SAPIENZA
UNIVERSITÀ DI ROMA

Robot Programming
C++ Inheritance

Giorgio Grisetti

Inheritance

C++ supports inheritance, with rules similar to Java and Python

A C++ class can derive from multiple base classes

A derived class/struct has all members of the base class plus its own

A pointer to the derived class is also a pointer to the base class but the opposite is not true

When deriving a class, the constructor chain should be preserved (if the default ctor has been deleted in the base, a specific one should be called)

Inheritance

```
<class|struct> Type [:<public|protected|private>
<BaseType>] { <body }
```

Example

```
class A1 {…};
class A2 {…};

class B: public A1, public A2 { ...}
```

Inheritance Example

```
#include <iostream> // discover this later
using namespace std; // discover this later

struct A {
  int my_value;
  A(int v) {
    cerr << "A["<<this<<"]"<<"ctor" << endl;
    my_value=v;
  }
  ~A(){
    cerr << "A["<<this <<"]"<<"dtor" << endl;
  }
  void print() {
    cerr << "A::print() [" << this << "]:
my_value=" << my_value << endl;
  }
};
```

```
struct B: public A {
  int my_other_value;
  // use the initializer list before {
  // it would not be possible to initialize
  // objects whose default ctor has been deleted
  B():
    A(0),  // base class initialization
    my_other_value(-1){
    cerr << "B [ " << this << "] ctor " << endl;
  }

  ~B(){
    cerr << "B [" << this << "]" << " dtor" << endl;
  }
  void print() {
     // call method of the base class using scoping
<classname>::<method_or_variable>
    A::print();
    cerr << "B::print() [" << this << "]:
my_other_value=" << my_other_value << endl;
  }

};
```

Inheritance Example

Simple test main

We try assigning objects to derived and base class variables

Assigning from base to derived does not work, since there might be non initialized fields

The other way, in general works (under the defaults)

```
int main(int argc, char** argv) {
  cerr << "create A " << endl;
  A a(3);
  a.print();

  cerr << "create B " << endl;
  B b;
  b.print();

  // legal assignment a is base class of
b, only common member vars are copied
  A a1=b;
  a1.print();
}
```

Late Binding

In Java all methods support overriding. In C++ only those that are declared virtual.

This choice favors efficiency, since pure “non virtual” classes/structs enable for more effective compile time optimizations.

To enable the overriding of a method, you need to put the keyword “**`virtual`**” before its declaration in the class.

In the derived class you might want (optional) to claim a certain method has been overriding one on the base class. Use the **`override`** clause after the signature

From that point on, all methods in the derived classes whose signature matches the one of the overridden method will be called.

You can still use a “specific” method of a base class calling it explicitly with the scoping operator `<Type>::method`

Late Binding Example

```
class A {
public:
  A(int value_):
    _value(value_){
  }
  void print() {
    cerr << "A:["<<this<<"], value:"<<_value<< endl;
  }
protected:
  int _value;
};

class B: public A {
public:
  B(int value_):
    A(value_) {}
  void print(){
    cerr << "B:["<<this<<"],value:"<<_value<<endl;
  }
};
```

```
class A_lb {
public:
  A_lb(int value_):
    _value(value_){

  }
  virtual void print() {
    cerr<<"A_lb:["<<this<<"],value:"<<_value<<endl;
  }
protected:
  int _value;
};

class B_lb: public A_lb {
public:
  B_lb(int value_):
    A_lb(value_) {}
  void print() override {
    cerr<<"B_lb:["<<this<<"],value:"<<_value<<endl;
  }
};
```

Late Binding Example

To exploit the power of polimorphism we need to use references or pointers.

The “concrete” object, is still an instance of a specific type in memory.

Structs and Classes in C++ are the same. The first default to public, the second to private.

It is a good custom to use structs for simple data-only items, while using classes for all the rest

```
int main(int argc, char ** argv) {
  cerr <<  "no late binding" << endl;
  {
    B b(10);
    A& a_ref=b;
    b.print();
    a_ref.print();
  }
  cerr <<  "late binding" << endl;
  {
    B_lb b(10);
    A_lb& a_ref(b);
    b.print();
    a_ref.print();

    // a is a copy of b
    A_lb copy_of_b_as_a(b);
    copy_of_b_as_a.print();
  }
}
```

Dynamic Cast

For classes that have at least one virtual member, it is possible to perform a safe upcast, called dynamic cast.

A pointer or a reference to the base class can be converted to a derived class

The operation succeeds only if the casted object is an instance of derived

On failure,

- If the dynamic_cast is operated on a pointer, te the result nullptr
- if the dynamic_cast is operated on a reference an exception is thrown (and can be caught and handled)

```
int main(int argc, char ** argv)
{
    B_lb b(10);
    A_lb a(5);
    A_lb* a_ptr = &a;
    A_lb* b_ptr = &b;
    B_lb*
p=dynamic_cast<B_lb*>(a_ptr)
    // p==0, cast failed
    p=dynamic_cast<B_lb*>(b_ptr)
    // p==b_ptr, cast succeeds
  }
}
```

Example of Multiple Inheritance

Multiple inheritance can be used when creating classes that inherit properties from multiple bases.

An PrintableDrawablePoint is an member of all bases, and can be passed to a function that knows only the bases.

```
struct Drawable {
  virtual void draw() =0;
};

struct Printable {
  virtual void print() = 0;
};

struct Point {
  float x,y;
  Point(float x_=0, float y_=0): x(x_), y(y_){}
};

struct PrintableDrawablePoint:
       public Point,
       public Printable,
       public Drawable {
   PrintableDrawablePoint(float x_=0, float y_=0):
     Point(x_,y_){}
   void draw() override { … };
   void print() override { … }
};
```

Example of Multiple Inheritance

Multiple inheritance can be used when creating classes that inherit properties from multiple bases.

An PrintableDrawablePoint is an member of all bases, and can be passed to a function that knows only the bases.

```
void rotate(Point& dest,
            const Point& src,
            float alpha ){
  float x=dest.x * cos(alpha) - dest.y * sin(alpha);
  float y=dest.x * sin(alpha) + dest.y * cos(alpha);
  dest.x=x;
  dest.y=y;
}

void drawAll(Drawable* dv, int size) {
  for (int i=0; i<size; ++i)
   dv[i]->draw();
}

int main() {
  PrintableDrawablePoint p(1,2);
  PrintableDrawablePoint pr;
  rotate(pr, p, M_PI/2);
  drawAll(&p,1);
};
```

Diamond Problem

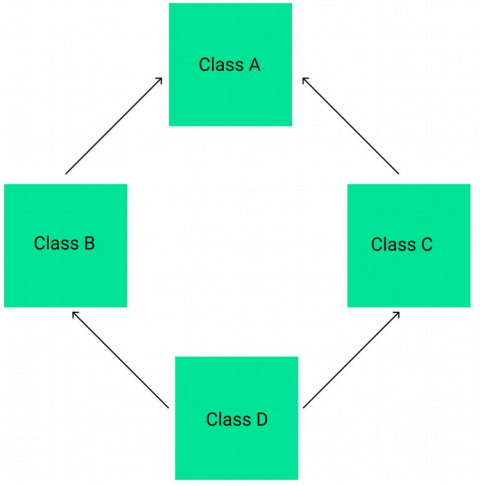
Multiple inheritance could lead to the diamond problem.

A base class A can be derived more than once.

When this happens, mutiple instances of A exist (one inside B, and one inside C)

The problem can be resolved through virtual inheritance (advanced topic)

Overall it is better to avoid the diamond as virtual inheritance incurs in performance penalties



RP Simulator

- Example of a simulator (done yesterday) that supports

  - multiple robots

  - multiple devices

- Specializes for differential drive and lidars

Concepts:

World: is a grid map, each pixel si a square of res x res cm

World loaded from an image (done by me)

Pixels darker than 127 represent an obstacle

The world is populated by multiple objects

Objects can be placed on top of other objects

RP simulator

The simulator evolves in time steps

At each steps it updates the world

Each object in the world knows what to do

- e.g. robot moves, lidar scans etc

The robot is controlled in

- linear and rotational velocities
- cannot be in an occupied area

  (handle collisions, if it bumps the velocities are set to zero and it does not move from where it is)

The lidar casts a fan of rays form the current position (Wherever it is), and walks along the ray.

A range measurement is either range_max or the distance of an object from the lidar along the ray.

The lidar has a max number of beams and an angular resolution.

The lidar does not need to be mounted necessarily on the robot

Exercises

Extend the simulator class with another type of robot that can be controlled in translational velocity (vx, vy) and angular velocity.

Add a destructor to the world_item, that if called erases the reference to an object stored in the simulator.

Add the Pan Unit device that allows to whatever is put on top to rotate along its z axis of a desired angle. Test the new device by adding one between the robot and the lidar, and see what happens.