embedded objects

(XY objects are embedded in Planet and spaceship)

first of all: Copy constructors 1/6

- like the default constructor, the ***Copy constructor*** is member function that the compiler generates
- the purpose of the copy constructor is to ***make a new object of the same class***, from an existing object that is passed as an argument.
- An inline copy constructor for XY class looks like this:

```
XY( const XY& xy)
        {
        x = xy.x;
        y = xy.y;
        }
```

- the automatically generated copy constructor simply does member wise copy of all the object data
- for complex classes (memory allocating etc.)is good practice to write own copy constructor
- notation XY& tells - the compiler passes the address of the XY object as argument, not a copy of the object
- An use of the copy constructor is like that:

```
XY alpha(1.0, 2.0);
XY beta = alpha;
XY gamma(alpha); // same as the second op, but using copy constructor
```

- *Copy constructor are used also when parameter passing , where formal parameter substitution / initialization takes place*

```
void f(XY xy);
XY alpha(2.0, 3.0);
func(alpha); //a copy constr. is called to copy 'alpha' to the argument list
```

- *The same is the situation with returning values from a function*

Copy constructors 2/6

Example !!!! Let's have:

```
struct string{          char *p;
                        int size;                    //the size of the string
                        string(int sz) { p = new char[size = sz];}
                        ~string() {delete p;}
                };
```

Now:

```
void f()        {
                string s1(10);
                string s2(20);
                s1 = s2;                //assignment !!!
                }
```

One of the pointers is lost, the other is doubled ?! The destruction destructs one object (and the other!)

We must redefine:

```
struct string{          char *p;
                        int size;                    //the size of the string
                        string(int sz) { p = new char[size = sz];}
                        void operator = (string&)    //for assignment operations – see later slides
                        ~string() {delete p;}
                        };
```

And:

```
void string::operator= (string& a) {
                if(this == &a) return;
                delete p;               //current pointer 'p' exists
                p = new char[size = a.size];    // must new 'p'
                strcpy(p, a.p);         }       // now everything is OK
```

To be continued

Copy constructors 3/6

This done, another problem arises. Where? :

```
void f()    {
            string s1(10); string s2;
            s2 = s1;
            }
```

Everything is working, <u>but we have constructed 1 string and destruct 2 strings !!</u>
That's because we did not forbidden operation '= ' to work with not initialized objects !!

Every time, we are thinking about operation '=' to work with initialized objects !!

So, another operation is needed to work with 'in-moment' constructed objects.
We are redefining our string class:

```
struct string{      char *p;
                    int size;                          //the size of the string
                    string(int sz) { p = new char[size = sz];}
                    void operator = (string&)
                    ~string() {delete p;}
                    string(string&);        // now added new operation
                    };
```

And :

```
void string::string(string& a)
        { p = new char[size = asize];
          strcpy(p, a.p);
        }
```

<u>Operation not for assignment, but for initialization of new constructed object (Copy constructor)</u>

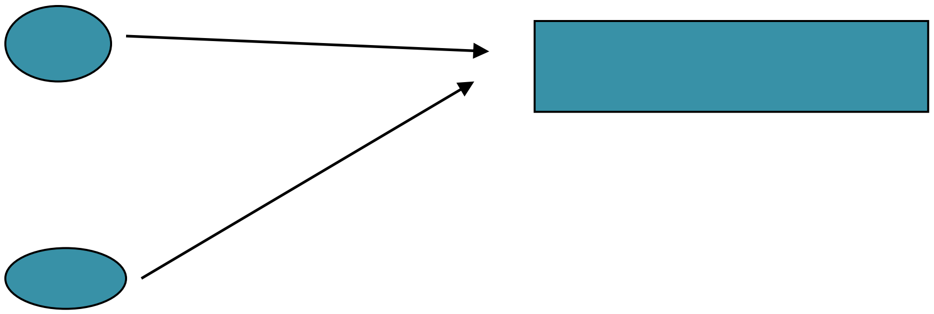
Generally speaking, we are including operation of type **X(X&)**

Copy constructors – second example 4/6

```
class PersonInfo
{
private:
   char *name;
   int age;

public:
   PersonInfo(char *n, int a)
      { name = new char[strlen(n) + 1];
        strcpy(name, n);
        age = a; }

   ~PersonInfo()
      { delete [] name; }

   const char *getName()
      { return name; }

   int getAge()
      { return age; }
};
```

Copy constructors 5/6

```
int main()
{
        PersonInfo person1("Molly McBride", 27);
        PersonInfo person2 = person1;

        cout << person1.getName() << endl;
        cout << person2.getName() << endl;
        return 0;
}
```



Copy constructors – second example with solution 6/6

```
class PersonInfo
{private:
   char *name;
   int age;

public:
   // Constructor
   PersonInfo(char *n, int a)
      { name = new char[strlen(n) + 1];
        strcpy(name, n);
        age = a; }

   // Copy Constructor
   PersonInfo(const PersonInfo &obj)
      { name = new char[strlen(obj.name) + 1];
        strcpy(name, obj.name);
        age = obj.age; }

   ~PersonInfo()
      { delete [] name; }

   const char *getName()
      { return name; }

   int getAge()
      { return age; }
};
```

Assignment operators

- It is like copy constructor except that it

  <u>*operates on an existing object rather than creating a new one*</u>

- the compiler generates default assignment operators and generates call to them
- if you were to write own assignment operator, it would look like:

```
const XY& operator=(const XY& xy)          // uses references
{
        x = xy.x;
        y = xy.y;
        return *this;        // returns XY reference
}
```

The result of assignment can be used only where 'const' parameter is specified

- using the operator is possible like that (because of returned XY reference):

```
xy1 = xy2 = XY(4.5, 5.0);
```

- Another use of assignment operator:

```
XY first(0.0,0.0);
XY second(2.0, 3.0);
first = second; // the first content is  overwritten
```

Assignment operators

```
class PersonInfo
{private:
   char *name;
   int age;

public:
   // Constructor
   PersonInfo(char *n, int a)
      { name = new char[strlen(n) + 1];
        strcpy(name, n);
        age = a; }

   // Copy Constructor
   PersonInfo(const PersonInfo &obj)
      { name = new char[strlen(obj.name) + 1];
        strcpy(name, obj.name);
        age = obj.age; }

   // Destructor
   ~PersonInfo()                { delete [] name; }

   // Accessor functions
   const char *getName()                  { return name; }

   int getAge()                 { return age; }

   // Overloaded = operator
   void operator=(const PersonInfo &right)
      { delete [] name;
        name = new char[strlen(right.name) + 1];
        strcpy(name, right.name);
        age = right.age; }
};
```

Assignment operators


```
// This program demonstrates the overloaded = operator.
#include "PersonInfo.h"

int main()
{
   // Create and initialize the jim object.
   PersonInfo jim("Jim Young", 27);

   // Create and initialize the bob object.
   PersonInfo bob("Bob Faraday", 32);

   // Creates a cloning object and initialize with jim.
   PersonInfo clone = jim;

   // Display the conents of the jim object.
   cout << "The jim Object contains: " << jim.getName();
   cout << ", " << jim.getAge() << endl;

   // Display the contents of the bob object.
   cout << "The bob Object contains: " << bob.getName();
   cout << ", " << bob.getAge() << endl;

   // Display the contents of the clone object.
   cout << "The clone Object contains: " << clone.getName();
   cout << ", " << clone.getAge() << endl << endl;
```

Program output:

The jim Object contains: Jim Young, 27
The bob Object contains: Bob Faraday, 32
The clone Object contains: Jim Young, 27

Assignment operators


```
// Assign bob to clone.
   cout << "Now the clone will change to bob and ";
   cout << "bob will change to jim.\n\n";
   clone = bob;   // Call overloaded = operator
   bob = jim;     // Call overloaded = operator

   // Display the contents of the jim object.
   cout << "The jim Object contains: " << jim.getName();
   cout << ", " << jim.getAge() << endl;

   // Display the contents of the bob object.
   cout << "The bob Object contains: " << bob.getName();
   cout << ", " << bob.getAge() << endl;

   // Display the contents of the clone object.
   cout << "The clone Object contains: " << clone.getName();
   cout << ", " << clone.getAge() << endl;

   return 0;
}
```

Program output:

```
Now the clone will change to bob and bob will change to jim

The jim Object contains: Jim Young, 27
The bob Object contains: Jim Young, 27
The clone Object contains: Bob Faraday, 32
```

Reference parameters (const vs. non-const)

- reference parameters are disguised pointer parameters. Useful if:
  - the function will use parameter to change a variable in the calling program. So the reference will be non-const
  - we want to avoid copying a large object into function call stack. So the reference will be const

```
void Show(const XY& xy)                    // global function with const reference parameter
{
    printf("x=%f, y= %f\n", xy.GeX(), xy.GetY());          //cannot change values
}
```

  - we can call ‘const’ parameters from declared as ‘const’ member functions.

How C++ references work

We have the following application code to construct an object of type planet:

```
XY current(1000.0, 2000.0);              //constructs current XY coordinate
XY prior(900.1, 1000.2);                 // construct prior XY coordinate
Planet Earth(current, prior, 2.7E+8);    // constructs planet object
```

How C++ references works

- ***Remember, we had the following class declarations for the objects used:***

```
class XY{
    public:
        double x,y;
        XY()        {x =0.0; y = 0.0;}                  //default
        XY(double a, double b;) {x = a; y = b;}         //explicit constructor

        XY(const XY& xy)                                // copy constructor
        {           x = xy.x;
                    y = xy.y;
        }
        const XY& operator=(const XY& xy)               //assignment operator
        { x = xy.x;
                    y = xy.y;
                    return *this;
        }
};
```

References at work


```
class Orbiter
{
    protected:
            XY m_current, m_prior, m_thrust;
            double m_mass;
    public:
            Orbiter(XY current, XY prior, double mass)
            {       m_current = current;
                    m_prior = prior;          //remember: data initialization!
                    m_mass = mass;            // we will change them later!
            }
            XY GetPosition() const;
            void Fly();
            virtual void Display() = 0;
};
```

References at work


```
class Planet : public Orbiter
{
    public:
        Planet  (XY current,    XY prior, double mass)
                                        :Orbiter(current,prior,mass){}

        void Display();
};
```

end of class declarations

Reference parameters


```
XY current(1000.0, 2000.0);
XY prior(900.1, 1000.2);
Planet earth(current, prior,2.7E+8);
```

What happen in practice when constructing objects in a program?

- ***With that declarations, the following sequence of XY method calls is necessary to make an object of type Planet (as in the application code we had):***

1. ***Explicit XY constructor creates 'current' & 'prior' objects in stack;***
2. ***The XY copy constructor copies the 'current' and 'prior' objects to the Planet constructor argument list.***
3. ***the XY copy constructor copies the 'current' and 'prior' objects from the Planet constructor's argument list to the Orbiter constructor's argument list** (see previous slide);*
4. ***the default XY constructor creates Orbiter's 'm_current' and 'm_prior' members and initializes them to (0,0);***
5. ***the XY assignment operator copies the 'current' and 'prior' objects from Orbiter constructor's argument list to the corresponding data members***

Reference parameters

- ***Let's rearrange the Orbiter and Planet connected with constructors code to improve the performance :***

```
class Orbiter
{   protected:
        double mass;
        XY m_prior, m_current, m_thrust;

    public:
        Orbiter  (XY& current,      XY& prior, double mass)
                : m_current(current),m_prior(prior),m_mass(mass){}

        const XY& GetPosition() const;
        void Fly();
        virtual void Display() = 0;
};
```

Reference parameters

```
class Planet : public Orbiter
{    public:
         Planet(XY& current, XY& prior, double mass)
                                        : Orbiter(current, prior, mass) {}
         void Display();
};
```

Remarks& improvements :

- now Orbiter and Planet constructors use XY references.
- Orbiter constructor is different :the initialization of data members differs.
- C++ allows syntax like m_mass(mass) even for built-in types
- now, instead of two calls to XY default constructor and two calls to the assignment operators (as in previous slide) the compiler generates 2 calls to XY copy constructor only (before m_mass(mass))
- all the statements after ':' including calls to the base class and constructors are executed before constructor body

Reference parameters

- ***For variety's sake – another syntax for creating Earth object:***

```
Planet earth( XY(222.0, 111.0), XY( 333.0, 444.0), 2.0e+5);
```

What happens?

1. ***So, temporary 'current' and 'prior' objects are constructed in argument list with XY explicit constructor***
2. ***The m_current and m_prior objects (parameters) are constructed/initialized with XY copy constructor, from the objects from step 1.Those objects were passed to the Orbiter constructor as references, thereby avoiding extra copy operations***

Returning references

- ***A function can return a reference (equivalent to returning a pointer)***

```
const double& XY::GetConstX() const {return x};
```

- ***So declared , the function returns a const reference to XY object and may be used on the right side of an assignment only. That is:***

```
my.GetConstX() = 1.0;                    // is wrong!!!
```

returning reference from a function

- ***mistake in C is the following:***

```
int *GetInt()
{
        int result = (int) (rand() / 1000);
        return &result;                 // don't do this!!
}
```

- ***the function returns a pointer to stack that will be used elsewhere after the function returns (the member variable is missing now) !!!***

- ***the equivalent C++ mistake:***

```
int& GetInt()
{
    int result = (int)(rand() / 1000);
    return result;
}
//the compiler is still returning a pointer to a temporary variable
```

Constructing embedded objects

II part

orbiter
Planet **Spaceship**

1. ***the compiler has the object declaration. So he knows the total memory needed for Spaceship object and allocates that memory***
2. ***all embedded objects (m_current, m_prior, m_thrust) are constructed***
3. ***the Orbiter constructor is called***
4. ***the m_orrientation embeded object is constucted***
5. ***the Spaceship constructor function is called***

It's the correct list for construction

- ***The class design and the syntax of Spaceship constructor determine exactly which constructors(default, explicit or copy) are called***

```
// This program demonstrates the order in which base and
// derived class constructors and destructors are called.
#include <iostream>
using namespace std;

// BaseClass declaration        *
class BaseClass
{
public:
   BaseClass()  // Constructor
      { cout << "This is the BaseClass constructor.\n"; }

   ~BaseClass() // Destructor
      { cout << "This is the BaseClass destructor.\n"; }
};

// DerivedClass declaration      *
class DerivedClass : public BaseClass
{
public:
   DerivedClass()  // Constructor
      { cout << "This is the DerivedClass constructor.\n"; }

   ~DerivedClass()  // Destructor
      { cout << "This is the DerivedClass destructor.\n"; }
};
```

```
//*********************************
// main function                  *
//*********************************

int main()
{
   cout << "We will now define a DerivedClass object.\n";

   DerivedClass object;

   cout << "The program is now going to end.\n";
   return 0;
}
```

Program output:

We will now define a DerivedClass object

This is the BaseClass constructor
This is the DerivedClass constructor
The program is now going to end
This is the DerivedClass destructor
This is the BaseClass destructor

Destructing embedded objects

II part

let Spaceship is to be destroyed:

he is a derived from Orbiter class and has embedded objects (like XY) defined both in base class and in derived class. So:

1. spaceship destructor is called
2. m_orientation embedded object is destroyed
3. Orbiter destructor is called
4. m_current, m_prior and mass embedded objects are destroyed
5. the memory for Spaceship is freed

remember mark:

```
class SpaceShip : public Orbiter
{private:        double m_fuel; XY m_orientation;
public:
  SpaceShip( XY current, XY prior, XY thrust, double mass, double fuel, XY orientation)
                                                   : Orbiter(current, prior, mass)
```

more about destruction

•destructors are not inherited. The compiler generates a <u>default destructor</u> for each class if you do not explicitly write one. That derived class destructor always calls its base class destructor. If a code is missing for derived class destructor, only destruction of base class members will complete.

The destruction of derived class will be incomplete in this way.

•If in the base class the destructor is declared as virtual:

```
virtual ~Orbiter() {}
```

the compiler generated default for destructor for the child class SpaceShip in the example, will first destroy all elements owned by SpaceShip and then calls the Orbiter destructor

Example: *let's try without virtual destructors*:

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

// Animal is a base class.
class Animal
{
public:
  // Constructor
  Animal()
    { cout << "Animal constructor executing.\n"; }

  // Destructor
  ~Animal()
    { cout << "Animal destructor executing.\n"; }
};

// The Dog class is derived from Animal
class Dog : public Animal
{
public:
  // Constructor
  Dog() : Animal()
    { cout << "Dog constructor executing.\n"; }

  // Destructor
  ~Dog()
    { cout << "Dog destructor executing.\n"; }
};
```

```
//*********************************************
// main function                             *
//*********************************************

int main()
{
   // Create a Dog object, referenced by an
   // Animal pointer.
   Animal *myAnimal = new Dog;

   // Delete the dog object.
   delete myAnimal;
   return 0;
}
```

Program output:

Animal constructor executing
Dog constructor executing
Animal destructor executing

To fix the previous problem: ***let's try with virtual destructors*:**

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

// Animal is a base class.
class Animal
{
public:
  // Constructor
  Animal()
    { cout << "Animal constructor executing.\n"; }

  // Destructor
  virtual ~Animal()
    { cout << "Animal destructor executing.\n"; }
};

// The Dog class is derived from Animal
class Dog : public Animal
{
public:
  // Constructor
  Dog() : Animal()
    { cout << "Dog constructor executing.\n"; }

  // Destructor
  ~Dog()
    { cout << "Dog destructor executing.\n"; }
};
```

```
//*************************************************
// main function                              *
//*************************************************

int main()
{
   // Create a Dog object, referenced by an
   // Animal pointer.
   Animal *myAnimal = new Dog;

   // Delete the dog object.
   delete myAnimal;
   return 0;
}
```

Program output:

Animal constructor executing
Dog constructor executing
Dog destructor executing
Animal destructor executing

Virtual destructors- again

- The default destructor of derived class always calls its base-class destructor.
- suppose you have a pointer to an object, derived from Orbiter and you want to destroy it.

```
Orbiter* pAny = new Spaceship(current, prior, thrust, mass, fuel, orientation);
…
delete pAny;
```

pAny is of type Orbiter*. So, only object elements specified in Orbiter class will be destroyed The Spaceship object's deletion would be incomplete: the destructor for XY object m_orientation would not be called.

How to solve the problem:

virtual ~Orbiter() {}

Now you don't need any code or declarations for derived class destructors unless you are not satisfied with the compiler-generated defaults.
For the previous example now:

delete pAny;

calls the proper derived-class destructor (for Spaceship), which first destroys all elements of spaceship and then calls the Orbiter destructor . OK!!!