

Lecture 02 – Programming Languages

Prof. Brendan Kochunas

NERS/ENGR 570 - Methods and Practice of Scientific Computing (F22)



COLLEGE OF ENGINEERING

NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES

UNIVERSITY OF MICHIGAN

Outline

- Motivation
- Types of programming languages
- Fortran
- Hands on exercises
- C and C++ (Time permitting)

Learning Objectives: By the end of Today's Lecture you should be able to

- *(Knowledge)* Differentiate between types of programming languages and their features/uses.
- *(Skill)* Access the CAEN Linux environment remotely.
- *(Skill)* Write simple “hello world” programs in Fortran, C, and C++ with VIM.
- *(Skill)* Declare variables that are matrices, and assign or edit their values in Fortran, C, and C++

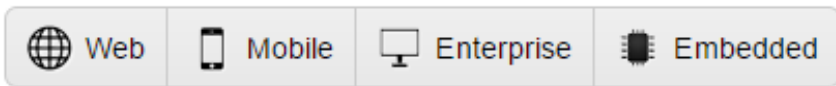


Motivation for Fortran and C/C++

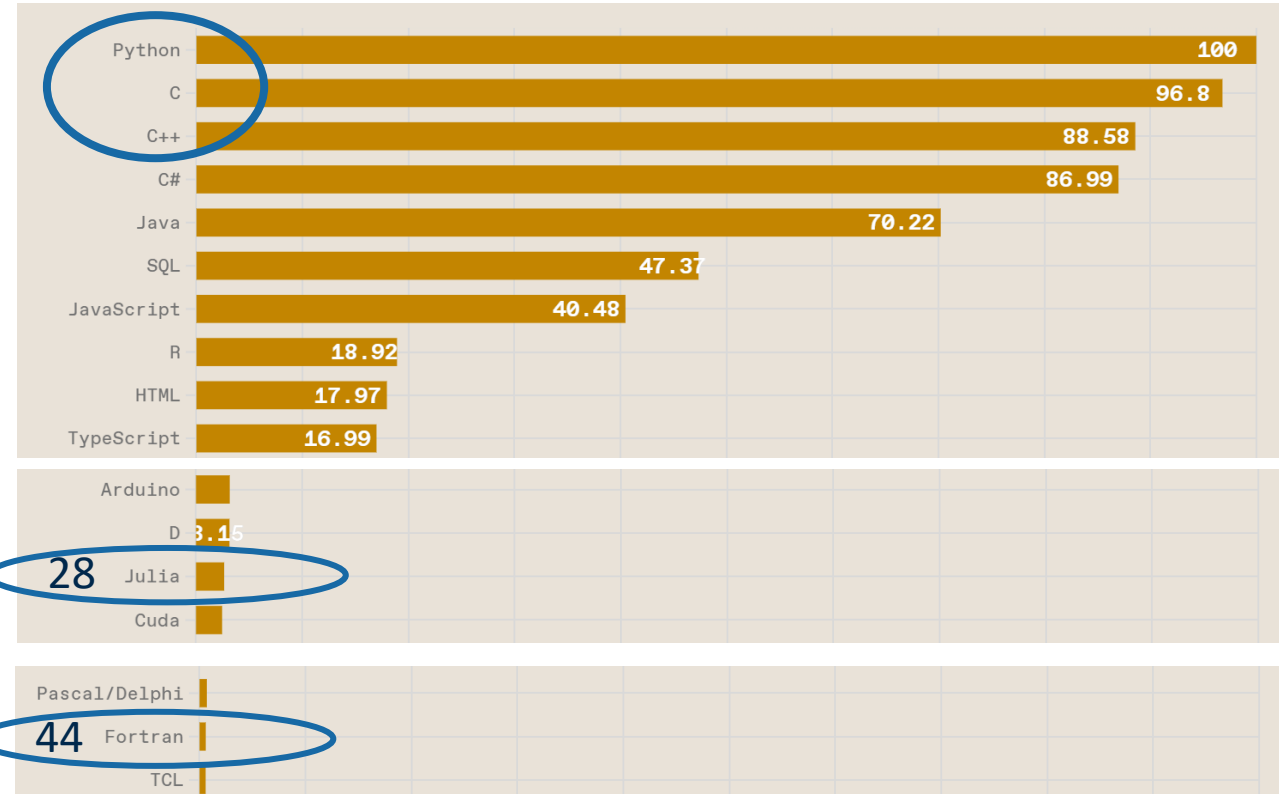
What are people using?

- IEEE Top programming languages
 - <https://spectrum.ieee.org/top-programming-languages>
 - Many other lists & metrics (TIOBE, PYPL ranking, etc.)

Language Types (click to hide)



- Web – Languages used for developing websites and applications
- Mobile – Languages used for applications on mobile devices
- Enterprise – Languages used for enterprise, desktop, & scientific applications
- Embedded – Languages used to program device controllers


















Most Popular Languages 2021

Rank	Language	Type	Score
1	Python	Web, Desktop, Embedded	100.0
2	Java	Web, Mobile, Desktop	95.4
3	C	Mobile, Desktop, Embedded	94.7
4	C++	Mobile, Desktop, Embedded	92.4
5	JavaScript	Web	88.1
6	C#	Web, Mobile, Desktop, Embedded	82.4
7	R	Desktop	81.7




















23	Shell	Desktop	54.5
24	Processing	Web, Desktop	50.6
25	Fortran	Desktop	45.2
26	Objective-C	Mobile	44.4
27	Lua	Web, Desktop	43.3
28	Cuda	Desktop	41.3







Most Popular Languages 2020

Rank	Language	Type	Score
1	Python ▼	  	100.0
2	Java ▼	  	95.3
3	C ▼	  	94.6
4	C++ ▼	  	87.0
5	JavaScript ▼		79.5
6	R ▼		78.6
7	Arduino ▼		73.2
























21	Shell ▼		52.0
22	Processing ▼	 	49.2
23	C# ▼	   	48.1
24	SAS ▼		45.2
25	Fortran ▼		43.0
26	Cuda ▼		41.0








Most Popular Programming Languages 2019

Rank	Language	Type	Score
1	Python	  	100.0
2	Java	  	96.3
3	C	  	94.4
4	C++	  	87.5
5	R		81.5
6	JavaScript		79.4
7	C#	   	74.5
8	Matlab		70.6

31	Cuda		37.3
32	VHDL		36.0
33	Verilog		33.4
34	ABAP		33.0
35	Delphi	  	30.1
36	Fortran		29.4

Most Popular Programming Languages 2018

Language Rank	Types	Spectrum Ranking
1. Python	  	100.0
2. C++	  	99.7
3. Java	  	97.5
4. C	  	96.7
5. C#	  	89.4
6. PHP		84.9
7. R		82.9
8. JavaScript	 	82.6
9. Go	 	76.4
10. Assembly		74.1
11. Matlab		72.8

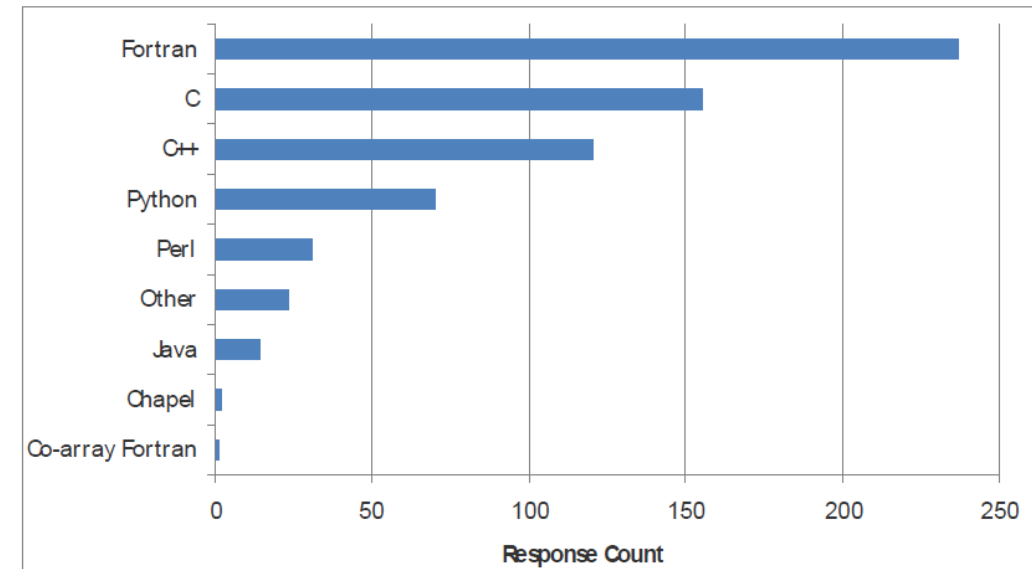
22. Fortran		49.5
23. SQL		49.3
24. Haskell	 	48.6
25. VHDL		45.4
26. Visual Basic		45.1
27. Cuda		43.0

Programming Languages in HPC

- Which programming models and languages do you use for code development?
 - Bull, M. Guo, X. Ioannis Liabotis, I. (Feb. 2011) Applications and user requirements for Tier-0 systems, PRACE Consortium
 - Stitt, T. and T. Robinson (2008) A Survey on Training and Education Needs for Petascale Computing, PRACE Consortium Partners (<https://prace-ri.eu/wp-content/uploads/PP-D3.3.1.pdf>)

6. Please indicate the importance of the following programming languages for your HPC code development:

	Not important	Somewhat important	Very important	Rating Average	Response Count
FORTRAN77	24.7% (22)	38.2% (34)	37.1% (33)	1.12	89
Fortran 95	12.9% (12)	20.4% (19)	66.7% (62)	1.54	93
Fortran 2003	50.0% (42)	32.1% (27)	17.9% (15)	0.68	84
C	27.0% (24)	36.0% (32)	37.1% (33)	1.10	89
C++	40.7% (35)	30.2% (26)	29.1% (25)	0.88	86
Java	87.3% (69)	11.4% (9)	1.3% (1)	0.14	79
Scripting Languages (Python, PERL, Ruby etc.)	27.3% (24)	45.5% (40)	27.3% (24)	1.00	88



Programming language needs continue to evolve

- “Simulation in the climate community is dominated by the nexus of the Fortran/C/C++ languages and the OpenMP/OpenACC programming models. While both Fortran and C/C++ are interoperable on almost all computing platforms available today, there is a strong desire for other productivity oriented scripting languages in a distributed environment (e.g., Python/pyMPI), as well as the desire to move away from the flat MPI model, which may be functional on the next generation of systems but will lose a factor of 10 to 50 times on GPU-accelerated systems.”
- “The legacy inertia behind Fortran, coupled with perceptions that Fortran compilers deliver superior performance, ensures its continued use. Conversely, the broad computing community’s use of C++ coupled with its access to novel and emerging architectures and new language constructs has resulted in a gradual shift to C++ as the basis for simulation”

Gerber, R., et. al, “Crosscut report: Exascale Requirements Reviews,”
U.S. DOE Office of Science, Tysons Corner, Virginia, March 9–10, (2017). doi:10.2172/1417653



Programming Languages

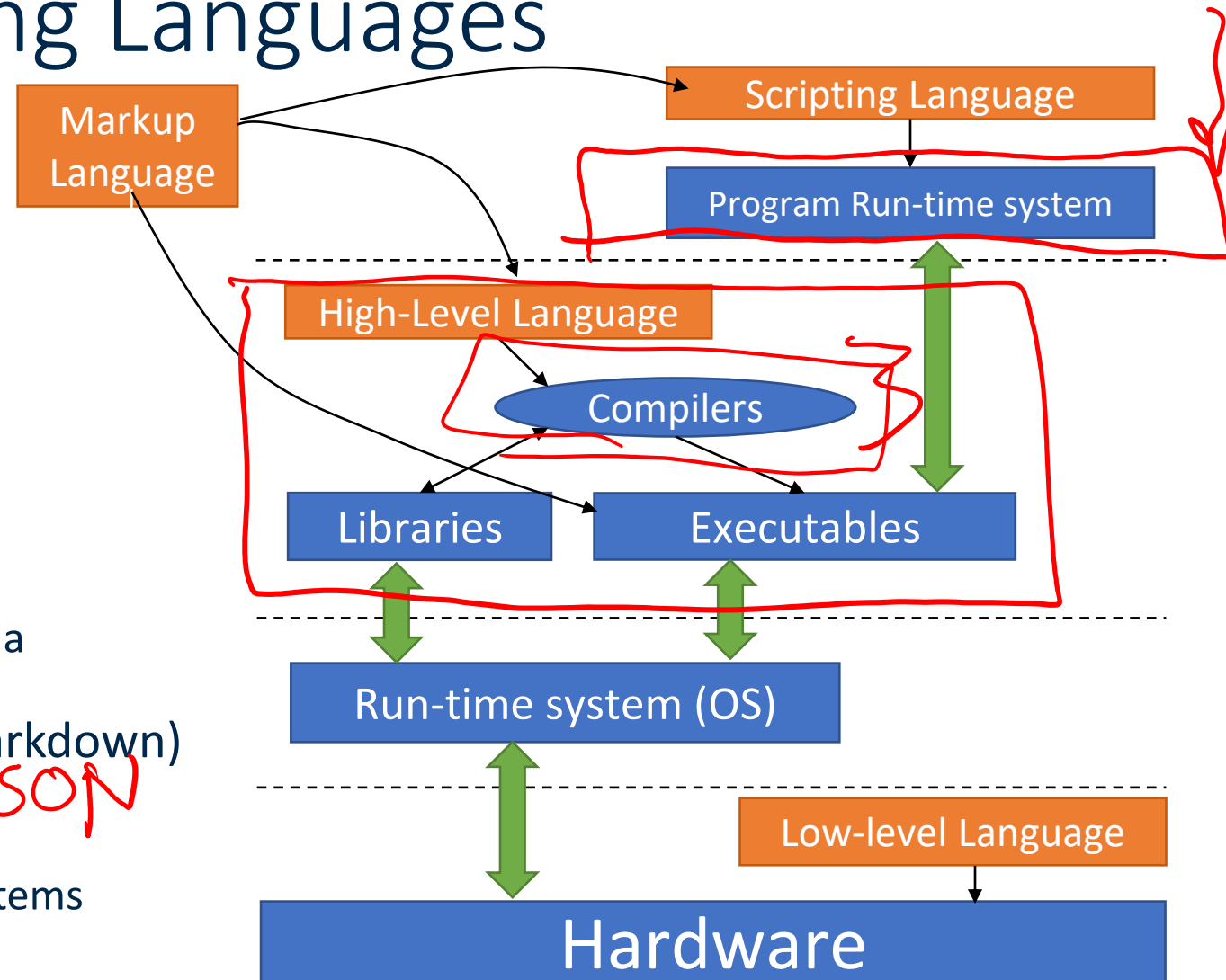
...generally speaking

What defines a programming language?

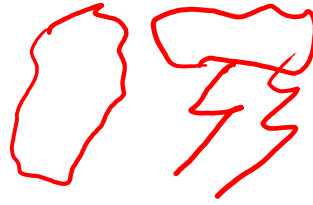
- Primarily: A formal standard
 - C/C++: C89, C99, C11 and C++98, C++03, C++11, C++14, C++17
 - Fortran: ISO/IEC standard 1539
 - Java
 - Python
- Syntax: e.g. the “grammar”
- Semantics: e.g. the “meaning” which is “vocabulary + grammar”
- Programming languages should define performance of execution
 - Typically syntax/semantics are carefully defined, but not so for performance model.

Types of Programming Languages

- Low Level language (Assembly)
 - Defined by hardware (less portable)
- High Level language (C, C++, Fortran)
 - Defined by run-time system (e.g. Operating System)
 - Portable, depends on compilers
- Scripting language (MATLAB, Python, Bash) *Java*
 - Defined by portable run-time system of a program
- Markup language (e.g. XML, YAML, Markdown) *JSON*
 - Used for annotation
 - Data transfer
 - Input to multiple types of programs/systems



Things a programming language should do



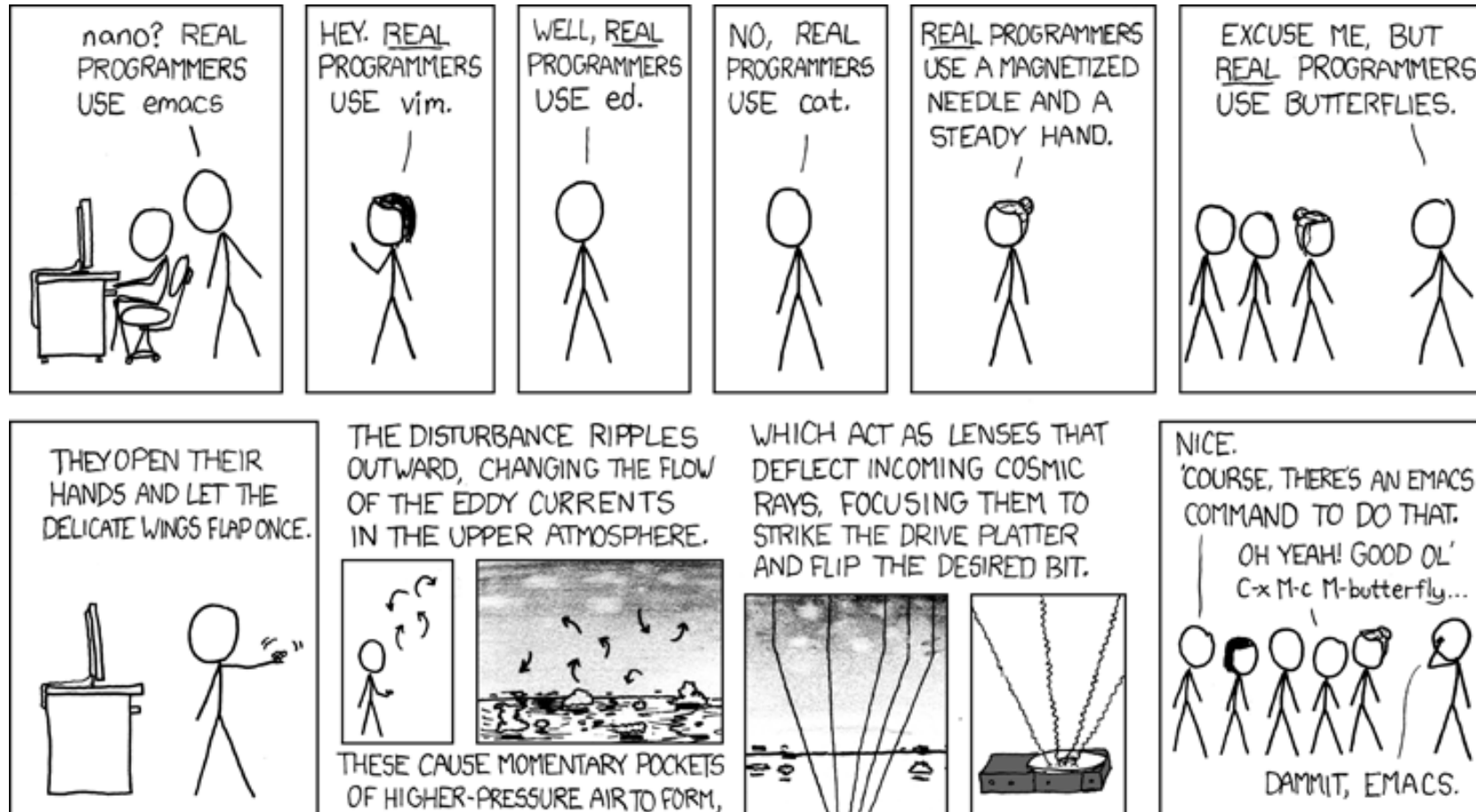
- Variable declaration
 - Scalars, arrays, pointers
 - Intrinsic types, programmer defined types
- Operators
 - Arithmetic: +, -, *, /
 - Relational: <, >, ==
 - Logical: and, or, not, xor
 - Others: e.g. exponentiation
- Execution control constructs
 - Branching constructs
 - e.g. if/else, switch (or case)
 - Looping constructs
 - For or Do
 - While
 - Goto
- Memory management
 - Automatic
 - Or provide programmer with intrinsics

Prepare for Hands on!

1. Connect to the CAEN Linux Environment
2. Open a terminal

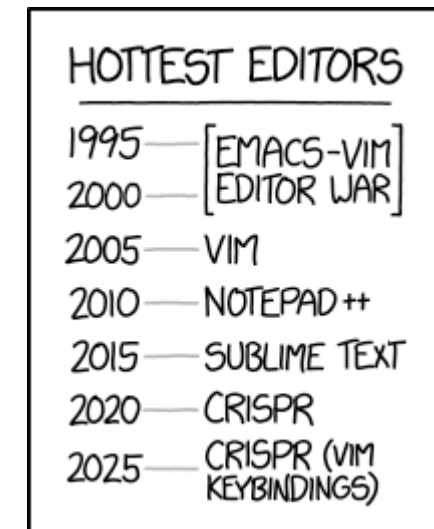
Which editor should I use?

<https://xkcd.com/378/>



Familiarize yourself with a command line editor

- Choices: vi, vim, nano, emacs, pico
- We recommend vi or vim
 - vi is *guaranteed* to be on all POSIX compliant Linux platforms
 - vim (VI Improved) is a little “nicer” to use
 - We want to be “real programmers” to impress people
- Getting training on text editors
 - vi or vim
 - Interactive Tutorial: <http://www.openvim.com/>
 - <http://vim-adventures.com/>
 - from command line: vimtutor
 - Cheat sheet: <http://www.viemu.com/vi-vim-cheat-sheet.gif>
 - nano or pico
 - <https://www.lifewire.com/beginners-guide-to-nano-editor-3859002>
 - <http://www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor/>



VS CODE
atom

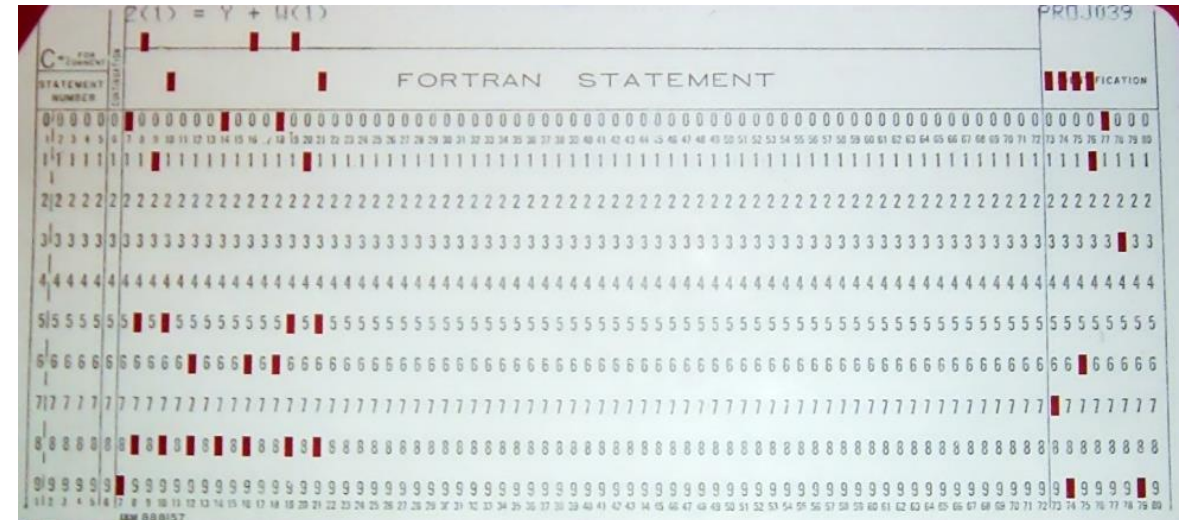
<https://xkcd.com/1823/>



Fortran

A Brief History of Fortran

- FORTRAN 66 (1966) – First “standardized” Fortran
- FORTRAN 77 (1977) – What most people think of when they hear “Fortran”
- Fortran 90/95 – Introduced modules, derived types, and dynamic memory allocation.
 - Basically caught up to features in C.
- Fortran 2003/2008 – Introduced extensive object-oriented capabilities
 - Basically caught up to C++ (although not quite).
 - Still not fully supported by compilers
- Fortran 2003 and later are considered “Modern Fortran”.



Fortran 77

- Fixed Format and case insensitive
 - Different columns have different rules
 - 1: Blank or comment symbol ("c" or "*")
 - 1-5: Statement Label
 - 6: Continuation of previous line
 - 7-72: Programming statements
 - 73-80: Sequence number
 - Not used to today, was used by punch card sorters in case you dropped your box of cards
 - ***NO DYNAMIC MEMORY ALLOCATION***
- Compiler support is very good
 - also very good at producing very fast code
- Missing modern programming language features
 - Detrimental to reuse
 - Certain things cannot be done (e.g. dynamic memory allocation)

```
c0          1          2          3
c2345678901234567890123456789012 ...
      program name

      implicit none

      double precision  a,b,c(100)

      a=1.0d0
      b=1.0d0

      write(*,500)
+  a+b

      goto 555

500      format (F10.4)
555      continue
      end
```

Fortran 90/95

- F90 Standard: ISO/IEC standard 1539:1991
- F95 Standard: ISO/IEC 1539-1:1997
- Free Form
 - Forget all the rules about columns
- Dynamic memory allocation
- Intrinsic support for array operations
 - (e.g. `a(:)=b(:)`)
- Pointers: references to memory
- Modules: reusable components of program
- Derived data types: custom data types
- Removed some problematic features of F77

```
module linear_system
  type :: linsys_t
    real,pointer :: a(:, :), b(:, :), x(:, : )
  end type

  integer :: n
end module

program main
  use linear_system
  implicit none

  real,pointer :: sol(:)
  type(linsys_t) :: ls1, ls2

  read(*,*) n
  allocate(ls1%a(n,n))
  allocate(ls1%b(n))
  allocate(sol(n))
  ls1%x => sol
  ! ...
```

Fortran 2003 & 2008

Fortran 2003: ISO/IEC 1539-1:2004

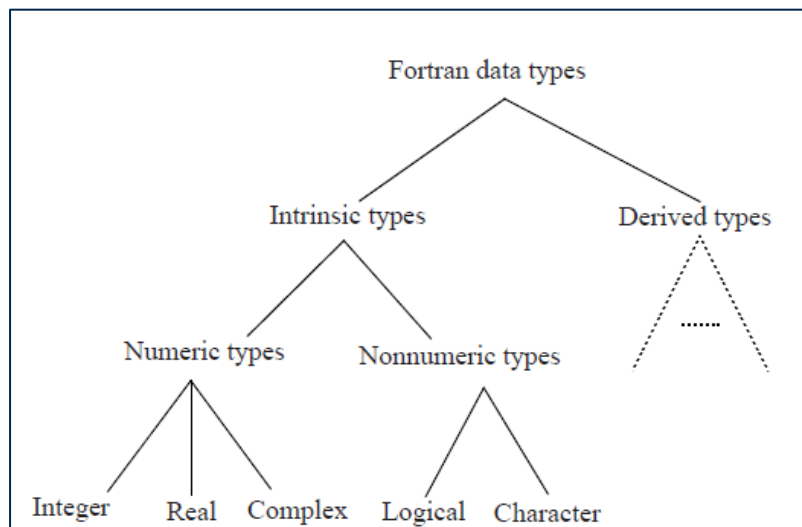
- Significant addition to F90/95
 - F90/95 programs are F2003 compliant
- Defined Object Oriented Features
 - Polymorphism and inheritance
 - Type-bound procedures (methods)
- Other
 - Procedure pointers
 - Standardized C-interoperability
 - IEEE floating point arithmetic
 - Enhanced intrinsics for command line processing

Fortran 2008: ISO/IEC 1539-1:2010

- Primarily added language features for enhanced parallelism
- Added submodules
 - Entity facilitating program structure
- Coarray Fortran
 - Simple extension for distributed parallel programming
 - Typically relies on underlying MPI implementation
- DO CONCURRENT
 - Loop iterations have no interdependencies

Data Types

Intrinsic Data Types	
Keyword	kinds
CHARACTER	ASCII, ISO_10646
LOGICAL	8-bit, 16-bit, 32-bit , 64-bit
INTEGER	8-bit, 16-bit, 32-bit , 64-bit
REAL	32-bit , 64-bit, 128-bit
COMPLEX	32-bit , 64-bit, 128-bit



- Values of data type “kinds” are not defined by standard and not available on all systems.
 - Therefore, these can be compiler dependent.
 - Careful you don’t rely on extensions.
 - Typically represented in “bytes” as integer
 - e.g. 32-bit = 4 bytes
- Problem: How to guarantee portability?
 - Intrinsic procedures
 - SELECTED_REAL_KIND (F95 and later)
 - SELECTED_INT_KIND (F95 and later)
 - SELECTED_CHAR_KIND (F2003 and later)
 - ISO_FORTRAN_ENV intrinsic module
 - Module is part of Fortran 2003 standard
 - Kind parameters defined in 2008 standard
- Inherent support for multi-dimensional arrays (up to 7-dimensions)
REAL :: a (:, :, :, :)

Declaration & Attributes

- Lots of redundancy in syntax
 - So, pick the way that you like
 - One suggestion: pick the approach that involves the least typing.

```

real (kind=4) , dimension (20) :: a
real (4) :: a(20) !same as above

real (8) , allocatable :: b(:, :)
real (8) , pointer :: c(:, :) => NULL()
    
```

Array properties	DIMENSION
Allocatable property	ALLOCATABLE
Pointer properties	POINTER TARGET
Value definition properties	DATA PARAMETER SAVE ASYNCHRONOUS VOLATILE
Module entity properties	PUBLIC PRIVATE PROTECTED BIND
Dummy argument properties	INTENT OPTIONAL VALUE
Procedure properties	EXTERNAL INTRINSIC

Operators & Special Characters

Assignment Operators	
=	assignment
=>	pointer assignment

Other	
Symbol(s)	Meaning
//	Concatenate strings
&	Line continuation, placed at end of line (F90 and later)
!	Comment
;	End of statement
%	Accesses component of derived type
(i)	Array index
(/1.0/)	Array literal
[i]	Coarray index

Relational Operators		
Fortran 77	Fortran 90 and later	Meaning
.EQ.	==	Equal to
.NE.	/=	Not equal to
.GT.	>	Greater than
.GE.	>=	Greater than or equal to
.LT.	<	Less than
.LE.	<=	Less than or equal to

Arithmetic Operators		Logical Operators	
+	addition	.AND.	and
-	subtraction	.OR.	or
*	multiplication	.NOT.	not
/	division	.EQV.	equivalent
**	exponentiation	.NEQV.	not equivalent

Program Entity Structures

Program

```
program name  
  !declarations  
  !statements  
contains  
  !internal procedures  
end program
```

Module

```
module name  
  !declarations  
contains  
  !internal procedures  
end module
```

Subroutine

```
subroutine name(args)  
  !declarations  
  !statements  
end subroutine
```

Function

```
function name(args) result(r)  
  !declarations  
  !statements  
end function
```

Submodules

```
module m  
  !declarations  
  
  interface  
    !submodule routines  
  end interface  
  
  contains  
    !internal procedures  
end module  
  
  submodule (m) sm  
    !declarations  
  contains  
    !internal procedures  
end submodule
```

Execution Control Constructs

Branching

```
!Fortran 2008 and later
SELECT TYPE(someType)
  CLASS IS(baseType)
  TYPE IS(extendendType)
  CLASS DEFAULT
END SELECT
```

```
IF(<cond_expr>) <statement>

IF(<cond_expr1>) THEN
ELSE IF(<cond_expr1>) THEN
ELSE
END IF

SELECT CASE(variable)
  CASE(val1,val2)
  CASE(val3)
  CASE DEFAULT
END SELECT
```

```
!Fortran 95 and later
WHERE (A > 0)
  B=A
ELSEWHERE (A < 0)
  B=0
ELSEWHERE !A == 0
  B=1
ENDWHERE
```

Loops

```
DO i=1,10
END DO
```

```
DO i=10,1,-1
END DO
```

```
i=1
DO WHILE(i < 10)
  i=i+1
END DO
```

```
DO WHILE(.TRUE.)
  IF(i < 10) CYCLE !Skip rest of loop
  BREAK !Exit loop
END DO
```

```
!Fortran 95 and later
FORALL(i=1:3, j=1:3, i < j)
  a(i,j)= !stuff
END FORALL
```

```
!Fortran 2008 and later
DO CONCURRENT i=1,10
  a(i)=b(i)*c(i)
END DO
```

Fortran I/O

- I/O based on ideas of “UNITS”
 - Unit is an integer tied to a file *or some other device* (e.g. the screen)
- Types of intrinsic files

	Formatted	Unformatted
Sequential	Text file	Binary
Direct	N/A	Binary (fixed record size)

• READ/WRITE

```

WRITE(*,*) "HELLO WORLD" !Standard Out
WRITE(ERROR_UNIT,*) "HELLO WORLD" !Standard Error
!(ERROR_UNIT from ISO_FORTRAN_ENV, typically 0)

!Formatted write to variable string
WRITE(string,'(f10.4)') 1.0

!Write to file that was previously opened on unit 35
WRITE(35,*) "Hey unit 35!"
    
```

Fortran I/O Statement	Purpose
OPEN	Opens a file
CLOSE	Closes a file
INQUIRE	Make inquiries about a file
READ	Read from a unit
WRITE	Write to a unit
FLUSH	For buffered I/O this clears the buffer
BACKSPACE	Move the position in a sequential file back one record
REWIND	Move the position in a sequential file back to the first record

- I/O Statements have very long “specifier lists”
- For OPEN statement
 - UNIT, FILE, STATUS, IOSTAT, ACCESS, FORM, ACTION, RECL, POSITION, DELIM, PAD

Fortran 2003 & 2008 Examples

Inheritance (2003)

```
type :: geom_t
  integer :: dim
end type

type, extends(geom_t) :: point_t
  real, allocatable :: coord(:)
end type
```

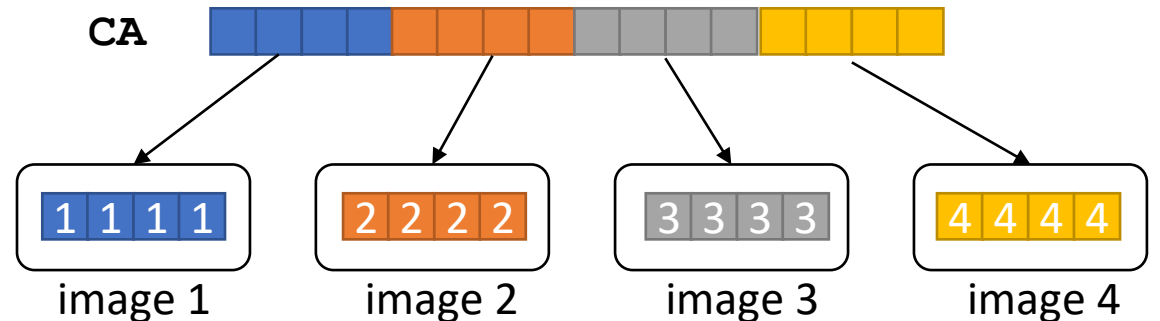
```
type(geom_t) :: g
type(point_t) :: p

!....
write(*,*) g%dim
write(*,*) p%dim
write(*,*) p%coord
```

Coarray (2008)

```
! Local access with ()
! Remote access with []
integer :: ca(4)[*]

ca(:)[this_image()]=this_image()
do image = 1,num_images()
  write(*,*) ca(:)[image]
end do
```



Fortran Compilers

List of Compiler Vendors

- Free and Open Source
 - **gfortran** GNU
 - g95
 - EKOPath
 - llvm (Dragonegg)
 - OpenUH, Open64, Open Watcom
- Commercial
 - Absoft
 - Cray
 - IBM XL
 - **ifort** Intel
 - Lahey
 - Numerical Algorithms Group (NAG)
 - pgf90 Portland Group (PGI)
 - Pathscale
 - Oracle

Compilers on Flux

- GNU
- Intel
- PGI
- NAG (special license required)
- Lahey (requires NAG)

Further Reading

Online

- General information (with good examples)
 - <http://fortranwiki.org>
- [GNU Compiler Documentation](#)
- [IBM Compiler Documentation \(Language Reference\)](#)

Books

- “Fortran 90/95 for Scientists and Engineers,” S. J. Chapman
- “Fortran 2003 Handbook,” J. Adams, W. Brainerd, R. Henderson, R. Maine, J. Martin, B. Smith
 - Available electronically through www.lib.umich.edu (when on campus)
- “Scientific Software Design: The Object-Oriented Way”, Damian Rouson, Jim Xia, and Xiaofeng Xu
- “Modern Fortran Explained”, Michael Metcalf, John Reid, Malcolm Cohen



C and C++

History of C/C++

- C - started in 1969 by Dennis Ritchie at Bell Labs
 - Concurrent with and related to the development of Unix operating system (also at Bell Labs)
 - Most widely used programming language – available across virtually all platforms and architectures
 - Informal specification “K&R” (Kernighan and Ritchie, the developers) published in 1978
 - ANSI standard since 1989
 - General purpose
 - Relatively fast
 - Current standard is C18
- C++ - also developed at Bell Labs, starting in 1979
 - Standardized in 1998 (International Organization for Standardization)
 - Added features/extensions that make development and management of large software projects much simpler (mainly classes)
 - Newer versions have more advanced functionality (templates, namespaces, abstract classes, and more)
 - Widely used for scientific computing purposes because of powerful Object-Oriented programming capabilities

C and C++ standards

C Standards

- C89
 - original ANSI standard (also C90)
 - still supported by compilers
- C99
 - new features include
 - function inlining,
 - `complex` type,
 - one-line comments with `"/ /"`,
 - variable-length arrays, and more
- C11
 - atomic operations,
 - multi-threading,
 - bounds-checked functions,
 - static assertions
- C17 / C18
 - Revision to C11
- C23
 - In development

C++ Standards

- C++98
 - First standard, introduces object-oriented features to C
 - Abstraction, Encapsulation, Inheritance, Polymorphism
 - Multiple inheritance
- C++03
 - Mostly fixed issues with previous standard to improve compiler consistency
- C++11
 - Multi-threading, initializer lists for all classes, type inference (`auto`)
- C++14
 - Variable templates, deprecated (for multi-threading), bug fixes, other
- C++17
 - Attribute namespace, inline variables, fold expression, class template argument deduction
- C++20
 - Feature test macros, 3-way comparison, co-routines, modules, updated STL
- C++23
 - In development, already partial support in some compilers

Data Types

- Intrinsic – basic data types that are defined and recognized by standard C
 - Also available as multi-dimensional arrays
- Derived – library or user-defined data type that is made up of some combination of data with intrinsic types
 - Standard library: vector (of any data type), string
 - User-defined: a struct or class that contains several variables of any combination of intrinsic and/or derived types

Intrinsic Data Types	
Keyword	Details
<code>unsigned char</code>	8-bit integer 0 to 255
<code>signed char</code>	8-bit integer -127 to 128
<code>unsigned int</code>	short (16-bit), long (32-bit or 64-bit), and long long (64-bit)
<code>signed int</code>	
<code>float</code>	32-bit
<code>double</code>	64-bit
<code>complex (C99)</code>	complex float and complex double
<code>bool</code>	True or False
<code>enum</code>	enumeration for a set

Definable (Derived) Data Types	
Keyword	Details
<code>union</code>	Define several variables to share same memory space
<code>struct</code>	Defined in terms of intrinsic data types (or other structs)

C/C++ Variable Declaration

- In C/C++, legal variable declarations can be made at any point within a code
 - Unlike Fortran, where all type declarations are made at the top of a routine or program
 - For C90, must be immediately after {
- All variables must be declared as a certain type (and should be initialized) before being used in some expression
- Variables remain defined throughout the scope in which they were declared. When a variable “falls out” of scope, its data is no longer accessible
- Scope is defined in C++ by opening and closing curly braces {}

```
float variable;  
int variable = value;
```

C Operators & Special Characters

Logical Operators	
&&	and
	or
!	not

Arithmetic Operators	
+	addition
-	subtraction
*	multiplication
/	division
%	modulus
=	assignment

Other	
Symbol(s)	Meaning
// or /* */	Comment and comment block
;	End of statement
{ }	Scope declaration
? :	Ternary operator (like a compacted if/else)
[]	Array declaration
*	Dereference variable (Unary)
&	Return memory address of variable (Unary)
-> or .	Member selection (-> is for pointers, . is for non-pointers)

Relational Operators	
==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Augmented Assignment	
++	a=a+1
--	a=a-1
+=	a=a+b
-=	a=a-b
*=	a=a*b
/=	a=a/b

Bitwise Operators	
&	Bitwise AND
	Bitwise OR
^	Exclusive OR
~	Toggle (flip) bits
>>	Left Shift
<<	Right Shift

Augmented assignment operators also available for bitwise operations

Program Entity Structure

Preprocessor statements (Header file declarations)

Global Declarations

In C++ Class declarations go here
(will cover examples in more
detail with OO programming)

Main program (only define once in program)

Functions

Other functions (optional)

```
/* Preprocessor commands
   and include statements */
#include <stdio.h>

//Global variables and interface definitions
int n;
void sub1(int* n);

/* Functions/Subroutines
   "main" is a special procedure
   in C/C++ corresponding to the
   "main program" */
int main()
{
    sub1(&n);
    return n;
}

/* "void" means return nothing, like a Fortran
   subroutine */
void sub1(int *n)
{
    *n=0;
}
```

C example

Header Files

- Header files are used to define variables and interfaces
 - Generally each source file has a corresponding header file, especially in C++
- The header data is appended to the top of the source code files by the compiler when an `#include "header.h"` statement is present.
- Header files allow compilers to “resolve dependencies” in your program.
- In C++ header files:
 - Variables are declared not only as a specific type but also public or private
 - Public: any function has access to read, use, and change this data
 - Private: only the object that owns the data can manipulate it
 - Methods of a class are defined by name, type of returned data, and types of all of the input data
 - GOOD practice: “namespace” entities in your header files.

C++ example header

```
namespace ners570
{
    class timer {
    public:
        tic();
        toc();
    private:
        double startTime_;
        double totalTime_;
    }
```

Usage

```
#include "ners570_timer.hpp"

//Declare variable "t" as a ners570
//timer, other headers might define
//classes named timer
ners590::timer t;
```


Execution Control Constructs

- IF/ELSE

- Followed by curly braces { }, executes code within braces depending on result of conditional checks
- `if (condition) { }`

- Switch

- A logical branch, achieves the same logical result as IF/ELSE with much simpler syntax in some cases, and a different sequence of logical checks
 - `switch(variable) :`
 - `case(value1)`
 - `case(value2)`

- `for(initialization, condition, step) { }`

- Repeat code and execute step (usually increment/decrement) after each loop until the condition is no longer met

C and C++ I/O

C

- Routines are defined in “standard library”, accessible through header files.
 - Primary library: `<stdio.h>`
- Some useful routines:
 - `printf, fprintf`
 - `fopen, fclose, fget, fscanf`

C++

- C++ I/O libraries: `<fstream>`, `<iostream>`
- `cout << "Hello, world!" << endl;`
 - Hello, world!
- `cout` outputs to screen, is in standard namespace (e.g. equivalent to `std::cout`, “`std::`” is implied)
- `cin <<` waits for input from the keyboard

Standard Library

- Different for C and C++
- ***A HUGE advantage over Fortran***
- Contains many useful types and functions
 - Example: `Vector`
 - a dynamic data container class with many methods for accessing more effectively than a standard array
 - `.push_back()`, `.pop_back()`, `.at()`, `.size()`
 - Dynamic memory management (`malloc`, `dmalloc`, `free`)
 - Random number generator (`rand`, `srand`, `RAND_MAX`)
 - `abs()` – absolute value
- Many other useful C libraries
 - `cstring`, `cmath`, `cstdio`

Angled brackets imply header is supplied by system or compiler

```
#include <vector>
#include "ners570_timer.hpp"
```

Quotes imply header file is source file defined by programmer

Further Reading

C

- [The Language](#)
- [Standard Library](#)

C++

- [The Language](#)
- [Standard Library](#)



Some more advanced concepts

Reference Counted Pointers (RCP)

- Also known as a smart pointer.
 - Automates “garbage cleanup” e.g. frees unused memory
- Keeps track of (e.g. counts) how many pointers are referencing a particular block of memory
 - When no more pointers are referencing the block of memory, it can be automatically freed.
 - Useful machinery for helping to prevent memory leaks in C++
- Part of C++11 Standard: `std::shared_ptr`
- Other implementations available from third party libraries: boost, Trilinos

Templating (C++)

- **Very powerful** feature in C++
 - Idea is like meta-programming.
- Write a program for a “template” type.
- Compiler generates machine code for all data types matching the template.

```
class calc
{
public:
    int multiply(int x, int y);
    int add(int x, int y);
};
int calc::multiply(int x, int y) { return x*y; }
int calc::add(int x, int y) { return x+y; }
```

```
template <class A_t> class calc
{
public:
    A_t multiply(A_t x, A_t y);
    A_t add(A_t x, A_t y);
};

template <class A_t> A_Type calc<A_t>::multiply(A_t x, A_t y)
{
    return x*y;
}

template <class A_t> A_t calc<A_t>::add(A_t x, A_t y)
{
    return x+y;
}
```

Creating a Template means you have written this routine once for integers, floats, doubles, bools, and any classes of the templated type!

Example from: <http://www.cprogramming.com/tutorial/templates.html>

Working with Fortran and C/C++ together

Interoperable Intrinsic Datatypes (Fortran 2003 and later)

Description	Fortran type declaration	C type declaration
Character	<code>CHARACTER (LEN=1, KIND=C_CHAR)</code>	<code>char</code>
True/False	<code>LOGICAL (C_BOOL)</code>	<code>_Bool</code>
default integer	<code>INTEGER (C_INT)</code>	<code>int</code>
floating point (32-bit)	<code>REAL (C_FLOAT)</code>	<code>float</code>
double precision (64-bit)	<code>REAL (C_DOUBLE)</code>	<code>double</code>
Integer 8-bit	<code>INTEGER (C_INT8_T)</code>	<code>int8_t</code>
Integer 16-bit	<code>INTEGER (C_INT16_T)</code>	<code>int16_t</code>
Integer 32-bit	<code>INTEGER (C_INT32_T)</code>	<code>int32_t</code>
Integer 64-bit	<code>INTEGER (C_INT64_T)</code>	<code>int64_t</code>
Long integer	<code>INTEGER (C_LONG)</code>	<code>long int</code>
Long long integer	<code>INTEGER (C_LONG)</code>	<code>long long int</code>

See documentation for ISO_C_BINDING module for complete listing.

Modules and Headers

- Fortran modules are like compiler generated header files
 - Makes Fortran a little bit more tricky to compile because modules must be compiled in the correct order to resolve dependencies.
 - Makes C/C++ a little more cumbersome because you are always having to maintain header files and source files.
- Header files are portable.
- *Compiled* module files are not.
- Fortran can also include “header files”, but the `INCLUDE` keyword in Fortran implies a direct insertion (e.g. copy-paste) of the contents of an `INCLUDED` file.
 - Contents must be valid Fortran code
 - Typically used for defining types or global variables, sort of deprecated as a bad programming practice.

```
program main
  use module1
  use module2

  include 'file1.h'

  !Rest of program

end program
```

Modules “module1” and “module2” must have been compiled prior to compiling the program.

Other Things to keep in mind

Item	Fortran	C	C++
Case Sensitive variable names	No	Yes	Yes
Statements end with “;”	Optional	Required	Required
Assumed Starting Index	1	0	0
Multi-dimensional array ordering in memory	In-out	Out-in	Out-in
Strings	Fixed length	End with null character	

Memory layout of multi-dimensional arrays

Fortran

	$i \longrightarrow$	a(i,j)	1	2
$j \downarrow$		1	1	2
		2	3	4

C/C++

	$i \longrightarrow$	a[i][j]	0	1
$j \downarrow$		0	1	3
		1	2	4

Other things you'll encounter

Programming Language Extensions

- Includes things like
 - Syntax or Semantics not defined in the standard
 - An example is Co-array Fortran
 - Until Fortran 2008 standard
 - Additional functions
 - Interfaces to OS
- BEWARE
 - Extensions are usually not portable between compilers

Directives and Preprocessor

- Primarily applicable to high level languages with compilers
- Preprocessor
 - Happens before compilation
 - Can control what source code is compiled
 - Mechanism for portability
- Directives
 - can appear as comments or preprocessor commands
 - Allows for safe compilation when feature is not available.