G64OOS (Spring 2014)

Lecture 05 r02

OO Programming in C++ (1/2)

Peer-Olaf Siebers

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

Motivation

- Today you will learn:
  - The differences between stack and heap memory
  - How to apply inheritance in your C++ programs
    - How you derive one class from another
    - How you access base methods from derived classes
    - How you override base methods
  - How to apply polymorphism in your C++ programs
    - What virtual methods are
    - How virtual methods enable you to use your base class polymorphically

Pointers

- A pointer is a special type of variable that holds the address of an object in memory

- Example
  - int howOld=50;
  - int *pAge=&howOld;
  - int yourAge=*pAge; // dereferencing

Pointers

- A pointer is a special type of variable that holds the address of an object in memory

- Example
  - int howOld=50;
  - int* pAge=&howOld;
  - int yourAge=*pAge; // dereferencing

- All pointers should be initialised when they are created

Stack and Heap

- Stack
  - Static memory for local variables + function parameters
  - Cleaned automatically when a function returns
- Heap
  - Dynamically allocated memory (by the programmer)
  - Heap is not cleaned automatically
  - The memory reserved remains available until it is explicitly cleaned
- Why pointers?
  - Managing data on the heap
  - Accessing class member data and functions
  - Passing variables by reference to functions

Using the Heap

- Using the "new" keyword
  - int *pPointer=new int;
  - *pPointer=10;

- Using the "delete" keyword
  - delete pPointer; // only returns memory

- Remember that the pointer itself is a local variable, so you need to return the memory before leaving the function in which the pointer is declared!
  - Every "new" should have a corresponding "delete"

Using the Heap

- Creating objects on the heap
  - Cat *pCat=new Cat
- Deleting the object
  - delete pCat;

- Accessing data members using pointers
  - To access an object on the stack: Use "." operator
  - To access an object on the heap: (*pCat).getAge(); or pCat->getAge;

Using the Heap

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

class Cat{
private:
    int age;
public:
    Cat(){
        age=2;
    }

    ~Cat(){
    }

    int getAge()const{
        return age;
    }

    void setAge(int newAge){
        age=newAge;
    }
};

main(){
    Cat *pCat=new Cat;
    cout<<"Age: "<<(*pCat).getAge()<<endl;
    pCat->setAge(5);
    cout<<"Age: "<<pCat->getAge()<<endl;
    delete pCat;
    return 0;
}
```

```
Age: 2
Age: 5
```

Using the Heap

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

class Cat{
private:
    int age;
public:
    Cat():age(2){}
    ~Cat(){}
    int getAge()const{return age;}
    void setAge(int newAge){age=newAge;}
};

main(){
    Cat *pCat=new Cat;
    cout<<"Age: "<<pCat->getAge()<<endl;
    pCat->setAge(5);
    cout<<"Age: "<<pCat->getAge()<<endl;
    delete pCat;
    return 0;
}
```

- Saving some space

```
Age: 2
Age: 5
```

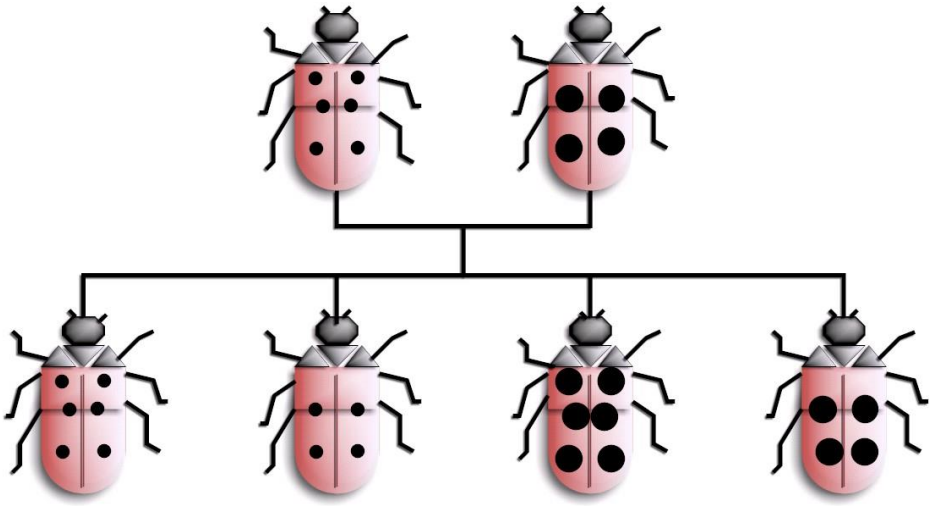
Using the Heap

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

class Cat{
private:
    int age;
public:
    // prototypes
    Cat();
    ~Cat();
    int getAge()const;
    void setAge(int newAge);
};

main(){
    Cat *pCat=new Cat;
    cout<<"Age: "<<pCat->getAge()<<endl;
    pCat->setAge(5);
    cout<<"Age: "<<pCat->getAge()<<endl;
    delete pCat;
    return 0;
}
// implementation (:: = scope operator)
Cat::Cat(){age=2;}
Cat::~Cat(){}
int Cat::getAge()const{return age;}
void Cat::setAge(int newAge){age=newAge;}
```

- Providing prototypes and adding implementation later

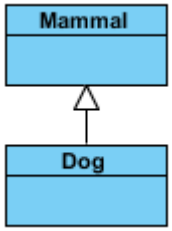
```
Age: 2
Age: 5
```

Inheritance and Derivation

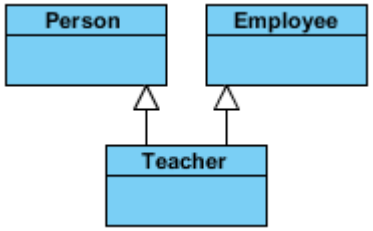


Inheritance and Derivation

- C++ attempts to represent "is a" or "is a kind of" relationships by defining classes that derive from one another
- Derived classes are super sets of the base classes
- Syntax for single derivation:
  - class Dog : public Mammal {};



- Syntax for multiple derivation:
  - class Teacher : public Person, public Employee {};



```
#include <iostream>
using namespace std;
// set up a collection of named integer constants
enum BREED{YORKIE,SHETLAND,DOBERMAN};

class Mammal{
protected:
    int age;
    int weight;
public:
    Mammal():age(2),weight(5){cout<<"Mammal constructor\n";}
    ~Mammal(){cout<<"Mammal destructor\n";}

    int getAge()const{return age;}
    void setAge(int newAge){age=newAge;}
    int getWeight()const{return weight;}
    void setWeight(int newWeight){weight=newWeight;}

    void speak()const{cout<<"Mammal speaks\n";}
    void sleep()const{cout<<"Mammal sleeps\n";}
};

class Dog:public Mammal{
private:
    BREED breed;
public:
    Dog():breed(YORKIE){cout<<"Dog constructor\n";}
    ~Dog(){cout<<"Dog destructor\n";}

    BREED getBreed()const{return breed;}
    void setBreed(BREED newBreed){breed=newBreed;}

    void wagTail()const{cout<<"Dog wags tail\n";}
    void begForFood()const{cout<<"Dog begs for food\n";}
};

int main(){
    Dog fido;
    fido.speak();
    fido.wagTail();
    cout<<"Fido is "<<fido.getAge()<<" years old\n";
    return 0;
}
```

```
Mammal constructor
Dog constructor
Mammal speaks
Dog wags tail
Fido's age is 2
Dog destructor
Mammal destructor
```

Passing arguments to base constructors

- Base class initialisation from a sub class constructor can be performed by writing the base class name followed by the parameters expected by the base class

```
#include <iostream>
using namespace std;
enum BREED{YORKIE,SHETLAND,DOBERMAN};

class Mammal{
protected:
    int age;
    int weight;
public:
    // prototypes
    Mammal();
    Mammal(int age);
    ~Mammal();
    // direct implementations (no prototype required)
    int getAge()const{return age;}
    void setAge(int newAge){age=newAge;}
    int getWeight()const{return weight;}
    void setWeight(int newWeight){weight=newWeight;}

    void speak()const{cout<<"Mammal speaks\n";}
    void sleep()const{cout<<"Mammal sleeps\n";}
};

class Dog:public Mammal{
private:
    BREED breed;
public:
    Dog();
    Dog(int age);
    Dog(int age,int weight);
    ~Dog();

    BREED getBreed()const{return breed;}
    void setBreed(BREED newBreed){breed=newBreed;}

    void wagTail()const{cout<<"Dog wags tail\n";}
    void begForFood()const{cout<<"Dog begs for food\n";}
};

int main(){
    Dog fido;
    fido.speak();
    fido.wagTail();
    Dog rover(6,10);
    cout<<"Rover weights "<<rover.getWeight()<<" kg\n";
    return 0;
}
// implementation (:: = scope oprator)
Mammal::Mammal():age(2),weight(5){cout<<"Mammal() constructor\n";}
Mammal::Mammal(int age):age(age),weight(5){cout<<"Mammal(int) constructor\n";}
Mammal::~Mammal(){cout<<"Mammal destructor\n";}
Dog::Dog():Mammal(),breed(YORKIE){cout<<"Executes Mammal() and then Dog() constructor body\n";}
Dog::Dog(int age):Mammal(age),breed(YORKIE){cout<<"Executes Mammal(int) and then Dog(int) constructor body \n";}
Dog::Dog(int age,int newWeight):Mammal(age),breed(YORKIE){setWeight(newWeight); cout<<"Executes Mammal(int) and then Dog(int,int) constructor body\n";}
Dog::~Dog(){cout<<"Dog destructor\n";}
```

- When passing arguments to base constructors you are not allowed to initialise a value in the base class
- Use the corresponding setter function of the base class instead

```
Mammal() constructor
Executes Mammal() and then Dog() constructor body
Mammal speaks
Dog wags tail
Mammal(int) constructor
Executes Mammal(int) and then Dog(int,int) constructor body
Rover weights 10 kg
Dog destructor
Mammal destructor
Dog destructor
Mammal destructor
```

Overriding Functions

- Objects from a derived class have access to all base class member functions and can override these
  - Overriding means changing the implementation of a base class function in a derived class
  - The overriding member function needs to have same return type and signature (function prototype) as the base class member function

- Do not confuse this with overloading!
  - When overloading a member function you create more than one member function with the same name but different signatures (in the pervious example we overloaded constructors)

Overriding Functions

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

class Mammal{
public:
    Mammal(){cout<<"Mammal constructor\n";}
    ~Mammal(){cout<<"Mammal destructor\n";}

    void speak()const{cout<<"Mammal speak";}
    void speak(string text)const{cout<<"Mammal speak text"<<"\n";}
    void sleep()const{cout<<"Mammal sleep";}
};

class Dog:public Mammal{
public:
    Dog(){cout<<"Dog constructor\n";}
    ~Dog(){cout<<"Dog destructor\n";}

    void speak()const{cout<<"Dog speak\n";}
};

int main(){
    Mammal max;
    max.speak();
    max.speak("Hello\n");
    Dog fido;
    fido.speak();
    fido.speak("Hello\n");
    return 0;
}
```

- If the base class has a member function that is overloaded, and the derived class overrides this function, all overloaded member functions in the base class with the same name will be hidden

```
28      fido.speak();
29      fido.speak("Hello\n");
30      return 0;
31  }
32
33
34
```

Logs & others

Code::Blocks | Search results | Build log | Build messages | Debugger

File	Line	Message
L:\Teaching\G6...		In function 'int main()':
L:\Teaching\G6...	29	error: no matching function for call to 'Dog::speak(const char [7])'
L:\Teaching\G6...	20	note: candidates are: void Dog::speak() const
		=== Build finished: 1 errors, 0 warnings ===

Overriding Functions

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

class Mammal{
public:
    Mammal(){cout<<"Mammal constructor\n";}
    ~Mammal(){cout<<"Mammal destructor\n";}

    void speak()const{cout<<"Mammal speak";}
    void speak(string text)const{cout<<"Mammal speak text"<<"\n";}
    void sleep()const{cout<<"Mammal sleep";}
};

class Dog:public Mammal{
public:
    Dog(){cout<<"Dog constructor\n";}
    ~Dog(){cout<<"Dog destructor\n";}

    void speak()const{cout<<"Dog speak\n";}
};

int main(){
    Mammal max;
    max.speak();
    max.speak("Hello\n");
    Dog fido;
    fido.speak();
    fido.Mammal::speak("Hello\n");
    return 0;
}
```

- It is still possible to access the hidden member functions by fully qualifying the name of the method

```
Mammal constructor
Mammal speakMammal speak text
Mammal constructor
Dog constructor
Dog speak
Mammal speak text
Dog destructor
Mammal destructor
Mammal destructor
```

Using Polymorphism and Derived Classes

- Polymorphism means that some code or operations or objects behave differently in different contexts
  - Inheritance means the Dog object inherits attributes and capabilities from the Mammal object
  - Polymorphism allows derived objects to be treated as if they where base objects
  - You can use polymorphism to declare a pointer to Mammal and assign to it the address of a Dog object you create on the heap
    - Mammal *pMammal=new Dog;
  - You can then use this pointer to invoke any member function on Mammal; the functions that are overridden in Dog will call the correct function if we use **virtual member functions**

Virtual Member Functions

```
1   #include <iostream>
2   using namespace std;
3
4   class Mammal{
5   public:
6       Mammal(){cout<<"Mammal constructor\n";}
7       ~Mammal(){cout<<"Mammal destructor\n";}
8
9       void speak()const{cout<<"Mammal speak\n";}
10      void sleep()const{cout<<"Mammal sleep\n";}
11  };
12
13  class Dog:public Mammal{
14  public:
15      Dog(){cout<<"Dog constructor\n";}
16      ~Dog(){cout<<"Dog destructor\n";}
17
18      void speak()const{cout<<"Dog speak\n";}
19  };
20
21  int main(){
22      // Mammal pointer that points to a dog object
23      Mammal *pDog=new Dog;
24      pDog->sleep();
25      pDog->speak();
26      delete pDog;
27      return 0;
28  }
```

- We have a Mammal pointer
  - speak() is not overridden
  - Dog destructor is not called

```
Mammal constructor
Dog constructor
Mammal sleep
Mammal speak
Mammal destructor
```

Virtual Member Functions

```
1   #include <iostream>
2   using namespace std;
3
4   class Mammal{
5   public:
6       Mammal(){cout<<"Mammal constructor\n";}
7       ~Mammal(){cout<<"Mammal destructor\n";}
8
9       virtual void speak()const{cout<<"Mammal speak\n";}
10      void sleep()const{cout<<"Mammal sleep\n";}
11  };
12
13  class Dog:public Mammal{
14  public:
15      Dog(){cout<<"Dog constructor\n";}
16      ~Dog(){cout<<"Dog destructor\n";}
17
18      void speak()const{cout<<"Dog speak\n";}
19  };
20
21  int main(){
22      // Mammal pointer that points to a dog object
23      Mammal *pDog=new Dog;
24      pDog->sleep();
25      pDog->speak();
26      delete pDog;
27      return 0;
28  }
```

- The keyword "virtual" signals that the derived class will probably want to override the virtual function

```
Mammal constructor
Dog constructor
Mammal sleep
Dog speak
Mammal destructor
```

Virtual Member Functions

```
1  #include <iostream>
2  using namespace std;
3
4  class Mammal{
5  public:
6      Mammal(){cout<<"Mammal constructor\n";}
7      virtual ~Mammal(){cout<<"Mammal destructor\n";}
8
9      virtual void speak()const{cout<<"Mammal speak\n";}
10     void sleep()const{cout<<"Mammal sleep\n";}
11 };
12
13 class Dog:public Mammal{
14 public:
15     Dog(){cout<<"Dog constructor\n";}
16     ~Dog(){cout<<"Dog destructor\n";}
17
18     void speak()const{cout<<"Dog speak\n";}
19 };
20
21 int main(){
22     // Mammal pointer that points to a dog object
23     Mammal *pDog=new Dog;
24     pDog->sleep();
25     pDog->speak();
26     delete pDog;
27     return 0;
28 }
```

- If any member functions in your class are virtual then the destructor should also be virtual!
  - This will override the Mammal destructor with a Dog destructor
  - Otherwise you have a memory leak as only the type of object that the pointer is supposed to point to (in our case Mammal) will be deleted

```
Mammal constructor
Dog constructor
Mammal sleep
Dog speak
Dog destructor
Mammal destructor
```

Runtime Binding

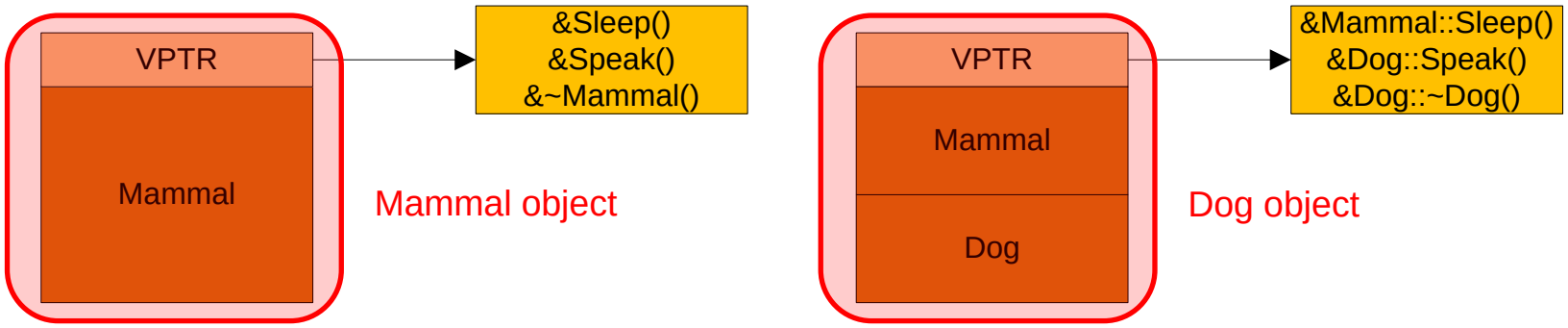
```
1  #include <iostream>
2  using namespace std;
3
4  class Mammal{
5  public:
6      Mammal(){cout<<"Mammal constructor\n";}
7      virtual ~Mammal(){cout<<"Mammal destructor\n";}
8      virtual void speak()const{cout<<"Mammal speak\n";}
9  };
10
11 class Cat:public Mammal{
12 public:
13     Cat(){cout<<"Cat constructor\n";}
14     ~Cat(){cout<<"Cat destructor\n";}
15     void speak()const{cout<<"Cat speak\n";}
16 };
17
18 int main(){
19     int size=3;
20     Mammal *array[size];
21     Mammal *ptr;
22     int choice;
23     for(int i=0;i<size;i++){
24         cout<<"1=cat; 2=mammal: ";
25         cin>>choice;
26         switch(choice){
27             case 1: ptr=new Cat; break;
28             default: ptr=new Mammal; break;
29         }
30         array[i]=ptr;
31     }
32     for(int i=0;i<size;i++){
33         array[i]->speak();
34         delete array[i];
35     }
36     return 0;
37 }
```

- It is impossible to know at compile time which object will be created and therefore which speak() method will be invoked
- The pointer "prt" is bound to its object at runtime; this is called **late binding** or **runtime binding**

```
1=cat; 2=mammal: 2
Mammal constructor
1=cat; 2=mammal: 1
Mammal constructor
Cat constructor
1=cat; 2=mammal: 2
Mammal constructor
Mammal speak
Mammal destructor
Cat speak
Cat destructor
Mammal destructor
Mammal speak
Mammal destructor
```

Runtime Binding

- How does late binding work?
  - When a virtual member function is created in an object the object must keep track of that member function
  - Compilers build a virtual function table (v-table) - one for each type and each object of that type keeps a v-table pointer (v-ptr)

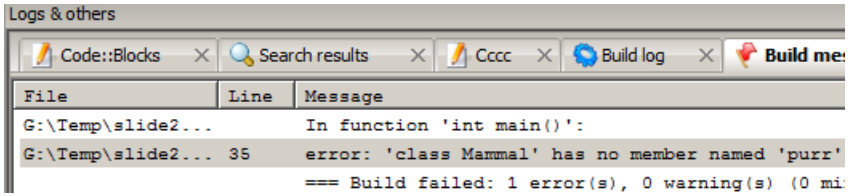


  - When the Dog constructor is called the v-ptr is adjusted to point to the virtual function overrides in the Dog object

Dynamic Casting

```
1  #include <iostream>
2  using namespace std;
3
4  class Mammal{
5  public:
6      Mammal(){cout<<"Mammal constructor\n";}
7      virtual ~Mammal(){cout<<"Mammal destructor\n";}
8      virtual void speak()const{cout<<"Mammal speak\n";}
9  };
10
11 class Cat:public Mammal{
12 public:
13     Cat(){cout<<"Cat constructor\n";}
14     ~Cat(){cout<<"Cat destructor\n";}
15     void speak()const{cout<<"Cat speak\n";}
16     void purr()const{cout<<"Cat purrs\n";}
17 };
18
19 int main(){
20     int size=3;
21     Mammal *array[size];
22     Mammal *ptr;
23     int choice;
24     for(int i=0;i<size;i++){
25         cout<<"1=cat; 2=mammal: ";
26         cin>>choice;
27         switch(choice){
28             case 1: ptr=new Cat; break;
29             default: ptr=new Mammal; break;
30         }
31         array[i]=ptr;
32     }
33     for(int i=0;i<size;i++){
34         array[i]->speak();
35         array[i]->purr();
36         delete array[i];
37     }
38     return 0;
39 }
```

- What happens if you want to add a member function to Cat that is inappropriate for Mammal?
- Calling "purr()" using your pointer to Mammal will produce a compiler error



Dynamic Casting

```
1  #include <iostream>
2  using namespace std;
3
4  class Mammal{
5  public:
6      Mammal(){cout<<"Mammal constructor\n";}
7      virtual ~Mammal(){cout<<"Mammal destructor\n";}
8      virtual void speak()const{cout<<"Mammal speak\n";}
9  };
10
11 class Cat:public Mammal{
12 public:
13     Cat(){cout<<"Cat constructor\n";}
14     ~Cat(){cout<<"Cat destructor\n";}
15     void speak()const{cout<<"Cat speak\n";}
16     void purr()const{cout<<"Cat purrs\n";}
17 };
18
19 int main(){
20     int size=3;
21     Mammal *array[size];
22     Mammal *ptr;
23     int choice;
24     for(int i=0;i<size;i++){
25         cout<<"1=cat; 2=mammal: ";
26         cin>>choice;
27         switch(choice){
28             case 1: ptr=new Cat; break;
29             default: ptr=new Mammal; break;
30         }
31         array[i]=ptr;
32     }
33     for(int i=0;i<size;i++){
34         array[i]->speak();
35         Cat *pRealCat=dynamic_cast<Cat *>(array[i]);
36         if(pRealCat) pRealCat->purr();
37         delete array[i];
38     }
39     return 0;
40 }
```

- Solution: Cast your base class pointer to your derived type
  - Using the "dynamic_cast" operator ensures that when you cast, you cast safely; base pointer is examined at runtime; if conversion is proper your new cat pointer is fine, else your new cat pointer will be pointing to "null"

```
1=cat; 2=mammal: 2
Mammal constructor
1=cat; 2=mammal: 1
Mammal constructor
Cat constructor
1=cat; 2=mammal: 2
Mammal constructor
Mammal speak
Mammal destructor
Cat speak
Cat purrs
Cat destructor
Mammal destructor
Mammal speak
Mammal destructor
```

Summary

- What did you learn?

References

- Slides are based on
  - Sams Teach Yourself C++ in 24 Hours (Chapter 10-11 and 16-18)