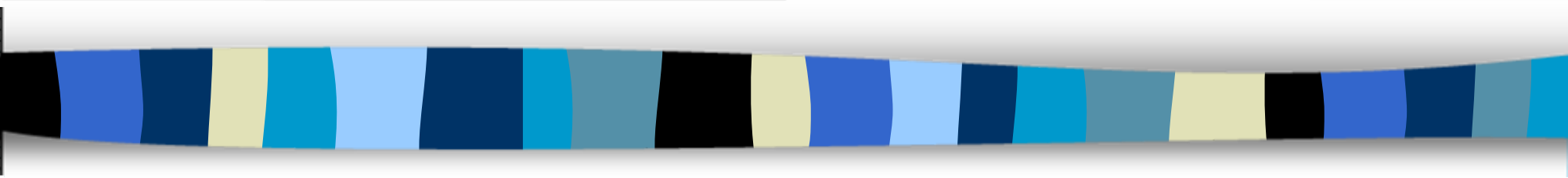
Classes and Data Abstraction

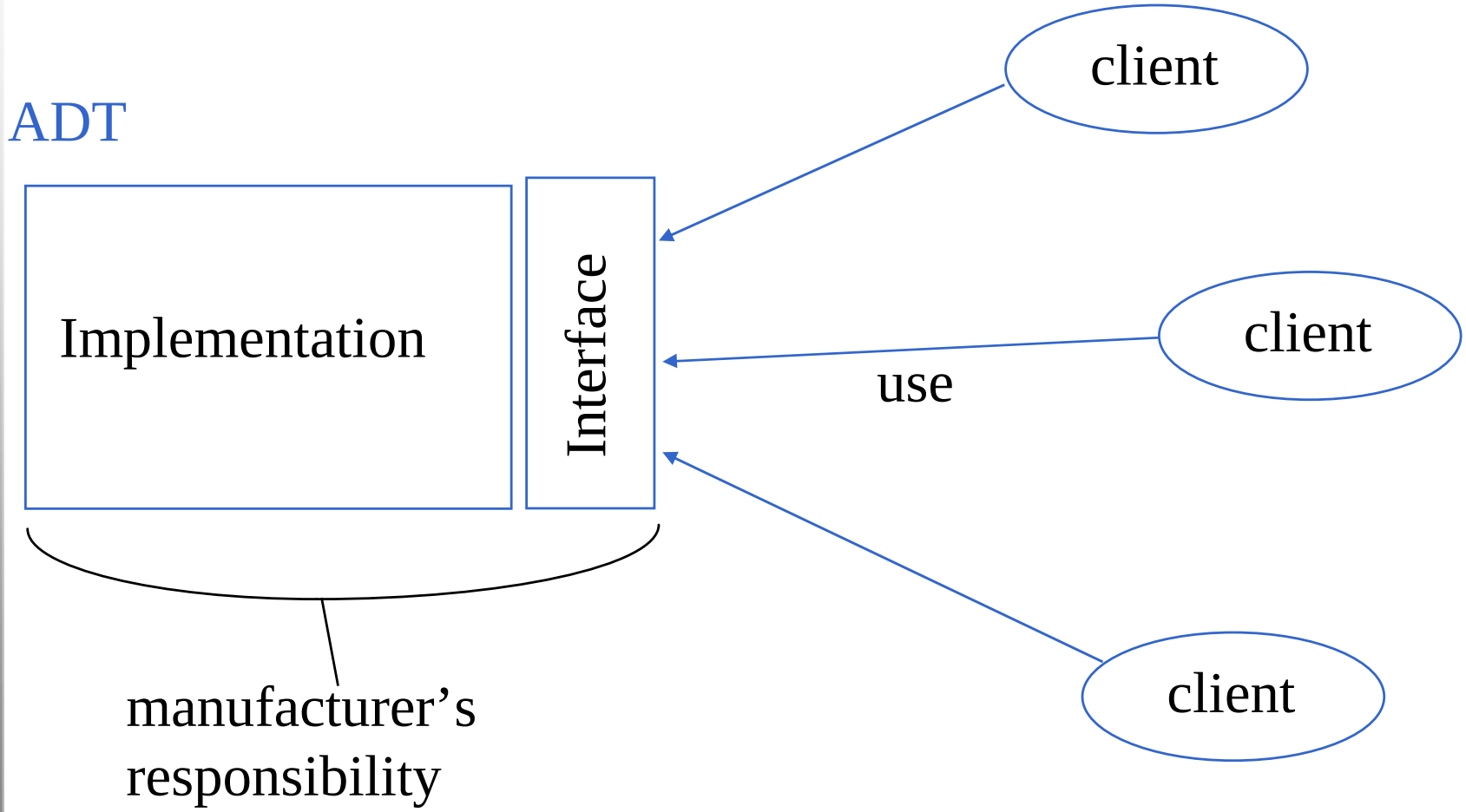


Contents

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

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`**.

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

```
// FIG16_1.CPP
// Create a structure, set its members, and print
   it.
#include <iostream.h>

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

void printMilitary(const Time &);  // prototype
void printStandard(const Time &);  // prototype
```

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

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

   cout << "Dinner will be held at ";
   printMilitary(dinnerTime);
   cout << " military time,\nwhich is ";
   printStandard(dinnerTime);
   cout << " standard time." << endl;
```

```
   // set members to invalid values
   dinnerTime.hour = 29;
   dinnerTime.minute = 73;
   dinnerTime.second = 103;

   cout << "\nTime with invalid values: ";
   printMilitary(dinnerTime);
   cout << endl;

   return 0;
}
```

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

```
// Print the time in standard format
void printStandard(const Time &t)
{
   cout << ((t.hour == 0 || t.hour == 12) ? 12 :
  t.hour % 12)
  << ":" << (t.minute < 10 ? "0" : "") << t.minute
  << ":" << (t.second < 10 ? "0" : "") << t.second
  << (t.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 Abstract Data Type with a Class

```
#include <iostream.h>
// Time abstract data type (ADT) definition
class Time {
public:
   Time();                    // default constructor
   void setTime(int, int, int);
   void printMilitary();
   void printStandard();
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::printMilitary()
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
  << (minute < 10 ? "0" : "") << minute << ":"
  << (second < 10 ? "0" : "") << second;
}

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

```
// Driver to test simple class Time
main()
{
   Time t;  // instantiate object t of class Time

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

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

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

Comments

- **`hour`**, **`minute`**, and **`second`** are private data members. They are normally not accessible outside the class. [Information Hiding]
- Use a constructor to initiailize the data members. This ensures that the object is in a consistate state when created.
- Outside functions set the values of data members by calling the setTime method, which provides error checking.

5. Classes as User-Defined Types

- Once the class has been defined, it can be used as a type in declarations as follows:

```
Time sunset,              //object of type Time
     arrayOfTimes[5],     //array of Time objects
     *pointerToTime,      //pointer to a Time object
```

6. Using Constructors

- Constructors can be overloaded, providing several methods to initialize a class.
- Time example:

Interface

```
Time();
Time(int hr, int min, int sec);
```

Implementation

```
Time::Time()
    { hour = minute = second = 0; }
Time::Time(int hr, int min, int sec)
    { setTime(hr, min, sec); }
```

Use:

```
Time t1;
Time t2(08,15,04);
```