Classes and Data Abstraction

Topic 5

Introduction

- Object-oriented programming (OOP)
  - Encapsulates data (attributes) and functions (behavior) into packages called classes
  - The data and functions of a class are intimately tied together.
  - A class is like a blue Print,
  - With the help of blue print builder can make a house, out of a class a programmer can create an object.

- One blue print can be reused many times to make many houses, similarly one class can be reused many times to make many objects of the same class.
- Procedural language programming tends to be action oriented, c++ programming is object oriented.
- The unit of the programming in c is the function where in C++ is the class
- Classes are also referred to as programmer defined types.

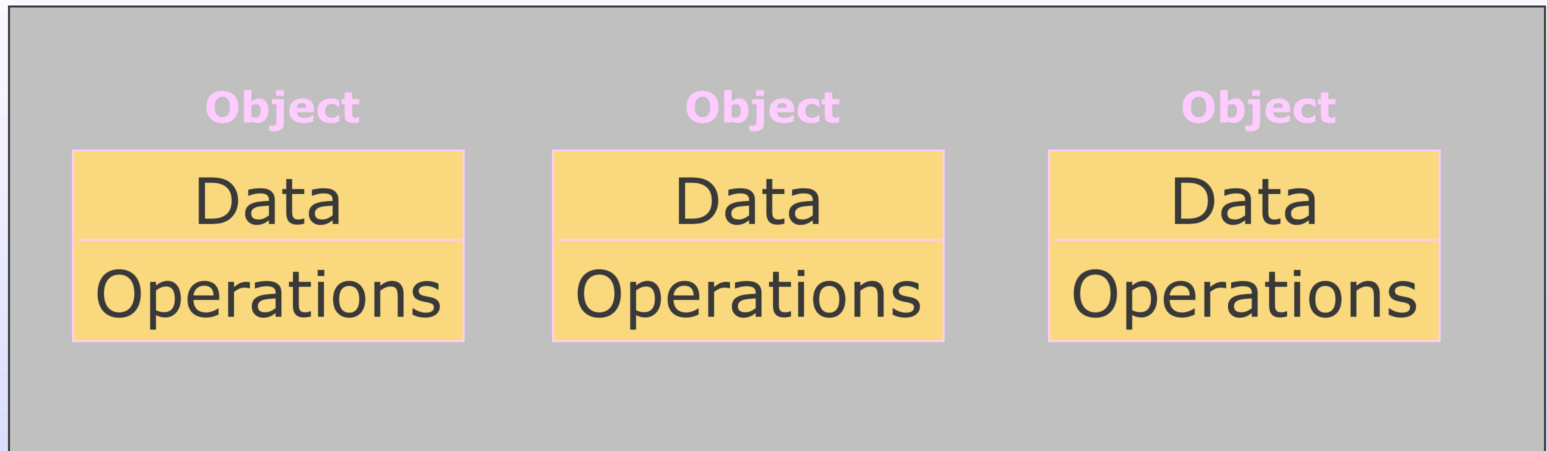
Introduction

- Each class contains data as well as the set of functions that manipulate the data.
- The data components of a class are called data members. The function components of a class are called member function.

- Information hiding
  - Class objects communicate across well-defined interfaces
  - Implementation details hidden within classes themselves
- User-defined (programmer-defined) types: classes
  - Data (data members)
  - Functions (member functions or methods)
  - Similar to blueprints – reusable
  - Class instance: object

What are Classes?

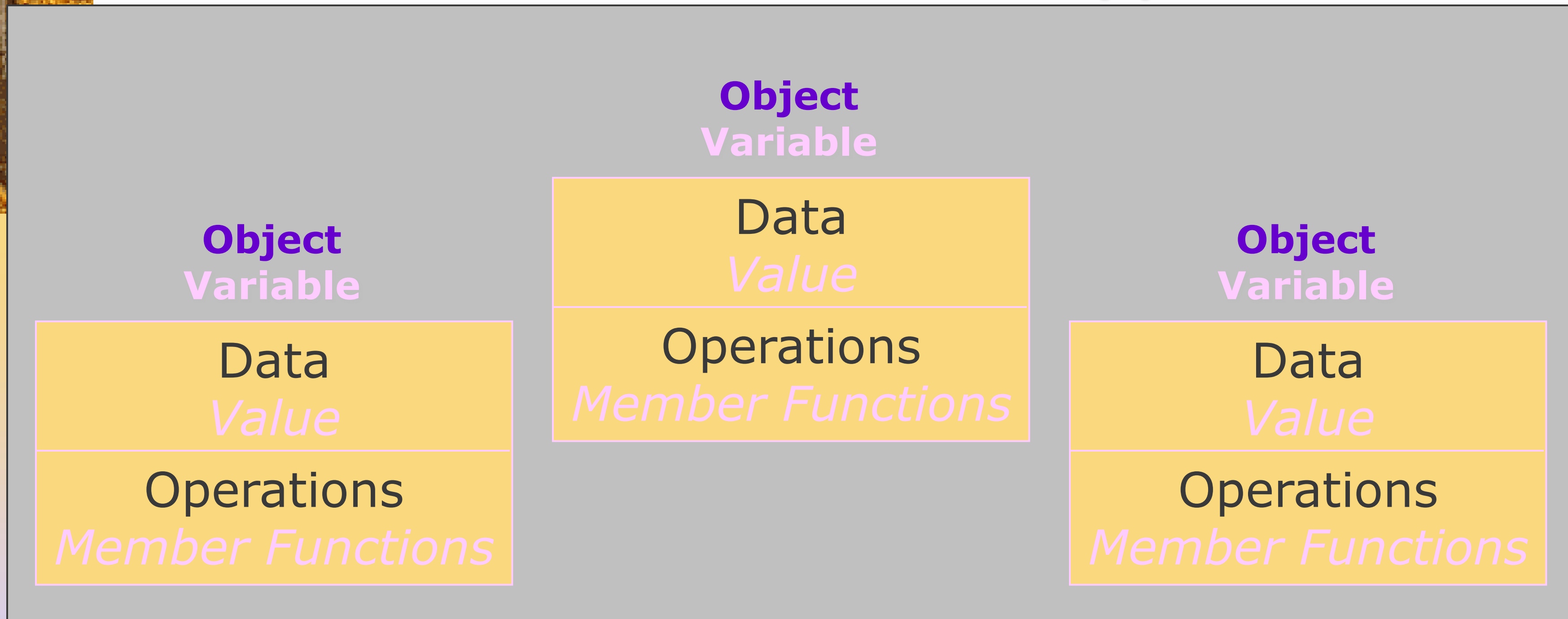
Class



What are Classes?

Class

User Defined Data Type



- A class is a *data type*
- You can use classes in the same way you use predefined data types (int, char, etc.)
- Defining your class the *right way* is important for it to behave like predefined data types → Abstract Data Type (ADT)
- An ADT is a user-defined data type that is well behaved as the predefined data types

Structures

- A data structure that can be used to store related data items with different types.
- The individual components of a struct is called a member.

Students

ID	Name	Major
1111	Nora	CS
2222	Sara	IS
3333	Mona	CS

- Student: ID variable
- Student: Name variable
- Student: Major variable

- Student
  - ID
  - Name
  - Major

Structures

Think of a structure as an object without any member functions

Object
Variable

Data
Value

~~Operations~~
~~*Member Functions*~~

Here, we'll have values of <u>different</u> data types that we would like to treat as a *single item*.

Structures

- How do I….
  - Define a structure?
  - Use a structure?

- Student
  - ID
  - Name
  - Major

```
struct Student
{
        int   id;
        char  name[10];
        char  major[2];

};
```

- Syntax:

```
struct Structure_Tag
{
    Type1        Member_Variable1;
    Type2        Member_Variable2;

    Typen        Member_Variablen;
};
```

- Using Structures
  - Declare:

    StudentRecord Student1, Student2;
  - Assignment: Student1 = Student2;
    - Student1.id = Student2.id;
    - Student1.grade = Student2.grade;
  - Read: cin >> Student1.id;
  - Write: cout << Student1.id;
  - Initialize: Student1 = {666,’A’}

- Syntax: Structure_Variable_Name.Member_Variable_Name

  Dot Operator

- Example:

```
struct StudentRecord
{
        int id;
        char grade;
};
int main ()
{
StudentRecord   Student1;
Student1.id                    = 555;
Student1.grade  = 'B';
cout<< Student1.id<< ', '<< Student1.grade<<endl;
}
```

Structures

- Two or more structure types may use the same member names

```
struct FertilizerStock
{
        double quantity;
        double nitrogen_content;
};
```

```
FertilizerStock  Item1;
```

↓

```
Item1.quantity
```

```
struct CropYield
{
        int quantity;
        double size;
};
```

```
CropYield      Apples;
```

↓

```
Apples.quantity
```

Structures

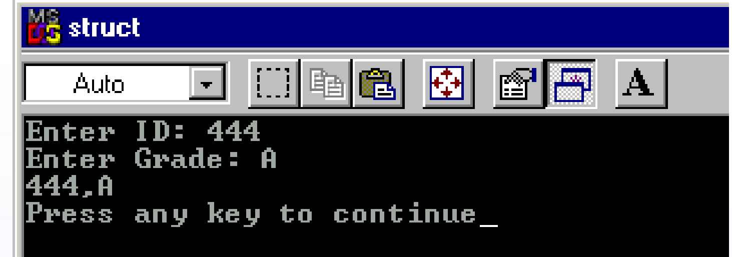
- Structures within structures (nested)

```
struct Date
{
        int month;
        int day;
        int year;
};
```

```
struct Employee
{
        int     id;
        Date birthday;
};
```

```
Employee person1;
cout<< person1.birthday.year;
```

Structures




```
#include <iostream>
struct StudentRecord
{                   int id;
                    char grade;
};
StudentRecord Get_Data   (StudentRecord          in_student);
int main ()
{          using namespace std;
           StudentRecord          Student1;
           Student1 = Get_Data (Student1);
           cout<< Student1.id<< ","<<Student1.grade<< endl;
           return 0;
}
StudentRecord Get_Data (StudentRecord in_student)
{          using namespace std;
           cout<<"Enter ID: ";        cin>> in_student.id;
           cout<<"Enter Grade: ";     cin>> in_student.grade;
           return  (in_student);
}
```

```
1   // Fig. 6.1: fig06_01.cpp
2   // Create a structure, set its members, and print it.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include <iomanip>
9
10  using std::setfill;
11  using std::setw;
12
13  // structure definition
14  struct Time {
15     int hour;     // 0-23 (24-hour clock format)
16     int minute;   // 0-59
17     int second;   // 0-59
18
19  }; // end struct Time
20
21  void printUniversal( const Time & );  // prototype
22  void printStandard( const Time & );   // prototype
23
```

Define structure type **`Time`** with three integer members.

Pass references to constant **`Time`** objects to eliminate copying overhead.

```
24  int main()
25  {
26     Time dinnerTime;          // variable of new type Time
27
28     dinnerTime.hour = 18;     // set hour member of dinnerTime
29     dinnerTime.minute = 30;   // set minute member of dinnerTime
30     dinnerTime.second = 0;    // set second member of dinnerTime
31
32     cout << "Dinner will be held at ";
33     printUniversal( dinnerTime );
34     cout << " universal time,\nwhich is ";
35     printStandard( dinnerTime );
36     cout << " standard time.\n";
37
38     dinnerTime.hour = 29;     // set hour to invalid value
39     dinnerTime.minute = 73;   // set minute to invalid value
40
41     cout << "\nTime with invalid values: ";
42     printUniversal( dinnerTime );
43     cout << endl;
44
45     return 0;
46
47  } // end main
48
```

Use dot operator to initialize structure members.

Direct access to data allows assignment of bad values.

```
49  // print time in universal-time format
50  void printUniversal( const Time &t )
51  {
52     cout << setfill( '0' ) << setw( 2 ) << t.hour << ":"
53          << setw( 2 ) << t.minute << ":"
54          << setw( 2 ) << t.second;
55
56  } // end function printUniversal
57
58  // print time in standard-time format
59  void printStandard( const Time &t )
60  {
61     cout << ( ( t.hour == 0 || t.hour == 12 ) ?
62             12 : t.hour % 12 ) << ":" << setfill( '0' )
63          << setw( 2 ) << t.minute << ":"
```

fig06_01.cpp
(3 of 3)

Use parameterized stream manipulator **setfill**.

Use dot operator to access data members.

```
Dinner will be held at 18:30:00 universal time,
which is 6:30:00 PM standard time.

Time with invalid values: 29:73:00
```