CS319: Scientific Computing (with C++)

More on inheritance; Wrap-up

Week 12: **9am** and **4pm**, 29 March 2017

Annotated slide for sections 1-3.

const

There are many aspects of the C++ language that we have not covered in this short course: there is an expectation that you will read beyond these notes.

But before we finish we'll study *interitance* in a little more detail. But first we'll consider the topic of **constants**.

A expression is constant it is value never changes. We are familiar with **literal consants**:

Eg 31 (int), 31.0 (float), "hello" (string constant), 'x' (char constant)

C++ also has three **keyword literals**: `true`, `false` and `nullptr`.

type bool

pointer.

double *p = nullptr.

const

We can also define our own constants, by putting the keywork `const` before a "variable" declaration:

modifier.

01const.cpp

```
   const double Pi=3.14159;
16 std::cout << "Pi=" << Pi << std::endl;
   
18 // Now change Pi
   Pi = 3.14; // This will not (and should not)  work
20 
   std::cout << "Now Pi=" << Pi << std::endl;
```

Now, if we compile:

```
01const.cpp: In function ‘int main()’:
01const.cpp:20:6: error: assignment of read-only variable ‘Pi’
   Pi = 3.14;
      ^
```

const

the alternative is the usual read/write variable.

const functions

We can add the `const` modifier to any data type, including a `class` that we define ourselves.

However, that creates issues... consider the following code, adpated from `Week11/02StaticVarInClass.cpp`

02constFunction.cpp

```
12  class Student
    {
14  private:
      std::string name;
16    static std::string university;
    public:
18    Student(std::string n) {name=n;};
      void set_name(std::string n) {name=n;};
20    void set_university(std::string u) {university=u;};
      std::string get_name(void) const {return(name);};
22    std::string get_university(void) const {return(university);};
    };
```

new constructor

Tells compiler that a const object can call these.

If we declare student A("Ann") const student B("Bob")

Then can't call B.set_name(); But can call B.get_name()

const function parameters

When writing a function header, we can designate some of the parameters as `const` meaning that the **copy** of the parameter that is passed cannot be changed in the function.

There are some times when this is necessary. Consider the following example...

02constParam.cpp

```
12  class Number
    {
    private:
14    double x;
    public:
16    Number() {x=0.0;};
      Number(const Number &copy) {x=copy.x;}; // notice const
18    void set_x(double X) {x=X;};
      double get_x(void) {return(x);};
20    Number operator+(Number &b)
        {
22        Number sum;
          sum.x = x + b.x;
24        return(sum);
        }
26  };
```

02constParam.cpp

```
28
int main(void)
30 {
   Number a;
32 a.set_x(121.234);
   Number b(a+a);
34 std::cout << "a.x= \t" << a.get_x() <<     std::endl;
   std::cout << "b.x= \t" << b.get_x() <<   std::endl;
36 return(0);
```

Why this works...

Here we first compute a+a,
This is done as a local const and
then passed to the copy constructor.

Recall... Inheritance

Last week, we introduced the concept of **inheritance**.

Inheritance

Given a class, construct from it a new class that has the properties of the original class, plus some new ones.

- The original class is called the **base**, **parent** or **super** class.
- The new one, is called the **derived**, **child** or **sub** class.

The derived class will inherit

- All the data members of the base class, though it may not be able to access them directly,
- The ordinary function members, but not constructors, destructors, or assignment operators.

Recall... Inheritance

Some programming/syntax aspects:

- The syntax for defining a derived class

```
class Derived : public Base
```

  Here `public` means that elements that were public to the base class are publicly accessible to the derived class. (This is the most typical case.)

```
class Derived : private Base
```

  Here `private` means everything remains private to the base class, and so can't be access by the derived class. (Not very useful).

When we first studied classes in C++, the only **access specifiers** we had were `private` and `public`.

Then we added **protected:**. These members act the same as private ones, except that they are directly accessible within a **derived** class.

Replacing members

Suppose, for example, that a base class has a method `void PrintData(void)`. It is possible to make a derived class that also contains a (new) method call `void PrintData(void)`.

The version of `PrintData(void)` from the derived class will superceed the one from the base class.

However, the original one does not disappear, it can still be called, but you need to use the base class name and scope resolution operator.

Note: If the base and derived classes have functions with the same names, but different **"signatures"**, then the function is overloaded, not replaced.

Replacing members

01Replacement.cpp

04

```
class Point {
protected:
  int x;
public:
  Point(int X=0) {x=X;};
  int GetX(void) {return(x);};
  void SetX(int X) {x=X;};
  void PrintData(void);
};

// The Point class has a method called PrintData()
void Point::PrintData(void)
{
  cout << "(" << x << ")\n";
}
```

Replacing members

01Replacement.cpp

```
class TwoPoint : public Point {
private:
  int y;
public:
  TwoPoint(int X=0, int Y=0) {x=X; y=Y;};
  int GetY(void) {return(y);};
  void SetY(int Y) {y=Y;};
  void PrintData(void);
};

// The TwoPoint class also has a method
// called PrintData()
void TwoPoint::PrintData(void)
{
  cout << "(" << x << "," << y << ")\n";
}
```

same signature (name + org list) as Point::PrintData().

Replacing members

```
int main(void)
{
  TwoPoint d;

  d.SetX(1); d.SetY(2);
  cout << "Calling the TwoPoint version of d.PrintData(): d=";
  d.PrintData();

  cout << "Calling the Point version of d.PrintData(): d=";
  d.Point::PrintData();
  return(0);
}
```

we parse d.Point::PrintData as d (Point::PrintData).

There are other times when we need to take care which of two functions that have the same *signature* and belonging the base and derived class is called, and need to declare one of them as `virtual`, but that is beyond the scope of this class...

Finished here 10am