An Introduction to Object Oriented Programming in C#

Table of Contents

Disclaimer

As this is an introductory article to Object Oriented Programming (OOP), I've made various simplifications, and even told a few outright lies (which shouldn't impact the beginner).

So take this with a grain of salt. But hopefully a very small one. And don't be amazed if, as your knowledge of OOP and C# grow, you find a few discrepancies between this article and the real thing.

Also, whenever I mention OOP, I mean it as it is implemented in C# and the .NET Framework. Other languages implement OOP using different syntax and semantics, but in this essay we'll focus exclusively on C#/.NET.

Overview

OOP is reminiscent of Structured Programming (SP), in the sense that it is

- A set of language features (e.g. in SP, *if/then/else*, *while* loops, etc.) that can make programming easier, more powerful, easier to debug, etc.
- A set of guidelines (e.g. in SP, don't use *goto* statements) of how to organize the data in your programs. This is called Object Oriented Design (OOD).
- And there's nothing you can do in an object-oriented program that you couldn't do without the features. It's just that OOP can make the programmer's life easier.

The two main goals of OOP are to let the programmer extend the base language (e.g. raw C#) into a more powerful one, and to avoid some of the sources of bugs that traditional (non-OO) languages have.

OOP is based on three main concepts. They are

- Encapsulation
- Inheritance
- Polymorphism

I'll define these later in this document.

Anecdote

Let me start with a personal anecdote. Years ago, I worked on a project that was written in traditional C, and designed to be implemented in three stages. When we finished Phase 1, we brought a few more programmers in to work on Phase 2, including a guy named Sam. They, of course, had to come up to speed on what we original programmers had done in Phase 1.

My part of the job had been to design and implement all the utility subroutines we needed (e.g. buffer management). One day I was walking by Sam's desk and noticed that he was looking at the source for the buffer headers. Since it was my code, I asked, purely out of curiosity, why he had source for the headers on his screen. He explained that he planned to write a routine that wanted to use the information in the headers. *Red Alert!* It turned out that we team leaders had decided that we needed a more sophisticated buffer management routine for Phase 2 and that I'd have to significantly modify the headers. Of course Sam knew nothing of this.

The good news is that I was able to tell Sam we were going to change things, and that he couldn't rely on the Phase 1 data structure, and I wound up writing his proposed routine myself, incorporating it into my Phase 2 design. But it was pure chance that I noticed him looking at the buffer headers as I passed by his desk. Otherwise we wouldn't have caught this problem until much later, when I finally modified the headers.

Traditional Problems

This anecdote illustrates several problems with traditional languages.

- In a C data structure (which the language calls a *struct*), there's no way (other than comments) to stop people from reading and writing any of the fields in the struct.

- A bit more subtly, the inner workings of a buffer (in this context) were nobody's business other than the owner of the buffer code. All programmers could *use* the buffers, but normally shouldn't be able to modify (or, arguably, even *see*) the implementation.

OOP provides ways to address these issues.

Another problem that OOP can help with addresses the problem of undefined variables. When data structures are defined by the programmer, he would normally define a *constructor* that properly initializes all the fields.

There are more advantages in OOP, but these examples will give you a start.

Classes

The basic component of OOP is the *class*. This defines a data structure (what OS/360 called a *control block*) consisting of named fields and their types, and gives the class a name. For example, an Employee might be defined as

```
class Employee {                  // The class is named Employee
       string       FirstName;
       string       LastName;
       string       SIN;          // in US, SSN
       string       Address;      // Would normally have City, etc, but
                                  // we'll simplify things
       int          EmployeeID;
       // etc
}            // Ends the class Employee definition
```

Note that a class just defines the fields. A class definition, by itself, takes up no storage. However we can create an instance of a class using the *new* keyword. For example,

```
Employee Wes = new Employee();    // Allocate memory on the heap
Employee Pat = new Employee();    // Allocate more memory
```

But there's an obvious problem here. None of the fields in the two instances are initialized to much of anything (strings are, by default, initialized to *null*, and int's to 0, but that's it). So we could try the following,

```
Wes.FirstName = "Wes";            // WRONG!
```

The bad news is that this doesn't compile. By default, all fields in a class are *private*; they can't be accessed (except in a way that we'll get to soon). Note that this addresses Sam's problem. Even if he can see the source code for the class, (by default) the programs he writes can't access the fields.

So we'll define a *constructor* for that class, as follows.

Class Constructor(s)

```
class Employee {                  // The class is named Employee
       string       FirstName;
```

```
        string        LastName;
        string        SIN;          // in US, SSN
        string        Address;
        int           EmployeeID;
        // etc

        // Define a constructor for the class. We know it’s a constructor since its
        // name is the same as the class name (Employee). Note that it’s public, so
        // a programmer can use it.
        public Employee(                  // Parameter list follows
               string        FirstName,
               string        LastName,
               string        SIN,
               string        Address,
               int           Employee_ID) {       // Note underscore
               // End of parameters, beginning of code

               // Note: Since I’ve called most of my parameters the same
               // as the fields in the class, there’s a symbol table
               // ambiguity. We’ll address this by using the this keyword,
               // which references the current instance of this class.
               this.FirstName /* Instance */ = FirstName /* Parameter */;
               this.LastName = LastName;
               this.SIN      = SIN;
               this.Address  = Address;
               EmployeeID    = Employee_ID;      // this not needed, names
                                                 // are different. But this.
                                                 // would be OK
        }
}
```

A constructor is often referred to, in short, as a *ctor* (see-tor).

Notice something very important.

We’ve defined a method (i.e. function or subroutine) ***inside*** the definition of the class. As we’ll see later, this is our first glimpse of *Encapsulation*.

So now we can write

```
Employee Wes = new Employee(“Wes”, “Rushton”, “111-111-11”, “14 Lady Sarah”, 1);
Employee Pat = new Employee(“Pat”, “Rushton”, “222-222-222”, “14 Lady Sarah”); // Bad
```

There are several things to notice here.

- In our first example, we could write *Employee Wes = new Employee();* and the fields would be initialized to their default values of *null* or 0. But now that we’ve defined at least one constructor, we’ll get a compile-time error if we tried it. In other words, you *can’t* create an instance of an Employee without giving it values[1].

[1] Yes, you could give them values of *null* and 0. Which may well be valid for your application. But at least you’ve set them to something specific.

- Inside our constructor, we can do much more than just assigning parameters to instance fields. For example, we could do error checking. Note that Wes' SIN is in the wrong format. We could add checks to see if it is exactly 11 characters long, that the two hyphens were in the right place and that all the other characters were numeric. And so on. And if any of these checks failed, in our ctor we could *throw* an *Exception*, which the calling program could *catch* and report, for example, an error in a data entry screen.
- Pat's ctor has the wrong number of parameters. As written, she'd get a compile-time error that there was no ctor that matched her parameter list. So let's add a second ctor, as follows.

```
class Employee {              // The class is named Employee
   string       FirstName;
   string       LastName;
   string       SIN;          // in US, SSN
   string       Address;
   int          EmployeeID;
   // etc

   public Employee(
         string       FirstName,
         string       LastName,
         string       SIN,
         string       Address,
         int          Employee_ID) {

         this.FirstName = FirstName;
         this.LastName  = LastName;
         this.SIN       = SIN;
         this.Address   = Address;
         EmployeeID     = Employee_ID;
   }

   public Employee(                  // Second ctor, with different parm list
         string       FirstName,
         string       LastName,
         string       SIN,
         string       Address) {

         this.FirstName = FirstName;
         this.LastName  = LastName;
         this.SIN       = SIN;
         this.Address   = Address;
         EmployeeID     = GenerateNewEmployeeID();     // Totally fictitious
                                 // routine that (presumably by referencing a
                                 // database), creates a new, unique, ID. Note
                                 // that I haven't bothered defining this
                                 // method anywhere, so this would generate a
                                 // compile-time error.
   }
}
```

The .NET Class Library

Pretty well all programming languages come with runtime libraries that give you ready access to subroutines to manipulate files, to find square roots, etc.

The .NET Framework is a *class library*, like a traditional subroutine library except that all the routines are implemented as classes (with various methods associated with each). Someday I might write a program that finds all the .NET classes on the system and produce a report telling how many classes there are and the number of methods they contain. I'm sure there would be tens of thousands of methods, maybe even pushing 100,000!

The Framework deserves its own document, but here are just a very few of the classes you have available.

String – the class the *string* data type is based upon. You can

- convert strings to upper or lower case (.ToUpper() and .ToLower())
- Do substrings (of course!) with .SubString()
- Check to see if a string starts with (.StartsWith()), ends with (.EndsWith()) or contains (.Contains()) a specified string
- Pad a string on the left or right to extend it to a specific length.
- Find the last place inside a string where a given string occurs (e.g. find the last \ in a path name to find the start of the file name.
- Do simple parsing, for example, taking a string and using .Split() to return an array of strings according to the delimiter you specify. For example, splitting on a blank to return the words in a sentence. Or do more sophisticated scanning with the Regex() (Regular Expression) class.

File – Create, Open, Close, Delete, Read, Write, Append, Replace, Read/Set file attributes (Creation Time, etc). All the usual suspects (and more).

Graphics – 2D and 3D, Rectangles, Ellipses (and thus Circles), Lines, Gradients, Bitmaps, Images, Fonts, Icons, Animation, etc, etc, etc

Multi-Threading – Start/Stop threads, Synchronization Routines (Semaphores, Mutual Exclusion, etc, etc, etc) and so on.

Database – SQL Server, Access and more

Windows Forms – Pretty well all aspects of putting a form together with controls (Buttons, Label, Combo Boxes, DateTime Pickers, and, as usual, more.

And this doesn't even properly scratch the surface. As I said, .NET comes with thousands of classes and tens of thousands of methods. In fact, no one person would likely even *hear* about, much less use.

The *var* Keyword

This isn't part of OOP, but let's simplify a bit of our coding with the var keyword. We've written

```
Employee Wes = new Wes(…);              // ctor parameter list not shown
```

This is a bit redundant. The initializer (to the right of the =) is of type Employee, so the variable we're defining (*Wes*) is presumably also of that type. So we can write

```
var Wes = new Employee(…);
```

The *var* keyword tells the compiler that the type of the variable we're defining is the same type as the (mandatory) initializer. So this is *exactly* the same as

```
Employee Wes = new Wes(…);
```

This comes in especially handy with complex data types. For example if you wanted to create an initially empty hashtable (which .NET calls a Dictionary) that mapped a string representing an ISBN into a (class) Book, you might write

```
Dictionary<string, Book> Books = new Dictionary<string, Book>();
```

With *var* you can simplify this down to

```
var Books = new Dictionary<string, Book>();
```

Finally, unlike some languages (especially JavaScript), the *var* keyword doesn't imply that the type of the variable can change at runtime.

For example,

```
int i = 5;
var i = 5;    // 100% the same as the previous line. And will even give a compile time
              // error about trying to redefine the variable i, as if you'd repeated
              // "int i = 5;"
var x = 10.0; // Double precision floating point
x = "Hello";  // Compile time error, trying to assign a string to a double
```

Encapsulation

OK, so we can *instantiate* (i.e. create a new *instance* of the class on the heap) our Employee class, but since the fields are private, code outside the class can't do anything with it. So what functionality would you like? Let's do two things, just to give you the idea. Let's

- Give the ability to change the LastName field
- Retrieve the base person's data into a string (perhaps for displaying)

Add the following code inside the Employee class.

```
public string SetLastName(string NewLastName) {
       LastName = NewLastName;
}
```

It’s as simple as that.

Or we could put in a sanity check.

```
public string SetLastName(string CurrentLastName, string NewLastName) {
       if (LastName != CurrentLastName) {
              throw new Exception(“Last name doesn’t match”);
       }
       LastName = NewLastName;
}
```

Going back all these years, we could have written (if C# had existed then!)

```
Pat.SetLastName(“Miller”, “Rushton”);
```

Note: How does the *SetLastName* method know which FirstName field to try to modify, since it doesn’t know about *Wes* or *Pat* (or any other instances)? The compiler passes a pointer to the instance as the first (hidden) parameter to each method. The keyword *this* references that hidden pointer. So, for example, *LastName = NewLastName* is implicitly treated as *this.LastName = NewLastName*.

Put another way, when you write

```
Pat.SetLastName(“Miller”, “Rushton”);
```

the compiler internally rewrites that as

```
SetLastName(&Pat, “Miller”, “Rushton”); // “&” is the AddressOf operator
```

Note that we’ve added the *public* qualifier to the method. A user of the class can see (and thus use) this method. Note that fields (and later, properties) can also be declared *public*, so users of the class can see (and even change) those fields. For small projects, especially personal ones, I tend to make most/all fields public.

To retrieve the instance data into a string, we could write

```
public override string ToString() {
       string result = FirstName + “ “ + LastName;
       result += “\r\n” /* Carriage Return, LineFeed (aka NewLine);
              // The += operator (for strings) concatenates
       result += Address;
}
```

Note the *override* keyword. All classes *inherit* from the master class called *object*, which has a ToString() method. We want to replace this with our own routine, so we must tell the compiler we’re *override*-ing it. More on inheritance soon.

Encapsulation – Bottom Line

When we define a data structure (a *class*) it presumably represents a thing that can do, well, things. A *car* object should be able to, what,

- Turn the ignition on and off
- Accelerate
- Brake
- Steer
- Change gears
- Lock and unlock the doors
- Raise and lower the windows
- Add gas
- Return various aspects of the car
    - It's current speed
    - Fuel level
    - Make and model of the car
    - Curb weight
    - Current distance travelled
    - Current fuel consumption
    - etc
- etc

In Object Oriented terms, we'd define a class called *Car*, perhaps with a ctor that specified the make and model of the car, its curb weight, and so on. And we'd define methods to turn the ignition on or off, accelerate, brake, etc, etc, etc.

The idea here is that we want to define *exactly* what a Car is, and what it can do, preferably in a nice single file inside a single set of { braces }.

Being able to do so helps the programmer conceptually. You know what a car is and what it can do, all wrapped up with a pretty blue bow!

To take a totally different example, a *Process* object could represent an operating system process. You could use such a class to

- Start a new process (perhaps with command line parameters)
- Stop it (cleanly with a shutdown message, or just abort it)
- Raise and lower its priority
- Get its current CPU time (kernel and user)
- Get its current working set size
- And so on

While some (perhaps even most) people think that Inheritance is the most important aspect of OOP, I find that encapsulation is the thing that helps me most. In a well-designed program, all of the Car (Employee, Process, whatever)-related methods aren't all over the place in multiple files and I can deal with all the concepts in my program one at a time. I don't have to keep track of the hundreds (thousands?) of subroutines in my program. Ten of them are in the Process class, thirty in the Car class, and so on. This helps me a lot!

Properties

Again, this isn't a part of OOP, per se, but a useful language feature of C#, and I want to include it here.

Hark back to the original Visual Basic for a minute. Suppose you have a data entry form with an Age field on it. If this says the user is under 18, you, the programmer, want to require a password from a parent. A nice UI would hide the password field (and its Label) unless the user is too young. In VB you could write

```
if Age < 18 Then
       lblPassword.Visible = True
       txtPassword.Visible = True
End If
```

But wait a sec. How would just setting a field in memory go through the complex operation of making a control (which is actually a window) visible or not? There's got to be more going on here than meets the eye!

Let's switch back to C#. In addition to fields and methods in a class, you can also define *properties*.

Let's take our earlier example of wanting to change an Employee's last name. So we'll do two things…

- We'll rename our *LastName* field to *_LastName* and change all references to *LastName* to *_LastName*. This is to avoid a symbol table collision with our next step which will be to…
- Define a *property* called *LastName*.

Our class will now look as follows:

```
class Employee {
       string       _LastName;          // This is the "backing field" for LastName
       …
       public string LastName {
              get {
                     return _LastName;
              }
              set {
                     _LastName = value;  // See comment on value below
              }
       }
       // All the other stuff (fields, ctors, whatever)
}
```

Here's the explanation:

The string *_LastName* is still private (by default), but the property *LastName* is public. The *get* and *set* keywords imply the existence of two behind-the-scenes methods called *get_LastName* and *set_LastName* respectively. The body of the *get*/*set* blocks are the bodies of these methods.

So when we write, say,

```
Wes.LastName = "Rushton";
```

This is the same as writing

```
Wes.set_LastName("Rushton");
```

Note: the variable *value* in the *set* clause looks like some kind of keyword, but isn't. It's just that the parameter to *set_LastName* is always named *value*.

So now let's look at our VB Visible property. Without getting bogged down in the Win32 API, let's assume there's a *ShowWindow* subroutine that takes a handle to a Window (HWND hWnd) and a bool that specifies whether the window is visible or not. And also a WindowStatus routine that lets us know if the window is visible, maximized, minimized, normal, etc. So our Windows class would look like:

```
class Window {
        public Visible {                        // No backing field needed in this case
                get {
                        var status = WindowStatus(this.hWnd);   // Assume hWnd is set
                        if (status == WindowState.Visible) {
                                return true;
                        } else {
                                return false;
                        }
                }                       // End of get
                set {
                        ShowWindow(hWnd, value);
                }                       // End of set
        }               // End of property
}               // End of class Window
```

So what does this buy us?

- The user isn't burdened with knowing about Win32 APIs, HWNDs, etc. And he doesn't even have to know about (potential) *GetVisibility* / *SetVisibility* methods. Just a simple reference to, or setting of, the property
- If things ever had to change (e.g. perhaps going from Win32 to Win64, or even to a different operating system), the implementation details are hidden. Of course, this would also be true if there were *GetVisibility* and *SetVisibility* routines.

But there are some neat things you can do in the get / set routines, especially the latter. Suppose you have an *Age* property and somewhere (*somewhere!*) in your code it gets a negative value. Piece of cake!

```
class Person {
	int    _Age;
	public int Age {
		get { return _Age; }
		set {
			if (value < 0) {
				System.Diagnostics.Debugger.Break();	// Programmatic
									// breakpoint
			} else {
				_Age = value;
			}
		}
	}
```

So now whenever any user code anywhere sets a negative age, the program stops running and the debugger comes up. Whereupon you can look at the call stack and find out exactly where the invalid value is being set!

More generally, you can check for the validity of any value assigned to a property and perhaps throw an exception if it doesn't pass muster.

Or consider a Manager who gives you a 10% raise[2]. The code might simply read

```
	Employee.Salary *= 1.10;
```

where the *set* routine sends an email to Human Resources telling them to register the raise.

You can play similar tricks in the *get* block. Imagine our GenerateNewEmployeeID() method above. Replace the call to a method with a simple

```
	this.EmployeeID = NewEmployeeID;
```

Or get even more creative. Imagine defining a TextFile class with a property, *NextLine* . Ignoring a constructor that takes a filename and opens it, we might write:

```
class TextFile {
	public string NextLine {
		get { return CurFile.ReadLine(); }
		private set { }     // Writing to the file is not allowed
	}
}
```

Now we've abstracted away the concept of reading lines of a file. To the programmer, it looks like he's just using a variable that magically updates itself whenever referenced.

As a variant of a property, you can define the *[]* (i.e. indexing / subscripting) operator. Suppose your class was a *collection class*, consisting of multiple objects (such as a hashtable/Dictionary). Instead of writing

[2] *Niiice* Manager!

```
var val = MyClassInstance.GetValueForKey("xyzzy");
```

You could arrange things so you could write

```
var val = MyClassInstance["xyzzy"];
```

Or define a database class whereby writing

```
tblAuthors.ISBN = "978-1-61614-801-0";
```

automatically reads in the author/title/etc of that book (or throws an Exception if not found).

As usual, there's more to properties, but this will get you started.

Inheritance

Consider a Manager[3].

Let's assume we've modified our Employee class to include (along with suitable ctor's, etc)

- A *DeptID* field (referencing some *Department* class, somewhere)
- A *Salary* property, type *float*(-ing point – single precision)
- A *ManagerID* field, indicating who this Employee's boss is
- An *IsManager* Boolean field

So we want to create *class Manager*, which is exactly like *class Employee*, except that it has a GiveRaise() method.

What we *could* do would be to copy and paste our Employee class and rename the class to Manager. But that does seem awfully wasteful. More importantly, if you make a change to the Employee class, how do you ensure that the corresponding change has been made to the Manager class???

The answer is *Inheritance*. We can write

```
class Manager : Employee {
}
```

And that would make class Manager an exact copy of class Employee. We say that Manager *inherits* from Employee and that Employee is the *base class* for Manager.

Side note: By "an exact copy of *class Employee*", we mean that, by default, the inherited class (*Manager*) has full access to all fields, properties and methods of its base class. So even though this might seem like an empty class definition, the *Manager* class would have fields *FirstName*, *LastName*, etc. Also methods *SetLastName* and *ToString*, and so on.

[3] While some might argue that an individual Manager isn't necessarily human, he *is an* (see "is-a" later) Employee. ☺

But we don’t want it to be an exact copy. We want to include a new method. So we’d write

```
class Manager : Employee {
	public void GiveRaise(int EmployeeID, float Amount) {
		// Details of how to give the raise would go here, but would surely
		// include a check that a manager could only give a raise to someone in
		// his own department, that he couldn’t give himself a raise (he’s in his
		// department!), and so on.
	}
}
```

Note: The security implications are this. It’s the responsibility of the programmer to instantiate the correct class, Employee or Manager. Presumably when a user logs on, a database search will determine if he’s a manager or not. So the code might be something like

```
var EmployeeInfo = GetDatabaseInfo(EmployeeID);
Employee Emp = null;
Manager  Mgr = null;
if (EmployeeInfo.IsManager) {
	Mgr = new Manager(…);
} else {
	Emp = new Employee(…);
}
```

Note: We had to initialize *Emp* and *Mgr* to something (in this case, *null*). Depending on the path in the *if* statement, only one of them would be initialized, and the C# compiler would have complained about an undefined variable.

Another example of Inheritance can be found in a Visual Studio Windows Forms application. By default it will create

```
public partial class Form1 : Form {
	…
}
```

Here, class Form1 inherits from class Form (which is the Windows Forms class name for a window). So you get an absolute *ton* of fields and methods that apply to a generic form (window). You can then add your own methods (e.g. event handlers for button clicks, etc, etc, etc).

Finally, note the memory layout for base and inherited classes. Our original Employee class took up 20 bytes in memory[4]. A class derived from that (e.g. Manager) would also take up 20 bytes per instance. If we’d added another field (say, a double precision *double* variable), then the memory layout of a Manager instance would have the same 20 bytes (in the same order), followed by 8 bytes for the *double*.

Note that methods (and properties) aren’t duplicated for each instance; there’s no need to.

[4] Each *string* object in the class is just a pointer to the actual contents of the string (or *null* if the string isn’t initialized yet), so (in a 32-bit environment) the four pointers take up 16 bytes. Plus 4 bytes for the *int* brings the total to 20 bytes.

The upshot of this is that in almost all cases, a derived class looks like a base class (with perhaps more data appended) and can be used as such. You're allowed to say (although it would be strange)

```
        Employee /* not "var" */ Emp = new Manager();
```

Inheritance – is-a vs has-a

A common problem for beginners in OOP is how to use inheritance properly.

It's common to define a hierarchy of classes. Consider how you'd model the concept of various types of vehicles – car, truck, bicycle, sled, Ski-Doo, pogo stick, etc, etc, etc.

Well, you could start by saying that, in general, a vehicle has a number of things in common. For simplicity's sake, let's pretend that the only things all vehicles have in common are their weight and the number of people they hold. And since we're talking about fairly high-level class concepts, I won't worry about constructors, methods, etc.

```
class Vehicle {
        float  Weight;              // In pounds, kilograms, whatever
        int    Capacity;            // Number of people it holds
}

class Car    : Vehicle { }
class Truck  : Car { }              // Well, a truck is (sort of) a car
class SkiDoo : Vehicle { }
class Bike   : Vehicle { }
```

etc.

Now most of these classes (e.g. not SkiDoo) have the concept of Tires. So you might be tempted to write

```
class Tires {
        int    NumberOfTires;
        float  Size;                // In inches or whatever
        // etc
}

class Car: Vehicle, Tires {
        // So now class Car might have fields Weight, Capacity, NumberOfTires and Size
}
```

This doesn't work for two reasons.

First, *Multiple Inheritance* is supported in C++, but has shown to be difficult to use properly. So other languages (C#, Java, etc) have avoided it and approached it in other ways (not described here). This definition of *Car: Vehicle, Tires* would not compile.

More importantly this is a conceptual problem. When you derive Manager from Employee, that's fine because a Manager is a special kind of ("is-a") Employee. But a Car isn't a special kind of Tire! Yes, a Car has ("has-a") a set of Tires, but isn't itself a Tire.

Similarly, you wouldn't derive class Employee from class Department. Yes, an Employee is associated with a ("has-a") department, but an Employee isn't (not "is-a") department.

Polymorphism – Introduction

Which means, of course, "many shapes". It's syntactic sugar for function pointers, without the programmer having to understand pointers (much less pointers to functions).

Personally, I don't find myself using this feature very often. But when I need it, it's very handy.

Suppose, in these days of social networking, that you have a number of people you need to send a message to. Yeah, email would probably work, but some people don't check their email all that often and prefer to be contacted via one of the various flavors of Instant Messaging.

What you'd like is to have a class with a single SendMessage(string text) method you could call that would take care of the details of sending the message via, say, ICQ or Skype or Yahoo Messenger or WhatsApp, etc., etc., etc.

In the good old days, we might address this by having a field in a Person class that has a pointer to the relevant sending routine[5] and calling the routine indirectly through the pointer. In C# we'd use *virtual methods*.

Let's define a *Person* class that will let us send. As usual we'll omit most fields, ctors, etc.

```
class Person {
	string ContactInfo;                    // Set by ctor (not shown here)
	                                       // Email address, @Name, whatever
	virtual void SendMessage(string text) {
		SendEmail(ContactInfo, text);          // Code not shown
	}
}
class IcqUser  : Person {
	virtual void SendMessage(string text) {
		SendIcqMessage(ContactInfo, text);     // Code not shown
	}
}
class SkypeUser  : Person {
	virtual void SendMessage(string text) {
		SendSkypeMessage(ContactInfo, text);   // Code not shown
	}
}
```

Now in our mainline we might write

```
enum ImType {                          // Symbolic (and type-safe) constants
                                       // "enum" means Enumeration
	ICQ       = 0,
	Skype     = 1;
```

[5] In the *real* old days, we'd write cascading *if/then/else* statements checking for each type in turn.

```
        Yahoo     = 2;
        WhatsApp  = 3;
}

var People = new List<Person>();        // Empty list
foreach record in database {            // Semi-pseudocode
        switch (record.PreferredContactType) {
                case ICQ:
                        People.Add(new IcqUser(…));       // Note that each of these
                                                          // is also (via inheritance
                                                          // of type Person and can be
                                                          // added to a List<Person>
                        break;
                case Skype:
                        People.Add(new SkypeUser(…));
                        break;
                // Add code for other types (not shown)
                default:
                        People.Add(new Person(…));        // Contact via email
                        break;
        }
}

foreach (var Individual in People) {                      // Back to real code
        Individual.SendMessage("Party at Patty's!!!");
}
```

To give one more short example, while the details are certainly different, is writing to a disk file really that much different from writing to a remote socket? Or to a memory buffer? It's just a stream[6] of data that needs to be sent somewhere.

Polymorphism – Implementation

For each class that defines a virtual method, the compiler creates a *vtable*, an array of function pointers for the virtual functions. It populates these with the addresses of the relevant functions. Then each instance of a class with virtual functions has an additional (hidden) field added that points to the vtable. Virtual methods are called indirectly through the vtable.

Note: Memory is used efficiently. There is one vtable per *class*, not one per *instance*.

In our example, there would be 3 vtables, each with a single entry. The vtable entry for class *Person* would have a pointer to the SendMessage method in that class. The vtable entries for class IcqUser and class SkypeUser would point to their versions of SendMessage.

Without virtual methods, an instance of class *Person* (again, ignoring all the other fields that a realistic *Person* class would have) would take up 4 bytes (in a 32 bit environment) for the reference to ContactInfo. But now, with virtual methods, it would take up 8 bytes. It's as if the class were defined as

[6] The .NET framework has an abstract class *Stream* from which is derived, among others, FileStream, NetworkStream and MemoryStream.

```
class Person {
       int[]  vtable;                     // Hidden from programmer
       string ContactInfo;
       // etc
}
```

Let's assume that instead of only one virtual method defined in class *Person*, we have, say, 5 of them. So each vtable would have 5 entries in it. We'll call the new ones Method_A, Method_B, Method_C and Method_D; Suppose we now call Method_B from any of our classes. Assuming we add them after the definition of *SendMessage*, the compiler knows that Method_B is the third (subscript 2 in origin zero) method. So the generated code would look like (in pseudocode)

```
       // Method_B("Hello world");
       FunctionPointer = vtable[2];
       call FunctionPointer("Hello world");
```

Conclusion

I remember back in my 4th year *Data Structures* class[7], one of our assignments (in those early Structured Programming days) was to write a certain program without using *go to* statements. This was all very new stuff back then and was worthy of a 4th year problem.

So OOP is new to many/most[8] and, like Structured Programming in its day, requires a certain new mindset to organize your programs in this new fashion. But once you learn it, you'll probably never (if you can help it) write another non-OOP program in your life.

[7] No pun intended. Well, not much of one! ☺
[8] Actually, it traces back to *Simula* in 1967.