Industrial Programming
Systems Programming & Scripting

Lecture 11: Systems Programming in C#

Characteristics of System Programming in C#

- Build algorithms and data structures from scratch
  - C# is a full-blown object-oriented language
- Use strong typing to help manage complexity of large pieces of software
  - Strong typing throughout the language
- Focus is often on speed of execution
  - Direct management of space and time
- Easy access to low-level operating system is crucial
  - Low-level system libraries

Example of low-level data-structures: Doubly Linked List

- **Goal**: Define a data structure that is space efficient and permits traversal in both directions
- **Method**: Explicit use of references into the heap
- Exercise in resource conscious programming

Basic Structure

```
class LinkedListNode {
 LinkedListNode next;
 LinkedListNode prev;
 private int data;

 public int MyData() { … }
 public void Insert(LinkedListNode node) { …
  }
 public void Remove() { … }
 public void ShowList () { ... }
 public LinkedListNode (int data) { ... }
}
```

Constructor

```
public LinkedListNode (int data) {
  this.data = data;
  // init references
  this.next = null;
  this.prev = null;
 }
```

Lookup

```
public int MyData() {
   return this.data;
}
```

Insertion

```
public void Insert(LinkedListNode node) {
   LinkedListNode nextNode = this.next;
   this.next = node;
   node.prev = this;
   node.next = nextNode;
   if (nextNode != null) { // pitfall
     nextNode.prev = node;
   }
 }
```

Removal (buggy)

```
public void RemoveBuggy() {

  this.prev.next = this.next;
  this.next.prev = this.prev;

  //nulls are put here to ensure stability
  this.next = null;
  this.prev = null;
 }
```

Removal

```
public void Remove() {
  if (this.prev != null) {
    this.prev.next = next; }
  if (this.next != null) {
    this.next.prev = prev; }

  //the nulls are put here to ensure stability
  this.next = null;
  this.prev = null;
 }
```

Showing

```
public void ShowList () {

  Console.WriteLine("{0}",this.MyData());
   if (this.next == null) {
      return;
   } else {
      this.next.ShowList();
   }
 }
```

Showing (reverse order)

```
public void ShowListReverse () {

  Console.WriteLine("{0}",this.MyData());
    if (this.prev == null) {
      return;
    } else {
      this.prev.ShowListReverse();
    }
}
```

C# 6.0: Null-conditional Operators

- These help to tackle NullReferenceExceptions.
- When accessing fields through several levels of a hierarchy, you can use the ? Operator to implicitly check for a null pointer, e.g.

```
myNode = right?.left;
```

- Before that you had to use conditionals like this:

```
if (right.left == null)
    myNode = null;
else
    myNode = right.left;
```

For details see: https://msdn.microsoft.com/en-gb/

C# 6.0: Null-conditional Operators

```
public void ShowList () {

  Console.WriteLine("{0}",this.MyData())
  ;
  this.next?.ShowList();
  }
 }
```

(Un-)managed vs (un-)safe

- **Managed code**: Code which runs within the confines of the .NET CLR.
- **Unmanaged code**: Code which does not run in the CLR, and are totally independent of it.
- **Safe code**: Managed code which has type safety and security embedded within.
- **Unsafe code**: Managed code which involves 'unsafe' operations, such as pointer operations which access memory directly.

Unsafe C# Code

- Unsafe C# code permits direct access to the memory with C-style pointers.
- Direct access data structures must be marked with the keyword `fixed`
- It must be marked with the keyword `unsafe`

Pointers in C#

- Within code marked as unsafe, it is possible to use C-style pointers, i.e.

&x represents the address of the data structure in x

*x de-references a pointer, i.e. it returns the value at location x in memory

- Address arithmetic can be used on pointers, e.g. to traverse an array

A Simple Example with Pointers

- The following method swaps the values of 2 integer variables:

```
unsafe static void Swap(int* x, int *y) {
    int z = *x;
    *x = *y;
    *y = z;
  }
```

- The method should be called like this:

```
int x =5; int y = 7;
Swap (&x, &y);
```

Pointer Arithmetic

- Display a memory area:

```
p = &arr;
for (int i=0; i < arr.Length; i++) {
      Console.WriteLine(*(p+i));
}
```

Example of unsafe C# code

- Copy a block of memory containing ints

```
public unsafe static void memcpy (int
   *p1, int *p2, int n) {
    int *p = p1;
    int *q = p2;
    for (int i = 0; i<n; i++) {
       *q++ = *p++;       }
 }
```

Calling unsafe code

- The memory being processed must be fixed so that garbage collection won't move it while running the unsafe code:

```
int[] iArray = new int[10];
int[] jArray = new int[10];
...
fixed (int *fromPtr = iArray) {
fixed (int *toPtr   = jArray) {
  memcpy(fromPtr, toPtr, 10);
}}
```

Call external functions from C#

- To call an external function, its type and some meta-information has to be declared. For example sum should be a C function, computing the sum of an array of integers:

```
[DllImport ("libsum.so", EntryPoint="sum")]
static unsafe extern int sum(int *p, int n);
```

Call external functions from C#

- We can call this function from C# like this

```
int []arr = new int[10];
for (int i = 0; i<arr.Length; i++) { arr[i]=i; }
fixed (int *p = arr) {
   Console.WriteLine("array initialised to [0..9] =
    {0}", showArr(arr));
   int s = sum(p, 10); // calls a C function
   Console.WriteLine("sum of array, computed on C
    side {0}", s);
}
```

External functions

- This is the C function, computing the sum:

```
int sum (int *p, int n) {
  int s;
  int *q;
  for (s = 0, q = p+n; p<q; s+=*p++) { }
  return s;
}
```

Compiling with external function

- To compile the code, several steps are necessary:
  - First compile the external C code:

```
gcc -O2 -fPIC -c -o libsum.o sum.c
gcc -shared -Wl,-soname,libsum.so -o
libsum.so libsum.o
```

  - Then compile the C# code

```
gmcs -unsafe sumWrapper.cs
```

  - Now you can execute it

```
mono sumWrapper.exe
```

Summary

- Explicit references can be used for resource conscious programming
- Care has to be taken when dereferencing
- This level of programming is similar to using explicit pointers in C; it is
  - powerful and
  - dangerous

Exercises

- Complete the linked list module as presented and write a Tester function.
- Write an append function, that takes 2 linked lists, represented by a reference to their start nodes, and add all elements of the 2nd list to the end of the 1st list
- Develop a 2nd version of append that leaves the input lists unchanged.
- Implement an in-place array reversal function, using explicit pointers.