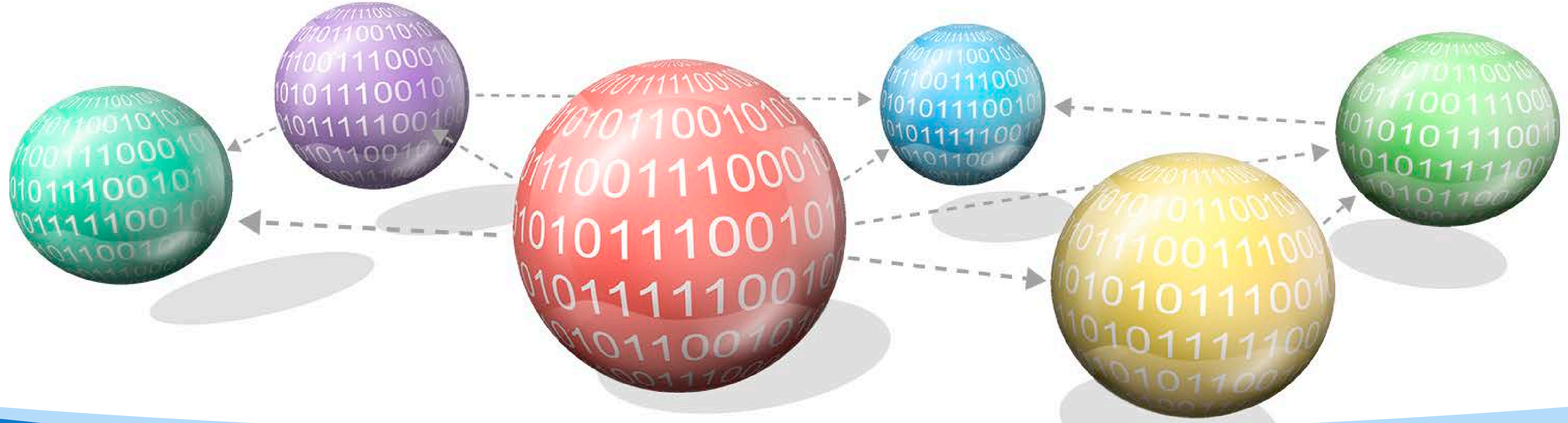


CE/CZ2002 Object-Oriented Design & Programming

Chapter 10: OO Concepts in C++

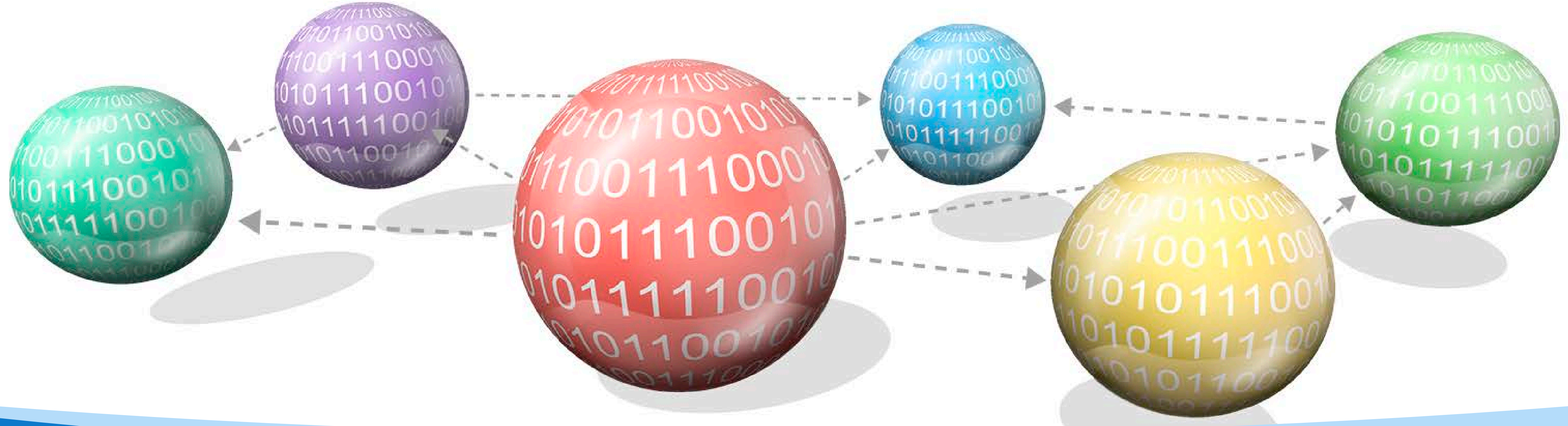
Mr Tan Kheng Leong
Lecturer, School of Computer Science and Engineering

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Objectives

By the end of this chapter, you should be able to:

- Explain the differences between C and C++
- Develop and build an OO application using C++
- Explain the differences between Java and C++



CE/CZ2002 Object-Oriented Design & Programming

Topic 1.1: C vs. C++ - C++ Fundamental

Chapter 10: OO Concepts in C++

Mr Tan Kheng Leong

Lecturer, School of Computer Science and Engineering

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

C and C++ : Siblings

- Historically, modern C and C++ are siblings – both are direct descendants of "Classic C" popularised by ***Kernighan and Ritchie***.
- C is primarily a subset of C++.
- ++ = **OO features**, and more…

Bjarne Stroustrup, creator of C++ (1979)

Retrieved December 12, 2016 from https://upload.wikimedia.org/wikipedia/commons/thumb/d/da/BjarneStroustrup.jpg/280px-BjarneStroustrup.jpg.

Year	**C++ Standard**	**Informal name**
2014	N3690 (working draft C++14)[15]	C++14

C++ Features not in C

Refer to video at: 01:10

- Classes and member functions
  - C use **struct** and global functions
- Derived classes and virtual functions
  - C use **struct**, global functions and pointers to fns
- Exceptions
  - C uses error code, error return values, etc.
- Function overloading
  - C give each function a distinct name
- **new**/**delete**
  - C use **malloc**(size)/**free**(..) and separate initialisation/cleanup
- **bool**
  - C use **int**
- **references & (addressOf)**
  - C use **pointers**
- .... **// /* */ - '//' added in C99**

C++ Keywords that are not in C

Refer to video at: 03:06

Keywords				
and	asm	bool	catch	class
delete	false	friend	namespace	new
not	operator	or	private	protected
public	template	this	throw	true
try	using	virtual	xor	……….

```
00000000                push    ebp
00000001                mov     ebp, esp
00000003                movzx   ecx, [ebp+arg_0]
00000007                pop     ebp
00000008                movzx   dx, cl
0000000C                lea     eax, [edx+edx]
0000000F                add     eax, edx
00000011                shl     eax, 2
00000014                add     eax, edx
00000016                shr     eax, 8
00000019                sub     cl, al
0000001B                shr     cl, 1
0000001D                add     al, cl
0000001F                shr     al, 5
00000022                movzx   eax, al
00000025                retn
```

C vs. C++: Simple Comparison

Refer to video at: 04:29

```c
#include <stdio.h>


int getFact(int n) {
  int c, fact = 1;
  if ( n < 0 )
       printf("Number should be non-negative.");
  else
       for (c = 1; c <= n; c++)
            fact = fact * c;
   return fact;
}

int main()
{
  int n = 1;

  printf("Enter a number to calculate it's factorial\n");
  scanf("%d", &n);

  printf("Factorial of %d = %d\n", n, getFact(n));

  return 0;
}
// save as <anyname>.c
```

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

int getFact(int n) {
  int c, fact = 1;
  if ( n < 0 )
std:: cout << "Number should be non-negative." << endl ;
  else
       for (c = 1; c <= n; c++)
            fact = fact * c;
   return fact;
}

int main()
{
  int n = 1;

  cout << "Enter a number to calculate it's factorial > ";
  cin >> n;

  cout << "Factorial of " << n << " = " << getFact(n) << endl;

  return 0;
}
// save as <anyname>.cpp
```

Interface / Implementation

Refer to video at: 06:06

- **Header files (.h)** is used for function declarations: declare the interfaces.
- **Source code (.cpp)** is used for defining class/function implementation.

```
// in myclass.h
class MyClass {
    private :
        int value ;
    public:
        void foo() ;
        int evaluate()  ;
};
```

Interface Declaration (header)

```
// in myclass.cpp
#include “myclass.h”

void MyClass::foo() {
   ….// implementation
}

 int MyClass:: evaluate() {
    …..
}
```

Implementation (implementation source code)

```
// in main.cpp
#include “myclass.h”

int main() {
     MyClass a;
     …….// usage
    return 0;
}
```

Using the Class

Interface / Implementation: Advantages of Header Files

Refer to video at:
07:46

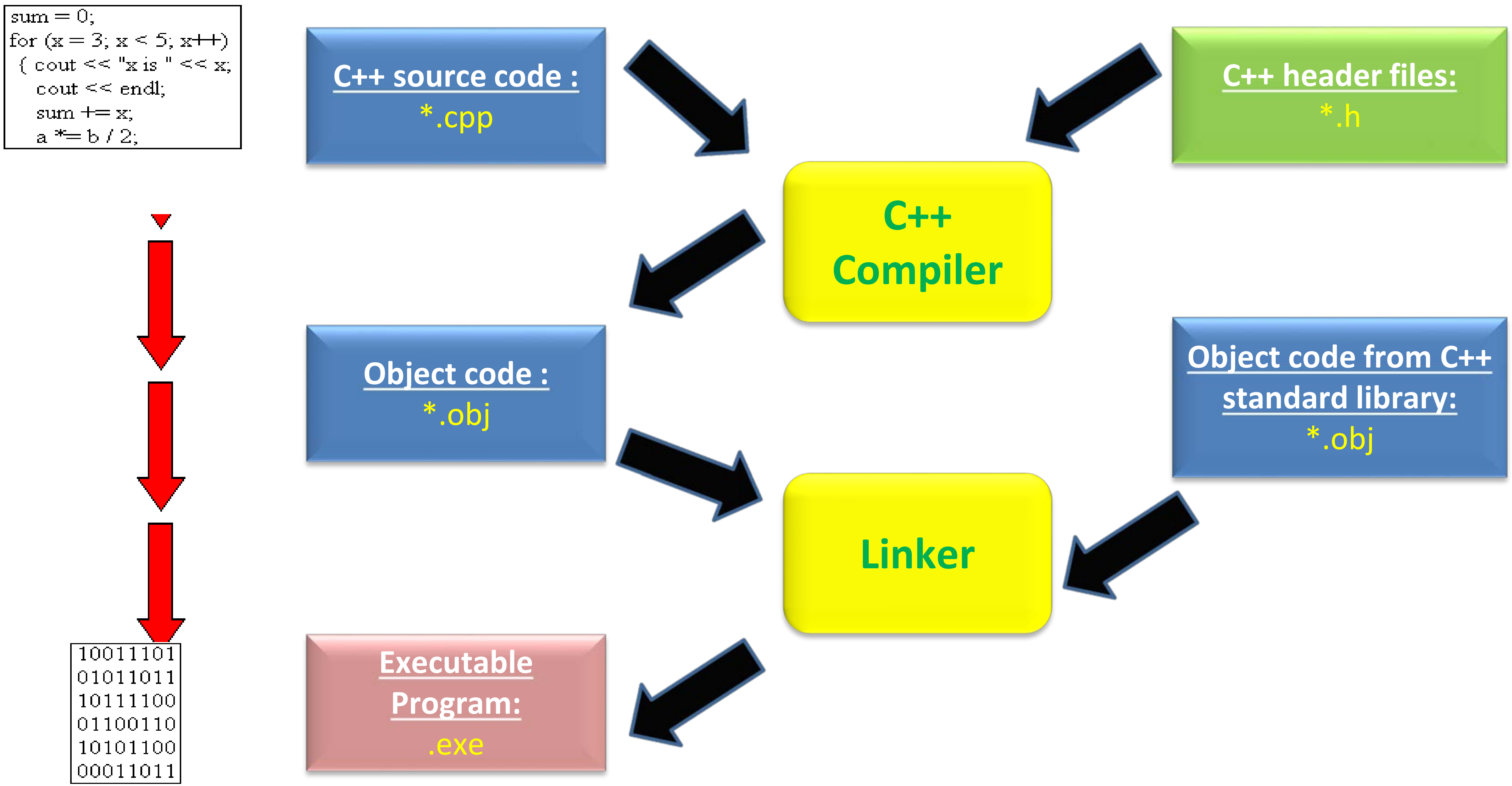
- It speeds up compile time. As your program grows, so does your code, and if everything is in a single file, then everything must be fully recompiled every time you make any little change.
- It keeps your code more organised. If you separate concepts into specific files, it's easier to find the code you are looking for when you want to make modifications (or just look at it to remember how to use it and/or how it works).
- It allows you to separate *interface* from *implementation*. Client classes only need the interface files (.h) and not implementation files (.cpp).

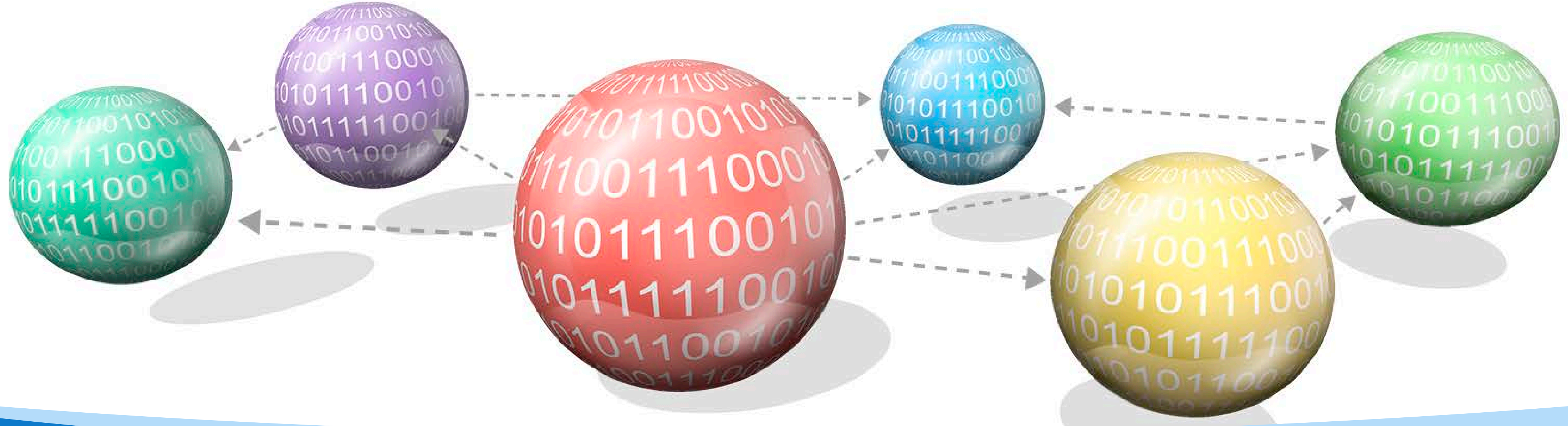
#include Guard

```
#ifndef HEADERFILE_H
#define _H
//...your header code here
#endif
```

C++ Translation

Refer to video at:
10:10





CE/CZ2002 Object-Oriented Design & Programming

Topic 1.2: C vs. C++ - C++ Structure

Chapter 10: OO Concepts in C++

Mr Tan Kheng Leong

Lecturer, School of Computer Science and Engineering

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

C++ Class Definition

Refer to video at: 00:07

```
class Point {
  private:
      int _x, _y;     // point coordinates,  default private
   public:
      void setX(const int val); // declaration
      void setY(const int val); // declaration
      int getX() { return _x; } // declaration + implementation
      int getY() { return _y; } // declaration + implementation
};
void Point::setX(const int val) { _x = val; }
void Point::setY(const int val) { _y = val;                 }
// scope operator  "::"
```

C++ Class Definition

```
class Point {
  private:
     int _x, _y;     // point coordinates,  default private
   public:
     void setX(const int val); // declaration
     void setY(const int val); // declaration
     int getX() { return _x; } // declaration + implementation
     int getY() { return _y; } // declaration + implementation
};
void Point::setX(const int val) { _x = val; }
void Point::setY(const int val) {  this->_y = val ;      }
// scope operator  "::"
```

Class Constructors

Refer to video at:
02:49

```
class Point {
    int _x, _y;    // point coordinates,  default private
  public:
    Point()   { _x = _y = 0; }
    Point(const int x, const int y)   { _x = x; _y = y; }
    …….
};
```

Initialisation list

```
Point() : _x(0), _y(0) {}
Point(const int x, const int y) : _x(x), _y(y) {}
```

Dynamic Initialisation

Class Constructors

```
class Point {
      int _x, _y;     // point coordinates,  default private
   public:
      Point() : _x(0), _y(0) {}
      Point(const int x, const int y) : _x(x), _y(y) {}

};
```

Class Destructors

Refer to video at: 04:14

```
class Point {
      int _x, _y;     // point coordinates,  default private
   public:
     Point() : _x(0), _y(0) {}
     Point(const int x, const int y) : _x(x), _y(y) {}

   ~Point() { /* do clean up */ }  // Destructor
   //prefixed by a tilde (~ ) of the defining class
     …….
};
```

Ensure that the allocated memory is released!

Object Creation

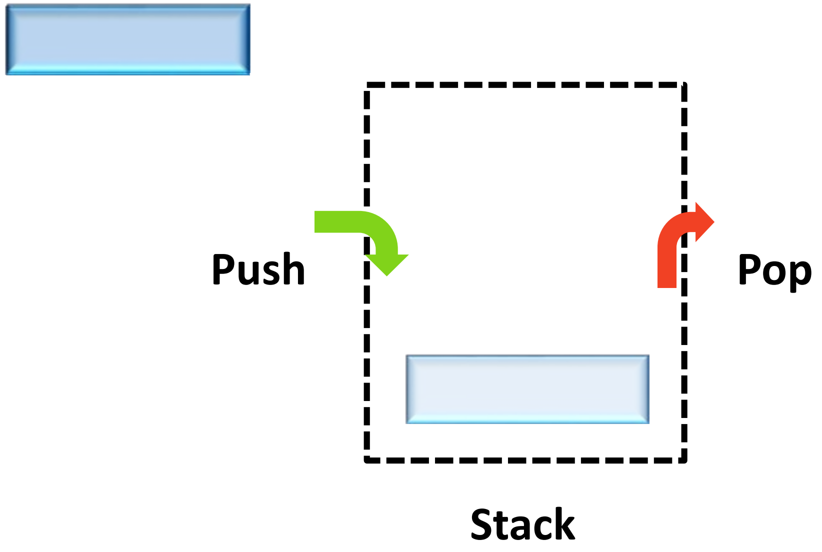
Refer to video at:
05:11

Method 1 (stack)

```
Point   aPoint;  // Point::Point()   Note : without ( )
Point   bPoint(12, 34); // Point::Point(const int, const int)
        aPoint.setY(20);
        int  y = aPoint.getY() ;
```

Pop

Stack	
bPoint	12
	34
aPoint	0
	20



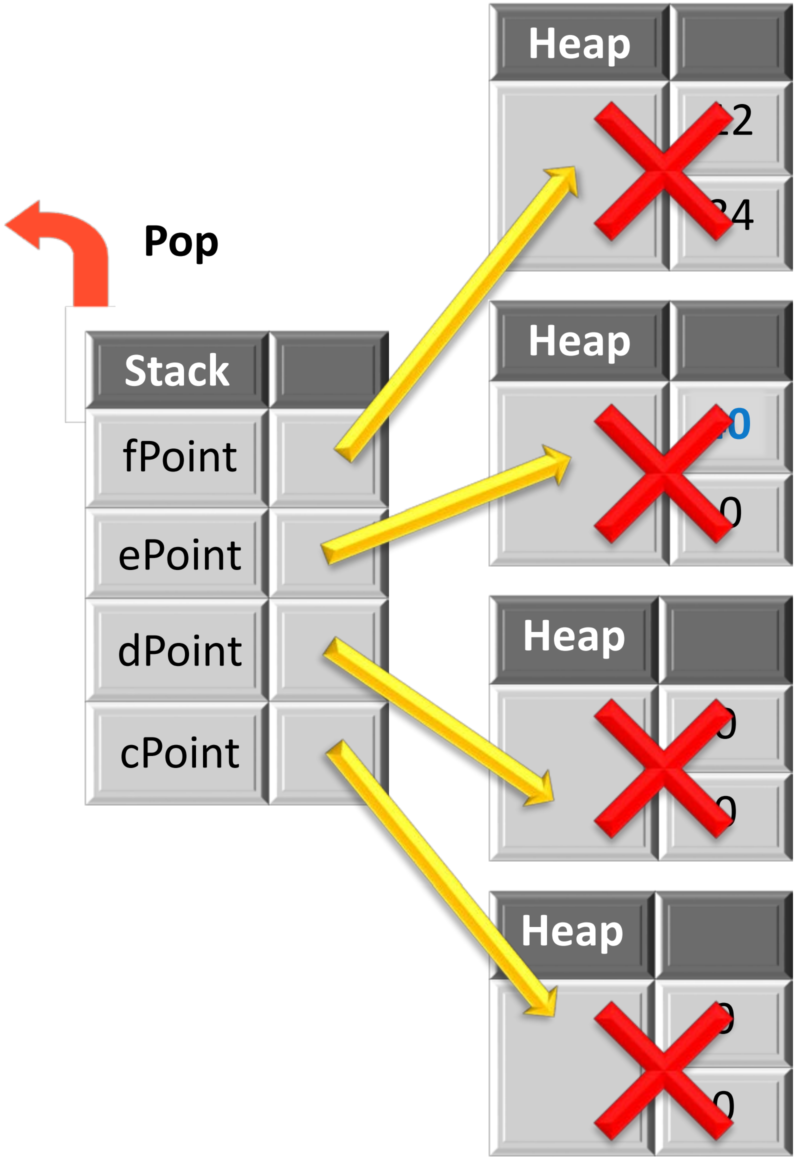
Object Creation

Refer to video at: 06:29

Method 2 (heap)

```
Point   *cPoint = new Point();  // Point::Point()
Point   *dPoint = new Point();  // Point::Point()
Point   *ePoint ;
     ePoint =  new Point();       // Point::Point()
Point   *fPoint = new Point(12, 34); // Point::Point(const int,
                                                     // const int)

ePoint->setX(40) ;
int x  = ePoint->getX() ;
delete  cPoint, dPoint, ePoint, fPoint ;
```



Class Inheritance

Refer to video at: 08:01

General Form :

class *derived-class-name* **:** **[**visibility-mode**]** *base-class-name*

{

....

};

visibility-mode : [optional] private / public

private : privately inherited (default)

‘public members’ of base class become

‘private members’ of derived class.

public : publicly inherited

‘public members’ of base class remain

‘public members’ of derived class.

Base

public : int data()

class Sub : **Base**

Sub

private : int data()

Class Inheritance

Refer to video at: 19:16

```
class Point3D : public Point {
    int _z;
public:
    Point3D() { setX(0); setY(0); _z = 0; }
    Point3D(const int x, const int y, const int z) {
        Point:: setX(x);   setY(y);   _z = z;
    }
    ~Point3D()        { /* Nothing to do */ }
    int getZ() { return _z; }
    void setZ(const int val) { _z = val; }
};
```

Class Inheritance

Using Base class constructor

```
class Point3D : public Point {
    int _z;
public:
    Point3D( const int x, const int y, const int z) : Point(x, y)
        { _z = z; }
.........
};
```

```
: Point(x, y) , _z(z)  { }
```

Specify the desired base constructors after a **single colon** just before the body of constructor.

Class Multiple Inheritance

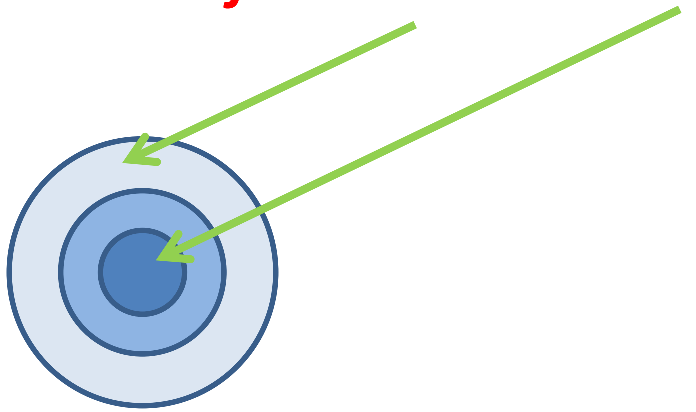
```
class DrawableString : public Point, public DrawableObject
{
.....
}
```

Class Inheritance

Refer to video at: 11:35

- ***Destruction***
- If an object is destroyed, for example by leaving its definition scope, the destructor of the corresponding class is invoked. If this class is derived from other classes their destructors are also called, leading to a recursive call chain.

Destructor called from child to ancestor

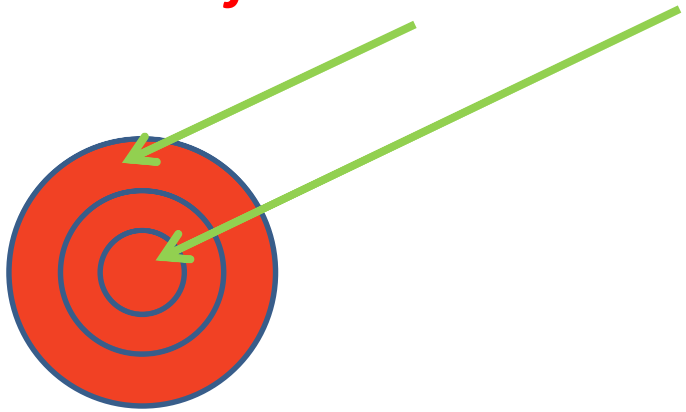


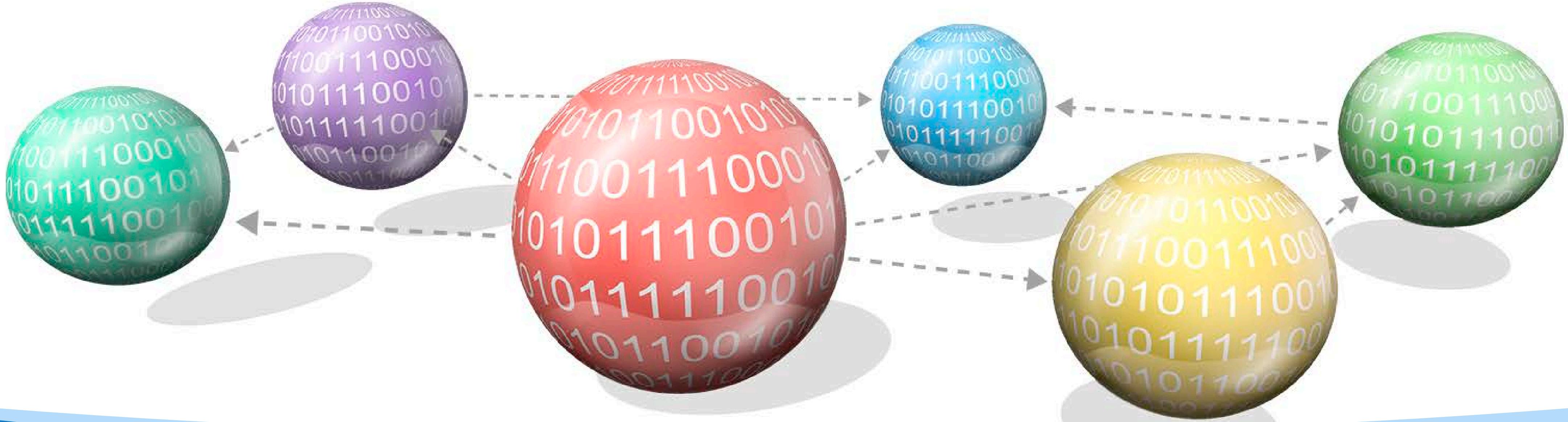
Class Inheritance

Refer to video at: 11:53

- ***Destruction***
- If an object is destroyed, for example by leaving its definition scope, the destructor of the corresponding class is invoked. If this class is derived from other classes their destructors are also called, leading to a recursive call chain.

Destructor called from child to ancestor





CE/CZ2002 Object-Oriented Design & Programming

Topic 1.3: C vs. C++
- C++ OO and Non-OO Features and Keywords

Chapter 10: OO Concepts in C++

Mr Tan Kheng Leong
Lecturer, School of Computer Science and Engineering

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Class Function Overloading

Refer to video at:
00:11

- **Just like in Java!**

```
int Add(int nX, int nY);   // integer version
double Add(double dX, double dY);   // floating point version
int Add(int nX, int nY, int nZ)

int GetRandomValue();
double GetRandomValue();
```

NOT Overloading

Class Function Overloading

Refer to video at: 00:30

- A **default parameter** is a function parameter that has a default value provided to it. Example,

```
void PrintValues(int nValue1, int nValue2=10)
{
    cout << "1st value: " << nValue1 << endl;
    cout << "2nd value: " << nValue2 << endl;
}
 int main()
{
    PrintValues(1); // nValue2 will use default parameter of 10
    PrintValues(3, 4); // override default value for nValue2
}
```

Class Function Overloading

- **Rules**:
  1) All default parameters must be the **rightmost** parameters. The following is not allowed:

     void PrintValue(int nValue1=10, int nValue2); // not allowed

  2) The **leftmost** default parameter should be the one most likely to be changed by the user.

     ~~PrintValues(, 3) ??~~

- **Constraints:**
  - Do NOT count towards the parameters that make the function unique. Example:

    PrintValues(3) ???

    void PrintValues(int nValue);

    NOT ALLOWED !!

    void PrintValues(int nValue1, ~~int nValue2=20~~);

From C to C++

Refer to video at:
02:10

- **Reference**
  - “alias” to ”real” variable or object
  - Ampersand (&) is used to define a reference
  - Cannot be NULL   eg, int &r ;
  - Example,

```
int ix;      /* ix is "real" variable */
int  &rx = ix;    /* rx is "alias" for ix */
ix = 1;    /* also rx == 1 */
rx = 2;   /* also ix == 2 */
int *p  = &ix ;  // addressOf ix assigned to pointer p.
int &q = *p ;   // dereference pointer p and assign to q as alias.
```

	addr	content
	q	
	p	
	rx	
rx	ix	1

```
int q = *p ;
```

- **Reference**
  - “alias” to ”real” variable or object
  - Ampersand (&) is used to define a reference
  - Cannot be NULL   eg, int &r ;
  - Example,

```
int ix;     /* ix is "real" variable */
int   &rx = ix;    /* rx is "alias" for ix */
ix = 1;    /* also rx == 1 */
rx = 2;   /* also ix == 2 */
int *p  = &ix ;  // addressOf ix assigned to pointer p.
int &q = *p ;   // dereference pointer p and assign to q as alias.
```

	addr	content
	q	
	p	
	rx	
rx	ix	2

```
int q = *p ;
```

From C to C++

- **Reference**
  - “alias” to ”real” variable or object
  - Ampersand (&) is used to define a reference
  - Cannot be NULL   eg, int &r ;
  - Example,

```
int ix;      /* ix is "real" variable */
int   &rx = ix;    /* rx is "alias" for ix */
ix = 1;    /* also rx == 1 */
rx = 2;   /* also ix == 2 */
int *p  = &ix ;  // addressOf ix assigned to pointer p.
int &q = *p ;   // dereference pointer p and assign to q as alias.
```

	addr	content
	q	2
	p	
	rx	
rx	ix	

```
int q = *p ;
```

Reference (&) vs. Pointer (*)

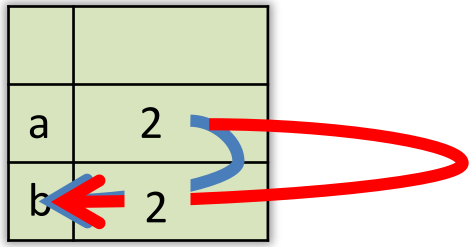
Refer to video at: 05:19

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

void addIt(int a) {  a  += a ; }
void doubleIt(int &a) {  a  *= 2 ; }
void tripleIt(int *a) {  *a  *= 3 ; }

int main() {
        int b = 2 ;
        cout << "b is now " << b << endl;
              addIt(b) ;
        cout << "after addIt(int a), b  is " << b << endl;
→             doubleIt(b) ;
        cout << "after doubleIt(int &a), b is " << b << endl;
               tripleIt(&b) ;
        cout << "after tripleIt(int *a), b is " << b << endl;
        // cin >> b;
}
```

OUTPUT:
b is now 2
after addIt(int a), b is 2
after doubleIt(int &a), b is 4
after tripleIt(int *a), b is 12



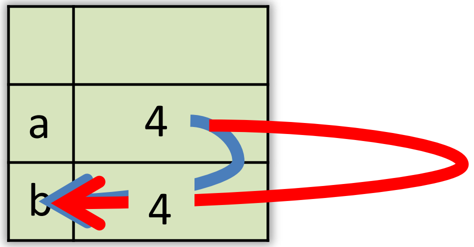
Reference (&) vs. Pointer (*)

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

void addIt(int a) {  a  += a ; }
void doubleIt(int &a) {  a  *= 2 ; }
void tripleIt(int *a) {  *a  *= 3 ; }

int main() {
        int b = 2 ;
        cout << "b is now " << b << endl;
              addIt(b) ;
        cout << "after addIt(int a), b  is " << b << endl;
              doubleIt(b) ;
        cout << "after doubleIt(int &a), b is " << b << endl;
               tripleIt(&b) ;
        cout << "after tripleIt(int *a), b is " << b << endl;
        // cin >> b;
}
```

OUTPUT:
b is now 2
after addIt(int a), b is 2
after doubleIt(int &a), b is 4
after tripleIt(int *a), b is 12



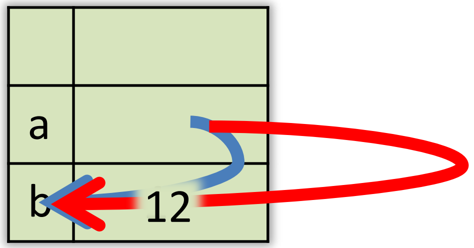
Reference (&) vs. Pointer (*)

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

void addIt(int a) {  a  += a ; }
void doubleIt(int &a) {  a  *= 2 ; }
void tripleIt(int *a) {  *a  *= 3 ; }

int main() {
        int b = 2 ;
        cout << "b is now " << b << endl;
              addIt(b) ;
        cout << "after addIt(int a), b  is " << b << endl;
              doubleIt(b) ;
        cout << "after doubleIt(int &a), b is " << b << endl;
              tripleIt(&b) ;
        cout << "after tripleIt(int *a), b is " << b << endl;
        // cin >> b;
}
```

OUTPUT:
b is now 2
after addIt(int a), b is 2
after doubleIt(int &a), b is 4
after tripleIt(int *a), b is 12



Polymorphism

Refer to video at: 08:26

- **Virtual**
  - To force method evaluation to be based on object type rather than reference type. [<ref type> <name> = new <obj type>(..)]
  - Without **virtual =>** non polymorphic (no dynamic binding)
  - **Example : virtual** void area() { cout << “…….." << endl ; }
  - **Virtual** function magic only operates on pointers(*) and references(&).
  - If a method is declared **virtual** in a class, it is **automatically virtual** in all **derived** classes.
- **Pure method => abstract** method (pure virtual)
  - By placing **"= 0"** in its declaration
  - **Example : virtual** void area() **= 0** ; // abstract method
  - **The class becomes an abstract** class

Example

Refer to video at: 11:30

```
#include <iostream>

using namespace std ;

class Shape {
    public :
    Shape()  { }
    virtual void area() { cout << "undefined" << endl ;}
    void name() { cout << " a shape" << endl ; }
};

class Rectangle : public Shape {
   private :
     int _length ;
     int _height ;
    public :
      Rectangle(int x, int y) : _length(x) , _height(y) { }
      void area() { cout << "area is " << _length * _height << endl ; }
      void name() { cout << " a Rectangle " << endl ; }
};
```

```
int main() {
   Rectangle rect(10,20);

  Shape *shapePtr = &rect ;
  Shape &shapeRef = rect ;
  Shape shapeVal = rect ;

  shapePtr->area() ;
                 shapeRef.area() ;
  shapeVal.area() ;
}
```

Output :
area is 200
area is 200
undefined

Example

```
#include <iostream>

using namespace std ;

class Shape {
    public :
    Shape()  { }
    ~~virtual~~ void area() { cout << "undefined" << endl ;}
    void name() { cout << " a shape" << endl ; }
};

class Rectangle : public Shape {
   private :
     int _length ;
     int _height ;
    public :
      Rectangle(int x, int y) : _length(x) , _height(y) { }
      void area() { cout << "area is " << _length * _height << endl ; }
      void name() { cout << " a Rectangle " << endl ; }
};
```

```
int main() {
   Rectangle rect(10,20);

  Shape *shapePtr = &rect ;
  Shape &shapeRef = rect ;
  Shape shapeVal = rect ;
  Shape shapeDeref = *shapePtr;
  shapePtr->area() ;
                  shapeRef.area() ;
  shapeVal.area() ;
  shapeDeref.area() ;
}
```

Output :

~~**area is 200**~~

~~**area is 200**~~

undefined

Example

Refer to video at: 14:48

```
#include <iostream>

using namespace std ;

class Shape {   // abstract class
    public :
    Shape()  { }
    virtual void area() = 0 ;   // pure method
    void name() { cout << " a shape" << endl ; }
};

class Rectangle : public Shape {
   private :
     int _length ;
     int _height ;
    public :
      Rectangle(int x, int y) : _length(x) , _height(y) { }
       void area() { cout << "area is " << _length * _height << endl ; }
       void name() { cout << " a Rectangle " << endl ; }
};
```

void showArea(~~Shape s~~) { .....}

```
int main() {
  Rectangle rect(10,20);

  Shape *shapePtr = &rect ;
  Shape &shapeRef = rect ;
  Shape shapeVal = rect ;

  shapePtr->area() ;
                shapeRef.area() ;
  shapeVal.area() ;
}
```

Example

```
#include <iostream>

using namespace std ;

class Shape {   // abstract class
    public :
    Shape()  { }
    virtual void area() = 0 ;   // pure method
    void name() { cout << " a shape" << endl ; }
};

class Rectangle : public Shape {
   private :
      int _length ;
      int _height ;
    public :
       Rectangle(int x, int y) : _length(x) , _height(y) { }
        void area() { cout << "area is " << _length * _height << endl ; }
        void name() { cout << " a Rectangle " << endl ; }
};
```

```
void showArea(Shape* s) { s->area();}
        showArea(&rect) ; // to call func
```

```
void showArea(Shape& s) { s.area() ;}
        showArea(rect) ;  // to call func
```

```
int main() {
   Rectangle rect(10,20);

  Shape *shapePtr = &rect ;
  Shape &shapeRef = rect ;
  Shape shapeVal = rect ;

  shapePtr->area() ;
                  shapeRef.area() ;
  shapeVal.area() ;
}
```

Polymorphism

Refer to video at: 16:31

- Safe down-cast
  - Use **dynamic_cast**
  - Type* t = **dynamic_cast**<Type*> (variable)
  - Returns NULL if the conversion was not possible
  - Only applicable to pointers
- If used very often, there is a good chance of your design being flawed

```
int main() {
   Shape *s = new Rectangle(10,20);
   Rectangle* rect1 = dynamic_cast<Rectangle*>(s) ;

   if( rect1 != NULL )  {
               cout << "valid  cast" << endl ;
  }
}
```

Output :

valid cast

About Arrays (1/3)

Refer to video at: 18:23

```
float  figure[3] ;
figure[0] = 0.79 ;  // first element is always index 0
figure[1] = 0.88;
figure[2] = 0.32 ; // last element, since size is 3


int  a[4] = { 1,2,3,4} ;
int  b[] = { 1,2,3,4,5} ;
int  c[5] = {1,2 } ; // less is ok, more is not
```

About Arrays (2/3)

```
int  *array1  = new int[4] ;
int   array2[5] ;

void func1(int [ ] a) {
        ........
} ;
void func2(int  *a) {
        ........
} ;
     func1(array2);
     func1(array1);
          func2(array2);
          func2(array1) ;
```

About Arrays (3/3): Array of Objects

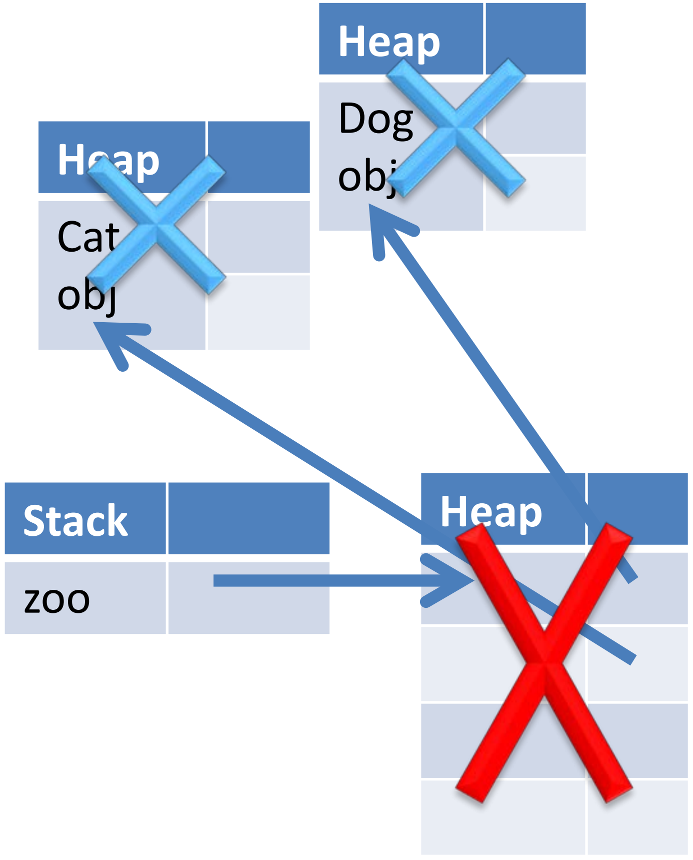
Refer to video at: 19:15

```
Cat *cats = new Cat[5];  // Cat is concrete class
delete [] cats ;
// if Mammal is abstract class
Mammal **zoo = new Mammal*[4] ;
zoo[0] = new Dog() ;
Or
```

```
Mammal *zoo[4] ;
zoo[0] = new Dog() ;
```

```
for(j=0;j<4;j++)
delete zoo[j];

delete [] zoo;
```



Operator Overloading

Refer to video at: 22:29

Write a **Complex Number** class in **Java** such that you can perform addition of 2 complex numbers and return a result (a new complex number). Test it by adding 3 complex numbers.

```
public class Complex {
    private double real, imag ;
    public Complex(double r, double i) { real = r ; imag = i ;}
    public Complex add(Complex c) {
      return new Complex(real + c.real, imag + c.imag);
    }
}
// usage :    Complex c1 = new Complex( 1.0, 2.0) ;
              Complex c2 = new Complex( 2.0, 3.0) ;
              Complex c3 = new Complex( 3.0, 4.0) ;
              Complex c4 = c1.add(c2).add(c3);
```

Operator Overloading

Refer to video at: 23:30

```
class Complex{ // in C++
    double _real, _imag;
    public:
      Complex() : _real(0.0), _imag(0.0) {}
      Complex(const double real, const double imag) : _real(real), _imag(imag) {}
      Complex add(const Complex op) { /* the usual */ }
      Complex mul(const Complex op);
      ...
    };
```

+	-	*	/

```
Complex  a(3,4), b(4,5), c ;
c=b.add(a);
// how about just  : c = b + a  ?;
```

Operator Overloading

+	-	*	/

```
class Complex{ // in C++
    double _real, _imag;
    public:
      Complex() : _real(0.0), _imag(0.0) {}
      Complex(const double real, const double imag) : _real(real), _imag(imag) {}
      Complex add(const Complex op) { /* the usual */ }
      Complex mul(const Complex op);
      ...
    };
    Complex operator +(const Complex op) {
      double real = _real + op._real, imag = _imag + op._imag;
      return(Complex(real, imag));
    }
```

```
Complex  a(3,4), b(4,5), c ;
c=b.add(a);
c = b + a  ;
```

Operator Overloading

Refer to video at: 24:46

+	-	*	/

```
class Complex {
    double _real, _imag;
    public:
      Complex() : _real(0.0), _imag(0.0) {}
      Complex(const double real, const double imag) : _real(real), _imag(imag) {}
      Complex operator +(const Complex op) {
          double real = _real + op._real, imag = _imag + op._imag;
          return(Complex(real, imag));
      }
      Complex operator *(const Complex op) {....}
      ...
    };
// evaluate  ((c1 + c2) * (c1 + c2)) * c3
// where c1, c2, c3 are complex numbers
```

Operator Overloading

Refer to video at: 25:21

- *Overload* almost all of its operators for newly created types (classes).

<<	|	+=	!	!=	==
<	>	<=	>=	++	-- ...

- In previous case, **operator +** is a **member** of class *Complex*. An expression of the form:

  c = a + b; where a, b, c are of *Complex* class is translated into a function call:

  c = a.operator+(b);

Complex a(3,4), b(4,5), c ;

c = a + b;

Operator Overloading

Refer to video at: 26:00

- **operator +** is **NOT** a member of class *Complex*

```
class Complex {
      double _real, _imag;
     public:
    Complex(const double real, const double imag):
            _real(real), _imag(imag) {}
      double real() { return _real; }
      double imag() { return _imag; }

};
// standalone function
Complex operator +(Complex op1, Complex op2) {
        double real = op1.real() + op2.real() ;
        double imag = op1.imag() + op2.imag();
    return(Complex(real, imag));
}
```

```
Complex operator +(const Complex op) {
        double real = _real + op._real ;
        double imag= _imag + op._imag;
        return(Complex(real, imag));
  }
```

Member	Non/member
1 operand **vs.**	2 operands
Unary **vs.**	Binary

e.g, a + = b **vs.** c = a + b

Friend

Refer to video at: 28:39

Friend allows non-member function access to private data of a class.

```
class Complex {
     double _real, _imag;  // private
    public:
          ....
    friend Complex operator +( const Complex , const Complex  );
};
 Complex operator +(const  Complex op1, const Complex op2) {
          double real = op1._real + op2._real ;
          double imag = op1._imag + op2._imag;
   return(Complex(real, imag));
}
```

- **Should not use friend unless necessary**
  - **Break the data hiding principle.**
  - **If used often it is a sign that it is time to restructure your inheritance.**

```
// friend class SomeClass;
```

From C to C++

Refer to video at: 30:51

- **const**
  - to declare particular aspects of a variable (or object) to be constant
  - **const** variable
  - Examples,

```
int i;                       // just an ordinary integer
int *ip;                     // uninitialised pointer to integer
int * const cp = &i;         // constant pointer to integer
const int ci = 7;            // constant integer
const int *cip;              // pointer to constant integer
const int * const cicp = &ci; // constant pointer to constant
                             // integer
```

x

const int

const int

x

int

const int

From C to C++

- **const**

```
class Point {
        int _x, _y;     // point coordinates,  default private
    public:
        void setX(const int val); // definition
        int getX() const ; // definition
};
```

- **Function parameter/s**

  Example,

```
void Point::setX(const int val) {
    val = 5 ;  // error!!    val is constant and not modifiable
}
```

- **Member Function** (read-only function)

  Example,

```
int Point::getY()  const {
    _x = 0 ; _ y= 5 ; // error!!    Member variables are not modifiable
    return _x;
}
```

Namespace

Refer to video at: 33:01

```
// in example.h
#include <iostream>
using namespace std ;

namespace root {
        namespace sub {
                class Test {
                        public :
                        void print() {
                            cout << "an example of namespace" << endl ;
                        }
                };
        }
}
```

```
#include example.h
using namespace root::sub ;
int main() {
        Test t ;
        t.print();
}
```

```
#include example.h
using namespace root ;
int main() {
        sub::Test t ;
        t.print();
}
```

```
#include example.h

int main() {
    root::sub::Test t ;
    t.print();
}
```

String Class

Refer to video at: 34:01

#include <string>

Function	Task
append()	Appends a part of string to another string.
at()	Obtains the character stored at a specified location.
compare()	Compares string against the invoking string.
empty()	Returns true if the string is empty; otherwise returns false.
erase()	Removes character as specified.
find()	Searches for the occurrence of a specified substring.
insert()	Inserts character at specified location.
length()	Gives number of elements in a string.
replace()	Replace specified characters with a given string.

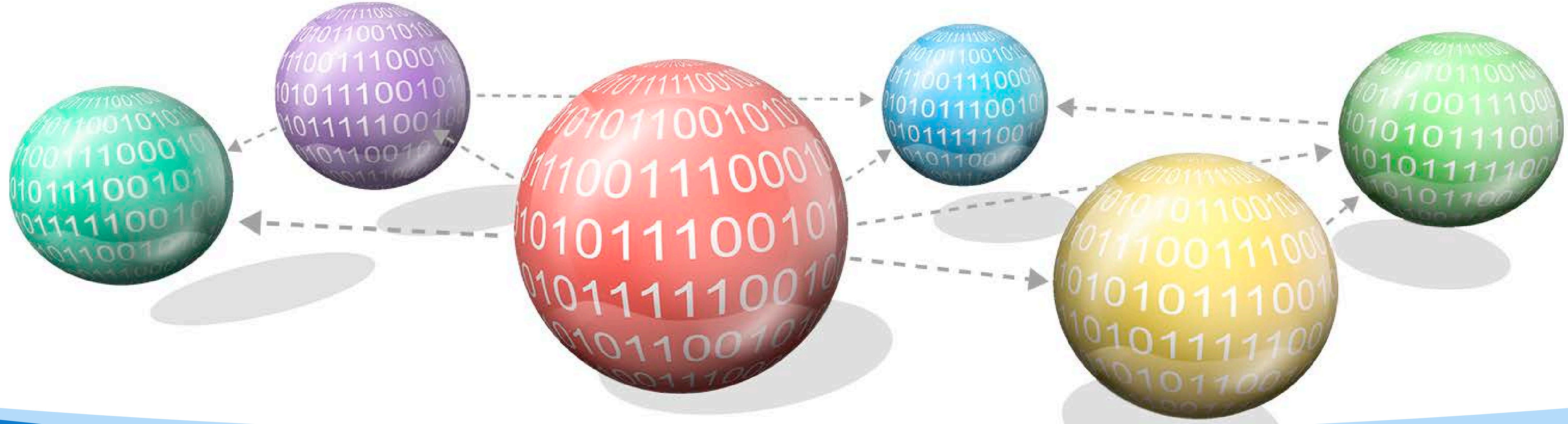
String Class

Manipulating string objects

```
#include <iostream>
#include <string>
using namespace std;
int main() {
   string s1(“12345”);
   string  s2(“abcde”) ;
   string  s3  = “abc” + s2 ;

   for (int i=0; i <s.length() ; i++)
       cout << s.at(i) ;
   for (int j=0; j <s.length() ; j++)
       cout << s[ j ] ;

     s1.insert(4,s2);  //  insert a string s2  into s1 at positon 4  : s1 = 1234abcde5
     s1.erase(4,5) ;  // remove 5 characters from s1 starting positon 4 : s1 = 12345
     s2.replace(1,3,s1) ;  // replace 3 characters in s2 with s1 starting position 4:
                           // s2 = a12345e
}.
```



CE/CZ2002 Object-Oriented Design & Programming

Topic 2: Java vs. C++

Chapter 10: OO Concepts in C++

Mr Tan Kheng Leong
Lecturer, School of Computer Science and Engineering

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Java Comparison to C++ (OO aspects)

- **Everything in Java must be in a class**. There are no global functions or global data.
- **There's no scope resolution operator :: in Java**. Java uses the dot for everything (including package). package (Java) vs. namespace (C++).
- **In Java, all objects of non-primitive types can be created only via new**. There's no equivalent to creating non-primitive objects "on the stack" as in C++.
- **In Java, Object handle (references) defined as class members are automatically initialised to null.** Initialisation of primitive class data members is guaranteed in Java.
- **There are no Java pointers in the sense of C and C++.** When you create an object with new, you get back a reference.
- **There are no destructors in Java**.
- **Java uses a singly-rooted hierarchy** inherited from the root class Object. In C++ you can start a new inheritance tree anywhere, so you end up with a forest of trees.

Java Comparison to C++ (OO aspects)

- **Inheritance in Java has the same effect as in C++, but the syntax is different.** (however, the super keyword in Java allows you to access methods only in the parent class, one level up in the hierarchy. Base-class scoping (::) in C++ allows you to access methods that are deeper in the hierarchy).
- **Java provides the interface keyword** which creates the equivalent of an abstract base class filled with abstract methods and with no data members.
- **There's no virtual keyword in Java** because all non-static methods always use dynamic binding. In Java, the programmer doesn't have to decide whether or not to use dynamic binding. The reason virtual exists in C++ is so you can leave it off for a slight increase in efficiency when you're tuning for performance (or, put another way, "if you don't use it you don't pay for it").
- **Java doesn't provide Multiple Inheritance (MI).**
- **There is method overloading, but no operator overloading in Java.**

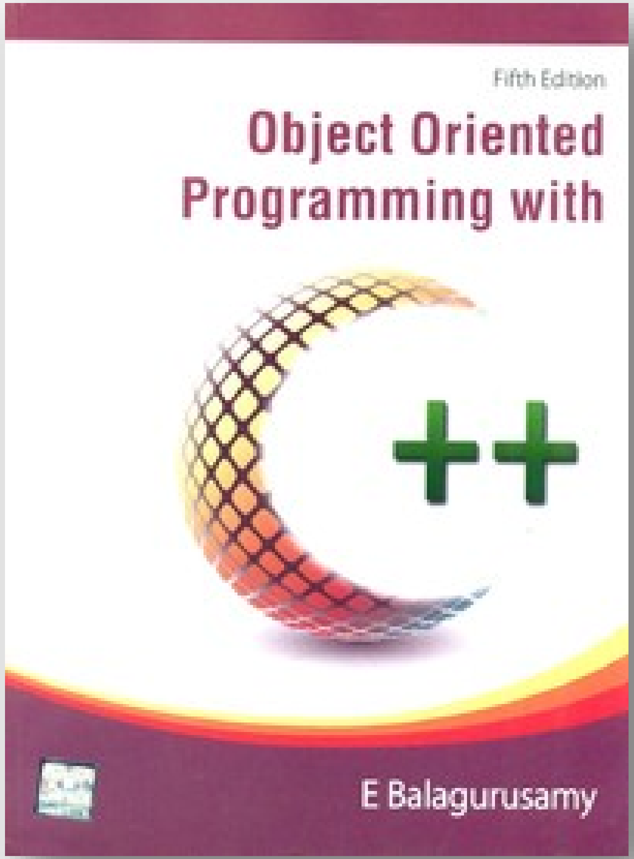
Java compare to C++ (other aspects)

Refer to "[C++ Versus Java.pdf]"

Key Points

- C is primarily a subset of C++.
- **Header files** help in speeding up compile time, keep code more organised, and allow us to separate interface from implementation.
- C++ allows object to be created in the **stack** without the use of '**new**'.
- C++ requires object to be cleaned after use to **avoid memory leak**. As a rule, for every '**new**' for creating object, there should be a corresponding '**delete**' to clean up the object.
- C++ has a class **destructor** which is called when **delete** is used.
- C++ uses **dynamic initialisation** to initialise class attributes and base object.
- C++ allows **Multiple Inheritance**.
- C++ uses class scope resolution operator **::** to identify explicitly the class to use.
- C++ provides **reference (&)** as an alternative to using **pointer (*)**.
- C++ allows dynamic binding only for **virtual** function.
- C++ allows default parameters in the function parameters.
- C++ allows **operator overloading** using the **operator keyword**.

Reference



Object Oriented Programming With C++ (5th Edition)

Author: E Balagurusamy