CS 6015: Software Practice

Spring 2024

Lecture 2: Classes and interface-like classes

Forward declaration

- Tells the compiler that there is a declaration of an entity before defining that entity. Used with:
  - Functions
  - User-defined types

```
#include <iostream>

int main() {
  int array[2] = {1, 2};
  int val = sum(2); --> Does this code work?
  std::cout << "Result: " << val <<
"\n";
}

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
  s += array[i];
  return s;
}
```

Forward declaration

- Tells the compiler that there is a declaration of an entity before defining that entity. Used with:
  - Functions
  - User-defined types

```
#include <iostream>

int sum(int);  //forward declaration
int main() {
  int array[2] = {1, 2};
  int val = sum(2);
  std::cout << "Result: " << val << "\n";
}

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
  s += array[i];
  return s;
}
```

-> declared here if no header files are used

C++ vs Java

C++ vs Java

- Object oriented programming languages
- Syntax
- Pointers
- Overloading
- Garbage collector

Classes in C++ vs classes in Java

- Create objects using pointers:

```
Posn *Obj1 = new Posn(2,3);
Use -> not . to access member variables and methods
```

- Use **`virtual`** keyword before member functions that need to be overridden in sub classes
- No interfaces in c++, we will use abstract classes
- C++ supports multiple inheritance
- Java uses ”garbage collection” to free unreachable objects

Simple Class

```
class Posn {
public:
   int x;
   int y;

   Posn(int x, int y) {
      this->x = x;
      this->y = y;
   }

   bool close_to_origin(int d) {
      return (abs(x) < d) && (abs(y) < d);
   }
};
```

Separating Declaration and Implementation

```
class Posn {
public:
  int x;
  int y;

  Posn(int x, int y);
  virtual bool close_to_origin(int d);
};


Posn::Posn(int x, int y) {
  this->x = x;
  this->y = y;
}

bool Posn::close_to_origin(int d) {
  return (abs(x) < d) && (abs(y) < d);
}
```

Separating Declaration and Implementation

```
class Posn {
public:
  int x;
  int y;

  Posn(int x, int y);
  virtual bool close_to_origin(int d);
};

Posn::Posn(int x, int y) {
  this->x = x;
  this->y = y;
}

bool Posn::close_to_origin(int d) {
  return (abs(x) < d) && (abs(y) < d);
}
```

connects to the **Posn** declaration

Separating Declaration and Implementation

posn.h

```
class Posn {
public:
  int x;
  int y;

  Posn(int x, int y);
  bool close_to_origin(int d);
};
```

posn.cpp

```
#include "posn.h"
Posn::Posn(int x, int y) {
  this->x = x;
  this->y = y;
}

bool Posn::close_to_origin(int d) {
  return (abs(x) < d) && (abs(y) < d);
}
```

Avoid **`#include "....h"`**
in **`.h /.hpp`** files

Recall Person / Employee example

Inheritance – Virtual method

main.cpp

```
#include "shape.hpp"

int main (int argc, char * argv[]) {

    Rectangle r1 (2,4);
    Circle c1 (5);

    printArea(r1);
    printArea(c1);
}
```

shape.hpp

```
class Shape {
public:
    virtual float getarea()=0;
};

class Circle: public Shape {
public:
    float radius;
    Circle(float radius);
    float getarea();
};

class Rectangle: public Shape {
    public:
        float width;
        float height;
        Rectangle(float width, float height);
        float getarea() ;
};

void printArea(Shape& shape);
```

shape.cpp

```
#include "shape.hpp"
#include <iostream>
using namespace std;


Rectangle::Rectangle(float width, float
height):width(width),height(height) {}


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


float Rectangle::getarea() {
    return width * height ;
}


float Circle::getarea() {
    return 3.14 * radius * radius ;
}


void printArea(Shape& shape) {
    cout<< "Area: " << shape.getarea() <<endl;
}
```

Inheritance – Virtual method

shape.hpp

```
//Shape: abstract class
class Shape {
public:
    //No implementation: pure virtual function
    virtual float getarea()=0;
};


// : equivalent to extends in java
class Circle: public Shape {
public:
    float radius;
    Circle(float radius);
    float getarea();
};
class Rectangle: public Shape {
    public:
        float width;
        float height;
        Rectangle(float width, float height);
        float getarea() ;
};

void printArea(Shape* shape);
```

main.cpp

```
#include <iostream>
#include "shape.hpp"

bool coinFlip();

int main (int argc, char * argv[]) {

    //importance of inheritance if we do not
know the type until runtime
    Shape *s;

    if(coinFlip()){
        s = new Rectangle(3,6);
    }
    else {
        s = new Circle(10);
    }

    std::cout<<"The aread with coinFlip is:
"<<s->getarea()<<std::endl;
}

bool coinFlip(){
    int x;
    std::cin>>x;
    if (x==0) {
        return true;
    } else {
        return false;
    }
}
```

Project: Related

Implementing Equality

```
class Animal {
};
```

```
class Tiger : public Animal {
  std::string color;
  int stripe_count;

  }
};
```

```
class Snake : public Animal {
  std::string color;
  int weight;
  std::string food;

  }
};
```

How to compare 2 tigers, 2 snakes?

Implementing Equality

```
class Animal {
  virtual bool equals(Animal *o) = 0;
};
```

```
class Tiger : public Animal {
  std::string color;
  int stripe_count;

};
```

```
class Snake : public Animal {
  std::string color;
  int weight;
  std::string food;

};
```

Add the method in the super class

Implementing Equality

```
class Animal {
  virtual bool equals(Animal *o) = 0;
};
```

```
class Tiger : public Animal {
  std::string color;
  int stripe_count;

  bool equals(Animal *a) {
    return ...;
  }
};
```

```
class Snake : public Animal {
  std::string color;
  int weight;
  std::string food;

  bool equals(Animal *a) {
    return ...;
  }
};
```

Implement in the derived classes

Implementing Equality

```
class Animal {
  virtual bool equals(Animal *o) = 0;
};
```

```
class Tiger : public Animal {
  std::string color;
  int stripe_count;

  bool equals(Animal *a) {
    Tiger *t = dynamic_cast<Tiger*>(a);
    if (t == NULL)
      return false;
    else
      return (this->color == t->color
              && this->stripe_count == t->stripe_count);
  }
};
```

```
class Snake : public Animal {
  std::string color;
  int weight;
  std::string food;

  bool equals(Animal *a) {
    return ...;
  }
};
```

Implement in the derived classes

Implementing Equality

```
class Animal {
  virtual bool equals(Animal *o) = 0;
};
```

```
class Tiger : public Animal {
  std::string color;
  int stripe_count;

  bool equals(Animal *a) {
    return ...;
  }
};
```

```
class Snake : public Animal {
  std::string color;
  int weight;
  std::string food;

  bool equals(Animal *a) {
    Snake *s = dynamic_cast<Snake*>(a);
    if (s == NULL)
      return false;
    else
      return (this->color == s->color
              && this->weight == s->weight
              && this->food == s->food);
  }
};
```

Project: Introduction

Simple Arithmetic Expressions

```
1

2 + 3

4 * 5

6*(7+8)

(1+2) * (3+4*6)
```

Each of those is an *expression*

An *expression* can be nested in another *expression*

To calculate the result, we can ignore details like whitespace and parentheses

Simple Arithmetic Expressions

```
1

2 + 3

4 * 5

6*(7+8)

(1+2) * (3+4*6)
```

Three kinds of *expression*s:

- number
- addition of two expressions
- multiplication of two expressions

Simple Arithmetic Expressions

```
1

2 + 3

4 * 5

6*(7+8)

(1+2) * (3+4*6)
```

⟨expr⟩ = ⟨number⟩
| ⟨expr⟩ **+** ⟨expr⟩
| ⟨expr⟩ ***** ⟨expr⟩

Simple Arithmetic Expressions

```
1

2 + 3

4 * 5

6*(7+8)

(1+2) * (3+4*6)
```

Abstract syntax *grammar*

⟨expr⟩ = ⟨number⟩
| ⟨expr⟩ **+** ⟨expr⟩
| ⟨expr⟩ ***** ⟨expr⟩

Simple Arithmetic Expressions

```
1

2 + 3

4 * 5

6*(7+8)

(1+2) * (3+4*6)
```

Abstract syntax *grammar*

⟨expr⟩ = ⟨number⟩

"or" | ⟨expr⟩ **+** ⟨expr⟩

| ⟨expr⟩ ***** ⟨expr⟩

Simple Arithmetic Expressions

```
1

2 + 3

4 * 5

6*(7+8)

(1+2) * (3+4*6)
```

Abstract syntax *grammar*

⟨expr⟩ = ⟨number⟩
| ⟨expr⟩ **+** ⟨expr⟩
| ⟨expr⟩ ***** ⟨expr⟩

Gray box corresponds to literal text

Simple Arithmetic Expressions

```
1

2 + 3

4 * 5

6*(7+8)

(1+2) * (3+4*6)
```

Abstract syntax *grammar*

⟨expr⟩ = ⟨number⟩
| ⟨expr⟩ **+** ⟨expr⟩
| ⟨expr⟩ ***** ⟨expr⟩

⟨expr⟩ corresponds to an interface, and
three classes should implement it

Arithmetic Representation Classes

⟨expr⟩ = ⟨number⟩
| ⟨expr⟩ **+** ⟨expr⟩
| ⟨expr⟩ ***** ⟨expr⟩

Arithmetic Representation Classes

⟨expr⟩ = ⟨number⟩
| ⟨expr⟩ **+** ⟨expr⟩
| ⟨expr⟩ ***** ⟨expr⟩

```
class Expr { };
```

```
class Num : public Expr {
};
```

```
class Add : public Expr {
};
```

```
class Mult : public Expr {
};
```

Arithmetic Representation Classes

⟨expr⟩ = ⟨number⟩
| ⟨expr⟩ + ⟨expr⟩
| ⟨expr⟩ * ⟨expr⟩

```
class Expr { };
```

```
class Num : public Expr {
public:
  int val;
};
```

```
class Add : public Expr {
public:
  Expr *lhs;
  Expr *rhs;
};
```

```
class Mult : public Expr {
public:
  Expr *lhs;
  Expr *rhs;
};
```

Arithmetic Representation Classes

⟨expr⟩ = ⟨number⟩
| ⟨expr⟩ + ⟨expr⟩
| ⟨expr⟩ * ⟨expr⟩

```
class Expr { };
```

```
class Num : public Expr {
public:
  int val;

  Num(int val) {
    this->val = val;
  }
};
```

```
class Add : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Add(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
};
```

```
class Mult : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Mult(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
};
```

Arithmetic Representation Classes

```
42

⇒

new Num(42)
```

⟨expr⟩ = ⟨number⟩
| ⟨expr⟩ + ⟨expr⟩
| ⟨expr⟩ * ⟨expr⟩

```
class Expr { };
```

```
class Num : public Expr {
public:
  int val;

  Num(int val) {
    this->val = val;
  }
};
```

```
class Add : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Add(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
};
```

```
class Mult : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Mult(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
};
```

Arithmetic Representation Classes

```
42+2

⇒

new Add(new Num(42),
        new Num(2));
```

⟨expr⟩ = ⟨number⟩
| ⟨expr⟩ **+** ⟨expr⟩
| ⟨expr⟩ ***** ⟨expr⟩

```
class Expr { };
```

```
class Num : public Expr {
public:
  int val;

  Num(int val) {
    this->val = val;
  }
};
```

```
class Add : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Add(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
};
```

```
class Mult : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Mult(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
};
```

Arithmetic Representation Classes

```
((42)+2)
```

⇒

```
new Add(new Num(42),
        new Num(2));
```

⟨expr⟩ = ⟨number⟩
| ⟨expr⟩ **+** ⟨expr⟩
| ⟨expr⟩ ***** ⟨expr⟩

```
class Expr { };
```

```
class Num : public Expr {
public:
  int val;

  Num(int val) {
    this->val = val;
  }
};
```

```
class Add : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Add(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
};
```

```
class Mult : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Mult(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
};
```

Arithmetic Representation Classes

```
((42)+2)
```

⇒

```
new Add(new Num(42),
        new Num(2));
```

⟨expr⟩ = ⟨number⟩
| ⟨expr⟩ + ⟨expr⟩

For tests, we'll need a way to compare expressions

```
class Expr { };
```

```
class Num : public Expr {
public:
  int val;

  Num(int val) {
    this->val = val;
  }
};
```

```
class Add : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Add(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
};
```

```
class Mult : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Mult(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
};
```

Arithmetic Representation Classes

⟨expr⟩ = ⟨number⟩
| ⟨expr⟩ + ⟨expr⟩
| ⟨expr⟩ * ⟨expr⟩

```
class Expr {
public:
  virtual bool equals(Expr *e) = 0;
};
```

```
class Num : public Expr {
public:
  int val;

  Num(int val) {
    this->val = val;
  }
};
```

```
class Add : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Add(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
};
```

```
class Mult : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Mult(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
};
```

Arithmetic Representation Classes

⟨expr⟩ = ⟨number⟩
| ⟨expr⟩ + ⟨expr⟩
| ⟨expr⟩ * ⟨expr⟩

```
class Expr {
public:
  virtual bool equals(Expr *e) = 0;
};
```

= 0 means each subclass *must* override

```
class Num : public Expr {
public:
  int val;

  Num(int val) {
    this->val = val;
  }
};
```

```
class Add : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Add(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
};
```

```
class Mult : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Mult(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
};
```

Backup slides: More examples

Simple class

posn.h

```
class Posn {
public:
  int x;
  int y;

  Posn(int x, int y);
  virtual bool close_to_origin(int d);
};
```

posn.cpp

```
#include "posn.h"
Posn::Posn(int x, int y) {
  this->x = x;
  this->y = y;
}

bool Posn::close_to_origin(int d) {
  return (abs(x) < d) && (abs(y) < d);
}
```

Referring to Classes

```
class Circle {
public:
  Posn* center;
  int radius;

  Circle(Posn* center, int radius) {
    this->center = center;
    this->radius = radius;
  }

  int area() {
    return radius * radius * 3.14;
  }
};
```

Referring to Classes

Use `Posn*` to refer to a `Posn` object

```
class Circle {
public:
  Posn* center;
  int radius;

  Circle(Posn* center, int radius) {
    this->center = center;
    this->radius = radius;
  }

  int area() {
    return radius * radius * 3.14;
  }
};
```

Referring to Classes

```
class Circle {
public:
  Posn* center;
  int radius

  Circle(Posn* center, int radius) {
    this->center = center;
    this->radius = radius;
  }

  int area() {
    return radius * radius * 3.14;
  }
};
```

```
new Circle(new Posn(1, -3), 10)
```

Referring to Classes

```
class Circle {
public:
  Posn* center;
  int radius;

  Circle(Posn* center, int radius) {
    this->center = center;
    this->radius = radius;
  }

  int area() {
    return radius * radius * 3.14;
  }
};
```

So far, could declare with
`class Posn;`
before `class Circle`...

Referring to Classes

```
class Circle {
public:
  Posn* center;
  int radius;

  Circle(Posn* center, int radius) {
    this->center = center;
    this->radius = radius;
  }

  int area() {
    return radius * radius * 3.14;
  }

  bool covers_origin() {
    return center->close_to_origin(radius);
  }
};
```

Referring to Classes

```
class Circle {
public:
  Posn* center;
  int radius;

  Circle(Posn* center, int radius) {
    this->center = center;
    this->radius = radius;
  }

  int area() {
    return radius * radius * 3.14;
  }

  bool covers_origin() {
    return center->close_to_origin(radius);
  }
};
```

Needs `#include "posn.h"`

Separating Declaration and Implementation

circle.h

```
Posn.h must be before circle.h,
otherwise, Posn* center will be unknown
to the compiler and in this case you need
to declare as class Posn here;

class Circle {
  public:
    Posn* center;
    int radius;

    Circle(Posn* center, int radius);
    int area();
    bool covers_origin();
  };
```

circle.cpp

```
#include "posn.h"
#include "circle.h"

Circle::Circle(Posn* center, int radius) {
  this->center = center;
  this->radius = radius;
}

int Circle::area() {
  return radius * radius * 3.14;
}

bool Circle::covers_origin() {
  return center->close_to_origin(radius);
}
```

Avoid `#include "....h"` in `.h` files

Subclasses

posn3d.h

```
#include "posn.h"

class Posn3D : public Posn {
  int z;

  Posn3D(int x, int y, int z);

  virtual bool close_to_origin(int d);
};
```

posn3d.cpp

```
#include "posn3d.h"

Posn3D::Posn3D(int x, int y, int z)
: Posn(x, y)
{
    this->z = z;
}

bool Posn3D::close_to_origin(int d) {
  return (Posn::close_to_origin(d)
          && (abs(z) < d));
}
```

Subclasses

Means **extends**

posn3d.h

```
#include "posn.h"

class Posn3D : public Posn {
  int z;

  Posn3D(int x, int y, int z);

  virtual bool close_to_origin(int d);
};
```

posn3d.cpp

```
#include "posn3d.h"

Posn3D::Posn3D(int x, int y, int z)
: Posn(x, y)
{
    this->z = z;
}

bool Posn3D::close_to_origin(int d) {
  return (Posn::close_to_origin(d)
          && (abs(z) < d));
}
```

Subclasses

Needed for subclassing

posn3d.h

```
#include "posn.h"

class Posn3D : public Posn {
  int z;

  Posn3D(int x, int y, int z);

  virtual bool close_to_origin(int d);
};
```

posn3d.cpp

```
#include "posn3d.h"

Posn3D::Posn3D(int x, int y, int z)
: Posn(x, y)
{
    this->z = z;
}

bool Posn3D::close_to_origin(int d) {
  return (Posn::close_to_origin(d)
          && (abs(z) < d));
}
```

Subclasses

posn3d.h

```
#include "posn.h"

class Posn3D : public Posn {
  int z;

  Posn3D(int x, int y, int z);

  virtual bool close_to_origin(int d);
};
```

posn3d.cpp

```
#include "posn3d.h"

Posn3D::Posn3D(int x, int y, int z)
: Posn(x, y)
{
    this->z = z;
}

bool Posn3D::close_to_origin(int d) {
  return (Posn::close_to_origin(d)
          && (abs(z) < d));
}
```

Superclass constructor

Superclass constructor: similar to super in java

Subclasses

posn3d.h

```
#include "posn.h"

class Posn3D : public Posn {
  int z;

  Posn3D(int x, int y, int z);

  virtual bool close_to_origin(int d);
};
```

posn3d.cpp

```
#include "posn3d.h"

Posn3D::Posn3D(int x, int y, int z)
: Posn(x, y)
{
    this->z = z;
}

bool Posn3D::close_to_origin(int d) {
  return (Posn::close_to_origin(d)
          && (abs(z) < d));
}
```

Super method call

Subclasses

posn3d.h

```
#include "posn.h"

class Posn3D : public Posn {
  int z;

  Posn3D(int x, int y, int z);

  virtual bool close_to_origin(int d);
};
```

Can't avoid `#include "posn.h"` in `posn3d.h`

posn3d.cpp

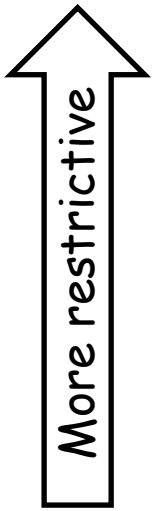
```
#include "posn3d.h"

Posn3D::Posn3D(int x, int y, int z)
: Posn(x, y)
{
    this->z = z;
}

bool Posn3D::close_to_origin(int d) {
  return (Posn::close_to_origin(d)
          && (abs(z) < d));
}
```

Access Modifiers

Inheritance: Access Modifiers



Private — Accessible *only* by the class objects

Protected — Accessible by the class objects and objects of all derived classes

Public — Accessible by all objects

Access Specifiers in the base class			
	private	protected	public
Private inheritance →	The member is inaccessible.	The member is private.	The member is private.
Protected inheritance	The member is inaccessible.	The member is protected.	The member is protected.
Public inheritance	The member is inaccessible.	The member is protected.	The member is public.

Access Modifiers: Example

```
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A    // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```