CRICOS PROVIDER 00123M

School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Polymorphism and Summary of OOP Concepts

adelaide.edu.au

seek LIGHT

Review

- Inheritance
  - Three accessibility keywords
- Friend
- Overloading functions
- Multiple Inheritance
  - Virtual keyword

- By the way! Before I forget! You don't need to work with c-strings anymore in this course. Work with "string"s which is more advanced library in c++
  - http://www.cplusplus.com/reference/string/string/
- And do an initial submit asap. You can change it as many times as you want before the deadline.

Few more minutes on ambiguity…

Ambiguity on Multiple Inheritance

```
class A
{
    public: int memberInCommonParent;
};

class B: public A
{
    public: int sameNameMember;
};

class C: public A
{
    public: int sameNameMember;
};

class D:public B, public C
{
    //let's have nothing!
} d;
```

```
int var;
var = d.memberInCommonParent;
var = d.memberInCommonParent;
var = d.sameNameMember;
var = d.sameNameMember;
```

```
int var;
var = d.B::memberInCommonParent;
var = d.C::memberInCommonParent;
var = d.B::sameNameMember;
var = d.C::sameNameMember;
```

Use virtual inheritance for fixing this ambiguity

Ambiguity on defined names

- Names must be unique in their space.
- You can declare namespaces to provide a scope for variables

```
namespace mynamespace1 {
    int globalVar;
    int function1()
    {
        return 0;
    }
}

namespace mynamespace2 {
    int globalVar;
    int function1()
    {
        return 0;
    }
}
```

```
int main() {
    using namespace mynamespace1;
    cout<< globalVar;
    cout<< function1();
    cout<< mynamespace2::globalVar;
    cout<< mynamespace2::function1();
```

```
int main() {
    using mynamespace1::globalVar;
    cout<< globalVar;
    cout<< function1();
    cout<< mynamespace2::globalVar;
    cout<< mynamespace2::function1();
```

- Note how the scope resolution operator (::) is used.

This session

- Polymorphism
- Overriding
- Redefining
- Which version of the function should be called?!

Polymorphism

- What is Polymorphism?

- The provision of a single interface to entities of different types (From Wikipedia!)

- We need to associate multiple meanings with one function name.
- In what ways can we do this in C++?

Compile-time checking

- How is it compiled?
- What do we need?

- Here, the C++ compiler decides which function to call before the program starts running.
- This is Redefinition (note that it uses compile-time checking).
- There's another way: run-time checking.

```
class Animal{
public:
  string toString(){
    return "An animal";
  }
};

class Bird : public Animal {
  string toString(){
    return "A bird";
  }
};

class SeaC : public Animal {
  string toString(){
    return "A sea creature";
  }
};

void print(Animal a){
  cout << a.toString() << endl;
}

int main(){
  print(Animal());
  print(Bird());
  print(SeaC());

  return 0;
}
```

Run-time checking

- C++ uses a mechanism **called *late binding or dynamic binding*** to determine which version of a function it calls at any particular time.
- This happens when the code is being executed and the system can determine which function to call, based on the subclass that is being used.
- This allows us to *really use* polymorphism.

The virtual keyword and the process

- How is it done in c++?
  - Declare the function in the parent class with keyword "virtual"

```
virtual int test(int n, int acc);
```

  - C++ makes a virtual table for that class.
  - The table is copied for a child class, and the addresses are overridden, as the functions are re-implemented in child. At compile time.
  - Whenever an object of one of child class is constructed, a pointer to the table of that class is also stored in it.
  - Which code to execute? Decided at runtime.

Overloading, Overriding & Redefining

- They are all based on sharing the same function name.

- Overloaded functions
  - Same function name but different parameter list
  - Quite irrelevant to our topic today!!
- Redefined functions
  - Same function signature but different implementation in derived and based classes. Called in the same way as ordinary functions.
- Overridden functions
  - Virtual keyword
  - Same function signature but different implementation in derived and based classes. Call by a reference to the *virtual method table*: Which function to call? decided at runtime

The virtual keyword

- If you make something virtual in the base class, it is automatically virtual when declared in the children. (It would be better to still label it virtual).
- You add the reserved word `virtual` to the declaration, not the implementation.
- When we change a virtual function's implementation, we are **overriding**. If we change the definition without `virtual`, it's **redefining**.

The virtual keyword

- If virtual functions are so handy, why not use them for all member functions?
  - Efficiency! your program risks being a lot slower if you use it for everything.
  - It's a lot more work to correctly track virtual functions.

Example

- Does the virtual keyword solve our problem?

```
class Animal{
public:
  virtual string toString(){
    return "An animal";
  }
};

class Bird : public Animal {
  virtual string toString(){
    return "A bird";
  }
};

class SeaC : public Animal {
  virtual string toString(){
    return "A sea creature";
  }
};

void print(Animal a){
  cout << a.toString() << endl;
}

int main(){
  print(Animal());
  print(Bird());
  print(SeaC());

  return 0;
}
```

The slicing problem

- We can assign an instance of the derived class to a variable of parent class.
- But we 'slice off' the added fields and functions.
- What does this code print?
- We can use pointers to get it right.

```
class Animal{
public:
  virtual void print();
  string color;
};

class Bird : public Animal {
public:
  virtual void print();
  string featherColor;
};

Animal a;
Bird b;

b.color = "white";
b.featherColor = "red";
a = b;

cout << a.color;
cout << a.featherColor;
```

Pointers to instances

- We have a Parent and a Child.
- Child has an additional variable, addField.
- They share a virtual `print()` function, that the Child class overrides:

```
Parent *pPtr;
Child *cPtr;
cPtr = new Child();

cPtr -> print();
pPtr = cPtr;
pPtr -> print();
```

- Late binding happens!

Example

```
class Animal{
public:
  virtual void print();
  string color;
};

class Bird : public Animal {
public:
  virtual void print();
  string featherColor;
};


Bird *bPtr;
bPtr= new Bird;
bPtr->color = "white";
bPtr->featherColor = "red";

Animal *aPtr;
aPtr = bPtr;

aPtr->print();
bPtr->print();
```

Example

Which virtual keyword is necessary?


```
class Animal{
public:
  virtual string toString(){
    return "An animal";
  }
};

class Bird : public Animal {
  virtual string toString(){
    return "A bird";
  }
};

class SeaC : public Animal {
  virtual string toString(){
    return "A sea creature";
  }
};

void print(Animal *a){
  cout << a->toString() << endl;
}

int main(){
  Animal a = Animal();
  Bird b = Bird();
  SeaC s = SeaC();

  print(&a);
  print(&b);
  print(&s);

  return 0;
}
```

Dynamic binding

- To enable late/dynamic binding for a function, we need to guarantee:
  - The function must be declared with the virtual keyword in the base class.
  - The variable that references the object for the function must contain the address of the object

Summary

- We covered polymorphism.
- You should now know the difference between overloading, redefining and overriding.
- You should also understand the different roles of compile-time and run-time checking, as well as what dynamic or late binding means.


THE UNIVERSITY
of ADELAIDE