Computers in Engineering COMP 208

Moving From Fortran to C – Part 2

Michael A. Hawker

Roots of a Quadratic in C

```
#include <stdio.h>
#include <math.h>
void main() {
   float a, b, c;
   float d;
   float root1, root2;
   scanf ("%f%f%f", &a, &b, &c);

   /* continued on next slide */
```

```
    if (a == 0.0) {
         if (b == 0.0) {
                 if (c == 0.0) {
                         printf ("All numbers are roots \n");
                 } else {
                   printf ("Unsolvable equation"); }
         } else { printf ("This is a linear form, root = %f\n", -c/b);}
    } else {
         d = b*b - 4.0*a*c ;
         if (d > 0.0) {
                 d = sqrt (d);
                 root1 = (-b + d)/(2.0 * a) ;
                 root2 = (-b - d)/(2.0 * a) ;
                 printf ("Roots are %f and %f \n", root1, root2);
         }
         else if (d == 0.0) {
                 printf ("The repeated root is %f \n", -b/(2.0 * a));
         } else {
                 printf ("There are no real roots \n");
                 printf ("The discriminant is %f \n", d);
         }
    }
}
```

Quadratic Roots Revisited

- Let’s make one slight change to the program:

```
   if (a = 0.0) {
        if (b == 0.0) {
                if (c == 0.0) {
                        printf ("All numbers are roots \n");
                } else {
                  printf ("Unsolvable equation"); }
        } else { printf ("This is a linear form, root = %f\n", -c/b);}
   } else {
        d = b*b - 4.0*a*c ;
        if (d > 0.0) {
                d = sqrt (d);
                root1 = (-b + d)/(2.0 * a) ;
                root2 = (-b - d)/(2.0 * a) ;
                printf ("Roots are %f and %f \n", root1, root2);
        }
        else if (d == 0.0) {
                printf ("The repeated root is %f \n", -b/(2.0 * a));
        } else {
                printf ("There are no real roots \n");
                printf ("The discriminant is %f \n", d);
        }
   }
}
```

Quadratic Roots Revisited

- Can you spot the change?
- What do you think the effect is?

```
    if (a = 0.0) {
          if (b == 0.0) {
                   if (c == 0.0) {
                            printf ("All numbers are roots \n");
                   } else {
                     printf ("Unsolvable equation"); }
          } else { printf ("This is a linear form, root = %f\n", -c/b);}
    } else {
          d = b*b - 4.0*a*c ;
          if (d > 0.0) {
                   d = sqrt (d);
                   root1 = (-b + d)/(2.0 * a) ;
                   root2 = (-b - d)/(2.0 * a) ;
                   printf ("Roots are %f and %f \n", root1, root2);
          }
          else if (d == 0.0) {
                   printf ("The repeated root is %f \n", -b/(2.0 * a));
          } else {
                   printf ("There are no real roots \n");
                   printf ("The discriminant is %f \n", d);
          }
    }
}
```

What Happens Here?

- The equivalent statement in Fortran would cause a syntax error
- The expression (a = 0.0) is not of type logical
- But C does not have a type “logical”
- What happens in C?

The C Assignment Operator

- In Fortran, assignment is a statement.
- In C, assignment is an operator
  - It can appear as part of an expression
  - When evaluated it causes a value to be assigned (as in Fortran)
  - It also returns a value that can be used in evaluating the expression
  - If it appears independently,with a ';' after it this value is discarded

The C Assignment Operator

Syntax:

variable = expression

Semantics

1. Evaluate the expression
2. Store the value in the expression in the variable
3. Return the value to the expression

Back to our example

- In C, `(a=0.0)` would assign 0 to a and then return 0 as the value of the expression
- The if condition, with value 0, would be taken as equivalent to “false”
- The else clause would be evaluated and when the root was calculated, there would be an attempt to divide by 0

```
   if (a = 0.0) {
        if (b == 0.0) {
                 if (c == 0.0) {
                         printf ("All numbers are roots \n");
                 } else {
                   printf ("Unsolvable equation"); }
        } else { printf ("This is a linear form, root = %f\n", -c/b);}
   } else {
        d = b*b - 4.0*a*c ;
        if (d > 0.0) {
                 d = sqrt (d);
                 root1 = (-b + d)/(2.0 * a) ;
                 root2 = (-b - d)/(2.0 * a) ;
                 printf ("Roots are %f and %f \n", root1, root2);
        }
        else if (d == 0.0) {
                 printf ("The repeated root is %f \n", -b/(2.0 * a));
        } else {
                 printf ("There are no real roots \n");
                 printf ("The discriminant is %f \n", d);
        }
   }
}
```

Inputting Values in C

- In the program for computing the roots of a quadratic, we began by reading the values for the coefficients a, b and c

Roots of a Quadratic in C

```
#include <stdio.h>
#include <math.h>
void main() {
   float a, b, c;
   float d;
   float root1, root2;
   scanf ("%f%f%f", &a, &b, &c);

   /* continued on next slide */
```

The Input Statement

Syntax:

```
scanf(“format string”,
      list of variable references) ;
```

Semantics:

- Each variable reference is has the form **`&name`**
- For now, ignore the & that appears in front of variable names
- It has to be there but we’ll explain why later

? – What’s that?

- C has many operators and functions that are not part of Fortran
- They come in useful in many applications
- One of these is the “?” operator

The ? Operator

- The ? (*ternary condition*) operator is a more efficient form for expressing simple if selection
- It has the following syntax:

```
e1 ? e2 : e3
```

- It is equivalent to:

```
if e1 then
  e2
else
  e3
```

The ? Operator

To assign the maximum of a and b to z:

```
z = (a>b) ? a : b;
```

which is equivalent to:

```
if (a>b)
 z = a;
else
 z=b;
```

Loops in C

- The basic looping construct in Fortran is the DO loop
- In C, there is a more complex looping command called the **`for`** loop

The C **for** Loop

The C for statement has the following form:

```
for  (e1; e2; e3)
    statement
```

The statement can be a block of statements inside braces { … }

Semantics of the **for** loop

```
for  (e1; e2; e3) statement
```

1. *e1* is evaluated just once before the loop begins. It initializes the loop.
2. *e2* is tested at the end of each iteration. It is the termination condition. If it evaluates to 0 (false) the loop terminates. Otherwise the loop continues.
3. *e3* is evaluated at the end of each iteration. It is used to modify the loop control. It is often a simple increment, but can be more complex.

A C **for** Loop Example

```
main(){
  int x;
  for (x=3; x>0; x--)
      printf("x=%d\n",x);
}
```

...outputs:

```
x=3
x=2
x=1
```

What do these do?

```
for (x=3;((x>3) && (x<9)); x++)

for (x=3,y=4;((x>=3) && (y<9)); x++,y+=2)

for (x=0,y=4,z=4000;z; z/=10)
```

Other Looping Constructs

- The for loop is very powerful and general
- Sometimes we just want to use a simpler, more specific looping command to make the program cleaner and clearer

The While Statement

The while has the syntax:

```
while (expression)
   statement
```

Semantics:

- The expression is evaluated.
- If it evaluates to 0 (false) the loop terminates, otherwise the statement is executed and we repeat the evaluation.

Example

```
int x=3;
main() {
  while (x>0) {
     printf("x=%d\n",x);
     x--;
    }
}
```

...outputs:

```
x=3
x=2
x=1
```

While Statements

- These are usually clear and make the program easier to follow
- The while statement

```
    while (e) s
```

is semantically equivalent to

```
    for (;e;) s
```

What do these do?

- As with many constructs in C, the while statement can interact with other features to become much less clear:

```
while (x--) …;
while (x=x+1)…;
while (x+=5)…;
while (i++ < 10)…;

while ( (ch = getchar()) != ‘q’)
     putchar(ch);
```

Do-while

- Both `while` and `do-while` execute a statement repeatedly and terminate when a condition becomes false
- In the do-while, the testing is done at the end of the loop

```
do
  statement
while (expression)
```

Break and Continue

The break statement is similar to the Fortran EXIT and allows us to exit from a loop

`break;` --exit from loop

The continue statement forces control back to the top of the loop

`continue;` --skip 1 iteration of loop

Using break and continue

```
while (scanf( “%d”, &value ) == 1 && value != 0) {
  if (value < 0) {
        printf(“Illegal value\n”);
        break;     /* Abandon the loop */
  }
  if (value > 100) {
        printf(“Invalid value\n”);
    continue; /* Skip to start loop again */
   }
   /* Process the value (guaranteed between 1 and 100) */
   ....;
} /* end while when value read is 0 */
```

scanf and printf Format Codes

- Syntax:
  - scanf(<formats>, <list of variables>);
  - printf(<formats>, <list of variables>);
- Formats:
  - d: decimal int
  - o: octal int
  - x: hexdecimal int
  - c: character
  - s: string
  - f: real number, floating point
  - e: real number, exponential format

Arrays in C

One dimensional arrays are defined as:

```
<type> name[size];
```

Array subscripts start at 0 and end one less than the array size.

For example the array

```
int list[50]
```

is an array of 50 integer values indexed from 0 to 49

Arrays in C

Accessing individual components is done by indexing.

This is similar to Fortran, except the syntax uses square brackets.

```
Thirdnumber = list[2];
list[5] = 100*list[3];
```

Multi-Dimensional Arrays

Multi-dimensional arrays are defined as:

```
int tableofnumbers[50][50];
```

For more dimensions add more [ ]:

```
int bigD[50][50][40][30]...[50];
```

Elements are accessed as:

```
anumber = tableofnumbers[2][3];
tableofnumbers[25][16]=100;
```

Character Strings

C Strings are defined as arrays of characters.

```
char name[50];
```

C has no string handling facilities built in and so the following are all **illegal**:

```
char first[50],last[50],full[100];
first="Arnold"; /* Illegal */
last="Schwarznegger"; /* Illegal */
full="Mr"+firstname+lastname;
  /*illegal*/
```

Strings

There is a special library of string handling routines in **`<string.h>`**

To print a string use printf with a **%s** control:

```
printf("%s",name);
```

In order to allow variable length strings the \0 character is used to indicate the end of a string.

If we have a string, `char name[50];` we can store the string "Nathan\0" in it.

Functions

Syntax of Function Definitions:

```
returntype name (parameter list)
 {
   localvariable declarations
   functioncode
  }
```

The parameter list is a list of names together with the associated type

Function Definition

```
float findaverage(float a, float b) {
      float average;
      average=(a+b)/2;
      return average;
}
```

We *use* the function as follows:

```
main() {
 float a=5,b=15,result;
 result=findaverage(a,b);
 printf("average=%f\n",result);
}
```

Void Functions

If you do not want to return a value use the return type void:

```
void squares() {
  int loop;
  for (loop=1;loop<10;loop++)
   printf("%d\n",loop*loop);
}
main() {
  squares();
}
```

Array Parameters

Single dimensional arrays can be passed to functions as follows:-

```
float findaverage(int size,float list[]){
  int i;
  float sum=0.0;
  for (i=0;i<size;i++)
     sum+=list[i];
  return sum/size ;
}
```

Note we do not specify the dimension of the array when it is a *parameter* of a function.

Multi-Dimensional Arrays

```
void print_table(int xsize, int ysize,
                 float table[][5]) {
   int x,y;
   for (x=0;x<xsize;x++) {
      for (y=0;y<ysize;y++)
         printf("\t%f",table[x][y]);
      printf("\n");
  }
}
```

Note we must specify the second (and subsequent) dimensions of the array but not the first dimension.

Function Prototypes

Before using a function C must *know* the type it returns and the parameter types.

Function protoypes allow us to specify this information before actually defining the function

This allows more structured and therefore easier to read code.

It also allows the C compiler to check the *syntax* of function calls.

Function Prototypes

If a function has been defined before it is used then you can just use the function.

If NOT then you must *declare* the function prototype. The prototye declaration simply states the type the function returns and the type of parameters used by the function.

Function Prototypes

It is good practice to prototype all functions at the start of the program, although this is not strictly necessary.

Function Prototypes

A function prototype has

1. the type the function returns,
2. the function name and
3. a list of parameter types in brackets

e.g.

```
int strlen(char []);
```

This declares that a function called `strlen` returns an integer value and accepts a string as a parameter.

Coercion or Type-Casting

Mixed mode operations are handled by C very much like Fortran handles them.

Integer values are converted to reals when assigning to a real and when performing an arithmetic operation using integers an reals

Floating point numbers are truncated to integers when assigning then to an integer variable

In C however, the programmer is able to control this using a cast operator () to force the coercion of one type into another

Coercion or Type-Casting

```
int integernumber;
float floatnumber=9.87;
integernumber=(int)floatnumber;
```

assigns 9 (the result is truncated) to integernumber.

```
int integernumber=10;
float floatnumber;
floatnumber=(float)integernumber;
```

assigns 10.0 to floatnumber.

Type Coercion

Coercion can be used with any of the simple data types including char, so:

```
int integernumber;
char letter='A';
integernumber=(int)letter;
```

assigns 65 (the ASCII code for 'A') to integernumber.

Coercion

Another use is to make sure division behaves as requested:

To divide two integers intnumber and anotherint and get a float

```
floatnumber =
    (float)intnumber /(float)anotherint;
```

ensures floating point division.

Global Variables

Global variables are defined in the file containing `main()` before the definition of main

They are shared by all functions in the file without having to pass them as parameters.

Thy can be initialized when declared, or initialized in `main()`

Global Variable Example

```
float sum=0.0;
int bigsum=0;
char letter='A';
func1 (int x, float y) {
  /* uses sum, bigsum, letter, x, y,
     local vars */
}
main()  {
  /* uses sum, bigsum, letter, func1,
      local vars  */
}
```