Objects and Classes

Saurav Samantaray

Department of Mathematics

Indian Institute of Technology Madras

August 20, 2024

C++ objects can represent: variables of a user-defined data type

English Measurement Class

- Create a class that can be used to store the measurement of certain distances.
- the units of measurement is British.
- what all would be the data?
- how to initialise an object i.e. assign data to a member of the class?
- how to access the data from an object of the class?

```
// englobj.cpp
// objects using English measurements
#include <iostream>
using namespace std;
class Distance
  {
  private:
    int feet;
    float inches;
  public:
//English Distance class
void setdist(int ft, float in)  //set Distance to args
  { feet = ft; inches = in; }
void getdist()              //get length from user
  {
  cout << "\nEnter feet: ";
  cin >> feet;
  cout << "Enter inches: ";
  cin >> inches;
  }
```

C++ Objects as Data Types

```
void showdist()          //display distance
  { cout << feet << " \t " << inches <<" \t " <<endl; }
};
//////////////////////////////////////////////////////
int main()
{
Distance dist1, dist2;  //define two lengths

dist1.setdist(11, 6.25); //set dist1
dist2.getdist();  //get dist2 from user

                         //display lengths
  cout << " \ndist1 = ";  dist1.showdist();
  cout << " \ndist2 = ";  dist2.showdist();
  cout << endl;
  return 0;
  }
```

- The `ENGLOBJ` example shows two ways that member functions can be used to give values to the data items in an object.
- Sometimes, however, it's convenient if an object can initialise itself when it's first created,
- without requiring a separate call to a member function.

Example

- We will create a class of objects that might be useful as a general-purpose programming element.
- A *counter* is a variable that counts things.
- Maybe it counts
  $->$ file accesses,
  $->$ or the number of times the user presses the Enter key,
  $->$ or the number of customers entering a bank.
- Each time such an event takes place, the counter is incremented (1 is added to it).
- The counter can also be accessed to find the current count.
- Let's assume that this counter is important in the program and must be accessed by many different functions.
- In procedural languages such as C, a counter would probably be implemented as a global variable.
- However, global variables complicate the program's design and may be modified accidentally.

This example, `COUNTER`, provides a counter variable that can be modified only through its member functions.

```
// counter.cpp
// object represents a counter variable
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter
{
private:
  unsigned int count;  //count
public:
  void set_count (int i)   // set count = i;
    {count = i;}
  void inc_count()  //increment count
    { count++; }
  int get_count()  //return count
    { return count; }
  };
```

- The Counter class has one data member: count, of type `unsigned int` (since the count is always positive).
- It has three member functions:
  $->$ `set_count`, which set the value of count;
  $->$ `inc_count()`, which adds 1 to `count`;
  $->$ and `get_count()`, which returns the current value of `count`.

Automatic Initialisation

- Most counts start at 0.
- When an object of type `Counter` is first created, we want its `count` to be initialised to 0
- one could use the `set_count()` function to do this, and call it with an argument of 0,
- or we could provide a `zero_count()` function, which would always set count to 0.

- A function would need to be executed every time we created a `Counter` object.
- `Counter c1; //every time we do this,`
  `c1.zero_count(); //we must do this too`
- This is mistake prone, because the programmer may forget to initialise the object after creating it.
- It's more reliable and convenient, especially when there are a great many objects of a given class, to cause each object to initialise itself when it's created.
- Automatic initialisation can be carried out using a special member function called a *constructor*.
- A **constructor** is a member function that is executed automatically whenever an object is created.

Automatic Initialisation in COUNTER Class

```
// counter.cpp
// object represents a counter variable
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter
{
private:
  unsigned int count;  //count
public:
  Counter() : count(0)  //constructor
    { /*empty body*/ }
  void inc_count()  //increment count
    { count++; }
  int get_count()  //return count
    { return count; }
  };
```

- Now, in the Counter class, the constructor `Counter()` does this.
- This function is called automatically whenever a new object of type `Counter` is created.
- Thus in `main()` the statement `Counter c1, c2;` creates two objects of type Counter.
- As each is created, its constructor, `Counter()`, is executed.
- This function sets the count variable to 0.
- So the effect of this single statement is to not only create two objects, but also to initialize their count variables to 0.

- There are some unusual aspects of constructor functions.
- First, they have exactly the same name (Counter in this example) as the class of which they are members.
- This is one way the compiler knows they are constructors. (it is no accident)
- Second, no return type is used for constructors. Why not?
- Since the constructor is called automatically by the system,
- there's no program for it to return anything to; a return value wouldn't make sense.
- This is the second way the compiler knows they are constructors.

Initializer List

- In the Counter class the constructor must initialise the count member to 0.
- One might think that this would be done in the constructor's function body, like this:

```
count()
{ count = 0; }
```

- However, this is not the preferred approach (although it does work).
- Here's how one should initialise a data member:

```
Counter() : count(0) {}
```

- The initialisation takes place following the member function declarator but before the function body.
- It's preceded by a colon.
- The value is placed in parentheses following the member data.

- If multiple members must be initialised, they're separated by commas.
- The result is the initialiser list (sometimes called by other names, such as the member-initialisation list).
- `someClass():m1(7), m2(33), m2(4) ← initialiser list`
  `{ }`
- Why not initialise members in the body of the constructor ?
- The reasons are complex,
- members initialised in the initialiser list are given a value before the constructor even starts to execute.
- This is important in some situations.
- For example, the initialiser list is the only way to initialise const member data and references.

- For a proof that the constructor is operating as advertised, one can rewrite the constructor to print a message when it executes.

```
Counter() :  count(0)
{ cout << '' I'm the constructor '' << endl
; }
```

- Constructors are pretty amazing when you think about it.
- If you define an `int`, for example, somewhere there's a constructor allocating four bytes of memory for it.

- It's convenient to be able to give variables of type Distance a value when they are first created.
- That is, we would like to use definitions like
```
Distance width(5, 6.25);
```
- which defines an object, `width`, and simultaneously initialises it to a value of 5 for feet and 6.25 for inches.
- To do this we write a constructor like this:
```
Distance(int ft, float in) :  feet(ft),
inches(in)
{ }
```
- This sets the member data feet and inches to whatever values are passed as arguments to the constructor. So far so good.
- However, we also want to define variables of type Distance without initialising them
```
Distance dist1, dist2;
```

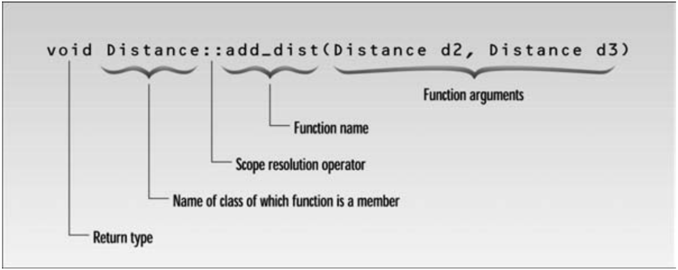
- We had programs with no constructor, but our definitions worked just fine.
- How could they work without a constructor?
- Because an implicit no-argument constructor is built into the program automatically by the compiler,
- it's this constructor that created the objects, even though we didn't define it in the class.
- This no-argument constructor is called the *default constructor*.
- Often we want to initialise data members in the default (no-argument) constructor as well.
- If we let the default constructor do it, we don't really know what values the data members may be given.
- If we care what values they may be given, we need to explicitly define the constructor.

- We can overload the *default constructor* infact we already did it
  ```
  Distance() :  feet(0), inches(0.0)
  //default constructor
  { } // no function body, doesn't do
  anything
  ```
- The data members are initialised to constant values, in this case the integer value 0 and the float value 0.0, for feet and inches respectively.
- Now we can use objects initialised with the no-argument constructor and be confident that they represent no distance (0 feet plus 0.0 inches) rather than some arbitrary value.
- Since there are now two explicit constructors with the same name, Distance(), we say the constructor is *overloaded*.
- Which of the two constructors is executed when an object is created depends on how many arguments are used in the definition

- So far we had seen member functions that were defined inside the class definition.
- The member function, add_dist(), is not defined within the Distance class definition.
- It is only declared inside the class, with the statement `void add_dist( Distance, Distance );`
- This tells the compiler that this function is a member of the class but that it will be defined outside the class declaration, someplace else in the listing.

```
void Distance::add_dist(Distance d2, Distance d3)
{
  inches = d2.inches + d3.inches; //add the inches
  feet = 0;                      //(for possible carry)
  if(inches >= 12.0)             //if total exceeds 12.0,
  {                              //then decrease inches
    inches -= 12.0;              //by 12.0 and
    feet++;                      //increase feet
  }                              //by 1
  feet += d2.feet + d3.feet;     //add the feet
}
```

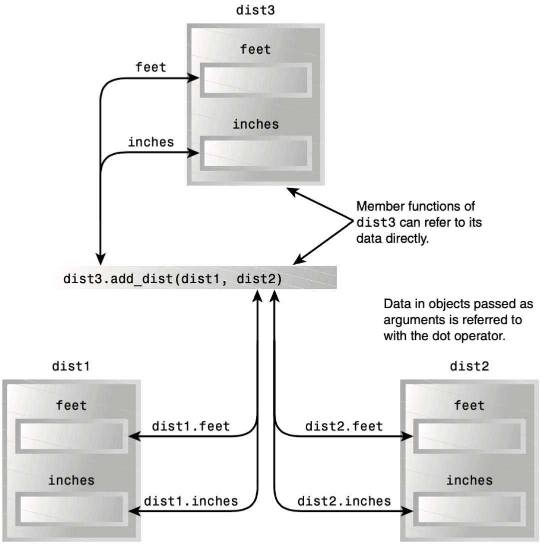
- The declarator in this definition contains some unfamiliar syntax.
- The function name, add_dist(), is preceded by the class name, Distance, and a new symbol—the double colon (::).
- This symbol is called the scope resolution operator.
- It is a way of specifying what class something is associated with.
- In this situation, Distance::add_dist() means "the add_dist() member function of the Distance class."



- The two distances to be added, `dist1` and `dist2`, are supplied as arguments to `add_dist()`.
- The syntax for arguments that are objects is the same as that for arguments that are simple data types such as `int`:
- The object name is supplied as the argument
- Close examination of add_dist() emphasizes some important truths about member functions.
- A member function is always given access to the object for which it was called: the object connected to it with the dot operator.
- But it may be able to access other objects.
- what objects can add_dist() access?
  `dist3.add_dist(dist1, dist2);`
- it can also access `dist1` and `dist2`, because they are supplied as arguments.

Objects as Arguments

- the member function always has access to the data of the object, even though it is not supplied as an argument
- “Execute the add_dist() member function of dist3.” When the variables feet and inches are referred to within this function, they refer to dist3.feet and dist3.inches.
- To summarize, every call to a member function is associated with a particular object (unless it’s a static function; we’ll get to that later).
- Using the member names alone (feet and inches), the function has direct access to all the members, whether private or public, of that object.
- It also has indirect access, using the object name and the member name, connected with the dot operator (dist1.inches or dist2.feet) to other objects of the same class that are passed as arguments.


dist3
feet
inches
feet
inches
Member functions of dist3 can refer to its data directly.
dist3.add_dist(dist1, dist2)
Data in objects passed as arguments is referred to with the dot operator.
dist1
feet
inches
dist1.feet
dist1.inches
dist2
feet
inches
dist2.feet
dist2.inches