C# for C++ programmers

A short history of C#: it's best understood as an offshoot of Java, tacking back towards C++. Java starts with C++ syntax, cuts error-prone or complex features, enforces and encourages a stronger object-oriented feel, and simplifies early pointers using a trick called Reference Types.

As a Java offshoot, C# adds back some features found in C++, but with slightly different syntax and functionality. It's verbose, has more sugar, is more aggressive with error messages and follows "release early and often" for new versions.

As a language aimed more at beginners, C# follows "use the same symbols if it feels the same". The idea is that having slightly different syntax for slightly different concepts is better *if you understand the concepts*. But if students tend to blur two ideas together, you may as well go with the flow and blur them in the language.

Put another way, C# has some insane design choices.

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

The confusing part of reference types is the 2-stage learning process, and the fact that we can skip stage one. As you'd expect, understanding pointers and heap vs. stack is all you need. That's considered advanced C#. Early to mid C# is a set of rules which are basically pointer heuristics. The trick is to understand those aren't some new concept – they're just an intermediate step.

Part I of this 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.

1 Small changes from C++, Pt I

1.1 Misc, Pt I

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;`.
- The dot is also the scope resolution operator. For example, our `c1.age = Cat::maxAge`, is written in `C#` as `c1.age = Cat.maxAge`. As an even more horrifying example, our `Cat::primeCat.age` becomes `Cat.primeCat.age` in `C#`.

  You can't use double-colon for normal scope resolution, but `C#` repurposes it: `global::` can be used to begin a path, as in `global::Cat.maxAge`. It's called the namespace alias qualifier, if you want to look it up.
- There's no official main entry point: no `int main()`. Like Java, the main entry is some function in some class, which the environment either somehow knows or you register somehow.
- There are no general `typedef`'s. You can't alias `age_t` to mean unsigned int. But there's a special typedef only for function pointers (later.)
- No macros. No `#define sqr(x) (x*x)`.
- Namespaces can't hold variables or functions – only classes and enums. Instead, dummy classes are often repurposed as namespaces. For example here a `C#` programmer would recognize `stringFuncs` as a dummy "namespace" class:

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

  A reminder, these are used with a dot: `stringFuncs.bestWord="cow";`.

  The vanilla `using` directive adds namespaces only: `using System;`. To add a class-as-namespace you need `using static stringFuncs;` (this is a recent addition.) `using`'s can't be inside of anything and and can't pick out individual functions – no `using stringFuncs.compareSomehow;`.

  To confuse you, `using` is also repurposed. Surrounding code in a `using`-block will automatically and immediately run your destructor when exited

(see the reference type section for why this is needed.) While a 3rd version of `using =` creates a namespace alias.

- There are no persistent local variables (the ones with the `static` keyword before a local variable.) (`static` keeps it's main meaning, marking things as class-scope globals.)

- `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.

- 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.

- Array changes:

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

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

    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, also has 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 can't add a reference or dereference:

  ```
  ff = charsInCommon; // an imaginary function
  x=ff("cow","wookie");
  ```

  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);
  ```

No inputs or outputs requires leaving off the angle-brackets: `System.Action f3;` for a void(void) function.

There's an older syntax, mostly replaced by the above, where you were required to first create a typedef, using the keyword `delegate`. The syntax requires dummy names for the parameters (which never need to match any actual names):

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

Then it works like normal: `myStrCmpFT ff;`. This and the newer `System.Func` method are fully interchangeable.

`C#` function pointers are called **delegate**'s, because of this older syntax.

- 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 was defined as an (int*)(int,int) function pointer
```

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. `C#` has something, **generics**, that looks like C++ templates but is very, very limited. The function isn't allowed to assume the type has anything except `=`. This is obviously a pretty big limitation.

  You can specify the type must be a class (any class), which gives the function access to `==`. You could use that to check a list for duplicates. Finally you can limit the input to a subtype of one particular class. That can be used to search a list of the superclass for that exact subtype.

  You can't do any of the bread&butter stuff 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 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 something into an error for possibly being bad style:

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

- There's no direct use of `C++`-style reference variables (such as `int& n = C[i].age;`). But it has return-by-reference functions assigning into true reference variables:

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

ref int n = ref getAgeRef(c1); // same as: int& n=&c1.age;
```

Notice all of the redundant `ref`'s. They're required. The referencee (age) must be on the heap.

- There are no header files or prototype functions. But there's also no need. C# environments don't need things pre-defined – they'll rummage around to find stuff.

  I still miss looking for functions using the relatively clean dot-h.

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

- 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.

  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 functions – they're cheating with reflection.

- LINQ library. This is a few things put together. 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, and they're really objects with delayed evaluation.

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.

A class's default constructor initializes everything. It also goes away 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
  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, replaced by this. There's no `public spot():  x(99)`.

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. But you can inherit from one class and 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
}
```

Notice how it leaves off most of the verbiage. You can only require functions – no variables, and no function bodies. These count as types and can be used normally (as pointers.)

`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.

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 they can't have things to use.)

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 subclasses:

```
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: 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(); }
}
```

`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.

Properties: these are shortcuts for writing getter/setter pairs. Ex using them to create a fake years variable:

```
class Cat {
  public int monthsOld;

  public int years { // years is our fake variable
    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.

As style, `C#` coders consider it proper OOP to use trivial getters/setters. A simple `public int age;` is always converted to this:

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

This is so popular that "auto-properties" were added to do it: `public int age { get; set; }` creates an inaccessible backing int and the (hidden) code above.

The official reason for this style is that public vars can be transparently converted into properties, except for a possible linker bug. The linker may not notice the change, and not recompile client files, giving lookups junk results.

You'll commonly see 0-input member functions rewritten using only a `get`, turning `c1.isKitten()` into `c1.isKitten`. `C#` programmers love the heck out of properties.

There's a common assign-through mistake. Suppose we have `c.name.first="x";`, and `name` is a struct returned by a `get`. That means `c.name` is only a copy, which means we're uselessly assigning to a copy.

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 ones that can't use pointers, and others that must use pointers. 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 consists of classes, and is two rules in one. The first is you can only declare pointers to classes, and they can only point to heap objects. The second is we won't use new syntax. We'll repurpose the old syntax as much as possible.

For example, suppose `Cat` is a class. We can use it like this:

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

`Cat c1;` has been repurposed to declare a *pointer* to a Cat. And `c1.name` implicitly dereferences `c1`. The middle line is exactly what it looks like – a standard `new` returning a pointer to what it created.

It goes together so nicely. We don't need to learn a new syntax to use pointers. So far we don't even need to know we're using them. And we don't need to give an error when you attempt to declare a normal Cat, since there isn't even any way to do that.

This second type, and this system, is called Reference Types. Confusingly, the variables are called References. Some observations about this system:

- Pointers can never be aimed at stack data. It's just not possible with this set-up. Only normal data can be on the stack, which can't be pointed to.

- 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.
- 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.
- You can't have the equivalent of `Cat**`. You fake it by making a small class holding the second pointer: `(**cpp).age` becomes `cpp.c.age`.

One more simplification is there's no `destroy` command. Automatic garbage collection makes that possible. But the real reason is we don't want users thinking about it. Declared ints go away on their own, and so do Cats.

Because of how garbage collection works, there really isn't a destroy command. A GC step scans all stack variables, marks anything pointed to, scans all pointers inside of those, and so on, then destroys anything untouched and repacks the rest (I think. It probably depends.) It doesn't work outside of that whole process. It runs "occasionally."

Other than the `c.name` shortcut, these are pointers in every way. They use `null`, `=` is pointer assignment, and `==` performs pointer compare:

```
Cat c1=null;
c1.name="Boots"; // run-time err: 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 pointing to the same thing
```

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
}
```

Fun facts: the system doesn't provide classes with a member-wise compare or member-wise copy.

A few more examples of things that look funny. By these rules returning a class is returning a pointer. This returns a pointer to one of the cats the caller gave use:

```
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. This linked-list node has the standard `next` pointer:

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

Back to a previous point, `Cat c;` is also declaring a pointer, but we're not sure if that was intended. In `C++` we'd choose between `List<Cat>` and `List<Cat*>`. If we see the the former, we know it's a simple list.

In `C#` we have to use `LinkedList<Cat>` for both – maybe we wanted Cat-pointers, but we probably wanted simple Cats. To know we have to look inside: `c=new Cat();` in a constructor means simple Cats.

That's one of the tricks to reading `C#` code. In this, the coder is probably thinking of `c1` as an actual variable, and `c2` as a reference to `c1`:

```
Cat c1=new Cat();
Cat c2=c1;
```

We know (and this is how it works) that `c1` and `c2` are two pointers to a heap Cat. `c1` is special only since we're considering it as the owner. But `C#` coders tend to think of `c1` and `c2` as actually being different types of variables.

2.2 Arrays as Ref Types

`C#` Arrays are Reference Types, because they are. A `C#` array variable is a non-const pointer to the actual array on the heap. The only `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; // standard pointer use
if(A==B) {} // pointer compare
A = null; // they can be null
```

There's no array pointer math. There's not any syntax for something like `*(A+1)`.

Being a reference type gives the same confusion about intention. Where we'd use `int A[5];` for a simple array and `int *A;` for a pointer to one; `C#` uses `int[] A;` for both. You have to look inside for either `A=new int[5];` or `A=someExternalArray;`.

2.3 Structs and Ref Types

When you create a class, you actually have the option to have it be a Reference Type, or not be one. `struct` means normal, `class` means reference type.

Some things that follow from this:

- `C#` structs and classes are fundamentally different.
- You can never have a pointer to a struct.
- Structs can't have virtual functions (since no pointers.)
- Declare and use them like normal: `Dog d; d.age=4;`, if `Dog` is a struct.
- Classes are vastly preferred over structs. If you will ever need even one pointer to one, you have to use `class` to make your object a reference type.
- `d1=d2;` for structs performs a standard member-wise copy.
- Misc class rules are changed, for speed (which is the only reason to use them.) Structs don't have the `int n=5;` auto-assign shortcut, since you aren't required to call the constructor.)

Structs are considered an advanced feature. You learn classes first, since they're the most useful and it's the way Java does it. Then you learn about a rarely used non-ref-type struct, which has funny semantics (compared to a class.)

This leads to a funny rule. For classes `C#` coders at first reflexively learn `Cat c=new Cat();`. For a while, the `new` is just there. To use a struct constructor, the language keeps the `new`, but it does nothing.

This works out, sort of:

```
someClass a1 = new someClass(); // actual heap allocation
someStruct b2 = new someStruct(); // simple constructor call
// second isn't required, but nice to be sure things are initted
```

This is the "things that seem similar should have the same syntax" concept. We think `new` means heap allocation, and the second line is a deliberate attempt to confuse us.

`C#` designers think both lines are setting up the variable. Put another way, coders would see it as a special rule "structs don't use `new`."

3 Misc features, Pt II

- 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) {...} // both are pass-by-ref

  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.

  `C#` coders tend to have trouble with this because of reference types. They learn things like this pretty quickly:

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

  This handles about 90% of the cases – classes are the preferred data type. And since `c` is considered a reference, things tend to blur into an informal rule that classes are already passed by reference. They have a vague feeling that ints purposely don't work this way.

  To be clear, passing a class is passing a pointer by value, and that's the best way to understand it.