C# RUNNING START

A quick ‘n dirty introduction to the C# language for the C++ programmer.

Disclaimer

- This document is a simple introduction to some of the things in C# programs that might be familiar to C++ programmers.
- It is hardly comprehensive.
- However it does mention a few things in C# that aren't in all versions of C++, just to give you a peek at a few additional features C# has.
- Finally, this is ***NOT*** a tutorial.

Data Types Mostly as Expected

- *short*, *int*, *long*. Also *byte* (which is unsigned 8-bit)
  - Always 16-, 32- and 64-bits respectively
- No *unsigned* keyword
  - *ushort*, *uint*, *ulong*. Also *sbyte* (signed 8-bit)
- *char*
  - *Always* a 2-byte Unicode character.
  - Use *byte* for true unsigned 8-bit.
- *string* is built-in.
- *float*, *double*
- Also *decimal*
  - 128 bits, 28 digit accuracy
  - Mostly for financial applications. Never used it.
- *enum*

Semicolons

- Same basic rules as C++
- Semicolons go at the end of statements and inside *if* statements
- Some statements (e.g. *if* statements with braces) don't need them since nothing can follow the closing brace (except the *else* clause, which has the same rule), so there's no need for a semicolon

Braces I

- Same basic rules as C++
- Braces are mostly used in 4 places

1. After a *class* declaration (e.g. class foo { … }
2. After a method declaration (e.g. void fu(int x) { … }
3. Inside a property: e.g.
   - int _Age;
   - int Age {
     - get { return _Age; }
     - set { _Age = value; }
   - }

Braces II

4. To group multiple statements into a *block* that acts like a single statement.
   - if (n == 1) {
     - DoThis();
     - DoThat();
   - }

Operators
All the Standard Ones

- +, -, *, /, ++x, x++
- <, <=, ==, !=, >, >=
- Array access: x[i] = foo; foo = x[i];
- &, |, ^, &&, ||
- x ? y : z
- C-style casting: (int)x
- Oh yeah, assignment: x = y ☺
- A few others

Statements
The Usual Suspects

- *if / then / else*
- *while*
- *do { … } while (…)*
- *for*
  - Including *break* and *continue*
- *switch*
  - Note: All *case* clauses, including *default*, must end with a *break*
  - In C# 8, there's also a *switch* operator.
- *return*
- *try / catch*

Foreach Statement

- *foreach (int i in MyIntArray) {*
  - *ProcessIt(i)*
- *}*
- Not limited to just arrays. Any data type with the concept of a first element, and the concept of a successor element can be processed by *foreach*.
  - Most of the .NET Framework collection classes (List, Dictionary, etc) have these.
  - It's hardly limited to simple array-like data structures.
- This is roughly comparable to a C++ iterator.
- I use *foreach* far more often than I use *for*.

The *var* Keyword I

- If we were to write *string name = "Joe";* this would be a bit redundant, since we're initializing *name* to a *string*, so why do we need the *string* datatype mentioned??
- And we don't. We could write *var name = "Joe";* and C# would *infer* the data type of *name* from the initialization clause.
- While the use of *var* is hardly needed/wanted in such a simple case, consider a *Dictionary* (a type of hash table) where we want the key to be a string, and the associated value to be an instance of a Person class.

The *var* Keyword II

- Rather than writing
  - Dictionary<string, Person> AddressBook = new Dictionary<string, Person>();
- We could write
  - var AddressBook = new Dictionary<string, Person>();
- Which to use is mostly a matter of taste, although there are advanced cases where *var* is necessary.

Classes and Methods Same Old, Same Old

- A C++ class and methods will often work unchanged in C#.
- Except that pseudo labels (e.g. *private:*, *protected:*, etc) aren't supported. Put the info on the field or method definition.
  - Not *private: int foo; void goo() {…}*
  - Instead *private int foo; private void goo() {…}*
- Prototypes don't exist. Classes and methods can be in any order.
  - All C# compilers *must be* multi-pass.

Storage Classes

- *Reference types* are class instances.
- *Value types* are all the rest, things like int, float, char, struct, etc.
- All class instances are implicitly references.
  - Not MyClass foo& = new MyClass()
  - Instead MyClass foo = new MyClass()
- Value types are kept on the stack, as in C/C++.
- Reference types are always on the heap.
  - MyClass foo[10]; // WRONG!
    - Reserves space on the stack for 10 references. Does not instantiate 10 objects.
  - MyClass[] foo = new MyClass[10]; // RIGHT
  - foo[0] = new MyClass(1); foo[1] = new MyClass(2); // etc

Generics (what C++ calls Templates)

- *class foo {*
  - *public min(int x, int y) { if (x < y) return x; else return y; }*
  - *public min(float x, float y) { if (x < y) return x; else return y; }*
- *}*
- Other than the data types, the source code for the two methods are identical.
- *class foo<T> {*
  - *public min(T x, T y) { if (x < y) return x; else return y; }*
- *}*
- This creates a family of methods, one for each type you specify. For example foo<int> is a different class from foo<float>
- Consider, class List<T> with methods, say, Add, Insert, etc)
- List<int> MyList = new List<int>(); // Allows only int's
- List<Person> MyList2 = new List<Person>(); // Only Persons

Missing Stuff Part I

- No *.h files. Metadata is kept in .exe / .dll (like in Java)
- No pointers. At all.
  - OK, I lied. You can have them inside *unsafe {…}* blocks, as long as you also have the compiler mark the .exe / .dll files as unsafe.
  - I've used pointers in C# exactly once (in 10 years), and that was when I was playing around. Otherwise I've never used them (or missed them) in any of my programs. References work just fine.

Missing Stuff Part II

- *new* for arrays works differently.
- *MyClass[] foo = new MyClass[10]* does *not* call the constructor for each instance.
- *foo[]* would contain 10 references to *null* (a C# keyword).
- You'd then have to loop (or whatever) to set them
  - *for (int i=0; i<10; i++) foo[i] = new MyClass(i);*
- C# is garbage collected, so there's no *delete* statement.
- Destructors (*~foo()*) exist, but work differently. More about that some other time.

Intermediate Stuff not in C++ Properties

- class Person {
  - private string _Name;
  - public string Name {
    - get { return _Name; }
    - set { _Name = Name; }
- }
- Person me = new Person();
- me.Name = “Daryl”;
- string BrotherName = me.Name;
- The compiler treats these as functions called get_Name() and set_Name(). So these are really function calls in disguise and you can do whatever you want inside get/set.

Intermediate Stuff not in C++ Indexers

- class Book { public string Title, ISBN; }
- class Books {
  - private Book[] _books; // Assume set somewhere
  - public Book this[string isbn] {
    - foreach (var book in _books) {
      - if (book.ISBN == isbn) return book;
    - }
  - return null;
  - }
- }
- In other words, this lets you use any data type as an index, not just integers

Advanced Stuff not in C++ LINQ (Language Integrated Query)

- A SQL-like feature built into the language.
- var BigCpuUsers = from proc in System.Diagnostics.Process.GetProcesses()
  where proc.TotalProcessorTime > new TimeSpan(0, 5, 0) /* 0 hours, 5 minutes) */
  select proc;

foreach (Process p in BigCpuUsers) p.Kill(); // Buh-bye!

Advanced Stuff not in C++ Async/Await

- When you read from a file, by default it's synchronous. You issue ReadFile() and it doesn't come back until the read is done.
- But sometimes you don't want this. For example, you don't always want your program to hang while you download a gigabyte file!
- Traditionally you'd need to say "start the read and when you're done, here's a subroutine to call". Meanwhile, proceed as normal (e.g. responding to mouse clicks, starting another parallel download, etc.)
- It can get messy juggling the code involved.
- C# has the *async* and *await* keywords to vastly simplify this.

Console.WriteLine (or, as C would call it, printf)

- Syntax: Console.WriteLine(string FormatString, values-t0-be-formatted)
- Uses *{n}* instead of %d, %s, etc
- Console.WriteLine("Item {0}: Name {1}", 5, "Widgets");
- Puts a Newline (\r\n in Windows, \n on Mac/Linux) at the end. Use Console.Write() if you don't want this.
- Used for console apps, but even in GUI apps, shows up in the debugger.
- Also, string msg = string.Format("Item {0}: Name {1}", 5, "Widgets");
  - Result: "Item 5: Name Widgets"
- Recently: $"Name is {MyName}; I'm {Age}"