Classes

Structured types

- The syntax and semantics of `struct` in C are retained in C++ with minor differences
  - In C++, the `struct` keyword may be dropped when using the structure name as a type
  - In C, the keyword `struct` can be dropped only if you declare the structure as a type using `typedef`
- Structure assignment, passing structures to functions, and returning structure values follow the rules set forth in ANSI C standard
- The `s->member` abbreviation for `(*s).member` is retained

Member functions

- In addition to data members, structures in C++ can have member functions
- Example structure date

```
struct date
{
    int dd,                 // Day of the month
        mm,                 // Month in integer form
        yy;                 // Year (could be full such as 1997)

    // Member functions of the structure (prototypes)

    void print();           // Print the date
    void advance ( int );   // Go forward/backward by specified no. of days
    date add ( int );       // Increment the date by specified number of days
    long diff ( date );     // Get the difference from current date
};

void date::print()
{
    cout << mm << "/" << dd << "/" << yy;
}

date today = { 9, 9, 1997 };

today.print();
```

  - The `date::` in the definition of the function header indicates that the function `print` is a member function of the structure `date`
  - Within the definition of `print`, structure members are used without a variable and without the `.` or `->` preceding them
  - In the call to the function, the members refer to the object preceding the period in `today.print()` (*implicit argument*)
  - Each member function is required to have at least one argument – the implicit argument which is the object to be operated on
  - Explicit arguments are also allowed (in parenthesis)
  - Other structures may have `print` function of their own

```
int days_in_month ( int month, int year )
{
    static int dpm[] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    if ( month != 2 )
        return ( dpm[month] );

    // February is treated as a special case

    return ( 28 + (year % 4 == 0) - (year % 100 == 0) + (year % 400 == 0) );
}

void date::advance ( int n )
{
    int d;

    dd += n;      // Advance the day by the number specified
    while ( dd > ( d = days_in_month ( mm, yy ) ) )
    {
        dd -= d;
        if ( ++mm > 12 )
        {
            mm = 1;
            yy++;
        }
    }

    while ( dd < 1 )    // Specified number of days is negative
    {
        if ( --mm < 1 )
        {
            mm = 12;
            yy--;
        }
        dd += days_in_month ( mm, yy );
    }
}
```

- It may be desirable to refer to the implicit argument in its entirety, or to pass its address to another function
- Achieved by referring to an implicit pointer variable `this` that contains the address of the implicit argument within the body of the function
- The function advance modified the original date; the function add returns a new date without modifying the original date

```
date date::add ( int n )
{
    date r = *this;      // Make a copy of the implicit argument
    r.advance( n );      // Advance the argument by n days
    return ( r );
}
```

- Why use member functions?
  1. Member functions of different structures have the same name

   - You can have `print` as a member of both `date` and `vector`
   - The compiler can tell which function to use for `x.print()` by looking at the type of `x`
2. Convenient to package all functions that allow access to structure members of a data type together
   - The user of the package gets a consistent user interface even if the internal data representation changes
   - Member functions for this come neatly packaged with the structure definition
3. Intuitive to single out the implicit argument as a special object
   - Think of a function `car.accelerate ( 5 )` as sending a message to the object `car`
   - Structure variables are like objects, member functions are like methods, and member function calls are like messages
4. Convenient to refer to the members of `this` simply by their name without having to apply the prefix
5. Some functions necessary for memory management (constructors and destructors) as well as `operator=`, `operator->`, `operator[]`, and `operator()` must be member functions
6. Functions dynamically selected at run-time (virtual functions) must be member functions

- Short member functions can be defined inside the structure definition
  - These functions automatically become inline functions
  - Can significantly increase code size, especially if an inline function calls another inline function
  - Inline functions may not be recognized by the debuggers
- Enumeration constants defined inside structures are considered members
  - When used by nonmember functions, they must be preceded by *structure name::*
  - Consider the following enumeration constants

```
struct date
{
    enum weekday { Monday = 1, Tuesday, Wednesday, Thursday, Friday, \
                   Saturday, Sunday };
};
```

  - It can be used as `Monday` inside a member function of `date` but must be used as `date::Monday` elsewhere

Classes

- Goal of OO programming
  - *Keep the code that uses the data completely unchanged when the internal representation is modified*
- The structure `date` can be represented in two different formats – julian or gregorian – with different benefits to be derived from each representation
- We may decide to keep the internal representation as julian and supply member functions `day()`, `month()`, and `year()`, in addition to the ones already created
- The privacy of data structures yields abstract data types such that the user of such structures never has to know the internal representation of the structures, or the algorithms of the member functions
  - The data is accessed through an interface that remains stable even if the internal representation changes
- Achieved by separating member data and functions into two sections – *private* and *public* – and encapsulating both of these sections into a *class*
  - There are three keywords (*access specifiers*) in C++ to separate the data and functions as per access privileges: `public`, `private`, and `protected`

  - Private section
    * Provides the representation for the structure
  - Public section
    * Provides the abstract data type interface

- Classes are responsible for information/implementation hiding
  - Classes include functions (methods) within structures (encapsulation)
  - This produces a data type with characteristics (attributes) and behaviors (methods/functions), but access control puts boundaries within that data type
    1. It decides what a user can or cannot use
    2. It separates interface from the implementation
       * If structure is used in a set of programs, but users can't do anything but send messages to the `public` interface, then you can change anything that is `private` without requiring modifications to their code

- A `class` is the same as a `struct`, except that all the members are hidden unless declared `public`
  - Effectively, a `struct` is a `class` with no `private` members

- Now, we can redefine `date` as

```
class date
{
    int dd,                 // Day of the month
        mm,                 // Month in integer form
        yy;                 // Year (could be full such as 1997)

    // Public member functions of the structure (prototypes)

    public:
        int  day() const;
        int  month() const;
        int  year() const;
        void print() const;    // Print the date
        int  read();           // Read the date
        void advance ( int );  // Go forward/backward by specified no. of days
        date add ( int );      // Increment the date by specified number of days
        long diff ( const date& );   // Get the difference from current date
};
```

  - Now, you can change the private section to `long julian;` and leave the public section completely unchanged (if you change the corresponding functions without changing the interface)
  - Any attempt to access the private members, such as

    ```
    date today;
    today.dd = 11;
    ```

    will be caught by the compiler and flagged as an error
  - You can also declare the member functions as private and the data as public

- The `private` members of a `class` can only be accessed by the member functions for the class (as well as `friend` functions)

- The `private` member functions can be called only inside other member functions of the same class

- `date::print()` can immediately access the data members but other functions need to call *access functions* such as `date::year()`

- Exercise: Convert `struct date` to `class date`, with the data being `private` and the *helper functions* being private as well

- It is possible to define a data member that is shared by all *instances* of a class by declaring it `static`
  - Let us keep track of how many calls are made to `date::print()` by adding a static field `print_count` to the structure

```
class date
{
    int dd, mm, yy;
    int days_in_month ( int, int );
    public:
        static int print_count;
// Other function prototypes
};

int date::print_count = 0;
```

  - There is only one location `print_count` for the entire set of objects of class `date`, not one for each object
  - This single location is accessible in any member function
  - `date::print()` can be changed to

```
void date::print()
{
    print_count++;
    // ...
}
```

  - Since `print_count` has been declared in the public section, it can be accessed from anywhere as `date::print_count`
  - Each static data member must be explicitly defined outside the `class`, even if the static member is private
  - The `static` variable in a class occurs only once just like the `static` local variable which occurs only once even if a recursive call is made to the same function

- You can also declare the member functions to be `static`
  - The `static` member functions have no implicit `this` argument and can only operate on `static` data members
  - These functions follow the same rules of invocation as the static member data (have to use the `date::` notation in the non-member functions

- Exercise: Make the function `days_in_month` and the array `dpm` into private static members of `class date`

- Exercise: Design a `string` class

```
class string
{
    char s[ MAXSIZE ];

    public:
        void   read();       // Read a string from cin, delimited by white space
        int    length();     // Get the length of the string
        int    print();      // Print the string
        string substring( int, int ); // Extract a substring
        int    find ( const string& ); // First occurrence of the substring
```

```
                                         // return -1 on failure
        string concat ( const string& ); // Concetenate the specified string
                                         // to given string
        string strcmp ( const string& ); // Compare given string with the
                                         // specified string
}
```

- The static members have two advantages over global variables
  1. The class mechanism protects them from uncontrolled modification
  2. The identifiers used by them are not blocked

Constructors

- Initialization
  - If an initializer is specified for an object, the initializer provides the initial value for the object

    ```
    int x = 5;
    int answer ( 42 );      // Preferred syntax for initialization
    ```

  - If no initializer is specified, the object is initialized to 0 of the appropriate type if the object is local static, global, or namespace
    * Local objects (automatic variables) and dynamic objects (created in free space or heap) are not initialized by default
  - Members of arrays and structures are default initialized if the array or structure is static
- Problem with `class date`: How can we set a date?
  - Statement like `d.dd = 11` is illegal
  - The access function `d.day()` can report a day but not set it
- We can write a member function

  ```
  void date::set ( int m, int d, int y );
  ```

  and use it to set dates as:

  ```
  date d;
  d.set ( 9, 11, 1997 );
  ```

- A better way is to define a constructor for the class that sets the value as the variable is declared as

  ```
  date d ( 9, 11, 1997 );
  ```

- A constructor is a special member function whose name is the same as the name of the structure or class

  ```
  class date
  {
      int dd, mm, yy;
      // ...
      public:
          date ( int, int, int );
          // ...
  };
  ```

```
date::date ( int month, int day, int year )
{
    dd = day;
    mm = month;
    yy = year;
}
```

  - Constructors are not real functions and are called only when the object is created
  - Once an object is created, its value cannot be modified by calling the constructor again
  - Constructors do not return anything
  - Constructors are essential in C++ to guarantee proper initialization of objects

- You can also supply more than one constructor

```
class date
{
    // ...
    date ( int, int, int );
    date ( const char *);
    date ();                  // initializes to today's date
    // ...
};

date d ( "September 11, 1997" );
date t;                       // initialized to today's date
```

- Default constructor
  - A special constructor that does not have a parameter
  - *It is always a good practice to define a default constructor*
    * C++ defines a default constructor for you if you do not define one
    * It is always better to define one than to use an implicit one
  - You can define default constructor with empty body
  - If you define a constructor with parameters, you *must* define a default constructor
    * If the default constructor is not defined, a declaration such as

      `date d;`

      makes the compiler think that you forgot to specify the parameters
    * In the absence of default constructor, you will get an error when you declare an array of objects
      · To avoid the error when the default constructor is not defined, you have to initialize each member of the array as in the following example:

        `date d[3] = { date(5,18,98), date(5,19,98), date(5,20,98) };`

- The correct constructor is automatically called, depending on the type of initializer
- Constructor may have *default arguments*

```
date::date ( int month, int day, int year = 0 )
{
    mm = month;
    dd = day;
    if ( year )    // is not zero
        yy = year;
```

```
    else
        // set year to current year
};

date d ( 9, 11 );      // Sets year to current year
```

- Default arguments (also possible in regular functions) allow us to use the same function for different number of arguments
- They relieve us of the need to create multiple functions with different number of arguments

• Constructors can create temporary objects

```
date today;      // Initialized to today's date
// ...
age = today.diff ( date ( 6, 16, 1959 ) ) / 365;
```

- This creates an unnamed temporary variable which is initialized and passed to `date::diff`, and is forgotten afterwards

• Constructors are used to initialize data members of the class type as well

```
class employee
{
    char name[30];
    date birthday;  // class type
    double salary;

    public:
        employee ( const char [], int, int, int, double );
// ...
};

employee::employee ( const char n[], int m, int d, int y, double s )
{
    strcpy ( name, n );
    birthday = date ( m, d, y );
    salary = s;
}
```

- This constructor is inefficient
- C++ says that all objects of a class are constructed before entering the `{ ... }` block of a constructor
- The above first constructs `birthday` with the constructor for `date` and then, overwrites it with the desired date
- We can direct the constructor for `employee` to invoke the constructor for `date` to initialize `birthday`

```
employee::employee ( const char n[], int m, int d, int y, double s )
    : birthday ( m, d, y )
{
    strcpy ( name, n );
    salary = s;
}
```

- You can also initialize `birthday` to be constructed from an existing date

```
employee::employee ( const char n[], date d, double s )
    : birthday ( d )
{
    strcpy ( name, n );
    salary = s;
}
```

  – Finally, you can also initialize `salary` using the same syntax

```
employee::employee ( const char n[], date d, double s )
    : birthday ( d ), salary ( s )
{
    strcpy ( name, n );
}
```

Friends

- Used to grant access to a function that is not a member of the current structure
- A global function can be declared to be a `friend` as also the member function of another structure, or an entire another structure
- Accomplished by declaring the function a `friend` *inside* the structure declaration
  - It is important to have the `friend` declaration inside the structure declaration because the compiler must be able to read the structure declaration and see every rule about the size and behavior of that data type, including the rule "Who can access my private implementation?"
  - This way you cannot declare a new class to be a `friend` of an existing class to gain access to the private members of the class
- Friends help to keep private things private
  - Used when two or more classes are designed to work together and need access to each other's implementation in ways that the rest of the world should not be allowed to have
- Example with the class of vectors in 3-space
  - Definition of the class

```
class vec3
{
    double vcoord[3];
    public:
        vec3();                         // Constructor
        double elem ( int );            // Get vector element (between 0 and 2)
        double sprod ( const vec3& );   // Scalar product
        vec3   xprod ( const vec3& );   // Cross product
}

vec3 :: vec3()
{
    vcoord[0] = vcoord[1] = vcoord[2] = 0;
}

double vec3 :: elem ( int i )
{
    if ( 0 <= i && i < 3 )
```

```
        return ( vcoord[i] );
    else
        error();
}

double vec3 :: sprod ( const vec3& v )
{
    double r = 0;
    for ( int i(0); i < 3; i++ )
        r += vcoord[i] * v.vcoord[i];
    return ( r );
}
```

– Let us also define a matrix class in the same vein

```
class mat3
{
    double mcoord[3][3];
    public:
        mat3();                         // Constructor
        double elem ( int, int );       // Get specified element
        // ...
}
```

– The access function `elem` provides safe access to the entries, for example

```
vec3 v;
double a = v.elem(3);  // Error
```

– The member function `sprod` does not have to worry about access checking as it has access to the private parts of the class

– Now, let us write a function that multiplies a matrix and a [column] vector, resulting in a vector

  * The function `multiply` must be a member of `vec3` class because it needs to build the result, and the public function `elem` can only look up the result, not build it

```
vec3 vec3 :: multiply ( const mat3& a )
// Multiply a and *this
{
    vec3 r;
    for ( int i(0); i < 3; i++ )
    {
        double sum = 0;
        for ( int j(0); j < 3; j++ )
            sum += a.elem(i,j) * vcoord[j];    // this->vcoord[j]
        r.vcoord[i] = sum;
    }
    return ( r );
}
```

  * Each call to `a.elem` checks the bounds before handing out the value leading to inefficiency
  * If `multiply` were a member function of `mat3`, it could bypass `elem` and directly access `a.mcoord[i][j]` and `v.vcoord[j]`, but a function cannot be a member of two classes
  * We overcome the problem by having `mat3` declare the function as a `friend`

```
class mat3
{
    double mcoord[3][3];
    public:
```

```
            mat3();                        // Constructor
            double elem ( int, int );      // Get specified element
            // ...
            friend vec3 vec3::multiply ( const Mat3& );
    }
```

– We could also write the function as a nonmember and declare it as a `friend` in both classes

```
vec3 multiply ( const mat3& a, const vec3& b )
{
    vec3 r;
    for ( int i(0); i < 3; i++ )
    {
        double sum = 0;
        for ( int j(0); j < 3; j++ )
            sum += a.mcoord[i][j] * b.vcoord[j];    // this->vcoord[j]
        r.vcoord[i] = sum;
    }
    return ( r );
}
```

- A complete class can be declared as a friend of another class as

```
class mat3
{
    // ...
    friend class vec3;
}
```

Class interfaces

- The class designer has to provide enough methods to satisfy the legitimate needs of a class user, yet hide all the details that change when the data representation is modified
- Let us look at some classes
  – A *safe* integer array type, with bounds checking and arbitrary starting index

```
class int_array
{
    // ...
    public:
        int_array ( int lo, int hi );
        int& operator[] ( int );
};
```

    * The array can be declared as

```
int_array a ( 10, 20 );
```

    * Because of overloading of [] operator, it can be used just like an ordinary array:

```
a[12] = 5;
cout << a[5];    // Error: index out of range
```

  – We can also define vector and matrix classes with arbitrary sizes and use them with standard mathematical operators, almost as if they were the built-in types

```
class vector
{
    // ...
    public:
        friend vector operator+ ( const vector&, const vector& );
        friend vector operator- ( const vector&, const vector& );
        friend vector operator* ( const vector&, const vector& ); // Dot product
        friend vector operator* ( double, const vector& );     // scalar product
        friend vector operator* ( const vector&, double );     // scalar product
        friend vector operator* ( const matrix&, const vector& );
        friend vector operator* ( const vector&, const matrix& );
        double length ( void );                            // Distance from origin
        double& operator[];                                // Access coordinates
}
```

- Develop the matrix class along same line and have declarations for +, -, dot product, scalar product, matrix multiplication, identity matrix, inversion of matrix, determinant, and access row vector

Unions

- Available in C++ as a space saving device but preferable to avoid using these in favor of derived classes
- Just like structures and classes, a union name also becomes a type name
- Unions can have member functions, constructors, and destructors
- Unions can be initialized with a type-sensitive constructor

```
union value
{
    int     ival;
    double  dval;
    char   *sval;

    value ( int i ) { ival = i; }
    value ( double d ) { dval = d; }
    value ( char * s ) { sval = new char[strlen(s)+1]; strcpy ( sval, s ); }
};


value v(12);    // Calls value(int) constructor
```

- Anonymous unions
  - New to C++
  - Consider the following structure

```
struct data
{
    char *name;
    char type;
    union
    {
        int ival;
        double dval;
```

```
        char *sval;
    }
}

data c;
```

– The notation `c.dval` directly accesses the value in the union and it is not required to have a name for the union