Structures

W2-L3

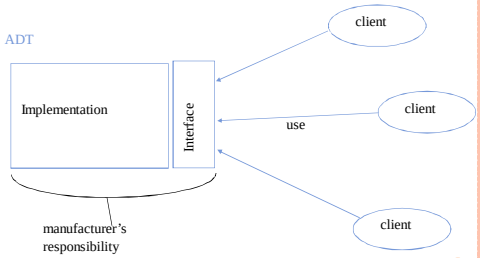
Contents

- What is Data Abstraction?
- C++: Classes and Data Abstraction
- Structure and classes
- Implementing a User-Defined Type Time with a Struct
- Implementing a Time Abstract Data Type with a Class
- Classes as User-Defined Types
- Enumeration

1. What is Data Abstraction?

- Abstract Data Types (ADTs)
  - type implementation & operations
  - hidden implementation
- types are central to problem solving
- a weapon against complexity
- built-in *and* user-defined types are ADTs

Clients and Manufacturers



Benefits

- Manufacturer Benefits:
  - easy to modify, maintain
  - profitable
  - reusable
- Client Benefits:
  - simple to use, understand
  - familiar
  - cheap
  - component-based

How Well are ADTs Supported in C?

- Does C enforce the use of the ADTs interface and the hiding of its implementation?
- *No*

C++

- C++ is a superset of C, which has added features to support object-oriented programming.
- C++ supports classes.
  - things very like ADTs

2. C++: Classes and Data Abstraction

- C++ supports Object-Oriented Programming (OOP).
- OOP models real-world objects with software counterparts.
- OOP encapsulates data (attributes) and functions (behavior) into packages called objects.
- Objects have the property of information hiding.

- Objects communicate with one another across interfaces.
- The interdependencies between the classes are identified
  - makes use of
  - a part of
  - a specialisation of
  - a generalisation of
  - etc

Structures

- Structure is a collection of simple variable
- Data item in a structure are called the members of the structure.
- In fact the syntax of structure is almost identical to that of a class.
- A structure is collection of data.
- But class is a collection of both data and function

Structures Within Structures

- You can nest structures within other structures
- In this program we want to create a data structure that stores the dimensions of a typical room: its length and width

```
struct Room
{
Distance length;
Distance width;
}
```

- Accessing nested structure member
  dining.length.feet = 13;

C and C++

- C programmers concentrate on writing functions.
- C++ programmers concentrate on creating their own user-defined types called classes.
- Classes in C++ are a natural evolution of the C notion of **`struct`**.

Structures and Classes

- Structures are usually used to hold data only, and classes are used to hold both data and functions.
- In C++ structures can in fact hold both data and functions
- Major difference between class and a structure is that in a class members are private by default while in a structure they are public by default.

3. Implementing a User-Defined Type Time with a Struct

```
#include <iostream.h>

struct Time {     // structure definition
   int hour;      // 0-23
   int minute;    // 0-59
   int second;    // 0-59
};

void printm(Time &);  // prototype
void printS(Time &);  // prototype
```

```
main()
{
   Time t1; // variable of new type Time

   // set members to valid values
 t1.hour = 18;
 t1.minute = 30;
 t1.second = 0;

   cout << "Dinner will be held at ";
 printM(t1);
   cout << " military time,\nwhich is ";
 printS(t1);
   cout << " standard time." << endl;
```

```
      // set members to invalid values

    t1.hour = 29;
    t1.minute = 73;
    t1.second = 103;

      cout << "\nTime with invalid values: ";
    printM(t1);
      cout << endl;

      return 0;
  }
```

```
// Print the time in military format
void printM(Time &t2)
{
   cout << (t2.hour < 10 ? "0" : "") << t2.hour << ":"
  << (t2.minute < 10 ? "0" : "") << t2.minute << ":"
  << (t2.second < 10 ? "0" : "") << t2.second;
}
```

```
// Print the time in standard format
void printS (Time &t2)
{
   cout << ((t2.hour == 0 || t2.hour == 12) ? 12 : t2.hour
  % 12)
  << ":" << (t2.minute < 10 ? "0" : "") << t2.minute
  << ":" << (t2.second < 10 ? "0" : "") << t2.second
  << (t2.hour < 12 ? " AM" : " PM");
}
```

Comments

- Initialization is not required --> can cause problems.
- A program can assign bad values to members of Time.
- If the implementation of the **struct** is changed, all the programs that use the **struct** must be changed. [No "interface"]

4. Implementing a Time problem with a Class

```
#include <iostream.h>
class Time {
public:
   Time();                // default constructor
   void setTime(int, int, int);
   void printM();
   void printS();
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};
```

```
// Time constructor initializes each data member to
  zero.
// Ensures all Time objects start in a consistent
  state.

Time::Time() { hour = minute = second = 0;}

// Set a new Time value using military time.
// Perform validity checks on the data values.
// Set invalid values to zero (consistent state)

void Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
}
```

```
// Print Time in military format

void Time::printM()
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
  << (minute < 10 ? "0" : "") << minute << ":"
  << (second < 10 ? "0" : "") << second;
}

// Print time in standard format

void Time::printS()
{
   cout << ((hour == 0 || hour == 12) ? 12 : hour % 12)
  << ":" << (minute < 10 ? "0" : "") << minute
  << ":" << (second < 10 ? "0" : "") << second
  << (hour < 12 ? " AM" : " PM");
}
```

```
// test simple class Time

main()
{
   Time t;  // instantiate object t of class Time

   cout << "The initial military time is ";
 t.printM();
   cout << "\nThe initial standard time is ";
 t.printS();

   t.setTime(13, 27, 6);
   cout << "\n\nMilitary time after setTime is ";
 t.printM();
   cout << "\nStandard time after setTime is ";
 t.printS();
```

```
   t.setTime(99, 99, 99);
   // attempt invalid settings
   cout << "\n\nAfter attempting invalid
  settings:\n"
        << "Military time: ";
 t.printM();
   cout << "\nStandard time: ";
 t.printS();
   cout << endl;

   return 0;
}
```

Output

- The initial military time is 00:00:00
- The initial standard time is 12:00:00 AM
- Military time after setTime is 13:27:06
- Standard time after setTime is 1:27:06 PM
- After attempting invalid settings:
- Military time: 00:00:00
- Standard time: 12:00:00 AM

Enumerations

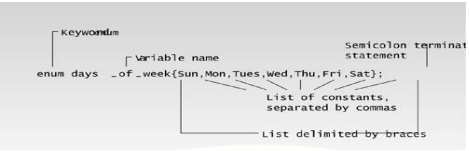
- Structures can be looked at as a way to provide user-defined data types
- A different approach to defining your own data type is the *enumeration.*
- *Enumerated types works when you know in advance a finite list of values.*

Enumerations Example

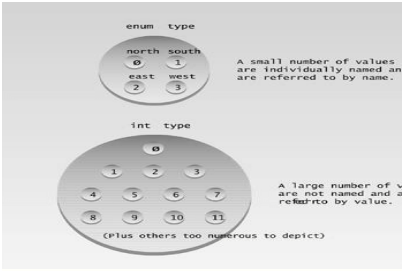
```
#include <iostream>
using namespace std;
//specify enum type
enum days_of_week { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
int main()
{
days_of_week day1, day2;  //define variables  of type days_of_week
day1 = Mon;               //give values to
day2 = Thu;               //variables
int diff = day2 - day1;   //can do integer arithmetic
cout << "Days between = " << diff << endl;
if(day1 < day2)           //can do comparisons
cout << "day1 comes before day2\n";
return 0;
}
```

Enumeration

- **Enum** declaration defines the set of all names that will be permissible values of the type.
- Permissible values are called enumerators.



Usage of **enum** and **int**



Enumeration

- In C must use keyword enum before the type name.

  enum days_of_week day1, day2
- In C++ not neccesary.
- Enumerations internally treated as integers.
- Enum declaration the first enumerator was given the integer value 0, the second the value 1, and so on. This ordering can be altered by using an equal sign to specify a starting point other than 0.

  enum Suit { clubs=1, diamonds, hearts, spades };

ENUMERATION

enum direction { north, south, east, west };
direction dir1 = south;
cout << dir1;

- C++ I/O treats variables of enum types as integers, so the output would be 1

VOID*

- In C it is legal to cast other pointers to and from a void *
- In C++ this is an error, to cast you should use an explicit casting command
- Example:

```
int N;
int *P = &N;
void *Q = P;          // illegal in
  C++
void *R = (void *) P; // ok
```

NULL IN C++

- C++ does not use the value NULL, instead NULL is always 0 in C++, so we simply use 0
- Example:

```
int *P = 0; // equivalent to
            // setting P to NULL
```

- Can check for a 0 pointer as if true/false:

```
if (!P)  // P is 0 (NULL)
  ...
else // P is not 0 (non-NULL)
  ...
```

BOOL

- C has no explicit type for true/false values
- C++ introduces type bool (later versions of C++)
  - also adds two new bool literal constants true (1) and false (0)
- Other integral types (int, char, etc.) are implicitly converted to bool when appropriate
  - non-zero values are converted to true
  - zero values are converted to false