Slides

Data Type

- a collection of related data elements plus operations that can be performed upon values of the data type.

Types

- Built-In (Language Defined)
  - † Array, Structures (Records), Classes
- Programmer Defined, Abstract Data Types (ADT)
  - † Lists, Stacks, Queues

Views

- Application
  - † usage in a particular program (variant box view)

- Abstract (Logical)
  - † organization viewed by the user (black box view)

- Implementation (Physical)
  - † coding methods used to represent the data and the operations (open box view)

Example

- Two-Dimensional Array
  - † Application View: maze, surface points
  - † Logical View: table, matrix
  - † Physical View
    - Stored sequentially (implies logical to physical mapping)
    - Index Limits (L1 .. U1, L2 .. U2)
    - Length = (U1 - L1 + 1)*(U2 - L2 + 1)

Accessing

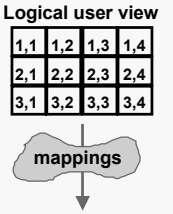
- Column Major: all elements in a column are stored in sequence
  FORTRAN - Column Major
- Row Major: all elements in a row are stored in sequence
  "C" , PASCAL - Row Major

Row Major Accessing:

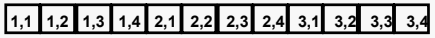
- Location of [i] [j] element (Row Major)

β + [(U2-L2+1) * (i-L1) + (j-L2)] * size of element

β = base address of array
(U2 - L2 + 1) = Size of Row
(i - L1) = number of rows to skip
(j - L2) = number of columns to skip

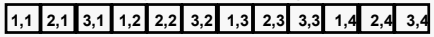


Physical (row-major) compiler programmer linear view



β

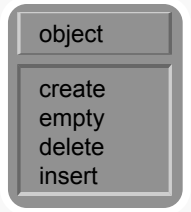
Physical (column-major) compiler programmer linear view



β

ADT

- New data type defined by programmer
- Includes:
  - † Set of Data Objects
  - † Set of Abstract Operations



Data Abstraction

- Design of abstract data objects and operations upon those objects.
- Abstraction in programming separates a data type's logical properties from its implementation.

Information Hiding

- Used in the design of functions and new data types.
- Each component should hide as much information as possible from the users of the component (function).
- Implementation details are hidden from the user by providing access through a well-defined communication interface.

Encapsulation

- the bundling of data and actions in such a way that the logical properties of the data and actions are separated from the implementation details [Dale].
- access to an ADT is **restricted** to a specified set of supplied operations
- Implies:
  - † User has no "Need to Know"
  - † User may **not** directly manipulate data elements
- Advantages
  - † Changes to the underlying operators or representations does not affect code in a client of the data type
  - † Extends programming languages

ADT Levels

The distinction between these terms is not well recognized. Some authors do not distinguish between information hiding and encapsulation, while others assign the opposite definitions given here.

Info Hiding / Encapsulation

What is the difference between Information Hiding and Encapsulation?

Information Hiding

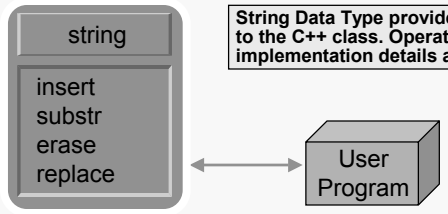
- a question of *program design*
- In many cases: Language Supported (functions, procedures)

Encapsulation

- a question of *language design*
- "...an abstraction is effectively encapsulated only when the language **prohibits** access to information hidden within the abstraction."
- Ada packages
- **C++** classes
  - † "**C"** modules

"C" modules offer limited encapsulation facilities.

Info Hiding Example: Strings



String Data Type provides interface to the C++ class. Operation implementation details are hidden.

Rationale for Classes

Bjarne Stroustrup from *The C++ Programming Language*, 3rd Edition, page 223:

> The aim of the C++ class construct is to provide the programmer with a tool for creating new types that can be used as conveniently as the built-in types.
>
> A type is a concrete representation of a concept.
>
> For example, the C++ built-in type double with its operations +, -, *, etc., provides a concrete approximation of the mathematical concept of a real number. A class is a user-defined type.
>
> We design a new type to provide a definition of a concept that has no counterpart among the built-in types.
>
> A program that provides types that closely match the concepts of the application tends to be easier to understand and to modify than a program that does not.
>
> A well-chosen set of user-defined types makes a program more concise. In addition, it makes many sorts of code analysis feasible. In particular, it enables the compiler to detect illegal uses of objects that would otherwise remain undetected until the program is thoroughly tested.

The C++ Class Type

The C++ class type provides a means to encapsulate heterogeneous data elements and the operations that can be performed on them in a single entity.

A class type is like a struct type in that it may contain data elements, called data members, of any simple or structured type.

A class type may also contain functions, called function members or methods, that may be invoked to perform operations on the data members.

The class type also provides mechanisms for controlling access to members, both data and function, via the use of the keywords public, private and protected. (Default access mode is private.)

A variable of a class type is referred to as an object, or as an instance of the class.

The struct language construct was extended to make it equivalent with a class, except its members are by default public. While structs may be used the same as classes they are rarely employed as such.

A Simple Date Class

Here's a simple class type declaration:

```
class DateType {
public:
   void Initialize(int newMonth, int newDay,
                   int newYear);
   int YearIs( )  const;          // returns year
   int MonthIs( ) const;          // returns month
   int DayIs( )   const;          // returns day
private:
   int Year;
   int Month;
   int Day;
};
```

Indicates Fn is a const member and cannot change any data member values.

The DateType class incorporates three data members, Year, Month, and Day, and four function members.

Note the class type declaration defines a data type — it does not declare a variable of that type.

Also note that the class type declaration above includes only prototypes of the function members, not their definitions.

Typically, the class type declaration is incorporated into a header file, providing a user with a description of the interface to the class, while the implementations of the class methods are contained in a cpp file.

Simple Date Class Continued

Given the class type declaration, a user may declare variables of that type in the usual way:

```
DateType Today, Tomorrow, AnotherDay;
```

No default initializations are performed. It is up to the user to assign values to the data members of these variables.

The data members of the DateType class are declared as being `private`. The effect is that the data members cannot be accessed in the way fields of a struct variable are accessed:

```
Today.Month = 9;
```

will generate a compile-time error. A user of a DateType variable may access only those members which were declared `public`. So, the user could initialize `Today` by using the public member function `Initialize( )`:

```
Today.Initialize(9, 28, 1998);
```

Similarly, a user can only obtain the value of the `Year` member by using the public member function `YearIs( )`:

```
int ThisYear = Today.YearIs( );
```

Note the use of the field selection operator '.'.

Method Definitions

Of course, the member functions of a class type must be defined. Moreover, it is possible for two different class types to have member functions with the same names. In fact, you've already seen that with file streams.

To clearly denote the connection between the function being defined and the class type to which it belongs, the function definition must indicate the relevant class type name by using the scope resolution operator (`::`):

```
// DateType::Initialize()
// Pre:   none
// Post:  self.Year  == newYear
//        self.Month == newMonth
//        self.Day   == newDay
void DateType::Initialize(int newMonth,
                          int newDay, int newYear)
{
   Year  = newYear;     // poor design here:
   Month = newMonth;    //      no error checking
   Day   = newDay;
}
```

Note that, as a member function of class DateType, Initialize( ) may access the data members directly:

> members (data or function) declared at the outermost level of a class type declaration have class scope; that is, they are accessible by **any** function member of an instance of that class type.

Implementation Organization

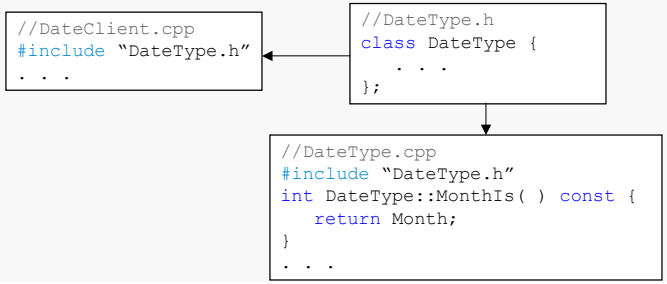
Suppose that a user of the DateType class writes a program consisting of a single source file, `DateClient.cpp`.

```
//DateClient.cpp
class DateType {
   . . .
};

. . .

int DateType::MonthIs( ) const {
   return Month;
}
. . .
```

For separate compilation, a typical organization of the class implementation would involve two files:

`DateType.h`	class declaration
`DateType.cpp`	function member definitions

The user would incorporate the DateType class files as follows:



Building on the Methods

In addition to using the class member functions directly, the user of a class may also implement higher-level functions that make use of the member functions. For example:

```
enum RelationType {Precedes, Same, Follows};

RelationType ComparedTo(DateType dateA, DateType dateB) {

   if (dateA.YearIs() < dateB.YearIs())
      return Precedes;
   if (dateA.YearIs() > dateB.YearIs())
      return Follows;
   if (dateA.MonthIs() < dateB.MonthIs())
      return Precedes;
   if (dateA.MonthIs() > dateB.MonthIs())
      return Follows;
   if (dateA.DayIs() < dateB.DayIs())
      return Precedes;
   if (dateA.DayIs() > dateB.DayIs())
      return Follows;
   return Same;
}
```

Then:

```
DateType Tomorrow, AnotherDay;

Tomorrow.Initialize(10, 6, 1881);
AnotherDay.Initialize(10, 12, 1885);

if ( ComparedTo(Tomorrow, AnotherDay) == Same ) {

   cout << "Think about it, Scarlett!" << endl;
}
```


Building on the Methods

Another example:

```
void PrintDate(DateType aDate) {
   PrintMonth( aDate.MonthIs( ) );
   cout << ' ';
   cout << aDate.DayIs( );
   cout << ", ";
   cout << setw(4) << aDate.YearIs( );
   cout << endl;
}

void PrintMonth(int Month) {
   switch (Month) {
   case  1: cout << "January"; return;
   case  2: cout << "February"; return;
     . . .
   case 12: cout << "December"; return;
   default: cout << "Juvember";
   }
}
```

Then:

```
DateType LeapDay;
LeapDay.Initialize(2, 29, 2000);

PrintDate(LeapDay);
```

will print:

```
February 29, 2000
```


Additional Methods

Of course, the DateType class designer could also have implemented a member function for comparing two dates:

```
// add to DateType.h:
   // add above DateType class declaration
   enum RelationType {Precedes, Same, Follows};
   // add to public members of DateType class declaration
   RelationType ComparedTo(DateType otherDate) const;
```

```
// add to DateType.cpp:
RelationType DateType::
             ComparedTo(DateType otherDate) const {

   if (Year < otherDate.YearIs())
      return Precedes;
   if (Year > otherDate.YearIs())
      return Follows;
   if (Month < otherDate.MonthIs())
      return Precedes;
   if (Month > otherDate.MonthIs())
      return Follows;
   if (Day < otherDate.DayIs())
      return Precedes;
   if (Day > otherDate.DayIs())
      return Follows;
   return Same;
}
```

```
//alternatively
if (Year < otherDate.Year)
```

Then, assuming the declarations and initializations from slide 14:

```
if ( Tomorrow.ComparedTo(AnotherDay) == Same ) {

   cout << "Think about it, Scarlett!" << endl;
}
```


Date Class Design

Class Completeness

- The Date class as given is woefully incomplete.
- In creating a class the designer must carefully consider all possible operations that may need to be performed.
- Decisions as to whether an operation should be part of a class or externally implemented by the user are affected by multiple factors:
  - Is the operation likely to be required for multiple applications of the class? (i.e. is the operation general rather than specific to a problem?)
  - Is the operation needed to insure the correctness / robustness of the class? (Do all of the operations maintain the info hiding / encapsulation of the class?)

Missing Date Operations

- Two of the more obvious missing functions are date increment and decrement functions (mutators):

```
// add to Date Class

void Next ();      //increment date
void Previous (); //decrement date
```

One could ask if it is the responsibility of the class designer or user to implement the above operations? Since the answer to the previous questions are yes, it is the class designer's responsibility. This is obvious when the designer considers multiple applications that will need the operations.
Implementation will require consideration of the length of a month, possible leap year/century, lower date calendar limit.

Taxonomy of Member Functions

Member functions implement operations on objects. The types of operations available may be classified in a number of ways. Here is one taxonomy from Nell Dale:

Constructor

an operation that creates a new instance of a class (object)

Destructor

an operation that destroys an object

Transformer (mutator)

an operation that changes the state of one, or more, of the data members of an object

Observer (reporter, accessor, selector, summary)

an operation that reports the state of one or more of the data members of an object, without changing them

Iterator

an operation that allows processing of all the components of a data structure sequentially

Recalling the DataType declaration, Initialize( ) is a mutator while YearIs( ), MonthIs( ) and DayIs( ) are observers.

Class Constructors

The DateType class has a pseudo-constructor member function, but the situation is not ideal. A user may easily forget to call Initialize( ) resulting in mysterious behavior at runtime.

It is generally preferred to provide a constructor which guarantees that any declaration of an object of that type must be initialized.

This may be accomplished by use of a member function:

```
//add to DateType class declaration
DateType(int aMonth, int aDay, int aYear);
//add to DateType class member functions
DateType::DateType(int aMonth, int aDay, int aYear) {

   if (  (aMonth >= 1 && aMonth <= 12)
      && (aDay >= 1) && (aYear >= 1) ) {
      Month = aMonth;
      Day   = aDay;
      Year  = aYear;
   }
   else {
      Month = Day = 1;     // default date
      Year  = 1980;
   }
}
```

- the name of the constructor member must be that of the class
- the constructor has no return value; `void` would be an error
- the constructor is called automatically if an instance of the class is defined; if the constructor requires any parameters they must be listed after the variable name at the point of definition.

Default Constructor

Assuming the DateType constructor given on the previous slide, definitions of an instance of DateType could look like:

```
DateType aDate(10, 15, 1998);

DateType bDate(4, 0, 1999);     // set to 1/1/1980
```

If you do not provide a constructor method, the compiler will automatically create a simple default constructor. This automatic default constructor:

- takes no parameters
- calls the default constructor for each data member that is an object of another class
- provides no initialization for data members that are not objects

Given the limitations of the automatic default constructor:

Always implement your own default constructor when you design a class!

Class Destructors

Class destructor functions provide a cleanup mechanism for class objects. The destructor function for a class has the same name s the class, but preceded with the tilde '~' character. Destructor functions, like constructors, take no parameters.

Destructor functions are implicitly invoked whenever an object goes out of scope. This can occur in two cases:

- The end of the block in which the object is instantiated is reached.
- A dynamically allocated object is explicitly deleted, (more about this later).

The primary purpose for which destructor functions are employed is for the reclamation of dynamically allocated memory.

Classes that do not explicitly define a destructor function will have a default destructor automatically provided by the compiler. This automatic default destructor :

- takes no parameters
- calls the default destructor for each data member that is an object of another class
- provides no cleanup for data members that are not objects

Multiple Constructors

It is possible to have more than one constructor for a class:

```
class Complex {
private:
   double Real, Imaginary;
public:
   Complex( );
   Complex(double RealPart, double ImagPart);
   .  .  .
   double Modulus( );
};

Complex::Complex( ) {
   Real      = 0.0;
   Imaginary = 0.0;
}

Complex::Complex(double RealPart, double ImagPart ) {
   Real      = RealPart;
   Imaginary = ImagPart;
}

double Complex::Modulus( ) {
   return (sqrt(Real*Real + Imaginary*Imaginary));
}

Complex x(4, -1);                // x == 4.0 - 1.0i
Complex y;                       // y == 0.0 + 0.0i

double  xMagnitude = x.Modulus();
```

So, how does the compiler determine which constructor to call?

Overloading, Briefly

In C++ it is legal, although not always wise, to declare two or more functions with the same name. This is called overloading.

However, it must be possible for the compiler to determine which definition is referred to by each function call. When the compiler encounters a function call and the function name is overloaded, the following criteria are used (in the order listed) to resolve which function definition is to be used:

Considering types of the actual and formal parameters:

1. Exact match (no conversions or only trivial ones like array name to pointer)
2. Match using promotions (bool to int; char to int; float to double, etc.)
3. Match using standard conversions (int to double; double to int; etc.)
4. Match using user-defined conversions (not covered yet)
5. Match using the ellipsis . . . in a function declaration (ditto)

Clear as mud, right? Keep this simple for now. Only overload a function name if you want two or more logically similar functions, like the constructors on the previous slide, and then only if the parameter lists involve different numbers of parameters.

Operator Overloading

Standard Operator Overloading

- C++ language operators, (e.g., "==", "++", etc.) can be overloaded to operate upon user-defined classes.

```
// add to DateType class declarations:
bool operator==(const DateType& otherDate) const ;
```

```
// add to DateType class member functions:
bool DateType::operator==
                (const DateType& otherDate) const {
   return( (Day   == otherDate.DayIs()   ) &&
           (Month == otherDate.MonthIs() ) &&
           (Year  == otherDate.YearIs()  ));
}
```

- To "call" the overloaded operator function:

```
DateType aDate(10, 15, 1998);
DateType bDate(10, 15, 1999);

if (aDate == bDate) { . . .
```

- Do not use dot operator to call:

```
aDate.==(bDate)   // error
```

  this is basically how the compiler translates the expression.
- overloading the other relational operators, '<', '>', would eliminate need for enum RelationType & ComparedTo Fn
- standard relational operators, (overloaded or not), cannot be used as case tags in switch statements.
- would force switch RelationType statement to be implemented as if else statements

Default Arguments

Initialized Parameters

- technique provided to allow formal parameters to be assigned default values that are used when the corresponding actual parameter, (argument), is omitted.

```
const int IBMPC = 1980;

// add to DateType class declarations:
DateType(int aMonth=1, int aDay=1, int aYear=IBMPC);
// add to DateType class member functions:
DateType::DateType(int aMonth, int aDay, int aYear){
   if (  (aMonth >= 1 && aMonth <= 12)
      && (aDay >= 1) && (aYear >= 1) ) {
      Month = aMonth;
      Day   = aDay;
      Year  = aYear;
   }
   else {
      Month = Day = 1;     // default date
      Year  = IBMPC;
   }
}
```

If a default argument is omitted in the call, the compiler "automagically" inserts the default value in the call.

```
DateType dDate(2,29);      // Feb 29, 1980
DateType eDate(3);         // March 1, 1980
DateType fDate();          // Jan 1, 1980
```

Omitted arguments in the call must be the rightmost arguments.

```
DateType dDate(,29);       // error
```


Default Argument Values Usage

Default Arguments in prototypes

- Omitted arguments in the function prototype must be the rightmost arguments.

```
// add to DateType class declarations:
DateType::DateType
        (int aMonth=1, int aDay, int aYear=IBMPC);
// error
```

Default Arguments - Guidelines

- Default arguments are specified in the first declaration/definition of the function, (i.e. the prototype).
- Default argument values should be specified with constants.
- In the parameter list in function declarations, all default arguments must be the rightmost arguments.
- In calls to functions with > 1 default argument, all arguments following the first (omitted) default argument must also be omitted.

Default Arguments and Constructors

- Default argument constructors can replace the need for multiple constructors.
- Default argument constructors can ensure that no object will be created in an non-initialized state.
- Constructors with completely defaulted parameter lists, (can be invoked with no arguments), becomes the class default constructor, (of which there can be only one).

Inline Member FNs

Class declaration inline functions

- inline functions should be placed in header files to allow compiler to generate the function copies.

```
class DateType {
public:
   void Initialize(int newMonth, int newDay,
                   int newYear);
   int YearIs  () const {return Year;};
   int MonthIs () const {return Month;};
   int DayIs   () const {return Day;};
private:
   int Year, Month, Day;
};
```

- Member functions defined in a class declaration are **implicitly** inlined.
- Efficiency is traded off at the expense of violating the information hiding by allowing the class clients to see the implementation.
- Reference to the class data members by the inline functions before their actual definition is perfectly acceptable due to the class scoping.
- To avoid confusion, the private members can be defined first in the class or the inline functions can be explicitly defined after the class declaration, (but in the header file).
- Changing a header file containing an inline function will result in recompilation of all files including the header.

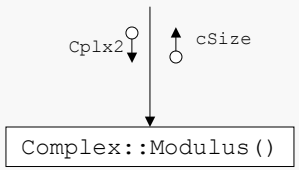
Structure Charts with Classes

To reflect the use of classes in your modular structure chart, do the following:

First, create a *class form* for each of your classes:

Name:	Complex (this is an extension of the example on slide 3.14)
Purpose:	To provide a means of representing and manipulating complex numbers
Constructors:	Complex( ) constructs 0.0 + 0.0i Complex(double x, double y) constructs x + yi
Operations *Mutators:*	setReal(double x) assigns Real the value x setImag(double y) assigns Imaginary the value y
Reporters:	Modulus( ) returns the modulus (magnitude) of the number
Data Members:	double Real double Imaginary

Second, in the function boxes in the modular structure chart, indicate if a call is to a member function of an object by using the class name and the scope resolution operator. Show the invoking object, (an implicit parameter) as an input parameter if the member function is a reporter const member function. Otherwise, show the invoking object as an output or input/output parameter:



Here, the return value is being assigned to a variable named `cSize` in the calling function.

Note: usage of the name of the class and the name of the object.

Example of a simple data class of inventory records.

```
// *********** INVENTORY CLASS DECLARATION ************

class InvItem {
  private:
    string SKU;             //StocK Unit #: KEY FIELD
    string Description;     //Item Details
    int    Retail;          //Selling Price
    int    Cost;            //Store Purchase Price
    int    Floor;           //Number of Items on display
    int    Warehouse;       //Number of Items in stock

  public:
    InvItem();              //default constructor
    InvItem(const string& iSKU, //parameter constructor
            const string& iDescription,
                  int    iRetail,
                  int    iCost,
                  int    iFloor,
                  int    iWarehouse);

     //Reporter Member Functions
     string  getSKU()         const;
     string  getDescription() const;
     int     getRetail()      const;
     int     getCost()        const;
     int     getFloor()       const;
     int     getWarehouse()   const;

    //Mutator Member Functions
    void  setDescription(const string& descript);
    void  setRetail(int customer);
    void  setCost  (int actual);
    void  setFloor (int display);
    void  setWarehouse(int stock);
}; // class InvItem
```


Inventory Class: Constructors

```
//-------------- Constructor Functions ------------------

////////////////////////////////////////////////////////////
// Default Constructor for InvItem Class
//
// Parameters:      none
// Pre:             none
// Post:            InvItem object has been initialized
//
InvItem::InvItem() {
          SKU          = "?";
          Description  = "?";
          Retail       = -1;
          Cost         = -1;
          Floor        = -1;
          Warehouse    = -1;
} // end InvItem()


////////////////////////////////////////////////////////?///
// Parameter Constructor for InvItem Class
//
// Parameters:      InvItemRec
// Pre:             none
// Post:            InvItem object has been set
//
InvItem::InvItem(const string& iSKU,
                 const string& iDescription,
                       int     iRetail,
                       int     iCost,
                       int     iFloor,
                       int     iWarehouse) {

          SKU          = iSKU;
          Description  = iDescription;
          Retail       = iRetail;
          Cost         = iCost;
          Floor        = iFloor;
          Warehouse    = iWarehouse;
}  // end InvItem()
```


Inventory Class: Reporters

```
//-------------- Reporter Functions ------------------
// Fn Headers omitted for space

string    InvItem::getSKU() const {

          return(SKU);

}  // end getSKU()

string    InvItem::getDescription() const {

          return(Description);

}  // end getDescription()

int       InvItem::getRetail() const {

          return(Retail);

}  // end getRetail()

int       InvItem::getCost() const {

          return(Cost);

}  // end getCost()

int       InvItem::getFloor() const {

          return(Floor);

}  // end getFloor()

int       InvItem::getWarehouse() const {

          return(Warehouse);

}  // end getWarehouse()
```


Inventory Class: Mutators

```
//-------------- Mutator Functions ------------------
// Fn Headers omitted for space

void      InvItem::setDescription(const string& descript) {

          Description = descript;

}  // end setDescription(const string descript)

void      InvItem::setRetail(int customer) {

          Retail = customer;

}  // end setRetail(int customer) {

void      InvItem::setCost(int actual) {

          Cost = actual;

}  // end setCost(int actual)

void      InvItem::setFloor(int display) {

          Floor = display;

}  // end setFloor(int display)

void      InvItem::setWarehouse(int stock) {

          Warehouse = stock;

}  // end setWarehouse(int stock)
```

Note: the class contains no mutator function for the SKU member. The SKU member is the primary/key field and thus is considered inviolate.