Lecture 10

C++ Programming

Arne Kutzner
Hanyang University / Seoul Korea

Inheritance

Inheritance

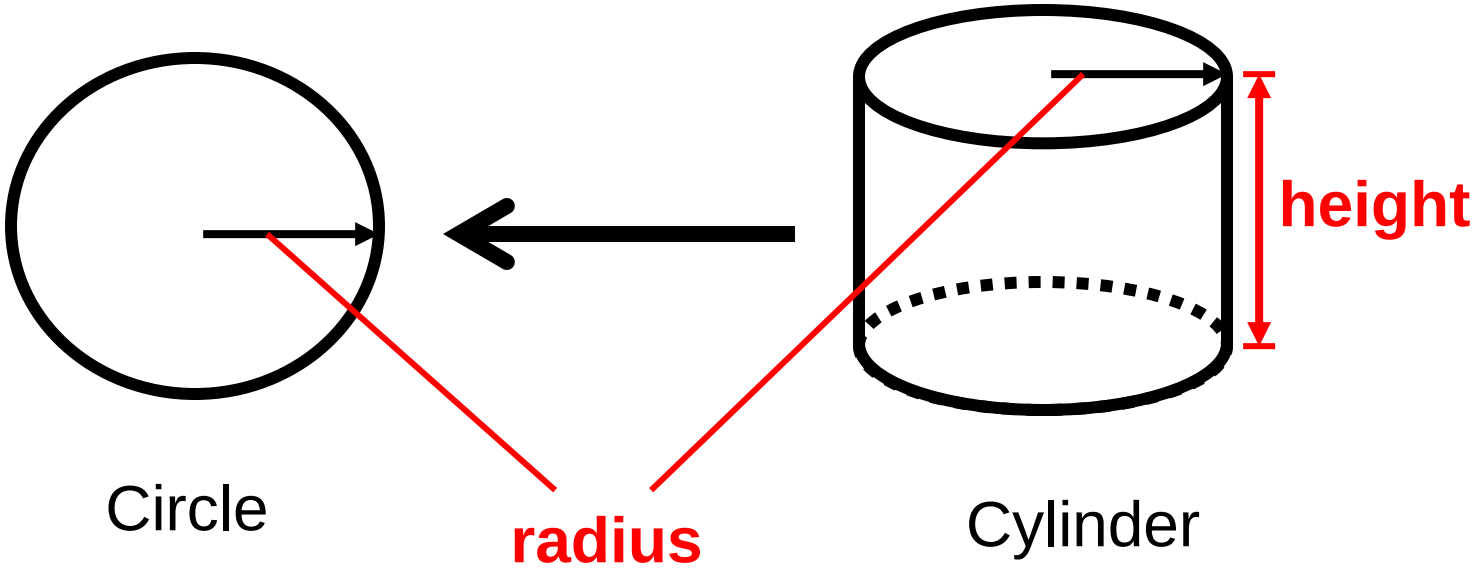
- You can create a new class that *inherits* from an existing class.
- The new class is a *specialization* of the existing class.
  - You can add new members and methods.
  - You can replace methods.
- Motivation: Reuse of existing code

Inheritance

- Example:
  - You have a class that represents “circles".
  - Create a new class that represents “cylinders".
  - Most of the behavior and attributes are the same, but a few are different/new (height of cylinder, different area computation, volume

Inheritance Example

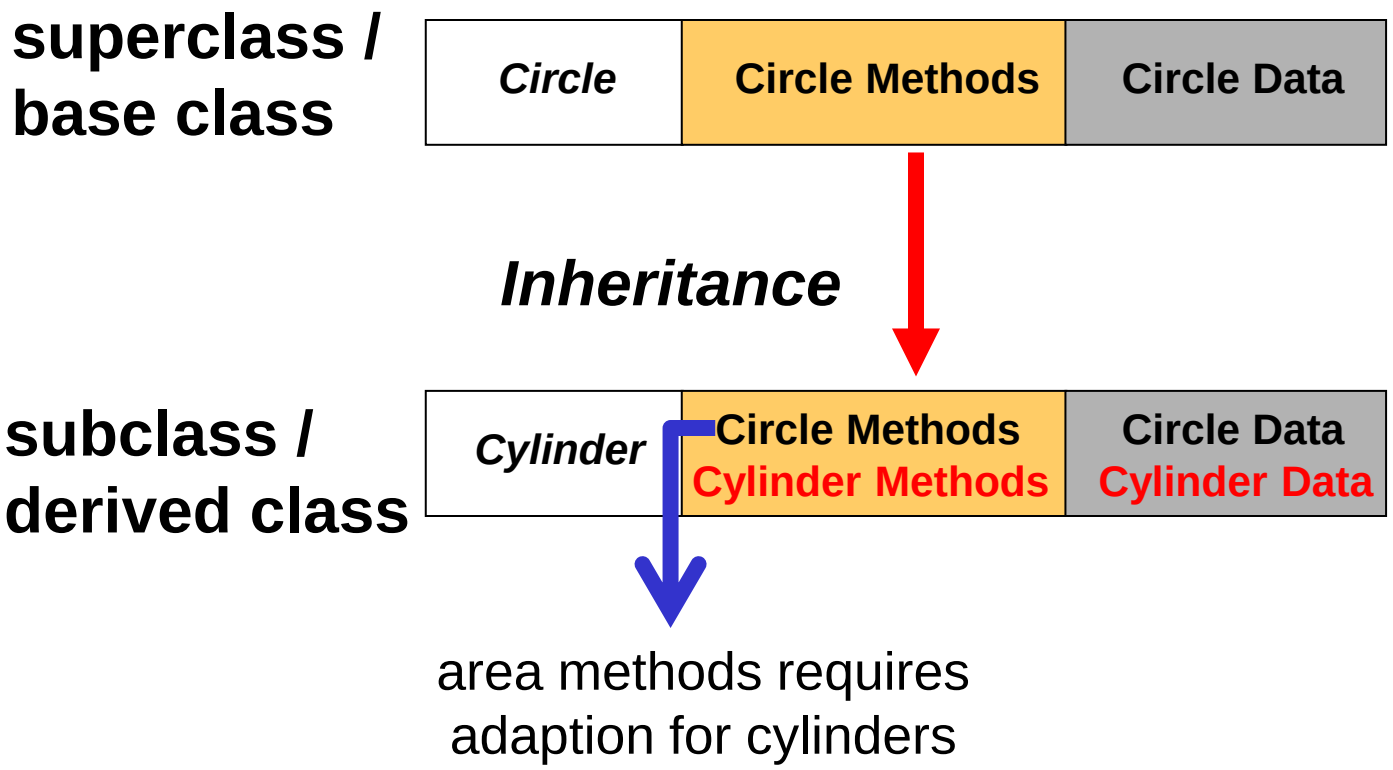
- A Cylinder is a extended (specialized) form of a Circle.



Terminology

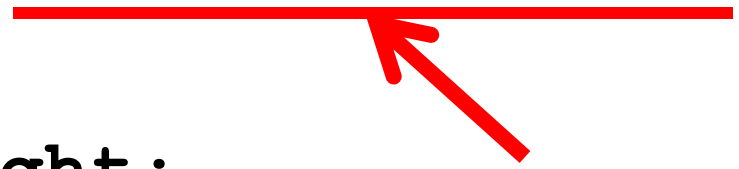
- The extended class is called the ***base class*** or ***superclass***.
- The newly created class is called the ***derived class*** *or* ***subclass****.*
- Objects of the derived class are also objects of the base class.
  (Idea behind the concept of Polymorphism)

Circle Example cont.



C++ Code

```
class Cylinder : public Circle {
  private :
     double height;
  public :
     double volume();
     Cylinder(double r, double h);
};
```



Here we indicate that Cylinder is a subclass of Circle

Method overriding

- We must adapt our area computation so that it fits cylinders:

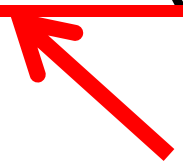
```
double Cylinder::area()
{
   return
      (Circle::area()) * 2
   + (2 * radius * 3.14159)
       * height;
}
```

Here we call the area computation of the super class

Problem: Call of superclass constructors in subclass constructors

- How to call a superclass constructor in a subclass constructor?
  Solution:

```
Cylinder::Cylinder(double r,
  double h) : Circle(r) {
  height = h;
}
```



Here we call a constructor from the Circle class

Protected Class attributes/methods

- If an attribute or method is protected, then derived classes have access to attributes and methods; otherwise they are like private.
- Example (Circle with protected radius):

```
class Circle {
protected :
    double radius;
public :
    double area();
};
```

Accessibility details for protected attributes/methods

class C1

protected:
int x;

C1 c1;

x = 2; **OK**
c1.x = 2; **OK**

sublass

class C2 : public C1

C1 c1;
C2 c2;

x = 2; **OK**
c1.x = 2; **!FAIL!**
c2.x = 2; **OK**

class C3

C1 c1;
C2 c2;

x = 2; **FAIL**
c1.x = 2; **FAIL**
c2.x = 2; **FAIL**

unrelated
to C1 and C2

Polymorphism, Dynamic Binding and Heap Objects

Object on the heap / Dynamic memory allocation

- Objects are allocated on the heap using the operator **`new`**
  Example:
  ```
  Circle* c = new Circle(9);
  ```
  The new-operator delivers a pointer (like **`malloc`** in C)
- Object allocated with new are removed from heap using the operator **`delete`**
  Example:
  ```
  delete c;
  ```
- For the dynamic allocation of arrays exists the special operator-pair **`new[]`**, **`delete[]`**.
- (C++ does not know automatic garbage collection for heap object as e.g. Java or C#)

Polymorphism

- An object of a subclass can be used by any code designed to work with an object of its superclass. Example:

```
void main() {
  Cylinder* cyl = new Cylinder(10 ,5);
  Circle* cir = new Circle(9);

  Circle* cir1 = cyl;

  std::cout << cir1 -> area();
}
```

Problem:
What area() implementation shall we use here?

Here we use Polymorphism

Virtual Functions

- A virtual function is a method in some base class that, if it is overridden in some subclass, will use dynamic binding.
- Virtual function are defined by using the keyword **`virtual`**.Example:

```
class Circle {
    public :
        double radius;
        Circle();
        virtual double area();
};
```

Indicates the wish for dynamic binding in the context of `area()`

Dynamic Binding

- It is decided during runtime which implementation has to be used.

Example:

```
void main() {
  Cylinder* cyl = new Cylinder(10 ,5);
  Circle* cir = new Circle(9);

  Circle* cir1 = cyl;

  std::cout << cir1 -> area();
}
```

Because of dynamic binding we use the Cylinder implementation of `area()`

Static class attributes

Static Data Members

- It is possible to have a single variable that is shared by all *instances* of a class (all the objects).
  - declare the variable as **`static`**.
- Data members that are static must be declared and initialize outside of the class.
  - at global or file scope.

Static data member example

```
class foo {
private:
  static int cnt;
public:
   foo() {
      cnt++;
      cout << "there are now " << cnt
           << " foo objects << endl;
  };
int foo::cnt = 1;
```

Declaration

Definition

See the file "static.cpp" for a complete example

Static Methods

- A static method is a class method that can be called without having an object.
- Static Methods can't access non-static data members! (they don't exist unless we have an object).
- Must use the :: operator to identify the method.

See the file "static.cpp" for a complete example

Static attribute declaration and definition

foo.h

```
class foo{

static int cnt;

…

};
```

Declaration

foo.cpp

```
#include "foo.h"

int foo::cnt=1;
```

Definition

f1.cpp

```
#include "foo.h"

...
```

f2.cpp

```
#include "foo.h"

...

int main() {

...
```

Friend declarations

Friends

- A Class can declare **other classes** as "friend classes".
- A Class can declare **external functions** as "friends".
- friends get access to private attributes and methods.

Friend Declaration

```
class foo {
private:
  int i,j;
…

friend class fee;
friend int printfoo( foo &f1);
};
```

this-Pointer, Classes and Files

`this`-pointer

- **`this`** returns always a reference to the current object. Example:

```
Circle::Circle(double radius){
   this -> radius = radius;
}
```

radius of the current (new) object

constructor's argument

Classes and Files

- The relationship between C++ class definitions and files depends on the coding guidelines used for programming.
- Most people do this:
  - class definition is in classname.h
  - any methods defined outside of the class definition are in classname.cpp

Operator Overloading

Introduction

- Operators can be overloaded like functions.
  - Background: I/O operators like “>>” and “<<“ get their implementation via operator overloading
- Operator overloading works with
  - **functions** as well as
  - **class-methods**

Operator overloading for “standard” functions

- Like a normal function definition but with operator-binding syntax with the function name. E.g. + operator:

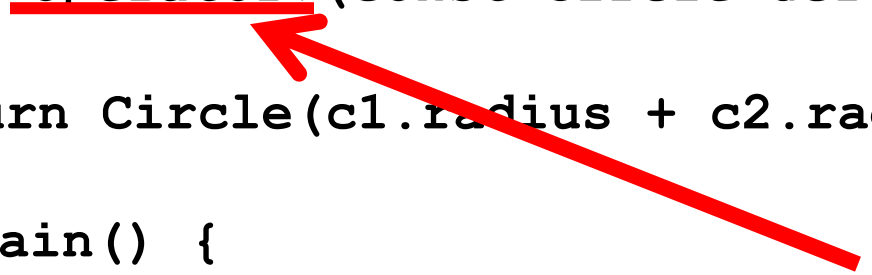
```
<datatype> operator+  (double a,
classA b) {
…
}
```

Addition for two circle objects via function overloading

- Overloading of the + operator for two Circle class objects

```
Circle operator+(const Circle &c1, const Circle &c2)
{
   return Circle(c1.radius + c2.radius);
}
void main() {
   Circle c1 = Circle(5.0);
   Circle c2 = Circle(3.0);

   std::cout << (c1 + c2).get_radius() << std::endl;
}
```

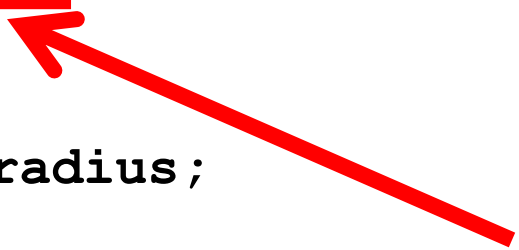


overloading of the + operator

Operator overloading via member functions

- Operators can be overloaded within a class definition. (Method overloading)
- Example – overloading of the + operator:

```
Circle Circle::operator+ (const Circle &other) const
{
    Circle c = *this;
    c.radius += other.radius;
    return c;
}
```



overloading of the + operator (here method overloading)

When to use what form of operator overloading?

- In the case of method overloading C++ translates expressions like
  <var> <op> <value>
  into
  ***<var>*.operator<*op*>(<*value*>**)
- But, if the first type is not a class-type, this translation is not possible. So, in this case we have to do it via operator overloading with classical functions.

Which operators can be overloaded?

- Unary operators:

```
+   -   *   &   ~   !   ++   --   ->
```

- Binary operators:

```
+   -   *   /   %   ^   &   |   <<   >>
= += -= *= /= %= ^= &= |=
<<= >>=
<   <=   >   >=   ==   !=   &&   ||
,   []   ()
new   new[]   delete   delete[]
```

Operator overloading and stream based I/O

- Overloading of the operators **<<** and **>>**
- Attribute access related problems can be solved using a **`friend`** declaration

  See file **`circle_with_operator_overloading.cpp`**
  - radius is private, so we have to use two friend declarations

Operator Overloading Restrictions

- You can't:
  - define a completely new operator
  - redefine the meaning of operators on built-in data types (like **`int`** or **`double`**).
  - Change operator precedence or associativity.