)

---

Faster implementation time

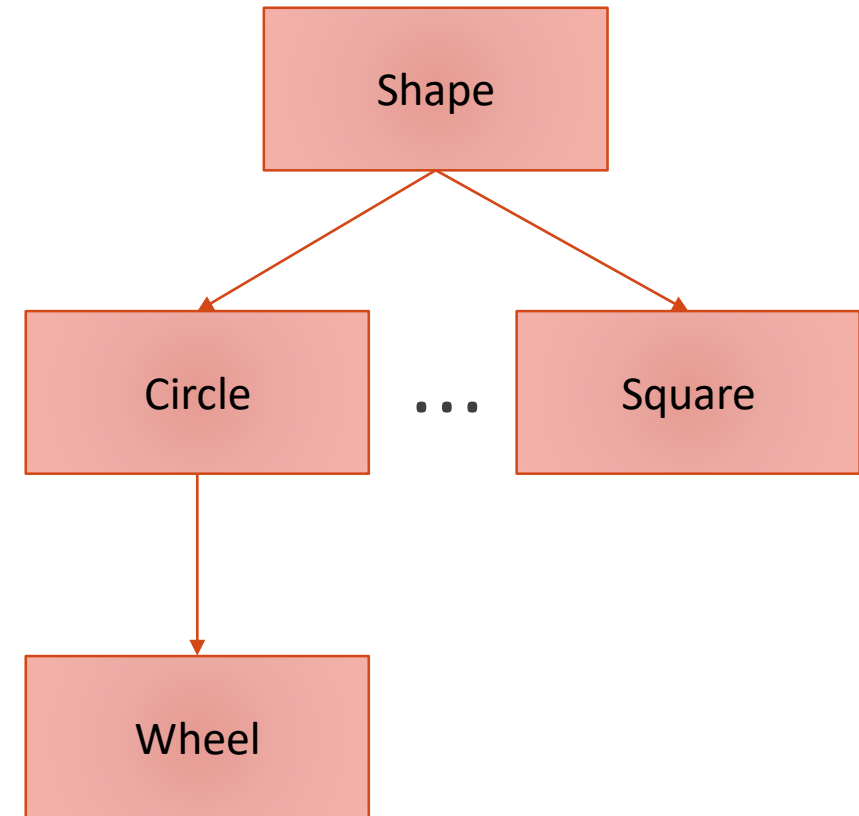
Fewer errors

Easier to maintain/update

....

```
class Shape
{
public:
    ...
private:
    ...
};

class Square: public Shape
{
    ...
};
```



# Polymorphism

---

Overloading

Templates

Overriding

Virtual functions

...

# Overloading

---

## Functions

```
class printData
{
public:
    void print(int i) {
        std::cout << i << std::endl;
    }

    void print(double f) {
        std::cout << f << std::endl;
    }

    void print(char c[]) {
        std::cout << c << std::endl;
    }
};
```

## Operators

```
Square operator+(const Square &rhs)
{
    Square square;
    square.side = this->side + rhs.side;
    return square;
}
```

```
Square square1(3);
Square square2(4);
Square square3 = square1 + square2;
```

# Templates

---

Use generic data type T

Replaced by concrete type at compile time

Enables “on-the-go” construction of a member of a family of functions and classes that perform the same operation on different data types

functions  function templates

classes  class templates

# Templates

---

## Function

```
template <class T, ...>
returntype function_name(arguments)
{
    // Body of function
}
```

## Class

```
template<class T1, class T2, ...>
class class_name
{
    ...
    // data items of template type
    T1 m_data1;
    // function of template argument
    void func1(T1 a, T2 &b);
    T1 func2(T2 *x, T2 *y);
    ...
};
```

# Iterators

---

Input

Output



Perform sequential single-pass input or output operations

Forward



Iterate through a range(forward)

Bidirectional



Also iterate through backwards

Random Access



Access ranges non-sequentially (standard pointers)



# Iterators

---

Const\_iterator

an iterator that points to a const element, while it can update itself (increment or decrement).

In practice, use const\_iterator whenever you can, use iterator if you have no other choice.

Const iterator

always points to the same element, but the element it points to don't have to be const

# Thanks

---