Data Analysis in Geophysics
ESCI 7205

Bob Smalley
Room 103 in 3892 (long building), x-4929

Tu/Th - 13:00-14:30
CERI MAC (or STUDENT) LAB

Lab – 23, 11/14/13

Q: how many programmers does it take to change a light bulb?
A: none, that's a hardware problem

Urban legend: but good!

At a computer expo (COMDEX), Bill Gates reportedly compared the computer industry with the auto industry and stated: "If GM had kept up with the technology like the computer industry has, we would all be driving $25.00 cars that got 1,000 miles to the gallon."

- - - - - - - - - - - - - - - - - -

In response to Bill's comments, General Motors issued a press release (by Mr. Welch himself) stating:

If GM had developed technology like Microsoft, we would all be driving cars with the following characteristics:

1. For no reason at all, your car would crash twice a day.

2. Every time they repainted the lines on the road, you would have to buy a new car.

3. Occasionally, executing a manoeuver such as a left-turn would cause your car to shut down and refuse to restart, and you would have to reinstall the engine.

4. When your car died on the freeway for no reason, you would just accept this, restart and drive on.

5. Only one person at a time could use the car, unless you bought 'Car95' or 'CarNT', and then added more seats.

6. Apple would make a car powered by the sun, reliable, five times as fast, and twice as easy to drive, but would run on only five per cent of the roads.

7. Oil, water temperature and alternator warning lights would be replaced by a single 'general car default' warning light.

8. New seats would force every-one to have the same size butt.

9. The airbag would say 'Are you sure?' before going off.

10. Occasionally, for no reason, your car would lock you out and refuse to let you in until you simultaneously lifted the door handle, turned the key, and grabbed the radio antenna.

11. GM would require all car buyers to also purchase a deluxe set of road maps from Rand-McNally (a subsidiary of GM), even though they neither need them nor want them. Trying to delete this option would immediately cause the car's performance to diminish by 50 per cent or more. Moreover, GM would become a target for investigation by the Justice Department.

12. Every time GM introduced a new model, car buyers would have to learn how to drive all over again because none of the controls would operate in the same manner as the old car.

13. You would press the 'start' button to shut off the engine.

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

Read more at (the definitive page for debunking urban legends)
http://www.snopes.com/humor/jokes/autos.asp#V1VP4pGj0UmT6rlq.99

Programming languages must be:

- totally unambiguous (unlike natural languages, for example, English),
- expressive --- it must be fairly easy to program common tasks
- practical --- it must be an easy language for the compiler to translate
- simple to use.

All programming languages have a very precise syntax (or grammar). This ensures all syntactically-correct programs have a single meaning.

users.aber.ac.uk/ruw/teach/340/fortran_course/lecture1.ppt

Intro to FORTRAN

Early 1950s		*"order codes" (primitive assemblers)*
1957	FORTRAN	*the first high-level programming language*
1958	ALGOL	*the first modern, imperative language*
1960	LISP, COBOL	*Interactive programming; business programming*
1962	APL, SIMULA	*the birth of OOP (SIMULA)*
1964	BASIC, PL/I	
1966	ISWIM	*first modern functional language (a proposal)*
1970	Prolog	*logic programming is born*
1972	C	*the systems programming language*
1975	Pascal, Scheme	*two teaching languages*
1978	CSP	*Concurrency matures*
1978	FP	*Backus' proposal*
1983	Smalltalk-80, Ada	*OOP is reinvented*
1984	Standard ML	*FP becomes mainstream (?)*
1986	C++, Eiffel	*OOP is reinvented (again)*
1988	CLOS, Oberon, Mathematica	
1990	Haskell	*FP is reinvented*
1990s	Perl, Python, Ruby, JavaScript	*Scripting languages become mainstream*
1995	Java	*OOP is reinvented for the internet*
2000	C#	

So why should we waste time learning a dying niche language.

(according to C aficionados and computer scientists everywhere for the last 30-40 years; like UNIX, FORTRAN will just not go away!)

CONS

- Legacy FORTRAN code can be difficult to read and refactor due to age.

- Older FORTRAN programs may not have obeyed any recognizable methodology.

- Also, GOTO statements.

(this sin alone is enough to justify capital punishment)

So why should we waste time learning a dying niche language

PROS

- A majority of supercomputers run programs written in FORTRAN

-Monster.com lists ~50 jobs requiring FORTRAN experience.

(20 of those 50 also require security clearance.)

So why should we waste time learning a dying niche language

PROS

-FORTRAN is still the fastest when it comes to number crunching:

intensive mathematical models, such as weather prediction,
computational science, air and fluid modeling, etc.

(exploration seismology is one of the "etc.". So is passive seismology but there's no $$ in that – except for CTBT.)

So why should we waste time learning a dying niche language

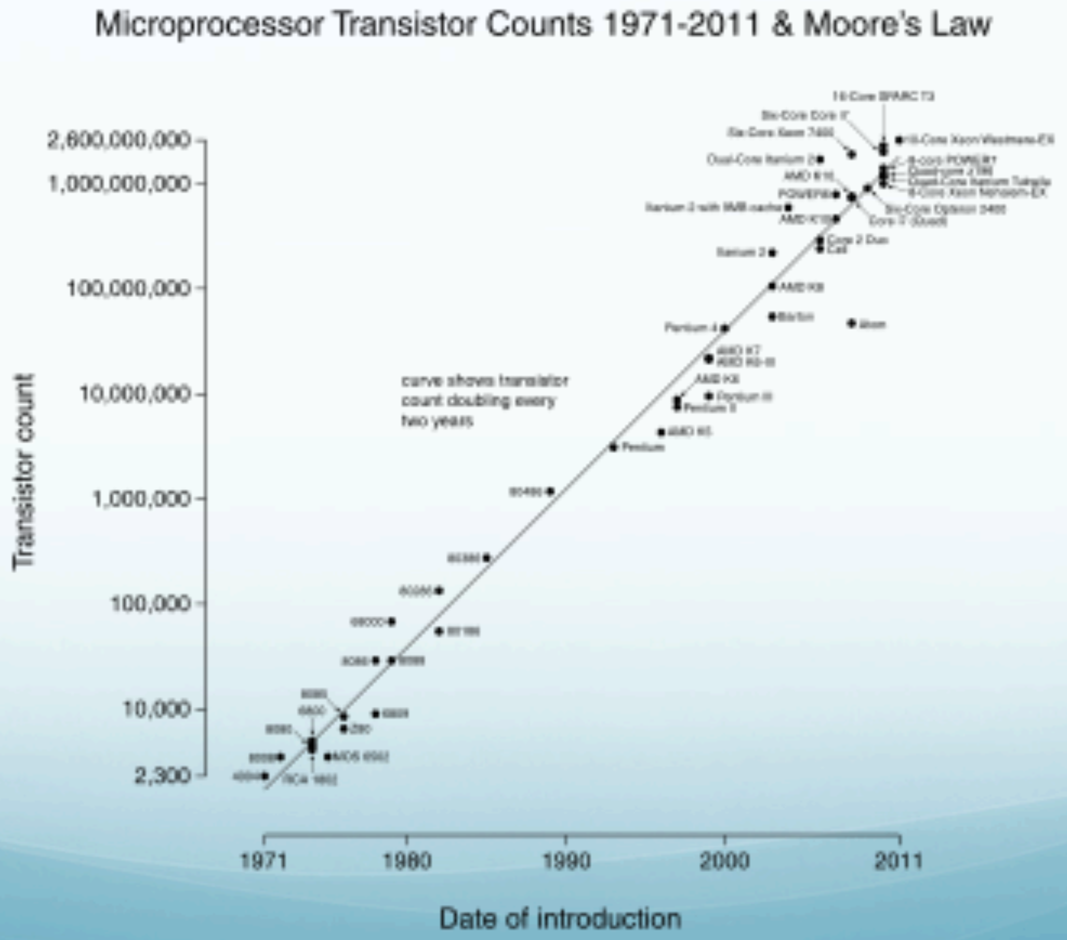
If you can’t win, insult!

-FORTRAN isn't unique. Everything FORTRAN does can also be done by more powerful languages, starting with C.

- FORTRAN is fast with mathematics, but Moore's Law is faster.

Moore's Law

observation that, over the history of computing hardware, the number of transistors on integrated circuits doubles approximately every two years.



So why should we waste time learning a dying niche language

If you can't win, insult!

-FORTRAN was an important step in programming languages, but it now only caters to the niche market of High Performance Computing.

Fortran was invented in 1957, C in 1972, neither is a spring chicken.

(C does not even contain I/O [not even a WOM - write only memory!].

C also did not contain any math functions, there was no "standard" math library well into the 80's, you had to write your own.

Even now, you have to link in libraries for just about anything and everything.)

Fortran was invented in 1957, C in 1972, neither is a spring chicken.

Each was developed to do a different task on the computer, number crun
ch vs. operating system.

There are no flying submarines, and for good reason, use the appropriate tool!

Good history of FORTRAN

`http://en.wikipedia.org/wiki/Fortran`

Latest versions (2003, 2008) support object oriented programming, structures (f90), pointers (f90), dynamic memory allocation, and all the other "good stuff" C is so proud of.

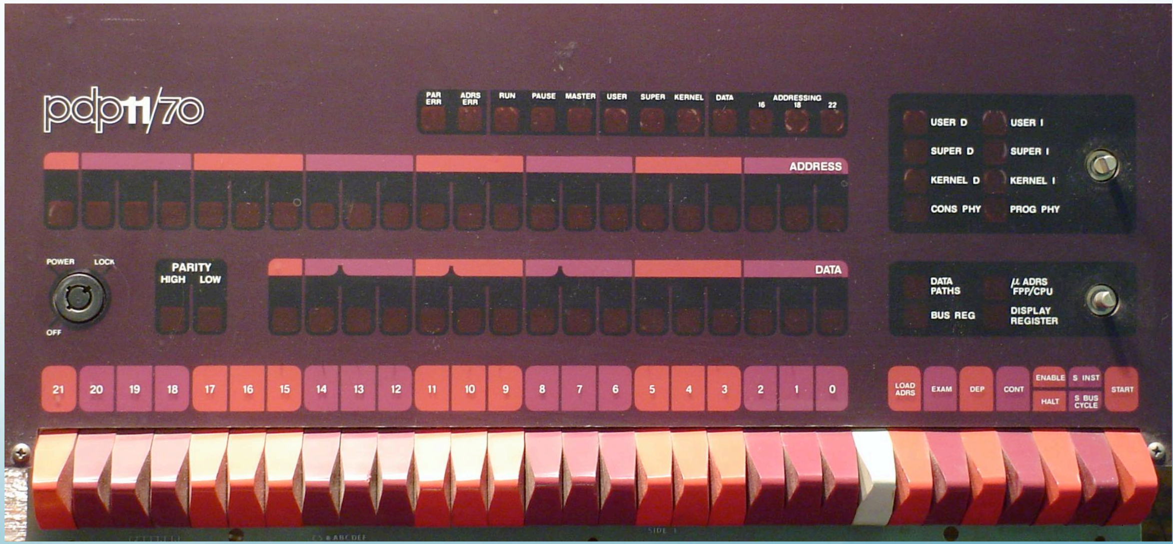
And it still has "goto"!

B 4 FORTRAN your choices were

i) Machine code

Address   instruction and additional information/data

```
8020    78
8021    A9  80
8023    8D  15  03
8026    A9  2D
8028    8D  14  03
802B    58
802C    60
802D    EE  20  D0
8030    4C  31  EA
```



http://c2.com/cgi/wiki?MachineCode

B 4 FORTRAN your choices were

ii) Assembler code

Now get to type on teletype and have to run a program (an assembler) to change what you wrote into what we saw on last page (exactly – line by line, element by element)

```
Start: .org $8020
SEI                  8020     78
LDA #$80             8021     A9 80
STA $0315            8023     8D 15 03
LDA #$2D             8026     A9 2D
STA $0314            8028     8D 14 03
CLI                  802B     58
RTS                  802C     60
INC $D020            802D     EE 20 D0
JMP $EA31            8030     4C 31 EA
```

FORTRAN (I): 1954-57

FORTRAN II : 1958

FORTRAN III : 1958 (not released to public)

FORTRAN IV : 1961

Then things sort of fell apart (the plusses and minuses of standards)

FORTRAN 77 : ~1977

Fortran 90, 95 : in 1990's

Fortran 2003: 2004

Fortran 2008: 2010

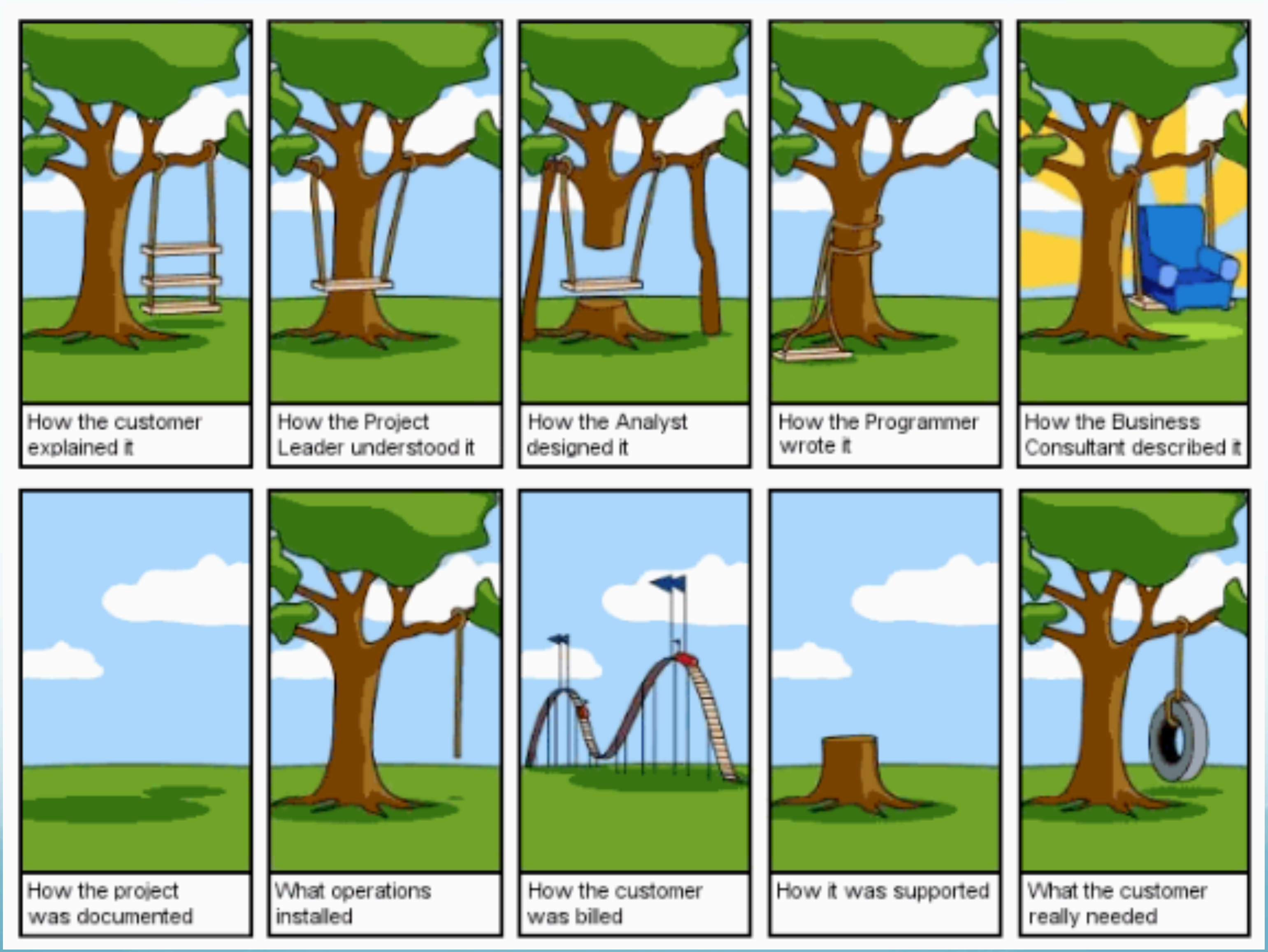
The big problem is "standards" and compatibility

Things like data files (or disks) from my computer working on yours, code portability, ...

Standards make it easy to get stuff done the old way on every machine, but kill innovation.

The Fortran community has had a hard time coming up the standards.

What happens when a committee does it


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

General approach to any computational problem

Statement of problem: clearer this can be done, the easier the development and implementation

Solution Algorithm: Exactly how problem will be solved.

Implementation: Breaking algorithm into manageable pieces that can be coded into language of choice, and putting all the pieces together to solve the problem.

Verification: ✓ implementation solves original problem. Often this is the most difficult step as don't know "correct" answer (reason for program in the first place).

Fortran derives from “FORmula TRANSlator” and was the first successful HLL, optimizing, computationally oriented, compiler.

What is a compiler?

So far we've been working with interpreted languages (shell, Matlab)

Or stuff we have not discussed as to what type of program it is

(SAC, GMT

both are compiled, the former written in FORTRAN, the latter in C – and proud of it – UNIX/C is evangelical).

A **compiler** is a computer program (or set of programs) that transforms source code written in a programming language (the *source language*) into another computer language (the *target language*, often having a binary form known as *object code*).

The most common reason for wanting to transform source code is to create an executable program.

http://en.wikipedia.org/wiki/Compiler

The object code/file is not yet a useable program. It has translated what we want to do (print for example), but does not yet have all the parts (the machine code that does the printing).

We need to "link" our object code to the missing parts.

A **linker** or **link editor** is a computer program that takes one or more object files generated by a compiler and combines them into a single executable program.

After - http://en.wikipedia.org/wiki/Linker_%28computing%29

One used to have to do this in two steps

Compile

then

Link

Now the linking step is typically done by the call to the compiler (although it can be skipped).

Compiled languages i.e., source code is created with an editor as a plain text file.

Program then needs to be “compiled” (converted from source code to machine instructions with relative addresses and undefined external routines still needed).

http://geoweb.mit.edu/~tah/12.010

Compiled languages i.e., source code is created with an editor as a plain text file.

External routines are those needed to do such operations as read disk files, write to the screen, read the keyboard strokes etc.)

The compiled routines (called object modules) need then to be linked or loaded.

Linking creates an executable with relative addresses resolved and external routine loaded from the system and user libraries.

The executable can then, in most cases, be run on any machine with the same architecture.

Compiling and linking can be done in one user step.

Many modern compilers come as part of a development environment

- sort of like Matlab, but for Fortran, C, C++ -

They have context sensitive editors that indent, color, etc., your code to “help” you.

They have interactive debuggers.

Stuff to organize, reuse, the parts.

Many modern compilers come as part of a development environment

Lots of tools to slow *you* down in the short run while promising to get *you* to nirvana in the long run.

Analysis of Fortran Program

Code is delimited by

```
PROGRAM

…

END PROGRAM
```

statements. Between these there are two distinct sections.

- Specification Part
- Execution Part

Analysis of Fortran Program

Specification Part

- specifies named memory locations (variables) for use
- specifies the type of the variable,

Analysis of Fortran Program

Execution Part

(does the work!)

- may read in data
- calculates something (FORmula TRANslator) (or does something)
- output something - data/results/action.

How to Write a Computer Program

5 main steps:

1. Specify problem,

2. develop algorithm, analyse and break down into series of steps towards solution,

3. write the code in some programming language,

4. compile (for Fortran, C, C++, etc.) and run

5. test the program.

It may be necessary to iterate between steps 3 and 5 in order to remove any mistakes.

The testing phase is very important (and hardest, followed by steps 1 and 2).

users.aber.ac.uk/ruw/teach/340/fortran_course/lecture1.ppt

Write first FORTRAN program

The famous "hello world"
Edit this into a file. On UNIX systems you must end the file name with "`.f`" or "`.f90`" (or "`.f95`"...).

```
      print *, "hello world"
      end
```

Note that there are <u>6 spaces</u> (many compilers accept tab, but this is an "extension") at the beginning of each line. This ~~is~~ <u>was</u> required for 50 years and 90% of existing FORTRAN files will be like this.
Save it.

Free and Fixed format

Original Fortran used a so-called fixed format, where the first 5 columns were used for labels, column 6 for a continuation character and columns 7-72 for code (and 73-80 for sorting). Such programs can be written in Fortran 90, but may need to have an extension `.f`.

The default inFortran 90 is free format, since there is far less need for labels. Free format programs must have extension `.f90` (or "`.f95`"...).

We will use free format from here on.

users.aber.ac.uk/ruw/teach/340/fortran_course/lecture1.ppt

Now we have our program written and need to do something to be able to run it.

Enter the "compiler"

```
gfortran your_filename.f90
```

This will produce a file named "`a.out`" (on UNIX/ LINUX machines).

This file will be executable (have executable privileges)

To “run” or “execute the program, just type its name (like any other program).

```
a.out
```

Or

```
./a.out
```

If you don’t have “dot” in your path.

Write your first Fortran program to print

“hello world”

Not very useful to continuously clobber your programs (but typical UNIX), so save it with a useful name (notice FORTRAN, or any other, compiler does not work with input redirection {<, <<} or output redirection {>}).

```
gfortran your_filename.f90 –o ex_name
```

This will produce a file named "ex_name"

Add some conventions recommended by computer science people

```
program hello
print *, “hello world”
end
```

Add “program” declaration at top of file with main program name.

The “*” in the print statement tells it where to print (“*” goes to standard output)

For now – assume variable assignment statement is same as Matlab, SAC, etc.

```
x=5
```

Makes variable named **x** equal to 5, for example.

But new twist wrt Matlab

Variables in FORTRAN can be integers (also exists in C, does not exist, or exists poorly, in Matlab), reals (floating point, "sort of" as in Matlab), complex ("sort of" as in Matlab, but do not exist in C), characters (everybody does it differently).

Variable names must start with a letter.

Can have numbers (but not first character).

Can have underscore "_", dollar sign "$"

Can't have other characters.

Variable name length limited:

6 characters (up to FORTRAN 77)
63 characters (Fortran 90 and higher)

Types of numbers in FORTRAN

(automatic error generator, shares this feature with MATLAB, C makes it impossible to do this error - or mix things even when you want to do it.)

If you don't do anything particular all variables that start with the letters

`i` through `m`

are integers

Size has changed through time – now they are 32 bits, = 4 bytes, long

This is called <u>implicit typing</u> (as in type=kind of variable)

It saves typing (I suppose this is now called keyboarding)

but

every typo ("keyboardo"?) becomes a new variable (as in Matlab).

It is very hard to debug this type error.

Types of numbers in FORTRAN

(automatic error generator, shares this feature with MATLAB, C makes it impossible even when you want to do it.)

If you don't do anything particular all variables that start with the letters

a through h and o through z

are reals (floating point)

Size has changed through time – now they are 32 bits, = 4 bytes, long

Changing Implicit typing

dangerous

this defines all variables that start with the letters **`i`** through **`n`** to be reals and **`a`** through **`h`** and **`o`** through **`z`** to be integer (and does not solve the typo creating new variable problem).

```
implicit real i-n
implicit integer a-h, o-z
```

Explicit typing

What if you want something besides an integer or a real of the default size?

Use explicit typing.

Explicit typing

What is available in terms of types.

integer
real
complex
logical
character

(pointer [is special integer])

Explicit typing

```
integer :: a, b, c
real :: i, j, k
```

We can explicitly declare what type each variable will be (so the variables **`a`**, **`b`** and **`c`** are integer and **`i`**, **`j`** and **`k`** are real).

They are the default size.

Explicit typing

Numeric choices continued

`integer, integer*4` (the same, 32 bit, 4 byte integer)

`integer*2` (16 bit, 2 byte integer)

`integer*1, byte` (8 bit, 1 byte integer)

`integer*8` (64 bit, 8 byte integer, called "double precision")

`integer*16` (128 bit, 16 byte integer, called "quad precision")

Explicit typing - part of variable definition

```
type, options :: names, initializations
```

the :: notation delimits the type and attributes from variable name(s) and their optional initial values, allowing full variable specification and initialization to be typed in one statement (in previous standards, attributes and initializers had to be declared in several statements).

Explicit typing - part of variable definition

```
type, options :: names, initializations
```

While the "::" is not required in above examples (as there are no additional attributes and initialization), most Fortran-90 programmers acquire the habit to use it everywhere.

Explicit typing

Numeric choices continued

Similar for reals (but can't go smaller than basic size)

```
real, real*4     (the same, 32 bit, 4 byte real)
integer, parameter :: sp = kind(1.0)
real(sp) :: aa = 0.
```

Explicit typing

All this works for double precision

```
real*8 (64 bit, 8 byte real, called
“double precision”)
real(8):: b
double precision :: e
real (kind=8):: f
integer, parameter :: dp=kind(1.0d0)
real(dp) :: bb = 0., dd = 0.0_dp
```

Explicit typing
All this works for quadruple precision

```
!gfortran does not seem to support
                       these for quad precision
real*16 :: g (128 bit, 16 byte real, called “quad
precision”)
real(16):: h
real (kind=16):: p
integer, parameter :: quad=kind(1.0q0)
real(kind=quad):: q
integer, parameter::quadk=selected_real_kind(32)
real(kind=quad):: r
```

Where `SELECTED_REAL_KIND(P,R)` returns the kind value of a real data type with decimal precision of at least `P` digits, exponent range of at least `R`

Explicit typing

Numeric choices continued

Similar for complex (but can't go smaller than basic size)

`complex, complex*4` (the same, 32 bit, 4 byte real)

`complex*8` (64 bit, 8 byte real, called "double precision")
`complex*16` (128 bit, 16 byte real, called "quad precision")
Etc.

Explicit typing

For completeness

`character`

`logical`

more on them later

Explicit typing

Recommended programming practice - put

```
implicit none
```

At beginning of all programs

This turns off implicit typing and <u>forces</u> you to declare each and every variable.

Now if you make a typo the compiler will complain and not finish making your executable program file.

Explicit typing

Realistically , everything but variables that are 4 byte integers or 4 byte floating points have to be declared.

This includes other types of variables (bytes, double precision, quad precision, complex, logical, character) and arrays.

So `implicit none` is not that onerous.

Fortran allows upper and lower case, but is not case sensitive and ignores whitespace (spaces, not sure about tabs).

```
PrOgRaM CaSe
impLicit none
InTeGeR i, i I
REAL X
I=1
x=2
ii=3
PRINT *, i, X, I I
i i=4
PRINT *, i, X, iI
```

Hard to read and maintain, but works!

```
540 $ gfortran case.f
541 $ a.out
           1   2.0000000                         3
           1   2.0000000                         4
542 $
```

Arithmetic

Fortran knows about arithmetic on

Integer
Real
Complex

numbers

Arithmetic operations

Add	+
Subtract	-
Multiply	*
Divide	/
Exponentiate	**

Arithmetic operations

Can apply to all types numbers

integers, reals, complex

and mixtures of them.

Integer arithmetic.

```
integer :: i,j,k
real :: x,y
real*16 :: z

i=5
j=2
k=i/j
```

What is k?

Try it and see what you get (write 2nd Fortran program)

Mixed mode arithmetic.

```
integer :: i,j,k
real :: x,y
i=5
j=2
x=7
print *, i/j*x, x*i/j
end
```

Try it and see what you get. (Write 3rd Fortran program)

What's going on?

Mixed mode arithmetic.

Can “force” types in calculations

Called a “cast”

```
z=real(i)/real(j)
```

Does not give same result as

```
z=real(i/j)
```

real, int, dble, cmplx

f90 can do some bizarre (undefined) things when mixing integers, reals, doubles, etc.

http://fortran90.org/src/best-practices.html

http://fortran90.org/src/gotchas.html#floating-point-numbers-gotcha

For now we're going to ignore most of this stuff or it will take a week to write the first program.

Simple read/write from terminal

```
print *, comma separated list of
stuff to print (variables,
constants)
```

```
write(*,*)  comma separated list of stuff to print (variables, constants)
```

```
accpet *,  comma separated list of stuff to print (variables, constants)
```

```
read(*,*)  comma separated list of stuff to print (variables, constants)
```

Simple read/write from terminal

The

"(*,*)"

in the read and write statements mean

First * - standard in or out, other things can go here, leave for now.

Second * - free format, numbers separated by spaces (it is a bit more complicated for character strings – leave for now)

Write a program to prompt the user for the temperature in degrees Fahrenheit and convert to to Celsius.

```
C=(F+40)*5/9-40
```

If you want to go the other way use

```
F=(C+40)*9/5-40
```

Declaring constants (using the options part of the declaration)

```
real, parameter :: pi=4.*atan(1.0)
```

Makes a constant named **`pi`**.

Need the decimal points here on the 4. and 1. or it thinks they are integers and complains

(on an executable line the 4 can be integer [no decimal point], but argument must be real [decimal pont or cast]).

Declaring constants (old way)

```
real pi
parameter (pi=4.*atan(1.0))
```

Control (structured=good)

```
if (logical expression1) then      “Block
- Lines of Fortran”
else if (logical expression2) then
     “Block - Lines of Fortran“
else if (logical expression3) then
     “Block - Lines of Fortran“
else
     “Block - Lines of Fortran“
end if
```

Does stuff in one (and only one) of the blocks. Red – basic `if-then-endif`, orange – optional – more testing `elseif`, or do it if all others fail `else`.

http://en.wikibooks.org/wiki/Fortran/Fortran_control

Control (traditional on left, new on right)

Greater than or less than

```
.GT.  .LT.                    >  <
```

Greater than or equal to or less than or equal to

```
.GE.  .LE.                    >=  <=
```

Equal to and Not equal to

```
.EQ.  .NE.                    ==  /=
```

To check more than one statement, use `.AND.`, `.OR.`, and `.NOT.`:

```
IF ((a .GT. b) .AND.NOT. (a < c))
THEN
```

Group with parenthesis.

Control (structured=good)

Do loops

```
do i=1,10
    print*,i**2
end do

do i=1,10
    isquare = i**2
    if (isquare == 25) cycle
    print*,isquare
end do
```

Top does loop 10 times, bottom skips rest of loop on condition, but keeps looping to end.

Control (structured=good)

Do loops

```
do i=1,10
    isquare = i**2
    if (isquare == 25) exit
    print*,isquare
end do
```

exits loop on some condition (could do with `do while`), does not finish looping.

Control (structured)

Do while loops

```
i=0
do while (resid >= 5.0D-10)
  resid = abs(x(i))
  write (*,*) ' Continue execution’
  i = i+1
end do
```

Control (or lack of it - unstructured)

Numbers in red are "labels" and you can "goto" (jump to) them

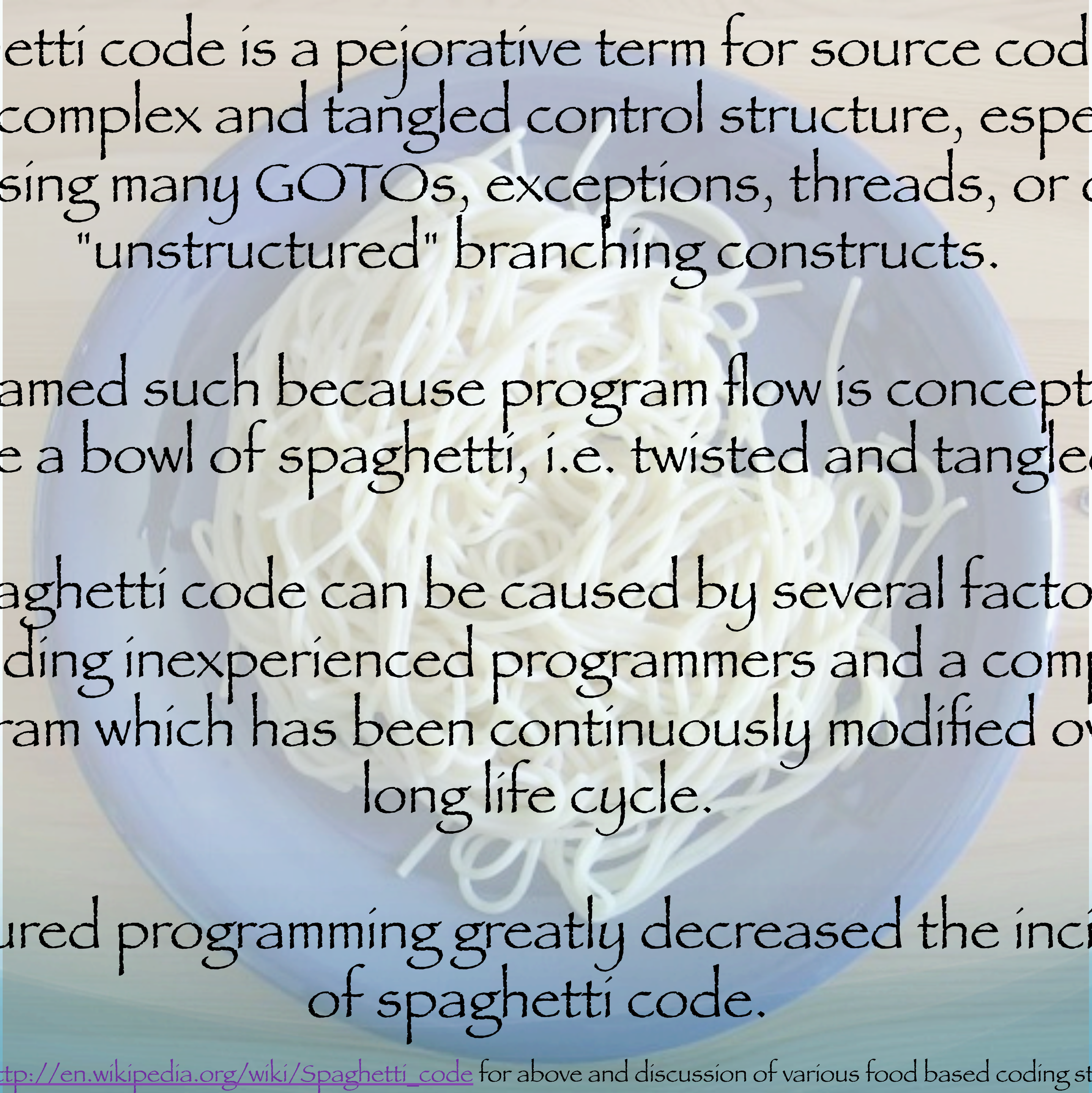
Factorial

```
INTEGER CNT,FACT
CNT=5
FACT=1
1 IF (CNT.EQ.0) GOTO 2
FACT=FACT*CNT
CNT=CNT-1
GOTO 1
2 PRINT*,FACT END
```

“goto”

is anathema in modern computer science and structured programming.

(http://en.wikipedia.org/wiki/Structured_programming) Structured (modular) programming is a programming paradigm aimed on improving the clarity, quality, and development time of a computer program by making extensive use of subroutines, block structures and for and while loops—in contrast to using simple tests and jumps such as the *goto* statement which could lead to "spaghetti code" which is both difficult to follow and to maintain.



Spaghetti code is a pejorative term for source code that has a complex and tangled control structure, especially one using many GOTOs, exceptions, threads, or other "unstructured" branching constructs.

It is named such because program flow is conceptually like a bowl of spaghetti, i.e. twisted and tangled.

Spaghetti code can be caused by several factors, including inexperienced programmers and a complex program which has been continuously modified over a long life cycle.

Structured programming greatly decreased the incidence of spaghetti code.

(see http://en.wikipedia.org/wiki/Spaghetti_code for above and discussion of various food based coding styles)

Control (unstructured=bad)

```
if (logical expression1) goto 10
“Lines of Fortran“
goto 11
10 continue !or executable statement
“Lines of Fortran“
11“Line of Fortran“ !executable statement
“Lines of Fortran“
do 12 i=1,10
“Lines of Fortran“
if (logical expression1) goto 12 “Lines of Fortran“
“Lines of Fortran“
12 continue !non executable statement,
```

could also be executable, instead of enddo.

Control (unstructured=bad)

goto

Most people now pretend it does not exist
(and it does not exist in many languages such as C).

But you will get lots of legacy code with goto's, so you should know what it is
(and it still works, with some restrictions).

Control (unstructured=bad)

```
if (logical expression1) goto 10
"Lines of Fortran"
do 12 i=1,10
"Lines of Fortran"
10 continue !or executable statement
"Lines of Fortran"
12 "Line of Fortran"
```

This is very bad/dangerous, but legal (in older fortran at least). The loop counter is not initialized, so it can be anything.

The **`continue`** statement is just a dummy line to hang a label on. It does not do anything else.

Modern way to do it
Factorial
(obvious it is a `do while` loop)

```
program FactorialProg
integer :: counter = 5
integer :: factorial = 1
do while (counter > 0)
factorial = factorial * counter
counter = counter - 1
end do
print *, factorial
end program FactorialProg
```

Modern way sometimes requires significant gymnastics in design to get around not using goto (when in nested-loops and if -blocks)

```
program doloops
integer i
do i = 1,10
if(i.eq.5) cycle !skips rest of this trip
print *, i       !through outer loop
end do
do i=11,20
if(i==15) exit !quits inner loop, does not
print*,i       !complete looping
end do
end
```

Modern way sometimes requires significant gymnastics in design to get around not using goto (when in nested loops and if blocks)

```
program nestedloops
integer i, j
do i=1,5
print*," i ",i
do j=1,4
print *," i, j ",i,j
if(i==3)cycle
print*," i ",i, " j ",j," i*j ",i*j
if(j==2)cycle
print*," i ",i, " j ",j," i**j ",i**j
enddo
print *
enddo
end
```

```
839 $ a.out
 i            1
 i, j            1           1
 i            1  j            1  i*j            1
 i            1  j            1  i**j            1
 i, j            1           2
 i            1  j            2  i*j            2
 i, j            1           3
 i            1  j            3  i*j            3
 i            1  j            3  i**j            1
 i, j            1           4
 i            1  j            4  i*j            4
 i            1  j            4  i**j            1

 i            2
 i, j            2           1
 i            2  j            1  i*j            2
 i            2  j            1  i**j            2
 i, j            2           2
 i            2  j            2  i*j            4
 i, j            2           3
 i            2  j            3  i*j            6
 i            2  j            3  i**j            8
 i, j            2           4
 i            2  j            4  i*j            8
 i            2  j            4  i**j           16

 i            3
 i, j            3           1
 i, j            3           2
 i, j            3           3
 i, j            3           4

 i            4
 i, j            4           1
 i            4  j            1  i*j            4
 i            4  j            1  i**j            4
 i, j            4           2
 i            4  j            2  i*j            8
 i, j            4           3
 i            4  j            3  i*j           12
 i            4  j            3  i**j           64
 i, j            4           4
 i            4  j            4  i*j           16
 i            4  j            4  i**j          256

 i            5
 i, j            5           1
 i            5  j            1  i*j            5
 i            5  j            1  i**j            5
 i, j            5           2
 i            5  j            2  i*j           10
 i, j            5           3
 i            5  j            3  i*j           15
 i            5  j            3  i**j          125
 i, j            5           4
 i            5  j            4  i*j           20
 i            5  j            4  i**j          625
```