**In C#, everything is an object** and **everything is a member of a class**. In other words, the only top-level objects in a C# program are classes.

## Data Types

C# divides all data types into two types: **value types** and **reference types**.

* A **value type** actually contains its value. All the predefined data types in C# are value types, except for *object* and *string*. If you define your own struct or enum, these will also be value types. This means that the simple data types in C# generally work in exactly the same way as in C++ when you assign values to them.
* A **reference type** contains only a reference to where the data is kept in memory. *Classes* are reference type, and that’s why object and string are references, in addition to array. Syntactically, this works the same way as references in C++, but in terms of what is actually happening, C# references are closer to C++ pointers. C# references can be reassigned to point to different data items, in much the same way that C++ pointers can. Also, C# references can be assigned the value *null* to indicate that they don’t refer to anything. Example:

MyClass M1 = new MyClass(); // In C#, new simply calls a constructor.

M1.Width = 20;

MyClass M2 = M1; // M2 now points to the same memory location as My1.

M2.Width = 30; // Now M1.Width = 30 too because M1 and M2 point to the same location.

M2 = null; // Now M2 doesn’t refer to anything. M1 still refers to the same object.

**Note**: There is actually another data type in C#: **pointer types**. But they’re considered as *‘unsafe’* and rarely used.

## Garbage Collector

In C#, we rarely need to care about dynamical deallocation of memory, like in C++. That’s because the **.NET garbage collector** periodically comes in and scans through the references in your code in order to identify which areas of the heap are currently in use by your program. It is then automatically able to remove all the objects that are no longer in use. This technique effectively saves you from having to free up any memory yourself on the heap.

|  |  |  |
| --- | --- | --- |
|  | **C#** | **C++** |
| Header files | - No header files. No #include | - Yes |
| Main function | Main().  - Defined inside a class | main()  - Defined outside of any class |
| Class | - Only way to access static members of classes: <ClassName>.<MemberName>  - Do not require a ; after a class definition  - No *access modifier* on the name of the base class. Inheritance is always public.  - A class can only be derived from one base class  - Each member is declared with an access modifier.  - Implementation of methods is always placed with the definition. | - Two ways:  + <ClassName>::<MemberName>  + <InstanceName>.<MemberName>  - Must have  - Must have    - Allows multiple inheritance  - Each access modifier is defined once at the class name and applied to all members of this class.  - Can write the implementation outside the class. |
| Static constructors | - C# allows the concept of a static constructor, which is executed once only, and can be used to initialize static variables. | - No |
| Destructors | - Less need for destructors  P31 | - Very essential |
| Method definition and declaration | - Methods are always defined fully in the class definition. | - Can separate method definition and declaration |
| Switch …case | - Can use a string as the test variable | - No. Only numbers |
| Variables | - Only be declared locally in a method or as members of a class. No global or static (that is scoped to a file) variables.  - Member variables (not local variables to methods) are by default initialized by being zeroed out if you do not explicitly initialize them. Numeric value is 0, bool is false, reference is null, structs have each of their members zeroed out. | - Yes  - Random data |
| Data types | - uint  - ulong  - object  - decimal  - var | - Unsigned int  - Unsigned long  - No  - No  - No |
| String | - Characters are always Unicode characters  - Strings are a defined reference type, not an array of characters. | - ANSI characters  - Array of characters |
| Passed by reference | public void MultiplyByTwo(***ref*** *double d,* ***out*** *double square*)   * ref parameter is initialized before being passed by reference to a method * out parameter is initialized within the called method before being used. | void MultiplyByTwo*(@double d, @double square*) |
| Default parameters  to methods | double DoSomething(*int someData*)  {  DoSomething(someData, true);  }  double DoSomething(int someData, bool condition)  {  // etc. | double DoSomething(*int someData, bool Condition* ***= true***)  {  // etc. |
| this | - Called this reference  this.MyField = 10; | - Called this pointer  this->m\_MyField = 10; |
| Virtual | - Use the virtual keyword in the base class and the override keyword in derived class. | - Only use the virtual keyword in the base class. |
| Abstract | public **abstract** void DoSomething(*int X)*; | public:  virtual void DoSomething(*int X*); |
| Constants | - Only apply to variables and references  - P36, 37 | - Can apply to variables, references, pointers, parameters of functions, member functions. |
| Arrays | - Array is an instance of the base class System.Array and a full-blown object stored on the heap under the control of the garbage collector.  - double [] array;  - double [] array = new double[10];  - int [,] array2d = new int[2,3] { {1, 0}, {3, 6}, {9, 12} }; | - Array is a set of variables packed together in memory and accessed via a pointer.  - double array[10];  - double \*pArray = new double[10];  - int myArray2d[2][3] = new int[2,3] { {1, 0}, {3, 6}, {9, 12} }; |
| Properties  Indexers | - Yes | - No |
| Namespace | using System; | using namespace std; |
| #define | No (member variables instead) | Yes |
| enum | public enum DEF\_NAME {  ALEX,  JOHN  }  When calling: DEF\_NAME.ALEX | enum DEF\_NAME {  ALEX,  JOHN  };  When calling: ALEX |