PH502: Scientific Programming Concepts

Irish Centre for High End Computing (ICHEC)

January 12, 2018

Overview

- Instructors: Adam Ralph (adam.ralph@ichec.ie), Buket Benek Gursoy (buket.gursoy@ichec.ie)
- 4 lectures
- 4 practicals
- 2 assignments. These will be given at the end of the practicals. Deadline: Four weeks. Please do the assignments yourself.
- The goal is to introduce the basic programming concepts that allow construction of simple programs that are fit for purpose.

ICHEC
Irish Centre for High-End Computing

- Overview of Computer Architecture
- Data types & Arithmetic Operations
- Loops & Conditional Statements
- Functions & Subroutines
- The Linux Shell & Compilation
- C Pointers & Memory Management
- Modern Fortran
- Makefiles, Version Control, Doxygen
- Algorithms
- Standard Libraries
- Object-Oriented Programming
- Scripting Languages
- Introduction to HPC

Imperative Programming Paradigm

Knowledge:

- Declarative: Describes properties of things.

$$\sqrt{x} \text{ is } y \text{ such that } y \geq 0 \text{ and } y^2 = x$$

- Imperative: Describes how to do things.

1. Start with a guess: G.
2. If $G * G$ is close enough to x, then G is a good approximation of $\sqrt{x}$. Stop.
3. Else, create a new guess by averaging G and x/G. I.e, $G_{new} = \frac{G+x/G}{2}$.
4. Using this new guess, go back to Step 2.

Computing Machines

Fixed-program computer: Specific design to do a specific computation.

- Pascal's Calculator, 1642: First mechanical calculator. Invented by Blaise Pascal (1623-1662). It could add and subtract two numbers directly and multiply and divide by repetition.
- Difference Engine, 1834: The first mechanical computer. Designed by Charles Babbage (1791-1871). He couldn't finish but a working version was built in 1991. The Difference Engine was used to calculate tables of values from polynomials using the method of finite differences.
- Atanasoff-Berry Computer, 1941: The first electronic-digital computer. Created by John Vincent Atanasoff (1903-1995). It is designed to solve systems of linear equations.
- Turing Machine, 1948: Designed by Alan Turing (1912-1954). The "Turing Machine" is a good representation of the Central Processing Unit (CPU) today. He used his computing machines to break German enigma codes in WWII.

Turing Machine

- A tape divided into cells each of which contains a symbol
- A head that can read/write/erase the symbol and move the tape left and right at a time
- A state register that stores the state of the turing machine
- A finite table of instructions

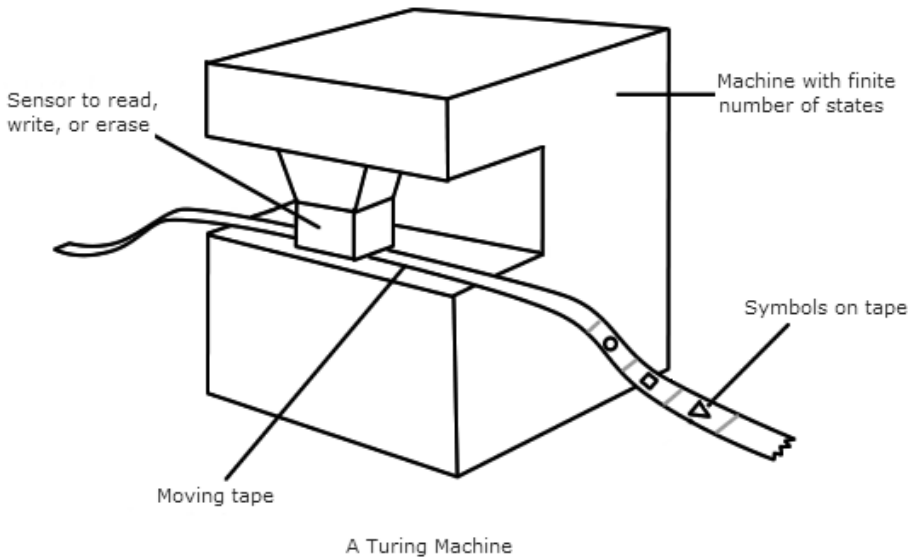
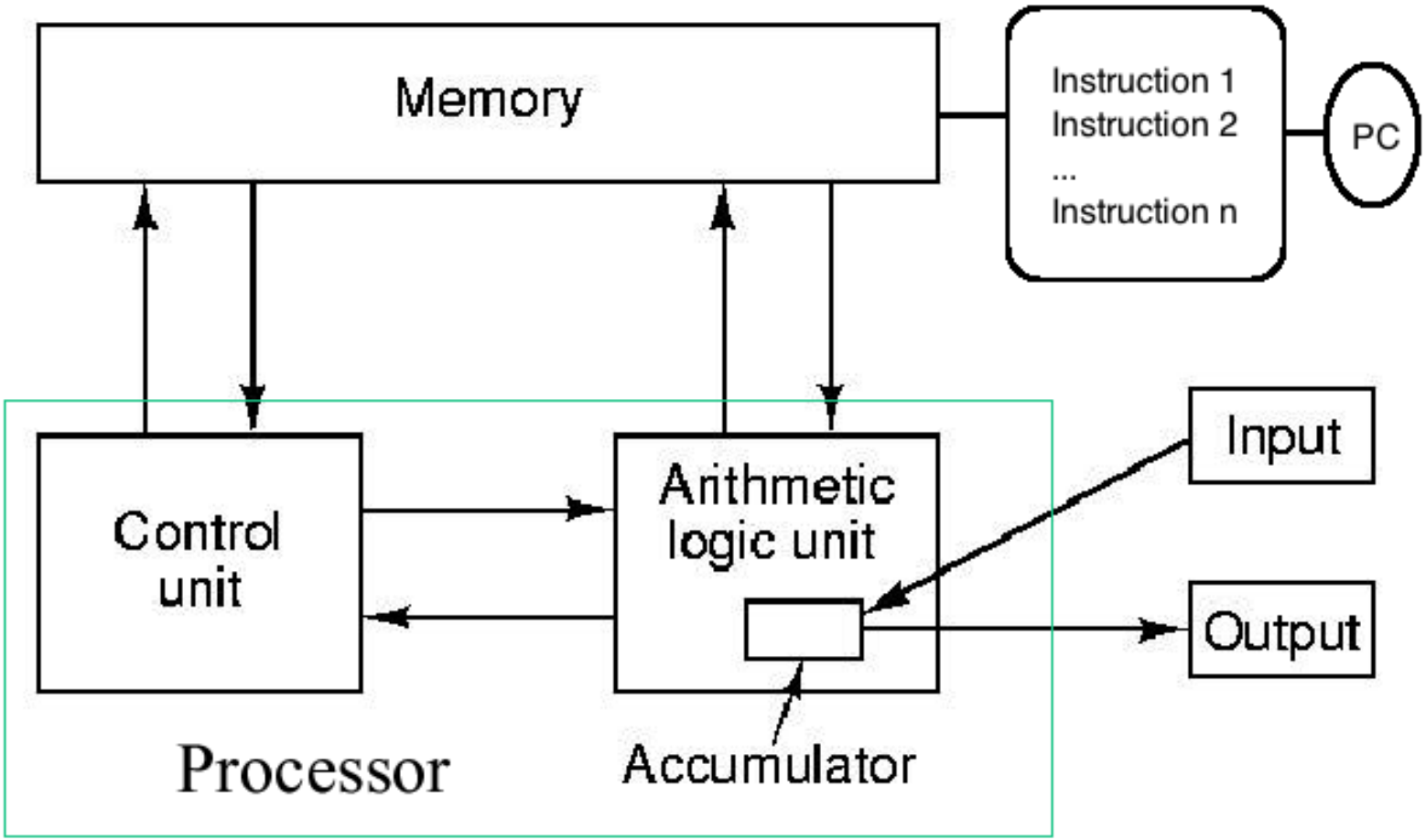


Figure: A Turing Machine

Von Neumann Architecture (1945)

Stored-program computer: Designed to run any computation by interpreting a sequence of program instructions that are read into it.



Programming and Computer Architecture

ICHEC
Irish Centre for High-End Computing

1. CPU: It performs virtually all computations; taking data from memory, executing it, and then returning the results back to memory.
2. Memory: It is the modifiable store but it also holds the data/program. Typically the size of memory will be in the order of 10GB.
   - Memory holds the executable code for different programs currently running on the machine, along with the executable code for the operating system itself. Each program has certain global variables associated with it and uses an area of memory called the stack, which holds all local variables and parameters. When a program completes execution, it releases its memory for reuse by other programs.
3. Disk: It is a permanent data/program store. The program resides on the disk initially, is loaded into the memory when executed. Objects saved to disk are called files. Disks are typically the order of a 1TB.

   The disk is much bigger than the memory, so why load the program into memory?

Memory Addresses

- There are two subgroups of computer architecture: 32-bit and 64-bit.
- Bit: The smallest unit of information; has two settings (0 and 1).
  - 32-bit can store 2^{32} different values and 64-bit can 2^{64}.
- Memory is a long sequence of 0s and 1s; Memory size is generally measured in the number bytes of information a computer can store.
- Byte: A group of eight bits.
  - 1 kilobyte=1024 bytes and 1 gigabyte=2^{30} bytes
- Maximum memory size: The amount of memory that a computer can have is limited by its architecture.

$$\text{32-bit} \quad \frac{2^{32}}{(2^{10})^3} = 4\text{GB}$$

$$\text{64-bit} \quad \frac{2^{64}}{(2^{10})^3} = 2^{34} \simeq 1.7 \times 10^{10}\text{GB}$$

Programming Languages

Program is a representation of a set of instructions.

- Low level - High level
- General - Targeted
- Interpreted - Compiled

Syntax $\approx$ Symbolic representation, Legal expression
Semantic $\approx$ Meaning

Two programs written in different languages could do the same thing (semantics) but the symbols used to write the program would be different (syntax).

Basic Program Layout

```
Main

   declare variables

   initialise variables or load initial state

   command to modify store /* O_1 */

   command to modify store /* O_2 */

   ...

   command to modify store /* O_n */

   save final state to disk as required
end
```

Variables

- Each location in memory is addressed by a number but in the program we give memory locations a name, called a variable. Each variable has two numbers associated with it, its location or $l-value$ and its value or $r-value$. Commands within a program set the $r-value$, the $l-value$ is determined automatically.
- Each variable also has a type. There are four main variable types:
  1. integer,
  2. real,
  3. character,
  4. logical.
- Declaring variables means to give it a unique type. This in turn decides how much memory (how many bytes) are reserved to hold it.
- All variables in C must be declared before use.

Decimal and Binary Representations

- As we have stated already computers naturally use a binary numbering system. However for our convenience, in a program, numbers are expressed in base 10.
  - Decimal representation: base 10, needs numerals 0 - 9.

$$233 = 2 \times 10^2 + 3 \times 10^1 + 3 \times 10^0$$

  - Binary representation: base 2, need numerals 0 1.

$$(110)_2 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = (6)_{10}$$

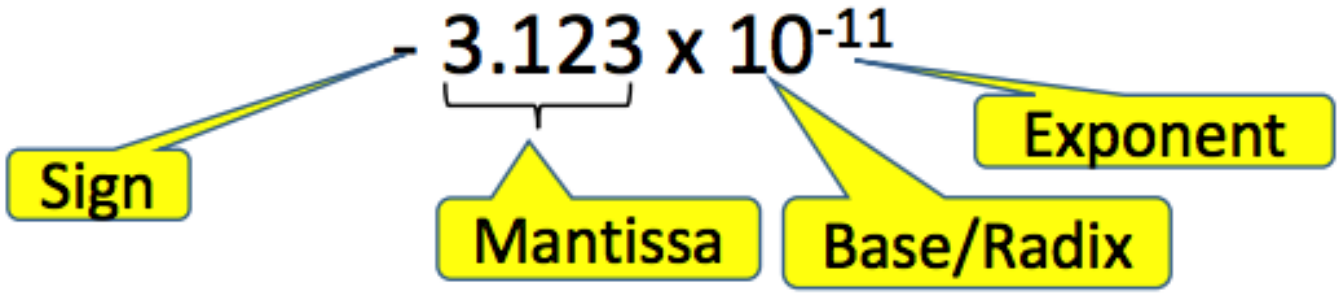
- Ex: Convert from 6 to Base 2.

Integers

- Numbers that can be written without a fractional component.
- Typically integers are represented by four bytes, which means that $2^{8\times4} = 2^{32}$ numbers can be represented.
- However, 1-bit is needed to determine whether the number is positive or negative, thus the maximum absolute value is $2^{31} \sim 4 \times 10^9$.
- An integer variable can store a value in the range -2147483648 to 2147483647.
- Creating an integer variable which has a value out of this range limit may have unpredictable results.

Floating Point Numbers

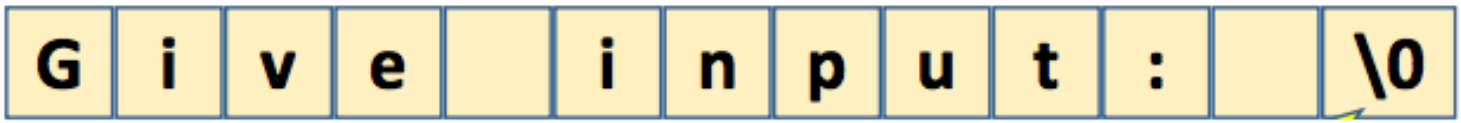
- Real numbers are stored differently to integers. The sign, mantissa and exponent must all be held in this space. The advantage in this, is that a larger range of numbers can be stored.



- There are two typical types of real number single precision, which is held in 4-bytes, and double precision, held in 8-bytes.
- Single precision has an approximate range of $10^{-44} \rightarrow 10^{+38}$. The range of double precision numbers is $\sim 10^{-323} \rightarrow 10^{+308}$. (Negative numbers can also be represented.)

Characters

- A character variable is 1 byte. That makes 2^8 =256 possible characters.
- In the computer, character variables are stored as numbers.
- There is a conversion table to convert the numerical value to the character called the ASCII (American Standard Code for Information Interchange) table. You can see the table on `http://www.asciitable.com`.
  - ASCII Encoding: 'A' = 65, 'B' = 66, '.'= 46, ' ' = 32
- A string is a sequence (array) of characters. It is terminated by a special end-of-string character in C.



To store "Give input: " (12 characters), the array size must be 13 or more.

Logical Variables

- Logical variables have only two values TRUE and FALSE.
- A logical variable could be represented in 1 bit but the smallest addressable memory unit is 1 byte.
- As we shall see later logical variables/expressions are useful in directing the flow of the program's execution.

Declaring Variables

```c
// C Code
int i,j,k;
float x,y,z;
double xx;
char abc, ABC;
char s[5];
```

```fortran
! Fortran Code
integer (kind=4) :: i,j,k
real (kind=4) :: x,y,z
real (kind=8) :: xx
character (len=1) :: abc
character (len=5) :: s
logical (kind=4) :: truefalse
```

Variable Names

ICHEC
Irish Centre for High-End Computing

- The name of a variable can be composed of letters, digits, and the underscore character.
- It must begin with either a letter or an underscore in C and a letter in Fortran.
- Upper and lowercase letters are distinct in C because C is case-sensitive.
- Fortran does not distinguish between upper and lower case, in fact, it assumes all input is upper case.
- Not allowed: 2foo, my foo, Reserved words

Arithmetic Operators

- By arithmetic we mean adding, dividing etc.
- There are three subclasses integer, floating point and logical.
- Binary operators are those that operate on two variables *e.g.* $a + b$. The '+' (adding) is a binary operator.
- Unitary operators act on just one variable *e.g.* -10. The '-' operator makes 10 negative.
- When the compiler interprets an arithmetic expression each operator has a precedence. For example when we write $3a + b$ we mean the 3 multiplies the a first, the result is then added to b. Multiplication has a higher priority than addition. When constructing expressions it is important to bear this priority order in mind.

Variable Assignment

- In C and FORTRAN the $=$ operator assigns a variable an $r-value$. It has not quite the same meaning as $=$ in mathematical expressions.
- Below is valid C and FORTRAN code, x, y are integers.

```c
// C Code
x = 1;
y = x;
x = 2;
```

```
! Fortran Code
x = 1
y = x
x = 2
```

- After the first line x has an $r-value$ of 1. The second line sets y to the same $r-value$ as x *i.e.* 1. After the third line x's $r-value$ is 2 but y's is still 1.
- The expression below adds one onto x's $r-value$ (mathematically it makes no sense).

```
x = x + 1;
```

Integer Arithmetic

- They are used mostly for indices and counters in scientific computing.
- The result of applying the integer arithmetic operators to integers is another integer.
- Integer division: The resulting integer is obtained by discarding the fractional part.
- Modulus operator (%): It evaluates to the remainder obtained after dividing two integers.
- Increment/decrement operators (++/- -): It increases/decreases integer value by one.
  - Prefix form will increment/decrememt the value and then return it.
  - Postfix form will return the value first and then increment/decrement it.

Integer Arithmetic

Examples

```c
int i,j,k;  float z; // Declarations
i = 3%2;          // Remainder (=1)
j = 10/3;         // Division (=3)
++i; --j;         // Increment (=2) / Decrement (=2)
k = i*j;          // Mult. stay within range (=4)
z = 3/4;          // (=0.0)
z = 3.0/4.0;      // (=0.750000)
```

```fortran
integer (kind=4) ::  i,j,k ! Declarations
real (kind=4) :: z
i = mod(3,2) ! Remainder (=1)
j = 10/3;    ! Division (=3)
i=i+1
j=j-1        ! Increment (=2) / Decrement (=2)
k = i*j;     ! Mult. stay within range (=4)
z = 3/4;     ! (=0.0000000E+00)
z = 3.0/4.0  ! (=0.7500000)
```

Floating Point Representation

- IEEE 754 binary floating point standard to represent floating point numbers: *e.g.*
$$((-1)^s \times m \times 2^e)_2.$$
- The sign bit, s, is 0 for positive numbers and 1 for negative numbers.
- The exponent, e, is an integer; it implies finite range.
- m=1.f where f is a binary fraction such that $(1)_2 \leq m < (10)_2$ (in decimal: $1 \leq m < 2$) ; It implies finite precision.
  Example: If $x = -13.125$, then
  Base 10: -1.3125×10 where $s = 1, m = (1.3125)_{10}, e = 1$
  Base 2: $-(1101.001)_2$ where $s = 1, m = (1.101001)_2, e = 3$
- Normalised number: We assume the first bit is 1.

Bit Pattern

- Single precision numbers include an 8-bit exponent field and a 23-bit fraction, for a total of 32 bits.
- Double precision numbers have an 11-bit exponent field and a 52-bit fraction, for a total of 64 bits.
  - The IEEE single precision floating-point representation of x has a precision of 24 binary digits:

$$x = (-1)^s \times (1.m_1 m_2 ... m_{23}) \times 2^e$$

  - The IEEE double precision floating-point representation of x has a precision of 53 binary digits:

$$x = (-1)^s \times (1.m_1 m_2 ... m_{52}) \times 2^e$$

- In decimal representation: Maximum number of decimal digits that can be approximated is $\log_{10}(2^{24}) \approx 7.225$ for single, $\log_{10}(2^{53}) \approx 15.955$ for double precision.
- For single precision $-126 \leq e < 127$; For double precision $-1022 \leq e < 1023$.

Reserved

- Zero is not directly representable in the straight format, due to the assumption of a leading 1. It is defined when an exponent field of all zero bits, and a fraction field of all zero bitsi: $\pm 0.0000\ldots 2^0$. -0 (negative zero) and $+0$ (positive zero) are distinct values.
- Denormalised numbers are when the exponent is zero but the fraction is not. Then the leading digit is assumed to be zero. This represents a number $(-1)^s \times 0.m_1 m_2 \ldots \times 2^0$.
- $\pm\infty$ when the exponent are all 1s and fraction of all 0s.Ex: $1.0/0.0, -1.0/0.0$
- NaN when exponent all 1s and non zero fraction; used to represent a value that does not represent a real number. Ex: $0/0$ or the square root of a negative number.

Precision and Accuracy

- Arithmetic with integers is exact, unless the answer is outside the range of integers that can be represented; floating point arithmetic is not exact since some real numbers require an infinite number of digits to be represented.
- Some simple decimal numbers cannot be represented exactly in binary. For example,

$$0.10 = (0.0001100110011...)_2$$

- Even some integers cannot be represented in the IEEE format. For example, int y = 33554431 (when assigned to 4-byte real) will be printed as

33554432.000000.

Rounding: Assume that $x = 1.m_1 m_2 ... m_n m_{n+1}$ but the floating point representation contains n binary digits.

- If m_{n+1} is 0, chop x to n digits.
- If m_{n+1} is 1, chop x to n digits and add 1 to the last digit of result.

Machine Epsilon

Accuracy: Correctness. For $\pi = 3.14159265359...$, 3.133333333 specifies π with 10 decimal digits of precision and two decimal digits of accuracy. How accurate can a number be stored in the floating point representation?

- The machine epsilon is the difference between 1 and the next larger number that can be stored.
- The smallest floating point number with the property that

$$1 + \epsilon > 1.$$

- In single precision: $\epsilon = 2^{-23} \approx 1.19 \times 10^{-7}$.
- In double precision: $\epsilon = 2^{-52} \approx 2.22 \times 10^{-16}$.

Floating Point Arithmetic

- Not only are there errors in representing numbers but additional errors are introduced by arithmetic operations.
- The IEEE standard specifies that these operations must return the correctly rounded result.
- Floating point addition is not associative and multiplication is not associative nor distributive. This is due to rounding. Assume each floating point number has 7 significant figures and let $a = 1234.567, b = 45.67834, c = 0.0004$.

$$\begin{aligned} a+b &= 1280.245 \\ (a+b)+c &= 1280.245 \\ b+c &= 45.67874 \\ a+(b+c) &= 1280.246 \end{aligned}$$

- Even though the above error is small, when performing many operations these small errors can accumulate.

Avoiding Over/Underflows

- An arithmetic operation can overflow (max value exceeded), result $r - value = \pm\infty$. Underflow is when $r - value$ is smaller than minimum, result set to ± 0.
- If ab and cd underflow because b and d are very small, then

$$\frac{(ab)}{(cd)} = NaN \quad (1)$$

$$\left(\frac{a}{c}\right) \times \left(\frac{b}{d}\right) \neq NaN \quad (2)$$

- If underflowing sets the result to zero then in the top equation there is a division by zero. Dividing by zero may result in a NaN or set to $\pm\infty$.
- It is not always possible to avoid such errors. Restructuring expressions can help as well as using double precision variables.

Logical Arithmetic

- Logical expressions can be implied when the result is either TRUE or FALSE, *e.g.* $a > b$, $a \leq b$.
- Important generic expressions are: a equals b, a not equal to b.
- Logical variables or expressions can be combined. Below is the truth table for logical operators, *Not*, *And* and *Or*. "T" is true and "F" is false.

Operator	a	b	Result
Not	T		F
	F		T
And	T	T	T
	T	F	F
	F	F	F
Or	T	T	T
	T	F	T
	F	F	F

C Operators

Ascending Precedence Order

Operator	Description
()	brackets
!	logical not
++, −−	add/remove 1
*, /	multiply/divide two numbers
%	integer remainder
+, -	add/subtract two numbers
<,<=	less than/or equal
>, >=	greater than/or equal
==, !=	equal to/not equal
&&, ||	logical And/Or
=	assignment

Fortran Operators

Ascending Precedence Order

Operator	Description
()	brackets
**	to the power of
*, /	multiply/divide two numbers
+, -	add/subtract two numbers
.LT., .LE.	less than/or equal
.GT., .GE.	greater than/or equal
.EQ., .NE.	equal to/not equal
.NOT.	logical Not
.AND.	logical And
.OR.	logical Or
=	assignment

Intrinsic Functions

- To make life easier for a programmer, intrinsic functions are available.
- The functions below take a single *double* argument and return a *double* value. Note that the trigonometric functions work with radians.

Function	Description
abs(x)	absolute value (FORTRAN)
fabs(x)	absolute value for (C)
cos(x), acos(x)	cosine and arccosine
exp(x)	e^x
log(x), log10(x)	natural, base 10 log
sin(x), asin(x)	sine and arcsine
sqrt(x)	square root
tan(x), atan(x)	tan and arctan

Program Units

- Programs are split into different units:
  1. Programmer working on large project can divide the workload by making different program units.
  2. Help to decompose the large program into small segments which makes programs easy to understand, maintain and debug.
  3. If repeated code occurs in a program, These can be used to include those codes and execute when needed.
- However there is only one compulsory program unit, main.

```c
int main(void) {
    int i,j,k;
    float x,y,z;

    i=1; j=2;
    x=1.0; y=2.0;

    k = i + j;   z = x*y;
    return 0;
}
```

```fortran
program equiv
  implicit none
  integer (kind=4) :: i,j,k
  real (kind=4) :: x,y,z

  i=1; j=2;
  x=1.0; y=2.0;

  k = i + j; z = x*y;
  stop
end program equiv
```

Loops

- In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.There may be a situation, when you need to execute a block of code several number of times.
- There are two types of loop:
  1. for loop,
  2. while loop.
- **for** loops are designed to execute the same set of instructions a fixed number of times.
- **while** loops repeat until a condition is meet. The danger here is that the loop never terminates.

For Loop

-

```
for(init; condition; update)
{
  code to be executed;
}
```

```
do init, condition, update
  code to be executed
end do
```

- Here is an example of a for loop, which sums the numbers $0 \rightarrow 9$.

```
int i, sum;
sum=0;
for (i=0; i<10; i++) {
  sum = sum + i;
}
```

```
integer (kind=4) :: i, sum
sum=0
do i = 0,9,1
   sum = sum + i
end do
```

- The variable 'i' is the loop counter. It is initialised at 0 and incremented by one each time around the loop. The loop terminates when $i = 10$, that is when the middle statement is false.

While Loop

-

```
while(condition)
{
  code to be executed;
}
```

```
do while(condition)
  code to be executed
end do
```

- The loop terminates when the condition is false.
- Using a while loop in the above examples:

```
int i, sum;
sum=0;
i = 0;
while (i<10) {
  sum = sum + i;
  i++;
}
```

```
integer (kind=4) :: i, sum
sum=0
i = 0
do while (i .lt. 10)
  sum = sum + i
  i = i + 1
end do
```

Loop Control Statements

- break (exit): Terminates the loop.
- continue (cycle): Causes the loop to skip the remainder of its body and causes the next iteration of the enclosing loop to begin.
- goto: Makes the program jump to a given label. Formally, the goto is never necessary and inpractice not being adviced to use.

  The Infinite Loop: A loop becomes infinite loop if a condition never becomes false.

```
for (; ; ) {
   printf("This loop will run forever.\n");
}
```

You can terminate an infinite loop by pressing Ctrl + C keys.

Conditional Statements

- Conditional statements allow different blocks of code to be executed given a logical variable or expression is true or false.
- When a query can be answered with a Yes, we say that the statement contained in the query is True. If we answer the query with a No, we say that the statement contained in the query is False.
- We will introduce the
  1. if-else,
  2. switch-case structure.
- **if-else**: allows us to check whether a statement is true or not.
- **switch-case**: permits us to execute different statements based on the different values of a parameter.

If Statements

- If a given statement is true we want to execute a particular command. However, if it is not true we do not want to do anything.

```
if (statement) {
   commands to be executed
}
```

```
if (statement) then
  commands to be executed
endif
```

- The second situation is when we want to execute one set of commands when the statement is true, and another set of commands when it is not.

```
if (statement) {
   commands to be executed
}
else {
   commands to be executed
}
```

```
if (statement) then
  commands to be executed
else
  commands to be executed
endif
```

If Statements

- An example of this is using an "if else" statement.

```
if (x > 1.0) {
    z = y/x;
}
else {
    z = 0.0;
}
```

```
if (x .gt. 1.0) then
  z = y/x
else
  z = 0.0
endif
```

- These statements can be combined to give,

```
if (i == 0) {
    z = x + y;
}
else if (i == 1) {
    z = x - y;
}
else {
    z = x * y;
}
```

```
if (i .eq. 0) then
  z = x + y
else if (i .eq. 1) then
  z = x - y
else
  z = x * y
endif
```

- conditional operator: Exp1 ? Exp2 : Exp3;

Comparing Floating Point Numbers

- When comparing floating point numbers contained in two locations, all comparisons are on the real number line in terms of their position.
- When comparing numbers the statements can be of the following forms: $x == y, x! = y, x > y, x >= y, x < y, x <= y$.
- When comparing numbers, representation and arithmetic errors should be considered.

```
float x,y;
// Not good
if (x == y) {
  do something}

// Better
if (fabs(x-y) < 0.001) {
  do something }
```

```
real (kind=4) :: x,y
if (x .eq. y) then
  do something
endif

if (abs(x-y) .lt. 0.001) &
                        then
  do something
endif
```

Switch/Case Statements

- Much like a nested if .. else statement. Its mostly a matter of preference which you use.
- The format of this structure is as follows:

```
switch(expression){
    case constant1:
        statement(s);
        break;
    case constant2:
        statement(s);
        break;
    default :
        statement(s);
}
```

```
select case (expression)
  case (constant1)
       statement(s)
  case (constant2)
       statement(s)
  case default
       statement(s)
end select
```

- If a condition is met in switch case then execution continues on into the next case clause also if it is not explicitly specified that the execution should exit the switch statement. This is achieved by using break keyword.

Switch/Case Statements

- This example has the same logic as the previous *if* – *else* statement.

```
switch (i) {
  case 0:
    z = x + y;
    break;
  case 1:
    z = x - y;
    break;
  default:
    z = x * y;
}
```

```
select case (i)
  case (0)
    z = x + y
  case (1)
    z = x - y
  case default
    z = x * y
end select
```

- The *default* case is executed if none of the other conditions are met. If absent then no action is taken.
- The case arguments can either be constant integers, characters or constant expressions.

Switch/Case C Example

- In the example below if the character *c* has a $r - value$ of '0','1','2' or '3' then one is added to *i*.

```
int i; char c;
i=0; c='4';
switch (c) {
  case '0':
  case '1':
  case '2':
  case '3':
    i = i + 1;
    break;
  default:
    break;
}
```

```
if(c=='0'||c=='1'||c=='2'||c=='3'){
  i=i+1;
}
```

Switch/Case Fortran Example

- In FORTRAN at most one case is selected. But multiple selection criteria can be given *e.g.*

```
integer (kind=4) :: i
character (len=1) :: c
i=0; c='4';
select case (c)
  case ('0','1','2','3')
    i = i + 1
  case default
end select
```

Nested Ifs

- If statements can be nested.

```c
if (i > 0) {
  if (i*j > 0) {
    printf("i,j +ve\n");
  }
}
```

```
if (i .gt. 0) then
  if (i*j .gt. 0) then
    write(*,*) ' i,j  +ve'
  endif
endif
```

- Or:

```c
if (i>0 && j>0) {
 printf("i,j +ve\n");
}
```

```
if (i.gt.0 .and. j.gt.0) then
  write(6,*) ' i,j  +ve'
endif
```

Nested Do

- Loops can also be nested.

```
for (i=0; i<10; i++) {
  for (j=0; j<10; j++) {
    sum = sum + i*j;
  }
}
```

```
do i = 0,9,1
  do j = 0,9,1
    sum = sum + i*j
  end do
end do
```

Loop Cycle Control

- **For** loops are designed to cycle a certain number of times. However loops can be exited early or cycles skipped.

```c
int i; float x;
for (i=0; i<100; i++) {
/* Ignore missing data */
   if (isnan(x)) {
     continue;
/* Means end of data, exit
   } else if (x<0.0) {
     break;
   } else {
     sum = sum + x;
   }
}
```

```
do i = 0,99
  if (isnan(x)) then
    cycle
  else if (x.lt.0.0) then
    exit
  else
    sum = sum + x
  endif
end do
```

- If loops are nested they these statements (continue/cycle and break/exit) refer to the current loop.

Labelled Loops

- FORTRAN allows labelled loops. This has two advantages:
  1. make the code more interpretable,
  2. allows cycle and exit to work throughout a nest.

```
loop1 : do i = 1,100
loop2 :   do j = 1,100
            if (x .lt. 0) exit loop1
            if (y .lt. 0) cycle loop2
          end do
        end do
```

Arrays

- Up to now the variables we have used hold only one $r-value$.
- It is possible for one variable name to hold many $r-values$, the variable is then called a variable array or just an array.
- An array is a sequence of data item of the same type.

One dimensional Arrays

- The array length is given in the brackets. All element in an array are of the same type

```
data_type array_name[array_size];
```

```
data_type :: array_name(array_size)
```

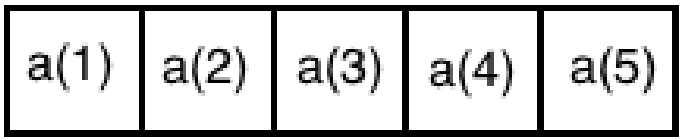
Some examples:

```
int a[10];
float x[100];
char abc[26], ABC[26];
```

```
integer (kind=4) :: a(10)
real (kind=4) :: x(100)
character (len=26) :: abclower
character (len=26) :: absupper
```

Arrays

- Size of array defines the number of elements in an array.
- Individual elements of the array can be accessed using an index. In C the elements are indexed from $0 \rightarrow n-1$ and Fortran from $1 \rightarrow n$, in a length n array.



- The size of array a is 5 times the size of int.
- Below is an example that initialises the array element to its index.

```c
int i,a[5];
for (i=0; i<5; i++) {
   a[i] = i;
}
```

```fortran
integer (kind=4) :: i,a(5)
do i=1,5,1
  a(i) = i
end do
```

- Access to a[6] can cause a fatal error during program execution.

Arrays

Higher Dimension Arrays

- Arrays can be of any dimension depending on the problem.

```
data_type array_name[size1][size2]...[sizeN];
data_type :: array_name(size1, size2, ..., sizeN)
```

- When using matrices it is convenient to use two dimensional arrays:

```
int i,j;
float A[10][10];
for (i=0; i<10 i++) {
  for (j=0; j<10; j++) {
     if (i == j) {
       A[i][i] = 1.0;
     } else {
       A[i][j] = 0.0;
     }
  }
}
```

```
integer (kind=4) :: i,j
real (kind=4) :: A(10,10)
do i = 1,10,1
  do j = 1,10,1
    if (i .eq. j) then
      A(i,i) = 1.0
    else
      A(i,j) = 0.0
    endif
  end do
end do
```

- Size of the array A is 10×10 times the size of float.

Arrays in Memory

- In memory a multidimensional array is saved like a 1D object. Thus there is no computational advantage to using them. There maybe a disadvantage if they are accessed badly.
- In C, the array is saved in row major form. That is to say $[i, j]$ and $[i, j+1]$ are contiguous in memory. In the Fortran the opposite is true $[i, j]$ and $[i+1, j]$ are contiguous in memory.

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

Row-major

1	2	3	4	5	6	7	8	9	10	11	12

Column-major

1	5	9	2	6	10	3	7	11	4	8	12

Functions & Subroutines

- If a certain piece of code is being used over and over again, it is inefficient to keep writing the same code in the main body of the program.
- FORTRAN and C allow us to create a piece of code which is outside the main body of the program but can be referenced from it. These code units are called functions or subroutines.
- A function works similarly to a mathematical function in that it takes input values and returns a value or values. For example

$$y = f(x) \qquad f(x) = x^2.$$

  In this case x is the input and y the output.
- A subroutine or procedure is similar to a function but it does not return a value.

Functions & Subroutines

- There is only one main program, the other subprograms are called fuctions/subroutines. Every program starts from the main.
- When a command (in main) uses a function or subroutine, we say that it has been called from main.
- Function Declaration:

```
return_type function_name(argument-list);
```

- Definition:

```
return_type function_name(argument-list)
{
   body of the function
}
```

-

```
Subroutine subroutine_name(argument-list)
   body of the subroutine
End Subroutine subroutine_name
```

C Function

```c
#include <stdio.h>
float degtorad(float arg);

int main(void) {
    float degang, radang;
    degang = 10.0;
    radang = degtorad(degang);
    printf(" Deg %f, Rad %f\n",degang,radang);
    return 0;
}

float degtorad(float arg) {
    float pi = 3.1415927;
    return( (pi * arg)/180.0 );
}
```

Fortran Example

```
program fexample
  real (kind=4) :: degang,radang
  degang = 10.0
  call degtorad(degang,radang)
  write(*,*) " Deg ",degang," Rad ",radang
end program fexample

subroutine degtorad(arg,arg2)
  real (kind=4), intent(in) :: arg
  real (kind=4), intent(out) :: arg2
  real (kind=4) :: pi=3.1415927
  arg2=(pi*arg)/180.0
end subroutine degtorad
```

Arguments

- In the above example the argument is passed by $r-value$. *degang*'s $r-value$ is copied to that of the dummy argument *ang*.
- A function must be self contained for it to work properly, it has no access to the variables defined in main (unless passed as a arguments).
- The variables in the function all have different $l-values$ to those in main, even if they have the same name.
- Memory is allocated for these variables each time the function is called and destroyed afterwards, including the dummy arguments.

Example

```c
#include <stdio.h>
float degtorad(float degang);
int main(void) {
    float degang, radang;
    degang = 10.0;
    radang = degtorad(degang);
    printf(" Deg %f, Rad %f\n",degang,radang);
    return 0;
}

float degtorad(float degang) {
    float pi = 3.1415927;
    degang = degang + 10.0;
    return( (pi * degang)/180.0 );
}
```

Variable Scope

Scope: A region of the program where a defined variable can have its existence and beyond that variable can not be accessed.

- The variable name must be unique within its scope.
- Local Variables: Variables declared within a code unit are local to that unit. That is to say that they do not exist outside the code unit. Local variables are created and destroyed along with the code unit usage.
- Global variables: Those that are declared outside any program unit. These variables are visible to all program units. They are created and destroyed at program start and termination.
- Arguments: Function Parameters. Special variables in that they provide an information conduit between program units. Space in memory is required for the variable in the calling routine and the dummy argument in the called routine, if passed by $r - value$. The dummy argument is destroyed when the routine terminates.

C Example

```c
#include <stdio.h>
#include <math.h>
float pi;
float degtorad(float arg);

int main(void) {
   float degang, radang;
   pi = atanf(1.0)*4.0;
   degang = 10.0;
   radang = degtorad(degang);
   printf(" Deg %f, Rad %f\n",degang,radang);
   return 0;
}

float degtorad(float arg) {
   return( (pi * arg)/180.0 );
}
```

- This program is slightly more efficient in that the variable pi is not created and destroyed each time the function is called.

Question

- A program can have same name for local and global variables but value of local variable inside a function will take preference.

```c
#include <stdio.h>
#include <math.h>
float pi;
float degtorad(float arg);

int main(void) {
   float degang, radang;
   float pi;
   pi = atanf(1.0)*4.0;
   degang = 10.0;
   radang = degtorad(degang);
   printf(" Deg %f, Rad %f\n",degang,radang);
   return 0;
}

float degtorad(float arg) {
   return( (pi * arg)/180.0 );
}
```

Schematic of Scope Rules