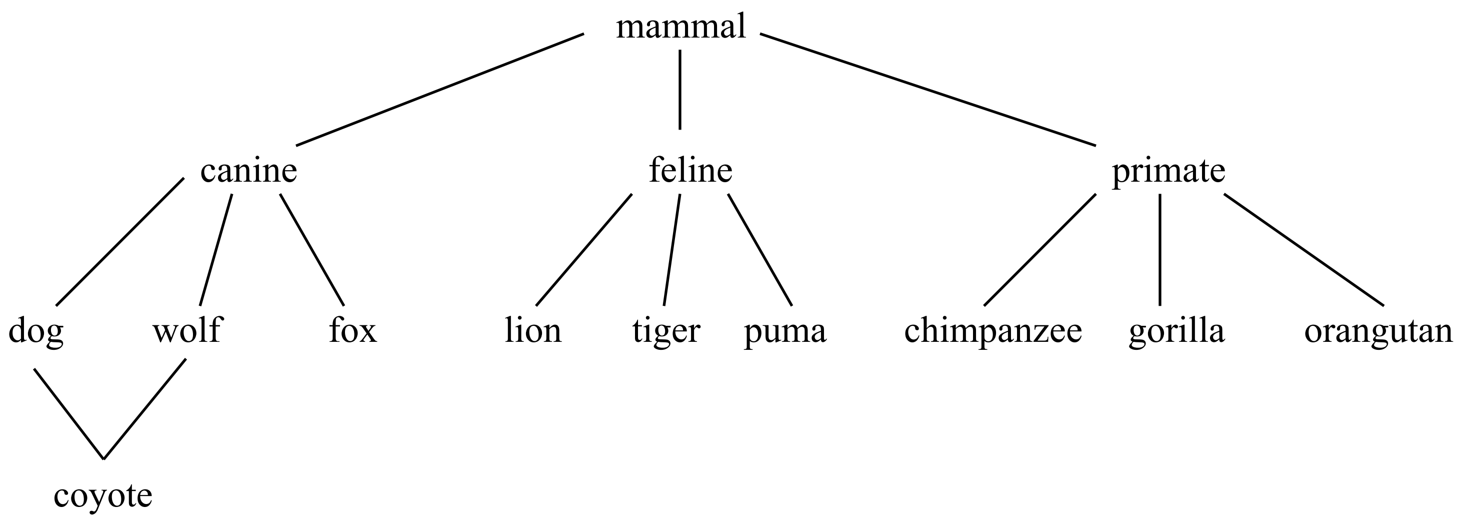
Derivation/Inheritance

Inheritance

- Inheritance is a relationship among classes where a subclass inherits the structure and behavior of its super-class.
  - Defines the “is a” or generalization/specialization hierarchy.
  - Structure: instance variables.
  - Behavior: instance methods.

Inheritance in C++

- C++ supports single and multiple inheritance



Class Derivation (Inheritance)

- In order to derive a class, the following two extensions to the class syntax are necessary
  - class heading is modified to allow a derivation list of classes from which to inherit members.
  - An additional class level, that of ***protected***, is provided. A protected class member behaves as a public member to a derived class

```
class Cat : public Animal
{
   protected:

   // data members

};
```

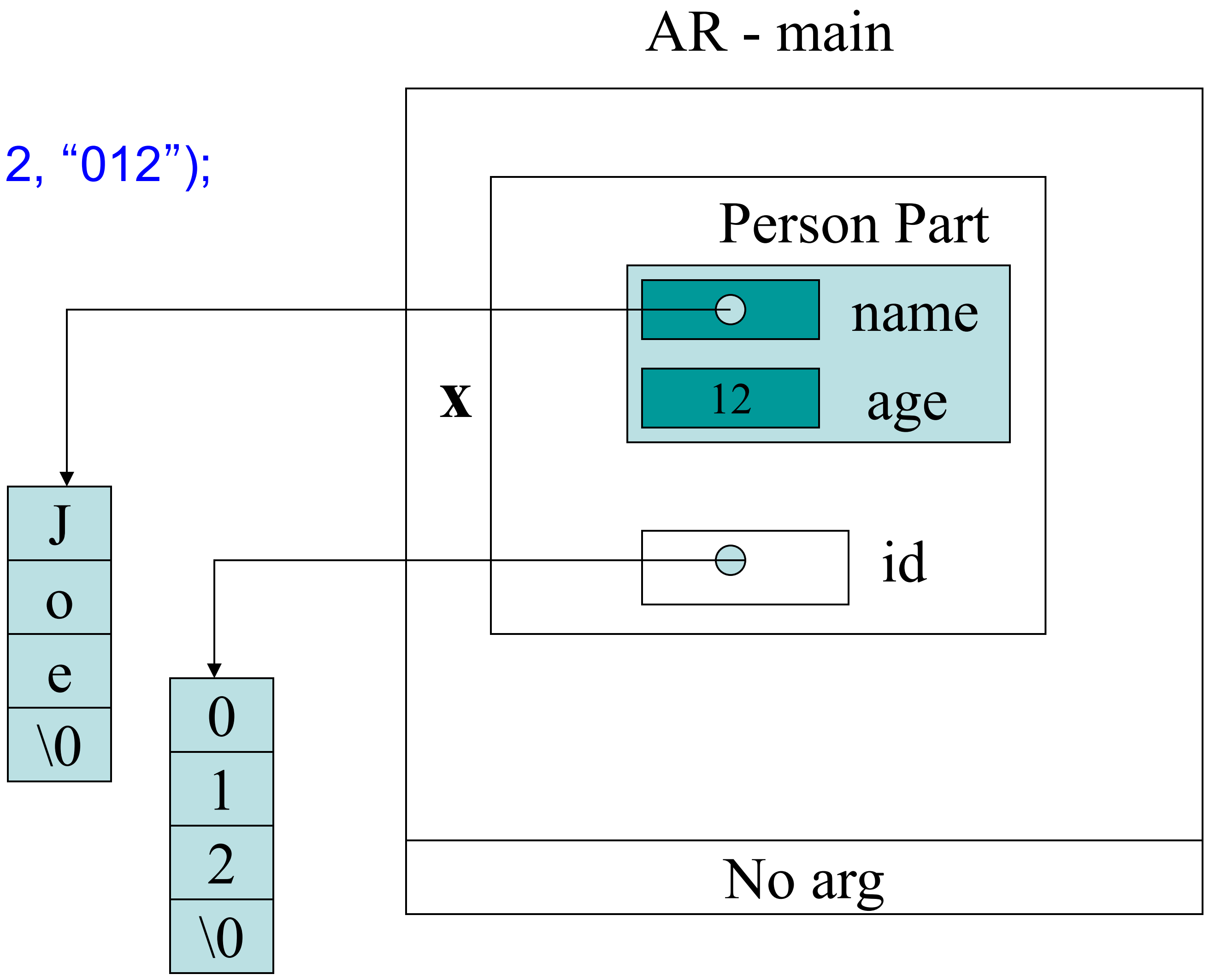
Class derivation - Example

```
class Person {
public:
  Person(char* n, int n)
  …
Protected:
  int age;
   char *name;
};
```

```
class Student: public Person
{
      public:
      Student(char* n, int a, char* i);
      …
      protected:
      char *id;
};
```

Example Continued

```
int main ()
{
   Student  x ("Joe", 12, "012");
    return 0;
}
```



Base Class Design

- Syntax for defining a base class is the same as an ordinary class with two exceptions:
  - Members intended to be inherited but not intended to be public are declared as ***protected*** members.
- Member functions whose implementation depends on representational details of subsequent derivations that are unknown at the time of the base class design are declared as *virtual functions*.

Base Class Design (Continued)

```
class Person {
    public:
      Person();
      virtual ~Person();
      virtual display();
      …
    protected:
      int age;
      char *name;
};
```

Inherited member access

- The derived class member functions can have access to inherited members directly or by using the the scope resolution operator:

```
void Student :: display() {
    cout << Person::name << age;
    }
```

In this example name also could be accessed directly without using scope resolution operator.

Inherited Member Access (Continued)

- In most cases, use of the class scope resolution operator is redundant. In two cases, however, using scope resolution operator is necessary:
  1. When an inherited member’s name is reused in the derived class.
  2. When two or more base classes define an inherited member with the same name.

Base Class Initialization

- Member initialization list is used to pass arguments to a base class constructor. The tag name of a base class is specified, followed by its argument list enclosed in parentheses.

```
class A {
int a;
public:
A(int x) {a = x;}
};
```

```
class B: public A{
      int b;
      public:
      B(int x, int y) : A(x){
      b = y;
       }
};
```

What does this line do?

Special Relationship between Base and Derived Class

- A derived class can be assigned to any of its public base classes without requiring an explicit cast.

  For example, consider class Student is derived from class Person and class Monitor is derived from class Student :

  ```
  Person x;
  Student y;
  Monitor z;
  x = y;                    // OK
  y =  (Student) x;                   // Needs cast
  x = z;                    // OK
  ```

- A derived class can be assigned to any of its public base classes without requiring an explicit cast. How is this feature related to polymorphism?

What is a ‘virtual Function’ & When Do Need It?

Virtual functions

- A virtual function is a special function invoked through a public base class reference or pointer; it is bound dynamically at run time.
- The instance invoked is determined by the class type of the actual object addressed by the pointer or reference.
- Resolution of a virtual function is transparent to the user

Virtual functions

- A virtual function is specified by prefacing a function declaration with keyword **virtual**.
- The class that declares a function as virtual must provide a definition for the function or must be declared as **pure virtual** function:
  - If definition is provided, serves as default instance for subsequent derived classes.
  - If pure virtual is declared, the class will be considered as abstract class. Means instances of that class cannot be created. The derived class from an abstract base class can define the function or will be also considered as an abstract class.

Using Virtual Functions

```
// animal.h
class Animal
{
  char name[20];
  public:
  // move is a pure virtual
  virtual void move() = 0;
  virtual  void display();
  void fun();
  // more functions
};
```

```
//  fish.hclass
Fish :public Animal
{
  char type[20];
  public:
  void display();
  void move();
};
```

```
// cat.h
class Cat :public Animal
{
  char type[20];
  public:
  void display();
  void move();
};
```

```
// bird.h
class Bird :public Animal
{
  char type[20];
  public:
  void display();
  void move();
};
```

```
// animal.cpp
#include <iostream>
using namespace std;

void Animal::display(){
    cout << "Animal";
 }

 // More functions go
       here
```

```
// fish.cpp
#include <iostream>
using namespace std;

 void Fish::display(){
   cout << "Fish";
 }

 void Fish::move() {
     cout << " Swimming";
 }
```

```
// cat.cpp
#include <iostream>
using namespace std;


void Cat::display(){
    cout << ”Cat";
  }

 void Cat::move() {
     cout << " Walking";
  }
```

```
// bird.cpp
#include <iostream>
using namespace std;

 void Bird::display(){
   cout << "Bird";
 }

 void Bird::move() {
     cout << " Flying";
 }
```

Virtual Functions

- The redefinition of a virtual function must match exactly the name, signature and the return type of the base class instance.
- Use of keyword virtual is optional.
- The virtual mechanism is handled implicitly by compiler.
- If redefinition of a virtual function does not match exactly, the function return type and signature, it is not handled as virtual. However, the subsequent class still can redefine a virtual function.

Virtual functions

- If class Fish needs to have an object, but you still don’t want to define function *display(),* you can define a null instance of display function;

```
class Fish: public Animal{
   char type[20];
   …
   public:
   display() {} // null function
}
```

A Good Reason to Declare a virtual Destructor

Why Virtual Destructor?

- Let’s have a look at the following example:

```
class A
{
    char * s1;
    public:
        A(int n)  { s1 = new char[n];}
       ~A() { delete [] s1;}
};
```

```
class B : public A
{
      char * s2;
    public:
        B(int n, int m):  A(n)  { s2 = new char[m];}
       ~B() { delete [] s2;}
};
```

```
    int main(void)  {
     A * p = new B(5, 6);
     delete p;
    }
```

Class Discussion:

- **What happens when we delete p, considering that pointer p is an A type?**
- **Answer will be discussed during the lecture**

What is a private or protected base class?

Public, protected and private base classes

- Public base class: The inherited members of a public base class maintain their access level within the derived class.
- Protected base class: The access level of public members of a protected base class will change to protected within the derived class.
- Private base class: The access level of public and protected members of a private base class will change to private within the derived class.

Example

```
class A {
  public: int x;
  protected: int y;
  private: int z;
  public: void fun();  // which data members are accessible
};

class B: protected A {
  public: int k;
  protected: int l;
  private: int m;
  public: void fun(); // which data members are accessible
};

class C: private B {
  public: int p;
  protected: int r;
  private: int s;
  public: void fun(); // which data members are accessible
};
```

```
class D: public C {
  public: int u;
  protected: int v;
  private: int w;
  public: void fun(); // which data members are accessible };
```