C# for C++ programmers

C# is roughly a mash-up of C++ and Java.

The big concept is Reference Types. Those are borrowed from Java and have spread to several common languages, so fun to understand. It's a clever system that allows implicit pointer and heap use, and enables automatic garbage collection.

C# is more verbose than C++, more comfortable adding and changing small features, but otherwise the style is difficult to pin down.

Part I is about changes/sugar from C#, Part II is about reference types, and then part III is about more changes/sugar which make sense once you know what reference types are. Finally part IV are some notes about the culture.

1 Small changes from C++, Pt I

1.1 Misc, Pt I

These are in no particular order:

- The `if(n=3)` problem was removed. C# turns that into a syntax error by having nothing implicitly cast to `bool`. You'll need to replace every `if(n)` with `if(n!=0)`, but that's not too bad.

  `if(b=true)` is still a horrific logic error, and pointers still implicitly convert null/non-null to T/F: `if(ptr)` works for `if(ptr!=null)`.

- Type-casts are backwards, the *type* must be in parens: `(int)n` instead of `int(n)`. This style was borrowed from Java.

- `char` is 2 bytes (represented in UTF16 format, which is mostly ASCII). For the times we want `char[]` as a byte-array, C# has a `byte` type.

- The dot does double-duty as the scope resolution operator. Our `c1.age = Cat::maxAge` becomes `c1.age = Cat.maxAge`. As an even more horrifying example, `Cat::primeCat.age` becomes `Cat.primeCat.age`.

  The idea is beginners can learn a single "inside of" operator.

The double-colon operator wasn't removed, but it was repurposed. `global::` can begin a path. It's called the namespace alias qualifier, if you want to look it up.

- There are no `typedef`'s. But the `using` keyword is overloaded to do it in a limited way: `using intList=System.Collections.Generic.List<int>;` (notice the dots used for scope resolution, again). But it's only file-wide and won't work for basic types: `using tailLen_t=int;` is an error.

  Oddly, there's a typedef syntax (`delegate`) for function pointers.

- No macros. `#define sqr(x) (x*x)` isn't allowed.

- Using an uninitialized variable is a *compile* error. The compiler is aggressive about this, sometimes flagging variables as uninit'ed when it can't follow your logic.

  Similarly, passing a potentially uninitialized variable to a function is a compile error. And so is passing a struct with any uninitialized fields. It doesn't matter whether the function attempts to use them.

  The goal is that a `C#` program will never read an old garbage value, or get those horrible intermittent errors from that.

- `var n=8;` or `var w="cow";` infers the type, similar to `C++`'s `auto`. Essentially the compiler replaces `var` with the correct type, which must be immediately known.

  It's meant to be an obscure feature, but in `C#` code you'll see it a lot, pointlessly replacing short type names.

- Namespaces can have only classes and enums - no variables or functions. You're expected to put those as `statics` in dummy classes:

```
class stringFuncs { // dummy class
  public static string bestWord;
  public static int compareSomehow(string w1, string w2) {... }
}
```

  Splitting things this way is a pain. And a reminder, these are used with a dot: `stringFuncs.bestWord="cow";`.

- `using`'s are limited. They can only be at the top of files and can't pick out individual items. The best you can do is `using System.Collections.Generic;`. You can't add dot-LinkedList to that.

  `using`'s can grab from namespaces or classes, but for classes you need an extra word: `using static stringFuncs;`. Arg.

So you know, there's a third completely different overload for `using`. Surrounding code in a `using`-block will immediately run destructors when exited.

- There are no persistent local variables:
  `void A() { static int n=0; ...}` doesn't exist.

  That's not really a problem - you're encouraged to write a small class with member variables as your persistent store.

- `C#` has pointer use in exactly the way `C++` does. But it can only be used in UnSafe mode, and your environment never supports unsafe mode. So, effectively, there are no explicit pointers. Reference Types handle this instead.

- Array changes:

  - Square-brackets are part of the type. This is pretty nice. Ex's:

    ```
    int[] A, B; // two arrays of ints

    // likewise parms and return values:
    int[] func_takeArray_returnArray(int[] A) { ... }
    ```

  - They know how long they are: they come with a `Length` member function. You never have to pass (array, actualLength) again.
  - Indexes are ranged-checked. `A[n]` throws a run-time error if you go outside. I know, that automatically makes the language be for children. But in practice it's nice for quick concept code – you can use arrays stupidly and get a nice exception and stack trace.
  - Besides `[][]` arrays, there are square multi-dimensional arrays. Ex:

    ```
    int[,] A = new int[4,9];

    int len0=N.GetLength(0); // 4
    int len1=N.GetLength(1); // 9
    int totalSlots=N.Length; // 36; total # of boxes (but why?)
    ```

  - Ragged arrays are backwards. They aren't just composition of the normal array rules to make an array of arrays. They have extra rules making them always read left-to-right. This example makes a size 4 array of arrays:

    ```
    int[][] A; // so far, so good
    A = new int[4][]; // ?? yes, this is correct
    A[0] = new int[9]; // no more surprises
    ```

    An example (stolen from StackExchange) clarifies this backwards rule even more. `int[][,]` reads *left-to-right* as a single array of square arrays:

```
int[][,] A = new int[4][,]; // <-yikes, backwards
print(A.Length); // 4

A[0] = new int[4,9];
A[1] = new int[5,2]; // diff sizes since still ragged
```

- Function calls can optionally add the parameter name in front of the argument. `max(3,4)` and `max(x:3, y:4)` are the same.

  Sometimes this is used as self-commenting code, or to flip the order for fun:

  ```
  void A(int cats, int dogs) { ... }
  A(dogs:5, cats:2); // same as A(2,5). Note dogs are first
  ```

  A real use is pick one parm from a long list of optionals:

  ```
  void doStuff(int a=0, int b=0, int c=0, int d=0) { ... }
  doStuff(d:5); // use the defaults for a,b,c, while filling in d
  ```

- Function pointers work mostly the same as in `C++`. This declares `ff` as a pointer to an `int(string,string)` function:

  ```
  System.Func<string, string, int> ff; // same as int (*f)(string,string);
  ```

  Note how the return type is in the back, not the front. And also it's not `System::Func`, like we'd do it. But they're used the same, except you don't (and can't) add a reference or dereference:

  ```
  ff = charsInCommon; // assigning a function to ff
  x=ff("cow","wookie"); // possibly "ow"
  ```

  That's enough to get the idea, but there's more on function pointers if you want:

  The "return-type-goes-last" syntax doesn't work when there's no return type (you can't write `<string,string,void>`). So `Sytem.Action` is the same as `System.Func` except the return type is understood to be void. Here `f2` returns nothing:

  ```
  System.Action<int> f2; // same as: void (*f2)(int);
  f2=beepNTimes;
  f2(3);
  ```

  That won't work if there aren't any inputs. Instead leave off the angle-brackets: `System.Action f3;` for a void(void) function.

  That's sort of a mess. There's also an older syntax where you were required to first create a typedef, using `delegate`:

```
delegate int strCmp_t(string a, string b);
// the same as: typedef int (*strCmp_t)(string,string)

strCmp_t f2; // delegate works like a typedef
```

Both ways are fully interchangeable. Because of this older syntax, `C#` function pointers are called **delegate**'s.

- It has anonymous functions (called Lambda functions, but they aren't – there's no context capture.) An example:

  ```
  gg =   (x,y)=>{ return x*2-y; };
  // assume gg is a correct function pointer var
  ```

  Notice how the function body is normal. For the part in front: the parameters never have types, only names, since it can always infer the types. The return type is also inferred. The `=>` is the anonymous function keyword.

  If the body is a single return statement, a shortcut allows you to leave out the `return` and the curly-braces:

  ```
  gg =  (x,y)=>x*2-y;
  ```

  There's another obsolete syntax with a repurposed `delegate` keyword. There's no reason to use it. But you'll see people using it.

- The uses of `C++` template functions are split over two language features. **generics** looks like C++ templates but without "duck typing". The body can't assume the type has anything except `=`. This is obviously a pretty big limitation.

  You can specify the type must be a class. Now the body can use `==` for pointer-compare. That's still not super-useful. Finally you can limit the input to any subtype of one particular class. That can be used to search a list of Animals for only Dogs or Cats, something like: `T aa = Animals[i] as T;`.

  Obviously, you can't do any of the useful stuff `C++` template functions allow.

  The second implementation of template functions is the `dynamic` keyword. It turns the variable after it into a template type:

  ```
  // this works for anything which has a Length:
  bool longEnough(dynamic n) { return n.Length>10; }
  ```

  The limit here is there's no clean way to enforce two inputs being the same type, or to declare a local of that type (I think there's a work-around using `typeof`).

- Hiding an outer-scope variable with an inner isn't allowed. To be clear, this is turning bad style into an error:

```
int i=8;
for(int i=0;i<10;i++) {} // C# error, inner i hides outer i
```

- Limited "real" reference variables: `int& n = C[i].age;` isn't allowed. But you can declare `ref int n` and assign it exclusively using return-by-reference functions:

```
// assigning reference var using return-by-ref function:
ref int n = ref getAgeRef(c1); // same as: int& n=&c1.age;

// a return-by-ref function:
ref int getAgeRef(Cat cc) { ref return cc.age; }
```

  Notice all of the redundant `ref`'s. They're required. A limitation is the referenced item must be on the heap (but it can be a value-type inside of a class, like `age` inside our Cat).

- There are no header files or prototype functions. But there's also no need. C# environments rummage around all your files to find stuff.

- The container classes: `List` replaces `Vector`. It's backed by an array. Oddly, it has a few `O(n)` member functions, and no pop_back. `LinkedList` is the `List` class. The hash-table `map` class is renamed `Dictionary`.

  The `ArrayList` class is an old-style Java mixed-type array. It holds a list of `Object`, which is the base class of everything. We don't like it.

- There's a `foreach` loop shortcut which iterates through arrays and most containers:

```
// W is an array, or List, or LinkedList:
foreach(string w in W) { if(w=="cow") print("moo"); }
```

  It works on any type which can return an external iterator (really, anything which inherits from the standard "ask me for a forward iterator" class).

- Serialization. Without too much trouble, any class can be written out as XML, then read back from it (I've never done it directly, but I assume it's simple). Generally your GUI will make use of it.

  Standard Serialization assumes your fields are simple "owned" data and works down serializing everything down the line inside of you. If you have data structures using pointers you'll have to work a lot harder.

- Reflection. The function `GetMethods` on a class returns a list of all function signatures and names, even private ones. You can then call them through those handles.

  This is useful for weird stuff, like making a crude visual editor for any class, or reading a class you didn't know about at compile time. You'll also sometimes see set-ups where code impossibly uses private variables and functions – they're cheating with reflection.

  Reflection is no help doing things you already know how to do.

1.2 Class/struct, Inheritance changes, pt-I

For structs and classes, the default access modifier is `private`. There's also no fall-through for `public` – you have to add it to each. A plain-data struct looks like this:

```
public struct Dog {
  public string name;
  public int age; // without this 2nd public, age would be private
}
```

This is another consequence of coming from Java. The idea is that proper OOP should make you work to make encapsulation-breaking public variables.

All member variables in a class are auto-inited, sort of. The default constructor does it. But the system stops adding it as soon as you write any other constructor.

A class is allowed to assign starting values inside the member var definitions. The compiler secretly moves them into every constructor (even the default.) Ex:

```
public class spot {
  public int n; // initialized to 0, but only in default constructor
  public int x=99; // runs x=99 in any constructor
  public string w=getNextName(); // even this. Also run in constructor
}
```

With inheritance, the dynamic cast looks like `baseClass as subclass`. An example:

```
Animal aa; // superclass of Cat
Cat cc = aa as Cat;
if(cc!=null) // we have a Cat
```

Multiple inheritance from classes isn't allowed. A work-around from Java is inheriting from any number of "interface" classes. They're pure contracts: abstract classes with only abstract virtual "=0" functions:

```
// C++ version:
class Worker {
  public:
  virtual int doWork()=0;
}

// equivalent C# version:
interface Worker {
  int doWork(); // assumed to be public, virtual and abstract
}
```

All they can do is require virtual functions. Interfaces can't have their own variables, or provide any useful code - no default code and no helper functions.

`C#` style is to name Interfaces with a leading capital-I. That makes you think interface and class pointers aren't fully interchangeable, but they are. Interface types can be declared or used as parameters the same as any other super-class.

Since you only inherit from one real class, the syntax to jump into your base class is simplified. Instead of `animal::cost()` use `base.cost()` (there's no syntax to look inside an inherited interface, since they can't have anything worth looking at).

Setting up virtual functions requires one extra step. As usual, write `virtual` in front in the base class. That says it *may* be virtual. The extra step is `override` in front of the subclass function:

```
class Animal { // base class, nothing special here
  public virtual eat() {} // new part: this _might_ be virtual
  public virtual sleep() {} // ditto
}

class Cat : Animal {
  public override eat() { ... } // this one is virtual
  public sleep(); { ... } // cat.sleep is NOT virtual.
}
```

You're suppose to write `new` in front for non-virtual subclass functions. But it's the default so no one ever does.

Notes: 1) yes, this is re-using `new` for a completely different purpose. 2) this is not a useful feature - just write `override` everywhere. 3) The official reason is that cat.eat may have started as a normal function in a subclass. Later, independently, the owner of Animal adds a virtual eat function. This rule helps flag how the old cat.eat probably doesn't satisfy the post-condition of virtual animal.eat, so shouldn't override it.

There are no `friend` functions.

Extension methods: this is pure sugar: normal functions which are called like member functions. They can't use private variables, and can't be virtual – they're regular functions in every way except the calling syntax. To write one, add `this` in front of the first parameter. Ex:

```
// 1st parm "this Cat" makes it an extension function:
public static passTime(this Cat c, int days) {
  ... // sleep and eat?
  // NOTE: we're limited to public Cat members
}
```

`c1.passTime(4);` calls it. The only useful thing this does is let you type c1-dot and search a dropdown. And yes, the `this` in the parm-list is another re-purposing of a keyword.

Properties: these are shortcuts for writing getter/setter pairs. Here they create a fake years variable:

```
class Cat {
  public int monthsOld;

  public int years { // syntax for get/set based on years
    get { return monthsOld/12; }
    set { if(value<0) value=0; monthsOld=value*12; } // value is a keyword
  }
```

You use `years` as if it was a variable. Instead of `n=c1.getYears();` and `c1.setYears(4);`. we can write `n=c1.years;` and `c1.years=4;`. The system calls `set` for L-values, and `get` for everything else.

These are very popular. This is considered the proper way to create a simple `age` variable:

```
class Cat {
  int _age; // private
  public int age {  get { return age; } set { age=value; } }
```

Do-nothing gets/sets are so popular that they have a shortcut - "auto-properties". `public int age { get; set; }` creates a backing var and the (hidden) code above.

The official reason for uselessly turning every public variable into a get/set pair is an old linker bug. Converting a variable into a property should be transparent, but the old bug could forget to recompile client code.

You can safely never use properties, but `C#` coders will regard you as a barbarian.

2 Reference Types

This is the major change from C++, borrowed from Java. It's a system intended to gloss over pointers and heap/stack, give a uniform syntax for pointer and non-pointer use, and allow automatic garbage collection.

2.1 Mechanics

Types are divided into those that can't use pointers, and others that must. The first group is simple enough: you can never have a pointer to a basic type. Pointers to ints, floats, strings, chars and bools don't exist in C#.

The other type, called reference types, are all of your classes. They can only be pointers and can only be new'd on the heap. It's impossible to declare a class normally. Because of that restriction, the language can simplify their use: you don't need any extra syntax to declare them as pointers, and they are automatically dereferenced.

Suppose Cat is a class:

```
Cat c1; // automatically assumed to be a pointer
c1 = new Cat(); // the standard way of making a usable Cat
c1.name="fritz"; // automatic dereferencing c1
```

`Cat c1=new Cat();` really is the simplest, standard way in Java and C# to create a simple Cat variable. It's making a for-real heap allocation, which seems excessive. But in C# we don't need to know anything about the heap for a while, because of automatic garbage collection.

Term-wise, c1 is called a reference. Get it? Cat is a reference type, so all cat-pointers are references. Some observations about this system:

- Pointers can never be aimed at stack data. It's just not possible with this set-up. Only "value types" can be on the stack, and it's impossible to create pointers to those.
- There's no address-of or dereference operator. No `&` `*` or `->`. We're banned from using them on normal types, and both are implicit with reference types.
- We're forced to use pointers and `new` when we didn't really want to. When we want to declare a regular "automatic" Cat, which is most of the time, we're forced to declare a pointer to a heap Cat.
- Due to the previous point, in C# we'll make what seems like a ridiculous amount of garbage. From our point of view, C# programs seem to have completely undisciplined memory management.
- No pointers-to-pointers. No `Cat**`. You fake it by making a small class holding the second pointer.

One more simplification is there's no `destroy` command. And as a bonus, there's no chance of dangling pointers. Garbage collection automatically runs every so often using the "find reachable data" method. If you point to something, you absolutely prevent it from being destroyed.

Beyond the syntax shortcut, references are pointers in every way. They can be `null`, `=` is pointer assignment, and `==` performs pointer compare:

```
Cat c1=null;
c1.name="Boots"; // crash (null reference exception)

if(c1!=null) c1.name="Boots"; // standard guard

Cat c1=new Cat(), c2=new Cat();
if(c1==c2) print("false. not pointing to same memory");

c2=c1; // pointer assign. Both are now pointing to same Cat
```

Passing a class variable to a function is passing a pointer by value:

```
void catFunction(Cat c1) {
  c1.name="Squeeky"; // use pointer to change shared cat
  c1=null; // merely changing our local copy of c1
}
```

Returning a Cat is returning a pointer. This returns a pointer to the original c1 or c2:

```
Cat longestName(Cat c1, Cat c2) {
  if(c1.name.Length>c2.name.Length) return c1;
  else return c2;
}
```

Classes as member variables are also pointers, which can fool you. In this, `c` starts as null, and needs to be created:

```
class Home {
  Cat c; // null. Someone must run: c=new Cat();
  ...
}
```

A standard linked-list node can look strange for the opposite reason. `next` looks like a nested `CatList`, but unlike `c`, is being used as a real pointer:

```
class CatList {
  public Cat c; // our Cat
  public CatList next; // merely a pointer
}
```

A big trick to reading code using reference variables is the purely mental distinction between normal vars and pointers. In this picture, in our minds, `c1` is a Cat, while `c2` is a pointer to a Cat.

```
Cat c1=new Cat(); // just a Cat. c1 considered locked-in
Cat c2=c1; // c2 intended to be used as pointer
```

Another way of showing that is how we can use `Vector<Cat*>` to pick out some items of a `Vector<Cat>`. In `C#` we have to use `List<Cat>` for both. It's not-so-obvious which is intended to be pointers.

2.2 Arrays as Ref Types

`C#` arrays are Reference Types. A `C#` array variable is a non-const pointer to the actual array on the heap. All legal `C#` array use looks like this:

```
int[] A =  new int[maxCows];
// NOTE: there's no advantage to using a constant, since arrays
//   are always allocated dynamically on the heap
int[] B = A; // B points to A
if(A==B) {} // pointer compare, true
A = null;
B = null; // array from line 1 can now be garbage collected
```

There's no pointer math with these. `B` can never point midway through `A`.

2.3 Structs and Ref Types

Java and other languages are happy with classes always being reference types. That system is fully usable. `C#` adds an unnecessary complication - the `struct` keyword defines a value-type class. For example:

```
struct Point {
  public double x, y;
}

Point p1; // finally a normal stack variable
p1.x=5; // this is fine
```

That seems pretty nice - we can use them in the normal way. But since it's a value type we can never have a pointer to one (which also means we can't have virtual functions). In practice, `class` is the preferred type - for anything you may ever want a pointer to. Making something a struct is considered a risky error-prone move in a desperate bid to increase speed.

As you may have noticed, the default for structs is still `private`. As you'd guess `=` is member-wise shallow copy, and `==` is memberwise compare.

A fun struct vs. class example; here `CatTracker` has one of each:

```
class CatTracker {
  public Cat c; // pointer
  public Point spot; // inline

  public CatTracker() {
    spot.x=0; spot.y=0; // this is safe
    c=new Cat(); // as usual, we need this for a Cat
    c.age=4;
      ...
  }
}
```

If we create `CatTracker ct = new CatTracker();` we have a heap CatTracker, holding an in-line Point `spot` and a pointer to Cat `c` which it's considered as owning.

One very odd syntax change is a *struct* constructor requires a useless `new`. This is to help beginners:

```
Point p1;
p1.x=6; p1.y=8; // this is fine, but can also use:
p1 = new Point(); // merely copying (0,0). NOT a heap allocation
```

This works since you can never create a struct on the heap. The `new` is clearly superfluous. The purpose is to let starting coders use `Thing t=new Thing();` without needing to know whether Thing is a class or struct. Either way, `t` is set-up for you.

A last issue with structs is the get-through bug. Suppose we have `c.p.x=7;` and `c` is using a `get` to return `p` (normal `C#` programmers always use a `get`). If `Point` was a class, we'd be fine - the `get` gives us a lovely pointer. But since it's a struct, the `get` returns a *copy* of `p` and the assignment to dot-x is uselessly changing that copy.

3 Misc features, Pt II

- `C#` has standard pass-by-reference. It's split into a specialized output-only version, and the normal in-out version. The normal version:

  ```
  void swap(ref int x, ref int y) {...} // both are pass-by-ref

  swap(ref num1, ref num2); // these extra refs are required
  ```

  Notice how the ampersand is replaced with `ref`. Additionally, the call must decorate variables with `ref`. The idea is to be sure the caller understands what's happening (yes, this is not as helpful as they think it is).

The output-only version replaces `ref` with `out` in both places, and changes the errors: 1) no error for being uninitialized. Recall this is normally a syntax error, 2) must assign to in the function, 3) attempting to read them is an error.

That's all there is to know. But culture-wise call-by-reference is an advanced topic. For one thing, almost everything is passed by pointer anyway, with no need for CBR:

```
void fixCat(Cat c) {
  if(c.age<0) c.age=0; // works just fine
}
```

The other problem is reference vs. reference. `C#` coders have been calling `Cat c1` a reference since day 1. This second meaning is hard to say - you're passing a reference by value, or a reference by reference, or a value-type by reference . . . it's a mouthful. At one point even the microsoft site incorrectly claimed all classes were passed by reference.

- LINQ library. This get mentioned a bit, and it's a hodge-podge.

  It has some SQL functions for directly querying a database, without needing some other API. Then it has a completely redundant, inferior syntax that reads like SQL. An example from the microsoft site:
  `var q = from c in customers where c.City == "Ral" select cust;`

  Then, lumped into the same `linq` namespace for no reason are common list-searching functions – return all items matching this predicate and so on. But none of them are allowed to modify the list, and most create objects with delayed evaluation - you often need to add `.toArray()` to the end.

4 The culture

If you're trying to read various `C#` things on the internet, some things I feel like I've figured out:

- They love new features, which is weird for a business language. The thing to know is you can read exuberant praise for a new feature, when it's a trivial shortcut.
- The manual is the microsoft `C#` site. It's incomplete in plenty of places, which apparently is considered fine.
- The internet thinks only things labelled `interface` are interfaces. Lkewise they tend to think only things labeled `get` and `set` are getters and setters.

They take hiding member variables very literally: a private int with a pass-through get/set is considered hidden.

The take away is when you're reading something about `C#` design that seems a little off, it can be helpful to think "does this person not understand OOP is a concept first, and an implementation second?"

- Features are made obsolete faster than google can keep up. You'll often find high-ranked results leading to not-very-old-pages explaining a clunky feature from 2 versions ago. Authors even get confused. You'll find pages on current features where the code also mixes in obsolete ones.