The C Programming Language

Here's a C program and the corresponding Fortran 90 program:

C

```
% cat helloworld.c
#include <stdio.h>
main () {
  /* No IMPLICIT NONE */
  printf("Hello world.\n");
}
% cc -o helloworldc \
    helloworld.c
% helloworldc
Hello world.
```

Fortran 90

```
% cat helloworld.f90
! No #include <stdio.h>
PROGRAM helloworld
  IMPLICIT NONE
  PRINT *, "Hello world."
END PROGRAM helloworld
% f90 -o helloworldf \
    helloworld.f90
% helloworldf
 Hello world.
```

Here's another corresponding pair of example programs:

C

```
% cat assn.c
#include <stdio.h>
main () { /* main */
  /* No IMPLICIT NONE */
  int x;

  x = 5;
  printf("x = %d\n", x);
} /* main */
% cc -o assnc assn.c
% assnc
x = 5
```

Fortran 90

```
% cat assn.f90
! No #include <stdio.h>
PROGRAM xvardec
  IMPLICIT NONE
  INTEGER :: x

  x = 5
  PRINT *, 'x = ', x
END PROGRAM xvardec
% f90 -o assnf assn.f90
% assnf
 x =            5
```

Some Elements of the C Language

The basic form of the C language is very much like the basic form of Fortran 90.

For example, in C we have:

- *reserved words* (like keywords in Fortran 90):

C	**Fortran 90**
`int, float, char`	`INTEGER, REAL, CHARACTER`
`for, while, do`	`DO, WHILE`
`extern`	`EXTERNAL`

  A complete list of reserved words can be found in *Problem Solving & Program Design in C*, Hanly & Koffman, Appendix E, page AP29.

- *user-defined identifiers* (like symbolic names in Fortran 90): `kilometers_per_mile`, `chickens_thought_of`, `input1`

- *units* of the program:
  - In Fortran 90, we have a program unit, function units and subroutine units (and other things as well).
  - In C, all units are function units.

- Basic data types

Type	C	Fortran 90
Integer	`int`	`INTEGER`
Real	`float`	`REAL`
Complex	Not implemented intrinsically	`COMPLEX`
Boolean	Not implemented intrinsically	`LOGICAL`
Character	`char`	`CHARACTER`

More Elements of the C Language

- Literal constants

Type	C	Fortran 90
Numeric		
Integer	`-22`, `0`, `1234567`	`-22`, `0`, `1234567`
Real	`-19.7`, `0.0`, `12345.67890`	`-19.7`, `0.0`, `12345.67890`
Real Exponential	`1.2345e5`, `-9.8765E-05`	`1.2345e5`, `-9.8765E-05`
Complex	Not intrinsic	`(-7.23,0.91)`
Boolean	`0` for FALSE, any other integer for TRUE	`.FALSE.`, `.TRUE.`
Character		
Single	`'h'`, `'N'`	`'h'`, `"N"`
String	`"hello"`, `"Henry Neeman"`	`'hello'`, `"Henry Neeman"`

- Variable Declarations

C

```
float x, y, z;
int   i, j, k;
char  is_prime;
```

Fortran 90

```
REAL    :: x, y, z
INTEGER :: i, j, k
LOGICAL :: is_prime
```

- Variable Initializations

C

```
float q        = 9.75;
int   n        = 13;
char  n_is_odd = 1;
```

Fortran 90

```
REAL    :: q        = 9.75
INTEGER :: n        = 13
LOGICAL :: n_is_odd = .TRUE.
```

Still More Elements of the C Language

- Assignment Statements

C

```
x        = 0.15;
i        = 122;
is_prime = 0;
```

Fortran 90

```
x        = 0.15
i        = 122
is_prime = .FALSE.
```

- Output Statements

C

```
printf("Hello world.\n");
```

Fortran 90

```
PRINT *, "Hello world."
```

- Input Statements

C

```
scanf("%f", &x);
```

Fortran 90

```
READ *, x
```

- Numeric Expressions

C

```
2 + 5 * 7 / (9.0 - 11)
```

Fortran 90

```
2 + 5 * 7 / (9.0 - 11)
```

- Boolean Expressions

C

```
!1 && 0 || 1

(x > a) && (x < b)
(q < 13) || (r < 12)
```

Fortran 90

```
.NOT. .FALSE. .AND. &
& (.FALSE. .OR. .TRUE.)
(x > a) .AND. (x < b)
(q < 13) .OR. (r < 12)
```

And Yet More Elements of the C Language

- IF blocks

C

```c
if ((x < a) ||
    (x > b)) {
  printf(
    "x outside [a,b]\n");
}

if (x < 0) {
  printf("x is neg\n");
}
else if (x > 1000) {
  printf("x is big\n");
}
else {
  printf("x is small\n");
}
```

Fortran 90

```fortran
IF ((x < a) .OR. &
&   (x > b)) THEN
  PRINT *, &
&   "x outside [a,b]"
END IF

IF (x < 0) THEN
  PRINT *, "x is neg"

ELSE IF (x > 1000) THEN
  PRINT *, "x is big"

ELSE
  PRINT *, "x is small"
ENDIF
```

- Loops

C

```c
for (i = 1; i <= 5; i++) {
  sum = sum + i;
}

inval = 0;
while (inval <= 0) {
  printf(
    "Positive #?\n");
  scanf("%d", &inval);
}
```

Fortran 90

```fortran
DO i = 1, 5
  sum = sum + i
END DO

inval = 0
DO WHILE (inval <= 0)
  PRINT *, &
&   "Positive #?"
  READ *, inval
END DO
```

Basic Structure of a C Program

```
% cat helloworld.c
#include <stdio.h>

main () {
  /* No IMPLICIT NONE */
  printf("Hello world.\n");
}
% cc -o helloworldc helloworld.c
% helloworldc
Hello world.
```

Notice that this example program has several different parts:

1. A `#include` statement (pronounced “pound include”).
2. A function called `main` that’s analogous to a Fortran 90 program unit.
3. An output statement.

Notice also some differences between C and Fortran 90:

1. No `PROGRAM` statement and no `END PROGRAM` statement.
2. No `IMPLICIT NONE` statement.
3. Comments are between
   `/*` and `*/`
4. Every statement either begins with a pound sign
   #
   or is followed by a *block* (set of statements inside curly braces) or ends with a semicolon.
5. The output statement looks weird compared to what we’ve seen in Fortran 90.

User-defined Identifiers in C

User-defined identifiers in C are very much like symbolic names in Fortran 90, and are subject to very similar rules:

1. They must consist of letters, digits and underscores only.
2. They must start with a letter **or an underscore**.
3. They cannot be the same word as a *reserved word*.
4. They **should not** be the same as *standard identifiers* (which we'll look at later).

However, there are some differences between user-defined identifiers in C and symbolic names in Fortran 90:

1. They can start with an underscore:
   `_x` or even `_9`
2. They can be more than 31 characters long.
3. They are **case sensitive**:
   `q` is not the same identifier as `Q`

In fact, the entire C language is completely **case sensitive**.

Variable Declarations

Like Fortran 90, C has several basic data types:

- Integers are denoted `int`.
- Reals are denoted `float`.
- There is no intrinsic complex type.
- There is no intrinsic Boolean type.
- Characters are denoted `char`.

There are other basic data types, but we won't be getting into them now.

The general form of a C variable declaration is:

datatype varname_1, *varname_2*, ... *varname_n*;

For example:

```
float x, y, z;
int   i, j, k;
char  middle_initial;
```

C also supports variable initializations:

datatype varname_1 = *value1*, ... *varname_n* = *valuen*;

For example:

```
float x = 1.2, y = 7.0, z = 1.234e-5;
int   i = 6, j = 9, k = 7;
char  middle_initial = 'J';
```

Assignments

Assignments in C look very much like assignments in Fortran 90, except that an assignment statement in C is followed by a semicolon:

destinationvariable `=` *expression*`;`

For example:

```
% cat assn2.c
#include <stdio.h>

main ()
{ /* main */
  float w, x, y, z;
  int i, j, k;

  w = 0.5; x = 5.0; y = 10.0;
  z =
    x + y * w;
  i = 12; j = 5; k = i / j;
  printf("x = %f, y = %f, z = %f\n",
         x, y, z);
  printf("i = %d, j = %d, k = %d\n",
         i, j, k);
} /* main */
% cc -o assn2 assn2.c
% assn2
x = 5.000000, y = 10.000000, z = 10.000000
i = 12, j = 5, k = 2
```

Notice that this program has multiple assignment statements on the same line:

```
w = 0.5; x = 5.0; y = 10.0;
```

It also has a statement that's spread out over multiple lines, with no continuation character:

```
z =
  x + y * w;
```

In C, multiple statements (of any kind, not just assignments) can appear on a single line, and a single statement can be split into multiple lines, because all *white space* (spaces, tabs, carriage returns) is equivalent, and because statements are separated by semicolons.

Outputting via `printf`

C doesn't have a `PRINT` statement like Fortran 90; instead, C has a function named `printf` that serves the same purpose:

```
printf("Hello world.\n");
```

The `printf` function can also be used to output the values of variables:

```
printf("x = %d\n", x);
printf("i = %d, 7.0 = %f, 1 + 2 / 3 = %d\n",
       i, 7.0, 1 + 2 / 3);
```

Notice the `%d` and `%f` between the quotation marks. What does that mean?

A call to the `printf` function consists of two parts:

1. a *format string*
2. a *print list* (which might be empty)

The *format string* is a collection of text and *placeholders*, which are the little `%d` and `%f` things you've seen in calls to the `printf` function. So, in the above examples, the format strings are:

```
"Hello world.\n"
"x = %d\n"
"i = %d, 7.0 = %f, 1 + 2 / 3 = %d\n"
```

What does the `\n` mean? It's referred to as a *newline*, and it causes a carriage return to be printed. In C, the `printf` function does not print a carriage return at the end of a line unless specifically told to, via the newline character.

The optional *print list*, which can have arbitrarily many elements, is a list of variables, literal constants and/or expressions whose types corresponds to the types of the *placeholders* in the format string. At runtime, the placeholders are replaced by the values of the elements of the print list, in the same order as the print list.

Inputting via `scanf`

Just as C doesn't have a `PRINT` statement, C also doesn't have a `READ` statement; instead, C has a function named `scanf` that serves the same purpose:

```
scanf("%f %d", &thisfloat, &thatint);
```

The `scanf` function is used to input the values of variables, so in the above example, it's used to input the value of a `float` variable named `thisfloat` and an `int` variable named `thatint`.

Notice that the arguments passed to `scanf` are very similar to the arguments passed to `printf`, but that the format string in the call to `scanf` contains just the placeholders.

What does the `&` in front of `thisfloat` mean?

It's called the *address operator*, and it's very complicated, so we're not going to get into it right now.

For now, accept on faith that you **MUST MUST MUST** use an address operator in front of every variable that you input via a call to `scanf`.

`scanf` Example

```
% cat scanftest.c
#include <stdio.h>

main () {
  float this;
  int that, theother;

  printf("Enter a float:\n");
  scanf("%f", &this);
  printf("You entered %f.\n", this);
  printf("Enter two ints:\n");
  scanf("%d %d", &that, &theother);
  printf("You entered %d and %d.\n",
    that, theother);
}
% cc -o scanftest scanftest.c
% scanftest
Enter a float:
 5.7
You entered 5.700000.
Enter two ints:
 2 3
You entered 2 and 3.
% scanftest
Enter a float:
 5.7
You entered 5.700000.
Enter two ints:
 2
 3
You entered 2 and 3.
% scanftest
Enter a float:
 5.7
You entered 5.700000.
Enter two ints:
 2,3
You entered 2 and 1073840608.
```

Notice: if you have multiple inputs on a line, separating them with a comma **doesn't work**.

Arithmetic Expressions in C

Just as in Fortran 90 (and most programming language), C supports arithmetic expressions, and these are **very** similar to arithmetic expressions in Fortran 90. For example, the operations supported in C are:

Operation Name	Kind	Operator	Usage	Effect
Identity	Unary	`+`	`+x`	None
		None	`x`	None
Negation	Unary	`-`	`-x`	Changes sign of `x`
Addition	Binary	`+`	`x + y`	Adds `x` and `y`
Subtraction	Binary	`-`	`x - y`	Subtracts `y` from `x`
Multiplication	Binary	`*`	`x * y`	Multiplies `x` by `y` ($x \times y$)
Division	Binary	`/`	`x / y`	Divides `x` by `y` ($x \div y$)
Remainder	Binary	`%`	`x % y`	Remainder of $x \div y$ (`int` only)

Notice that C doesn't have the exponentiation operator `**` like in Fortran 90, but it does have a remainder operator `%`, which works only for integer division.

The priority order of evaluations in C is similar to Fortran 90, but not identical:

1. parentheses
2. unary identity and negation, **right to left**
3. multiplication, division and remainder, left to right
4. addition and subtraction, left to right

What are the differences between C and Fortran 90?

1. Unary identity and negation have higher priority than multiplication and division, and are performed **right to left** rather than left to right.
2. The remainder operator has the same priority as multiplication and division.

Arithmetic Expressions Example

C

```
% cat exprsc.c
#include <stdio.h>

main () {

  printf(
    "1  -  2  -  3  = %d\n",
     1  -  2  -  3);
  printf(
    "1  - (2  -  3) = %d\n",
     1  - (2  -  3));
  printf(
    "24  /  2  *  4  = %d\n",
     24  /  2  *  4);
  printf(
    "24  / (2  *  4) = %d\n",
     24  / (2  *  4));
  printf(
    "27.0 / 5.0 = %f\n",
     27.0 / 5.0);
  printf(
    "27   / 5   = %d\n",
     27   / 5);
  printf(
    "27   %% 5   = %d\n",
     27   % 5);
}
% cc -o exprsc exprsc.c
% exprsc
1  -  2  -  3  = -4
1  - (2  -  3) = 2
24  /  2  *  4  = 48
24  / (2  *  4) = 3
27.0 / 5.0 = 5.400000
27   / 5   = 5
27   % 5   = 2
```

Fortran 90

```
% cat exprsf.f90

PROGRAM exprs
  IMPLICIT NONE
  PRINT *, &
&   "1  -  2  -  3  = ", &
&    1  -  2  -  3
  PRINT *, &
&   "1  - (2  -  3) = ", &
&    1  - (2  -  3)
  PRINT *, &
&   "24  /  2  *  4  = ", &
&    24  /  2  *  4
  PRINT *, &
&   "24  / (2  *  4) = ", &
&    24  / (2  *  4)
  PRINT *, &
&   "27.0 / 5.0 = ", &
&    27.0 / 5.0
  PRINT *, &
&   "27   / 5   = ", &
&    27   / 5
  PRINT *, &
&   "MOD(27,5)  = ", &
&    MOD(27,5)
END PROGRAM exprs
% f90 -o exprsf exprsf.f90
% exprsf
 1  -  2  -  3  =      -4
 1  - (2  -  3) =       2
 24  /  2  *  4  =      48
 24  / (2  *  4) =       3
 27.0 / 5.0 =  5.400
 27   / 5   =       5
 MOD(27,5)  =       2
```

Notice, in the C program, the double percent sign in the call to the `printf` function for the remainder of 27 divided 5:

`"27 %% 5 = %d\n"`

Because the `%` in a format string indicates the start of a placeholder (e.g., `%d`, `%f`), we use `%%` to indicate the literal `%` character.

Other Properties of Arithmetic Expressions

In C as in Fortran 90, arithmetic expressions can be in *single mode* (all integer operands or all floating point operands) or in *mixed mode* (combined integer and floating point). The rules for C are the same as the rules for Fortran 90 (and many other programming languages).

Likewise, the rule about division by zero – it causes the program to crash – is the same for C as for Fortran 90 (and many other programming languages).

Assignments with Arithmetic Expressions

Just as in Fortran 90, in C we can assign the result of an arithmetic expression to a variable:

```
x = a * b + c / 12;
```

Syntactic Sugar: Assignment Operators

C has special operators called *assignment operators* that allow simultaneous arithmetic and assignment, because these kinds of assignments are extremely common, and C programmers like to type as few keystrokes as possible:

```
a += 2.0;  /* same as a = a + 2.0;  */
b -= 7.5;  /* same as b = b - 7.5;  */
c *= 1E+5; /* same as c = c * 1E+5; */
d /= 12;   /* same as d = d / 12;   */
e %= 3;    /* same as e = e % 3;    */
```

C also provides special operators called the *increment* and *decrement* operators:

```
j++; /* same as j = j + 1; */
k--; /* same as k = k - 1; */
```

The *increment* and *decrement* operators are strange, because they can appear on either the left side or the right side of a variable:

```
++j; /* same as j = j + 1; */
--k; /* same as k = k - 1; */
```

Assignment Operator Example

```
% cat assnop.c
#include <stdio.h>

main () {
  float a, b, c;
  int   d, e, j, k;

  a = 5.0; b = 2.5; c = 999.0; d = 132; e = 8;
  j = 5; k = 8;
  printf("Before calculating:\n");
  printf("  a=%f, b=%f, c=%f,\n", a, b, c);
  printf("  d=%d, e=%d,\n", d, e);
  printf("  j=%d, k=%d\n", j, k);
  a += 2.0;  /* same as a = a + 2.0;  */
  b -= 7.5;  /* same as b = b - 7.5;  */
  c *= 1E+5; /* same as c = c * 1E+5; */
  d /= 12;   /* same as d = d / 12;   */
  e %= 3;    /* same as e = e % 3;    */
  j++; /* same as j = j + 1; */
  k--; /* same as k = k - 1; */
  printf("After calculating:\n");
  printf("  a=%f, b=%f, c=%f,\n", a, b, c);
  printf("  d=%d, e=%d,\n", d, e);
  printf("  j=%d, k=%d\n", j, k);
  ++j; /* same as j = j + 1; */
  --k; /* same as k = k - 1; */
  printf("After calculating again:\n");
  printf("  j=%d, k=%d\n", j, k);
}
% cc -o assnop assnop.c
% assnop
Before calculating:
  a=5.000000, b=2.500000, c=999.000000,
  d=132, e=8,
  j=5, k=8
After calculating:
  a=7.000000, b=-5.000000, c=99900000.000000,
  d=11, e=2,
  j=6, k=7
After calculating again:
  j=7, k=6
```

Increment & Decrement Strangeness

The increment and decrement operators have a curious property: they can be embedded in expressions, in which case order matters:

```
% cat incdec.c
#include <stdio.h>
main () {
  int a = 5, b = 7;
  int resultib, resultia, resultdb, resultda;
  int inc_before = 2, inc_after = 2;
  int dec_before = 5, dec_after = 5;

  printf("Before calculating:\n");
  printf("  a=%d, b=%d\n", a, b);
  printf("  inc_before=%d, inc_after=%d\n",
    inc_before, inc_after);
  printf("  dec_before=%d, dec_after=%d\n",
    dec_before, dec_after);
  resultib = a + b * ++inc_before;
  resultia = a + b *   inc_after++;
  resultdb = a + b * --dec_before;
  resultda = a + b *   dec_after--;
  printf("resultib = %d, inc_before = %d\n",
    resultib, inc_before);
  printf("resultia = %d, inc_after  = %d\n",
    resultia, inc_after);
  printf("resultdb = %d, dec_before = %d\n",
    resultdb, dec_before);
  printf("resultda = %d, dec_after  = %d\n",
    resultda, dec_after);
}
% cc -o incdec incdec.c
% incdec
Before calculating:
  a=5, b=7
  inc_before=2, inc_after=2
  dec_before=5, dec_after=5
resultib = 26, inc_before = 3
resultia = 19, inc_after  = 3
resultdb = 33, dec_before = 4
resultda = 40, dec_after  = 4
```

If the operator appears before the variable name, then the variable is updated **before** its value is used in the expression, otherwise it's updated **after** it's used.

Converting Fortran 90 to C

Let's convert this Fortran 90 program to C.

```
PROGRAM stats
  IMPLICIT NONE
  REAL,PARAMETER    :: stddev_term_power = 2.0
  REAL,PARAMETER    :: stddev_power = 0.5
  INTEGER,PARAMETER :: number_of_elements = 4
  INTEGER,PARAMETER :: decrement = 1
  REAL :: x1, x2, x3, x4
  REAL :: mean, stddevsum, stddev
  PRINT *, "Enter the ", number_of_elements, &
&          " elements."
  READ *, x1, x2, x3, x4
  mean = (x1 + x2 + x3 + x4) / number_of_elements
  PRINT *, "The mean of the ", number_of_elements, &
&          " elements is ", mean, "."
  stddevsum = (x1 - mean) ** stddev_term_power + &
&             (x2 - mean) ** stddev_term_power + &
&             (x3 - mean) ** stddev_term_power + &
&             (x4 - mean) ** stddev_term_power
  stddev =       &
&   (stddevsum / &
&    (number_of_elements - decrement)) ** stddev_power
  PRINT *, "The standard deviation of the ", &
&          number_of_elements,               &
&          " elements is ", stddev, "."
END PROGRAM stats
```

Converting Fortran 90 to C (continued)

Let’s convert this Fortran 90 program to C.

```
PROGRAM eng2metric
  IMPLICIT NONE
  REAL,PARAMETER :: kilometers_per_mile = 1.61
  REAL,PARAMETER :: meters_per_kilometer = 1000.0
  REAL,PARAMETER :: minutes_per_hour = 60.0
  REAL,PARAMETER :: seconds_per_minute = 60.0
  REAL :: distance_in_miles, distance_in_kilometers
  REAL :: speed_in_miles_per_hour, &
&         speed_in_meters_per_second
  PRINT *, "What’s the distance in miles?"
  READ *, distance_in_miles
  distance_in_kilometers = &
&   distance_in_miles * kilometers_per_mile
  PRINT *, "The distance in kilometers is ", &
&          distance_in_kilometers, "."
  PRINT *, "What’s the speed in miles per hour?"
  READ *, speed_in_miles_per_hour
  speed_in_meters_per_second = &
&   (speed_in_miles_per_hour * &
&    kilometers_per_mile *     &
&    meters_per_kilometer) /   &
&   (minutes_per_hour * seconds_per_minute)
  PRINT *, "The speed in meters per second is ", &
&          speed_in_meters_per_second, "."
END PROGRAM eng2metric
```