Table of Contents

- Rationale for Classes
- The C++ Class Type
- A Simple Date Class
- Method Definitions
- Implementation Organization
- Building on the Methods
- Additional Methods
- Taxonomy of Methods
- Default Constructors
- Default Constructor Use
- Multiple Constructors
- Overloading, Briefly

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 tend 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.

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

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.

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

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;
};

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:

```cpp
DataType Today, Tomorrow, AnotherDay;
```

As usual, 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 `DataType` 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:

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

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

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

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

```cpp
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 disambiguate 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;
}
```

Note that, as a member function of class `DataType`, `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.
For separate compilation, a typical organization of the class implementation would involve two files:

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

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

Then the user would incorporate the DateType class files as follows:

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

//DateType.h
class DateType {
  ...
};

//DateType.cpp
#include "DateType.h"
int DateType::MonthIs() const {
  return Month;
}
...
```

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:

```cpp
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:

```cpp
// add to DateType.h:
enum RelationType { Precedes, Same, Follows };

RelationType ComparedTo(DateType dateA, DateType dateB);

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

Then, assuming the declarations and initializations from slide 7:

```cpp
if ( Tomorrow.ComparedTo(AnotherDay) == Same ) {
    cout << "Think about it, Scarlett!" << endl;
}
```
**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 (page 64):

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

**Transformer (mutator)**
- an operation that changes the state of one, or more, of the data members of an object

**Observer**
- 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 DataTyper declaration, Initialize( ) is a constructor while YearIs( ), MonthIs( ) and DayIs( ) are observers.

**Default Constructors**

The DateTyper class has a 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 default 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:

```cpp
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 declared; if the constructor requires any parameters they must be listed after the variable name at the point of declaration
Default Constructor Use

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

```
DateType aDate(10, 15, 1998);
DateType bDate(4, 0, 1999); // set to 1/1/1800
```

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!
```

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 Modulus( ) {
   return (sqrt(Real*Real + Imaginary*Imaginary));
}
```

```
Complex x(4.0, -1.0); // 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?
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 or at least significantly different types of parameters.