C# for C++ programmers

A short history of C#: Java came first. It's somewhat like C++, with the goals of simplifying early pointers, having a more object-oriented feel and cutting error-prone features. C# started life as microsoft's version of Java. Fairly quickly many of the features cut by Java were re-added, with changes, in stutter-steps.

C#'s dominating non-C++ feature, taken from Java, is the restricted implicit pointer system called **reference types**. These also give you free garbage collection.

Compared to C++, C# is more verbose - some keywords serve no purpose but are required anyway for readability. The error messages are a little more aggressive. Things that are usually a good idea in C++ are "do it or it's an error" in C#. The language also changes faster with less discussion. You'll see more obsolete features and inconsistent styles. Often current-seeming documentation is actually out-of-date.

This is in 3 sections. The first is small changes from C++. The second section is new features. The third is an explanation of C#'s version of reference types (which is pretty much the same as Java's.) The third section is the most important, but it's confusing so I put it last.

1 Small changes from C++

These are in no particular order:

- Implicit cast to `bool` was removed. Mistakes like `if(n=0)` are errors in C#. You no longer need `if(0==n)` to guard against a missing equals, but `if(n)` now needs to be written out as `if(n!=0)`. Pointers still implicitly convert – `if(ptr)` is fine and is the same as `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 when you need to abuse `char` as a byte, there's a `byte` type.
- The `var` type guesses the type, similar to the new use of `auto` in C++. As usual, this is just a shortcut. In every way `var n=8;` is the same as `int n=8;`.

- Single dots do double duty as the old member look-up operator and the scope resolution operator. The main effect is you'll see things like `cats.claws` and need to pause. Cats could be a class with static claws, or a boring variable with member claws.

  `C#` does have a `::` double-colon operator. But they call it the namespace alias qualifier and it can only be used to begin a path: `global::`.

- There's no official main entry point. It's some function in some class. Your environment will probably generate the class and a stub function for you, and will know to run it.

- There are no general `typedef`'s. There's a special-case typedef for function pointers (later.)

- Namespaces can't hold variables or functions. Instead classes with everything `static` are used to hold "global" data. You get things like:

```
class stringFunctions {
  public static string bestWord;

  public static bool hasThese(string w, char ch, int wantedAmt) {... }
}
```

  `using cowSpace;` only works for namespaces, but a new `using static stringFunctions;` will dump all statics in a class.

- `static` can't be used in functions to create persistent local variables.

- No explicit pointers. There are, but they can only be used in UnSafe mode, and you're never in unsafe mode. Normal pointer use is handled by implicit pointers in the Reference Type system (see part 3.)

- Using an uninitialized variable is a *compile* error. The compiler is aggressive about this, sometimes flagging variables when it can't follow your logic and thinks the var could possibly escape being inited.

  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.

- Arrays know how long they are. They come with a `Length` member function. All `[]`'s are also ranged-checked, throwing an error if outside.

- Arrays put square-brackets as part of the type (there are more differences, in part 3 on Reference Types.) This is pretty nice. Ex's:

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

// syntax for parameter and return value:
int[] getAndReturnArray(int[] A) { ... }
```

- This is pure sugar: calling a function can specify the name of the parameter, using name-colon-value. 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);
```

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

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

- Function pointers. These work mostly the same as in `C++`. They're officially called **delegate**'s, due the the original keyword. 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: return type is in back, not in front

// these are used the same as normal:
ff = charsInCommon; // an imaginary function
x=ff("cow","wookie");
```

That "return-type-last" syntax fails when there's no return type. For that you need to change `Func` to `Action`. Here `f2` returns nothing:

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

And *that* syntax fails when there are 0 inputs, so use just `System.Action f3;` for a void(void) function.

Function pointers are the only things with a typedef. It should look familiar. `delegate` is the keyword. You're required to provide dummy names for the parameters:

```
// typedef myStringCmpFunc:
delegate int myStringCmpFuncType(string a, string b);

myStringCmpFuncType ff; // once made, use as a normal type
```

The two methods are interchangable. `myStringFuncType` can be used in a `System.Func<string,string,int>`.

- Anonymous functions are like C++ Lamba functions, but simpler. The function is written as usual inside curly-braces; the keyword `=>` goes in front, preceded by the parameter names. The types (input/return) aren't written – they're obvious from context:

```
System.Func(int, int, int) gg; // start with function pointer

// assign a nameless function:
gg= (x,y)=>{ return x*2-y; };
```

  A shortcut allows you to write a a single value as the function body, with no curly-braces. That one value is automatically returned:

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

  There's one more obsolete syntax reusing the `delegate` keyword. You'll see examples using it, but these are better. In `C#` these are officially called **lambda functions**.

- The uses of `C++` template functions are split over two language features. `C#` has something that looks like C++ templates but is very, very limited. They aren't allowed to assume the type can do anything except `=`. This is obviously a pretty big limitation.

  You can specify the type must be any class, which gives you access to `==`. Still not all that useful. You can specify it must be a sub-type of some class, giving access to everything there (but then why not use inheritance?)

  The other part of template functions is the `dynamic` keyword. It turns the variable after it into a template type, but there's no way to specify some other variable must be that same type. Ex of `dynamic`:

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

- Hiding an outer-scope variable with an inner is considered error-prone coding, so isn't allowed. This is an error in `C#`:

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

- Reference parameters replace the ampersand with the keyword `ref`. Additionally, the call must decorate variables with `ref`:

```
void swap(ref int x, ref int y) {...} // &'s are now ref's, moved to front

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

A specialized version is added for output-only reference parms. The keyword is changed to `out` and the compile errors are changed: 1) no error for being uninitialized, 2) must assign to them in the function, 3) attempting to read from them is an error.

- No direct use of reference variables. Nothing like `int& n = C[i].age;` is allowed. But there are limited return-by-reference functions. They can only be used on heap variables:

```
// returns a reference to age:
ref int getAgeRef(Cat cc) { ref return cc.age; }

// "ref int ca" is like int& ca}:
ref int ca = ref getAgeRef(c1); // same as: int& c1Age=&c1.age;
```

  Notice the extra `ref`'s in front of the return and the call. Not needed, but required.

- There are no header files or prototype functions. On the one hand, you don't need to include a dot-h. `C#` searches every file, every time. On the other hand, if you want to break a class into prototypes and code, you can't (collapsing all of your functions in the IDE is about the same thing.)

- Container classes: `List` is similar to `Vector`. It's backed by an array and has some `O(n)` member functions. `LinkedList` is the `List` class. The `map` class is renamed `Dictionary`.

  The `ArrayList` class is an old-style Java array. It holds a list of `Object`, which is the base class of everything

- A `foreach` loop shortcut iterates through arrays and most containers:

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

  It simply keys off of the iterator (assumes you have one, automatically calls W.begin(), next() and checks for W.end()).

- The `using` directive is more limited. It can't be inside of anything and and can't pick out individual functions. There's a `using` *statement*, which looks like it can include individual items, but it's for something completely different.

- Square multi-dimensional arrays, using a `[,]` syntax. Ex:

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

int len0=N.GetLength(0); // 4
int len1=N.GetLength(1); // 9
int totalSlots=N.Length; // 36; confusing, since Length is for 1-D arrays
```

- 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][,];
  print(A.Length); // 4

  A[0] = new int[2,3];
  A[1] = new int[5,2];
    ...
  ```

1.1 Class/struct, Inheritance changes

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.

Using a struct constructor requires a do-nothing `new`. This is horribly confusing at first. The explanation is in the last section, on Reference Types. This typical example initializes `d1`:

```
Dog d1;
d1 = new Dog(); // assigns ("",0) (the default constructor)
d1 = Dog(); // error - the new is useless, but non-optional
```

Again, the `new` is required, but does nothing. In `C#` new doesn't always mean `new`.

Member variables in a struct aren't initialized; the ones in classes are. Additionally, you're allowed to assign starting values in the class declaration. The compiler secretly moves them into the constructor. Ex:

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

As you might guess, the C++ initializer syntax is gone. There's no `public spot(): x(99)`. The "assign in the declaration" rule replaces it.

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, but there's a work-around. `C#` has a special rule to make empty, completely abstract classes with abstract virtual "=0" functions. It calls those interfaces, and you can inherit from any number of them (and also from 1 real class.)

To compare, a C++ abstract "interface" class, then redone as a formal `C#` interface:

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

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

Notice how we leave off most of the verbiage in the `C#` version. All `interface` functions are automatically public, virtual, and must-implement. You're not allowed to write function bodies and can't declare any variables. `C#` interfaces act as a pure contract, and nothing else.

Interfaces work like real classes in every way. Multiple inheriting looks the same, as does using them for variable types or parameters:

```
class CatEmployee : Cat, Worker { ... }
Worker w1 = new CatEmployee();
int doubleShift(Worker w) { return w.doWork() + w.doWork(); }
```

`C#` style muddies this. Interfaces are commonly marked with a capital-I in front. That gives the impression you need to know, but you don't. An interface works the same as any other abstract base class.

Since you only inherit from one real class, the syntax to jump into your base class is simplified. Instead of using the name, usethe `base` keyword. `animal::cost()` translates into `C#` as `base.cost()`.

Virtual functions require one extra step. As usual, write `virtual` in front in the base class. But you also need to write `override` in front of the function in all children::

```
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(); { ... } // this is legal. cat.sleep is NOT virtual.
}

class Dog : Animal {
  public new eat() {...} // dog.eat is NOT virtual (new is the keyword)
  public override sleep() {...} // as opposed to cats, dog.sleep is virtual
}
```

Some background: as you may know, all functions in Java are virtual. There isn't even a keyword since there's no alternative. You might expect `C#` to split the middle, maybe by having virtual as the default, making you work to turn it off. But as we see, non-virtual is still the default.

The new oddball rule is about increased granularity. Each sub-class is allowed to decide which of its functions should be virtual. So conceptually `virtual` in a base class means some subclass functions *may* choose virtual. Write `override` to make it virtual, `new` to say it isn't. But the default is non-virtual so no-one ever writes `new`.

Notes: 1) yes, this is re-using `new` for a completely different purpose. 2) This is not a useful feature - just write `override` everywhere. The official reason is cat.eat may have started as a normal function, not overriding anything. Then Animal later acquires a virtual eat function with a different meaning. This rule helps flag how the cat.eat was never meant to be a proper override.

There are no `friend` functions. Reflection allows anyone to examine and use anyone else's private stuff – you'll see some `friend`-like functions written using that.

Extension methods: this is pure sugar: you can write 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:

```
public static passTime(this Cat c, int days) {
  for(int i=0; i<days; i++) { c.eat(); c.sleep(); }
}

// use:
c1.passTime(7); // compiles to: passTime(c1, 7);
```

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 a second use of the old `this` keyword.

Properties: this last new class feature has lots written about it, lots of extensions and is in very common use. But it does nothing special - you can fully use `C#` without ever writing one of these.

Instead of writing simple getter/setter function pairs, you can create a fake variable. You write a zero-input get function which is called when you read from it, and a 1-input set function called whenever code writes to it. A typical example, this creates a fake yearsOld variable:

```
class Cat {
  public int monthsOld;

  public int yearsOld { // yearsOld is our fake variable
    get { return monthsOld/12; }
    set { if(value<0) value=0; monthsOld=value*12; } // value is a keyword
  }
```

Clients use `if(c1.yearsOld<3)`. That runs `get` as a function. Then `c1.yearsOld=3;` sets `value` to 3 and runs `set` as a function. Notice how get and set share a type – the `int` in front of the property name. It's the return type of get and the input type of set.

One common use is to avoid parens. Write a `get` but no `set`. You can think of it as a read-only variable, or as a paren-less function:

```
class Cat {
  public int monthsOld;
  public bool isAdult {  get { return monthsOld>14; } }
```

We can now call `if(c1.isAdult)`. In practice short things like this alternate between being written as properties or functions. There's no advantage to either.

They are most often used as a proper setter/getter pair hiding a private variable. Conceptually we had `public int age;`, but then rewrote it:

```
class Cat {
  private int _age;
  public int age {
```

```
    get { return _age; }
    set { if(value<0) value=0; _age=value; }
  }
}
```

A common problem is how these hide functions. When you use a "variable", check if the drop-down lists it as a property and shows is as `get; set`. That means you're really calling a function. Sometimes a long, complex function with side-effects.

There's a common assign-through mistake: a `get` returning a struct will return a *copy*; attempting to assign to a field does nothing. In this example `c1.name.first="spike";` looks fine, but does nothing:

```
// a struct used by Cat:
struct name_t { public string first, last; }

class Cat {
  private name_t nm;

  public name_t name {
    get { return _name; }
    set { if(value.last=="") value.last=" the cat";  _name=value; }
  }
}
```

If you think of it as what it really is: `c1.getName().first="spike";` the problem should be obvious.

One nice advantage of properties over getters/setters is the ability to seamlessly convert public variable use. Suppose your client code used a public `age` variable: `if(c1.age<7) c1.age=7;`. In C++ we'd need to rewrite that as `if(c1.age()<7) c1.setAge(7);`. In `C#` we can convert `age` into a property (like above.) The client code is unchanged.

Despite being OOP in theory, `C#`-style says it's fine to see and use fields directly. Except you need to launder them through do-nothing properties. You have no intention of ever adding checks later. This is just the style. This sort of thing, a class with effectively 2 public vars, is common:

```
class Puma {
  private float cl;
  float clawLen { get { return cl; } set { cl=value; }}

  private string bfd;
  string bestFood { get { return bfd;} set { bfd=value; }}
}
```

There is one obscure reason for this. Some(?) old(?) linkers have a bug where they won't notice converting a public var into a property. They won't recompile client code, instead giving garbage results. So that's why the "public vars must be trivial properties" style caught on.

Because of how much time it takes coders to type out do-nothing properties for every public variable, there's a shortcut, named auto-properties. Write `public int age { get; set; }` and `age` is a property backed by a secret private war (wait? It's secret? Shouldn't I be using it directly from within the class? Since the property does nothing, you may as well use `age` everywhere.) This is no help writing a real property and acts like a public variable in every way. The only purpose is to use that style.

To sum up, you never need to write properties. But everyone is nuts for them so you'll need to interact with properties. And you'll see so much extra about them it seems they must be important, but they aren't.

2 Additional features

- 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. Commonly a `C#` GUI uses it to provide easy serialization.)

- Garbage collection. It runs automatically at no particular time. There's no `delete` command. Most `C#` programmers don't even `null` variables – they'll go out-of-scope soon enough.

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

  Because of this, you're allowed to create new classes on the fly. Someone else can use reflection to figure out the contents.

  You'll also sometimes see set-ups where code impossibly uses private variables and function – they're cheating with reflection.

- LINQ library. This is a few things put together. It's another thing people get far too excited about. It has some functions for directly querying a database. It also has an SQL-like syntax which you should never use (it's for data-base people who don't like functions.) An example from the microsoft site:
  ```
  var q = from c in customers where c.City == "Ral" select cust;
  ```

  The last thing are common list-searching functions – things like countIf, return all items matching this predicate and so on. None of them are allowed to modify the list.

3 Reference Types

This is the major change from C++, borrowed from Java. It's a system intended to make pointers simpler and allow automatic garbage collection.

3.1 Mechanics

Basic types and structs can't be allocated on the heap, and can't be pointers or be pointed-to. Class and array variable can *only* be pointers and *must* be allocated on the heap.

Whenever you declare a class or array variable, you're really declaring a pointer, currently null. You never explicitly dereference a pointer – you write just `c1.name` and it really means `(*c1).name`.

If `Cat` is a class, the simplest way to create and use one looks like this:

```
Cat c1 = new Cat(); // c1 is a pointer to a heap Cat
c1.name="fritz"; // automatic dereferencing c1
```

Note that automatic garbage collection will clean this up when `c1` goes out of scope. Wanton use of `new` is no big deal.

Some observations about this system:

- Structs and classes aren't the same anymore.
- If you'll ever want a pointer to a type, which you probably will, it needs to be a `class` instead of struct. This makes class the preferred type.
- Pointers can never be aimed at stack data. It's just not possible with this set-up.
- You never have a choice to create a pointer. Your type determines it for you. There's never a `Cat c;` vs. `Cat *c;` choice.
- There's no address-of operator. You don't need it for reference types, and can't use it on normal types. There's no dereference operator, since it's always automatic.
- Structs can't have virtual functions (since that requires pointing to one, which can never happen.)
- Since the only way to create a class variable is pointer and new, it's hard to know when we *really* wanted to create a pointer (more on the later.)

In `C#` terms, these implicit pointers are **references** (which goes along with Reference Type – clever, right?) But they're pointers in every way. Some examples:

References use `null` and test for `null` as usual:

```
Cat c1=null;
c1.name="Boots"; // run-time err: null reference exception

if(c1!=null) c1.name="Boots";
```

`==` with class variables is pointer comparison. If we have two Cats, and they're identical, `==` tells us they aren't the same cat:

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

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
}
```

Because functions can be used to change class fields, you'll sometimes read incorrectly that classes are passed-by-reference. Of course they're not – it's just the standard "pointers are almost as good as call-by-ref" trick. You can pass classes by reference: `(ref Cat c1)`, but, like all pointers, you rarely need to.

Returning a class is returning a pointer. Here `oldestCat` takes two pointers and returns one. There's nothing special about this except it looks weird having and using so many "these are really pointer" vars:

```
Cat oldestCat(Cat c1, Cat c2) {
  if(c1.age>c2.age) return c1; else return c2;
}

Cat c1=new Cat(), c2=new Cat() ; // assume these are given values
Cat c3=oldestCat(c1, c2);
// c3 is pointing to either c1 or c2
```

This example shows how mentally class variables are divided into "regular" and "pointers". `c3` is meant to be a pointer - it's purpose is to move around, pointing to someone else's cat. But `c1` and `c2` are meant to be just regular cats. In `C++`, seeing a pointer and `new` means we specifically did extra work to create persistent data. In `C#` it doesn't mean anything – it's the only choice.

The same rules apply when structs/classes are member variables. In this, struct `Dog` is allocated with us, but `Cat` is again merely a pointer:

```
class CutePetPair {
  Dog d; // Dog struct is contained in the class
  Cat c; // merely a pointer
```

```
  void init() {
    d.name="Spot"; // safe, we contain the storage for d
    c=new Cat();  // required to allocate the space
    c.name="puss-puss";
  }
}
```

Then one more example of the "regular" vs. "pointer" division, here's a crude linked-list. It looks like an illegal recursive data structure, until you remember the nested `catList` is a pointer.

```
class catList {
  public Cat c; // our cat, not considered a pointer
  public catList next; // pointer to next item

  public catList() { c=new Cat(); next=null; }

  // a standard insertAfter:
  public insertAfter(string catName) {
    catList newItem=new catList();
    newItem.c.name=catName;
    newItem.next=next; // the usual pointer insert..
    next=newItem; // ...which looks weird with no stars
  }
}

catList CL=new catList();
```

Back to something I mentioned before, structs and class intentionally look the same. You're encouraged to always initialize structs from the default constructor, which has that do-nothing `new`; and the automatic dereference makes member use look the same:

```
someStruct a = new someStruct(); // stack variable, assigned from constructor
someClass b = new someClass(); // class variable, allocated on heap

a.code="xy"; // regular member variable
b.ident="zq"; // implicit pointer dereference
```

The place that similarity can bite you is `f1=f2;`. Most simple data-only structures are written as structs, which makes that a standard member-wise copy. But you may not be sure of the type, can't easily tell from context, and it's really a class, using a pointer copy.

Interestingly, classes don't have a member-wise copy. There's no way to write `(*f1)=(*f2)`, they would have had to make up some special syntax, and it just seemed easier to leave it out and have you write a Copy function.

Arrays

Like classes, array variables are always non-const pointers to something on the heap. In C# terms, they're also Reference Types. There's nothing like `int A[10]`, and no array names which are const-pointers. Brief C# array use:

```
double[] A =  new double[20]; // simplest and only way to create an array
double[] B = A; // they work like real pointers, sharing an array
if(A==B) {} // pointer compare
A = null; // they can be null
```

Then, to repeat, arrays are ranged-checked and have a `Length`. You never have to pass an array and the size.

This is nothing new, but suppose we want a simple array of 10 cats: `Cat C[10];` in C++. Creating that in C# looks like this:

```
Cat[] C = new Cat[10];
// currently an array of 10 null pointers
for(int i=0; i<A.Length; i++) // using the Length function (which is a get, so no ()'s)
  A[i]=new Cat();

A[i].name="fluffy"; // we still have the auto-dereference
```

To repeat an earlier note: somewhere else in the code we needed a pointer to a cat. That meant cat needed to be a class. Which means all Cats need to use pointers and new.

Then one more example of "real pointer" thinking, which works in C#. Mentally each item in tracked cats will point to someone else's cat, or be null:

```
Cat[] TrackedCats = new Cat[5];
TrackedCats[1]=c1; // assign a few to any other cats
TrackedCats[4]=C[2];

for(int i=0; i<TrackedCats.Length; i++)
  if(TrackedCats[i]!=null) print(TrackedCats[i].name);
```