**Rationale for Classes**

- 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.

**Rationale for Classes**

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

**A Simple Date Class**

```cpp
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.
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 (::):

```cpp
// 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;
}
```

Scope

- 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.

```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

```cpp
//DateType.h
#include "DateType.h"

class DateType {
    . . .
}

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

- The user would incorporate the DateType class files as follows:

```cpp
//DateClient.cpp
#include "DateType.h"

class DateType {
    . . .
}

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

- Data and functions outside the class are at a file scope and do not have direct access to the data and methods inside the class.
- Variables inside a method have method scope.
- If a variable in a function has the same name as an already created class variable, then the function variable hides the class variable.
- The hidden class variable can be used through the scope resolution operator `::`.

Calling Class Functions

```cpp
bool LessThan(DateType dateA, DateType dateB)
{
    if (dateA.YearIs() < dateB.YearIs())
        return true;
    else
        return false;
}
```

Accessing Other Instances of the Same Object

```cpp
// add to DateType.cpp:
RelationType DateType::LessThan(DateType otherDate) const {
    if (Year < otherDate.YearIs())
        return true;
    else
        return false;
}
// alternatively
if (Year < otherDate.Year)
```

Class Completeness

- The Date class as given is incomplete.
- 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 insur the correctness / robustness of the class? (Do all of the operations maintain the info hiding / encapsulation of the class?)

Missing Date Methods

- What about a Next() function to increment the date?
- What about a Previous() function to decrement the date?
- THIS IS WHY DESIGN IS SO IMPORTANT!!
  It is the designers job, not the coders job to think of the missing functions and the integrity of the class.

Taxonomy of Methods

- 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
Taxonomy of Methods

- **Accessor** (reporter, observer, 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

Utility Methods

- Private methods are called utility functions because **only** other member functions use them
- Recalling the DataType declaration, Initialize( ) is a mutator while YearIs( ), MonthIs( ) and DayIs( ) are accessors.

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.

Constructor For DateType

```cpp
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;
    }
}
```

Using Constructors

- Assuming the DateType constructor given on the previous slide, definitions of an instance of DateType could look like:
  ```cpp
dateType aDate(10, 15, 1998);
dateType bDate(4, 0, 1999);  // set to 1/1/1980
```
- You’ve seen this before!
  - string myString;
  - istringstream istr(myString);
Default Constructor

- If you do not provide a constructor method, the compiler will automatically create a simple 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
- Always implement your own default constructor when you design a class!

Back to Structure Charts!

- 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:

### A Simple Date Class

```cpp
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;
};
```

**Method Definitions**

```cpp
// 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;
}
```

**Constructor For DateType**

```cpp
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;
    }
}
```
Constructors and Const

- Remember some methods are declared constant?
  ```cpp
  void getObjectID() const;
  void CreateAndDestroy::getObjectID() const
  {
    return objectID;
  }
  ```
- The "constness" of an object begins as soon as the constructor finishes its job and ends when the destructor is called

Const Problem

- How do you initialize const data when const data has to be initialized when it is declared?
- Through the use of a member initializer list
  ```cpp
  MyClass::MyClass() : DataConstant1(10), DataConstant2(1) {
    //whatever else the constructor needs // to do
  }
  ```

Const Problem Solved

- Can use this to initialize all member data
- Must use this to initialize const member data

Destructors

- Called whenever the class object goes out of scope as a cleanup mechanism
  - The end of the block in which the object is instantiated is reached.
  - A dynamically allocated object is explicitly deleted. (more about this later).
- Has only one signature
- MyClass::~MyClass();
- The primary purpose for which destructor functions are employed is for the reclamation of dynamically allocated memory.

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
- DOES NOT CLEAN UP DYNAMIC MEMORY!

Class Composition

- Classes can have other objects as data members
- To initialize the member objects appropriately, use the member initializer list
Example
class MyClass
{
    private:
        AnotherClass aObject;
        SomeOtherClass sOObject;
        const DataType x;
};

Back to the Const
MyClass::MyClass() : aObject(x),
    sOObject(y), x(z)
{
}

Friends
- Friends are allowed access to private data
- A friend can be another class or even a stand alone function
- Friends must be explicitly stated as a friend, before it can be considered a friend

More Friends
- Friendship is not symmetric or transitive
- Example
class Bill
{
    friend class Mary;
    ...
}

static members
- One copy shared among all objects
- Example
  - Martians only become aggressive when there are at least 5 around
  - Use static variable martianCount to let all Martians know how many Martians are currently around

Initializing Static Members
class Martian
{
    public:
        static int getCount();
    private:
        static int Count;
}
Note: a static method can only manipulate static data or call other static methods.
Statics, Cont.
- In implementation file, initialize static member at the top, not in any method
- Use Scope Resolution Operators to indicate which scope
- Example
  - Int Martian::Count = 0

Overloading
- In C++ it is legal, although not always wise, to declare two or more functions with the same name. This is called overloading.
  - For example, multiple constructors!
- However, it must be possible for the compiler to determine which definition is referred to by each function call.

More Overloading
- When the compiler encounters a function call and the function name is overloaded, criteria are used to resolve which function definition is to be used.
- Rule of thumb: Only overload a function name if you want two or more logically similar functions, like constructors, and then only if the parameter lists involve different numbers of parameters.

Operator Overloading
```cpp
// 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() );
}
```

Default Arguments
- If a default argument is omitted in the call, the compiler "automagically" inserts the default value in the call!
```cpp
// add to DateType class declarations, NOT the implementation!!!
DateType(int aMonth=1, int aDay=1, int aYear=1980);

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

More Operator Overloading
- We could have overloaded the relational operator ‘<’ instead of making a LessThan method in our DateType class
Default Arguments, Constructors

- Default argument constructors can replace the need for multiple constructors.
- Default argument constructors can ensure that no object will be created in a 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).

Default Argument Rules

- Default arguments are specified in the first declaration/definition of the method, (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.

Inline Methods

- 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

Inline Methods

```cpp
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;
};
```

Watch Out

- Returning a reference (like a pointer) to a private member
  – Will not work! Why?

What happens when you say =

- DateType a, b;
- a = b; //what happens?
- For simple classes, with no dynamic data, everything works fine.
- Memberwise assignment takes place.
  – Get into trouble when we start looking at pointers
Using a pointer to reference Components of a class

class Test
{
  public:
    Test(int = 0);
    void print() const;
  private:
    int x;
};

Test::Test (int value) : x(value) {};
void Test::print() const
{
  cout << "x = " << x;
  cout << "this->x = " << this->x;
  cout << "(*this).x = " << (*this).x;
}

int main()
{
  Test testObject( 12 );
testObject.print();
return 0;
}

x = 12
this->x = 12
(*this).x = 12

Separate Compilation Revisited

- Single Source File Programs
  - Programs where all the code is contained within 1 file.
  - Large programs results in problems.

- Disadvantages
  - Very long compile time
  - Errors requires recompilation of entire program
  - Difficult to edit
Separate Compilation Revisited

- **Multi-File Programs**
  - Code for a program is stored in several source files.

- **Advantages**
  - Decreases re-compile time for errors.
  - Modification of code in one file does NOT require compilation of other files, (exception: if function interfaces have changed).
  - Programs can be broken into smaller, simpler subsystems.

More Advantages

- Separate compilation helps support structured methods and modular decomposition for developing large systems.
- Allows languages to be used in conjunction with other programming languages.
- Eases testing in large systems.
- Allows access to system functions, code libraries and packages.
- Facilitates code reusability.

Separate Compilation Steps

- **Step 1**
  - Source files compiled to object files.

- **Step 2**
  - Object files linked to form executable image.

C++ “Module” Structure

- Interface File: (header file .h) contains public declarations of all articles that are accessible (visible) and usable by external modules that include the interface file.

- Articles consist of constants, typedefs, enum types, class/struct declarations and function prototypes (only parameter types need to be specified, parameter names included are ignored by the compiler).

Implementation File:

- (code file .c or .cpp) contains private declarations and definitions of all articles that are inaccessible (invisible) and NOT usable by external modules that include the interface file.

- The full declarations of the function prototypes given in the header file are specified.

- Articles declared in the implementation file that are NOT declared in the interface file are considered local/internal to the module and can only be accessed by the module’s code NOT by external code.
External Definitions

- Definitions that occur outside all functions in a file.
- Scope extends to the end of the file.
- Cannot be accessed outside of file, unless declared as an extern identifier in separately compiled files.
  ```
  extern int x;
  extern void fn(long);
  ```
- Extern declarations are NOT definitions (no storage is reserved, no initialization can be performed).

Header Inclusion Problem

- Separate files may use the same header file.
  - Assume a programmer has stored system wide constants in: const.h
  - Assume const.h is included in the modulea header file: modulea.h
  - Assume moduleb includes modulea.h and const.h.
- const.h header declarations would be duplicated in moduleb.cpp after preprocessing.

Conditional Compilation

- Preprocessor directives:
  ```
  #if #ifdef #ifndef #elif #else #endif
  #undef #define
  ```
- Usage in all files that include const.h
  ```
  #ifndef CONST_H
  #define CONST_H
  #include "const.h"
  #endif
  ```

Header Inclusion Problem Fixed

- Inclusion of const.h instructs the preprocessor to check if CONST_H has been previously defined during preprocessing. If it has not then it is defined and the const.h declarations are copied into the source, otherwise no inclusion occurs.