UNSW SYDNEY

Core Concepts of Common Language Runtime (CLR) in C++

A/Prof Shiyang Tang

MTRN3500

Computing Applications in Mechatronics Systems

Basic Concept of CLR

The **Common Language Runtime (CLR)** is the virtual machine that runs and manages applications written in **.NET** languages like C#, F#, and **C++/CLI** (Common Language Infrastructure).

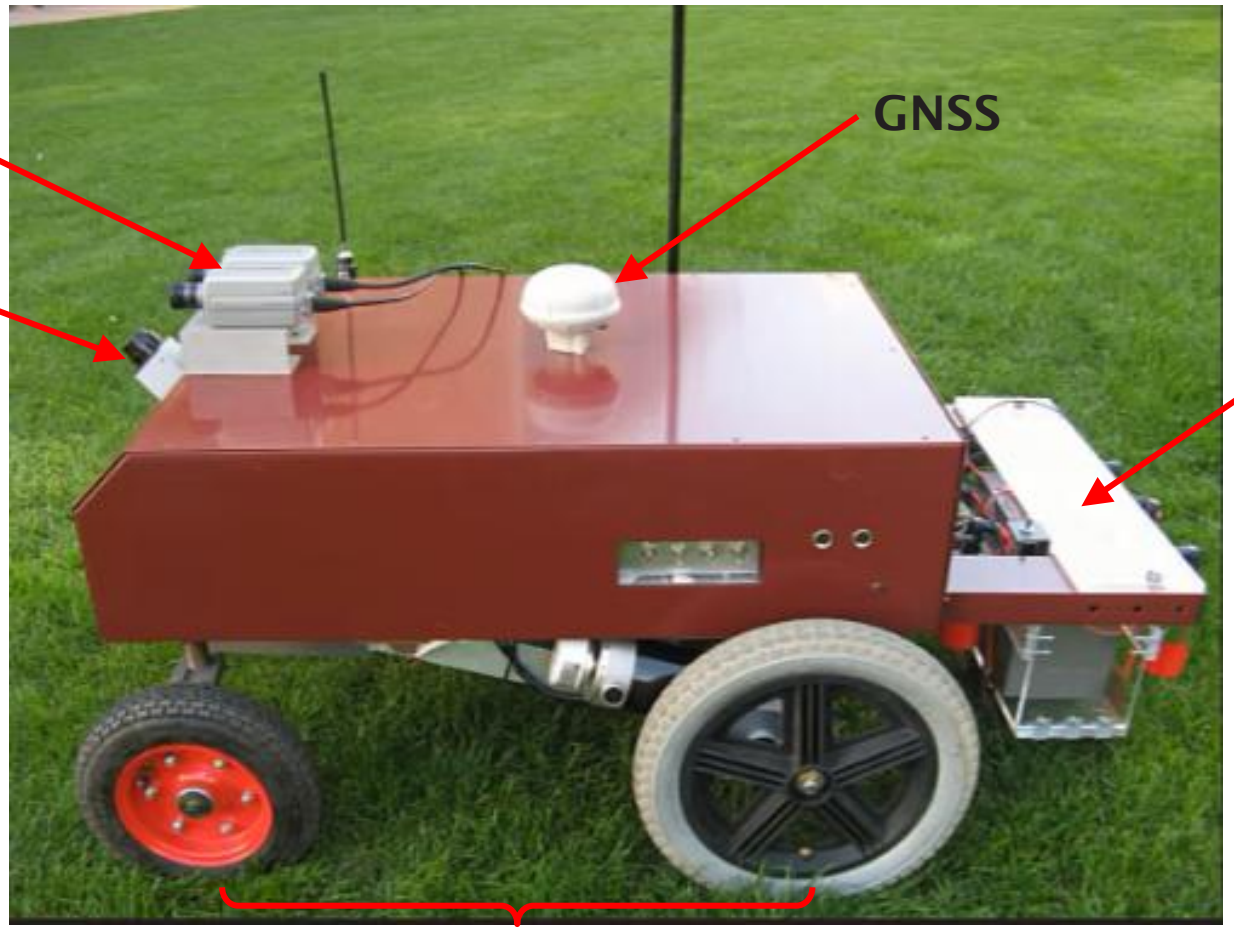
.NET is a free, open-source, cross-platform developer platform created by Microsoft for building a wide range of applications (e.g., web, desktop, mobile, cloud, gaming, IoT, and AI) using multiple programming languages like C#, F#, Visual Basic, and C++/CLI.

How **.NET** Works?

- You write code in a high-level language like C# or C++/CLI.
- It compiles to Intermediate Language (IL).
- The CLR executes the IL code using Just-In-Time (JIT) compilation to convert it to native machine code.
- The runtime provides memory management, type safety, security, and more.

C++/CLI allows developers to write managed code (code that is compiled to Microsoft Intermediate Language) and executed by the CLR, while retaining the power and syntax of C++.

Why Use CLR in MTRN3500?



Multithreading

What Does the CLR Do?

A **runtime** is the environment in which a program is executed. It includes all the systems, libraries, services, and tools needed to run the code after it has been compiled.

The CLR is like the engine that runs your **.NET** programs.

CLR Job	Explanation
Runs your code	It takes compiled .NET code (IL, or Intermediate Language) and turns it into actual machine code that the computer can understand.
Manages memory	You don't have to manually delete things. CLR uses garbage collection to clean up unused objects.
Handles errors	If your program crashes or something goes wrong, CLR manages the exceptions.
Keeps things secure	CLR checks types and access rules so one part of code doesn't break another.
Supports multiple languages	CLR can run code written in C#, F#, VB.NET, or C++/CLI, as long as it's .NET.
Handles threads	It manages multithreading, making sure things run smoothly in parallel.

What Does the CLR Do?

Imagine the CLR is like the chef in a restaurant kitchen:

You (the developer) give the chef a recipe (your .NET code).

The recipe is written in any language (C#, F#, C++/CLI).

The chef (CLR) turns the recipe into a dish (machine code).

It handles:

- Cooking (executing code)
- Cleaning (garbage collection)
- Timing (threading)
- Safety (checking ingredients don't spoil = type safety)

C++ vs C++/CLI

Feature	Standard C++ (Native C++)	C++/CLI (Managed C++)
Execution Model	Compiled to native machine code	Compiled to MSIL (Intermediate Language) and runs on CLR
Memory Management	Manual (`new`/`delete`) or smart pointers (`std::unique_ptr`, etc.)	Automatic garbage collection (`gcnew`, no delete)
Syntax	Traditional C++ syntax	Adds .NET's specific syntax
Runtime	No managed runtime (depends on OS and compiler)	Runs on the Common Language Runtime (CLR)
Standard Library	Uses STL (`std::vector`, `std::string`, etc.)	Can use both STL and .NET libraries (`System::String`, `System::IO`)
Event Handling	No built-in event model; implemented manually.	Integrated .NET event and delegate system.
Multithreading	`std::thread`, and OS-level threading APIs.	Uses `System::Threading` namespace with CLR-managed threads and thread pools.

Key Syntax Differences

Concept	Native C++	C++/CLI	Explanation
Pointer to object	`*`	Not used for managed objects; use `^` instead	In C++/CLI, `^` is a handle to an object on the CLR-managed heap.
Reference	`&`	`%` (tracking reference)	% is used for passing handles or managed objects by reference.
Object creation	`new`	`gcnew`	`gcnew` allocates objects on the managed heap, with automatic garbage collection.
Accessing functions	`.` (object) or `->` (pointer)	`->` (handle)	Managed objects accessed through `^` handles always use `->`.
Memory management	`delete` (manual)	Automatic via garbage collector	No explicit deletion for managed objects; CLR frees memory automatically.
Struct	`struct` (value type)	`value struct` (stack-based) or `ref struct` (heap-based)	Distinction between stack and heap allocation is explicit in C++/CLI.
Class	`class` (always heap or stack depending on usage)	`value class` (value type) or `ref class` (reference type)	`ref class` objects live on the managed heap and require `gcnew`.
Namespaces	`namespace std {}`	Same syntax, plus .NET namespaces like `System`	Allows integration with .NET class libraries.
Events	No built-in event keyword	Fully integrated with .NET event model	Can subscribe/unsubscribe to .NET events using delegates.
Threading	OS-level threads (`std::thread`, WinAPI)	.NET threading via `System::Threading`	CLR handles thread creation, scheduling, and cleanup.

Handle (^) in C++/CLI

In C++/CLI, a **handle** is denoted by the caret "^" symbol and is used to point to a managed object – an object that lives on the **managed heap** controlled by the .NET CLR.

- * is used for native (unmanaged) objects.
- ^ is used for managed objects that live on the CLR's garbage-collected heap.
- Using * with managed objects is illegal and causes compilation errors.

Example 1:

Basic declaration: `int^ a = gcnew int(3500);`

Part	Meaning
`int^`	A boxed integer
`gcnew`	Allocates a new object on the CLR **managed heap**
`^`	Denotes a handle (like * for pointers in native C++)

Handle (^) in C++/CLI

Example 2

Basic declaration: `String^ str = gcnew String("MTRN3500");`

Accessing member: `Console::WriteLine(str->Length);`

Part	Meaning
`String^`	Declares a handle to a managed `System::String` object
`gcnew`	Allocates a new object on the CLR managed heap
`->`	Used to access members of managed objects, just like with raw pointers in C++.

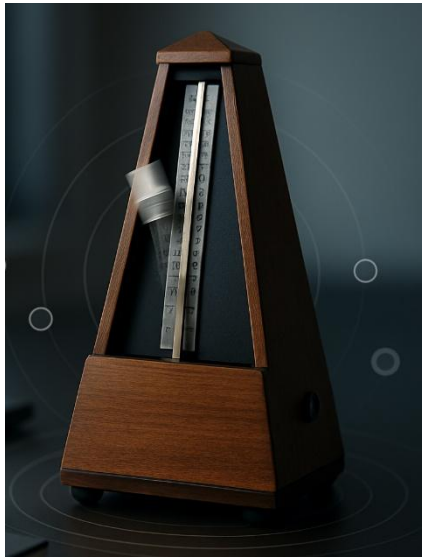
You should **not** use "." to access members through handles (like `str.Length`).

You do not call **delete** on a handle. The CLR will automatically clean it up when it's no longer referenced.

Using C++/CLI for Events

Standard C++ has no built-in event model, whereas C++/CLI integrates fully with the .NET event system.

Feature	Standard C++	C++/CLI
Native Event Support	No built-in event system	Full support via .NET's event keyword
Typical Solutions	Function pointers, callbacks, etc	Uses .NET delegates and event keyword
Memory Management	Manual (no garbage collection)	Automatic (GC-managed delegates and objects)
Multithreading	Manual	Automatic with .NET



Event Source	Common Events	Triggers
Timers	Elapsed	Repeated intervals
UI (WinForms)	Click, MouseMove, TextChanged	User interaction
File System	Changed, Created	File activity
I/O & Network	DataReceived	Serial, sockets
Custom	User-defined	App-specific triggers
Threading	Completed, RunWorkerCompleted	Async finish
Runtime	UnhandledException, ProcessExit	CLR/system lifecycle

Using C++/CLI for Threading

Threading allows your program to perform multiple tasks at the same time.

- One thread = one path of execution.
- Multithreading = multiple tasks running in parallel (or concurrently, depending on CPU/core count).

In C++/CLI, threading is managed using the .NET Framework, especially from the `System::Threading` namespace.

Key Classes for Threading in C++/CLI:

Class	Description
`Thread`	Creates and manages a new thread
`ThreadStart`	Delegate that defines the entry method for a thread
`ThreadPool`	Allows reuse of threads for short-lived tasks
`Monitor, lock`	Synchronisation primitives (to avoid race conditions)
`Mutex, Semaphore`	Advanced thread coordination tools

Appendix 1: Namespace in C++/CLI

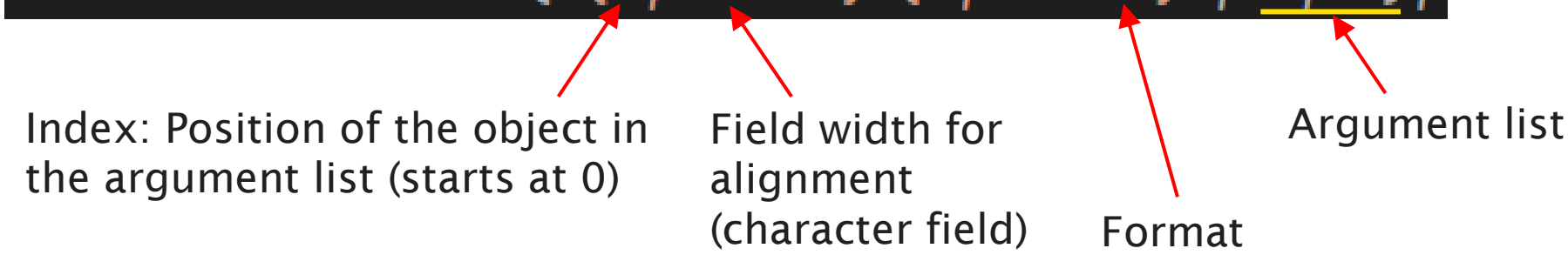
A **namespace** is a way to group related functions, classes, and variables together to avoid name conflicts in large programs or libraries.

`System` is a .NET namespace, which contains core .NET classes (like **Console**, **String**, etc.).

Language	Namespace Used	Remark
C++	`Std`	Refers to the C++ Standard Library (STL), like `vector`, `string`, `cout`, etc.
C++/CLI	`System` and other .NET namespaces	Refers to the **.NET** class library (managed types like `System::String`, `System::Console`)

Appendix 2: Format Printing

```
Console::WriteLine("{0, 15:D5} {1, 15:F3}", i, a);
```



Format	Type	Example	Description
C	Currency	{0:C} → $1,234.00	Locale-aware currency
D	Decimal integer	{0:D5} → 00123	Integer with zero-padding
E	Exponential	{0:E2} → 1.23E+003	Scientific notation
F	Fixed-point	{0:F2} → 123.46	Decimal with fixed digits
G	General	{0:G} → 123.456	Auto scientific/decimal
N	Number	{0:N} → 1,234.56	Thousands separator
P	Percent	{0:P} → 12.34%	Multiply by 100 and append %
X	Hex	{0:X} → 7B	Hexadecimal (uppercase X, lowercase x)

Appendix 3: Stack vs Heap

The **stack** is a special region of memory where function calls and local variables live.

- It works like a stack of plates, i.e. last in, first out (LIFO).
- Memory is freed automatically when a function returns.

The **heap** is a big pool of memory for dynamic allocation.

- You allocate memory explicitly (in C++) using `new` (or `gcnew` in C++/CLI).
- In native C++, you must delete it manually.
- In .NET/C++/CLI, the Garbage Collector cleans it up for you.

Feature	Stack	Heap
Allocation type	Automatic	Manual (C++) or dynamic (GC in .NET)
Memory size	Small (limited)	Large (but slower to access)
Speed	Very fast	Slower
Lifetime	Managed automatically (scope-based)	You decide (until delete or GC)
Syntax (C++)	`int x = 5;`	`int* x = new int(5);`
Deallocation (C++)	Automatic	Must manually call delete

Appendix 4: Value vs Ref class/struct

Feature	`value class/struct`	`ref class/struct`
Memory Location	Typically stored on the **stack**, or inline inside another object	Stored on the **managed heap**
Access Syntax	Used directly (no need for ^ or gcnew)	Accessed via a **handle** (^), created with **gcnew**
Lifetime	Automatically destroyed when it goes out of scope	Managed by the Garbage Collector
Copy behavior	Copy-by-value (deep copy)	Copy-by-reference (shallow copy unless cloned)

- `value class` → behaves like native C++ **struct** → stack/inlined → no need for **gcnew** or ^
- `ref class` → behaves like Java/C# classes → heap-allocated → needs **gcnew** and ^ (however, you still can use *stack semantics,* and the object is on the heap)

`value struct GNSS {…}`

```
array<GNSS>^ GNSSData = { GNSS(1, 2), GNSS(3, 4), GNSS(5, 6), GNSS(7, 8) };
```

the individual `GNSS(…, …)` instances are initially created on the **stack**, but copied into the **managed heap** when placed into the managed array (`array<GNSS>^`).

Appendix 6: “%” Operator in C++/CLI

In C++/CLI, the “%” operator declares a **tracking reference**; it's similar to the “&” (reference) in standard C++, but used for managed objects (i.e., objects on the CLR heap accessed via ^ handles).

Symbol	Used With	Purpose
*	Pointer (int*)	Native pointer to an object (unmanaged)
&	Reference (int&)	Native reference to an unmanaged object
^	Handle (String^)	Managed reference (CLR/GC-managed object)
%	Tracking reference (String^%)	Reference to a handle (^)

Use “%” when you:

- Want to pass a handle by reference to a function (so the function can reassign it)
- Want to avoid making copies of handles
- Need to modify the original handle

Appendix 7: Attaching Event Handler

```
MyTimer->Elapsed += gcnew System::Timers::ElapsedEventHandler(&OnElapsed);
```

Subscribing to the "Elapsed" event

Attach a handler to the event

Create a delegate object (`ElapsedEventHandler`) that wraps the function (`OnElapsed`).

A function pointer to the handler function

This line of code means: "Whenever the timer elapses, call the `OnElapsed` function."

```
Timer (interval = 500 ms)
      |
      |----> Elapsed Event
                  |
                  |----> Calls delegate
                              |
                              |----> Calls OnElapsed(sender, e)
```

Appendix 8: Using Threads

Create and initialise a new thread:

Handle to the Thread object

A delegate type defined in .NET (takes no parameters, return void).

```
Thread^ GNSSThread = gcnew Thread(gcnew ThreadStart(GNSS));
```

Creates a new Thread object on the managed heap. The constructor of Thread takes the ThreadStart^ delegate

Creates a delegate object on the managed heap that wraps the GNSS() function.

```
GNSSThread->Start();
```

Start() tells the .NET CLR to **start the thread**, using the function attached (GNSS()) via the delegate.

Appendix 9: Run the Threads

```
Thread^ GNSSThread = gcnew Thread(gcnew ThreadStart(GNSS));
Thread^ LiDARThread = gcnew Thread(gcnew ThreadStart(LiDAR));

GNSSThread->Start();
LiDARThread->Start();
```

Both threads start independently and **run concurrently**.

Console output is shared, so only one thread can print at a time – it looks like they alternate.

The **illusion of alternation** comes from:

- Equal `Sleep()` durations (50 ms)
- Fair scheduling
- Short tasks (just printing)

GNSS: `Sleep(50)`, Lidar: `Sleep(50)`

```
GNSS acquisition completed
Laser scan acquired
GNSS acquisition completed
Laser scan acquired
GNSS acquisition completed
Laser scan acquired
GNSS acquisition completed
Laser scan acquired
GNSS acquisition completed
Laser scan acquired
GNSS acquisition completed
```

GNSS: `Sleep(50)`, LiDAR: `Sleep(10)`

Time (ms)	GNSS Thread Output	Lidar Thread Output
0	GNSS: "GNSS acquisition..."	Lidar: "Laser scan ..."
10		Lidar: "Laser scan ..."
20		Lidar: "Laser scan ..."
30		Lidar: "Laser scan ..."
40		Lidar: "Laser scan ..."
50	GNSS: "GNSS acquisition..."	Lidar: "Laser scan ..."
60		Lidar: "Laser scan ..."
70		Lidar: "Laser scan ..."
80		Lidar: "Laser scan ..."
90		Lidar: "Laser scan ..."
100	GNSS: "GNSS acquisition..."	Lidar: "Laser scan ..."

```
GNSS acquisition completed
Laser scan acquired
Laser scan acquired
Laser scan acquired
Laser scan acquired
GNSS acquisition completed
Laser scan acquired
Laser scan acquired
Laser scan acquired
GNSS acquisition completed
Laser scan acquired
Laser scan acquired
Laser scan acquired
Laser scan acquired
GNSS acquisition completed
Laser scan acquired
Laser scan acquired
Laser scan acquired
Laser scan acquired
```

Appendix 10: “static” Variable and Function

“static” means: “This belongs to the class itself, not to any one object.”

A **static function** is a class-level function, meaning:

- It does not belong to any specific object of the class – it is shared across all objects.
- It can only access other static members (like static variables or functions).
- It is invoked directly from the class (or via any function).

```
ref class MathHelper
{
public:
    static int Add(int a, int b)
    {
        return a + b;
    }
};
```

```
int result = MathHelper::Add(3, 5);
Console::WriteLine("Sum = {0}", result);
```

- No need to create a MathHelper object.
- Just call the function directly on the class, as it's static.

`static Barrier^ ThreadBarrier;`

This ensures both threads wait until both functions are ready. This only works because ThreadBarrier is static; otherwise, synchronisation would fail.

`static bool Quit;`

Both threads need to stop when the user presses a key. Using a static Quit flag so that both threads can read.

`static void GNSS(…)/static void LiDAR(…)`

- ThreadStart works best with static functions.
- If you want to use instance functions, you must pass **both the object and the function**.