Programming Proverbs

- 4. ‘‘Be aware of other approaches.’’

- Henry F. Ledgard, ‘‘Programming Proverbs: Principles of Good Programming with Numerous Examples to Improve Programming Style and Proficiency’’, (Hayden Computer Programming Series), Hayden Book Company, 1st edition, ISBN-13: 978-0810455221, December 1975.

Structs and arrays

- recall that a `struct` allows us to create aggregate or compound data types

- useful to store related data types in one container
  - we can also initialise these together

struct example: student record

```
using System;

public struct student
{
  public int enrolment;
  public string name;
  public int year;
}
```

struct example: student record

```
public class test
{
  public static void Main ()
  {
    student s;

    s.name = "Fred Bloggs";
    s.enrolment = 12345678;
    s.year = 1;

    Console.WriteLine ("student {0} has enrolment number {1} \
 and is in the {2} year", s.name, s.enrolment, s.year);
  }
}
```

struct example: student record

- when run we get the following output:

```
student Fred Bloggs has enrolment number 12345678 and is in the 1 year
```

- using the `struct` keeps related data together
- consider the worse alternative

struct example: student record

```
public class test
{
  public static void Main ()
  {
     int enrolment;
     string name;
     int year;

     name = "Fred Bloggs";
     enrolment = 12345678;
     year = 1;

     Console.WriteLine ("student {0} has enrolment number {1} \
 and is in the {2} year", name, enrolment, year);
  }
```

- which has some redeeming feature (no need for `s.fieldname`) but it is less extensible
  - for example suppose we want to implement a list of students

An array of structs

```
class test
{
  static void Main ()
  {
    const int nStudents = 40;
    student[] all = new student [nStudents];

    for (int i = 0; i < nStudents; i++)
    {
       Console.WriteLine ("Please enter name, enrolment and year");

       all[i].enrolment = Int32.Parse (Console.ReadLine ());
       all[i].name = Console.ReadLine ();
       all[i].year = Console.ReadLine ();
    }
  }
}
```

Properties of structs

- structs can be assigned
- structs are passed by **value**, just like fundamental data types
- consider this example:

Properties of structs

```
using System;

public struct student
{
  public int enrolment;
  public string name;
  public int year;
}
```

Properties of structs

```
public class test
{
  public static student tester (student s)
  {
    if (s.name == ".") {
      s.name = "Joe Smith";
      s.enrolment = 11111111;
      s.year = 3;
    }
    return s;
  }
```

Properties of structs

```
  public static void Main ()
  {
    student s;

    s.name = ".";
    s.enrolment = 12345678;
    s.year = 1;
    s = tester (s);

    Console.WriteLine ("student {0} has enrolment number {1} \
 and is in the {2} year", s.name, s.enrolment, s.year);
  }
}
```

Properties of structs

- notice that we can pass `struct` as a value parameter, `s`, is copied into `tester`
- changing `s`, inside function `tester` only affects the local copy of `s` (the parameter)
- the code `returns` this copy which is then assigned to the variable `s` in function `Mains` scope

Properties of structs

- now consider this example:

```
using System;

public struct student
{
  public int enrolment;
  public string name;
  public int year;
}
```

Properties of structs

```
public class test
{
  public static student tester (student s)
  {
    if (s.name == ".") {
      s.name = "Joe Smith";
      s.enrolment = 11111111;
      s.year = 3;
    }
    return s;
  }
```

Properties of structs

```
  public static void Main ()
  {
    student s, t;

    s.name = ".";
    s.enrolment = 12345678;
    s.year = 1;
    t = tester (s);

    Console.WriteLine ("student {0} has enrolment number {1} \
 and is in the {2} year", s.name, s.enrolment, s.year);
    Console.WriteLine ("student {0} has enrolment number {1} \
 and is in the {2} year", t.name, t.enrolment, t.year);
  }
}
```

Properties of structs

- notice the values, `s`, and `t`, inside function `Main`
- here is the output of this program when run:
- ```
  student . has enrolment number 12345678 and is in the 1 year
  student Joe Smith has enrolment number 11111111 and is in the 3 year
  ```
  ```

Properties of structs

- structs can be thought of as a fundamental data type
  - when performing assignment
  - being passed as a parameter
  - returned from a function
  - used as another type in an `array` or `struct`

- as they are passed by value

Properties of structs

- consider the following code
  - note the function no longer returns a value
  - also note the final print out of the `struct` variables

```
using System;

public struct student
{
  public int enrolment;
  public string name;
  public int year;
}
```

Properties of structs

```
public class test
{
  public static void tester (student s)
  {
    if (s.name == ".") {
      s.name = "Joe Smith";
      s.enrolment = 11111111;
      s.year = 3;
    }
  }
```

Properties of structs

```
  public static void Main ()
  {
    student s, t;

    s.name = ".";
    s.enrolment = 12345678;
    s.year = 1;
    t = s;
    tester (s);

    Console.WriteLine ("student {0} has enrolment number {1} and \
 is in the {2} year", s.name, s.enrolment, s.year);
    Console.WriteLine ("student {0} has enrolment number {1} and \
 is in the {2} year", t.name, t.enrolment, t.year);
  }
}
```

Properties of structs

```
student . has enrolment number 12345678 and is in the 1 year
student . has enrolment number 12345678 and is in the 1 year
```

Using a ref parameter to change the value of a struct

- we should of course correct the previous code by using a `ref` parameter
  - recall that the `ref` parameter passes the address of the variable rather than the value

Using a ref parameter to change the value of a struct

```
using System;

public struct student
{
  public int enrolment;
  public string name;
  public int year;
}
```

Using a ref parameter to change the value of a struct

```
public class test
{
  public static void tester (ref student s)
  {
    if (s.name == ".") {
      s.name = "Joe Smith";
      s.enrolment = 11111111;
      s.year = 3;
    }
  }
```

Using a ref parameter to change the value of a struct

```
  public static void Main ()
  {
    student s, t;

    s.name = ".";
    s.enrolment = 12345678;
    s.year = 1;
    t = s;
    tester (ref s);

    Console.WriteLine ("student {0} has enrolment number {1} and \
 is in the {2} year", s.name, s.enrolment, s.year);
    Console.WriteLine ("student {0} has enrolment number {1} and \
 is in the {2} year", t.name, t.enrolment, t.year);
  }
}
```

Using a ref parameter to change the value of a struct

```
student Joe Smith has enrolment number 11111111 and is in the 3 year
student . has enrolment number 12345678 and is in the 1 year
```

Initialising an array of student structs using a function method

```
using System;

public struct student
{
  public int enrolment;
  public string name;
  public int year;
}
```

Initialising an array of student structs using a function method

```
class test
{
  public static student setup ()
  {
    student s;

    s.name = Console.ReadLine ();
    s.enrolment = Int32.Parse (Console.ReadLine ());
    s.year = Int32.Parse (Console.ReadLine ());
    return s;
  }
```

Initialising an array of student structs using a function method

```
  static void Main ()
  {
    const int nStudents = 40;
    student[] all = new student [nStudents];

    for (int i = 0; i < nStudents; i++)
    {
       Console.WriteLine ("Please enter name, enrolment and year");
       all[i] = setup ();
       Console.WriteLine ("{0}  {1}  {2}", all[i].name, all[i].enrolment, all[i].year);
    }
  }
}
```

Initialising an array of structs using a ref parameter

```
using System;

public struct student
{
  public int enrolment;
  public string name;
  public int year;
}
```

Initialising an array of structs using a ref parameter

```
class test
{
  public static void setup (ref student s)
  {
    s.name = Console.ReadLine ();
    s.enrolment = Int32.Parse (Console.ReadLine ());
    s.year = Int32.Parse (Console.ReadLine ());
  }
```

Initialising an array of structs using a ref parameter

```
  static void Main ()
  {
    const int nStudents = 40;
    student[] all = new student [nStudents];

    for (int i = 0; i < nStudents; i++)
    {
       Console.WriteLine ("Please enter name, enrolment and year");
       setup (ref all[i]);
       Console.WriteLine ("{0}  {1}  {2}",
                          all[i].name,
                          all[i].enrolment,
                          all[i].year);
    }
  }
}
```

Classes

- sometimes it is desirable to associate specific functions to `structs`.
  We have seen in previous lectures how to:
  - initialise a struct
  - format output

- it can be desirable to pass `struct` content by reference automatically

- such a data type is known as a `class`
  - and the its functions are called **methods**

Example of a class

- suppose we wanted to build an integer calculator in software
  - we want it to read text such as: `12+5`
  - and emit the answer: `17`
- or: `12*5`
- or: `100/10`
- we could make our program remember the last value and then allow the user to type: `hex`

Typical integer calculator session

```
-> 100/8
12
-> hex
C
-> 5*4/2
10
-> (5-3)*2
4
```

Implementing the calculator

- we notice that if we start to write the code we need to look for digits and symbols

- by the time a sequence of digits finish we have read a symbol
  - ie: `100/`
  - we have read `/` which has told us that the number `100` is complete
  - unfortunately we need to also remember `/` and call the divide routine (method) a little later

- it would be good to implement operator precedence
  - for a limited number of operators: `+`, `-`, `*`, `/`, `(` and `)`

Pseudo code for the calculator

```
line = read_a_string ()
s = my_stream (line)
WriteLine ("{0}", expression (s))

function int expression (my_stream s)
{
   left = term (s)
   while (s.getch () is + or -)
      if (it was '+')
         right = term (s)
         left = left + right
      else
         right = term (s)
         left = left - right
   return left
}
```

Pseudo code for the calculator

```
function int term (my_stream s)
{
   left = factor (s)
   while (s.getch () is * or /)
      if (it was '*')
         right = factor (s)
         left = left * right
      else
         right = factor (s)
         left = left / right
   return left
}
```

Pseudo code for the calculator

```
function int factor (my_stream s)
{
   if (s.getch () is a digit)
      return number (s)
   else if s.getch () is '('
      e = expression (s)
      s.expect (')')
      return e
   s.error ("was expecting a digit or a (")
   return 0
}
```

Class my_stream

- is initialised by a string

- has methods
  - `getch` returns the next character
  - `putch` puts back ch, onto the input stream
  - `expect` expects the next character to be, `ch`, and issues an error message it is not, `ch`
  - `error` generate an error message

Version 2 of the pseudo code for expression

```
int expression (my_stream s)
{
   int left = term (s);

   while ((s.putch (s.getch ()) == '+')
          || s.putch (s.getch ()) == '-')
      if (s.getch () == '+')
         right = term (s);
         left = left + right;
      else
         right = term (s);
         left = left - right;
   return left;
}
```

Code observations

- the pseudo code is drifting closer to final C# code
- the use of `s.putch (s.getch ())` allows us to peep at the next character
  - maybe it is better to refine our `class my_stream` and introduce another method `peep`
- `peep` returns the a copy of the next character, leaving the original in place

Version 3 of the pseudo code for expression

```
int expression (my_stream s)
{
   int left = term (s);

   while ((s.peep () == '+')
          || s.peep () == '-')
      if (s.getch () == '+')
         right = term (s);
         left = left + right;
      else
         right = term (s);
         left = left - right;
   return left;
}
```

Conclusion

- the `my_stream class` needs to be implemented and needs to have the following public methods
- an initialiser
- `getch`, `putch`, `expect`, `error` and `peep`

Conclusion

- as the expression, term and factor are run they consume characters from `my_stream`
  - the instance of `my_stream` (in our example, `s`) needs to be updated every time a character is removed
  - we need one value of `s` (all the parameters mentioning `s` are in effect references to the initial value).
  - which matches the C# implementation of classes (always passed by reference)