

# ***Object Oriented Programming via Fortran 90/95***

*Ed Akin*

Rice University  
Mechanical Engineering and Materials Science Department  
Houston, Texas

May 29, 2001



# Contents

<b>Preface</b>	<b>vii</b>
<b>1 Program Design</b>	<b>1</b>
1.1. Introduction . . . . .	1
1.2. Problem Definition . . . . .	3
1.3. Modular Program Design . . . . .	6
1.4. Program Composition . . . . .	9
1.4.1. Comments . . . . .	9
1.4.2. Statements . . . . .	9
1.4.3. Flow Control . . . . .	11
1.4.4. Functions . . . . .	13
1.4.5. Modules . . . . .	15
1.4.6. Dynamic Memory Management . . . . .	15
1.5. Program evaluation and testing . . . . .	15
1.6. Program documentation . . . . .	17
1.7. Object Oriented Formulations . . . . .	18
1.8. Exercises . . . . .	21
<b>2 Data Types</b>	<b>23</b>
2.1. Intrinsic Types . . . . .	23
2.2. User Defined Data Types . . . . .	25
2.3. Abstract Data Types . . . . .	27
2.4. Classes . . . . .	29
2.5. Exercises . . . . .	31
<b>3 Object Oriented Programming Concepts</b>	<b>33</b>
3.1. Introduction . . . . .	33
3.2. Encapsulation, Inheritance, and Polymorphism . . . . .	34
3.2.1. Example Date, Person, and Student Classes . . . . .	37
3.3. Object Oriented Numerical Calculations . . . . .	38
3.3.1. A Rational Number Class and Operator Overloading . . . . .	39
3.4. Discussion . . . . .	42
3.5. Exercises . . . . .	48
<b>4 Features of Programming Languages</b>	<b>51</b>
4.1. Comments . . . . .	51
4.2. Statements and Expressions . . . . .	52
4.3. Flow Control . . . . .	57
4.3.1. Explicit Loops . . . . .	58
4.3.2. Implied Loops . . . . .	60
4.3.3. Conditionals . . . . .	61
4.4. Subprograms . . . . .	68

4.4.1. Functions and Subroutines . . . . .	68
4.4.2. Global Variables . . . . .	72
4.4.3. Bit Functions . . . . .	74
4.4.4. Exception Controls . . . . .	74
4.5. Interface Prototype . . . . .	75
4.6. Characters and Strings . . . . .	76
4.7. User Defined Data Types . . . . .	80
4.7.1. Overloading Operators . . . . .	84
4.7.2. User Defined Operators . . . . .	86
4.8. Pointers and Targets . . . . .	86
4.8.1. Pointer Type Declaration . . . . .	87
4.8.2. Pointer Assignment . . . . .	88
4.8.3. Using Pointers in Expressions . . . . .	88
4.8.4. Pointers and Linked Lists . . . . .	88
4.9. Accessing External Source Files and Functions . . . . .	89
4.10. Procedural Applications . . . . .	90
4.10.1. Fitting Curves to Data . . . . .	90
4.10.2. Sorting . . . . .	92
4.11. Exercises . . . . .	99
<b>5 Object Oriented Methods</b>	<b>103</b>
5.1. Introduction . . . . .	103
5.2. The Drill Class . . . . .	103
5.3. Global Positioning Satellite Distances . . . . .	106
5.4. Exercises . . . . .	118
<b>6 Inheritance and Polymorphism</b>	<b>119</b>
6.1. Introduction . . . . .	119
6.2. Example Applications of Inheritance . . . . .	121
6.2.1. The Professor Class . . . . .	121
6.2.2. The Employee and Manager Classes . . . . .	121
6.3. Polymorphism . . . . .	124
6.3.1. Templates . . . . .	125
6.3.2. Subtyping Objects (Dynamic Dispatching) . . . . .	130
6.4. Exercises . . . . .	133
<b>7 OO Data Structures</b>	<b>135</b>
7.1. Data Structures . . . . .	135
7.2. Stacks . . . . .	135
7.3. Queues . . . . .	139
7.4. Linked Lists . . . . .	142
7.4.1. Singly Linked Lists . . . . .	142
7.4.2. Doubly Linked Lists . . . . .	148
7.5. Direct (Random) Access Files . . . . .	149
7.6. Exercises . . . . .	153
<b>8 Arrays and Matrices</b>	<b>155</b>
8.1. Subscripted Variables: Arrays . . . . .	155
8.1.1. Initializing Array Elements . . . . .	158
8.1.2. Intrinsic Array Functions . . . . .	159
8.1.3. Colon Operations on Arrays (Subscript Triplet) . . . . .	159
8.1.4. Array Logical Mask Operators . . . . .	163
8.1.5. User Defined Operators . . . . .	165
8.1.6. Connectivity Lists and Vector Subscripts . . . . .	166

8.1.7. Component Gather and Scatter . . . . .	168
8.2. Matrices . . . . .	170
8.2.1. Matrix Algebra . . . . .	172
8.2.2. Inversion . . . . .	174
8.2.3. Factorizations . . . . .	174
8.2.4. Determinant of a Matrix . . . . .	175
8.2.5. Matrix Calculus . . . . .	176
8.2.6. Computation with Matrices . . . . .	176
8.3. Exercises . . . . .	178
<b>9 Advanced Topics</b>	<b>181</b>
9.1. Templates . . . . .	181
9.2. Subtyping Objects (Dynamic Dispatching) . . . . .	183
9.3. Non-standard Features . . . . .	184
<b>A Bibliography</b>	<b>187</b>
<b>B Fortran 90 Overview</b>	<b>191</b>
B.1. List of Language Tables . . . . .	191
B.2. Alphabetical Table of Fortran 90 Intrinsic Routines . . . . .	17
B.3. Syntax of Fortran 90 Statements . . . . .	29
<b>C Selected Exercise Solutions</b>	<b>47</b>
C.1. Problem 1.8.1 : Checking trigonometric identities . . . . .	47
C.2. Problem 1.8.2 : Newton-Raphson algorithm . . . . .	47
C.3. Problem 1.8.3 : Game of life . . . . .	48
C.4. Problem 2.5.1 : Conversion factors . . . . .	49
C.5. Problem 3.5.3 : Creating a vector class . . . . .	50
C.6. Problem 3.5.4 : Creating a sparse vector class . . . . .	56
C.7. Problem 4.11.1 : Count the lines in an external file . . . . .	61
C.8. Problem 4.11.3 : Computing CPU time usage . . . . .	62
C.9. Problem 4.11.4 : Converting a string to upper case . . . . .	62
C.10. Problem 4.11.8 : Read two values from each line of an external file . . . . .	63
C.11. Problem 4.11.14 : Two line least square fits . . . . .	63
C.12. Problem 4.11.15 : Find the next available file unit . . . . .	65
C.13. Problem 5.4.4 : Polymorphic interface for the class 'Position_Angle' . . . . .	66
C.14. Problem 6.4.1 : Using a function with the same name in two classes . . . . .	67
C.15. Problem 6.4.3 : Revising the employee-manager classes . . . . .	67
<b>D Companion C++ Examples</b>	<b>69</b>
D.1. Introduction . . . . .	69
<b>E Glossary of Object Oriented Terms</b>	<b>77</b>
<b>F Subject Index</b>	<b>0</b>
<b>G Program Index</b>	<b>1</b>



# Preface

There has been an explosion of interest in, and books on object-oriented programming (OOP). Why have yet another book on the subject? In the past a basic education was said to master the three r's: reading, 'riting, and 'rithmetic. Today a sound education in engineering programming leads to producing code that satisfy the four r's: readability, reusability, reliability, and really-efficient. While some object-oriented programming languages have some of these abilities Fortran 90/95 offers all of them for engineering applications. Thus this book is intended to take a different tack by using the Fortran 90/95 language as its main OOP tool. With more than one hundred pure and hybrid object-oriented languages available, one must be selective in deciding which ones merit the effort of learning to utilize them. There are millions of Fortran programmers, so it is logical to present the hybrid object-oriented features of Fortran 90/95 to them to update and expand their programming skills. This work provides an introduction to Fortran 90 as well as to object-oriented programming concepts. Even with the current release (Fortran 95) we will demonstrate that Fortran offers essentially all of the tools recommended for object-oriented programming techniques. It is expected that Fortran 200X will offer additional object-oriented capabilities, such as declaring "extensible" (or virtual) functions. Thus, it is expected that the tools learned here will be of value far into the future.

It is commonly agreed that the two decades old F77 standard for the language was missing several useful and important concepts of computer science that evolved and were made popular after its release, but it also had a large number of powerful and useful features. The following F90 standard included a large number of improvements that have often been overlooked by many programmers. It is fully compatible with all old F77 standard code, but it declared several features of that standard as obsolete. That was done to encourage programmers to learn better methods, even though the standard still supports those now obsolete language constructs. The F90 standards committee brought into the language most of the best features of other more recent languages like Ada, C, C++, Eiffel, etc. Those additions included in part: structures, dynamic memory management, recursion, pointers (references), and abstract data types along with their supporting tools of encapsulation, inheritance, and the overloading of operators and routines. Equally important for those involved in numerical analysis the F90 standard added several new features for efficient array operations that are very similar to those of the popular MATLAB environment. Most of those features include additional options to employ logical filters on arrays. All of the new array features were intended for use on vector or parallel computers and allow programmers to avoid the bad habit of writing numerous serial loops. The current standard, F95, went on to add more specific parallel array tools, provided "pure" routines for general parallel operations, simplified the use of pointers, and made a few user friendly refinements of some F90 features. Indeed, at this time one can view F90/95 as the only cross-platform international standard language for parallel computing. Thus Fortran continues to be an important programming language that richly rewards the effort of learning to take advantage of its power, clarity, and user friendliness.

We begin that learning process in Chapter 1 with an overview of general programming techniques. Primarily the older "procedural" approach is discussed there, but the chapter is closed with an outline of the newer "object" approach to programming. An experienced programmer may want to skip directly to the last section of Chapter 1 where we outline some object-oriented methods. In Chapter 2, we introduce the concept of the abstract data types and their extension to classes. Chapter 3 provides a fairly detailed introduction to the concepts and terminology of object-oriented programming. A much larger supporting glossary is provided as an appendix.

For the sake of completeness Chapter 4 introduces language specific details of the topics discussed in

the first chapter. The Fortran 90/95 syntax is used there, but in several cases cross-references are made to similar constructs in the C++ language and the MATLAB environment. While some readers may want to skip Chapter 4, it will help others learn the Fortran 90/95 syntax and/or to read related publications that use C++ or MATLAB. All of the syntax of Fortran 90 is also given in an appendix.

Since many Fortran applications relate to manipulating arrays or doing numerical matrix analysis, Chapter 5 presents a very detailed coverage of the powerful intrinsic features Fortran 90 has added to provide for more efficient operations with arrays. It has been demonstrated in the literature that object-oriented implementations of scientific projects requiring intensive operations with arrays execute much faster in Fortran 90 than in C++. Since Fortran 90 was designed for operations on vector and parallel machines that chapter encourages the programmer to avoid unneeded serial loops and to replace them with more efficient intrinsic array functions. Readers not needing to use numerical matrix analysis may skip Chapter 5.

Chapter 6 returns to object-oriented methods with a more detailed coverage of using object-oriented analysis and object-oriented design to create classes and demonstrates how to implement them as an OOP in Fortran 90. Additional Fortran 90 examples of inheritance and polymorphism are given in Chapter 7. Object-oriented programs often require the objects to be stored in some type of “container” or data structure such as a stack or linked-list. Fortran 90 object-oriented examples of typical containers are given in Chapter 8. Some specialized topics for more advanced users are given in Chapter 9, so beginning programmers could skip it.

To summarize the two optional uses of this text; it is recommended that experienced Fortran programmers wishing to learn to use OOP cover Chapters 2, 3, 6, 7, 8, and 9, while persons studying Fortran for the first time should cover Chapters 1, 2, 3, and. Anyone needing to use numerical matrix analysis should also include Chapter 5.

A OO glossary is included in an appendix to aid in reading this text and the current literature on OOP. Another appendix on Fortran 90 gives an alphabetical listing on its intrinsic routines, a subject based list of them, a detailed syntax of all the F90 statements, and a set of example uses of every statement. Selected solutions for most of the assignments are included in another appendix along with comments on those solutions. The final appendix gives the C++ versions of several of the F90 examples in the text. They are provided as an aid to understanding other OOP literature. Since F90 and MATLAB are so similar the corresponding MATLAB versions often directly follow the F90 examples in the text.

Ed Akin, Rice University, 2002



## Chapter 1

# Program Design

### 1.1 Introduction

The programming process is similar in approach and creativity to writing a paper. In composition, you are writing to express ideas; in programming you are expressing a computation. Both the programmer and the writer must adhere to the syntactic rules (grammar) of a particular *language*. In prose, the fundamental idea-expressing unit is the sentence; in programming, two units — *statements* and *comments* — are available.

Standing back, composition from technical prose to fiction should be organized broadly, usually through an outline. The outline should be expanded as the detail is elaborated, and the whole re-examined and re-organized when structural or creative flaws arise. Once the outline settles, you begin the actual composition process, using sentences to weave the fabric your outline expresses. *Clarity* in writing occurs when your sentences, both internally and globally, communicate the outline succinctly and clearly. We stress this approach here, with the aim of developing a *programming style that produces efficient programs that humans can easily understand*.

To a great degree, no matter which language you choose for your composition, the idea can be expressed with the same degree of clarity. Some subtleties can be better expressed in one language than another, but the fundamental reason for choosing your language is your audience: People do not know many languages, and if you want to address the American population, you had better choose English over Swahili. Similar situations happen in programming languages, but they are not nearly so complex or diverse. The number of languages is far fewer, and their differences minor. Fortran is the oldest language among those in use today. C and C++ differ from it somewhat, but there are more similarities than not. MATLAB's language, written in C and Fortran, was created much later than these two, and its structure is so similar to the others that it can be easily mastered. The C++ language is an extension of the C language that places its emphasis on object oriented programming (OOP) methods. Fortran added object oriented capabilities with its F90 standard, and additional enhancements for parallel machines were issued with F95. The Fortran 2000 standard is planned to contain more user-friendly constructs for polymorphism and will, thus, enhance its object-oriented capabilities. This creation of a new language and its similarity to more established ones are this book's main points: More computer programming languages will be created during your career, but these new languages will probably not be much different than ones you already know. Why should new languages evolve? In MATLAB's case, it was the desire to express matrix-like expressions easily that motivated its creation. The difference between MATLAB and Fortran 90 is infinitesimally small compare to the gap between English and Swahili.

An important difference between programming and composition is that in programming you are writing for two audiences: people and computers. As for the computer audience, what you write is "read" by interpreters and compilers specific to the language you used. They are *very* rigid about syntactic rules, and perform *exactly* the calculations you say. It is like a document you write being read by the most detailed, picky person you know; every pronoun is questioned, and if the antecedent is not perfectly clear, then they throw up their hands, rigidly declaring that the *entire* document cannot be understood. Your picky friend might interpret the sentence "Pick you up at eight" to mean that you will literally lift him or her off the ground at precisely 8 o'clock, and then demand to know whether the time is in the morning or

afternoon and what the date is.

Humans demand even more from programs. This audience consists of two main groups, whose goals can conflict. The larger of the two groups consists of *users*. Users care about how the program presents itself, its *user interface*, and how quickly the program runs, how *efficient* it is. To satisfy this audience, programmers may use statements that are overly terse because they know how to make the program more readable by the computer's compiler, enabling the compiler to produce faster, but less human-intelligible program. This approach causes the other portion of the audience—programmers—to boo and hiss. The smaller audience, of which *you* are also a member, must be able to read the program so that they can enhance and/or change it. A characteristic of programs, which further distinguishes it from prose, is that you and others will seek to modify your program in the future. For example, in the 1960s when the first version of Fortran was created, useful programs by today's standards (such as matrix inversion) were written. Back then, the user interface possibilities were quite limited, and the use of visual displays was limited. Thirty years later, you would (conceivably) want to take an old program, and provide a modern user interface. *If* the program is structurally sound (a good outline and organized well) and is well-written, re-using the “good” portions is easily accomplished.

The three-audience situation has prompted most languages to support *both* computer-oriented and human-oriented “prose”. The program's meaning is conveyed by *statements*, and is what the computer interprets. Humans read this part, which in virtually all languages bears a strong relationship to mathematical equations, and also read *comments*. Comments are *not* read by the computer at all, but are there to help explain what might be expressed in a complicated way by programming language syntax. *The document or program you write today should be understandable tomorrow*, not only by you, but also by others. Sentences and paragraphs should make sense after a day or so of gestation. Paragraphs and larger conceptual units should not make assumptions or leaps that confuse the reader. Otherwise, the document you write for yourself or others served no purpose. The same is true with programming; the program's organization should be easy to follow and the way you write the program, using both statements and comments, should help you and others understand how the computation proceeds. The existence of comments permits the writer to directly express the program's outline in the program to help the reader comprehend the computation.

These similarities highlight the parallels between composition and programming. Differences become evident because programming is, in many ways, more demanding than prose writing. On one hand, the components and structure of programming languages are far simpler than the grammar and syntax of any verbal or written language. When reading a document, you can figure out the misspelled words, and not be bothered about every little imprecision in interpreting what is written. On the other, simple errors, akin to misspelled words or unclear antecedents, can completely obviate a program, rendering it senseless or causing it to go wildly wrong during execution. For example, there is no real dictionary when it comes to programming. You can define variable names containing virtually any combination of letters (upper and lower case), underscores, *and* numbers. A typographical error in a variable's name can therefore lead to unpredictable program behavior. Furthermore, computer execution speeds are becoming faster and faster, meaning that increasingly complex programs can run very quickly. For example, the program (actually groups of programs) that run NASA's space shuttle might be comparable in size to Hugo's *Les Misérables*, but its complexity and immediate importance to the “user” far exceeds that of the novel.

As a consequence, program design must be extremely structured, having the ultimate intentions of performing a specific calculation efficiently with attractive, understandable, efficient programs. Achieving these general goals means breaking the program into components, writing and testing them separately, then merging them according to the outline. Toward this end, we stress *modular programming*. Modules can be on the scale of chapters or paragraphs, and share many of the same features. They consist of a sequence of statements that by themselves express a meaningful computation. They can be merged to form larger programs by specifying what they do and how they *interface* to other packages of software. The analogy in prose is agreeing on the character's names and what events are to happen in each paragraph so that events happen to the right people in the right sequence once the whole is formed. Modules can be re-used in two ways. As with our program from the 1960s, we would “lift” the matrix inversion routine and put a different user interface around it. We can also re-use a routine within a program several times. For example, solving the equations of space flight involves the inversion of many matrices. We would

want our program to use the matrix inversion routine over and over, presenting it with a different matrix each time.

The fundamental components of good program design are

1. Problem definition, leading to a program specification
2. Modular program design, which refines the specification
3. Module composition, which translates specification into executable program
4. Module/program evaluation and testing, during which you refine the program and find errors
5. Program documentation, which pervades all other phases

The result of following these steps is an efficient, easy-to-use program that has a user's guide (how does someone else run your program) and internal documentation so that other programmers can decipher the algorithm.

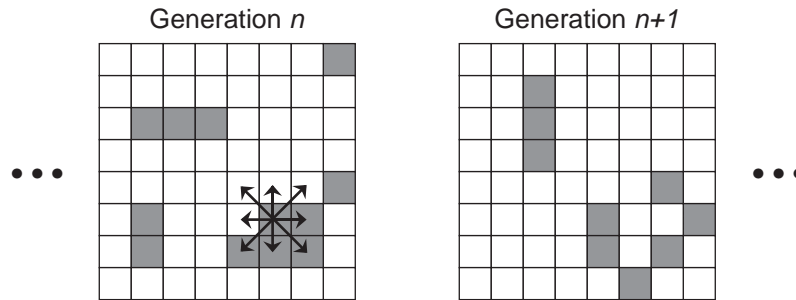
Today it is common in a university education to be required to learn at least one foreign language. Global interactions in business, engineering, and government make such a skill valuable to one's career. So it is in programming. One often needs to be able to read two or three programming languages—even if you compose programs in only one language. It is common for different program modules, in different languages, to be compiled separately and then brought together by a “linker” to form a single executable. When something goes wrong in such a process it is usually helpful to have a reading knowledge of the programming languages being used.

When composing to express ideas there are, at least, two different approaches to consider: poetry and prose. Likewise, in employing programming languages to create software there are distinctly different approaches available. The two most common ones are “procedural programming” and “object-oriented programming.” The two approaches are conceptually sketched in Fig. 1.1. They differ in the way that the software development and maintenance are planned and implemented. Procedures may use objects, and objects usually use procedures, called *methods*. Usually the object-oriented code takes more planning and is significantly larger, but it is generally accepted to be easier to maintain. Today when one can have literally millions of users active for years or decades, maintenance considerations are very important.

## 1.2 Problem Definition

The problem the program is to solve must be well specified. The programmer must broadly frame the program's intent and context by answering several questions.

- *What must the program accomplish?*  
From operating the space shuttle to inverting a small matrix, some thought must be given to *how* the program will do what is needed. In technical terms, we need to define the *algorithm* employed in small-scale programs. In particular, numeric programs need to consider well how calculations are performed. For example, finding the roots of a general polynomial *demands* a numeric (non-closed form) solution. The choice of algorithm is influenced by the variations in polynomial order and the accuracy demanded.
- *What inputs are required and in what forms?*  
Most programs interact with humans and/or other programs. This interaction needs to be clearly specified as to *what* format the data will take and *when* the data need to be requested or arrive.
- *What is the execution environment and what should be in the user interface?*  
Is the program a stand-alone program, calculating the quadratic formula for example, or do the results need to be plotted? In the former case, simple user input is probably all that is needed, but the programmer might want to write the program so that its key components could be used in other programs. In the latter, the program probably needs to be written so that it meshes well with some pre-written graphics environment.



**Figure 1.1:** Here, the game is played on an  $8 \times 8$  square array, and the filled squares indicate the presence of life. The arrows emanating from one cells radiate to its eight neighbors. The rules are applied to the  $n^{\text{th}}$  generation to yield the next. The row of three filled cells became a column of three, for example. What is going to happen to this configuration the next generation?

- What are the required and optional outputs, and what are their formats (printed, magnetic, graphical, audio)?

In many cases, output takes two forms: *interactive* and *archival*. Interactive output means that the programs results must be provided to the user or to other programs. Data format must be defined so that the user can quickly see or hear the programs results. Archival results need to be stored on long-term media, such as disk, so that later interpretation of the file's contents is easy (recall the notion of being able to read tomorrow what is written today) and that the reading process is easy.

The answers to these questions help programmers organize their thoughts, and can lead to decisions about programming language and operating environment. At this point in the programming process, the programmer should know what the program is to do and for whom the program is written. We don't yet have a clear notion of how the program will accomplish these tasks; that comes down the road. This approach to program organization and design is known as *top-down* design. Here, broad program goals and context is defined first, with additional detail filled in as needed. This approach contrasts with *bottom-up* design, where the detail is decided first, then merged into a functioning whole. For programming, top-design makes more sense, but you as well as professional programmers are frequently lured into writing code immediately, usually motivated by the desire to "get something running and figure out later how to organize it all." That approach is motivated by expediency, but usually winds up being more inefficient than a more considered, top-down approach that takes longer to get off the ground, but with increased likelihood of working more quickly. The result of defining the programming problem is a *specification*: how is the program structured, what computations does it perform, and how should it interact with the user.

### An Extended Example: The Game of Life

To illustrate how to organize and write a simple program, let's structure a program that plays *The Game of Life*. Conway's "Game of Life" was popularized in Martin Gardner's Mathematical Games column in the October 1970 and February 1971 issues of *Scientific American*. This game is an example of what is known in computer science as *cellular automata*. An extensive description of the game can be found in *The Recursive Universe* by William Poundstone (Oxford University Press, 1987).

The rules of the game are quite simple. Imagine a rectangular array of square cells that are either empty (no living being present) or filled (a being lives there). As shown in Fig. 1.1, each cell has eight neighboring cells. At each tick of the clock, a new generation of beings is produced according to how many neighbors surround a given cell.

- If a cell is empty, fill it if three of its neighboring cells are filled; otherwise, leave it empty.
- If a cell is filled, it
  - dies of loneliness if it has zero or one neighbors,
  - continues to live if it has two or three neighbors,
  - dies of overcrowding if it has more than three neighbors.

The programming task is to allow the user to “play the game” by letting him or her define initial configurations, start the program, which applies the rules and displays each generation, and stop the game at any time the user wants, returning to the initialization stage so that a new configuration can be tried. To understand the program task, we as programmers need to pose several questions, some of which might be

- What computer(s) are preferred, and what kind of display facilities do they have?
- Is the size of the array arbitrary or fixed?
- Am I the only programmer?

No matter how these questions are answered, we start by forming the program’s basic outline. Here is one way we might outline the program in a procedural fashion.

1. Allow the user to initialize the rectangular array or quit the program.
2. Start the calculation of the next generation.
  - (a) Apply game rules to the current array.
  - (b) Generate a new array.
  - (c) Display the array.
  - (d) Determine whether the user wants to stop or not.
    - i. If not, go back to 2a.
    - ii. If so, go to step 1

Note how the idea of reusing the portion of the program that applies game rules arises naturally. This idea is peculiar to programming languages, having no counterpart in prose (It’s like being told at the end of a chapter to reread it!). This kind of *looping* behavior also occurs when we go back and allow the user to restart the program.

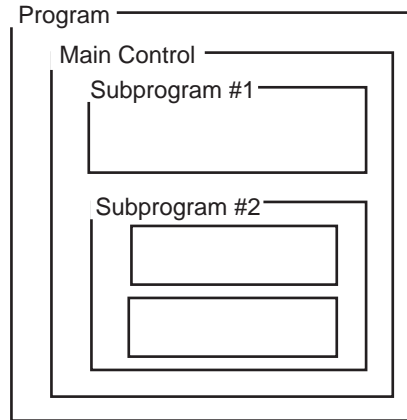
---

This kind of outline is a form of *pseudocode*:<sup>†</sup> A programming language-like expression of how the program operates. Note that at this point, the programming process is language-independent. Thus *informal pseudocode* allows us to determine the program’s broad structure. We have not yet resolved the issue of how, or if, the array should be displayed: Should it be refreshed as soon as a generation is calculated, or should we wait until a final state is reached or a step limit is exceeded? Furthermore, if calculating each generation takes a fair amount of time, our candidate program organization will not allow the user to stop the program until a generation’s calculations have been finished. Consequently, we may, depending on the speed of the computer, want to limit the size of the array. A more detailed issue is how to represent the array internally. These issues can be determined later; programmers frequently make notes at this stage about how the program would behave with this structure. Informal pseudocode should remain in the final program in the form of comments.

*Writing a program’s outline is not a meaningless exercise.* How the program will behave is determined at that point. An alternative would be to ask the user how many generations should be calculated, then calculate all generations, and display the results as a movie, allowing the user to go backward, play in slow motion, freeze-frame, etc. Our outline will not allow such visual fun. Thus, programmers usually design several candidate program organizations, understand the consequences of each, and determine which best meets the specifications.

---

<sup>†</sup>The use of the word “code” is interesting here. It means program as both a noun and a verb: From the earliest days of programming, what the programmer produced was called *code*, and what he or she did was “code the algorithm.” The origin of this word is somewhat mysterious. It may have arisen as an analogy to Morse code, which used a series of dots and dashes as an alternative to the alphabet. This code is tedious to read, but ideal for telegraphic transmission. A program is an alternate form of an algorithm better suited to computation.



**Figure 1.2:** Modular program organization relies on self-contained routines where the passage of data (or messages) from one to the other is very well defined, and each routine’s (or objects) role in the program becomes evident.

### 1.3 Modular Program Design

We now need to define what the routines are and how they are interwoven to archive the program’s goals. (We will deepen this discussion to include objects and messages when we introduce object-oriented formulations in Sec. 1.7.) What granularity—how large should a routine be—comes with programming experience and depends somewhat on the language used to express it. A program typically begins with a main segment that controls or directs the solution of the problem by dividing it into sub-tasks (see Figure 1.2). Each of these may well be decomposed into other routines. This step-wise refinement continues as long as necessary, as long as it benefits program clarity and/or efficiency. This *modular program design* is the key feature of modern programming design practice. Furthermore, routines can be tested individually, and replaced or rewritten as needed. Before actually writing each routine, a job known in computer circles as the *implementation*, the program’s organization can be studied: Will the whole satisfy design specifications? Will the program execute efficiently? As the implementation proceeds, each routine’s *interface* is defined: How does it interact with its master—the routine that *called* it—and how are data exchanged between the two? In some most languages, this interface can be *prototyped*: The routine’s interface—what it expects and what values it calculates—can be defined and the whole program merged together and compiled to check for consistency without performing *any* calculations. In small programs, where you can have these routine definitions easily fitting onto one page, this prototyping can almost be performed visually. In complex programs, where there may be hundreds or thousands of routines, such prototyping *really* pays off. Once the interfaces begin to form, we ask whether they make sense: Do they exchange information efficiently? Does each routine have the information it needs or should the program be reorganized so that data exchange can be accomplished more efficiently?

From another viewpoint, you should develop a programming style that “hedges your bets:” Programs should be written in such a way that allows their components to be used in a variety of contexts. Again, using a modular programming style, the fundamental components of the calculation should be expressed as a series of subroutines or functions, the interweaving of which is controlled by a main program that reads the input information and produces the output. A modular program can have its components extracted, and used in other programs (program re-use) or interfaced to environments. So-called monolithic programs, which tend not to use routines and express the calculation as a single, long-winded program, should not be written.

We emphasize that this modular design process proceeds *without* actually writing program statements. We use a programming-like language, known as *formal pseudocode*, to express in prose what routines call others and how. This prose might re-express a graphic representation of program organization, such as that shown in Figure 1.2. In addition, expressing the program’s design in pseudocode eases the transition to program composition, the actual programming process. The components of formal pseudocode at this point are few:

```

[1]  ! This is a comment line in Fortran 90
[2]
[3]  program main                ! a program called main
[4]                                ! begin the main program
[5]    print *, "Hello, world"    ! * means default format
[6]  end program main            ! end the main program

[1]  // This is a comment line in C++
[2]  #include <iostream.h>      // standard input output library
[3]
[4]  main ()                    // a program called main
[5]                                // begin the main program
[6]    cout << "Hello, world" << endl ; // endl means new line
[7]    return 0;                // needed by some compilers
[8]                                // end the main program

[1]  % This is a comment line in MATLAB
[2]
[3]  function main ()           % a program called main
[4]                                % begin the main program
[5]    disp ('Hello, world');    % display the string
[6]                                % end the main program

```

**Figure 1.3:** 'Hello World' Program and Comments in Three Languages

- *comments* that we allow to include the original outline and to describe computational details;
- *functions* that express each routine, whether it be computational or concerned with the user interface;
- *conditionals* that express changing the flow of a program; and
- *loops* that express iteration.

**Comments.** A comment begins with a comment character, which in our pseudocode we take to be the exclamation point `!`, and ends when the line ends. Comments can consume an entire line or the right portion of some line.

```

! This is a comment: you can read it, but the computer won't
statements
statement ! From the comment character to end of this line is a comment
statements

```

The statements cited in the above lines share the status of the sentence that characterizes most written languages. It is made up of components specific to the syntax of the programming language in use. For example, most programming books begin with a program that does nothing but print "Hello world" on the screen (or other output device). The pseudocode for this might have the following form:

```

! if necessary, include the device library

initiate my program, say main

    send the character string ``Hello world`` to the output device library

terminate my program

```

Figure 1.3 illustrates this in three common languages, beginning with F90. At this point one can now say that they are multi-lingual in computer languages. Here, too, we may note that, unlike the other two languages shown, in Fortran when we begin a specific type of software construct, we almost always explicitly declare where we are ending its scope. Here the construct pair was `program` and `end program`, but the same style holds true for `if` and `end if` pairs, for example. All languages have rules and syntax to terminate the scope of some construct, but when several types of different constructs occur in the same program segment, it may be unclear in which order they are terminating.

**Functions.** To express a program's organization through its component routines and routines, we use the notation of mathematical *functions*. Each program routine accepts inputs, expressed as arguments of a function, performs its calculations, and returns the computational results as functional values.

```

output_1 = routine (input_1,...,input_m)

```

or

```
call routine (input_1,..., input_m, output_1,..., output_n)
```

In Fortran, a routine evaluating a single output object, as in the first style, is called a *function* and, otherwise, it is called a *subroutine*. Other languages usually use the term function in both cases. Each routine's various inputs and results are represented by *variables*, which, in sharp contrast to mathematical variables, have text-like names that indicate what they contain. These names contain *no* spaces, but may contain several words. There are two conventions for variable names containing two or more words: either words are joined by the underbar character “\_” (like `next_generation`) or each word begins with an uppercase letter (like `NextGeneration`). The results of a routine's computation are always indicated by a sequence of variables on the *left* side of the equals sign `=`. The use of an equals sign does not mean mathematical equality; it is a symbol in our pseudocode that means “assign a routine's results to the variables (in order) listed on the left.”

**Conditionals.** To create something other than a sequential execution of routines, conditionals form a test on the values of one or more variables, and continue execution at one point or another depending on whether the test was true or false. That is usually done with the `if` statement. It either performs the instruction(s) that immediately follow (after the `then` keyword) if some condition is valid (like `x > 0`) or those that follow the `else` statement if the condition is not true.

```
if test then
    statement group A ! executed if true
else
    statement group B ! executed if false
end if
```

The test here can be very complicated, but is always based on values of variables. Parentheses should be used to clarify exactly what the test is. For example,

```
((x > 0) and (y = 2))
```

One special statement frequently found in `if` statements is `stop`: This command means to stop or abort the program, usually with a fatal error message.

Conditionals allow the program to execute non-sequentially (the *only* mode allowed by statements). Furthermore, program execution order can be data-dependent. In this way, how the program behaves—what output it produces and how it computes the output—depends on what data, or messages, it is given. *This means that exact statement execution order is determined by the data, and/or messages, and the programmer—not just the programmer.* It is this aspect of programming languages that distinguishes them from written or spoken languages. An analogy might be that chapters in a novel are read in the order specified by the reader's birthday; what that order might be *is* determined by the novelist through logical constructs. The tricky part is that in programming languages, each execution order *must* make sense and not lead to inconsistencies or, at worst, errors: The novel must make sense in all the ways the novelist allows. This data- and message-dependent execution order can be applied at *all* programming levels, from routine execution to statements. Returning to our analogy to the novel, chapter (routine) order and sentence (statement) order depend on the reader's birthday. Such complexity in prose has little utility, but does in programming. How else can a program be written that informs the user on what day of the week and under what phase of the moon she was born given the birth date?

**Loops.** Looping constructs in our formal pseudocode take the form of *do loops*, where the keyword `do` is paired with the key phrase `end do` to mean that the expressions and routine invocations contained therein are calculated in order (from top to bottom), then calculated again starting with the first, then again, then again, ..., forever. The loop ceases only when we explicitly exit it with the `exit` command. The pseudocode loop shown below on the left has the execution history shown on the right.

<pre>do   y = routine_1(x)   z = routine_2(y)   x = routine_3(z)   if x &gt; 0 then     exit   end if end do</pre>	<pre>y = routine_1(x) z = routine_2(y) x = routine_3(z) [let's say x=-1] y = routine_1(x) z = routine_2(y) x = routine_3(z) [let's say x=1] [program ends]</pre>
--	--



Loop	Pseudocode
Indexed loop	do index=b,i,e statements end do
Pre-test loop	while (test) statements end while
Post-test loop	do statements if test exit end do

**Table 1.1:** Pseudocode loop constructs

*Infinite loops* occur when the Boolean expression always evaluates to true; these are usually not what the programmer intended and represent one type of program error—a “bug.”<sup>†</sup> The constructs enclosed by the loop can be *anything*: statements, logical constructs, and other loops! Because of this variety, programs can exhibit extremely complex behaviors. How a program behaves depends *entirely* on the programmer and how their definition of the program flows based on user-supplied data and messages. The pseudocode loops are defined in Table 1.1.

## 1.4 Program Composition

Composing a program is the process of expressing or translating the program design into computer language(s) selected for the task. Whereas the program design can often be expressed as a broad outline, each routine’s algorithm must be expressed in complete detail. This writing process elaborates the formal pseudocode and contains more explicit statements that more greatly resemble generic program statements.

Generic programming language elements fall into five basic categories: the four we had before—comments, loops, conditionals, and functions—and *statements*. We will expand the variety of comments, conditionals, loops, and functions/subroutines, which define routines and their interfaces. The new element is the statement, the workhorse of programming. It is the statement that actually performs a concrete computation. In addition to expanding the repertoire of programming constructs for formal pseudocode, we also introduce what these constructs are in MATLAB, Fortran, and C++. As we shall see, formal pseudocode parallels these languages; the translation from pseudocode to executable program is generally easy.

### 1.4.1 Comments

Comments need no further elaboration for pseudocode. However, programmers are encouraged to make heavy use of comments.

### 1.4.2 Statements

Calculation is expressed by *statements*, which share the structure (and the status) of the sentence that characterizes virtually all written language. Statements that are always executed one after the other as written. A statement in most languages has a simple, well-defined structure common to them all.

variable = expression

---

<sup>†</sup>This term was originated by Grace Hopper, one of the first programmers. In the early days of computers, they were partially built with mechanical devices known as relays. A relay is a mechanical switch that controls which way electric current flows: The realization of the logical construct in programming languages. One day, a previously working program stopped being so. Investigation revealed that an insect had crawled into the computer and had become lodged in a relay’s contacts. She then coined the term “bug” to refer not only to such hardware failures, but to software ones as well since the user becomes upset no matter which occurs.

Statements are intended to bear a great resemblance to mathematical equations. This analogy with mathematics can appear confusing to the first-time programmer. For example, the statement `a = a+1`, which means “increment the variable `a` by one” makes perfect sense as a programming statement, but no sense as an algebraic equality since it seems to say that  $0 = 1$ . Once you become more fluent in programming languages, what is mathematics and what is programming become easily apparent. Statements are said to be *terminated* when a certain character is encountered by the interpreter or the compiler. In Fortran, the termination character is a carriage return or a semicolon (`;`). In C++, *all* statements must be terminated with a semicolon or a comma; carriage returns do *not* terminate statements. MATLAB statements may end with a semicolon `;` to suppress display of the calculated expression’s value. Most statements in MATLAB programs end thusly.

Sometimes, statements become quite long, becoming unreadable. Two solutions to improve clarity can be used: decompose the expression into simpler expressions or use *continuation markers* to allow the statement to span more than one line of text. The first solution requires you to use intermediate variables, which only results in program clutter. Multiline statements can be broken at convenient arithmetic operators, and this approach is generally preferred. C++ has no continuation character; statements can span multiple text lines, and end only when the semicolon is encountered. In MATLAB, the continuation character sequence comprise three periods `...` placed at the end of each text line (before the carriage return or comment character). In Fortran, a statement is continued to the next line when an ampersand `&` is the last character on the line.

**Variables.** A *variable* is a named sequence of memory locations to which values can be assigned. As such, every variable has an address in memory, which most languages conceal from the programmer so as to present the programmer with a *storage model* independent of the architecture of the computer running the program. Program variables correspond roughly to mathematical variables that can be integer, real, or, complex-valued. Program variables can be more general than this, being able in some languages to have values equal to a user-defined data type or object which, in turn, contains sequences of other variables. Variables in all languages have *names*: a sequence of alphanumeric characters that cannot begin with a number. Thus, `a`, `A`, `a2`, and `a9b` are feasible variable names (i.e., the interpreter or compiler will not complain about these) while `3d` is not. Since programs are meant to be read by humans as well as interpreters and compilers, such names may not lead to program clarity *even if* they are carefully defined and documented. The compiler/interpreter does not care whether humans can read a program easily or not, but you should: *Use variable names that express what the variables represent*. For example, use `force` as a name rather than `f`; use `i`, `j`, and `k` for indices rather than `ii` or `il`.

In most languages, variables have *type*: the kind of quantity stored in them. Frequently occurring data types are integer and floating point, for example. Integer variables would be chosen if the variable were only used as an array index; floating point if the variable might have a fractional part.

In addition to having a name, type, and address, each variable has a value of the proper type. The value should be assigned before the variable is used elsewhere. Compilers should indicate an error if a variable is used before it has been assigned a value. Some languages allow variables to have aliases which are usually referred to as “pointers” or “references”. Most higher level languages also allow programmers to create “user defined” data types.

**Assignment Operator.** The symbol `=` in a statement means *assignment* of the expression into the variable provided on the left. This symbol does not mean algebraic equality; it means that once expression is computed, its value is stored in the variable. Thus, statements that make programming sense, like `a=a+1`, make no mathematical sense because `=` means different things in the two contexts. Fortran 90, and other languages, allow the user to extend the meaning of the assignment symbol (`=`) to other operations. Such advanced features are referred to as “operator overloading”.

**Expressions.** Just as in mathematics, expressions in programming languages can have a complicated structure. Most encountered in engineering programs amount to a mathematical expression involving variables, numbers, and functions of variables and/or numbers. For example the following are all valid statements.

```
A = B
x = sin(2*z)
force = G*mass1*mass2/(r*r)
```

Thus, mathematical expressions obey the usual mathematical conventions, but with one added complexity: Vertical position cannot be used help express what the calculation is; program expressions have only one dimension. For example, the notation  $\frac{a}{b}c$  clearly expresses to you how to perform the calculation. However, the one-dimensional equivalent, obtained by smashing this expression onto one line, becomes ambiguous: does  $a/bc$  mean divide  $a$  by  $b$  then multiply by  $c$ , or divide  $a$  by the product of  $b$  and  $c$ ? This ambiguity is relieved in program expressions in two ways. The first, the human-oriented way, demands the use of parentheses—grouping constructs—to clarify what is being meant, as in  $(a/b)c$ . The language-oriented way makes use of *precedence rules*: What an expression means is inferred from a set of rules that specify what operations take effect first. In our example, because division is stronger than multiplication,  $a/bc$  means  $(a/b)c$ . Most people find that frequent reliance on precedence rules leads to programs that take a long time to decipher; the compiler/interpreter is “happy” either way.

Expressions make use of the common arithmetic and relational operators. They may also involve function evaluations; the `sin` function was called in the second expression given in the previous example. Programming expressions can be as complicated as the arithmetic or Boolean-algebra ones they emulate.

### 1.4.3 Flow Control

If a program consisted of a series of statements, statements would be executed one after the other, in the order they were written. Such is the structure of all prose, where the equivalent of a statement is the sentence. Programming languages differ markedly from prose in that statements can be meaningfully executed over and over, with details of each execution differing each time (the value of some variable might be changed), or some statements skipped, with statement ordering dependent on which statements were executed previously or upon external events (the user clicked the mouse). With this extra variability, programming languages can be more difficult for the human to trace program execution than the effort it takes to read a novel. In written languages, sentences can be incredibly complex, much more so than program statements; in programming, the sequencing of statements—*program flow*—can be more complex.

The basic flow control constructs present in virtually all programming languages are *loops*—repetitive execution of a series of statements—and *conditionals*—diversions around statements.

**Loops.** Historically, the loop has been a major tool in designing the flow control of a procedure and one would often code a loop segment without giving it a second thought. Today massively parallel computers are being widely used and one must learn to avoid coding explicit loops in order to take advantage of the power of such machines. Later we will review which intrinsic tools are included in F90 for use on parallel (and serial) computers to offer improved efficiency over explicit loops.

The loop allows the programmer to repeat a series of statements, with a parameter—the *loop variable*—taking on a different value for each repetition. The loop variable can be an integer or a floating-point number. Loops can be used to control iterative algorithms, such as the Newton-Raphson algorithm for finding solutions to nonlinear equations, to accumulate results for a sequential calculation, or to merely repeat a program phrase, such as awaiting for the next typed input. Loops are controlled by a logical expression, which when evaluated to `true` allows the loop another iteration and when false terminates the loop and commences program execution with the statement immediately following those statements enclosed within the loop.

There are three basic kinds of looping constructs, the choice of which is determined by the kind of iterative behavior most appropriate to the computation. The *indexed loop* occurs most frequently in programs. Here, one loop variable varies across a range of values. In pseudocode, the index’s value begins at  $b$ , increments each time through the loop by  $i$ , and the loop ends when the index exceeds  $e$ . For example:

```
do j = b, e, i
```

or using the default increment of unity:

```
do j = b, e
```

As an example of an indexed loop, let’s explore summing the series of numbers stored in the array `A`. If we knew the number of elements in the array when we write the program, the sum can be calculated

explicitly without using a loop.

$$\text{sum} = A_1 + A_2 + A_3 + A_4$$

However, we have already said that our statements must be on a single line, so we need a way to represent the subscript attached to each number. We develop the convention that a subscript is placed inside parentheses like

$$\text{sum} = A(1) + A(2) + A(3) + A(4)$$

Such programs are very inflexible, and this *hard-wired* programming style is discouraged. For example, suppose in another problem the array contains 1,000 elements. With an indexed loop, a more flexible, easier to read program can be obtained. Here, the index assumes a succession of values, its value tested against the termination value *before* the enclosed statements are executed, with the loop terminating once this test fails to be true. The following generic indexed loop also sums array elements, but in a much more flexible, concise way.

```
sum = 0
for i = 1,n
    sum = sum + A(i)
end for
```

Here, the variable *n* does *not* need to be known when the program is written; this value can wait until the program executes, and can be established by the user or after data is read.

In F90 the extensive support for matrix expressions allows *implicit loops*. For example, consider the calculation of  $\sum_{i=1}^N x_i y_i$ . The language provides at least three ways of performing this calculation. Assuming the vectors *x* and *y* are column vectors,

```
1. sum_xy = 0
   N = size(x)
   do i = 1,N
       sum_xy = sum_xy + x(i)*y(i)
   end do

2. sum_xy = sum(x*y)

3. sum_xy = dot_product(x,y)
```

The first method is based on the basic loop construct, and yields the slowest running program of the three versions. In fact, avoiding the *do* statement by using implicit loops will almost always lead to faster running programs. The second, and third statements employ intrinsic functions and/or operators designed for arrays. In many circumstances, calculation efficiency and clarity of expression must be balanced. In practice, it is usually necessary to set aside memory to hold subscripted arrays, such as *x* and *y* above, before they can be referenced or used.

**Conditionals.** Conditionals test the veracity of logical expressions, and execute blocks of statements accordingly (see Table 1.2). The most basic operation occurs when we want to execute a series of statements when a logical expression, say *test*, evaluates to *true*. We call that a simple if conditional; the beginning and end of the statements to be executed when *test* evaluates to *true* are enclosed by special delimiters, which differ according to language. When only one statement is needed, C++ and Fortran allow that one statement to end the line that begins with the if conditional. When you want one group of statements to be executed when *test* is *true* and another set to be executed when *false*, you use the if-else construct. When you want to test a series of logical expressions that are not necessarily complementary, the nested-if construct allows for essentially arbitrarily complex structure to be defined. In such cases, the logical tests can interlock, thereby creating programs that are quite difficult to read. Here is where program comments become essential. For example, suppose you want to sum only the positive numbers less than or equal to 10 in a given sequence. Let's assume the entire sequence is stored in array *A*. In informal pseudocode, we might write

```
loop across A
  if A(i) > 0 and A(i) <= 10 add to sum
end of loop
```

More formally, this program fragment as a complete pseudocode would be

```
sum = 0
```

Conditional	Pseudocode
if	if (test) statement
if	if test then statements end if
if-else	if test then statements A else statements B end if
nested if	if test1 then statements A if test2 then statements B end if % end of test2 end if

**Table 1.2:** Syntax of pseudocode conditionals

```

do i=1,n
  if (A(i) > 0) & (A(i) <= 10)
    sum = sum + A(i)
  end if
end do

```

Several points are illustrated by this pseudocode example. First of all, the statements that can be included with a loop can be arbitrary, comprised of simple statements, loops, and conditionals in any order. This same generality applies to statements within a conditional as well. Secondly, logical expressions can themselves be quite complicated. Finally, note how each level of statements in the program is indented, visually indicated the subordination of statements within higher level loops or conditionals. This stylistic practice lies at the heart of *structured programming*: explicit indication of each statement within the surrounding hierarchy. In modern programming, the structured approach has become the standard because it leads to greater clarity of expression, allowing others to understand the program more quickly and the programmer to find bugs more readily. Employing this style only requires the programmer to use the space key liberally when typing the program. Since sums are computed so often you might expect that a language would provide an intrinsic function to compute it. For F90 and MATLAB you would be correct.

## 1.4.4 Functions

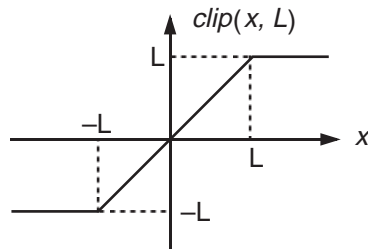
Functions, which define sub-programs having a well-defined interface to other parts of the program, are an essential part of programming languages. For, if properly developed, these functions can be included in future programs, and they allow several programmers to work on complex programs. The function takes an ordered sequence of messages, objects, or variables as its *arguments*, and *returns* to the calling program a value (or set of values) that can be assigned to an object or variable. Familiar examples of a function are the mathematical ones: the `sin` function takes a real-valued argument, uses this value to calculate the trigonometric sine, and returns that value, which can be assigned to a variable.

```
y = sin(x)
```

Note that the argument need not be a variable: a number can be explicitly provided or an expression can be used. Thus, `sin(2.3)` and `sin(2*cos(x))` are all valid. Functions may require more than one argument. For example, the `atan2` function, which computes the arctangent function in such a way that the quadrant of the calculated angle is unambiguous, needs the *x* and *y* components of the triangle.

```
z = atan2(x, y)
```

Note that the order of the arguments—the *x* component must be the first—and the number of arguments—both *x* and *y* are needed—matter for all functions: The calling program's argument ordering



**Figure 1.4:** Input-output relationship for the function  $\text{clip}(x)$ . So long as  $|x| < L$ , this function equals its argument; for larger values, the output equals the clipping constant  $L$  no matter how large the input might be.

and number must agree with those imposed by the function's definition. Said another way, the interface between the two must agree. Analogous to plugs and electric sockets in the home, a three prong plug won't fit into a two-hole socket, and, if you have a two-prong plug, you must plug it in the right way. A function is usually defined separately, outside the body of any program or other function. We call a program's extent its *scope*. In MATLAB, a program's scope is equivalent to what is in a file; in C and C++, scope is defined by brace pairs; and in Fortran, scope equals what occurs between function declaration and its corresponding end statement. Variables are also defined within a program's and a function's scope. What this means is that a variable named  $x$  defined within a function is available to all statements occurring within that function, and different functions can use the same variable name *without any conflict occurring*. What this means is that two functions  $f_1$  and  $f_2$  can each make use of a variable named  $x$ , and the value of  $x$  depends on which function is being referred to. In technical terms, the scope of every variable is limited to its defining function. At first, this situation may seem terribly confusing ("There are two variables both of which are named  $x$ ?"); further thought brings the realization that this convention is what you want. Because each function is to be a routine—a program having a well-defined interface, execution of the function's internal statements must not depend on the program that uses it. This convention becomes especially important when different people write the programs or functions. Thus, such local variables—those defined locally within a function—do not conflict, and they are stored in different memory locations by the compiler or interpreter.

This limited scope convention can be countermanded when you explicitly declare variables to be *global*. Such variables are now potentially available to all functions, and each function cannot define a variable having the same name. For example, you may well want a variable pointedly named `pi` to be available to all functions; you can do so by declaring it to be a global variable. To demonstrate scope, consider the following simple example. Here, we want to clip the values stored in the array  $x$  and store the results in the array  $y$ .

*Main Pseudocode Program*

```
! Clip the elements of an array
limit = 3
do i=1,n
  y(i) = clip(x(i), limit)
end do
```

*Function Pseudocode Definition*

```
! function clip(x, edge)
! x - input variable
! edge - location of breakpoint
function clip(x, edge)
  if abs(x) > edge then
    y = sign(x)*edge
  else
    y = x
  end if
end
```

The clipping function has the generic form show in Figure 1.4. Thus, values of the argument that are less than  $L$  in magnitude are not changed, while those exceeding this limit are set equal to the limiting value. In the program example, note that the name of the array in the calling program— $x$ —is the same as the argument's name used in the definition of the function. Within the scope of a program or function, an array and a scalar variable cannot have the same name. In our case, because each variable's scope is limited to the function or program definition, no conflict occurs: Each is well defined and the meaning should be unambiguous. Also note that the second argument has a different name in the program than in the function. No matter how the arguments are defined, we say that they are *passed* to the function, with

the function's variables set equal to values specified in the calling program. These interface rules allows the function to be used in other programs, which means that we can reuse functions whenever we like!

### 1.4.5 Modules

Another important programming concept is that of packaging a group of related routines and/or selective variables into a larger programming entity. In the Ada language they are called *packages*, while C++ and MATLAB call them *classes*. F90 has a generalization of this concept that it calls a *module*. As we will see later the F90 module includes the functionality of the C++ classes, as well as other uses such as defining global constants. Therefore, we will find the use of F90 modules critical to our object-oriented programming goals. In that context modules provide us with the means to take several routines related to a specific data type and to encapsulate them into a larger programming unit that has the potential to be reused for more than one application.

### 1.4.6 Dynamic Memory Management

From the very beginning, several decades ago, there was a clear need to be able to dynamically allocate and deallocate segments of memory for use by a program. The initial standards for Fortran did not allow for this. It was necessary to invoke machine language programs to accomplish that task or to write tools to directly manage arrays by defining “pseudo-pointers” to manually move things around in memory or to overwrite space that was no longer needed. It was very disappointing that the F77 standard failed to offer that ability, although several “non-standard” compilers offered such an option. Beginning with the F90 standard a full set of dynamic memory management abilities is now available within Fortran. Dynamic memory management is mainly needed for arrays and pointers. Both of these will be considered later, with a whole chapter devoted to arrays. Both of these entities can be declared as `ALLOCATABLE` and later one will `ALLOCATE` and then `DEALLOCATE` them. There are also new “automatic arrays” that have the necessary memory space supplied and then freed as needed.

Pointers are often used in “data structures”, abstract data types, and objects. To check on the status of such features one can invoke the `ALLOCATED` intrinsic and use `ASSOCIATED` to check on the status of pointers and apply `NULLIFY` to pointers that need to be freed or initialized. Within F90 allocatable arrays cannot be used in the definitions of derived types, abstract data types, or objects. However, allocatable pointers to arrays can be used in such definitions. To assist in creating efficient executable codes, entities that might be pointed at by a pointer must have the `TARGET` attribute.

Numerous examples of dynamic memory management will be seen later. Persons that write compilers suggest that, in any language, it is wise to deallocate dynamic memory in the reverse order of its creation. The F90 language standard does not require that procedure but you see that advice followed in most of the examples.

## 1.5 Program evaluation and testing

Your fully commented program, written with the aid of an *editor*, must now come alive and be translated into another language that more closely matches computer instructions; it must be *executed* or *run*. Statements expressed in MATLAB, Fortran, or C++ may not directly correspond to computational instructions. However, the Fortran syntax was designed to more clearly match mathematical expressions. These languages are designed to allow humans to define computations easily and also allow easy translation. Writing programs in these languages provides some degree of *portability*: A program can be executed on very different computers without modification. So-called *assembly languages* allow more direct expression of program execution, but are very computer specific. Programmers that write in assembly language must worry about the exquisite details of computer organization, so much so that writing of what the computation is doing takes much longer. What they produce might run more rapidly than the same computation expressed in Fortran, for example, but no portability results and programs become incredibly hard to debug.

Programs become executable machine instructions in two basic ways. They are either *interpreted* or *compiled*. In the first case, an interpreter reads your program and translates it to executable instructions “on the fly.” Because interpreters essentially examine programs on a line-by-line basis, they usually allow instructions accept typed user instructions as well as fully written programs. MATLAB is an example of

an interpreter.<sup>†</sup> It can accept typed commands created as the user thinks of them (plot a graph, see that a parameter must have been typed incorrectly, change it, and replot, for example) or entire programs. Because interpreters examine programs locally (line-by-line), program execution tends to be slower than when a compiler is used.

Compilers are programs that take your program in its entirety and produce an executable version of it. Compiler output is known as an *executable* file that, in UNIX for example, can become a command essentially indistinguishable from others that are available. C++ is an example of a language that is frequently compiled rather than interpreted. Compilers will produce much more efficient (faster running) programs than interpreters, but if you find an error, you must edit and re-compile before you can attempt execution again. Because compilation takes time, this cycle can be time-consuming if the program is large.

Interpreters are themselves executable files written in compiled languages: MATLAB is written in C. Executable programs produced by compilers are stand-alone programs: *Everything*—user input and output, file reading, etc.—must be handled by the user’s program. In an interpreter, you can supplement a program’s execution by typed instructions. For example, in an interpreter you can type a simple command to make the variable `a` equal to 1; in a compiled program, you must include a program that asks for the value of `a`. Consequently, users frequently write programs in the context of an interpreter, understand how to make the program better by interacting with it, and then re-express it in a compiled language.

Both interpreters and compilers make extensive use of what are known as *library* commands or functions. A natural example of a library function is the `sin` function: Users typically do not want to program explicitly the computation of the trigonometric sine function. Instead, they want to be able to pull it “off the shelf” and use as need be. Library modules are just programs written in a computer language like you would write. Consequently, both interpreters and compilers allow user programs to become part of the library, which is usually written by many programmers over a long period of time. It is through modules available in a library that programming teams cooperate. Library modules tend to be more extensive and do more things in an interpreter. For example, MATLAB provides a program that produces pseudo-three-dimensional plots of functions. Such routines usually do not come with a compiler, but may be purchased separately from graphics programming specialists. For compiled languages, we refer to *linking* the library routines to the user’s program (in interpreters, this happens as a matter of course). A linker is a program that takes modules produced by the compiler, be they yours or others, associates the modules, and produces the executable file we mentioned earlier. Most C++ compilers “hide” the linking step from you; you may think you are typing just the command to compile the program, but it is actually performing that step for you. When you are compiling a module not intended for stand-alone execution, a compiler option that you type can prevent the compiler from performing the linking step.

*Debugging* is the process of discovering and removing program errors. Two main types of errors occur in writing programs: what we would generally term “typos” and what are design errors. The first kind may be readily found (where is the function `sni`?) or more subtle (you type `aa` instead of `a` for a variable’s name and `aa` also exists!). The second kind of error can be hard or subtle to find. The main components of this process are

1. Search the program module by eye as you do a “mental run through” of its task. This kind of error searching begins when you first think about program organization, and continues as you refine the program. Why write a program that is logically flawed?
2. If written in a compiled language, compile the program to find syntax errors or warnings about unused or undefined variables. If in an interpreted language, attempt preliminary execution to obtain similar error messages. Linking also can locate modules or libraries that are improperly referenced.
3. Running the executable file with typical data sets often causes the program to abort—a harsh word that expresses the situation where the program goes crazy and ceases to behave—and the system to supply an error message, such as division by zero. Error messages *may* help locate the programming error.

---

<sup>†</sup>This statement is only partially true. MATLAB does have some features of a compiler, like looking ahead to determine if interface errors exist with respect to functions called by the main program.



Easy errors to find are *syntactic* errors: You have violated the language's rules of what a well-formed program must be. Usually, the interpreter or compiler complain bitterly on encountering a syntax error. Compilers find these at compile time (when the program is compiled), interpreters at run time. Design errors are only revealed when we supply the program with data. The programmer should design test data that exercises each of the program's valid operations. Invalid user input (asking for the logarithm of a negative number, for example) should be caught by the program and warning messages sent to the user.

The previous description of generic programming languages indicates why finding bugs can be quite complicated. Programs can exhibit quite complex behaviors, and tracing incorrect behaviors can be correspondingly difficult. One often hears the (true) statement "Computers do what we say, not what we want." Users frequently want computers to be smart, fixing obvious design (mental) errors because they obviously conflict with what we want. However, this situation is much like what the novelist faces. Inexact meaning can confuse the reader; he or she does not have a direct pathway to the novelist's mind. As opposed to the novelist, extensive testing of your program can detect such errors and make your program approach perfection. Many operating systems supply interactive *debugger* programs that can trace the execution of a program in complete detail. They can display the values of any variable, stop at selected positions for evaluation, execute parts of the code in a statement-by-statement fashion, etc. These can be very helpful in finding difficult-to-locate bugs, but they still cannot read your mind.

Be that as it may, what can the programmer do when the program compiles (no syntactic errors), doesn't cause system error messages (no dividing by zero), but the results are *not* correct? The simplest approach is to include extra statements in your program, referred to as debugging statements, that display (somewhat verbosely) values of internal variables. For example, in a loop you would print the value of the loop index and all variables that the loop might be affecting. Because this output can be voluminous, the most fruitful approach is to debug smaller problems. With this debugging information, you can usually figure out the error, correct it, *and* change the comments accordingly. Without the latter, your program and your internal documentation are out-of-sync.

Once debugged, you could delete the debugging statements you added. A better approach is to just hide them. You can do this two ways: Comment them out or encase them in a conditional that is true when the program is in "debugging mode." The commenting approach completely removes the debugging statements from the program execution stream, and allows you to easily put them back if further program elaborations result in errors. The use of conditionals does put an overhead on computational efficiency, but usually a small one.

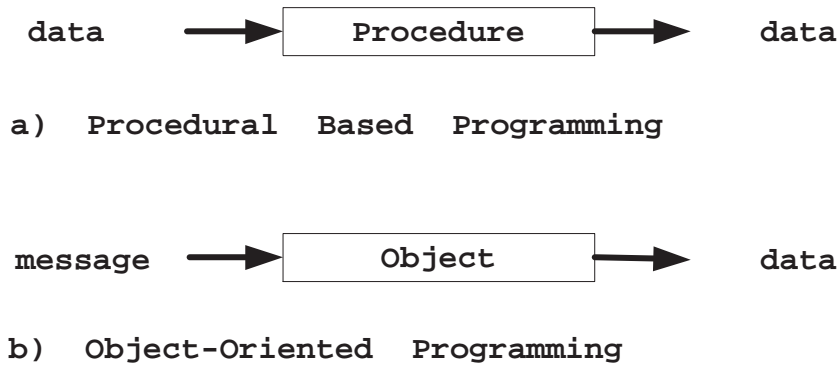
## 1.6 Program documentation

Comments inside a program are intended to help you and others understand program design and how it is organized. Frequently, comments describe what each variable means, how program execution is to proceed, and what each module's interface might be (what are the expected inputs and their formats, and what outputs are produced). Program comments occur in the midst of the program's source, and temporarily interrupts the highly restricted syntax of most programming languages. Comments are entirely ignored by the interpreter or compiler, and are allowed to enhance program clarity for humans.

*Documentation* includes program comments, but also includes external prose that describes what the program does, how the user interface controls program behavior, and how the display of results means. Making an executable program available to users does not help them understand how to use it. In UNIX, *all* provided commands are accompanied by what are referred to as *manual pages*: concise descriptions of what the program does, all user options, and descriptions of what error messages means. *Programs are useless without such documentation.* Many programs provide such documentation whenever the user types something that clearly indicates a lack of knowledge about how to use the program. This kind of documentation must also be supplemented by prose that a user can read. Professional programmers frequently write the documentation as the program is being designed. This simultaneous development of the program and documentation of how it is used often uncovers user interface design flaws.

## 1.7 Object Oriented Formulations

The above discussion of subprograms follows the older programming style where the emphasis is placed on the procedures that a subprogram is to apply to the supplied data. Thus, it is referred to as *procedural programming*. The alternate approach focuses on the data and its supporting functions, and is known as an *object oriented* approach and is the main emphasis of this work. It also generalizes the concept of data types and is usually heavily dependent on user defined data types and their extension to abstract data types. These concepts are sketched in Fig. 1.5.



**Figure 1.5:** Two Approaches to Programming

The process of creating an “object-oriented” (OO) formulation involves at least three stages: Object-Oriented Analysis (OOA), Object-Oriented Design (OOD), and Object-Oriented Programming (OOP). Many books have been written on each of these three subjects. Formal graphical standards for representing the results of OOA and OOD have been established and are widely used in the literature. Here the main emphasis will be placed on OOP on the assumption that the two earlier stages have been completed. In an effort to give some level of completeness, summaries of OOA and OOD procedures are given in Tables 1–1 and 1–2, respectively. Having completed OOA and OOD studies one must select a language to actually implement the design. More than 100 object-oriented languages are in existence and use today. They include “pure” OO languages like Crisp, Eiffel, Rexx, Simula, Smalltalk, etc. and “hybrid” OO languages like C++ , F90 , Object Pascal, etc. In which of them should you invest your time? To get some insight into answers to this question we should study the advice of some of the recognized leaders in the field. In his 1988 book on OO software construction B. Myers listed seven steps necessary to achieve object-orientedness in an implementation language. They are summarized in Table 1-3 and are all found to exist in F90 and F95 . Thus we proceed with F90 as our language of choice. The basic F90 procedures for OOP will be illustrated in some short examples in Chapter 3 after covering some preliminary material on abstract data types in Chapter 2. Additional OOP applications will also be covered in later chapters.

**Table 1–1. OO Analysis Summary**

Find objects and classes :

- Create an abstraction of the problem domain.
- Give attributes, behaviors, classes, and objects meaningful names.
- Identify structures pertinent to the system's complexity and responsibilities.
- Observe information needed to interact with the system, as well as information to be stored.
- Look for information re-use; are there multiple structures; can sub-systems be inherited?

Define the attributes :

- Select meaningful names.
- Describe the attribute and any constraints.
- What knowledge does it possess or communicate?
- Put it in the type or class that best describes it.
- Select accessibility as public or private.
- Identify the default, lower and upper bounds.
- Identify the different states it may hold.
- Note items that can either be stored or re-computed.

Define the behavior :

- Give the behaviors meaningful names.
- What questions should each be able to answer?
- What services should it provide?
- Which attribute components should it access?
- Define its accessibility (public or private).
- Define its interface prototype.
- Define any input/output interfaces.
- Identify a constructor with error checking to supplement the intrinsic constructor.
- Identify a default constructor.

Diagram the system :

- Employ an OO graphical representation such as the Coad/Yourdon method or its extension by Graham.

**Table 1–2.** OO Design Summary

- Improve and add to the OOA results during OOD.
- Divide the member functions into constructors, accessors, agents and servers.
- Design the human interaction components.
- Design the task management components.
- Design the data management components.
- Identify operators to be overloaded.
- Identify operators to be defined.
- Design the interface prototypes for member functions and for operators.
- Design code for re-use through “kind of” and “part of” hierarchies.
- Identify base classes from which other classes are derived.
- Establish the exception handling procedures for all possible errors.

**Table 1–3.** 7 Steps to Object-Orientedness (B. Myer, 1988)

1. Object-based modular structure :

- Systems are modularized on the basis of their data structure (in F90).

2. Data Abstraction :

- Objects should be described as implementations of abstract data types (in F90).

3. Automatic memory management :

- Unused objects should be deallocated by the language system (most in F90, in F95).

4. Classes :

- Every non-simple type is a module, and every high-level module is a type (in F90).

5. Inheritance :

- A class may be defined as an extension or restriction of another (in F90).

6. Polymorphism and dynamic binding :

- Entities are permitted to refer to objects of more than one class and operations can have different realizations in different classes (partially in F90/F95, expected in Fortran 2000).

7. Multiple and repeated inheritance :

- Can declare a class as heir to more than one class, and more than once to the same class (in F90).

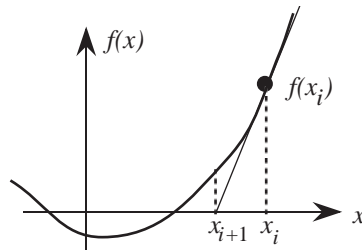
## 1.8 Exercises

### 1 Checking trigonometric identities

We know that the sine and cosine functions obey the trigonometric identity  $\sin^2 \theta + \cos^2 \theta = 1$  no matter what value of  $\theta$  is used. Write a pseudocode, or MATLAB, or F90 program that checks this identity. Let it consist of a loop that increments across  $N$  equally spaced angles between 0 and  $\pi$ , and calculates the quantity in question, printing the angle and the result. Test your program for several values of  $N$ . (Later we will write a second version of this program that does not contain *any* analysis loops, using instead MATLAB's, or F90's, ability to calculate functions of arrays.)

### 2 Newton-Raphson algorithm

A commonly used numerical method of solving the equation  $f(x) = 0$  has its origins with the beginnings of calculus. Newton noted that the slope of a function tended to cross the  $x$ -axis near a function's position of zero value (called a *root*).



Because the function's slope at some point  $x_i$  equals its derivative  $f'(x_i)$ , the equation of the line passing through  $f(x_i)$  is  $f'(x_i)x + (f(x_i) - f'(x_i)x_i)$ . Solving for the case when this expression equals the next trial root  $x_{i+1}$ .

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

The algorithm proceeds by continually applying this iterative equation until the error is "small." The definition of "small" is usually taken to mean that the absolute relative difference between successive iterates is less than some tolerance value  $\epsilon$ . (Raphson extended these concepts to an array of functions.)

- (a) In pseudocode, write a program that performs the Newton-Raphson algorithm. Assume that functions that evaluate the function and its derivative are available. What is the most convenient form of loop to use in your program?
- (b) Translate your pseudocode into F90, or MATLAB, and apply your program to the simple function  $f(x) = e^{2x} - 5x - 1$ . Use the functional expressions directly in your program or make use of functions.

### 3 Game of Life pseudocode

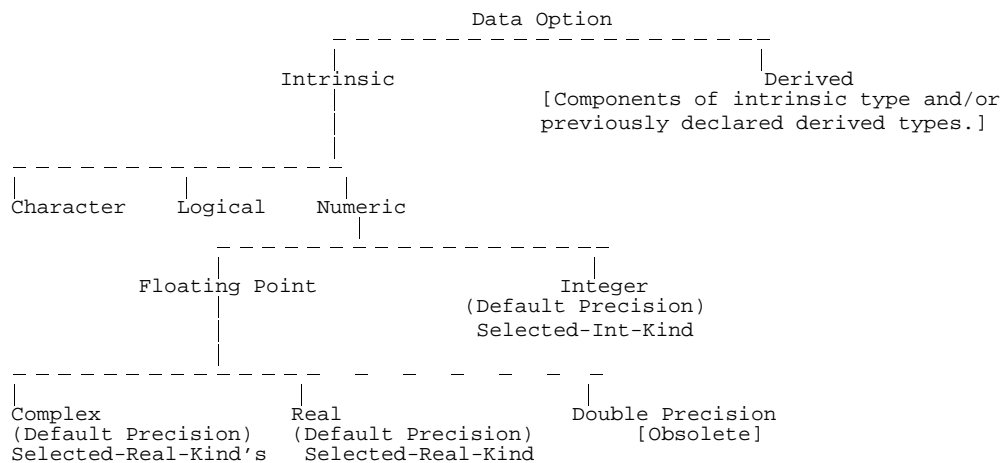
Develop a pseudocode outline for the main parts of the "Game of Life" which was discussed earlier and shown in Fig. 1.3. Include pseudocode for a function to compute the next generation.

## Chapter 2

# Data Types

Any computer program is going to have to operate on the available data. The valid data types that are available will vary from one language to another. Here we will examine the intrinsic or built-in data types, user-defined data types or structures and, finally, introduce the concept of the abstract data type which is the basic foundation of object-oriented methods. We will also consider the precision associated with numerical data types. The Fortran data types are listed in Table 2–1. Such data can be used as constants, variables, pointers and targets.

**Table 2–1.** F90/95 Data Types and Pointer Attributes



## 2.1 Intrinsic Types

The simplest data type is the `LOGICAL` type which has the Boolean values of either `.true.` or `.false.` and is used for relational operations. The other non-numeric data type is the `CHARACTER`. The sets of valid character values will be defined by the hardware system on which the compiler is installed. Character sets may be available in multiple languages, such as English and Japanese. There are international standards for computer character sets. The two most common ones are the English character sets defined in the ASCII and EBCDIC standards that have been adapted by the International Standards Organization (ISO). Both of these standards for defining single characters include the digits (0 to 9), the 26 upper case letters (A to Z), the 26 lower case letters (a to z), common mathematical symbols, and many non-printable codes known as control characters. We will see later that strings of characters are still referred to as being of the `CHARACTER` type, but they have a length that is greater than one. In other languages such a data type is often called a *string*. [While not part of the F95 standard, the ISO Committee created a user-defined type known as the `ISO_VARIABLE_LENGTH_STRING` which is available as a F95 source module.]

For numerical computations, numbers are represented as integers or decimal values known as *floating point numbers* or *floats*. The former is called an `INTEGER` type. The decimal values supported in Fortran are the `REAL` and `COMPLEX` types. The range and precision of these three types depends on the hardware being employed. At the present, 1999, most computers have 32 bit processors, but some offer 64 bit processors. This means that the precision of a calculated result from a single program could vary from one brand of computer to another. One would like to have a portable precision control so as to get the same answer from different hardware; whereas some languages, like C++, specify three ranges of precision (with specific bit widths). Fortran provides default precision types as well as two functions to allow the user to define the “kind” of precision desired.

**Table 2–2.** Numeric Types on 32 Bit Processors

<i>Type</i>	<i>Bit Width</i>	<i>Significant Digits</i>	<i>Common Range</i>
integer	16	10	–32,768 to 32,767
real	32	6	$-10^{37}$ to $10^{37}$
double precision <sup>†</sup>	64	15	$-10^{307}$ to $10^{307}$
complex	2@32	2@6	two reals

<sup>†</sup>obsolete in F90, see `selected_real_kind`

Still, it is good programming practice to employ a precision that is of the default, double, or quad precision level. Table 2–2 lists the default precisions for 32 bit processors. The first three entries correspond to types *int*, *float*, and *double*, respectively, of C++. Examples of F90 integer constants are

–32      0      4675123      24\_short      24\_long

while typical real constant examples are

–3.      0.123456      1.234567e+2      0.0      0.3\_double  
7.6543e+4\_double      0.23567\_quad      0.3d0

In both cases, we note that it is possible to impose a user-defined precision kind by appending an underscore ( `_` ) followed by the name of the integer variable that gives the precision kind number. For example, one could define

```
long = selected_int_kind(9)
```

to denote an integer in the range of  $-10^9$  to  $10^9$ , while

```
double = selected_real_kind(15,307)
```

defines a real with 15 significant digits with an exponent range of  $\pm 307$ . Likewise, a higher precision real might be defined by the integer kind

```
quad = selected_real_kind(18,4932)
```

to denote 18 significant digits over the exponent range of  $\pm 4932$ . If these kinds of precision are available on your processors, then the F90 types of “integer (long),” “real (double),” and “real (quad)” would correspond to the C++ precision types of “long int,” “double,” and “long double,” respectively. If the processor cannot produce the requested precision, then it returns a negative number as the integer kind number. Thus, one should always check that the kind (i.e., the above integer values of long, double, or quad) is not negative, and report an exception if it is negative.

The old F77 intrinsic type of `DOUBLE PRECISION` has been declared obsolete, since it is now easy to set any level of precision available on a processor. Another way to always define a double precision real on any processor is to use the “kind” function such as

```
double = kind(1.0d0)
```

where the symbol ‘d’ is used to denote the I/O of a double precision real. For completeness it should be noted that it is possible on some processors to define different kinds of character types, such as “greek” or “ascii”, but in that case, the kind value comes before the underscore and the character string such as: `ascii_“a string”`.

```

[ 1] Module Math_Constants      ! Define double precision math constants
[ 2]   implicit none
[ 3]   ! INTEGER, PARAMETER :: DP = SELECTED_REAL_KIND (15,307)
[ 4]   INTEGER, PARAMETER :: DP = KIND (1.d0) ! Alternate form
[ 5]   real(DP), parameter:: Deg_Per_Rad  = 57.295779513082320876798155_DP
[ 6]   real(DP), parameter:: Rad_Per_Deg  = 0.017453292519943295769237_DP
[ 7]
[ 8]   real(DP), parameter:: e_Value      = 2.71828182845904523560287_DP
[ 9]   real(DP), parameter:: e_Recip     = 0.3678794411714423215955238_DP
[10]   real(DP), parameter:: e_Squared   = 7.389056098930650227230427_DP
[11]   real(DP), parameter:: Log10_of_e  = 0.4342944819032518276511289_DP
[12]
[13]   real(DP), parameter:: Euler       = 0.5772156649015328606_DP
[14]   real(DP), parameter:: Euler_Log   = -0.5495393129816448223_DP
[15]   real(DP), parameter:: Gamma       = 0.577215664901532860606512_DP
[16]   real(DP), parameter:: Gamma_Log   = -0.54953931298164482237662_DP
[17]   real(DP), parameter:: Golden_Ratio = 1.618033988749894848_DP
[18]
[19]   real(DP), parameter:: Ln_2        = 0.6931471805599453094172321_DP
[20]   real(DP), parameter:: Ln_10       = 2.3025850929940456840179915_DP
[21]   real(DP), parameter:: Log10_of_2  = 0.3010299956639811952137389_DP
[22]
[23]   real(DP), parameter:: pi_Value    = 3.141592653589793238462643_DP
[24]   real(DP), parameter:: pi_Ln       = 1.144729885849400174143427_DP
[25]   real(DP), parameter:: pi_Log10    = 0.4971498726941338543512683_DP
[26]   real(DP), parameter:: pi_Over_2   = 1.570796326794896619231322_DP
[27]   real(DP), parameter:: pi_Over_3   = 1.047197551196597746154214_DP
[28]   real(DP), parameter:: pi_Over_4   = 0.7853981633974483096156608_DP
[29]   real(DP), parameter:: pi_Recip     = 0.3183098861837906715377675_DP
[30]   real(DP), parameter:: pi_Squared  = 9.869604401089358618834491_DP
[31]   real(DP), parameter:: pi_Sq_Root  = 1.772453850905516027298167_DP
[32]
[33]   real(DP), parameter:: Sq_Root_of_2 = 1.4142135623730950488_DP
[34]   real(DP), parameter:: Sq_Root_of_3 = 1.7320508075688772935_DP
[35]
[36] End Module Math_Constants
[37]
[38] Program Test
[39]   use Math_Constants      ! Access all constants
[40]   real :: pi              ! Define local data type
[41]   print *, 'pi_Value: ', pi_Value      ! Display a constant
[42]   pi = pi_Value; print *, 'pi = ', pi  ! Convert to lower precision
[43] End Program Test          ! Running gives:
[44]   ! pi_Value: 3.1415926535897931      ! pi = 3.14159274

```

**Figure 2.1:** Defining Global Double Precision Constants

To illustrate the concept of a defined precision intrinsic data type, consider a program segment to make available useful constants such as  $\pi$  (3.1415...) or Avogadro's number ( $6.02 \dots \times 10^{23}$ ). These are real constants that should not be changed during the use of the program. In F90, an item of that nature is known as a `PARAMETER`. In Fig. 2.1, a selected group of such constants have been declared to be of double precision and stored in a `MODULE` named `Math_Constants`. The parameters in that module can be made available to any program one writes by including the statement “`use math_constants`” at the beginning of the program segment. The figure actually ends with a short sample program that converts the tabulated value of  $\pi$  (line 23) to a default precision real (line 42) and prints both.

## 2.2 User Defined Data Types

While the above intrinsic data types have been successfully employed to solve a vast number of programming requirements, it is logical to want to combine these types in some structured combination that represents the way we think of a particular physical object or business process. For example, assume we wish to think of a chemical element in terms of the combination of its standard symbol, atomic number and atomic mass. We could create such a data structure type and assign it a name, say `chemical_element`, so that we can refer to that type for other uses just like we might declare a real variable. In F90 we would define the structure with a `TYPE` construct as shown below (in lines 3–7):

```

[ 1] program create_a_type
[ 2]   implicit none
[ 3]   type chemical_element      ! a user defined data type
[ 4]     character (len=2) :: symbol
[ 5]     integer          :: atomic_number
[ 6]     real              :: atomic_mass

```



```
[ 7]     end type
```

Having created the new data type, we would need ways to define its values and/or ways to refer to any of its components. The latter is accomplished by using the component selection symbol “%”. Continuing the above program segment we could write:

```
[ 8]     type (chemical_element) :: argon, carbon, neon      ! elements
[ 9]     type (chemical_element) :: Periodic_Table(109)      ! an array
[10]     real                      :: mass                   ! a scalar
[11]
[12]     carbon%atomic_mass = 12.010                        ! set a component value
[13]     carbon%atomic_number = 6                          ! set a component value
[14]     carbon%symbol = "C"                               ! set a component value
[15]
[16]     argon = chemical_element ("Ar", 18, 26.98) ! construct element
[17]
[18]     read *, neon                                         ! get "Ne" 10 20.183
[19]
[20]     Periodic_Table( 5) = argon                          ! insert element into array
[21]     Periodic_Table(17) = carbon                          ! insert element into array
[22]     Periodic_Table(55) = neon                            ! insert element into array
[23]
[24]     mass = Periodic_Table(5) % atomic_mass               ! extract component
[25]
[26]     print *, mass                                         ! gives 26.9799995
[27]     print *, neon                                         ! gives Ne 10 20.1830006
[28]     print *, Periodic_Table(17)                         ! gives C  6  12.0100002
[29] end program create_a_type
```

In the above program segment, we have introduced some new concepts:

- define argon, carbon and neon to be of the `chemical_element` type (line 7).
- define an array to contain 109 `chemical_element` types (line 8).
- used the selector symbol, %, to assign a value to each of the components of the carbon structure (line 15).
- used the intrinsic “structure constructor” to define the argon values (line 15). The intrinsic construct or initializer function must have the same name as the user-defined type. It must be supplied with all of the components, and they must occur in the order that they were defined in the `TYPE` block.
- read in all the neon components, in order (line 17). [The “\*” means that the system is expected to automatically find the next character, integer and real, respectively, and to insert them into the components of neon.]
- inserted argon, carbon and neon into their specific locations in the periodic table array (lines 19–21).
- extracted the `atomic_mass` of argon from the corresponding element in the `periodic_element` array (line 23).
- print the real variable, `mass` (line 25). [The “\*” means to use a default number of digits.]
- printed all components of neon (line 26). [Using a default number of digits.]
- printed all the components of carbon by citing its reference in the periodic table array (line 27). [Note that the printed real value differs from the value assigned in line 12. This is due to the way reals are represented in a computer, and will be considered elsewhere in the text.]

A defined type can also be used to define other data structures. This is but one small example of the concept of code re-use. If we were developing a code that involved the history of chemistry, we might use the above type to create a type called *history* as shown below.

```
type (chemical_element) :: oxygen

type history
character (len=31)      :: element_name
integer                :: year_found
type (chemical_element) :: chemistry
```

```

end type history

type (history) :: Joseph_Priestley           ! Discoverer
oxygen = chemical_element ("O", 76, 190.2)   ! construct element
Joseph_Priestley = history ("Oxygen", 1774, oxygen) ! construct
print *, Joseph_Priestley ! gives Oxygen 1774 O 76 1.9020000E+02

```

Shortly we will learn about other important aspects of user-defined types, such as how to define operators that use them as operands.

## 2.3 Abstract Data Types

Clearly, data alone is of little value. We must also have the means to input and output the data, subprograms to manipulate and query the data, and the ability to define operators for commonly used procedures. The coupling or encapsulation of the data with a select group of functions that defines everything that can be done with the data type introduces the concept of an abstract data type (ADT). An ADT goes a step further in that it usually hides from the user the details of how functions accomplish their tasks. Only knowledge of input and output interfaces to the functions are described in detail. Even the components of the data types are kept private.

The word *abstract* in the term *abstract data type* is used to: 1) indicate that we are interested only in the essential features of the data type, 2) to indicate that the features are defined in a manner that is independent of any specific programming language, and 3) to indicate that the instances of the ADT are being defined by their behavior, and that the actual implementation is secondary. An ADT is an abstraction that describes a set of items in terms of a hidden or encapsulated data structure and a set of operations on that data structure.

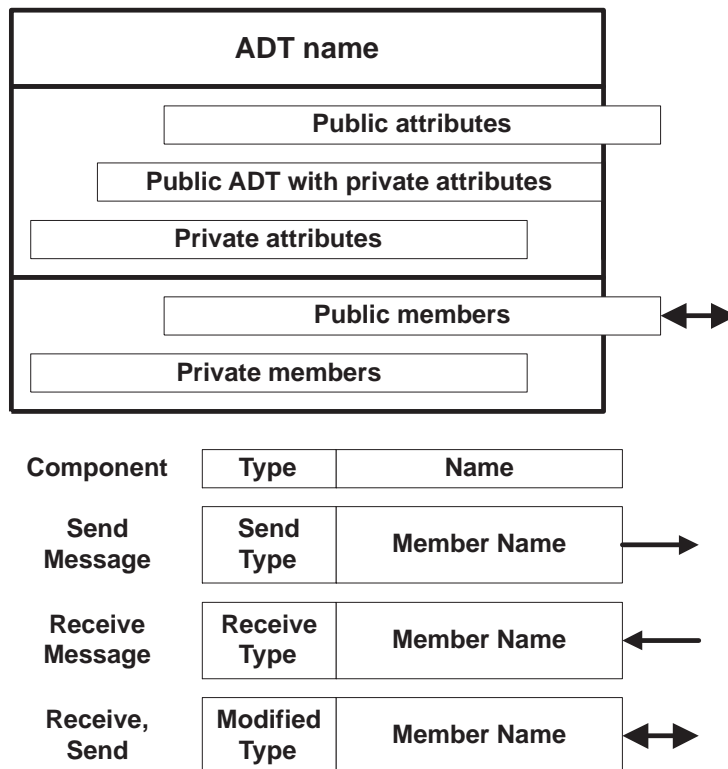
Previously we created user-defined entity types such as the `chemical_element`. The primary difference between entity types and ADTs is that all ADTs include methods for operating on the type. While entity types are defined by a name and a list of attributes, an ADT is described by its name, attributes, encapsulated methods, and possibly encapsulated rules.

Object-oriented programming is primarily a data abstraction technique. The purpose of abstraction and data hiding in programming is to separate behavior from implementation. For abstraction to work, the implementation must be encapsulated so that no other programming module can depend on its implementation details. Such encapsulation guarantees that modules can be implemented and revised independently. Hiding of the attributes and some or all of the methods of an ADT is also important in the process. In F90 the `PRIVATE` statement is used to hide an attribute or a method; otherwise, both will default to `PUBLIC`. Public methods can be used outside the program module that defines an ADT. We refer to the set of public methods or operations belonging to an ADT as the public interface of the type.

The user-defined data type, as given above, in F90 is not an ADT even though each is created with three intrinsic methods to construct a value, read a value, or print a value. Those methods cannot modify a type; they can only instantiate the type by assigning it a value and display that value. (Unlike F90, in C or C++ a user-defined type, or “struct”, does not have an intrinsic constructor method, or input/output methods.) Generally ADTs will have methods that modify or query a type’s state or behavior.

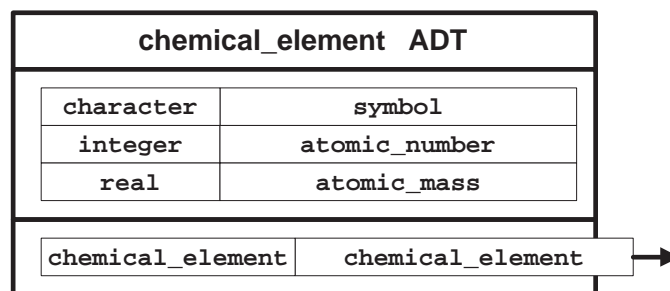
From the above discussion we see that the intrinsic data types in any language (such as complex, integer and real in F90 ) are actually ADTs. The system has hidden methods (operators) to assign them values and to manipulate them. For example, we know that we can multiply any one of the numerical types by any other numerical type.

We do not know how the system does the multiplication, and we don’t care. All computer languages provide functions to manipulate the intrinsic data types. For example, in F90 a square root function, named *sqr*t, is provided to compute the square root of a real or complex number. From basic mathematics you probably know that two distinctly different algorithms must be used and the choice depends on the type of the supplied argument. Thus, we call the *sqr*t function a generic function since its single name, *sqr*t, is used to select related functions in a manner hidden from the user. In F90 you can not take the square root of an integer; you must convert it to a real value and you receive back a real answer. The



**Figure 2.2:** Graphical Representation of ADTs

above discussions of the methods (routines) that are coupled to a data type and describe what you can and can not do with the data type should give the programmer good insight into what must be done to plan and implement the functions needed to yield a relatively complete ADT.

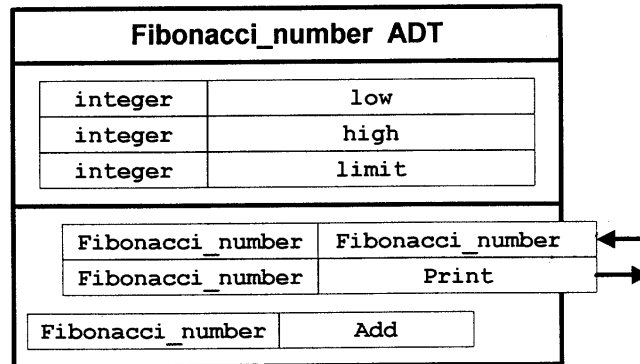


**Figure 2.3:** Representation of the Public Chemical \_Element ADT

It is common to have a graphical representation of the ADTs and there are several different graphical formats suggested in the literature. We will use the form shown in Fig. 2.4 where a rectangular box begins with the ADT name and is followed by two partitions of that box that represent the lists of attribute data and associated member routines. Items that are available to the outside world are in sub-boxes that cross over the right border of the ADT box. They are the parts of the public interface to the ADT. Likewise those items that are strictly internal, or private, are contained fully within their respective partitions of the ADT box. There is a common special case where the name of the data type itself is available for external use, but its individual attribute components are not. In that case the right edge of the private attributes lists lie on the right edge of the ADT box. In addition, we will often segment the smallest box for an item to give its type (or the most important type for members) and the name of the item. Public

member boxes are also supplemented with an arrow to indicate which take in information (<--), or send out information (-->). Such a graphical representation of the previous `chemical_element` ADT, with all its items public, is shown in Fig. 2.4.

The sequence of numbers known as Fibonacci numbers is the set that begins with one and two and where the next number in the set is the sum of the two previous numbers (1, 2, 3, 5, 8, 13, ...). A primarily private ADT to print a list of Fibonacci numbers up to some limit is represented graphically in Fig. 2.5.

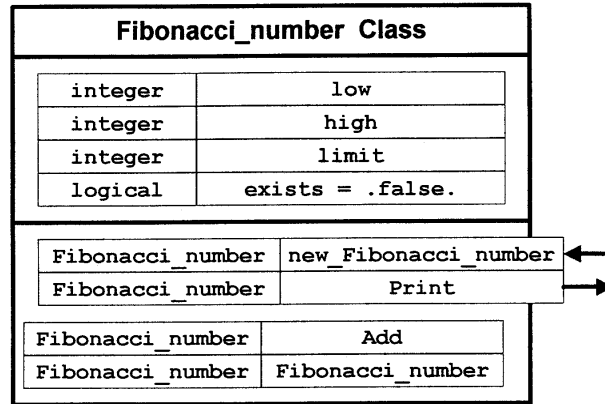


**Figure 2.4:** Representation of a `Fibonacci_Number` ADT

## 2.4 Classes

A class is basically the extension of an ADT by providing additional member routines to serve as *constructors*. Usually those additional members should include a *default constructor* which has no arguments. Its purpose is to assure that the class is created with acceptable default values assigned to all its data attributes. If the data attributes involve the storage of large amounts of data (memory) then one usually also provides a *destructor* member to free up the associated memory when it is no longer needed. F95 has an automatic deallocation feature which is not present in F90 and thus we will often formally deallocate memory associated with data attributes of classes.

As a short example we will consider an extension of the above `Fibonacci_Number` ADT. The ADT for Fibonacci numbers simply keeps up with three numbers (low, high, and limit). Its intrinsic initializer has the (default) name `Fibonacci`. We generalize that ADT to a class by adding a constructor named `new_Fibonacci_number`. The constructor accepts a single number that indicates how many values in the infinite list we wish to see. It is also a default constructor because if we omit the one optional argument it will list a minimum number of terms set in the constructor. The graphical representation of the `Fibonacci_Number` class extends Fig. 2.4 for its ADT by at least adding one public constructor, called `new_Fibonacci_number`, as shown in Fig. 2.5. Technically, it is generally accepted that a constructor should only be able to construct a specific object once. This differs from the intrinsic initializer which could be invoked multiple times to assign different values to a single user-defined type. Thus, an additional logical attribute has been added to the previous ADT to allow the constructor, `new_Fibonacci_number`, to verify that it is being invoked only once for each instance of the class. The coding for this simple class is illustrated in Fig. 2.6. There the access restrictions are given on lines 4, 5, and 7 while the attributes are declared on line 8 and the member functions are given in lines 13-33. The validation program is in lines 36-42, with the results shown as comments at the end (lines 44-48).



**Figure 2.5:** Representation of a Fibonacci\_Number Class

```

[ 1] ! Fortran 90 OOP to print list of Fibonacci Numbers
[ 2] Module class_Fibonacci_Number      ! file: Fibonacci_Number.f90
[ 3]   implicit none
[ 4]   public :: Print                  ! member access
[ 5]   private :: Add                  ! member access
[ 6]   type Fibonacci_Number           ! attributes
[ 7]     private
[ 8]     integer :: low, high, limit    ! state variables & access
[ 9]   end type Fibonacci_Number
[10]
[11] Contains                          ! member functionality
[12]
[13]   function new_Fibonacci_Number (max) result (num) ! constructor
[14]   implicit none
[15]     integer, optional :: max
[16]     type (Fibonacci_Number) :: num
[17]     num = Fibonacci_Number (0, 1, 0) ! intrinsic
[18]     if ( present(max) ) num = Fibonacci_Number (0, 1, max) ! intrinsic
[19]     num%exists = .true.
[20]   end function new_Fibonacci_Number
[21]
[22]   function Add (this) result (sum)
[23]   implicit none
[24]     type (Fibonacci_Number), intent(in) :: this ! cannot modify
[25]     integer :: sum
[26]     sum = this%low + this%high ; end function add ! add components
[27]
[28]   subroutine Print (num)
[29]   implicit none
[30]     type (Fibonacci_Number), intent(inout) :: num ! will modify
[31]     integer :: j, sum ! loops
[32]     if ( num%limit < 0 ) return ! no data to print
[33]     print *, 'M Fibonacci(M)' ! header
[34]     do j = 1, num%limit ! loop over range
[35]       sum = Add(num) ; print *, j, sum ! sum and print
[36]       num%low = num%high ; num%high = sum ! update
[37]     end do ; end subroutine Print
[38] End Module class_Fibonacci_Number
[39]
[40] program Fibonacci                  !** The main Fibonacci program
[41] implicit none
[42] use class_Fibonacci_Number         ! inherit variables and members
[43] integer, parameter :: end = 8     ! unchangeable
[44] type (Fibonacci_Number) :: num
[45] num = new_Fibonacci_Number(end)    ! manual constructor
[46] call Print (num)                  ! create and print list
[47] end program Fibonacci              ! Running gives:
[48]
[49] ! M Fibonacci(M) ; ! M Fibonacci(M)
[50] ! 1 1 ; ! 5 8
[51] ! 2 2 ; ! 6 13
[52] ! 3 3 ; ! 7 21
[53] ! 4 5 ; ! 8 34

```

**Figure 2.6:** A Simple Fibonacci Class

## 2.5 Exercises

1. Create a module of global constants of common a) physical constants, b) common units conversion factors.
2. Teams in a Sports League compete in matches that result in a tie or a winning and losing team. When the result is not a tie the status of the teams is updated. The winner is declared better than the loser and better than any team that was previously bettered by the loser. Specify this process by ADTs for the League, Team, and Match. Include a logical member function `is_better_than` which expresses whether a team is better than another.

## Chapter 3

# Object Oriented Programming Concepts

### 3.1 Introduction

The use of Object Oriented (OO) design and Object Oriented Programming (OOP) are becoming increasingly popular. Thus, it is useful to have an introductory understanding of OOP and some of the programming features of OO languages. You can develop OO software in any high level language, like C or Pascal. However, newer languages such as Ada, C++, and F90 have enhanced features that make OOP much more natural, practical, and maintainable. C++ appeared before F90 and currently, is probably the most popular OOP language, yet F90 was clearly designed to have almost all of the abilities of C++. However, rather than study the new standards many authors simply refer to the two decades old F77 standard and declare that Fortran can not be used for OOP. Here we will overcome that misinformed point of view.

Modern OO languages provide the programmer with three capabilities that improve and simplify the design of such programs: *encapsulation*, *inheritance*, and *polymorphism* (or generic functionality). Related topics involve *objects*, *classes*, and *data hiding*. An *object* combines various classical data types into a set that defines a new variable type, or structure. A *class* unifies the new entity types and supporting data that represents its state with routines (functions and subroutines) that access and/or modify those data. Every object created from a class, by providing the necessary data, is called an *instance* of the class. In older languages like C and F77, the data and functions are separate entities. An OO language provides a way to couple or encapsulate the data and its functions into a unified entity. This is a more natural way to model real-world entities which have both data and functionality. The encapsulation is done with a “module” block in F90, and with a “class” block in C++. This encapsulation also includes a mechanism whereby some or all of the data and supporting routines can be hidden from the user. The accessibility of the specifications and routines of a class is usually controlled by optional “public” and “private” qualifiers. *Data hiding* allows one the means to protect information in one part of a program from access, and especially from being changed in other parts of the program. In C++ the default is that data and functions are “private” unless declared “public,” while F90 makes the opposite choice for its default protection mode. In a F90 “module” it is the “contains” statement that, among other things, couples the data, specifications, and operators before it to the functions and subroutines that follow it.

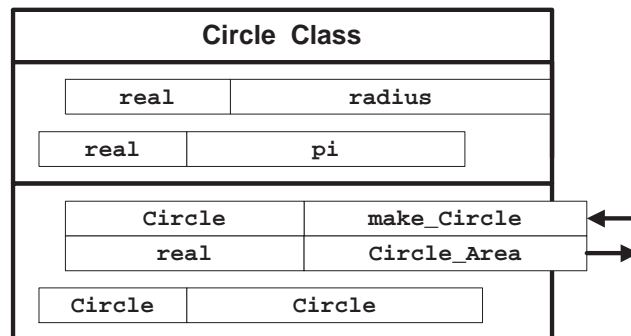
Class hierarchies can be visualized when we realize that we can employ one or more previously defined classes (of data and functionality) to organize additional classes. Functionality programmed into the earlier classes may not need to be re-coded to be usable in the later classes. This mechanism is called *inheritance*. For example, if we have defined an `Employee_class`, then a `Manager_class` would inherit all of the data and functionality of an employee. We would then only be required to add only the totally new data and functions needed for a manager. We may also need a mechanism to re-define specific `Employee_class` functions that differ for a `Manager_class`. By using the concept of a class hierarchy, less programming effort is required to create the final enhanced program. In F90 the earlier class is brought into the later class hierarchy by the “use” statement followed by the name of the “module” statement block that defined the class.

*Polymorphism* allows different classes of objects that share some common functionality to be used in code that requires only that common functionality. In other words, routines having the same generic name

are interpreted differently depending on the class of the objects presented as arguments to the routines. This is useful in class hierarchies where a small number of meaningful function names can be used to manipulate different, but related object classes. The above concepts are those essential to object oriented design and OOP. In the later sections we will demonstrate by example additional F90 implementations of these concepts.

## 3.2 Encapsulation, Inheritance, and Polymorphism

We often need to use existing classes to define new classes. The two ways to do this are called *composition* and *inheritance*. We will use both methods in a series of examples. Consider a geometry program that uses two different classes: `class_Circle` and `class_Rectangle`, as represented graphically in Figs. 3.1 and 3.2. and as partially implemented in F90 as shown in Fig. 3.3. Each class shown has the data types and specifications to define the object and the functionality to compute their respective areas (lines 3–22). The operator `%` is employed to select specific components of a defined type. Within the geometry (main) program a single routine, `compute_area`, is invoked (lines 38 and 44) to return the area for *any* of the defined geometry classes. That is, a generic function name is used for all classes of its arguments and it, in turn, branches to the corresponding functionality supplied with the argument class. To accomplish this branching the geometry program first brings in the functionality of the desired classes via a “use” statement for each class module (lines 25 and 26). Those “modules” are coupled to the generic function by an “interface” block which has the generic function name `compute_area` (lines 28, 29). There is included a “module procedure” list which gives one class routine name for each of the classes of argument(s) that the generic function is designed to accept. The ability of a function to respond differently when supplied with arguments that are objects of different types is called *polymorphism*. In this example we have employed different names, `rectangular_area` and `circle_area`, in their respective class modules, but that is not necessary. The “use” statement allows one to rename the class routines and/or to bring in only selected members of the functionality.



**Figure 3.1:** Representation of a Circle Class

Another terminology used in OOP is that of *constructors* and *destructors* for objects. An intrinsic constructor is a system function that is automatically invoked when an object is declared with all of its possible components in the defined order (see lines 37 and 43). In C++, and F90 the intrinsic constructor has the same name as the “type” of the object. One is illustrated in the statement

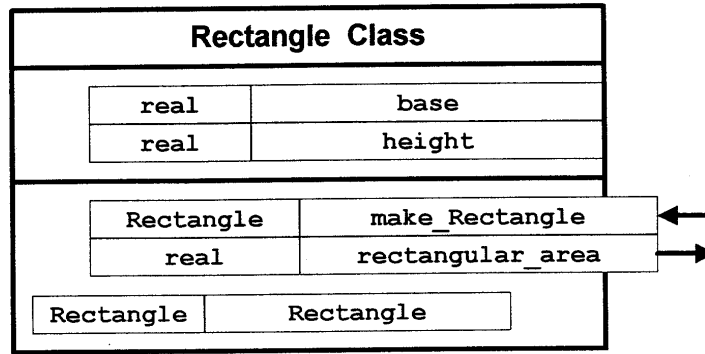
```
four_sides = Rectangle (2.1,4.3)
```

where previously we declared

```
type (Rectangle) :: four_sides
```

which, in turn, was coupled to the `class_Rectangle` which had two components, base and height, defined in that order, respectively. The intrinsic constructor in the example statement sets component





**Figure 3.2:** Representation of a Rectangle Class

```

[ 1] ! Areas of shapes of different classes, using different
[ 2] ! function names in each class
[ 3] module class_Rectangle ! define the first object class
[ 4] implicit none
[ 5] type Rectangle
[ 6] real :: base, height ; end type Rectangle
[ 7] contains ! Computation of area for rectangles.
[ 8] function rectangle_area ( r ) result ( area )
[ 9] type ( Rectangle ), intent(in) :: r
[10] real :: area
[11] area = r%base * r%height ; end function rectangle_area
[12] end module class_Rectangle
[13]
[14] module class_Circle ! define the second object class
[15] real :: pi = 3.1415926535897931d0 ! a circle constant
[16] type Circle
[17] real :: radius ; end type Circle
[18] contains ! Computation of area for circles.
[19] function circle_area ( c ) result ( area )
[20] type ( Circle ), intent(in) :: c
[21] real :: area
[22] area = pi * c%radius**2 ; end function circle_area
[23] end module class_Circle
[24]
[25] program geometry ! for both types in a single function
[26] use class_Circle
[27] implicit none
[28] use class_Rectangle
[29] ! Interface to generic routine to compute area for any type
[30] interface compute_area
[31] module procedure rectangle_area, circle_area ; end interface
[32]
[33] ! Declare a set geometric objects.
[34] type ( Rectangle ) :: four_sides
[35] type ( Circle ) :: two_sides ! inside, outside
[36] real :: area = 0.0 ! the result
[37]
[38] ! Initialize a rectangle and compute its area.
[39] four_sides = Rectangle ( 2.1, 4.3 ) ! implicit constructor
[40] area = compute_area ( four_sides ) ! generic function
[41] write ( 6,100 ) four_sides, area ! implicit components list
[42] 100 format ("Area of ",f3.1," by ",f3.1," rectangle is ",f5.2)
[43]
[44] ! Initialize a circle and compute its area.
[45] two_sides = Circle ( 5.4 ) ! implicit constructor
[46] area = compute_area ( two_sides ) ! generic function
[47] write ( 6,200 ) two_sides, area
[48] 200 format ("Area of circle with ",f3.1," radius is ",f9.5 )
[49] end program geometry ! Running gives:
[50] ! Area of 2.1 by 4.3 rectangle is 9.03
[51] ! Area of circle with 5.4 radius is 91.60885

```

**Figure 3.3:** Multiple Geometric Shape Classes

base = 2.1 and component height = 4.3 for that instance, four\_sides, of the type Rectangle. This intrinsic construction is possible because all the expected components of the type were supplied. If all the components are not supplied, then the object cannot be constructed unless the functionality of the

```

[ 1] function make_Rectangle (bottom, side) result (name)
[ 2] !       Constructor for a Rectangle type
[ 3] implicit none
[ 4]   real, optional, intent(in) :: bottom, side
[ 5]   type (Rectangle)          :: name
[ 6]   name = Rectangle (1.,1.)   ! default to unit square
[ 7]   if ( present(bottom) ) then ! default to square
[ 8]     name = Rectangle (bottom, bottom) ; end if
[ 9]   if ( present(side) ) name = Rectangle (bottom, side) ! intrinsic
[10] end function make_Rectangle
[11] . . .
[12] type ( Rectangle ) :: four_sides, square, unit_sq
[13] !       Test manual constructors
[14] four_sides = make_Rectangle (2.1,4.3) ! manual constructor, 1
[15] area = compute_area ( four_sides)    ! generic function
[16] write ( 6,100 ) four_sides, area
[17] !       Make a square
[18] square = make_Rectangle (2.1)        ! manual constructor, 2
[19] area = compute_area ( square)        ! generic function
[20] write ( 6,100 ) square, area
[21] !       "Default constructor", here a unit square
[22] unit_sq = make_Rectangle ( )         ! manual constructor, 3
[23] area = compute_area (unit_sq)        ! generic function
[24] write ( 6,100 ) unit_sq, area
[25] . . .
[26] ! Running gives:
[27] ! Area of 2.1 by 4.3 rectangle is 9.03
[28] ! Area of 2.1 by 2.1 rectangle is 4.41
[29] ! Area of 1.0 by 1.0 rectangle is 1.00

```

**Figure 3.4:** A Manual Constructor for Rectangles

class is expanded by the programmer to accept a different number of arguments.

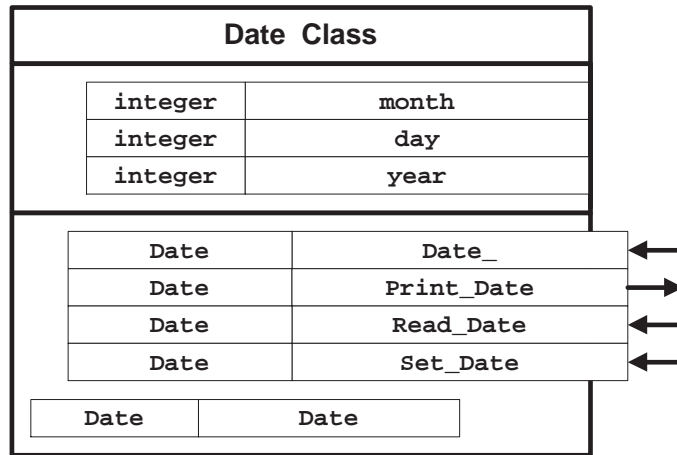
Assume that we want a special member of the `Rectangle` class, a square, to be constructed if the height is omitted. That is, we would use `height = base` in that case. Or, we may want to construct a unit square if both are omitted so that the constructor defaults to `base = height = 1`. Such a manual constructor, named `make_Rectangle`, is illustrated in Fig. 3.4 (see lines 5, 6). It illustrates some additional features of F90. Note that the last two arguments were declared to have the additional type attributes of “optional” (line 3), and that an associated logical function “present” is utilized (lines 6 and 8) to determine if the calling program supplied the argument in question. That figure also shows the results of the area computations for the corresponding variables “square” and “unit\_sq” defined if the manual constructor is called with one or no optional arguments (line 5), respectively.

In the next section we will illustrate the concept of data hiding by using the `private` attribute. The reader is warned that the intrinsic constructor can not be employed if any of its arguments have been hidden. In that case a manual constructor must be provided to deal with any hidden components. Since data hiding is so common it is probably best to plan on providing a manual constructor.

### 3.2.1 Example Date, Person, and Student Classes

Before moving to some mathematical examples we will introduce the concept of data hiding and combine a series of classes to illustrate composition and inheritance<sup>†</sup>. First, consider a simple class to define dates and to print them in a pretty fashion, as shown in Figs. 3.5 and 3.6. While other modules will have access to the `Date` class they will not be given access to the number of components it contains (3), nor their names (month, day, year), nor their types (integers) because they are declared “private” in the defining module (lines 5 and 6). The compiler will not allow external access to data and/or routines declared as private. The module, `class_Date`, is presented as a source “include” file in Fig. 3.6, and in the future will be reference by the file name `class_Date.f90`. Since we have chosen to hide all the user defined components we must decide what functionality we will provide to the users, who may have only executable access. The supporting documentation would have to name the public routines and describe their arguments and return results. The default intrinsic constructor would be available only to those that know full details about the components of the data type, and if those components are “public.”

<sup>†</sup>These examples mimic those given in Chapter 11 and 8 of the J.R. Hubbard book “Programming with C++,” McGraw-Hill, 1994, and usually use the same data for verification.



**Figure 3.5:** Graphical Representation of a Date Class

The intrinsic constructor, `Date` (lines 14 and 34), requires all the components be supplied, but it does no error or consistency checks. My practice is to also define a “public constructor” whose name is the same as the intrinsic constructor except for an appended underscore, that is, `Date_`. Its sole purpose is to do data checking and invoke the intrinsic constructor, `Date`. If the function `Date_` (line 10) is declared “public” it can be used outside the module `class_Date` to invoke the intrinsic constructor, even if the components of the data type being constructed are all “private.” In this example we have provided another manual constructor to set a date, `set_Date` (line 31), with a variable number of optional arguments. Also supplied are two subroutines to read and print dates, `read_Date` (line 27) and `print_Date` (line 16), respectively.

A sample main program that employs this class is given in Fig. 3.7, which contains sample outputs as comments. This program uses the default constructor as well as all three programs in the public class functionality. Note that the definition of the class was copied in via an “include” (line 1) statement and activated with the “use” statement (line 4).

Now we will employ the `class_Date` within a `class_Person` which will use it to set the date of birth (DOB) and date of death (DOD) in addition to the other `Person` components of name, nationality, and sex. As shown in Fig. 3.8, we have made all the type components “private,” but make all the supporting functionality public, as represented graphically in Fig. 3.8. The functionality shown provides a manual constructor, `make_Person`, routines to set the DOB or DOD, and those for the printing of most components. The source code for the new `Person` class is given in Fig. 3.9. Note that the manual constructor (line 12) utilizes “optional” arguments and initializes all components in case they are not supplied to the constructor. The `Date_` public function from the `class_Date` is “inherited” to initialize the DOB and DOD (lines 18, 57, and 62). That function member from the previous module was activated with the combination of the “include” and “use” statements. Of course, the include could have been omitted if the compile statement included the path name to that source. A sample main program for testing the `class_Person` is in Fig. 3.10 along with comments containing its output. It utilizes the constructors `Date_` (line 7), `Person_` (line 10), and `make_Person` (line 24).

Next, we want to use the previous two classes to define a `class_Student` which adds something else special to the general `class_Person`. The student person will have additional “private” components for an identification number, the expected date of matriculation (DOM), the total course credit hours earned (credits), and the overall grade point average (GPA), as represented in Fig. 3.11. The source lines for the type definition and selected public functionality are given in Fig. 3.12. There the constructors are `make_Student` (line 19) and `Student_` (line 47). A testing main program with sample output is illustrated in Fig. 3.13. Since there are various ways to utilize the various constructors three alternate methods have been included as comments to indicate some of the programmers options. The first two `include` statements (lines 1, 2) are actually redundant because the third `include` automatically brings in those first two classes.

```

[ 1] module class_Date          ! filename: class_Date.f90
[ 2] implicit none
[ 3] public :: Date ! and everything not "private"
[ 4]
[ 5] type Date
[ 6]   private
[ 7]   integer :: month, day, year ; end type Date
[ 8]
[ 9] contains ! encapsulated functionality
[10]
[11] function Date_ (m, d, y) result (x) ! public constructor
[12]   integer, intent(in) :: m, d, y    ! month, day, year
[13]   type (Date)          :: x        ! from intrinsic constructor
[14]   if ( m < 1 .or. d < 1 ) stop 'Invalid components, Date_'
[15]   x = Date (m, d, y) ; end function Date_
[16]
[17] subroutine print_Date (x)          ! check and pretty print a date
[18]   type (Date), intent(in)         :: x
[19]   character (len=*) , parameter :: month_Name(12) = &
[20]     (/ "January ", "February ", "March   ", "April    ", &
[21]        "May      ", "June     ", "July    ", "August   ", &
[22]        "September", "October  ", "November", "December"/)
[23]   if ( x%month < 1 .or. x%month > 12 ) print *, "Invalid month"
[24]   if ( x%day < 1 .or. x%day > 31 ) print *, "Invalid day "
[25]   print *, trim(month_Name(x%month)), ' ', x%day, " ", x%year;
[26] end subroutine print_Date
[27]
[28] subroutine read_Date (x)           ! read month, day, and year
[29]   type (Date), intent(out) :: x ! into intrinsic constructor
[30]   read *, x ; end subroutine read_Date
[31]
[32] function set_Date (m, d, y) result (x) ! manual constructor
[33]   integer, optional, intent(in) :: m, d, y ! month, day, year
[34]   type (Date)                  :: x
[35]   x = Date (1,1,1997)           ! default, (or use current date)
[36]   if ( present(m) ) x%month = m ; if ( present(d) ) x%day  = d
[37]   if ( present(y) ) x%year  = y ; end function set_Date
[38]
[39] end module class_Date

```

**Figure 3.6:** Defining a Date Class

```

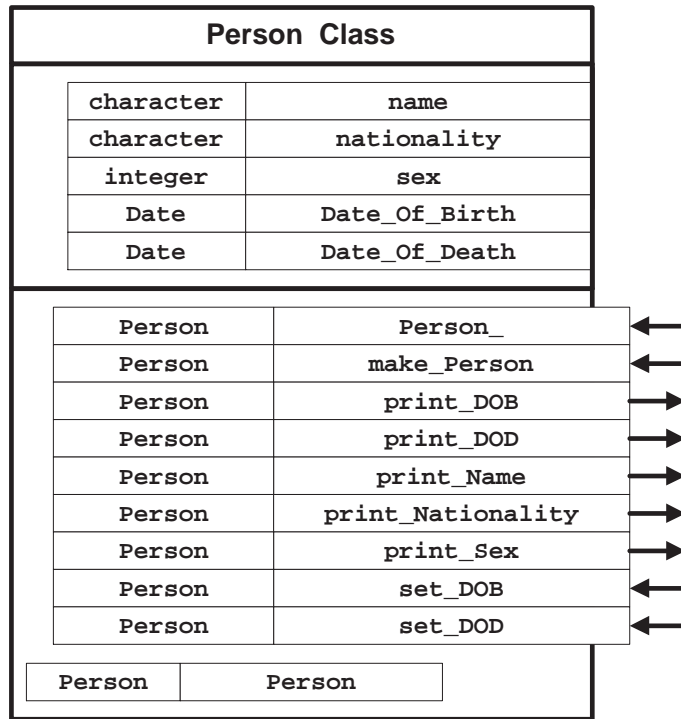
[ 1] include 'class_Date.f90' ! see previous figure
[ 2] program main
[ 3]   use class_Date
[ 4]   implicit none
[ 5]   type (Date) :: today, peace
[ 6]
[ 7]   ! peace = Date (11,11,1918) ! NOT allowed for private components
[ 8]   peace = Date_ (11,11,1918) ! public constructor
[ 9]   print *, "World War I ended on " ; call print_Date (peace)
[10]   peace = set_Date (8, 14, 1945) ! optional constructor
[11]   print *, "World War II ended on " ; call print_Date (peace)
[12]   print *, "Enter today as integer month, day, and year: "
[13]   call read_Date(today) ! create today's date
[14]
[15]   print *, "The date is "; call print_Date (today)
[16] end program main ! Running produces:
[17] ! World War I ended on November 11, 1918
[18] ! World War II ended on August 14, 1945
[19] ! Enter today as integer month, day, and year: 7 10 1997
[20] ! The date is July 10, 1997

```

**Figure 3.7:** Testing a Date Class

### 3.3 Object Oriented Numerical Calculations

OOP is often used for numerical computation, especially when the standard storage mode for arrays is not practical or efficient. Often one will find specialized storage modes like linked lists, or tree structures used for dynamic data structures. Here we should note that many matrix operators are intrinsic to F90, so one is more likely to define a `class_sparse_matrix` than a `class_matrix`. However, either class would allow us to encapsulate several matrix functions and subroutines into a module that could be reused easily in other software. Here, we will illustrate OOP applied to rational numbers and introduce



**Figure 3.8:** Graphical Representation of a Person Class

the important topic of operator overloading. Additional numerical applications of OOP will be illustrated in later chapters.

### 3.3.1 A Rational Number Class and Operator Overloading

To illustrate an OOP approach to simple numerical operations we will introduce a fairly complete rational number class, called `class_Rational` which is represented graphically in Fig. 3.14. The defining F90 module is given in Fig. 3.15. The type components have been made private (line 5), but not the type itself, so we can illustrate the intrinsic constructor (lines 38 and 102), but extra functionality has been provided to allow users to get either of the two components (lines 52 and 57). The provided routines shown in that figure are:

<code>add_Rational</code>	<code>convert</code>	<code>copy_Rational</code>	<code>delete_Rational</code>
<code>equal_integer</code>	<code>gcd</code>	<code>get_Denominator</code>	<code>get_Numerator</code>
<code>invert</code>	<code>is_equal_to</code>	<code>list</code>	<code>make_Rational</code>
<code>mult_Rational</code>	<code>Rational_</code>	<code>reduce</code>	

Procedures with only one return argument are usually implemented as functions instead of subroutines.

Note that we would form a new rational number,  $z$ , as the product of two other rational numbers,  $x$  and  $y$ , by invoking the `mult_Rational` function (line 90),

```
z = mult_Rational (x,y)
```

which returns  $z$  as its result. A natural tendency at this point would be to simply write this as  $z = x * y$ . However, before we could do that we would have to have to tell the operator, “\*”, how to act when provided with this new data type. This is known as *overloading* an intrinsic operator. We had the foresight to do this when we set up the module by declaring which of the “module procedures” were equivalent to this operator symbol. Thus, from the “interface operator (\*)” statement block (line 14) the system now knows that the left and right operands of the “\*” symbol correspond to the first and second arguments in the function `mult_Rational`. Here it is not necessary to overload the assignment operator, “=”, when both of its operands are of the same intrinsic or defined type. However, to convert

```

[ 1] module class_Person          ! filename: class_Person.f90
[ 2] use class_Date
[ 3] implicit none
[ 4] public :: Person
[ 5] type Person
[ 6]   private
[ 7]   character (len=20) :: name
[ 8]   character (len=20) :: nationality
[ 9]   integer             :: sex
[10]   type (Date)         :: dob, dod    ! birth, death
[11] end type Person
[12] contains
[13] function make_Person (nam, nation, s, b, d) result (who)
[14] !   Optional Constructor for a Person type
[15]   character (len=*), optional, intent(in) :: nam, nation
[16]   integer, optional, intent(in) :: s    ! sex
[17]   type (Date), optional, intent(in) :: b, d ! birth, death
[18]   type (Person) :: who
[19]   who = Person (" ", "USA", 1, Date_(1,1,0), Date_(1,1,0)) ! defaults
[20]   if ( present(nam) ) who % name = nam
[21]   if ( present(nation) ) who % nationality = nation
[22]   if ( present(s) ) who % sex = s
[23]   if ( present(b) ) who % dob = b
[24]   if ( present(d) ) who % dod = d ; end function
[25]
[26] function Person_ (nam, nation, s, b, d) result (who)
[27] !   Public Constructor for a Person type
[28]   character (len=*), intent(in) :: nam, nation
[29]   integer, intent(in) :: s    ! sex
[30]   type (Date), intent(in) :: b, d ! birth, death
[31]   type (Person) :: who
[32]   who = Person (nam, nation, s, b, d) ; end function Person_
[33]
[34] subroutine print_DOB (who)
[35]   type (Person), intent(in) :: who
[36]   call print_Date (who % dob) ; end subroutine print_DOB
[37]
[38] subroutine print_DOD (who)
[39]   type (Person), intent(in) :: who
[40]   call print_Date (who % dod) ; end subroutine print_DOD
[41]
[42] subroutine print_Name (who)
[43]   type (Person), intent(in) :: who
[44]   print *, who % name ; end subroutine print_Name
[45]
[46] subroutine print_Nationality (who)
[47]   type (Person), intent(in) :: who
[48]   print *, who % nationality ; end subroutine print_Nationality
[49]
[50] subroutine print_Sex (who)
[51]   type (Person), intent(in) :: who
[52]   if ( who % sex == 1 ) then ; print *, "male"
[53]   else ; print *, "female" ; end if ; end subroutine print_Sex
[54]
[55] subroutine set_DOB (who, m, d, y)
[56]   type (Person), intent(inout) :: who
[57]   integer, intent(in) :: m, d, y ! month, day, year
[58]   who % dob = Date_ (m, d, y) ; end subroutine set_DOB
[59]
[60] subroutine set_DOD(who, m, d, y)
[61]   type (Person), intent(inout) :: who
[62]   integer, intent(in) :: m, d, y ! month, day, year
[63]   who % dod = Date_ (m, d, y) ; end subroutine set_DOD
[64] end module class_Person

```

**Figure 3.9:** Definition of a Typical Person Class

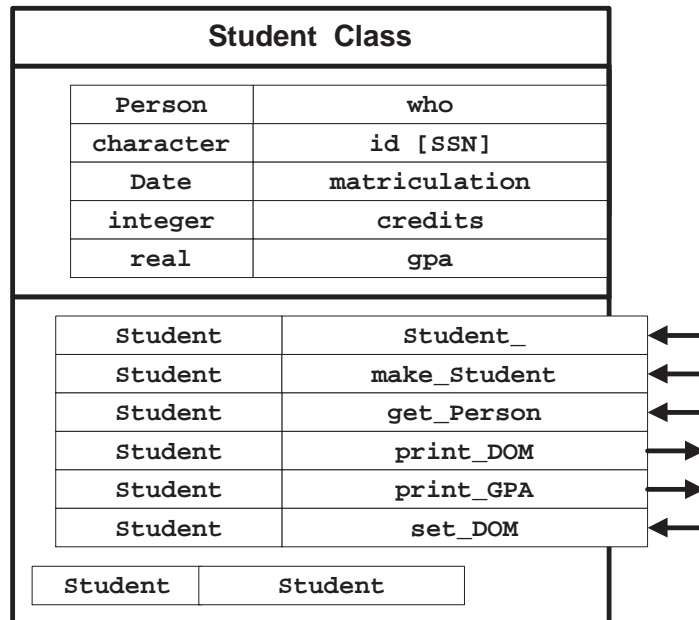
an integer to a rational we could, and have, defined an overloaded assignment operator procedure (line 10). Here we have provided the procedure, `equal_Integer`, which is automatically invoked when we write: `type(Rational)y; y = 4`. That would be simpler than invoking the constructor called `make_rational`. Before moving on note that the system does not yet know how to multiply an integer times a rational number, or visa versa. To do that one would have to add more functionality, such as a function, say `int_mult_rn`, and add it to the “module procedure” list associated with the “\*” operator. A typical main program which exercises most of the rational number functionality is given in Fig. 3.16, along with typical numerical output. It tests the constructors `Rational_` (line 8), `make_Rational`

```

[ 1] include 'class_Date.f90'
[ 2] include 'class_Person.f90'           ! see previous figure
[ 3] program main
[ 4]   use class_Date ; use class_Person   ! inherit class members
[ 5]   implicit none
[ 6]   type (Person) :: author, creator
[ 7]   type (Date)   :: b, d               ! birth, death
[ 8]   b = Date_(4,13,1743) ; d = Date_(7, 4,1826) ! OPTIONAL
[ 9]   ! Method 1
[10]   ! author = Person ("Thomas Jefferson", "USA", 1, b, d) ! NOT if private
[11]   author = Person_ ("Thomas Jefferson", "USA", 1, b, d) ! constructor
[12]   print *, "The author of the Declaration of Independence was ";
[13]   call print_Name (author);
[14]   print *, ". He was born on "; call print_DOB (author);
[15]   print *, " and died on ";    call print_DOD (author); print *, ".";
[16]   ! Method 2
[17]   author = make_Person ("Thomas Jefferson", "USA") ! alternate
[18]   call set_DOB (author, 4, 13, 1743)              ! add DOB
[19]   call set_DOD (author, 7, 4, 1826)                ! add DOD
[20]   print *, "The author of the Declaration of Independence was ";
[21]   call print_Name (author)
[22]   print *, ". He was born on "; call print_DOB (author);
[23]   print *, " and died on ";    call print_DOD (author); print *, ".";
[24]   ! Another Person
[25]   creator = make_Person ("John Backus", "USA")      ! alternate
[26]   print *, "The creator of Fortran was "; call print_Name (creator);
[27]   print *, " who was born in ";    call print_Nationality (creator);
[28]   print *, ".";
[29] end program main                                     ! Running gives:
[30] ! The author of the Declaration of Independence was Thomas Jefferson.
[31] ! He was born on April 13, 1743 and died on July 4, 1826.
[32] ! The author of the Declaration of Independence was Thomas Jefferson.
[33] ! He was born on April 13, 1743 and died on July 4, 1826.
[34] ! The creator of Fortran was John Backus who was born in the USA.

```

**Figure 3.10:** Testing the Date and Person Classes



**Figure 3.11:** Graphical Representation of a Student Class

(lines 14, 18, 25), and a simple destructor `delete_Rational` (line 38). The intrinsic constructor (line 6) could have been used only if all the attributes were public, and that is considered an undesirable practice in OOP. The simple destructor actually just sets the “deleted” number to have a set of default components. Later we will see that constructors and destructors often must dynamically allocate and deallocate, respectively, memory associated with a specific instance of some object.

```

[ 1] module class_Student                ! filename class_Student.f90
[ 2] use class_Person                   ! inherits class_Date
[ 3] implicit none
[ 4] public :: Student, set_DOM, print_DOM
[ 5] type Student
[ 6]   private
[ 7]   type (Person)      :: who      ! name and sex
[ 8]   character (len=9) :: id       ! ssn digits
[ 9]   type (Date)        :: dom      ! matriculation
[10]   integer            :: credits
[11]   real               :: gpa      ! grade point average
[12] end type Student
[13] contains ! coupled functionality
[14]
[15] function get_person (s) result (p)
[16]   type (Student), intent(in) :: s
[17]   type (Person)              :: p      ! name and sex
[18]   p = s % who ; end function get_person
[19]
[20] function make_Student (w, n, d, c, g) result (x) ! constructor
[21] !   Optional Constructor for a Student type
[22]   type (Person),          intent(in) :: w ! who
[23]   character (len=*) , optional, intent(in) :: n ! ssn
[24]   type (Date),            optional, intent(in) :: d ! matriculation
[25]   integer,                optional, intent(in) :: c ! credits
[26]   real,                   optional, intent(in) :: g ! grade point ave
[27]   type (Student)          :: x ! new student
[28]   x = Student_(w, " ", Date_(1,1,1), 0, 0.) ! defaults
[29]   if ( present(n) ) x % id = n              ! optional values
[30]   if ( present(d) ) x % dom = d
[31]   if ( present(c) ) x % credits = c
[32]   if ( present(g) ) x % gpa = g ; end function make_Student
[33]
[34] subroutine print_DOM (who)
[35]   type (Student), intent(in) :: who
[36]   call print_Date(who%dom) ; end subroutine print_DOM
[37]
[38] subroutine print_GPA (x)
[39]   type (Student), intent(in) :: x
[40]   print *, "My name is "; call print_Name (x % who)
[41]   print *, " , and my G.P.A. is ", x % gpa, "." ; end subroutine
[42]
[43] subroutine set_DOM (who, m, d, y)
[44]   type (Student), intent(inout) :: who
[45]   integer,          intent(in)   :: m, d, y
[46]   who % dom = Date_( m, d, y) ; end subroutine set_DOM
[47]
[48] function Student_ (w, n, d, c, g) result (x)
[49] !   Public Constructor for a Student type
[50]   type (Person),          intent(in) :: w ! who
[51]   character (len=*) , intent(in) :: n ! ssn
[52]   type (Date),            intent(in) :: d ! matriculation
[53]   integer,                intent(in) :: c ! credits
[54]   real,                   intent(in) :: g ! grade point ave
[55]   type (Student)          :: x ! new student
[56]   x = Student (w, n, d, c, g) ; end function Student_
[57] end module class_Student

```

**Figure 3.12:** Defining a Typical Student Class

When considering which operators to overload for a newly defined object one should consider those that are used in sorting operations, such as the greater-than, >, and less-than, <, operators. They are often useful because of the need to sort various types of objects. If those symbols have been correctly overloaded then a generic object sorting routine might be used, or require few changes.

### 3.4 Discussion

The previous sections have only briefly touched on some important OOP concepts. More details will be covered later after a general overview of the features of the Fortran language. There are more than one hundred OOP languages. Persons involved in software development need to be aware that F90 can meet almost all of their needs for a OOP language. At the same time it includes the F77 standard as a subset and thus allows efficient use of the many millions of Fortran functions and subroutines developed in the past. The newer F95 standard is designed to make efficient use of super computers and massively parallel



```

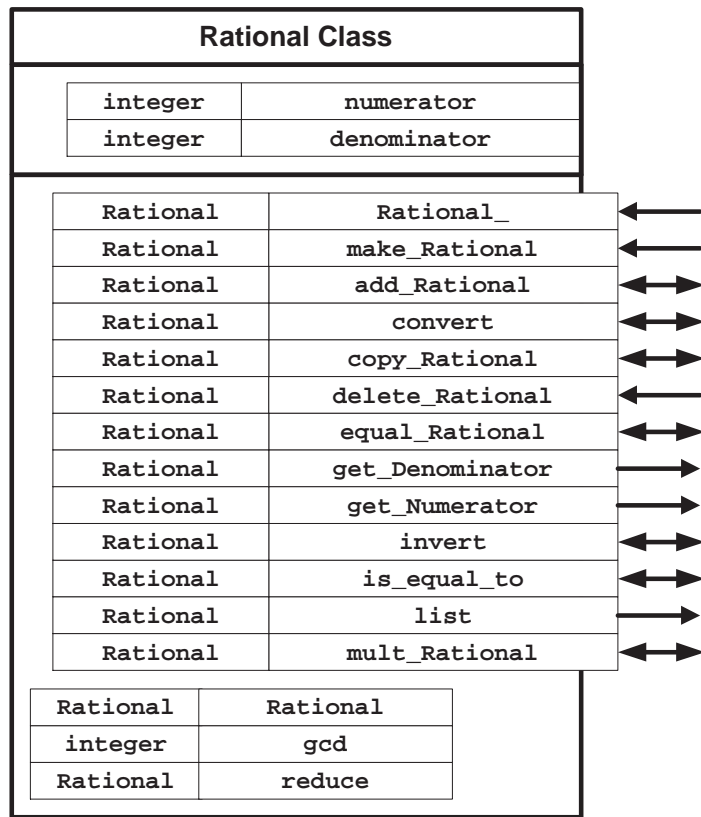
[ 1] include 'class_Date.f90'
[ 2] include 'class_Person.f90'
[ 3] include 'class_Student.f90' ! see previous figure
[ 4] program main                ! create or correct a student
[ 5]   use class_Student          ! inherits class_Person, class_Date also
[ 6]   implicit none
[ 7]   type (Person) :: p ; type (Student) :: x
[ 8]   ! Method 1
[ 9]     p = make_Person ("Ann Jones", "", 0) ! optional person constructor
[10]     call set_DOB (p, 5, 13, 1977)        ! add birth to person data
[11]     x = Student_(p, "219360061", Date_(8,29,1955), 9, 3.1) ! public
[12]     call print_Name (p)                  ! list name
[13]     print *, "Born      :"; call print_DOB (p)      ! list dob
[14]     print *, "Sex       :"; call print_Sex (p)       ! list sex
[15]     print *, "Matriculated: "; call print_DOM (x)    ! list dom
[16]     call print_GPA (x)                    ! list gpa
[17]   ! Method 2
[18]     x = make_Student (p, "219360061") ! optional student constructor
[19]     call set_DOM (x, 8, 29, 1995)      ! correct matriculation
[20]     call print_Name (p)                  ! list name
[21]     print *, "was born on :"; call print_DOB (p)      ! list dob
[22]     print *, "Matriculated: "; call print_DOM (x)    ! list dom
[23]   ! Method 3
[24]     x = make_Student (make_Person("Ann Jones"), "219360061") ! optional
[25]     p = get_Person (x)                    ! get defaulted person data
[26]     call set_DOM (x, 8, 29, 1995)        ! add matriculation
[27]     call set_DOB (p, 5, 13, 1977)       ! add birth
[28]     call print_Name (p)                  ! list name
[29]     print *, "Matriculated: "; call print_DOM (x)    ! list dom
[30]     print *, "was born on :"; call print_DOB (p)      ! list dob
[31] end program main                        ! Running gives:
[32] ! Ann Jones
[33] ! Born      : May 13, 1977
[34] ! Sex       : female
[35] ! Matriculated: August 29, 1955
[36] ! My name is Ann Jones, and my G.P.A. is 3.0999999.
[37] ! Ann Jones was born on: May 13, 1977 , Matriculated: August 29, 1995
[38] ! Ann Jones Matriculated: August 29, 1995 , was born on: May 13, 1977

```

**Figure 3.13:** Testing the Student, Person, and Date Classes

machines. It includes most of the High Performance Fortran features that are in wide use. Thus, efficient use of OOP on parallel machines is available through F90 and F95.

None of the OOP languages have all the features one might desire. For example, the useful concept of a “template” which is standard in C++ is not in the F90 standard. Yet the author has found that a few dozen lines of F90 code will define a preprocessor that allows templates to be defined in F90 and expanded in line at compile time. The real challenge in OOP is the actual OOA and OOD that must be completed before programming can begin, regardless of the language employed. For example, several authors have described widely different approaches for defining classes to be used in constructing OO finite element systems. Additional example applications of OOP in F90 will be given in the following chapters.



**Figure 3.14:** Representation of a Rational Number Class

```

[ 1] module class_Rational                                ! filename: class_Rational.f90
[ 2] implicit none
[ 3] ! public, everything but following private routines
[ 4] private :: gcd, reduce
[ 5] type Rational
[ 6]     private ! numerator and denominator
[ 7]     integer :: num, den ; end type Rational
[ 8]
[ 9]     ! overloaded operators interfaces
[10]     interface assignment (=)
[11]         module procedure equal_Integer ; end interface
[12]     interface operator (+)
[13]         module procedure add_Rational ; end interface
[14]     interface operator (*)
[15]         module procedure mult_Rational ; end interface
[16]     interface operator (==)
[17]         module procedure is_equal_to ; end interface
[18] contains
[19]     function add_Rational (a, b) result (c)
[20]         type (Rational), intent(in) :: a, b
[21]         type (Rational) :: c
[22]         c % num = a % num*b % den + a % den*b % num
[23]         c % den = a % den*b % den
[24]         call reduce (c) ; end function add_Rational
[25]
[26]     function convert (name) result (value) ! rational to real
[27]         type (Rational), intent(in) :: name
[28]         real :: value ! decimal form
[29]         value = float(name % num)/name % den ; end function convert
[30]
[31]     function copy_Rational (name) result (new)
[32]         type (Rational), intent(in) :: name
[33]         type (Rational) :: new
[34]         new % num = name % num
[35]         new % den = name % den ; end function copy_Rational
[36]
[37]     subroutine delete_Rational (name) ! deallocate allocated items
[38]         type (Rational), intent(inout) :: name ! simply zero it here
[39]         name = Rational (0, 1) ; end subroutine delete_Rational
[40]
[41]     subroutine equal_Integer (new, I) ! overload =, with integer
[42]         type (Rational), intent(out) :: new ! left side of operator
[43]         integer, intent(in) :: I ! right side of operator
[44]         new % num = I ; new % den = 1 ; end subroutine equal_Integer
[45]
[46]     recursive function gcd (j, k) result (g) ! Greatest Common Divisor
[47]         integer, intent(in) :: j, k ! numerator, denominator
[48]         integer :: g
[49]         if ( k == 0 ) then ; g = j
[50]         else ; g = gcd ( k, modulo(j,k) ) ! recursive call
[51]         end if ; end function gcd
[52]
[53]     function get_Denominator (name) result (n) ! an access function
[54]         type (Rational), intent(in) :: name
[55]         integer :: n ! denominator
[56]         n = name % den ; end function get_Denominator

```

(Fig. 3.15, A Fairly Complete Rational Number Class (continued))

```

[ 57] function get_Numerator (name) result (n)      ! an access function
[ 58]   type (Rational), intent(in) :: name
[ 59]   integer                        :: n          ! numerator
[ 60]   n = name % num ; end function get_Numerator
[ 61]
[ 62] subroutine invert (name)                      ! rational to rational inversion
[ 63]   type (Rational), intent(inout) :: name
[ 64]   integer                        :: temp
[ 65]   temp = name % num
[ 66]   name % num = name % den
[ 67]   name % den = temp ; end subroutine invert
[ 68]
[ 69] function is_equal_to (a_given, b_given) result (t_f)
[ 70]   type (Rational), intent(in) :: a_given, b_given ! left == right
[ 71]   type (Rational)              :: a, b          ! reduced copies
[ 72]   logical                      :: t_f
[ 73]   a = copy_Rational (a_given) ; b = copy_Rational (b_given)
[ 74]   call reduce(a) ; call reduce(b)              ! reduced to lowest terms
[ 75]   t_f = (a%num == b%num) .and. (a%den == b%den) ; end function
[ 76]
[ 77] subroutine list(name)                         ! as a pretty print fraction
[ 78]   type (Rational), intent(in) :: name
[ 79]   print *, name % num, "/", name % den ; end subroutine list
[ 80]
[ 81] function make_Rational (numerator, denominator) result (name)
[ 82]   ! Optional Constructor for a rational type
[ 83]   integer, optional, intent(in) :: numerator, denominator
[ 84]   type (Rational)               :: name
[ 85]   name = Rational(0, 1)          ! set defaults
[ 86]   if ( present(numerator) ) name % num = numerator
[ 87]   if ( present(denominator) ) name % den = denominator
[ 88]   if ( name % den == 0 ) name % den = 1 ! now simplify
[ 89]   call reduce (name) ; end function make_Rational
[ 90]
[ 91] function mult_Rational (a, b) result (c)      ! to overload *
[ 92]   type (Rational), intent(in) :: a, b
[ 93]   type (Rational)              :: c
[ 94]   c % num = a % num * b % num
[ 95]   c % den = a % den * b % den
[ 96]   call reduce (c) ; end function mult_Rational
[ 97]
[ 98] function Rational_ (numerator, denominator) result (name)
[ 99]   ! Public Constructor for a rational type
[100]   integer, optional, intent(in) :: numerator, denominator
[101]   type (Rational)               :: name
[102]   if ( denominator == 0 ) then ; name = Rational (numerator, 1)
[103]   else ; name = Rational (numerator, denominator) ; end if
[104] end function Rational_
[105]
[106] subroutine reduce (name)                     ! to simplest rational form
[107]   type (Rational), intent(inout) :: name
[108]   integer                        :: g          ! greatest common divisor
[109]   g = gcd (name % num, name % den)
[110]   name % num = name % num/g
[111]   name % den = name % den/g ; end subroutine reduce
[112] end module class_Rational

```

**Figure 3.15:** A Fairly Complete Rational Number Class

```

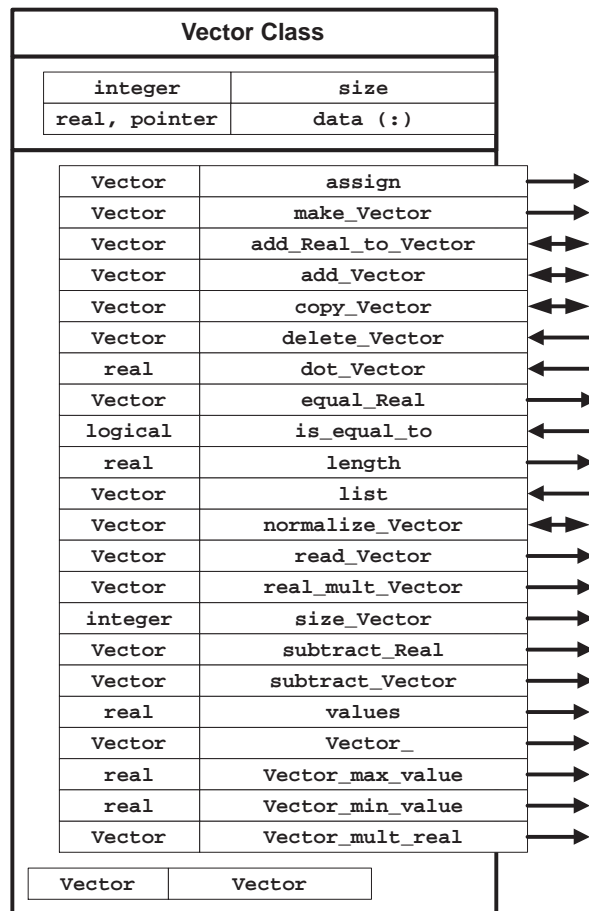
[ 1] include 'class_Rational.f90'
[ 2] program main
[ 3] use class_Rational
[ 4] implicit none
[ 5] type (Rational) :: x, y, z
[ 6] ! ----- only if Rational is NOT private -----
[ 7] ! x = Rational(22,7) ! intrinsic constructor if public components
[ 8]
[ 9] x = Rational_(22,7) ! public constructor if private components
[10] write (*, '("public x = ")', advance='no'); call list(x)
[11] write (*, '("converted x = ", g9.4)', convert(x))
[12] call invert(x)
[13] write (*, '("inverted 1/x = ")', advance='no'); call list(x)
[14]
[15] x = make_Rational () ! default constructor
[16] write (*, '("made null x = ")', advance='no'); call list(x)
[17] y = 4 ! rational = integer overload
[18] write (*, '("integer y = ")', advance='no'); call list(y)
[19] z = make_Rational (22,7) ! manual constructor
[20] write (*, '("made full z = ")', advance='no'); call list(z)
[21] ! Test Accessors
[22] write (*, '("top of z = ", g4.0)', get_numerator(z))
[23] write (*, '("bottom of z = ", g4.0)', get_denominator(z))
[24] ! Misc. Function Tests
[25] write (*, '("making x = 100/360, ")', advance='no')
[26] x = make_Rational (100,360)
[27] write (*, '("reduced x = ")', advance='no'); call list(x)
[28] write (*, '("copying x to y gives ")', advance='no')
[29] y = copy_Rational (x)
[30] write (*, '("a new y = ")', advance='no'); call list(y)
[31] ! Test Overloaded Operators
[32] write (*, '("z * x gives ")', advance='no'); call list(z*x) ! times
[33] write (*, '("z + x gives ")', advance='no'); call list(z+x) ! add
[34] y = z ! overloaded assignment
[35] write (*, '("y = z gives y as ")', advance='no'); call list(y)
[36] write (*, '("logic y == x gives ")', advance='no'); print *, y==x
[37] write (*, '("logic y == z gives ")', advance='no'); print *, y==z
[38] ! Destruct
[39] call delete_Rational (y) ! actually only null it here
[40] write (*, '("deleting y gives y = ")', advance='no'); call list(y)
[41] end program main ! Running gives:
[42] ! public x = 22 / 7 ! converted x = 3.143
[43] ! inverted 1/x = 7 / 22 ! made null x = 0 / 1
[44] ! integer y = 4 / 1 ! made full z = 22 / 7
[45] ! top of z = 22 ! bottom of z = 7
[46] ! making x = 100/360, reduced x = 5 / 18
[47] ! copying x to y gives a new y = 5 / 18
[48] ! z * x gives 55 / 63 ! z + x gives 431 / 126
[49] ! y = z gives y as 22 / 7 ! logic y == x gives F
[50] ! logic y == z gives T ! deleting y gives y = 0 / 1

```

**Figure 3.16:** Testing the Rational Number Class

### 3.5 Exercises

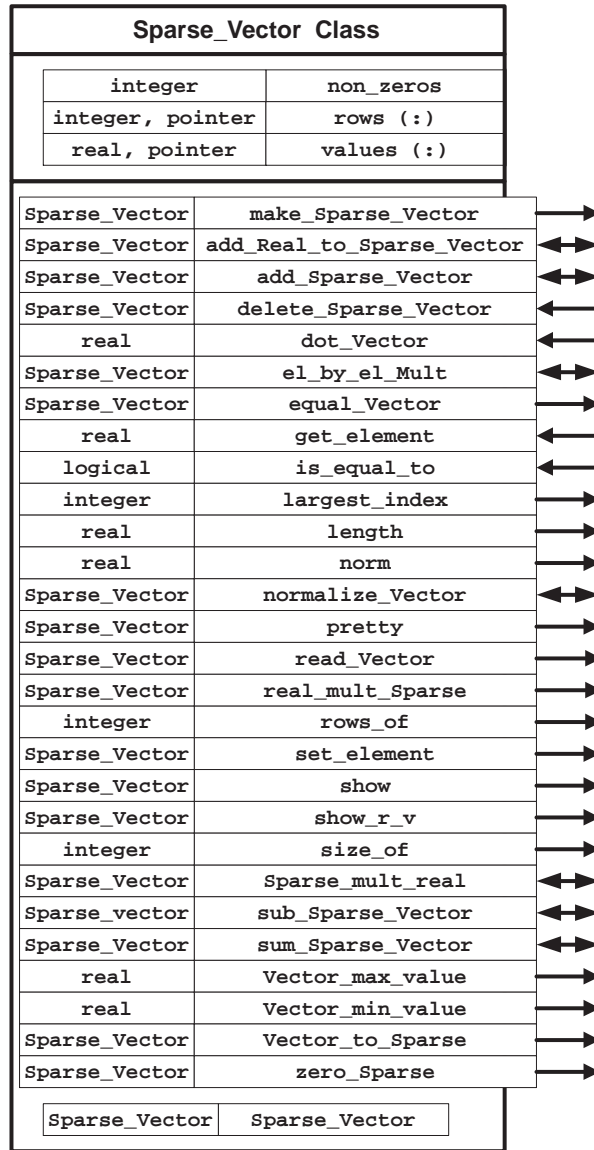
1. Use the `class_Circle` to create a `class_Sphere` that computes the volume of a sphere. Have a method that accepts an argument of a `Circle`. Use the radius of the `Circle` via a new member `get_Circle_radius` to be added to the `class_Circle`.
2. Use the `class_Circle` and `class_Rectangle` to create a `class_Cylinder` that computes the volume of a right circular cylinder. Have a method that accepts arguments of a `Circle` and a height, and a second method that accepts arguments of a `Rectangle` and a radius. In the latter member use the height of the `Rectangle` via a new member `get_Rectangle_height` to be added to the `class_Rectangle`.
3. Create a vector class to treat vectors with an arbitrary number of real coefficients. Assume that the `class_Vector` is defined as follows:



Overload the common operators of (+) with `add_Vector` and `add_Real_to_Vector`, (-) with `subtract_Vector` and `subtract_Real`, (\*) with `dot_Vector`, `real_mult_Vector` and `Vector_mult_real`, (=) with `equal_Real` to set all coefficients to a single real number, and (==) with routine `is_equal_to`.

Include two constructors `assign` and `make_Vector`. Let `assign` convert a real array into an instance of a `Vector`. Provide a destructor, means to read and write a `Vector`, normalize a `Vector`, and determine its extreme values.

4. Modify the above Vector class to extend it to a `Sparse_Vector_Class` where the vast majority of the coefficients are zero. Store and operate only on the non-zero entries.



## Chapter 4

# Features of Programming Languages

The preceding chapter described the programming process as starting with a clearly specified task, expressing it mathematically as a set of algorithms, translating the algorithms in pseudocode, and finally, translating the pseudocode into a “real” programming language. The final stages of this prescription work because most (if not all) computational languages have remarkable similarities: They have statements, the sequencing of which are controlled by various loop and conditional constructs, and functions that foster program modularization. We indicated how similar MATLAB, C++, and Fortran are at this level, but these languages differ the more they are detailed. It is the purpose of this chapter to describe those details, and bring you from a superficial acquaintance with a computational language to fluency. Today, the practicing engineer needs more than one programming language or environment. Once achieving familiarity with one, you will find that learning other languages is easy.

When selecting a programming tool for engineering calculations, one is often faced with two different levels of need. One level is where you need to quickly solve a small problem once, such as a homework assignment, and computational efficiency is not important. You may not care if your code takes ten seconds or one hundred seconds to execute; you want convenience. At that level it may make sense to use an engineering environment like MATLAB, or Mathematica. At the other extreme you may be involved in doing a wide area weather prediction where a one-day run time, instead of a ten-day run time, defines a useful versus a non-useful product. You might be developing a hospital laboratory system for reporting test results to an emergency room physician where an answer in ten seconds versus an answer in ten minutes can literally mean the difference between life or death for a patient. For programming at this level one wants an efficient language. Since such projects can involve programming teams in different countries, you want your language to be based on an international standard. Then you would choose to program a language such as C++ or F90. Since most students have experienced only the first need level, they tend to overvalue the first approach and devalue the second. This chapter will illustrate that the skills needed for either approach are similar.

The structure of this chapter follows our usual progression to learning a language: What are *variables*, how can variables be combined into *expressions*, what constructs are available to control program *flow*, and how are *functions* defined so that we can employ modularity. The basics are described in Chapter 1; we assume you are familiar with the language basics described there. Initially, this chapter will parallel the program composition section of Chapter 1 as applied in the C++, F90, and MATLAB languages, and then it will bring in more advanced topics.

The features of F90 that are to be discussed here have been combined in a series of tables and placed in Appendix B. It is expected that we will want to refer to those tables as we read this section as well as later when we program. At times, references to C++ and MATLAB have been given to show the similarities between most languages and to provide an aid for when having to interface in reading codes in those languages.

## 4.1 Comments

In MATLAB and Fortran, a single character—‘%’ in MATLAB, ‘!’ in F90—located anywhere in a line of text means that the *remainder* of the text on that line comprises the comment. In C, an entirely different



Language	Syntax	Location
MATLAB	% comment (to end of line)	anywhere
C++	// comment (to end of line)	anywhere
F90	! comment (to end of line)	anywhere
F77	* comment (to end of line)	column 1

**Table 4.1:** Comment syntax

structure for comments occurs. Comments begin with the two-character sequence ‘/\*’ and end with the *next* occurrence of the two-character sequence ‘\*/’. In C, comments can occur anywhere in a program; they can consume a portion of a line, temporarily interrupting a statement, or they can span multiple lines of text. C++ allows the use of the C comment syntax, but has added a more popular two-character sequence ‘//’ to proceed a comment to the end of a line. Table 4.1 gives a summary of these comments syntax. It is also in the “Fortran 90 Overview” for quick reference. Samples of comment statements are shown in Fig. 1.3, which gives the corresponding versions of the classic “hello world” program included in most introductory programming texts.

## 4.2 Statements and Expressions

Before introducing statements and expressions, a word about documenting what you program. We encourage the heavy usage of comments. The three languages of concern here all allow comment lines and comments appended to the end of statements. Their form is given above in Fig. 1.3 and Table 4.1.

The above languages currently allow variable names to contain up to 31 characters and allow the use of the underscore, ‘\_’, to aid in clarity by serving as a virtual space character, as in `my_name`. Another useful convention is to use uppercase first letters for words comprising part of a variable’s name: `MyName`. Fortran and MATLAB allow a program line to contain up to 132 characters, while C++ has no limit on line length. Since the old F77 standard was physically limited to holes punched in a card, it allowed only a line length of 72 characters, a maximum name length of six characters, and did not allow the use of the underscore in a name. In this text, we will usually keep line lengths to less than 65 characters in order to make the programs more readable.

A statement in these three languages has a structure common to them all:

```
variable = expression
```

The built-in, or intrinsic, data types allowed for variables are summarized in Table 4.2. Additional user defined types will be considered later. The expressions usually involves the use of arithmetic operators and/or relational operators which are given in Tables 4.3 and 4.4, respectively. The order in which the language applies these operators is called their precedence, and they are shown in Table 4.5. They are also in the “Fortran 90 Overview” for quick reference.

In moving from MATLAB to high level languages one finds that it is necessary to define the type of each variable. Fortran has a default naming convention for its variables and it allows an easy overriding of that built in “implicit” convention. Since most engineering and mathematical publications used the letters from “i” through “n” as subscripts, summation ranges, loop counters, etc. Fortran first was released with implicit variable typing such that all variables whose name begin with the letters “i” through “n”, inclusive, defaulted to integers, unless declared otherwise. All other variables default to be real, unless declared otherwise. In other words, you can think of the default code as if it contained the statements:

```
IMPLICIT INTEGER (I-N)      ! F77 and F90 Default
IMPLICIT REAL   (A-H, O-Z) ! F77 and F90 Default
```

The effect is automatic even if the statements are omitted. Explicit type declarations override any given IMPLICIT types. For example, if the code had the above implicit defaults one could also explicitly identify the exceptions to those default rules, such as the statements:

```
INTEGER :: Temp_row
```

Storage	MATLAB <sup>a</sup>	C++	F90	F77
byte		char	character::	character
integer		int	integer::	integer
single precision		float	real::	real
double precision		double	real*8::	double precision
complex		<sup>b</sup>	complex::	complex
Boolean		bool	logical::	logical
argument			parameter::	parameter
pointer		*	pointer::	
structure		struct	type::	

<sup>a</sup>MATLAB4 requires no variable type declaration; the only two distinct types in MATLAB are strings and reals (which include complex). Booleans are just 0s and 1s treated as reals. MATLAB5 allows the user to select more types.

<sup>b</sup>There is no specific data type for a complex variable in C++; they must be created by the programmer.

**Table 4.2:** Intrinsic data types of variables

Description	MATLAB <sup>a</sup>	C++	Fortran <sup>b</sup>
addition	+	+	+
subtraction <sup>c</sup>	-	-	-
multiplication	* and .*	*	*
division	/ and ./	/	/
exponentiation	^ and .^	pow <sup>d</sup>	**
remainder		%	
increment		++	
decrement		--	
parentheses (expression grouping)	( )	( )	( )

<sup>a</sup>When doing arithmetic operations on matrices in MATLAB, a period ('.') must be put before the operator if scalar arithmetic is desired. Otherwise, MATLAB assumes matrix operations; figure out the difference between '\*' and '.\*'. Note that since matrix and scalar addition coincide, no '+.' operator exists (same holds for subtraction).

<sup>b</sup>Fortran 90 allows the user to change operators and to define new operator symbols.

<sup>c</sup>In all languages the minus sign is used for negation (i.e., changing sign).

<sup>d</sup>In C++ the exponentiation is calculated by function *pow*(*x*, *y*).

**Table 4.3:** Arithmetic operators

```
REAL :: Interest = 0.04 ! declare and initialize
CHARACTER (Len=8) :: Months_of_year(12)
```

We will also see that the programmer can define new data types and explicitly declare their type as well. The F90 standard discourages the use of any IMPLICIT variables such as

```
IMPLICIT COMPLEX (X-Z)          ! Complex variables
IMPLICIT DOUBLE PRECISION (A-H) ! Double Precision reals
```

and encourages the use of

```
IMPLICIT NONE
```

which forces the programmer to specifically declare the type of each and every variable used, and is referred to as *strong typing*. However, you need to know that such default variable types exist because they are used in many millions of lines of older Fortran code and at some point you will need to use or change such an existing program.

Description	MATLAB	C++	F90	F77
Equal to	==	==	==	.EQ.
Not equal to	~=	!=	/=	.NE.
Less than	<	<	<	.LT.
Less or equal	<=	<=	<=	.LE.
Greater than	>	>	>	.GT.
Greater or equal	>=	>=	>=	.GE.
Logical NOT	~	!	.NOT.	.NOT.
Logical AND	&	&&	.AND.	.AND.
Logical inclusive OR	!		.OR.	.OR.
Logical exclusive OR	xor		.XOR.	.XOR.
Logical equivalent	==	==	.EQV.	.EQV.
Logical not equivalent	~=	!=	.NEQV.	.NEQV.

**Table 4.4:** Relational operators (arithmetic and logical)

MATLAB Operators	C++ Operators	F90 Operators <sup>a</sup>	F77 Operators
()	() [] -> .	()	()
+ -	! ++ -- + - * & (type) sizeof	**	**
* /	* / %	* /	* /
+ - <sup>b</sup>	+ - <sup>b</sup>	+ - <sup>b</sup>	+ - <sup>b</sup>
< <= > >=	<< >>	//	//
== ~=	< <= > >=	== /= < <= > >=	.EQ. .NE. .LT. .LE. .GT. .GE.
~	== !=	.NOT.	.NOT.
&	&&	.AND.	.AND.
		.OR.	.OR.
=		.EQV. .NEQV.	.EQV. .NEQV.
	?:		
	= += -= *= /= %= &= ^=  = <<= >>=		
	,		

<sup>a</sup>User-defined unary (binary) operators have the highest (lowest) precedence in F90.

<sup>b</sup>These are binary operators representing addition and subtraction. Unary operators + and - have higher precedence.

**Table 4.5:** Precedence pecking order

```

[ 1] program main
[ 2] ! Examples of simple arithmetic in F90
[ 3] implicit none
[ 4] integer :: Integer_Var_1, Integer_Var_2      ! user inputs
[ 5] integer :: Mult_Result, Div_Result, Add_Result
[ 6] integer :: Sub_Result, Mod_Result
[ 7] real    :: Pow_Result, Sqrt_Result
[ 8]
[ 9] print *, 'Enter two integers:'
[10] read *, Integer_Var_1, Integer_Var_2
[11]
[12] Add_Result = Integer_Var_1 + Integer_Var_2
[13] print *, Integer_Var_1, ' + ', Integer_Var_2, ' = ', Add_
[14]
[15] Sub_Result = Integer_Var_1 - Integer_Var_2
[16] print *, Integer_Var_1, ' - ', Integer_Var_2, ' = ', Sub_
[17]
[18] Mult_Result = Integer_Var_1 * Integer_Var_2
[19] print *, Integer_Var_1, ' * ', Integer_Var_2, ' = ', Mult_
[20]
[21] Div_Result = Integer_Var_1 / Integer_Var_2
[22] print *, Integer_Var_1, ' / ', Integer_Var_2, ' = ', Div_
[23]
[24] Mod_Result = mod (Integer_Var_1, Integer_Var_2) ! remai
[25] print *, Integer_Var_1, ' mod ', Integer_Var_2, ' = ', Mod_
[26]
[27] Pow_Result = Integer_Var_1 ** Integer_Var_2 ! raise t
[28] print *, Integer_Var_1, ' ^ ', Integer_Var_2, ' = ', Pow_
[29]
[30] Sqrt_Result = sqrt( real(Integer_Var_1))
[31] print *, 'Square root of ', Integer_Var_1, ' = ', Sqrt_Result
[32]
[33] end program main          ! Running produces:
[34] ! Enter two integers:
[35] ! 25 + 4 = 29
[36] ! 25 - 4 = 21
[37] ! 25 * 4 = 100
[38] ! 25 / 4 = 6, note integer
[39] ! 25 mod 4 = 1
[40] ! 25 ^ 4 = 3.9062500E+05
[41] ! Square root of 25 = 5.0000000

```

**Figure 4.1:** Typical Math and Functions in F90

An example program that employs the typical math operators in F90 is shown in Fig. 4.1. It presents examples of addition (line 11), subtraction (line 14), multiplication (line 17), division (line 20), as well as the use of the remainder or modulo function (line 23), exponentiation (line 26), and square root operators (line 29). In addition it shows a way of inputting data from the default input device (line 9). The results are appended as comments (lines 33-40). Observe that a program must include one and only one segment that begins with the word `program` (line 1) and ends with the line `end program` (line 32). If a name is assigned to the program then it must be appended to both of these lines. Often the name of `main` is used, as here, but it is not required as it is in C++. A C++ formulation of this example is included for comparison in the appendix as are several other examples from this chapter.

A special expression available in MATLAB and F90 uses the colon operator (`:`) to indicate forming a vector (row matrix) of numbers according to an arithmetic progression. In MATLAB, the expression `b:i:e` means the vector  $[b \ (b+i) \ (b+2i) \ \cdots \ (b+Ni)]$ , where  $(b+Ni)$  is the largest number less than or equal to (greater than or equal to if  $i$  is negative) the value of the variable  $e$ . Thus, `b` means “beginning value”,  $i$  means the increment, and  $e$  the end value. The expression `b:e` means that the increment equals one. You can use this construct to excise a portion of a vector or matrix. For example, `x(2:5)` equals the vector comprised by the second through fifth elements of `x`, and `A(3:5,i:j)` creates a matrix from the third, fourth, and fifth rows,  $i^{\text{th}}$  through  $j^{\text{th}}$  columns of the matrix `A`. F90 uses the convention of `b:e:i` and has the same defaults when `:i` is omitted. This operator, also known as the *subscript triplet*, is described in Table 4.6.

Of course, expressions often involve the use of functions. A tabulation of the built-in functions in our languages is given in Table 4.7 and the F90 overview, as are all the remaining tables of this chapter. The arguments of functions and subprograms have some important properties that vary with the language used. Primarily, we are interested in how actual arguments are passed to the dummy arguments in the subprogram. This data passing happens by either of two fundamentally different ways: by reference, or

B = Beginning, E = Ending, I = Increment

Syntax	F90	MATLAB	Use	F90	MATLAB
Default	B:E:I	B:I:E	Array subscript ranges	yes	yes
$\geq B$	B:	B:	Character positions in a string	yes	yes
$\leq E$	:E	:E	Loop control	no	yes
Full range	:	:	Array element generation	no	yes

**Table 4.6:** Colon Operator Syntax and its Applications.

Description	MATLAB	C++	F90	F77
exponential	exp(x)	exp(x)	exp(x)	exp(x)
natural log	log(x)	log(x)	log(x)	log(x)
base 10 log	log10(x)	log10(x)	log10(x)	log10(x)
square root	sqrt(x)	sqrt(x)	sqrt(x)	sqrt(x)
raise to power ( $x^r$ )	x.^r	pow(x,r)	x**r	x**r
absolute value	abs(x)	fabs(x)	abs(x)	abs(x)
smallest integer $\geq x$	ceil(x)	ceil(x)	ceiling(x)	
largest integer $\leq x$	floor(x)	floor(x)	floor(x)	
division remainder	rem(x,y)	fmod(x,y)	mod(x,y) <sup>a</sup>	mod(x,y)
modulo			modulo(x,y) <sup>a</sup>	
complex conjugate	conj(z)		conjg(z)	conjg(z)
imaginary part	imag(z)		imag(z)	aimag(z)
drop fraction	fix(x)		aint(x)	aint(x)
round number	round(x)		nint(x)	nint(x)
cosine	cos(x)	cos(x)	cos(x)	cos(x)
sine	sin(x)	sin(x)	sin(x)	sin(x)
tangent	tan(x)	tan(x)	tan(x)	tan(x)
arc cosine	acos(x)	acos(x)	acos(x)	acos(x)
arc sine	asin(x)	asin(x)	asin(x)	asin(x)
arc tangent	atan(x)	atan(x)	atan(x)	atan(x)
arc tangent <sup>b</sup>	atan2(x,y)	atan2(x,y)	atan2(x,y)	atan2(x,y)
hyperbolic cosine	cosh(x)	cosh(x)	cosh(x)	cosh(x)
hyperbolic sine	sinh(x)	sinh(x)	sinh(x)	sinh(x)
hyperbolic tangent	tanh(x)	tanh(x)	tanh(x)	tanh(x)
hyperbolic arc cosine	acosh(x)			
hyperbolic arc sine	asinh(x)			
hyperbolic arctan	atanh(x)			

<sup>a</sup>Differ for  $x < 0$ .

<sup>b</sup>atan2(x,y) is used to calculate the arc tangent of  $x/y$  in the range  $[-\pi, +\pi]$ . The one-argument function atan(x) computes the arc tangent of  $x$  in the range  $[-\pi/2, +\pi/2]$ .

**Table 4.7:** Mathematical functions

by value. One should understand the difference between these two mechanisms.

“Passing by reference” means that the address in memory of the actual argument is passed to the subprogram instead of the value stored at that address. The corresponding dummy argument in the subprogram has the same address. That is, both arguments refer to the same memory location so any change to that argument within the subprogram is passed back to the calling code. A variable is passed by reference to a subroutine whenever it is expected that it should be changed by the subprogram. A related term is “dereferencing”. When you dereference a memory address, you are telling the computer to get the information located at the address. Typically, one indirectly gives the address by citing the

<i>Description</i>	<b>C++</b>	<b>F90</b>	<b>F77</b>	<b>MATLAB</b>
Conditionally execute statements	if { }	if end if	if end if	if end
Loop a specific number of times	for k=1:n { }	do k=1,n end do	do # k=1,n # continue	for k=1:n end
Loop an indefinite number of times	while { }	do while end do	— —	while end
Terminate and exit loop	break	exit	go to	break
Skip a cycle of loop	continue	cycle	go to	—
Display message and abort	error()	stop	stop	error
Return to invoking function	return	return	return	return
Conditional array action	—	where	—	if
Conditional alternate statements	else else if	else elseif	else elseif	else elseif
Conditional array alternatives	— —	elsewhere —	— —	else elseif
Conditional case selections	switch { }	select case end select	if end if	if end

**Table 4.8:** Flow Control Statements.

name of a pointer variable or a reference variable.

“Passing by value” means that the value of the actual argument stored at its address in memory is copied and the copy is passed to the dummy argument in the subprogram. Thus any change to the argument within the subprogram is **not** passed back to the calling code. The two passing methods do not clearly show the intended use of the argument within the subprogram. Is it to be passed in for use only, passed in for changing and returned, or is it to be created in the subprogram and passed out for use in the calling code? For additional safety and clarity modern languages provide some way to allow the programmer to optionally specify such intent explicitly.

Both C++ and MATLAB use the pass by value method as their default mode. This means the value associated with the argument name, say `arg_name`, is copied and passed to the function. That copying could be very inefficient if the argument is a huge array. To denote that you want to have the C++ argument passed by reference you must precede the argument name with an ampersand (&), e.g. `&arg_name`, in the calling code. Then within the subprogram the corresponding dummy variable must be dereferenced by preceding the name with an asterisk (\*), e.g. `*arg_name`. Conversely, Fortran uses the passing by reference method as its default mode. On the rare occasions when one wants to pass by value simply surround the argument name with parentheses, e.g. `(arg_name)`, in the calling code. In either case it is recommended that you cite each argument with the optional “intent” statement within the subprogram. Examples of the two passing options are covered in Sec. 4.5.

## 4.3 Flow Control

The basic flow control constructs present in our selected engineering languages are *loops*—repetitive execution of a block of statements—and *conditionals*—diversions around blocks of statements. A typical set of flow control statement types are summarized in Table 4.8. Most of these will be illustrated in detail in the following sections.

### 4.3.1 Explicit Loops

The following discussion will introduce the important concept of loops. These are required in most programs. However, the reader is warned that today the writing of explicit loops are generally *not* the most efficient way to execute a loop operation in Fortran90 and MATLAB. Of course, older languages like F77 and C do require them, so that the time spent here not only covers the explicit loop concepts but aids one in reading older languages. Our pseudocode for the common loops is :

Loop	MATLAB	C++	Fortran
Indexed loop	for index=matrix statements end	for (init;test;inc) { statements }	do index=b,e,i statements end do
Pre-test loop	while test statements end	while (test) { statements }	do while (test) statements end do
Post-test loop		do { statements } while (test)	do statements if (test) exit end do

**Table 4.9:** Basic loop constructs

Loop	Pseudocode
Indexed loop	for index=b,i,e statements end for
Pre-test loop	while (test) statements end while
Post-test loop	do statements if test exit end do

In engineering programming one often needs to repeatedly perform a group of operations. Most computer languages have a statement to execute this powerful and widely-used feature. In Fortran this is the DO statement, while in C++ and MATLAB it is the FOR statement. This one statement provides for the initialization, incrementing and testing of the loop variable, plus repeated execution of a group of statements contained within the loop. In Fortran77, the loop always cites a label number that indicates the extent of the statements enclosed in the loop. This is allowed in F90, but not recommended, and is considered obsolete. Instead, the END DO indicates the extent of the loop, and the number label is omitted in both places. F90 does allow one to give a name to a loop. Then the structure is denoted as NAME:DO followed by END DO NAME. Examples of the syntax for these statements for the languages of interest are given in Table 4.9.

A simple example of combining loops and array indexing is illustrated in Figs. 4.2 and 4.3. Note in Fig. 4.2 that the final value of a loop counter (called *Integer\_Var* here) upon exiting the loop (line 10) can be language or compiler dependent despite the fact that they are same here. In Fig. 4.3, we introduce for the first time a variable with a single subscript (line 5) and containing five numbers (integers) to be manually initialized (lines 8-10) and then to be listed in a loop (lines 12-15) over all their values. Note that C++ stores the first entry in an array at position zero (see appendix listing), MATLAB uses position one, and F90 defaults to position one.

C++ and Fortran 90 allow a special option to create loops that run “forever.” These could be used, for example, to read an unknown amount of data until terminated, in a non-fatal way, by the input statement. In C++, one omits the three loop controls, such as

```
for (;;) { // forever loop
    loop_block
} // end forever loop
```

while in F90, one simply omits the loop control and gives only the DO command:

```
do ! forever
```



```

[ 1] program main
[ 2] ! Examples of a simple loop in F90
[ 3] implicit none
[ 4]   integer Integer_Var
[ 5]
[ 6]   do Integer_Var = 0,4,1
[ 7]     print *, 'The loop variable is:', Integer_Var
[ 8]   end do ! over Integer_Var
[ 9]
[10]   print *, 'The final loop variable is:', Integer_Var
[11]
[12] end program main                ! Running produces:
[13] ! The loop variable is: 0
[14] ! The loop variable is: 1
[15] ! The loop variable is: 2
[16] ! The loop variable is: 3
[17] ! The loop variable is: 4
[18] ! The final loop variable is: 5 <- NOTE

```

**Figure 4.2:** Typical Looping Concepts in F90

```

[ 1] program main
[ 2] ! Examples of simple array indexing in F90
[ 3] implicit none
[ 4]   integer, parameter :: max = 5
[ 5]   integer Integer_Array(max) ! =(/ 10 20 30 40 50 /), or set below
[ 6]   integer loopcount
[ 7]
[ 8]   Integer_Array(1) = 10    ! F90 index starts at 1, usually
[ 9]   Integer_Array(2) = 20 ; Integer_Array(3) = 30
[10]   Integer_Array(4) = 40 ; Integer_Array(5) = 50
[11]
[12]   do loopcount = 1, max      ! & means continued
[13]     print *, 'The loop counter is: ', loopcount, &
[14]       ' with an array value of: ', Integer_Array(loopcount)
[15]   end do ! over loopcount
[16]
[17]   print *, 'The final loop counter is: ', loopcount
[18]
[19] end program main
[20] ! Running produces:
[21] ! The loop counter is: 1 with an array value of: 10
[22] ! The loop counter is: 2 with an array value of: 20
[23] ! The loop counter is: 3 with an array value of: 30
[24] ! The loop counter is: 4 with an array value of: 40
[25] ! The loop counter is: 5 with an array value of: 50
[26] ! The final loop counter is: 6

```

**Figure 4.3:** Simple Array Indexing in F90

```

      loop_block
end do ! forever

```

Most of the time, an infinite loop is used as a *loop\_while\_true* or a *loop\_until\_true* construct. These will be considered shortly.

### 4.3.2 Implied Loops

Fortran and MATLAB have shorthand methods for constructing “implied loops.” Both languages offer the colon operator to imply an incremental range of integer values. Its syntax and types of applications are given in Table 4.6 (page 56). The allowed usages of the operator differ slightly between the two languages. Note that this means that the loop controls are slightly different in that the *do* control employs commas instead of colons. For example, two equivalent loops are

Fortran	MATLAB
do k=B,E,I A(k) = k**2 end do	for k=B:I:E A(k) = k^2 end

Fortran offers an additional formal implied *do* loop that replaces the *do* and *end do* with a closed pair of parentheses in the syntax:

```
(object, k = B,E,I)
```

where again the increment, `I`, defaults to unity if not supplied. The above implied `do` is equivalent to the formal loop

```
do k=B,E,I
  define object
end do
```

However, the object defined in the implied loop can only be utilized for four specific Fortran operations: 1) read actions, 2) print and write actions, 3) data variables (not value) definitions, and 4) defining array elements. For example,

```
print *, (4*k-1, k=1,10,3) ! 3, 15, 27, 39
read *, (A(j,:), j=1,rows) ! read A by rows, sequentially
```

The implied `do` loops can be nested to any level like the standard `do` statement. One simply makes the inner loop the object of the outer loop, so that

```
((object_j_k, j=min, max), k=k1,k2,inc)
```

implies the nested loop

```
do k=k1,k2,inc
  do j=min, max
    use object_j_k
  end do ! over j
end do ! over k
```

For example,

```
print *, ((A(k)*B(j)+3), j=1,5), k=1,max)
! read array by rows in each plane
read *, ((A(i,j,k), j=1,cols), i=1,rows), k=1,max)
```

Actually, there is even a simpler default form of implied `dos` for reading and writing arrays. That default is to access arrays by columns. That is, process the leftmost subscript first. Thus, for an array with three subscripts,

```
read *, A  $\iff$  read *, ((A(i,j,k), i=1,rows), j=1,cols), k=1,planes)
```

Both languages allow the implied loops to be employed to create an array vector simply by placing the implied loop inside the standard array delimit symbols. For example, we may want an array to equally distribute  $N + 1$  points over the distance from zero to  $D$ .

```
F90:      X = ((k,k=0,N))/) * D/(N+1)
MATLAB:  X = [0:N] * D / (N+1),
```

which illustrates that MATLAB allows the use of the colon operator to define arrays, but F90 does not.

In addition to locating elements in an array by the regular incrementing of loop variables, both Fortran90 and MATLAB support even more specific selections of elements: by random location via vector subscripts, or by value via logical masks such as `where` and `if` in F90 and MATLAB, respectively.

### 4.3.3 Conditionals

Logic tests are frequently needed to control the execution of a block of statements. The most basic operation occurs when we want to do something when a logic test gives a true answer. We call that a simple `IF` statement. When the test is true, the program executes the block of statements following the `IF`. Often only one statement is needed, so C++ and Fortran allow that one statement to end the line that begins with the `IF` logic. Frequently we will *nest* another `IF` within the statements from a higher level `IF`. The common language syntax forms for the simple `IF` are given below in Table 4.10, along with the examples of where a second true group is nested inside the first as shown in Table 4.11.

The next simplest case is where we need to do one thing when the answer is true, and a different thing when the logic test is false. Then the syntax changes simply to an `IF {true group} ELSE {false group}` mode of execution. The typical `IF-ELSE` syntaxes of the various languages are given in Table 4.12. Of course, the above statement groups can contain other `IF` or `IF-ELSE` statements nested within them. They can also contain any valid statements, including `DO` or `FOR` loops.

The most complicated logic tests occur when the number of cases for the answer go beyond the two (true-false) of the `IF-ELSE` control structure. These multiple case decisions can be handled with the `IF-ELSEIF-ELSE` control structures whose syntax is given in Table 4.13. They involve a sequence of logic

MATLAB	Fortran	C++
<pre>if l_expression     true_group end</pre>	<pre>IF (l_expression) THEN     true_group END IF</pre>	<pre>if (l_expression) {     true_group; }</pre>
	<pre>IF (l_expression) true_statement</pre>	<pre>if (l_expression)     true_statement;</pre>

**Table 4.10:** IF Constructs. The quantity *l\_expression* means a logical expression having a value that is either TRUE or FALSE. The term *true\_statement* or *true\_group* means that the statement or group of statements, respectively, are executed if the conditional in the *if* statement evaluates to TRUE.

MATLAB	Fortran	C++
<pre>if l_expression1     true_group A if l_expression2     true_group B end     true_group C end statement_group D</pre>	<pre>IF (l_expression1) THEN     true_group A IF (l_expression2) THEN     true_group B END IF     true_group C END IF statement_group D</pre>	<pre>if (l_expression1) {     true_group A if (l_expression2) {     true_group B }     true_group C } statement_group D</pre>

**Table 4.11:** Nested IF Constructs.

MATLAB	Fortran	C++
<pre>if l_expression     true_group A else     false_group B end</pre>	<pre>IF (l_expression) THEN     true_group A ELSE     false_group B END IF</pre>	<pre>if (l_expression) {     true_group A } else {     false_group B }</pre>

**Table 4.12:** Logical IF-ELSE Constructs.

tests, each of which is followed by a group of statements that are to be executed if, and only if, the test answer is true. There can be any number of such tests. They are terminated with an ELSE group of default statements to be executed if *none* of the logic tests are true. Actually, the ELSE action is optional. For program clarity or debugging, it should be included even if it only prints a warning message or contains a comment statement. Typical “if” and “if-else” coding is given in Figs. 4.4, 4.5, and 4.6. Figure 4.4 simply uses the three logical comparisons of “greater than” (line 9), “less than” (line 12), or “equal to” (line 15), respectively. Figure 4.5 goes a step further by combining two tests with a logical “and” test (line 9), and includes a second else branch (line 11) to handle the case where the *if* is false. While the input to these programs were numbers (line 7), the third example program in Fig. 4.6 accepts logical input (lines 6,8) that represents either true or false values and carries out Boolean operations to negate an input (via NOT in line 9), or to compare two inputs (with an AND in line 11, or OR in line 17, etc.) to produce a third logical value.

Since following the logic of many IF-ELSEIF-ELSE statements can be very confusing both the C++ and Fortran languages allow a CASE selection or “switching” operation based on the value (numerical or character) of some expression. For any allowed specified CASE value, a group of statements is executed. If the value does not match any of the specified allowed CASE values, then a default group of statements are executed. These are illustrated in Table 4.14.

MATLAB	Fortran	C++
<pre> if l_expression1   true group A elseif l_expression2   true group B elseif l_expression3   true group C else   default group D end </pre>	<pre> IF (l_expression1) THEN   true group A ELSE IF (l_expression2) THEN   true group B ELSE IF (l_expression3) THEN   true group C ELSE   default group D END IF </pre>	<pre> if (l_expression1) {   true group A } else if (l_expression2) {   true group B } else if (l_expression3) {   true group C } else {   default group D } </pre>

**Table 4.13:** Logical IF-ELSE-IF Constructs.

```

[ 1] program main
[ 2] ! Examples of relational "if" operator in F90
[ 3] implicit none
[ 4]   integer :: Integer_Var_1, Integer_Var_2      ! user inputs
[ 5]
[ 6]   print *, 'Enter two integers:'
[ 7]   read  *, Integer_Var_1, Integer_Var_2
[ 8]
[ 9]   if ( Integer_Var_1 > Integer_Var_2 ) &
[10]     print *, Integer_Var_1, ' is greater than ', Integer_Var_2
[11]
[12]   if ( Integer_Var_1 < Integer_Var_2 ) &
[13]     print *, Integer_Var_1, ' is less than ', Integer_Var_2
[14]
[15]   if ( Integer_Var_1 == Integer_Var_2 ) &
[16]     print *, Integer_Var_1, ' is equal to ', Integer_Var_2
[17]
[18] end program main
[19]
[20] ! Running with 25 and 4 produces:
[21] ! Enter two integers:
[22] ! 25  is greater than 4

```

**Figure 4.4:** Typical Relational Operators in F90

```

[ 1] program main
[ 2] ! Illustrate a simple if-else logic in F90
[ 3] implicit none
[ 4]   integer Integer_Var
[ 5]
[ 6]   print *, 'Enter an integer: '
[ 7]   read  *, Integer_Var
[ 8]
[ 9]   if ( Integer_Var > 5 .and. Integer_Var < 10 ) then
[10]     print *, Integer_Var, ' is greater than 5 and less than 10'
[11]   else
[12]     print *, Integer_Var, ' is not greater than 5 and less than 10'
[13]   end if ! range of input
[14]
[15] end program main
[16] !
[17] ! Running with 3 gives: 3  is not greater than 5 and less than 10
[18] ! Running with 8 gives: 8  is greater than 5 and less than 10

```

**Figure 4.5:** Typical If-Else Uses in F90

Fortran90 offers an additional optional feature called *construct names* that can be employed with the above IF and SELECT CASE constructs to improve the readability of the program. The optional name, followed by a colon, precedes the key words IF and SELECT CASE. To be consistent, the name should also follow the key words END IF or END SELECT which always close the constructs. The construct name option also is available for loops where it offers an additional pair of control actions that will be explained later. Examples of these optional F90 features are given in Table 4.15.

While C++ and MATLAB do not formally offer this option, the same enhancement of readability can

```

[ 1] program main
[ 2] ! Examples of Logical operators in F90
[ 3] implicit none
[ 4] logical :: Logic_Var_1, Logic_Var_2
[ 5] print *, 'Print logical value of A (T or F):'
[ 6] read *, Logic_Var_1
[ 7] print *, 'Print logical value of B (T or F):'
[ 8] read *, Logic_Var_2
[ 9] print *, 'NOT A is ', (.NOT. Logic_Var_1)
[10]
[11] if ( Logic_Var_1 .AND. Logic_Var_2 ) then
[12]   print *, 'A ANDED with B is true'
[13] else
[14]   print *, 'A ANDED with B is false'
[15] end if ! for AND
[16]
[17] if ( Logic_Var_1 .OR. Logic_Var_2 ) then
[18]   print *, 'A ORed with B is true'
[19] else
[20]   print *, 'A ORed with B is false'
[21] end if ! for OR
[22]
[23] if ( Logic_Var_1 .EQV. Logic_Var_2 ) then
[24]   print *, 'A EQiValent with B is true'
[25] else
[26]   print *, 'A EQiValent with B is false'
[27] end if ! for EQV
[28]
[29] if ( Logic_Var_1 .NEQV. Logic_Var_2 ) then
[30]   print *, 'A Not EQiValent with B is true'
[31] else
[32]   print *, 'A Not EQiValent with B is false'
[33] end if ! for NEQV
[34]
[35] end program main
[36] ! Running with T and F produces:
[37] ! Print logical value of A (T or F): T
[38] ! Print logical value of B (T or F): F
[39] ! NOT A is F
[40] ! A ANDED with B is false
[41] ! A ORed with B is true
[42] ! A EQiValent with B is false
[43] ! A Not EQiValent with B is true

```

**Figure 4.6:** Typical Logical Operators in F90

F90	C++
<pre> SELECT CASE (expression)   CASE (value 1)     group 1   CASE (value 2)     group 2     :   CASE (value n)     group n   CASE DEFAULT     default group END SELECT </pre>	<pre> switch (expression) {   case value 1 :     group 1     break;   case value 2 :     group 2     break;     :   case value n :     group n     break;   default:     default group     break; } </pre>

**Table 4.14:** Case Selection Constructs.

be achieved by using the trailing comment feature to append a name or description at the beginning and end of these logic construct blocks.

Both C++ and Fortran allow statement labels and provide controls to branch to specific labels. Today you are generally advised **not** to use a GO TO and its associated label! However, they are common in many F77 codes. There are a few cases where a GO TO is still considered acceptable. For example, the pseudo-WHILE construct of F77 requires a GO TO.

F90 Named IF	F90Named SELECT
<pre> name: IF (logical_1) THEN   true group A ELSE IF (logical_2) THEN   true group B ELSE   default group C ENDIF name </pre>	<pre> name: SELECT CASE (expression)   CASE (value 1)     group 1   CASE (value 2)     group 2   CASE DEFAULT     default group END SELECT name </pre>

**Table 4.15:** F90 Optional Logic Block Names.

Fortran	C++
<pre> DO 1 ... DO 2 ...   IF (disaster) THEN     GO TO 3   END IF   ... 2 END DO 1 END DO 3 next statement </pre>	<pre> for (...) {   for (...) {     if (disaster)       go to error     ...   } } error: </pre>

**Table 4.16:** GO TO Break-out of Nested Loops. This situation can be an exception to the general recommendation to avoid GO TO statements.

F77	F90	C++
<pre> DO 1 I = 1,N   IF (skip condition) THEN     GO TO 1   ELSE     false group   END IF 1 continue </pre>	<pre> DO I = 1,N   IF (skip condition) THEN     CYCLE ! to next I   ELSE     false group   END IF END DO </pre>	<pre> for (i=1; i&lt;n; i++) {   if (skip condition)     continue; // to next   else if     false group   end } </pre>

**Table 4.17:** Skip a Single Loop Cycle.

```

initialize test
IF (l_expression) THEN
  true statement group
  modify logical value
  GO TO #
END IF

```

The GO TO can also be effectively utilized in both Fortran and C++ to break out of several nested loops. This is illustrated in Table 4.16. The “break-out” construct can be used in the situation when, as a part of a subroutine, you wanted the program exit the loop and also exit the subroutine, returning control to the calling program. To do that, one would simply replace the GO TO statement with the RETURN statement. In F90, one should also append the comment “! to calling program” to assist in making the subroutine more readable.

You may find it necessary to want to skip a cycle in loop execution and/or exit from a single loop. Both Fortran and C++ provide these control options without requiring the use of a GO TO. To skip a loop cycle, Fortran90 and C++ use the statements CYCLE and continue, respectively, and EXIT and break to abort a loop. These constructs are shown in Tables 4.17 and 4.18. Other forms of the GO TO in F77 were declared obsolete in F90, and should not be used. The Fortran abort examples could also use the RETURN option described above in the rare cases when it proves to be more desirable or efficient.

As mentioned earlier, F90 allows the programmer to use “named” DO constructs. In addition to im-

F77	F90	C++
<pre>DO 1 I = 1,N   IF (exit condition) THEN     GO TO 2   ELSE     false group   END IF 1 CONTINUE 2 next statement</pre>	<pre>DO I = 1,N   IF (exit condition) THEN     EXIT ! this do   ELSE     false group   END IF END DO next statement</pre>	<pre>for (i=1; i&lt;n; i++) {   if (exit condition)     break;// out of loop   else if     false group   end } next statement</pre>

**Table 4.18:** Abort a Single Loop.

```
main: DO ! forever
test: DO k=1,k_max
  third: DO m=m_max,m_min,-1
    IF (test condition) THEN
      CYCLE test ! loop on k
    END IF
  END DO third ! loop on m
  fourth: DO n=n_min,n_max,2
    IF (main condition) THEN
      EXIT main ! forever loop
    END DO fourth ! on n
  END DO test ! over k
END DO main
next statement
```

**Table 4.19:** F90 DOs Named for Control.

proving readability, this feature also offers additional control over nested loops because we can associate the `CYCLE` and `EXIT` commands with a specific loop (Table 4.19). Without the optional name, the `CYCLE` and `EXIT` commands act only on the inner-most loop in which they lie. We will see later that Fortran90 allows another type of loop called `WHERE` that is designed to operate on arrays.

#### 4.3.3.1 Looping While True or Until True

It is very common to need to perform a loop so long as a condition is true, or to run the loop until a condition becomes true. The two are very similar and both represent loops that would run forever unless specifically terminated. We will refer to these two approaches as `WHILE` loops and `UNTIL` loops. The `WHILE` logic test is made first in order to determine if the loop will be entered. Clearly, this means that if the logic test is false the first time it is tested, then the statement blocks controlled by the `WHILE` are never executed. If the `WHILE` loop is entered, something in the loop must eventually change the value of a variable in the logic test or the loop would run forever. Once a change causes the `WHILE` logic test to be false control is transferred to the first statement following the `WHILE` structure. By way of comparison, an `UNTIL` loop is always entered at least once. Upon entering the loop, a beginning statement group is executed. Then the logic test is evaluated. If the test result is true, the loop is exited and control is passed to the next statement after the group. If the test is false, then an optional second statement group is executed before the loop returns to the beginning statement group. The pseudo-code for these two similar structures are given as follows :

while true	until true
<pre>logic_variable = true begin: if (logic_variable) then % true   true_group   re-evaluate logic_variable go to begin else % false   exit loop end if</pre>	<pre>logic_variable = false begin: statements if (logic_variable) then   exit the loop else % false   false_group   re-evaluate logic_variable go to begin end if</pre>

Since these constructs are commonly needed, several programming languages offer some support for them. For example, Pascal has a REPEAT UNTIL command and C++ has the DO-WHILE pair for the until-true construct. For the more common while-true loops, C++ and MATLAB offer a WHILE command, and Fortran 90 includes the DO WHILE. F77, however, only has the obsolete IF-GO TO pairs as illustrated in a previous example. Many current programmers consider the WHILE construct obsolete because it is less clear than a DO-EXIT pair or a “for-break” pair. Indeed, the F90 standard has declared the DO WHILE as obsolete and eligible for future deletion from the language. We can see how the loop-abort feature of C++ and F90 includes both the WHILE and UNTIL concepts. For example, the F90 construct

```

initialize logical_variable
DO WHILE (logical_variable) ! is true
  true_group
  re-evaluate logical_variable
END DO ! while true
:

```

is entirely equivalent to the aborted endless loop

```

initialize logical_variable
DO ! forever while true
  IF (.NOT. logical_variable) EXIT ! as false
  true_group
  re-evaluate logical_variable
END DO ! while true
:

```

Likewise, a minor change includes the UNTIL construct.

```

DO ! forever until true
  beginning statements and initialization
  IF (logical_expression) EXIT ! as true
  false_group
  re-evaluate logical_variable
END DO ! until true

```

When approached in the C++ language, we have the WHILE loop.

```

initialize logical_variable
while (logical_variable)
{ // is true
  true_group
  re-evaluate logical_variable
} // end while true

```

Recalling the standard for syntax,

```

for (expr_1; expr_2; expr_3)
{
  true_group
} // end for

```

could be viewed as equivalent to the above WHILE in for form.

```

expr_1;
while (expr_2)
{ // is true
  true_group
  expr_3;
} // end while true

```

If one omits all three for expressions, then it becomes an “infinite loop” or a “do forever” which can represent a WHILE or UNTIL construct by proper placement of the break command. Furthermore, C has the do-while construct that is equivalent to Pascal’s REPEAT-UNTIL.

```

do // forever until true
  statements
  evaluate logical_variable
while (logical_variable) // is true

```

The syntax for the classical WHILE statements in C++, Fortran and MATLAB are given in Table 4.20. Fortran90 has declared the DO WHILE as obsolete, and recommends the DO-EXIT pair instead! Using infinite loops with clearly aborted stages is a less error-prone approach to programming.



MATLAB	C++
<pre> initialize test while l_expression   true group   change test end </pre>	<pre> initialize test while (l_expression) {   true group   change test } </pre>
F77	F90
<pre> initialize test # continue IF (l_expression) THEN   true group   change test go to # END IF </pre>	<pre> initialize test do while (l_expression)   true group   change test end do </pre>

**Table 4.20:** Looping While a Condition is True.

Function Type	MATLAB <sup>a</sup>	C++	Fortran
program	<pre> statements [y1...yn]=f(a1,...,am) [end of file] </pre>	<pre> main(argc,char **argv) {   statements   y = f(a1,I,am); } </pre>	<pre> program main type y type a1,...,type am statements y = f(a1,...,am) call s(a1,...,am) end program </pre>
subroutine		<pre> void f (type a1,...,type am) {   statements } </pre>	<pre> subroutine s(a1,...,am) type a1,...,type am statements end </pre>
function	<pre> function [r1...rn] =f(a1,...,am) statements </pre>	<pre> type f (type a1,...,type am) { statements } </pre>	<pre> function f(a1,...,am) type f type a1,...,type am statements end </pre>

<sup>a</sup>Every function or program in MATLAB must be in separate files.

**Table 4.21:** Function definitions. In each case, the function being defined is named *f* and is called with *m* arguments *a1*, ..., *am*.

## 4.4 Subprograms

The concept of modular programming requires the use of numerous subprograms or procedures to execute independent segments of the calculations or operations. Typically, these procedures fall into classes such as functions, subroutines, and modules. We will consider examples of the procedures for each of our target languages. These are shown in Table 4.21.

Recall that Table 8.6 compared several intrinsic functions that are common to both F90 and MATLAB. For completeness, all of the Fortran90 functions are listed both alphabetically and by subject in Appendix B. Similar listings for MATLAB can be found in the *MATLAB Primer*.

### 4.4.1 Functions and Subroutines

Historically, a function was a subprogram that employed one or more input arguments and returned a single result value. For example, a square root or logarithm function would accept a single input value and return a single result. All of the languages of interest allow the user to define such a function, and they

One-Input, One-Result Procedures	
MATLAB	function out = name (in)
F90	function name (in) ! name = out
	function name (in) result (out)
C++	name (in, out)*

Multiple-Input, Multiple-Result Procedures	
MATLAB	function [inout, out2] = name (in1, in2, inout)
F90	subroutine name (in1, in2, inout, out2)*
C++	name(in1, in2, inout, out2)*

\* Other arrangements acceptable

**Table 4.22:** Arguments and return values of subprograms.

all provide numerous intrinsic or built-in functions of this type. As you might expect, such a procedure is called a *function* in C++, Fortran and MATLAB. As an example of such a procedure, consider the calculation of the mean value of a sequence of numbers defined as

$$\text{mean} = \frac{1}{n} \sum_{k=1}^n x_k .$$

In Fortran90, a subprogram to return the mean (average) could be

```
function mean(x)
! mean = sum of vector x, divided by its size
  real :: mean, x(:)
  mean = sum(x)/size(x)
end function mean
```

Note that our function has employed two other intrinsic functions: `size` to determine the number of elements in the array `x`, and `sum` to carry out the summation of all elements in `x`. Originally in Fortran, the result value was required to be assigned to the name of the function. That is still a valid option in F90, but today it is considered better practice to specify a result value name to be returned by the function. The `mean` function is a MATLAB intrinsic and can be used directly.

To illustrate the use of a result value, consider the related “median” value in F90.

```
function mid_value(x) result(median)
! return the middle value of vector x
  real :: median, x(:)
  median = x(size(x)/2) ! what if size = 1 ??
end function mid_value
```

To apply these two functions to an array, say `y`, we would simply write `y_ave = mean(y)`, and `y_mid = mid_value(y)`, respectively. While Fortran allows a “function” to return only a single object, both C++ and MATLAB use that subprogram name to return any number of result objects. Fortran employs the name “subroutine” for such a procedure. Such procedures are allowed to have multiple inputs and multiple outputs (including none). The syntax of the first line of these two subprogram classes are shown in Table 4.22. Note that a typical subprogram may have no arguments, multiple input arguments (`in1`, `in2`, `inout`), multiple result arguments (`inout`, `out2`), and arguments that are used for both input and result usage (`inout`). These example names have been selected to reflect the fact that a programmer usually intends for arguments to be used for input only, or for result values only, or for input, modification, and output. It is considered good programming practice to declare such intentions to aid the compiler in detecting unintended uses. F90 provides the `INTENT` statement for this purpose, but does not require its use.

Having outlined the concepts of subprograms, we will review some presented earlier and then give some new examples. Figure 1.3 presented a clipping function which was earlier expressed in pseudocode. A corresponding Fortran implementation of such a clipping function is given in Fig. 4.7. Note that it is very similar to the pseudocode version.

```

[ 1] program main
[ 2] !   clip the elements of an array
[ 3] implicit none
[ 4]   real, parameter :: limit = 3
[ 5]   integer, parameter :: n = 5
[ 6]   real :: y(n), x(n)
[ 7] !   Define x values that will be clipped
[ 8]   x = (/ (-8. + 3.*k, k = 1,n) /) ! an implied loop
[ 9]   do i = 1, n
[10]     y(i) = clip (x(i), limit)
[11]   end do
[12]   print *, x
[13]   print *, y
[14]
[15] contains ! methods
[16]
[17] function clip (x, L) result (c)
[18] !   c = clip(x, L) - clip the variable x, output
[19] !   x = scalar variable, input
[20] !   L = limit of the clipper, input
[21] !
[22]   real, intent(in) :: x, L ! variable types
[23]   real :: c ! variable types
[24]   intent (in) x, L ! argument uses
[25]   if ( abs(x) <= L ) then ! abs of x less than or equal L
[26]     c = x; ! then use x
[27]   else ! absolute of x greater than L ?
[28]     c = sign(L,x) ! sign of x times L
[29]   end if ! of value of x
[30]   end function ! clip
[31] end program main
[32] !
[33] ! produces:
[34] ! -5.00000000 -2.00000000 1.00000000 4.00000000 7.00000000
[35] ! -3.00000000 -2.00000000 1.00000000 3.00000000 3.00000000

```

**Figure 4.7:** Clipping a Set of Array Values in F90

For the purpose of illustration an alternate F90 version of the Game of Life, shown earlier in Chapter 1 as pseudocode, is given in the assignment solutions section. Clearly we have not introduced all the features utilized in these example codes so the reader should continue to refer back to them as your programming understanding grows.

A simple program that illustrates program composition is `maximum.f90`, which asks the user to specify several integers from which the program finds the largest. It is given in Fig. 4.8. Note how the main program accepts the user input (lines 15,20), with the `maxint` function (line 22) finding the maximum (lines 25-34). Perhaps modularity would have been better served by expressing the input portion by a separate function. Of course, this routine is not really needed since F90 provides intrinsic functions to find maximum and minimum values (`maxval`, `minval`) and their locations in any array (`maxloc`, `minloc`). A similar C++ program composition is shown for comparison in the appendix.

```

[ 1] program maximum ! of a set of integers (see intrinsic maxval)
[ 2]   implicit none
[ 3]   interface ! declare function interface prototype
[ 4]     function maxint (input, input_length) result(max)
[ 5]       integer, intent(in) :: input_length, input(:)
[ 6]       integer              :: max
[ 7]     end function ! maxint
[ 8]   end interface
[ 9]
[10]   integer, parameter :: ARRAYLENGTH=100
[11]   integer              :: integers(ARRAYLENGTH);
[12]   integer              :: i, n;
[13]
[14]   ! Read in the number of integers
[15]   print *, 'Find maximum; type n: '; read *, n
[16]   if ( n > ARRAYLENGTH .or. n < 0 ) &
[17]     stop 'Value you typed is too large or negative.'
[18]
[19]   do i = 1, n                ! Read in the user's integers
[20]     print *, 'Integer ', i, '?'; read *, integers(i)
[21]   end do ! over n values
[22]   print *, 'Maximum: ', maxint (integers, n)
[23] end program maximum
[24]
[25] function maxint (input, input_length) result(max)
[26] ! Find the maximum of an array of integers
[27] integer, intent(in) :: input_length, input(:)
[28] integer              :: i, max
[29]
[30]   max = input(1); ! initialize
[31]   do i = 1, input_length ! note could be only 1
[32]     if ( input(i) > max ) max = input(i);
[33]   end do ! over values
[34] end function maxint ! produces this result:
[35] ! Find maximum; type n: 4
[36] ! Integer 1? 9
[37] ! Integer 2? 6
[38] ! Integer 3? 4
[39] ! Integer 4? -99
[40] ! Maximum: 9

```

**Figure 4.8:** Search for Largest Value in F90

Global Variable Declaration	
MATLAB	global list of variables
F77	common /set_name/ list of variables
F90	module set_name save type (type_tag) :: list of variables end module set_name
C++	extern list of variables

Access to Global Variables	
MATLAB	global list of variables
F77	common /set_name/ list of variables
F90	use set_name, only subset of variables use set_name2 list of variables
C++	extern list of variables

**Table 4.23:** Defining and referring to global variables.

## 4.4.2 Global Variables

We have seen that variables used inside a procedure can be thought of as dummy variable names that exist only in the procedure, unless they are members of the argument list. Even if they are arguments to the procedure, they can still have names different from the names employed in the calling program. This approach can have disadvantages. For example, it might lead to a long list of arguments, say 20 lines, in a complicated procedure. For this and other reasons, we sometimes desire to have variables that are accessible by any and all procedures at any time. These are called *global variables* regardless of their type.

Generally, we explicitly declare them to be global and provide some means by which they can be accessed, and thus modified, by selected procedures. When a selected procedure needs, or benefits from, access to a global variable, one may wish to control which subset of global variables are accessible by the procedure. The typical initial identification of global variables and the ways to access them are shown in Table 4.23, respectively.

An advanced aspect of the concept of global variables are the topics of inheritance and object-oriented programming. Fortran90, and other languages like C++, offer these advanced concepts. In F90, inheritance is available to a module and/or a main program and their “internal sub-programs” defined as those procedures following a `contains` statement, but occurring before an `end module` or the `end program` statement. Everything that appears before the `contains` statement is available to, and can be changed by, the internal sub-programs. Those inherited variables are more than local in nature, but not quite global; thus, they may be thought of as *territorial* variables. The structure of these internal sub-programs with inheritance is shown in Fig. 4.9

Perhaps the most commonly used global variables are those necessary to calculate the amount of central processor unit (cpu) time, in seconds, that a particular code segment used during its execution. All systems provide utilities for that purpose but some are more friendly than others. MATLAB provides a pair of functions, called `tíc` and `toc`, that act together to provide the desired information. To illustrate the use of global variables we will develop a F90 module called `tíc_toc` to hold the necessary variables along with the routines `tíc` and `toc`. It is illustrated in Fig. 4.10 where the module constants (lines 2-6) are set (lines 17, 26) and computed (line 28) in the two internal functions.

```

module or program name_inherit
  Optional territorial variable, type specification, and calls
  contains

    subroutine Internal_1
      territorial specifications and calls
      contains

        subroutine Internal_2
          local computations
        end subroutine Internal_2

        subroutine Internal_3
          local computations
        end subroutine Internal_3

      end subroutine Internal_1
    end name_inherit

```

**Figure 4.9:** F90 Internal Subprogram Structure.

```

[ 1] module tic_toc
[ 2] ! Define global constants for timing increments
[ 3]   implicit none
[ 4]   integer :: start ! current value of system clock
[ 5]   integer :: rate  ! system clock counts/sec
[ 6]   integer :: finish ! ending value of system clock
[ 7]   real    :: sec   ! increment in sec, (finish-start)/rate
[ 8]   ! Usage: use tic_toc ! global constant access
[ 9]   ! call tic ! start clock
[10]   ! . . . ! use some cpu time
[11]   ! cputime = toc () ! for increment
[12] contains ! access to start, rate, finish, sec
[13]   subroutine tic
[14]   ! -----
[15]   ! Model the matlab tic function, for use with toc
[16]   ! -----
[17]     implicit none
[18]     call system_clock ( start, rate ) ! Get start value and rate
[19]   end subroutine tic
[20]
[21]   function toc ( ) result(sec)
[22]   ! -----
[23]   ! Model the matlab toc function, for use with tic
[24]   ! -----
[25]     implicit none
[26]     real :: sec
[27]     call system_clock ( finish ) ! Stop the execution timer
[28]     sec = 0.0
[29]     if ( finish >= start ) sec = float(finish - start) / float(rate)
[30]   end function toc
[31] end module tic_toc

```

**Figure 4.10:** A Module for Computing CPU Times

Action	C++	F90
Bitwise AND	&	iand
Bitwise exclusive OR	^	ieor
Bitwise exclusive OR		ior
Circular bit shift		ishftc
Clear bit		ibclr
Combination of bits		mvbits
Extract bit		ibits
Logical complement	~	not
Number of bits in integer	sizeof	bit_size
Set bit		ibset
Shift bit left	<<	ishft
Shift bit right	>>	ishft
Test on or off		btest
Transfer bits to integer		transfer

**Table 4.24:** Bit Function Intrinsics.

### 4.4.3 Bit Functions

We have discussed the fact that the digital computer is based on the use of individual bits. The subject of bit manipulation is one that we do not wish to pursue here. However, advanced applications do sometimes require these abilities, and the most common uses have been declared in the so-called *military standards* USDOD-MIL-STD-1753, and made part of the Fortran90 standard. Several of these features are also a part of C++. Table 4.24 gives a list of those functions.

### 4.4.4 Exception Controls

An exception handler is a block of code that is invoked to process specific error conditions. Standard exception control keywords in a language are usually associated with the allocation of resources, such as files or memory space, or input/output operations. For many applications we simply want to catch an unexpected result and output a message so that the programmer can correct the situation. In that case we may not care if the exception aborts the execution. However, if one is using a commercial execute only program then it is very disturbing to have a code abort. We would at least expect the code to respond to a fatal error by closing down the program in some gentle fashion that saves what was completed before the error and maybe even offer us a restart option. Here we provide only the minimum form of an exceptions module that can be used by other modules to pass warnings of fatal messages to the user. It includes an integer flag that can be utilized to rank the severity of possible messages. It is shown in Fig. 4.11. Below we will summarize the F90 optional error flags that should always be checked and are likely to lead to a call to the exception handler.

**Dynamic Memory:** The `ALLOCATE` and `DEALLOCATE` statements both use the optional flag `STAT =` to return an integer flag that can be tested to invoke an exception handler. The integer value is zero after a successful (de)allocation, and a positive value otherwise. If `STAT =` is absent, an unsuccessful result stops execution.

**File Open/Close:** The `OPEN`, `CLOSE`, and `ENDFILE` statements allow the use of the optional keyword `IOSTAT =` to return an integer flag which is zero if the statement executes successfully, and a positive value otherwise. They also allow the older standard exception keyword `ERR =` to be assigned a positive integer constant label number of the statement to which control is passed if an error occurs. An exception handler could be called by that statement.

**File Input/Output:** The `READ`, `WRITE`, `BACKSPACE`, and `REWIND` statements allow the `IOSTAT =` keyword to return a negative integer if an end-of-record (EOR) or end-of-file (EOF) is encountered, a zero if there is no error, and a positive integer if an error occurs (such as reading a character during an

```

[ 1] module exceptions
[ 2]   implicit none
[ 3]   integer, parameter :: INFO = 1, WARN = 2, FATAL = 3
[ 4]   integer              :: error_count = 0
[ 5]   integer              :: max_level   = 0
[ 6] contains
[ 7]
[ 8]   subroutine exception (program, message, flag)
[ 9]     character(len=*)    :: program
[10]     character(len=*)    :: message
[11]     integer, optional :: flag
[12]
[13]     error_count = error_count + 1
[14]
[15]     print *, 'Exception Status Thrown'
[16]     print *, ' Program :', program
[17]     print *, ' Message :', message
[18]     if ( present(flag) ) then
[19]       print *, ' Level   :', flag
[20]       if ( flag > max_level ) max_level = flag
[21]     end if ! flag given
[22]   end subroutine exception
[23]
[24]   subroutine exception_status ( )
[25]     print *
[26]     print *, "Exception Summary:"
[27]     print *, " Exception count = ", error_count
[28]     print *, " Highest level   = ", max_level
[29]   end subroutine exception_status
[30] end module exceptions

```

**Figure 4.11:** A Minimal Exception Handling Module

integer input). They also allow the `ERR =` error label branching described above for the file open/close operations.

In addition, the `READ` statement also retains the old standard keyword `END =` to identify a label number to which control transfers when an end-of-file (EOF) is detected.

**Status Inquiry:** Whether in `UNIT` mode or `FILE` mode, the `INQUIRE` statement for file operations allows the `IOSTAT =` and `ERR =` keywords like the `OPEN` statement. In addition, either mode supports two logical keywords: `EXISTS =` to determine if the `UNIT` (or `FILE`) exists, and `OPENED =` to determine if a (the) file is connected to this (an) unit.

**Optional Arguments:** The `PRESENT` function returns a logical value to indicate whether or not an optional argument was provided in the invocation of the procedure in which the function appears.

**Pointers and Targets:** The `ASSOCIATED` function returns a logical value to indicate whether a pointer is associated with a specific target, or with any target.

## 4.5 Interface Prototype

Compiler languages are more efficient than interpreted languages. If the compiler is going to correctly generate calls to functions, or subprograms, it needs to know certain things about the arguments and returned values. The number of arguments, their type, their rank, their order, etc. must be the same. This collection of information is called the “interface” to the function, or subprogram. In most of our example codes the functions and subprograms have been included in a single file. In practice they are usually stored in separate external files, and often written by others. Thus, the program that is going to use these external files must be given a “prototype” description of them. In other words, a segment of prototype, or interface, code is a definition that is used by the compiler to determine what parameters are required by the subprogram as it is called by your program. The interface prototype code for any subprogram can usually be created by simply copying the first few lines of the subprogram (and maybe the last one) and placing them in an interface directory.

To successfully compile a subprogram modern computer science methods sometimes require the programmer to specifically declare the interface to be used in invoking a subprogram, even if that subprogram is included in the same file. This information is called a “prototype” in C and C++, and an “interface” in F90. If the subprogram already exists, one can easily create the needed interface details by making



a copy of the program and deleting from the copy all information except that which describes the arguments and subprogram type. If the program does not exist, you write the interface first to define what will be expected of the subprogram regardless of who writes it. It is considered good programming style to include explicit interfaces, or prototype code, even if they are not required.

If in doubt about the need for an explicit interface see if the compiler gives an error because it is not present. In F90 the common reasons for needing an explicit interface are: 1) Passing an array that has only its rank declared. For example, `A(:, :), B(:, :)`. These are known as “assumed-shape” arrays; 2) Using a function to return a result that is: *a)* an array of unknown size, or *b)* a pointer, or *c)* a character string with a dynamically determined length. Advanced features like optional argument lists, user defined operators, generic subprogram names (to allow differing argument types) also require explicit operators.

In C++ before calling an external function, it must be declared with a prototype of its parameters. The general form for a function is

```
function_type function_name ( argument_type_list );
```

where the `argument_type_list` is the comma separated list of pairs of type and name for each argument of the function. These names are effectively treated as comments, and may be different from the names in the calling program, or even omitted. The use of a prototype was shown in Fig. 4.8 and is used again in Fig. 4.12 which also illustrates passing arguments by reference or by value.

An interface block for external subprograms was not required by F77 (thereby leading to hard to find errors), but is strongly recommended if F90 and is explicitly required in several situations. The general form for a F90 interface is

```
interface interface_name
  function_interface_body
  subroutine_interface_body
  module_procedure_interface_body
end interface interface_name
```

where a typical `function_interface_body` would be

```
function_type function_name (argument_name_list) result ( name )
  implicit none
  argument_type, intent_class :: name_list
end function function_name
```

where the `argument_name_list` is the comma separated list of names. Of course, the `function_type` refers to the result argument name. These names may be different from the names in the calling program. A typical `subroutine_interface_body` would be

```
subroutine subroutine_name (argument_name_list)
  implicit none
  argument_type, intent_class :: name_list
end subroutine subroutine_name
```

where the `argument_name_list` is the comma separated list of names. The topic of a module procedure is covered elsewhere. The use of an interface block was shown in Fig. 4.8 and used in two new codes, shown in Fig. 4.12, and the corresponding C++ code in the appendix, which also illustrate passing arguments by reference (line 23) and by value (line 19) in both F90 and C++. The important, and often confusing, topic of passing by reference or value was discussed in Sec. 4.2 and is related to other topics to be considered later, such as the use of “pointers” in C++ and F90, and the “intent” attribute of F90 arguments. Passing by reference is default in F90 while passing by value is default in C++.

## 4.6 Characters and Strings

All of our example languages offer convenient ways to manipulate and compare strings of characters. The characters are defined by one of the international standards such as ASCII, which is usually used on UNIX, or the EBCDIC set. These contain both printable and non-printable (control) characters. On a UNIX system, the full set can be seen with the command `man ascii`. In the 256-character ASCII set, the upper case letters begin at character number 65, ‘A’, and the corresponding lower case values are

```

[ 1] program main
[ 2] implicit none
[ 3] ! declare the interface prototypes
[ 4] interface
[ 5]   subroutine Change (Refer)
[ 6]     integer :: Refer; end subroutine Change
[ 7]   subroutine No_Change (Value)
[ 8]     integer :: Value; end subroutine No_Change
[ 9] end interface
[10]
[11] ! illustrate passing by reference and by value in F90
[12]
[13] integer :: Input_Val, Dummy_Val
[14]
[15] print *, "Enter an integer: "
[16] read *, Input_Val; print *, "Input value was ", Input_Val
[17]
[18] ! pass by value
[19] call No_Change ( (Input_Val) ) ! Use but do not change
[20] print *, "After No_Change it is ", Input_Val
[21]
[22] ! pass by reference
[23] call Change ( Input_Val ) ! Use and change
[24] print *, "After Change it is ", Input_Val
[25] end program
[26]
[27] subroutine Change (Refer)
[28] ! changes Refer in calling code IF passed by reference
[29] integer :: Refer
[30] Refer = 100;
[31] print *, "Inside Change it is set to ", Refer
[32] end subroutine Change
[33]
[34] subroutine No_Change (Value)
[35] ! does not change Value in calling code IF passed by value
[36] integer :: Value
[37] Value = 100;
[38] print *, "Inside No_Change it is set to ", Value
[39] end subroutine No_Change
[40]
[41] ! Running gives:
[42] ! Enter an integer: 12
[43] ! Input value was 12
[44] ! Inside No_Change it is set to 100
[45] ! After No_Change it is 12
[46] ! Inside Change it is set to 100
[47] ! After Change it is 100

```

**Figure 4.12:** Passing Arguments by Reference and by Value in F90

32 positions higher (character 97 is 'a'). These printable characters begin at character 32, as shown in Table 4.25 for the ASCII standard. The first 33 characters are “non-printing” special control characters. For example, NUL = null, EOT = end of transmission, BEL = bell, BS = backspace, and HT = horizontal tab. To enter a control character, one must simultaneously hold down the CONTROL key and hit the letter that is 64 positions higher in the list. That is, an end of transmission EOT is typed as CONTROL-D. The code SP denotes the space character, and we will use the underscore “\_” to represent a blank in strings.

We can employ the standard relational operators (e.g., less than) to compare strings and would find that 'bad' < 'dog' < 'same' == 'same \_ \_', that 'word' > 'WORD', and that 'four' < 'one' < 'two' while '1' < '2' < '4'. Note that the above equality occurred because trailing blanks are not considered in relational operations, *but* leading blanks are considered: 'same' ≠ '\_ \_ same'. The F90 function `adjustL` removes leading blanks and appends them to the right end. Thus, it adjusts the string to the left, so that 'same' == `adjustL('_ _ same')`. This and other F90 intrinsic character functions are summarized in Table 4.26.

All blanks are considered when determining the length of a character string. In F90 the intrinsic function `LEN` provides these data so that `LEN('same')` = 4, `LEN('_ _ same')` = 6, and `LEN('same _ _')` = 7. There is another intrinsic function, `LEN_TRIM`, that provides the string length ignoring trailing blanks. By way of comparison: `LEN_TRIM('same')` = 4, `LEN_TRIM('_ _ same')` = 6, and `LEN_TRIM('same _ _')` = 4. Each character in a string or any internal substrings may be referenced by the colon operator. Given a character variable we can define a substring, say `sub` as

```

sub = variable(K:L) for 0 < K,L <= LEN(variable)
    = null           for K > L

```

0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	NL	11	VT	12	NP	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	"	35	#	36	\$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124		125	}	126	~	127	DEL

**Table 4.25:** The ASCII Character Set

ACHAR (I)	Character number I in ASCII collating set
ADJUSTL (STRING)	Adjust left
ADJUSTR (STRING)	Adjust right
CHAR (I) *	Character I in processor collating set
IACHAR (C)	Position of C in ASCII collating set
ICHAR (C)	Position of C in processor collating set
INDEX (STRING, SUBSTRING) <sup>a</sup>	Starting position of a substring
LEN (STRING)	Length of a character entity
LEN_TRIM (STRING)	Length without trailing blanks
LGE (STRING_A, STRING_B)	Lexically greater than or equal
LGT (STRING_A, STRING_B)	Lexically greater than
LLE (STRING_A, STRING_B)	Lexically less than or equal
LLT (STRING_A, STRING_B)	Lexically less than
REPEAT (STRING, NCOPIES)	Repeated concatenation
SCAN (STRING, SET) <sup>a</sup>	Scan a string for a character in a set
TRIM (STRING)	Remove trailing blank characters
VERIFY (STRING, SET) <sup>a</sup>	Verify the set of characters in a string
STRING_A//STRING_B	Concatenate two strings

<sup>a</sup>Optional arguments not shown.

**Table 4.26:** F90 Character Functions

```
= error          for K or L > LEN(variable).
```

For example, given the string 'howl', then we can define `bird = string(2:4) = 'owl'`, and `prep = string(1:3) = 'how'`.

The F90 and F77 operator used to concatenate strings into larger strings is `//`. Continuing the last example, we see that the concatenation `string(1:3)//'_'//string(2:4)//'?'` is `'how_owl?'`, while the concatenation `'same_'//'word'` becomes `'same_word'` and `'bad'//'_'//'dog'` becomes `'bad_dog'`. Programs illustrating the reading and concatenating two strings are given in Fig. 4.13, and in the companion C++ code in the appendix.

Sometimes one needs to type in a non-printing character, such as a tab or a newline. To allow this, special transmissions have been allowed for, as summarized in Table 4.27.

Remember the ASCII character features: the uppercase letters correspond to numbers 65 through 90 in the list, while the lowercase letters are numbers 97 through 122, so that if we wanted to convert "G" to

```

[ 1] program main
[ 2] ! Compare two strings
[ 3] ! Concatenate two character strings together
[ 4] ! Get the combined length
[ 5] implicit none
[ 6] character(len=20) :: String1, String2
[ 7] character(len=40) :: String3
[ 8] integer           :: length
[ 9]
[10]     print *, 'Enter first string (20 char max):'
[11]     read  '(a)', String1    ! formatted
[12]
[13]     print *, 'Enter second string (20 char max):'
[14]     read  '(a)', String2    ! formatted
[15]
[16]     ! compare
[17]     if ( String1 == String2 ) then
[18]         print *, "They are the same."
[19]     else
[20]         print *, "They are different."
[21]     end if
[22]
[23]     ! concatenate
[24]     String3 = trim (String1) // trim (String2)
[25]
[26]     print *, 'The combined string is:', String3
[27]     length = len_trim (String3)
[28]     print *, 'The combined length is:', length
[29]
[30] end program main
[31] ! Running with "red" and "bird" produces:
[32] ! Enter first string (20 char max): red
[33] ! Enter second string (20 char max): bird
[34] ! They are different.
[35] ! The combined string is: redbird
[36] ! The combined length is: 7
[37] ! Also "the red" and "bird" works

```

**Figure 4.13:** Using Two Strings in F90

<i>Action</i>	<i>ASCII Character</i>	<i>F90 Input<sup>a</sup></i>	<i>C++ Input</i>
Alert (Bell)	7	Ctrl-G	\a
Backspace	8	Ctrl-H	\b
Carriage Return	13	Ctrl-M	\r
End of Transmission	4	Ctrl-D	Ctrl-D
Form Feed	12	Ctrl-L	\f
Horizontal Tab	9	Ctrl-I	\t
New Line	10	Ctrl-J	\n
Vertical Tab	11	Ctrl-K	\v

<sup>a</sup>“Ctrl-” denotes control action. That is, simultaneous pressing of the CONTROL key *and* the letter following.

**Table 4.27:** How to type non-printing characters.

“g” we could use commands such as:

```

character (len = 1) :: lower_g, UPPER_G
lower_g = achar(iachar('G') + 32)

```

or visa versa:

```

UPPER_G = achar(iachar('g') - 32)

```

since they differ by 32 locations. Likewise, since the zero character “0” occurs in position 48 of the ASCII set we could convert a single digit to the same numerical value with:

```

integer :: number_5
number_5 = iachar('5') - 48

```

and so forth for all ten digits. To convert a string of digits, such as '5623', to the corresponding number 5623, we could use a looping operation.

```

[ 1] program main
[ 2] ! Convert a character string to an integer in F90
[ 3] implicit none
[ 4] character(len=5) :: Age_Char
[ 5] integer          :: age
[ 6]
[ 7]     print *, "Enter your age: "
[ 8]     read *, Age_Char                ! a character string
[ 9]
[10] ! convert using an internal file read
[11] read (Age_Char, fmt = '(i5)') age ! convert to integer
[12]
[13]     print *, "Your integer age is      ", age
[14]     print '(" Your binary age is      ", b8)', age
[15]     print '(" Your hexadecimal age is ", z8)', age
[16]     print '(" Your octal age is       ", o8)', age
[17]
[18] end program main
[19] !
[20] ! Running gives:
[21] ! Enter your age: 45
[22] ! Your integer age is      45
[23] ! Your binary age is      101101
[24] ! Your hexadecimal age is 2D
[25] ! Your octal age is       55

```

**Figure 4.14:** Converting a String to an Integer with F90

```

character (len = 132) :: digits
integer              :: d_to_n, power, number
! Now build the number from its digits
if (digits == ' ') then
    print *, 'warning, no number found'
    number = 0
else
    number = 0
    k = len_trim(digits)
    do m = k, 1, -1 ! right to left
        d_to_n = iachar(digits(m:m)) - 48
        power = 10**(k-m)
        number = number + d_to_n*power
    end do ! over digits
    print *, 'number = ', number

```

However, since loops can be inefficient, it is better to learn that, in F90, an “internal file” can be (and should be) employed to convert one data type to another. Here we could simply code:

```

! internal file called convert
write(convert, '(A)') digit
read(convert, '(I4)') number

```

to convert a character to an integer (or real) number. Converting strings to integers is shown in the codes given in Fig. 4.14 (line 11) and the corresponding C++ appendix routine. Similar procedures would be used to convert strings to reals. The C++ version (see appendix) uses the intrinsic function “atoi” while the F90 version uses an internal file for the conversion.

One often finds it useful to change the case of a string of characters. Some languages provide intrinsic functions for that purpose. In C++ and MATLAB the function to convert a string to all lower case letters are called `tolower` and `lower`, respectively. Here we define a similar F90 function called `to_lower` which is shown in Fig. 4.15 along with a testing program in Fig. 4.16. Note that the testing program uses an interface to `tolower` (lines 4-13) assuming that routine was compiled and stored external to the testing program. The `tolower` function employs the intrinsic function `index` (line 16) to see if the *k*-th character of the input string is an upper case letter. The intrinsic function `len` is also used (line 8) to force the `new_string` to be the same length as the original string.

## 4.7 User Defined Data Types

Variables, as in mathematics, represent some quantity; unlike mathematics, many languages force the programmer to define what *type* the variable is. Generic kinds of type are integer, floating point (single, double, and quadruple precision), and complex-valued floating point. Table 4.2 (page 53) presents the data types inherent in the various languages. Most beginning programmers find the requirement most

```

[ 1] function to_lower (string) result (new_string) ! like C
[ 2] ! -----
[ 3] !           Convert a string or character to lower case
[ 4] !           (valid for ASCII or EBCDIC processors)
[ 5] ! -----
[ 6] implicit none
[ 7] character (len = *) , intent(in) :: string      ! unknown length
[ 8] character (len = len(string)) :: new_string    ! same length
[ 9] character (len = 26), parameter ::             &
[10]         UPPER = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ', &
[11]         lower = 'abcdefghijklmnopqrstuvwxyz'
[12] integer :: k      ! loop counter
[13] integer :: loc    ! position in alphabet
[14] new_string = string      ! copy everything
[15] do k = 1, len(string)    ! to change letters
[16]     loc = index ( UPPER, string(k:k))           ! first upper
[17]     if (loc /= 0) new_string(k:k) = lower(loc:loc) ! convert it
[18] end do ! over string characters
[19] end function to_lower

```

**Figure 4.15:** Converting a String to Lower Case with F90

```

[ 1] program up_down ! test character case inversion functions
[ 2] implicit none
[ 3] character (len = 24) :: test='ABCDefgh1234abcdZYXWzyxw'
[ 4]
[ 5] interface
[ 6]     function to_lower (string) result (new_string)
[ 7]         character (len = *) , intent(in) :: string
[ 8]         character (len = len(string)) :: new_string
[ 9]     end function to_lower
[10]     function to_upper (string) result (new_string)
[11]         character (len = *) , intent(in) :: string
[12]         character (len = len(string)) :: new_string
[13]     end function to_upper
[14] end interface
[15]
[16] print *, test
[17] print *, to_lower (test)
[18] print *, to_upper (test)
[19] end program ! running gives
[20] ! ABCDefgh1234abcdZYXWzyxw
[21] ! abcdefgh1234abcdzyxwzyxw
[22] ! ABCDEFGH1234ABCDZYXWZYXW

```

**Figure 4.16:** Testing String Conversions with F90

languages impose of defining explicitly each variable's type to be tedious, unnecessary, and a source of bugs. It's tedious because the programmer must think not only about what the variable represents, but also how the computations calculate its value, unnecessary because mathematics doesn't work that way (the variable  $x$  represents a quantity regardless whether it turns out to be an integer or a complex value), and bug-creating because computations involving different types and assigned to a typed variable can yield nonmathematical results (for example, dividing the integers 1 with 3 and assigning the results to an integer yields a zero value).

MATLAB is one language in which variables are not explicitly typed. (Beginning programmers cheer!) Internally, MATLAB represents numbers in double precision floating point. If a variable's value corresponds to an integer, MATLAB will gleefully print it that way, effectively hiding its floating point representation. A surprise occurs when a calculation accidentally becomes complex: MATLAB will (silently) change what the variable represents from being real to being complex. For example, MATLAB will, without complaint, calculate  $x = \log(-1)$  and assign the value  $3.14159i$  to  $x$ . In many applications, the expression that yielded the value of  $-1$  because of an error, and MATLAB will let the error propagate. (Beginning programmers sigh!) Most, if not all typed languages will immediately announce the evaluation of the logarithm of a negative number, and halt execution. By explicitly defining the kinds of values a variable will assume helps programming clarity and run-time debugging to some degree.

C++ has four intrinsic (i.e., built-in) types of data—integer, single and double precision reals, and character—and F90 has the similar set: integer, real, complex, logical, and character. F90 also allows the user to create a specific precision level for integer and real data. C++ has specified byte sizes for three character, six integer, one single precision real, and two double precision real data types for a total of twelve intrinsic data types.

C, C++	Variable.component.sub_component
F90	Variable%component%sub_component

**Table 4.28:** Referencing Defined Data Type Structure Components.

C, C++	<pre>struct data_tag {     intrinsic_type_1 component_names;     intrinsic_type_2 component_names; } ;</pre>
F90	<pre>type data_tag     intrinsic_type_1 :: component_names;     intrinsic_type_2 :: component_names; end type data_tag</pre>

**Table 4.29:** Defining New Types of Data Structure

C, C++	<pre>struct data_tag {     intrinsic_type_1 component_names;     struct tag_2 component_names; } ;</pre>
F90	<pre>type data_tag     intrinsic_type :: component_names;     type (tag_2) :: component_names; end type data_tag</pre>

**Table 4.30:** Nested Data Structure Definitions.

In addition to intrinsic types, C, C++ and F90 allow the formation of new types of data—*structures*—that are collections of values of not necessarily the same type. These procedures are named `struct` or `type` in C and F90, respectively.

To go along with this freedom, F90 allows you to define new operations to act on the derived types. While C++ retains the *struct* keyword, it is viewed as a *class* with only public data members and no functions. In other words, in C++ *class* is a generalization of *struct* and, thus, *class* is the preferred keyword to use. As an example of a task made easier by derived data, consider creating parts of a data structure to be used in an address book. We will need a variable that can have components and sub-components. They are referenced by a special syntax and defined as illustrated in Tables 4.28 and 4.29. This procedure for defining a new type of data structure can be “nested” by referring to other derived type entities defined earlier in the program. These concepts are shown in Table 4.30. One should declare the data type of all variables used in a program module. This is also true for user defined data structures. Table 4.31 outlines the forms of these statements, how structures are initialized, and how component values are assigned.

There are times when either the derived type variable or its components, or both, could be subscripted objects (i.e., arrays). Then care must be taken in the interpretation of which variable or component is being addressed. Table 4.32 illustrates the typical combinations with the F90 syntax.

As a concrete example, consider a `phone_type` and `address_type` definition.

C, C++	<pre> struct data_tag variable_list; /* Definition */ struct data_tag variable = {component_values}; /* Initialization */ variable.component.sub_component = value; /* Assignment */ </pre>
F90	<pre> type (data_tag) :: variable_list ! Definition variable = data_tag (component_values) ! Initialization variable%component%sub_component = value ! Assignment </pre>

**Table 4.31:** Declaring, initializing, and assigning components of user-defined datatypes.

<pre> INTEGER, PARAMETER :: j_max = 6 TYPE meaning_demo   INTEGER, PARAMETER :: k_max = 9, word = 15   CHARACTER (LEN = word) :: name(k_max) END TYPE meaning_demo TYPE (meaning_demo) derived(j_max) </pre>	
Construct	Interpretation
derived	All components of all derived's elements
derived(j)	All components of j <sup>th</sup> element of derived
derived(j)%name	All k_max components of name within j <sup>th</sup> element of derived
derived%name(k)	Component k of the name array for all elements of derived
derived(j)%name(k)	Component k of the name array of j <sup>th</sup> element of derived

**Table 4.32:** F90 Derived Type Component Interpretation.

F90	C++
<pre> type phone_type   integer :: area_code, number, extension end type phone_type type address_type   integer :: number   character (len = 35) :: street, city   character (len = 2) :: state   integer :: zip_code end type address_type </pre>	<pre> struct phone_type {   int area_code, number, extension; }; struct address_type {   int number;   char street[35], city[35];   char state[2];   int zip_code; }; </pre>

These could be used to define part of a person\_type

F90	C++
<pre> type person_type   character (len = 50) :: name   type (phone_type) :: phone   type (address_type) :: address   integer :: born_year end type person_type </pre>	<pre> struct person_type {   char name[50];   struct phone_type phone;   struct address_type address;   int born_year; }; </pre>

We define two people with

F90	C++
<pre> type (person_type) :: sammy, barney </pre>	<pre> struct person_type sammy, barney; </pre>

or build an address book array filled with the above data structures by defining



## F90

```
integer, parameter :: number = 99
type (person_type), dimension (number) :: address_book
```

## C++

```
#define NUMBER 99
struct person_type address_book[NUMBER];
```

and then initialize, or “construct” sammy’s phone and zip code as

## F90

```
sammy%phone = phone_type (713, 5278100, 0)
sammy%zip_code = 770051892
```

## C++

```
sammy.phone = {713, 5278100, 0};
sammy.zip_code = 770051892;
```

and print them with

## F90

```
print *, sammy%phone
print *, sammy%address%zip_code
```

## C++

```
printf("(%d)%d, extension %d",
       sammy.area_code,
       sammy.number,
       sammy.extension);
printf("%d", sammy.zip_code);
```

and then define specific members for barney with the “constructor”

## F90

```
barney = person_type("Barn Owl", &
                    phone_type(0,0,0), &
                    sammy%address, 1892, "Sammy's cousin")
```

## C++

```
barney = {"Barn Owl", {0,0,0},
          sammy.address, 1892,
          "Sammy's cousin"};
```

Note the difference in the defined type constructors. Two are actually used here because the second component must be defined as a `phone_type`. C++ just uses brackets to enclose the supplied components of each user defined type. F90 has an intrinsic function that is created automatically by the type definition and it accepts all of the components required by the type. That is why the function name “`phone_type`” appears in the intrinsic constructor routine “`person_type`”. Finally, put them in the book.

## F90

```
address_book(1) = sammy
address_book(2) = barney
```

## C++

```
address_book[1] = sammy;
address_book[2] = barney;
```

Fig. 4.17 presents a sample code for utilizing user defined structure types using F90 (there is a C++ version in the appendix). First a “person” structure is created (lines 4-7) by using only the intrinsic types of integers and characters. It then is used in turn within an additional data structure (line 10). The components of the structures are read (lines 18, 21, 24) and output (lines 26,27). For more general data, suggested in the comments, formatted input/output controls would be necessary.

### 4.7.1 Overloading Operators

As a complete short example of utilizing many of the new programming features that come with user defined data structures we will consider the use of a familiar old mathematics system, fractions. Recall that a fraction is the ratio of two integers. We will therefore define a new data type called *Fraction*. It

```

[ 1] program main ()
[ 2] ! Define structures and components, via F90
[ 3] implicit none
[ 4] type Person ! define a person structure type
[ 5]   character (len=20) :: Name
[ 6]   integer           :: Age
[ 7] end type Person
[ 8]
[ 9] type WhoWhere ! use person type in a new structure
[10]   type (Person) :: Guest
[11]   character (len=40) :: Address
[12] end type WhoWhere
[13]
[14] ! Fill a record of the WhoWhere type components
[15] type (WhoWhere) Record;
[16]
[17]   print *, "Enter your name: "
[18]   read  *, Record % Guest % Name
[19]
[20]   print *, "Enter your city: "
[21]   read  *, Record % Address
[22]
[23]   print *, "enter your age: "
[24]   read  *, Record % Guest % Age
[25]
[26]   print *, "Hello ", Record % Guest % Age, " year old ", &
[27]     Record % Guest % Name, " in ", Record % Address
[28]
[29] end program main
[30]
[31] ! Running with input: Sammy, Houston, 104 gives
[32] ! Hello 104 year old Sammy in Houston
[33] !
[34] ! But try: Sammy Owl, Houston, 104 for a bug

```

**Figure 4.17:** Using Multiple Structures in F90

will simply consist of two integer types, named *num* and *denom*, respectively. New data types can be defined in any program unit. For maximum usefulness we will place the definition in a module named *Fractions*. To use this new data type we will want to have subprograms to define a fraction, list its components, and multiply two fractions together, and to equate one fraction to another. In addition to the intrinsic constructor function *fraction* we will create a manual constructor function called *assign* and it will have two arguments, the numerator value, and denominator value, and will use them to return a fraction type. The listing subroutine, called *list\_Fraction*, simply needs the name of the fraction to be printed. The function, *mult\_Fraction*, accepts two fraction names, and returns the third fraction as their product. Finally, we provide a function that equates the components of one fraction to those in a new fraction.

This data structure is presented in Fig. 4.18. There we note that the module starts with the definition of the new data type (lines 2-4), and is followed with the “contains” statement (line 12). The subprograms that provide the functionality of the fraction data type follow the “contains” statement and are thus coupled to the definition of the new type. When we have completed defining the functionality to go with the new data type we end the module.

In this example the program to invoke the fraction type follows in Fig. 4.19. To access the module, which defines the new data type and its supporting functions, we simply employ a “use” statement at the beginning of the program (line 2). The program declares three *Fraction* type variables (line 3): *x*, *y*, and *z*. The variable *x* is defined to be 22/7 with the intrinsic type constructor (line 5), while *y* is assigned a value of 1/3 by using the function *assign* (line 7). Both values are listed for confirmation. Then we form the new fraction,  $z = 22/21$ , by invoking the *mult\_Fraction* function (line 9),

```
z = mult_Fraction (x, y)
```

which returns *z* as its result. A natural tendency at this point would be to simply write this as  $z = x * y$ . However, before we could do that we would have to tell the operators, “\*” and “=”, how to act when provided with this new data type. This is known as *overloading* an intrinsic operator. We had the foresight to do this when we set up the module by declaring which of the “module procedure”s were equivalent to each operator symbol. Thus from the “interface operator (\*)” statement block the system now knows that the left and right operands of the “\*” symbol correspond to the first and second arguments in the

```

[ 1] module Fractions      ! F90 "Fraction" data structure and functionality
[ 2]   implicit none
[ 3]   type Fraction        ! define a data structure
[ 4]     integer :: num, den ! with two "components"
[ 5]   end type Fraction
[ 6]
[ 7]   interface operator (*) ! extend meaning to fraction
[ 8]     module procedure mult_Fraction ; end interface
[ 9]
[10]   interface assignment (=) ! extend meaning to fraction
[11]     module procedure equal_Fraction ; end interface
[12]
[13] contains ! functionality
[14]   subroutine assign (name, numerator, denominator)
[15]     type (Fraction), intent(inout) :: name
[16]     integer, intent(in)              :: numerator, denominator
[17]
[18]     name % num = numerator      ! % denotes which "component"
[19]     if ( denominator == 0 ) then
[20]       print *, "0 denominator not allowed, set to 1"
[21]       name % den = 1
[22]     else; name % den = denominator
[23]     end if ; end subroutine assign
[24]
[25]   subroutine list(name)
[26]     type (Fraction), intent(in) :: name
[27]
[28]     print *, name % num, "/", name % den ; end subroutine list
[29]
[30]   function mult_Fraction (a, b) result (c)
[31]     type (Fraction), intent(in) :: a, b
[32]     type (Fraction)              :: c
[33]
[34]     c%num = a%num * b%num ! standard = and * here
[35]     c%den = a%den * b%den ; end function mult_Fraction
[36]
[37]   subroutine equal_Fraction (new, name)
[38]     type (Fraction), intent(out) :: new
[39]     type (Fraction), intent(in)  :: name
[40]
[41]     new % num = name % num ! standard = here
[42]     new % den = name % den ; end subroutine equal_Fraction
[43] end module Fractions

```

**Figure 4.18:** Overloading operations for new data types

function `mult_Fraction`. Likewise, the left and right operands of “=” are coupled to the first and second arguments, respectively, of subroutine `equal_Fraction`. The testing main and verification results are in Fig. 4.19 Before moving on note that the system does not yet know how to multiply a integer times a fraction, or visa versa. To do that one would have to add more functionality, such as a function, say `int_mult_frac`, and add it to the "module procedure" list associated with the “\*” operator.

When considering which operators to overload for a newly defined data type one should consider those that are used in sorting operations, such as the greater-than, `>`, and less-than, `<`, operators. They are often useful because of the need to sort various types of data. If those symbols have been correctly overloaded then a generic sorting routine might be used, or require few changes.

## 4.7.2 User Defined Operators

In addition to the many intrinsic operators and functions we have seen so far, the F90 user can also define new operators or extend existing ones. User defined operators can employ intrinsic data types and/or user defined data types. The user defined operators, or extensions, can be unary or binary (i.e., have one or two arguments). The operator symbol must be included between two periods, such as `.op.`. Specific examples will be given in the next chapter.

## 4.8 Pointers and Targets

The beginning of every data item must be stored in computer memory at a specific address. The address of that data item is called a *pointer* to the data item, and a variable that can hold such an address is called a *pointer variable*. Often it is convenient to have a pointer to a variable, an array, or a sub-array. F90, C++ and MATLAB provide this sophisticated feature. The major benefits of the use of pointers is that

```

[ 1] program main
[ 2]   use Fractions
[ 3]   implicit none
[ 4]   type (Fraction) :: x, y, z
[ 5]
[ 6]   x = Fraction (22,7)      ! default constructor
[ 7]   write (*,('default  x = '), advance='no') ; call list(x)
[ 8]   call assign(y,1,3)      ! manual constructor
[ 9]   write (*,('assigned y = '), advance='no') ; call list(y)
[10]   z = mult_Fraction (x,y)  ! function use
[11]   write (*,('x mult y = '), advance='no') ; call list(z);
[12]   print *, "Trying overloaded * and = for fractions:"
[13]   write (*,('y * x gives '), advance='no') ; call list(y*x) ! multi
[14]   z = x*y                 ! new operator uses
[15]   write (*,('z = x*y gives '), advance='no') ; call list(z) ! add
[16] end program main          ! Running gives:
[17] ! default  x = 22/7      ! assigned y = 1/3    ! x mult y = 22/21
[18] ! Trying overloaded * and = for fractions:
[19] ! y * x gives 22/21      ! z = x*y gives 22/21

```

**Figure 4.19:** Testing overloading for new data types

	C++	F90
Declaration	type_tag *pointer_name;	type (type_tag), pointer :: pointer_name
Target	&target_name	type (type_tag), target :: target_name
Examples	char *cp, c; int *ip, i; float *fp, f; cp = & c; ip = & i; fp = & f;	character, pointer :: cp integer, pointer :: ip real, pointer :: fp cp => c ip => i fp => f

**Table 4.33:** Definition of pointers and accessing their targets.

they allow dynamic data structures, such as “linked lists” and “tree structures,” and they allow recursive algorithms. Note that rather than containing data themselves, pointer variables simply exist to point to where some data are stored. Unlike C and MATLAB the F90 pointers are more like the “reference variables” of the C++ language in that they are mainly an alias or synonym for another variable, or part of another variable. They do not allow one to easily get the literal address in memory as does C. This is why programmers that write computer operating systems usually prefer C over F90. But F90 pointers allow easy access to array partitions for computational efficiency, which C++ does not. Pointers are often used to pass arguments by reference.

The item to which a pointer points is known as a *target* variable. Thus, every pointer has a logical status associated with it which indicates whether or not it is currently pointing to a target. The initial value of the association is `.false.`, or undefined.

#### 4.8.1 Pointer Type Declaration

For every type of data object that can be declared in the language, including derived types, a corresponding type of pointer and target can be declared (Table 4.33).

While the use of pointers gives programmers more options for constructing algorithms, they also have a potential severely detrimental effect on the program execution efficiency. To ensure that compilers can produce code that execute efficiently, F90 restricts the variables, to which a pointer can point, to those specifically declared to have the attribute `target`. This, in part, makes the use of pointers in F90 and C++ somewhat different. Another major difference is that C++ allows arithmetic to be performed on the pointer address, but F90 does not.

So far, we have seen that F90 requires specific declarations of a pointer and an potential target. However, C++ employs two unary operators, `&` and `*`, to deal with pointers and targets, respectively. Thus, in C++ the operator `&variable_name` means “the address of” `variable_name`, and the C++ operator `*pointer_name` means “the value at the address of” `pointer_name`.

C, C++	pointer_name = NULL
F90	nullify (list_of_pointer_names)
F95	pointer_name = NULL()

**Table 4.34:** Nullifying a pointer to break target association.

```

[ 1] program pt_expression
[ 2] !
[ 3] !      F90 example of using pointers in expressions
[ 4] implicit none
[ 5] integer, POINTER :: p, q, r
[ 6] integer, TARGET  :: i = 1, j = 2, k = 3
[ 7]
[ 8] q => j           ! q points to integer j
[ 9] p => i           ! p points to integer i
[10] !
[11] !      An expression that "looks like" pointer arithmetic
[12] !      automatically substitutes the target value:
[13] !
[14] q = p + 2         ! means: j = i + 2 = 1 + 2 = 3
[15] print *, i, j, k   ! print target values
[16] p => k             ! p now points to k
[17] print *, (q-p)     ! means print j - k = 3 - 3 = 0
[18] !
[19] !      Check associations of pointers
[20] print *, associated(r) ! false
[21] r => k             ! now r points to k, also
[22] print *, associated(p,i) ! false
[23] print *, associated(p,k) ! true
[24] print *, associated(r,k) ! true
[25] end program pt_expression

```

**Figure 4.20:** Using F90 Pointers in Expressions.

## 4.8.2 Pointer Assignment

F90 requires that a pointer be associated with a target by a single pointer assignment statement. C allows, but does not require, a similar statement. (See Table 4.33). After such a statement, the pointer has a new association status and one could employ the F90 intrinsic inquiry function `associated(pointer_name, target_name)` to return `.true.` as the logical return value. If one wishes to break or nullify a pointer's association with a target, but not assign it another target, one can nullify the pointer as shown in Table 4.34.

## 4.8.3 Using Pointers in Expressions

The most important rule about using pointers in F90 expressions is that, where ever a pointer occurs, it is treated as its associated target. That is, the target is automatically substituted for the pointer when the pointer occurs in an expression. For example, consider the actions in Fig. 4.20 (where the results are stated as comments).

## 4.8.4 Pointers and Linked Lists

Pointers are the simplest available mechanism for dynamic memory management of arrays such as stacks, queues, trees, and linked lists. These are extraordinarily flexible data structures because their size can grow or shrink during the execution of a program. For linked lists the basic technique is to create a derived type that consists of one or more data elements and at least one pointer. Memory is allocated to contain the data and a pointer is set to reference the next occurrence of data. If one pointer is present, the list is a singly-linked list and can only be traversed in one direction: head to tail, or vice versa. If two pointers are present: the list is a doubly-linked list and can be traversed in either direction. Linked lists allow the data of interest to be scattered all over memory and uses pointers to weave through memory, gathering data as required. Detailed examples of the use of linked lists are covered in Chapter 8.

As a conceptual example of when one might need to use linked-lists think of applications where one never knows in advance how many data entries will be needed. For example, when a surveyor determines the exact perimeter of a building or plot of land, critical measurements are taken at each

angle. If the perimeter has  $N$  sides, the surveyor measures the length of each side and the interior angle each side forms with the next. Often the perimeter has visual obstructions and offsets around them must be made, recorded, and corrected for later. Regardless of how careful the surveyor is, errors are invariably introduced during the measurement process. However, the error in angle measurements can be bounded.

The program for implementing the recording and correcting of the angles in a survey could be written using a singly linked list. A linked list is chosen because the programmer has no idea how many sides the perimeter has, and linked lists can grow arbitrarily. Because of the linked list's ability to absorb a short or long data stream, the user does not have to be asked to count the number of legs in the traverse. The program begins by declaring a derived type that contains one angle measurement and a pointer to the next measurement. A count is kept of the number of legs in this loop and the forward pointer for the last angle read is cleared (set to null) to signal the end of list. After all the data are read, the entire list of angles is reviewed to get the total of the measurements. This starts by revisiting the head of the list and adding together all the angle measurements until a null pointer is encountered, signaling the end of list. Then the error can be computed and distributed equally among the legs of the traverse.

## 4.9 Accessing External Source Files and Functions

At times one finds it necessary, or efficient to utilize other software from libraries, other users, or different paths in your directories. Of course, you could always use the brute force approach and use a text editor to copy the desired source code into your program. However, this is unwise not only because it wastes storage, but more importantly gives multiple copies of a module that must all be found and changed if future revisions are needed or desired. Better methods of accessing such codes can be defined either inside your program, or external to it in the “linking” phase after compiling has been completed.

High level languages like C, C++, and F90 allow one or more approaches for accessing such software from within your code. One feature common to all these languages is the availability of an “include” statement which gives the system path to the desired code file. At compile time, and only then, a temporary copy of the indicated code from that file is literally copied and inserted into your program at the location of the corresponding “include” statement.

It is common practice, but not required, to denote such code fragments with name extensions of “.h” and “.inc”, in C++ and F90, respectively. For example, to use a program called “class\_Person” one could insert the following statement in your program:

```
C, C++: include <class_Person.h>
F90    : include 'class_Person.inc'
```

if the files, class\_Person.h or class\_Person.inc, were in the same directory as your program. Otherwise, it is necessary to give the complete system path to the file, such as,

```
include '/home/caam211/Include/inv.f90'
include '/home/caam211/Include/SolveVector.f90'
```

which give source links to the caam211 course files for the function `inv(A)` for returning the inverse of a matrix  $A$ , and the function `SolveVector(A,B)` which returns the solution vector  $X$  for the matrix system  $A \cdot X = B$ .

In F90 one can also provide a “module” that defines constants, user defined types, supporting sub-programs, operators, etc. Any of those features can be accessed by first including such a F90 module before the main program and later invoking it with a “use” statement which cites the “module” name. For example, the F90 program segments:

```
include '/home/caam211/Include/caam211_operators.f90'
Program Lab2_A_2
  . . .
  call test_matrix ( A, B, X ) ! form and invert test matrix
  . . .
  subroutine test_matrix ( A, B, X )
    use caam211_operators ! included above
    implicit none
    real :: A(:,,:), B(:,), X(:)
    real :: A_inv(size(A,1),size(A,1)) ! automatic array allocation
    A_inv = inv(A)
    X = A .solve. B ! like X = A \ B in Matlab
    . . .
```

gives a source link to the `caam211` course “module” source file named `caam211_operators.f90` which contains subprograms, such as the function `inv()`, and operator definitions like `.solve.` which is equivalent to the “\” operator in MATLAB.

In the last example the omission of the “include” statement would require a compiler dependent statement to allow the system to locate the module cited in the “use” statement. For the National Algorithms Group (NAG) F90 compiler that link would be given as

```
f90 -o go /home/caam211/Include/caam211_operators.f90 my.f90
```

if the above segment was stored in the file named `my.f90`, while for the Cray F90 compiler a path flag, `-p`, to the compiled version is required, such as:

```
f90 -o go -p /home/caam211/Include/caam211_op_CRAY.o my.f90
```

Either would produce an executable file, named “go” in this example.

## 4.10 Procedural Applications

In this section we will consider two common examples of procedural algorithms: fitting curves to experimental data, and sorting numbers, strings, and derived types. Sorting concepts will be discussed again in Chapter 7.

### 4.10.1 Fitting Curves to Data

We must often deal with measurements and what they result in: data. Measurements are never exact because they are limited by instrument sensitivity and are contaminated by noise. To determine trends (how measurements are related to each other), confirm theoretical predictions, and the like, engineers must frequently *fit* functions to data. The “curve” fit is intended to be smoother than a raw plot of the data, hopefully revealing more about the underlying relation between the variables than would otherwise be apparent.

Often, these functions take *parametric* form: The functional form is specified, but has unknown coefficients. Suppose you want to fit a straight line to a dataset. With  $y$  denoting the measurement and  $x$  the independent variable, we wish to fit the function  $y = f(x) = mx + b$  to the data. The fitting process amounts to determining a few quantities of the assumed linear functional form—the parameters  $m$  and  $b$ —from the data. You know that two points define a straight line; consequently, only two of the  $(x, y)$  pairs need be used. But which two should be used? In virtually all real-world circumstances, the measurements do *not* precisely conform to the assumed functional form. Thus, fitting a curve by selecting a few values (two in the linear case) and solving for the function’s parameters produces a circumspect “fit”, to say the least. Instead, the most common approach is to use *all* the data in the curve fitting process. Because you frequently have much more data than parameters, you have what is known as an *over-determined* problem. In most cases, no parameter values produce a function that will fit all the data exactly. Over-determined problems can be solved by specifying an *error criterion* (what is an error and how large is the deviation of data from the assumed curve) and finding the set of parameter values that minimizes the error criterion. With this approach, we can justifiably claim to have found the best parameter choices.

### The “Least Squares” Approach

Far and away the most common error criterion is the *mean-squared error*: Given measurement pairs  $(x_i, y_i)$ ,  $i = 1, \dots, N$ , the mean squared error  $\epsilon^2$  equals the average across the dataset of  $(y_i - f(x_i))^2$ , the squared error between the  $i^{\text{th}}$  measurement and the assumed parametric function  $f(x_i)$ .

$$\epsilon^2 = \frac{1}{N} \sum_{i=1}^N (y_i - f(x_i))^2$$

Least squares fitting of functions to data amounts to minimizing the dataset’s mean squared error with respect to the parameters.

To illustrate the least-squares approach, let's fit a linear function to a dataset. Substituting the assumed functional form  $f(x) = mx + b$  into the expression for the mean-squared error, we have

$$\epsilon^2 = \frac{1}{N} \sum_{i=1}^N (y_i - (mx_i + b))^2$$

We can find a set of equations for the parameters  $m$  and  $b$  that minimize this quantity by evaluating the derivative of  $\epsilon^2$  with respect to each parameter and setting each to zero.

$$\frac{d\epsilon^2}{dm} = \frac{1}{N} \sum_{i=1}^N -2x_i(y_i - (mx_i + b)) = 0$$

$$\frac{d\epsilon^2}{db} = \frac{1}{N} \sum_{i=1}^N -2(y_i - (mx_i + b)) = 0$$

After some simplification, we find that we have two *linear* equations to solve for the fitting parameters.

$$\begin{aligned} m \cdot \left( \frac{1}{N} \sum_{i=1}^N x_i^2 \right) + b \cdot \left( \frac{1}{N} \sum_{i=1}^N x_i \right) &= \frac{1}{N} \sum_{i=1}^N x_i y_i \\ m \cdot \left( \frac{1}{N} \sum_{i=1}^N x_i \right) + b &= \frac{1}{N} \sum_{i=1}^N y_i \end{aligned}$$

Thus, finding the least-squares fit of a straight line to a set of data amounts to solving a set of two linear equations, the coefficients of which are computed from the data. Note that the four summations in the last equation have the same range count (N) and could be evaluated in a single explicit loop.

#### An Aside

Because fitting data with a linear equation yields a set of two easily solved equations for the parameters, one approach to fitting *nonlinear* curves to data is to convert the nonlinear problem into a linear one. For example, suppose we want to fit a *power law* to the data:  $f(x) = ax^b$ . Instead of minimizing the mean squared error directly, we transform the data so that we are fitting it with a linear curve. In the power law case, the logarithm of the fitting curve is linear in the parameters:  $\log f(x) = \log a + b \log x$ . This equation is not linear in the parameter  $a$ . For purposes of least-squares fits, we instead treat  $a' = \log a$  as the linear fit parameter, solve the resulting set of linear equations for  $a'$ , and calculate  $a = \exp a'$  to determine the power law fitting parameter. By evaluating the logarithm of  $x_i$  and  $y_i$  and applying the least squares equations governing the fitting of a linear curve to data, we can fit a power-law function to data. *Thus, calculating a linear least squares fit to data underlies general approximation of measurements by smooth curves.* For an insight to the types of relationships that can be determined, see the following summary.

$x$ -axis	$y$ -axis		Relationship
Linear	Linear	$y = mx + b$	linear
Linear	Logarithmic	$\log y = mx + b$	exponential: $y = e^b \cdot e^{mx}$
Logarithmic	Linear	$y = m \log x + b$	logarithmic
Logarithmic	Logarithmic	$\log y = m \log x + b$	power-law: $y = e^b \cdot x^m$

We can now specify the computations required by the least squares fitting algorithm mathematically.

#### Algorithm: Least-Squares Fitting of Straight Lines to Data

1. **Given**  $N$  pairs of data points  $(x_i, y_i)$

2. **Calculate**<sup>†</sup>  $a_{11} = \frac{1}{N} \sum_{i=1}^N x_i^2$ ,  $a_{12} = \frac{1}{N} \sum_{i=1}^N x_i$ ,  $a_{21} = \frac{1}{N} \sum_{i=1}^N x_i$ ,  $a_{22} = 1$ ,  $c_1 = \frac{1}{N} \sum_{i=1}^N x_i y_i$ , and  $c_2 = \frac{1}{N} \sum_{i=1}^N y_i$ .

<sup>†</sup>Note that these calculations can be performed in one loop rather than four.



### 3. Solve the set of linear equations

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

which for two equations can be done by hand to yield

$$m = (a_{12} \cdot c_2 - N \cdot c_1) / (a_{12} \cdot a_{21} - N \cdot a_{11})$$

$$b = (c_2 - m \cdot a_{12}) / N$$

### 4. Calculate the mean squared error $\epsilon^2 = \frac{1}{N} \sum_{i=1}^N (y_i - (mx_i + b))^2$ .

## Implementing the Least Squares Algorithm

In F90, such calculations can be performed two different ways: one expresses the looping construct directly, the other uses more efficient intrinsic array routines inside F90. Assuming the  $\{x_i\}$  are stored in the vector `x`, the coefficient `a12` can be calculated (at least) two ways.

1. 

```
sum_x = 0
N = size(x)
do i = 1,N
    sum_x = sum_x + x(i)
end do
a12 = sum_x/N
```
2. 

```
a12 = sum(x)/size(x)
```

Clearly, the second method produces a somewhat simpler expression than the first, and is vastly superior to the first. In the sample code that follows in Fig. 4.21 we use the intrinsic array functions but encourage the reader to check the results with a single loop that computes all six terms need to find `m` and `b`.

There are a few new features demonstrated in this example code. In line 6 we have specified a fixed unit number to associate with the data file to be specified by the user. But we did not do an `INQUIRE` to see if that unit was already in use. We will accept a user input filename (lines 8, 25 and 28) that contains the data to be fitted. An interface (lines 12-21) is provided to external routines that will determine the number of lines of data in the file and the read those data into the two arrays. Those two routines are given elsewhere. Of course, the memory for the data arrays must be dynamically allocated (line 35) before they can be read (line 37). After the least squares fit is computed (line 40) and printed the memory space for the data is freed (line 44).

In the `lsq_fit` subroutine (line 47) the three items of interest are passed in the array `fit`. (Routine `lsq_fit` could have been written as a function, try it.) Observe that `y` must be the same length as array `x` so the `size` intrinsic was used to ensure that (line 56). The data summations are evaluated with the `sum` intrinsic (lines 62-64) and it is used again to evaluate the mean squared error `mse` (line 72) as described in step 4 of the algorithm. The test data (lines 78-89) and results (lines 92-96) are given as comments as usual. Since no explicit loops have been used this form would be more efficient on vector computers and some parallel computers.

### 4.10.2 Sorting

One of the most useful computational routines is sorting: Ordering a sequence of data according to some rule. For example, the alphabetized list of filenames produced by a system directory command is far easier to read than an unsorted list would be. Furthermore, data can be fruitfully sorted in more than one way. As an example, you can sort system files by their creation date.

Sorting algorithms have been well studied by computer scientists in a quest to find the most efficient. We use here the *bubble sort algorithm*, perhaps the oldest, but not most efficient. This algorithm makes multiple passes over a list, going down the list interchanging adjacent elements in the list if needed to put them in order. For example, consider the list [`b`, `e`, `a`, `d`, `f`, `c`], shown in Fig. 4.22, that we

```

[ 1] program linear_fit
[ 2] ! -----
[ 3] !       F90 linear least-squares fit on data in file
[ 4] !       specified by the user.
[ 5] ! -----
[ 6] implicit none
[ 7] integer, parameter :: filenumber = 1 ! RISKY
[ 8] real, allocatable :: x(:), y(:) ! data arrays
[ 9] character (len = 64) :: filename ! name of file to read
[10] integer :: lines ! number of input lines
[11] real :: fit(3) ! final results
[12]
[13] interface
[14]   function inputCount(unit) result(linesOfInput)
[15]     integer, intent(in) :: unit ! file unit number
[16]     integer :: linesOfInput ! result
[17]   end function inputCount
[18]   subroutine readData (inFile, lines, x, y)
[19]     integer, intent(in) :: inFile, lines ! file unit, size
[20]     real, intent(out) :: x(lines), y(lines) ! data read
[21]   end subroutine readData
[22] end interface
[23]
[24] ! Get the name of the file containing the data.
[25] write (*,*) 'Enter the filename to read data from:'
[26] read (*, '(A64)') filename
[27]
[28] ! Open that file for reading.
[29] open (unit = filenumber, file = filename)
[30]
[31] ! Find the number of lines in the file
[32] lines = inputCount (filenumber)
[33] write (*,*) 'There were ', lines, ' records read.'
[34]
[35] ! Allocate that many entries in the x and y array
[36] allocate (x(lines), y(lines))
[37]
[38] call readData (filenumber, lines, x, y) ! Read data
[39] close (filenumber)
[40]
[41] call lsq_fit (x, y, fit) ! least-squares fit
[42] print *, "the slope is ", fit(1) ! display the results
[43] print *, "the intercept is ", fit(2)
[44] print *, "the error is ", fit(3)
[45] deallocate (y, x)
[46] contains
[47]

```

Fig. 4.21, A Typical Least Squares Linear Fit Program (continued)

wish to sort to alphabetical order. In the first pass, the algorithm begins by examining the first two list elements ( $b$ ,  $e$ ). Since they are in order, these two are left alone. The next two elements ( $e$ ,  $a$ ) are not in order; these two elements of the list are interchanged. In this way, we “bubble” the element  $a$  toward the top and  $e$  toward the bottom. The algorithm proceeds through the list, interchanging elements if need be until the last element is reached. Note that the bottom of the list at the end of the first pass contains the correct entry. This effect occurs because of the algorithm’s structure: The “greatest” element will always propagate to the list’s end. Once through the pass, we see that the list is in better, but not perfect, order. We must perform another pass just like the first to improve the ordering. Thus, the second pass need consider only the first  $n - 1$  elements, the third  $n - 2$ , etc. The second pass does make the list better formed. After more passes, the list eventually becomes sorted. To produce a completely sorted list, the bubble-sort algorithm requires no more passes than the number of elements in the list minus one.

The following F90 routines illustrate some of the initial features of a simple procedural approach to a simple process like the bubble-sort algorithm. We begin by considering the sorting of a list of real numbers as shown in subroutine `Sort_Reals` in Fig. 4.22.

In line 1 we have passed in the size of the array, and the actual array (called `database`). Note that the `database` has `intent (inout)` because we plan to overwrite the original `database` with the newly sorted order, which is done in lines 18–20. For efficiency sake we have included an integer counter, `swaps_Made`, so that we can determine if the sort has terminated early. If we wished to apply the same bubble-sort algorithm to an integer array all we would have to do is change the procedure name and lines 6 and 10 that describe the type of data being sorted (try it).

```

[48]  subroutine lsq_fit (x, y, fit)
[49]  ! -----
[50]  !           Linear least-squares fit, A u = c
[51]  ! -----
[52]  !  fit  = slope, intercept, and mean squared error of fit.
[53]  !  lines = the length of the arrays x and y.
[54]  !  x    = array containing the independent variable.
[55]  !  y    = array containing the dependent variable data.
[56]  implicit none
[57]  real,    intent(in)    :: x(:), y(size(x))
[58]  real,    intent(out)  :: fit(3)
[59]  integer  :: lines
[60]  real     :: m,      b,      mse
[61]  real     :: sumx, sumx2, sumy, sumxy
[62]
[63]  !  Summations
[64]  sumx = sum ( x )      ; sumx2 = sum ( x**2 )
[65]  sumy = sum ( y )      ; sumxy = sum ( x*y )
[66]
[67]  !  Calculate slope intercept
[68]  lines = size(x)
[69]  m = (sumx*sumy - lines*sumxy)/(sumx**2 - lines*sumx2)
[70]  b = (sumy - m*sumx)/lines
[71]
[72]  !  Predicted y points and the sum of squared errors.
[73]  mse = sum ( (y - m*x - b)**2 )/lines
[74]  fit(1) = m ; fit(2) = b ; fit(3) = mse      ! returned
[75]  end subroutine lsq_fit
[76]
[77]  end program linear_fit
[78]
[79]  ! Given test set 1 in file lsq_1.dat:
[80]  ! -5.000000 -2.004481
[81]  ! -4.000000 -1.817331
[82]  ! -3.000000 -1.376481
[83]  ! -2.000000 -0.508725
[84]  ! -1.000000 -0.138670
[85]  !  0.000000  0.376678
[86]  !  1.000000  0.825759
[87]  !  2.000000  1.036343
[88]  !  3.000000  1.815817
[89]  !  4.000000  2.442354
[90]  !  5.000000  2.636355
[91]  ! Running the program yields:
[92]  !
[93]  ! Enter the filename to read data from: lsq_1.dat
[94]  ! There were 11 records read.
[95]  ! the slope is      0.4897670746
[96]  ! the intercept is  0.2988743484
[97]  ! the error is      0.2139159478E-01

```

**Figure 4.21:** A Typical Least Squares Linear Fit Program

That is true because the compiler knows how to apply the  $>$  operator to all the standard numerical types in the language. But what if we want to sort character strings, or other types of objects? Fortran has lexical operators (like LGE) to deal with strings, but user defined objects would require that we overload the  $>$  operator, if the expected users would not find the overloading to be confusing. In other words, you could develop a fairly general sort routine if we changed lines 6 and 10 to be

```

[ 6]  type (Object), intent(inout) :: database (lines)
[10]  type (Object)                :: temp

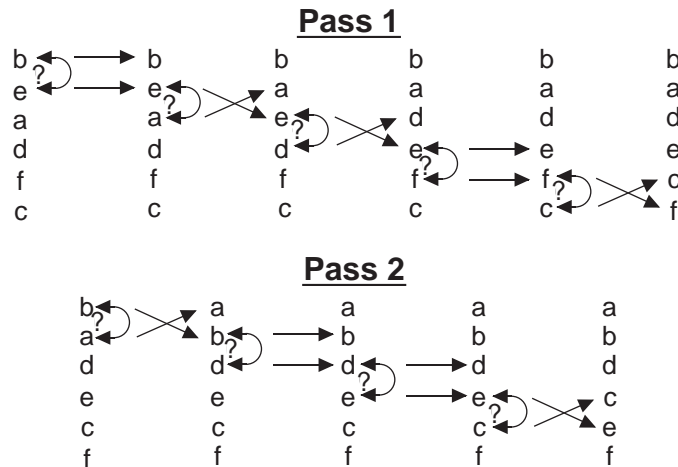
```

and provided an overloading of  $>$  so that line 17 makes sense for the defined Object (or for selected component of it).

To illustrate the sort of change that is necessary to sort character strings consider subroutine Sort\_String Fig. 4.23:

To keep the same style as the previous algorithm and overload the  $>$  operator we would have to have a procedure that utilizes the lexical operators in lines 24 and 25, along with the interface definition on lines 12 through 17, do define the meaning of  $>$  in the context of a string. While the concept of a “template” for a code to carry out a bubble-sort on any list of objects it may not always be obvious what  $>$  means when it is overloaded by you or some other programmer.

Note that in the two above sorting examples we have assumed that we had the authority to change the original database, and that it was efficient to do so. Often that is not the case. Imagine the case where the database represents millions of credit card users, each with a large number components of numbers,



**Figure 4.22:** Example passes of the bubble-sort algorithm through data.

```
[ 1] subroutine Sort_Reals (lines, database)
[ 2] ! Bubble Sort of (changed) Real Database
[ 3]
[ 4] implicit none
[ 5] integer, intent(in) :: lines ! number of records
[ 6] real, intent(inout) :: database (lines) ! records in database
[ 7]
[ 8] integer :: swaps_Made ! number of swaps made in one pass
[ 9] integer :: count ! loop variable
[10] real :: temp ! temporary holder for making swap
[11]
[12] do ! Repeat this loop forever... (until we break out of it)
[13] swaps_Made = 0 ! Initially, we've made no swaps
[14] ! Make one pass of the bubble sort algorithm
[15] do count = 1, (lines - 1)
[16] ! If item is greater than the one after it, swap them
[17] if ( database (count) > database (count + 1) ) then
[18] temp = database (count)
[19] database (count) = database (count + 1)
[20] database (count + 1) = temp
[21] swaps_Made = swaps_Made + 1
[22] end if
[23] end do
[24] ! If we made no swaps, break out of the loop.
[25] if ( swaps_Made == 0 ) exit ! do count swaps
[26] end do
[27] end subroutine Sort_Reals
```

**Figure 4.23:** Bubble Sort of a Real Array

character strings, or general objects. If many workers are accessing those data for various sorting needs you probably would not allow the original dataset to be changed for reasons of safety or security. Then we consider an alternative to moving around the actual database components. That is, we should consider using moving pointers to large data components, or pseudo-pointers such as an ordering array. The use of an ordering array is shown in Fig. 4.24 where subroutine Integer\_Sort now includes an additional argument.

The third argument has intent (out), as shown in line 7, and is an integer array of the same length as the original database which has now been changed to intent (in) so the compiler will not allow us to change the original data. If the data are properly sorted as supplied then it should not be changed and the new order should be the same as the original sequential input. That is why line 13 initializes the return order to a sequential list. Then we slightly change the previous sort logic so that lines 19 through 23 now check what's in an ordered location, and change the order number when necessary, but never change the original data. After exiting this routine you could list the information, in sorted order, without changing the original data simply by using vector subscripts in a print statement like:

```
print *, database (order).
```

```

[ 1] subroutine Sort_String (lines, database)
[ 2] ! Bubble Sort of (Changed) String Database
[ 3] implicit none
[ 4]
[ 5] integer,          intent(in)      :: lines          ! input size
[ 6] character(len=*), intent(inout) :: database (lines) ! records
[ 7]
[ 8] character (len = len(database (1))) :: temp ! swap holder
[ 9] integer :: swaps_Made              ! number of swaps in a pass
[10] integer :: count                  ! loop variable
[11]
[12] interface ! to_lower
[13]   function to_lower (string) result (new_String)
[14]     character (len = *) , intent(in) :: string
[15]     character (len = len(string))    :: new_String
[16]   end function to_lower
[17] end interface ! to_lower
[18]
[19] do ! Repeat this loop forever... (until we break out of it)
[20]   swaps_Made = 0 ! Initially, we've made no swaps
[21]   ! Make one pass of the bubble sort algorithm
[22]   do count = 1, (lines - 1)
[23]     ! If the element is greater than the one after it, swap them
[24]     if ( LGT (to_lower (database (count )),
[25]               to_lower (database (count + 1))) ) then
[26]       temp = database (count )
[27]       database (count ) = database (count + 1)
[28]       database (count + 1) = temp
[29]       swaps_Made = swaps_Made + 1
[30]     end if
[31]   end do
[32]   ! If we made no swaps, break out of the loop.
[33]   if ( swaps_Made == 0 ) exit ! do count swaps
[34] end do
[35] end subroutine Sort_String

```

**Figure 4.24:** Bubble Sort of an Array of Character Strings

```

[ 1] subroutine Integer_Sort (lines, database, order)
[ 2] ! Ordered Bubble Sort of (Unchanged) Integer Database
[ 3]
[ 4] implicit none
[ 5] integer, intent(in) :: lines          ! number of records
[ 6] integer, intent(in) :: database (lines) ! records in database
[ 7] integer, intent(out) :: order (lines) ! the order array
[ 8]
[ 9] integer :: swaps_Made              ! number of swaps made in one pass
[10] integer :: count                  ! loop variable
[11] integer :: temp                   ! temporary holder for making swap
[12]
[13] order = (/ (count, count = 1, lines) /) ! default order
[14] do ! Repeat this loop forever... (until we break out of it)
[15]   swaps_Made = 0 ! Initially, we've made no swaps
[16]   ! Make one pass of the bubble sort algorithm
[17]   do count = 1, (lines - 1)
[18]     ! If item is greater than the one after it, swap them
[19]     if ( database (order (count)) >
[20]           database (order (count + 1)) ) then
[21]       temp = order (count)
[22]       order (count) = order (count + 1)
[23]       order (count + 1) = temp
[24]       swaps_Made = swaps_Made + 1
[25]     end if
[26]   end do
[27]   ! If we made no swaps, break out of the loop.
[28]   if ( swaps_Made == 0 ) exit ! do count swaps
[29] end do
[30] end subroutine Integer_Sort

```

**Figure 4.25:** An Ordered Bubble Sort of an Integer Array

Clearly you could write a very similar program using a true “pointer” array since they are now standard in Fortran.

Next we will start to generalize the idea of sorting to include the sorting of objects that may have numerous components. Assume the each record object to be read is defined as in Fig. 4.25.

There may be thousands, or millions, of such records to be read from a file, sorted by name and/or number, and then displayed in sorted order. Program test\_bubble, in Fig. 4.26 illustrates one approach to such a problem. Here since the database of records are to read from a file we do not yet know how many

```

[ 1] module record_Module
[ 2] !-----
[ 3] ! record_Module holds the "record" type
[ 4] !-----
[ 5] ! record is a data structure with two names and an id number.
[ 6] type record
[ 7]   character (len=24) :: last_Name   ! last name
[ 8]   character (len=24) :: first_Name  ! first name
[ 9]   integer           :: id          ! id number
[10] end type record
[11] end module record_Module

```

**Figure 4.26:** A Typical Record in a List to be Sorted

```

[ 1] program test_bubble
[ 2] !-----
[ 3] ! test_bubble asks for a filename for a file of names and id
[ 4] ! numbers, loads in the data from a file into the database,
[ 5] ! finds sorting orders, and prints sorted data
[ 6] !-----
[ 7] use record_Module      ! need this to use the 'record' type
[ 8] implicit none
[ 9] ! We define the database as an allocatable array of records.
[10] type (record), allocatable :: database (:)
[11]
[12] ! These arrays hold the sorted order of the database entries.
[13] integer, allocatable :: sort_by_Name (:)
[14] integer, allocatable :: sort_by_Number (:)
[15]
[16] character (len = 64) :: file_Name   ! file to read data from
[17] integer              :: lines       ! number of lines of input
[18] integer              :: file_Number ! the input file number
[19] integer              :: loop_Count  ! loop counter
[20]
[21]   file_Number = 1      ! arbitrarily set file_Number to 1
[22]
[23]   write (*,*) 'Enter the filename to read data from:'
[24]   read  (*, '(A64)') file_Name
[25]
[26]   ! Open our file and assign the number to 'file_Number'
[27]   open (unit = file_Number, file = file_Name)
[28]
[29]   ! Find the number of lines in the input file with input_Count.
[30]   lines = input_Count (file_Number)
[31]   write (*,*) 'There are ', lines, ' records.'
[32]
[33]   ! Allocate that many entries in the database and order arrays
[34]   allocate ( database (lines) )
[35]   allocate ( sort_by_Name (lines), sort_by_Number (lines) )
[36]
[37]   ! Read the data from file into the database and close the file.
[38]   call read_Data (file_Number, lines, database)
[39]   close (file_Number)
[40]
[41]   ! Sort the database by name; the order will be in sort_by_Name.
[42]   call String_Sort (lines, database (:)%last_Name, sort_by_Name)
[43]   write (*,*) 'Data sorted by name: ' ; write (*,*)
[44]
[45]   ! Print out the data in the database sorted by name
[46]   call show_Data (lines, database, sort_by_Name)
[47]   write (*,*) 'Data sorted by number: ' ; write (*,*)
[48]
[49]   ! Sort the database by id numbers; new order is sort_by_Number.
[50]   call Integer_Sort (lines, database (:)%id, sort_by_Number)
[51]
[52]   ! Print out the data in the database sorted by number.
[53]   call show_Data (lines, database, sort_by_Number)
[54] end program test_bubble

```

**Figure 4.27:** Testing of Ordered Bubble Sorts

there are to be stored. Therefore, it is declared allocatable in line 13, and allocated later in line 34 after we have evaluated the file size of a file named by the user. Although not generally necessary we have selected to have an order array for names and a different one for numbers. The are `sort_by_Name`, and `sort_by_Number`, respectively and are treated in a similar fashion to the database memory allocation as noted in lines 13–14, and line 35.

In line 21 we have arbitrarily set a unit number to be used for the file. That is okay for a very small code, but an unnecessary and unwise practice in general. The Fortran intrinsic `inquire` allows one to

determine which units are inactive and we could create a function, say `Get_Next_Unit`, to select a safe unit number for our input operation. After accepting a file name we open the unit, and count the number of lines present in the file (see line 30). Had the database been on the standard input device, and not contained any non-printing control characters, we could have easily read it with the statement

```
read *, database
```

However, it does contain tabs (ASCII character number 9), and is in a user defined file instead of the standard input device so line 38 invokes subroutine `read_Data` to get the data base. Of course, once the tabs and commas have been accounted for and the names and id number extracted it uses an intrinsic constructor on each line to form its database entry like:

```
database (line_Count) = Record (last, first, id)
```

After all the records have been red into the database note that line 42 extracts all the last names with the syntax

```
database (:) last_Name
```

so they are copied into subroutine `String_Sort`, as its second argument, and the ordered list `sort_by_Name` is returned to allow operations that need a last name sort. Likewise, subroutine `Integer_Sort`, shown above, is used in line 50 to sort the id numbers and save the data in order list `sort_by_Number`. The ordered lists are used in `show_Data`, in lines 46 and 53, to display the sorted information, without changing the original data.

If the supplied file, say `namelist`, contained data in the format of (String comma String tab Number) with the following entries:

```
[ 1] Indurain, Miguel 5623
[ 2] van der Aarden, Eric 1245
[ 3] Rominger, Tony 3411
[ 4] Sorensen, Rolf 341
[ 5] Yates, Sean 8998
[ 6] Vandiver, Frank 45
[ 7] Smith, Sally 3821
[ 8] Johnston, David 3421
[ 9] Gillis, Malcolm 3785
[10] Johns, William 7234
[11] Johnston, Jonathan 7234
[12] Johnson, Alexa 5190
[13] Kruger, Charlotte 2345
[14] Butera, Robert 7253
[15] Armstrong, Lance 2374
[16] Hegg, Steve 9231
[17] LeBlanc, Lucien 23
[18] Peiper, Alan 5674
[19] Smith-Jones, Nancy 9082
```

The output would be:

```
[ 1] ! Enter the filename to read data from: namelist
[ 2] ! There are 19 records.
[ 3] !
[ 4] ! Data sorted by name:
[ 5] !
[ 6] ! Armstrong          Lance          2374
[ 7] ! Butera             Robert         7253
[ 8] ! Gillis             Malcolm        3785
[ 9] ! Hegg                Steve          9231
[10] ! Indurain            Miguel         5623
[11] ! Johns               William        7234
[12] ! Johnson             Alexa          5190
[13] ! Johnston            David          3421
[14] ! Johnston            Jonathan       7234
[15] ! Kruger              Charlotte      2345
[16] ! LeBlanc             Lucien         23
[17] ! Peiper              Alan           5674
[18] ! Rominger            Tony           3411
[19] ! Smith               Sally          3821
[20] ! Smith-Jones         Nancy          9082
[21] ! Sorensen            Rolf           341
[22] ! van der Aarden      Eric           1245
[23] ! Vandiver            Frank          45
[24] ! Yates              Sean           8998
[25] !
```

and

```

[26] ! Data sorted by number:
[27] !
[28] ! LeBlanc           Lucien           23
[29] ! Vandiver          Frank            45
[30] ! Sorensen           Rolf            341
[31] ! van der Aarden     Eric            1245
[32] ! Kruger             Charlotte       2345
[33] ! Armstrong          Lance           2374
[34] ! Rominger          Tony            3411
[35] ! Johnston          David           3421
[36] ! Gillis            Malcolm          3785
[37] ! Smith             Sally           3821
[38] ! Johnson           Alexa           5190
[39] ! Indurain          Miguel          5623
[40] ! Peiper            Alan            5674
[41] ! Johns             William         7234
[42] ! Johnston          Jonathan        7234
[43] ! Butera            Robert          7253
[44] ! Yates             Sean            8998
[45] ! Smith-Jones       Nancy           9082
[46] ! Hegg             Steve           9231

```

Pass 1						Pass 2					Pass 3	
Level						Level					Level	
1	2	3	4	5	6	1	2	3	4	5	1	2
Sorted						Sorted					Sorted	
b	b	b	b	b	b	b	a	a	a	a	a	a
e	e	a	a	a	a	a	b	b	b	b	b	b
a	a	e	d	d	d	d	d	d	d	d	d	c
d	d	d	e	e	e	e	e	e	e	c	c	d
f	f	f	f	f	c	c	c	c	c	e	e	e
c	c	c	c	c	f	f	f	f	f	f	f	f

Is_Was*					Is_Was				Is_Was	
1	1		1	1		1	3	3	3	3
2	3		3	3		3	1	1	1	1
3	2		4	4		4	4	4	4	6
4	4		2	2		2	2	6	6	4
5	5		5	6		6	6	2	2	2
6	6		6	5		5	5	5	5	5

\*  $\text{Is\_Was}(j) = k$ . What is position  $j$  was position  $k$ .

Figure 4.28: Sorting via an Order Vector, Array (Is\_Was)  $\rightarrow$  a b c d e f

## 4.11 Exercises

- 1 Frequently we need to know how many lines exist in an external file that is to be used by our program. Usually we need that information to dynamically allocate memory for the arrays that will be constructed from the file data to be read. Write a F90 program or routine that will accept a unit number as input, open that unit, loop over the lines of data in the file connected to the unit, and return the number of lines found in the file. (A external file ends when the iostat from a read is less than zero.)
- 2 A related problem is to read a table of data from an external file. In addition to knowing the number of lines in the file it is necessary to know the number of entities (columns) per line and to verify that all lines of the file have the same number of columns. Develop a F90 program for that purpose. (This is the sort of checking that the MATLAB load function must do before loading an array of data.)



- 3 Write a program that displays the current date and time and uses the module `tic_toc`, in Fig. 4.10, to display the CPU time required for a calculation.
- 4 Develop a companion function called `to_upper` that converts a string to all upper case letters. Test it with the above program.
- 5 Develop a function that will take an external file unit number and count the number of lines in the file connected to that unit. This assumes that the file has been “opened” on that unit. The interface to the function is to be:

```
interface
  function inputCount(unit) result(linesOfInput)
    integer, intent(in) :: unit          ! file unit number
    integer              :: linesOfInput ! result
  end function inputCount
end interface
```

- 6 Assume the file in the previous problem contains two real values per line. Develop a subroutine that will read the file and return two vectors holding the first and second values, respectively. The interface to the subroutine is to be:

```
interface
  subroutine readData (inFile, lines, x, y)
    integer, intent(in) :: inFile, lines ! file unit, size
    real,    intent(out) :: x(lines), y(lines) ! data read
  end subroutine readData
end interface
```

- 7 Written replies to the questions given below will be required. All of the named files are provided in source form as well as being listed in the text. The cited Figure number indicates where some or all of the code is discussed in the text.

(a) *Figure 1.3* — `hello.f90`

What is necessary to split the printing statement so that “Hello,” and “world” occur on different program lines? That is, to continue it over two lines?

(b) *Figure 4.1* — `arithmetic.f90`

What is the meaning of the symbol `( mod )` used to get the `Mod_Result`?

What is the meaning of the symbol `( ** )` used to get the `Pow_Result`?

(c) *Figure 4.3* — `array_index.f90`

Is it good practice to use a loop index outside the loop? Why?

(d) *Figure 4.4* — `more_or_less.f90`

What does the symbol `( > )` mean here?

What does the symbol `( == )` mean here?

(e) *Figure 4.5* — `if_else.f90`

What does the symbol `( .and. )` mean here? Can its preceding and following arguments be interchanged (is it commutative)?

(f) *Figure 4.6* — `and_or_not.f90`

What does the symbol `( .not. )` mean here?

What does the symbol `( .or. )` mean here? Can its preceding and following arguments be interchanged (is it commutative)?

(g) *Figure 4.7* — `clip.f90`

What does the symbol `( <= )` mean here?

(h) *Figure 4.8* — `maximum.f90`

What are the input and output arguments for the `maxint` function?

- 8 The vertical motion of a projectile at any time,  $t$ , has a position given by  $y = y_0 + V_0 * t - 1/2 * g * t^2$ , and a velocity of  $V = V_0 - g * t$  when upward is taken as positive, and where the initial conditions on the starting position and velocity, at  $t = 0$ , are  $y_0$  and  $V_0$ , respectively. Here the gravitational acceleration term,  $g$ , has been taken downward. Recall that the numerical value of  $g$  depends on the units employed. Use metric units with  $g = 9.81 m/s^2$  for distances measured in meters and time in seconds.

Write a C++ or F90 program that will accept initial values of  $y_0$  and  $V_0$ , and then compute and print  $y$  and  $V$  for each single input value of time,  $t$ . Print the results for  $y_0 = 1.5$  meters and  $V_0 = 5.0 m/s$  for times  $t = 0.5, 2.0$ , and  $4.0$  seconds.

- 9 Modify the projectile program written in Problem 2 to have it print the time, position, and velocity for times ranging from  $0.0$  to  $2.0$  seconds, in increments of  $0.05$  seconds. If you use a direct loop do not use real loop variables. Conclude the program by having it list the approximate maximum (positive) height reached and the time when that occurred. The initial data will be the same, but should be printed for completeness. The three columns of numbers should be neat and right justified. In that case the default print format (print \* in F90) will usually not be neat and one must employ a “formatted” print or write statement.

- 10 The Greatest Common Divisor of two positive integers can be computed by at least two different approaches. There is a looping approach known as the Euclidean Algorithm which has the following pseudocode:

```
Rank two positive integers as max and min.
do while min > 0
  Find remainder of max divided by min.
  Replace max by min.
  Replace min by the remainder
end do
Display max as the greatest common divisor.
```

Implement this approach and test with  $max = 532 = 28 * 19$  and  $min = 112 = 28 * 8$ . The names of the remainder functions are given in Table 4.7.

Another approach to some algorithms is to use a “recursive” method which employs a subprogram which calls itself. This may have an advantage in clarifying the algorithm, and/or in reducing the round off error associated with the computations. For example, in computer graphics Bernstein Polynomials are often used to display curves and surfaces efficiently by using a recursive definition in calculating their value at a point.

The Greatest Common Divisor evaluation can also be stated in terms of a recursive function, say gcd, having max and min as its initial two arguments. The following pseudocode defines the function:

```
gcd(max, min) is
  a) max if min = 0, otherwise
  b) gcd(min, remainder of max divided by min) if min > 0
```

Also implement this version and verify that it gives the same result as the Eulerian Algorithm. Note that F90 requires the use of the word “recursive” when defining the subprogram statement block. For example,

```
recursive function gcd(...) result(g)
  ....
end function gcd
```

- 11** It is not uncommon for data files to be prepared with embedded tabs. Since it is a non-printing control character you can not see it in a listing. However, if you read the file expecting an integer, real, or complex variable the tab will cause a fatal read error. So one needs a tool to clean up such a file.

Write a program to read a file and output a duplicate copy, except that all tabs are replaced with a single space. One could read a complete line and check its characters, or read the file character by character. Remember that C++ and F90 have opposite defaults when advancing to a new line. That is, F90 advances to the next line, after any read or write, unless you include the format control, `advance = 'no'`, while C++ does not advance unless you include the new line control, `"<<endl"`, and C does not advance unless you include the new line control, `"\n"`.

- 12** Engineering data files consisting of discrete groups of variable types often begin with a control line that lists the number of rows and columns of data, of the first variable type, that follow beginning with the next line. At the end of the data block, the format repeats: control line, variable type data block, etc. until all the variable types are read (or an error occurs where the end of file is encountered). Write a program that reads such a file which contains an integer set, a real set, and a second real set.
- 13** Neither C++ or F90 provides an inverse hyperbolic tangent function. Write such a function, called `arctanh`. Test it with three different arguments against the values given by MATLAB.
- 14** Often if one is utilizing a large number of input/output file units it may be difficult to keep up with which one you need. One approach to dealing with that problem may be to define a `unit_Class` or to create an `units_Module` to provide functionality and global access to file information. In the latter case assume that we want to provide a function to simply find a unit number that is not currently in use and utilize it for our input/output action:

```
interface
  function get_next_io_unit () result (next)
    integer :: next    ! the next available unit number
  end function get_next_io_unit
end interface
```

Use the Fortran INQUIRE statement to build such a utility. If you are familiar with Matlab you will see this is similar to its `fopen` feature.

## Chapter 5

# Object Oriented Methods

### 5.1 Introduction

In Section 1.7 we outlined procedures that should be considered while conducting the object-oriented analysis and object-oriented design phases that are necessary before the OOP can begin. Here we will expand on those concepts, but the reader is encouraged to read some of the books on those subjects. Many of the references on OOA and OOD rely heavily on detailed graphical diagrams to describe the classes, their attributes and states, and how they interact with other classes. Often those OO methods do not go into any programming language specific approaches. Our interest is on OOP so we usually will assume that the OOA and OOD have been completed and supplied to us as a set of tables that describe the application, and possibly a software interface contract. Sometimes we will use a subset of the common OO methods diagrams to graphically represent the attributes and members of our classes. Since they being used for OOP the graphical representations will contain, in part, the intrinsic data type descriptions of the language being employed, as well as the derived types created with them.

### 5.2 The Drill Class

Our first illustration of typical OO methods will be to apply them to a common electric drill. It feeds a rotating cutting bit through a workpiece, thereby removing a volume of material. The effort (power or torque) required to make the hole clearly depends on the material of the workpiece, as well as the attributes of the drill.

Table 5.1 contains a summary of the result of an OO analysis for the drill object. They are further processed in Table 5.2 which gives the results of the OO design phase. When the OOD phase is complete we can create the graphical representation of our `Drill` class as shown in Fig. 5.1. At this point one can begin the actual OOP in the target language. The coding required for this object is so small we could directly put it all together in one programming session. However, that is usually not the case. Often segments of the coding will be assigned to different programming groups that must interface with each other to produce a working final code. Often this means that the OOP design starts with defining the interfaces to each member function. That is, all of the given and return arguments must be defined with respect to their type, whether they are changed by the member, etc. Such an interface can be viewed as a contract between a software supplier and a client user of the software. Once the interface has been finalized, it can be written and given to the programmer to flesh out the full routine, but the interface itself can not be changed.

The interface prototype for our drill object members are given in Fig. 5.2. In this case the remaining coding is defined by a set of equations that relate the object attributes, selected member results, material data, and a few conversion constants to obtain the proper units. Those relationships are given as:

### Attributes

What knowledge does it possess or require?

- Rotational speed (revolutions per minute)
- Feed rate per revolution (mm/rev)
- Diameter of the bit (mm)
- Power consumed (W)

### Behavior

What questions should it be able to answer?

- What is the volumetric material removal rate? (mm<sup>3</sup>/s)
- What is the cutting torque? (N·m)
- What is the material being removed?

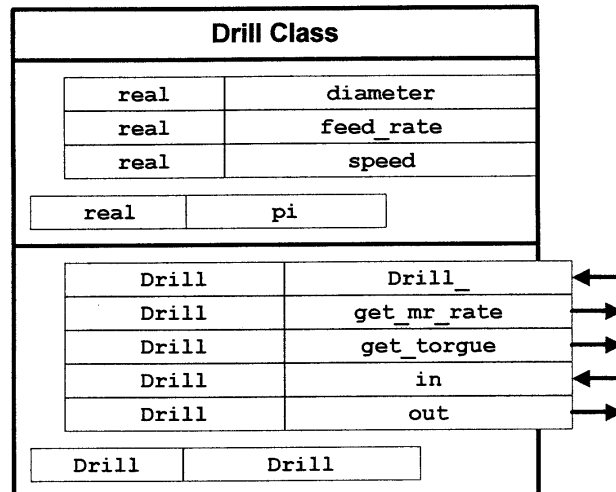
### Interfaces

What entities need to be input or output?

- Material data
- Torque produced
- Power

**Table 5.1:** Electric Drill OO Analysis

Area :	$A = \pi d^2 / 4$	(mm <sup>2</sup> )
Angular velocity :	$\omega$ , 1 rev/min = $\frac{2\pi}{60}$ rad/s	(rad/s)
Material removal rate :	$M = A \cdot \text{feed} \cdot \omega$	(mm <sup>3</sup> /s)
Power :	$P = m \cdot u = T \cdot \omega$	(W)
Torque :	$T = P/\omega$ , 1 m = 1000 mm	(N·mm)
Diameter :	d	(mm)
Feed rate :	feed	(mm/rev)
Material dissipation :	u	(W·s/mm <sup>3</sup> )



**Figure 5.1:** Graphical Representation of an Electric Drill Class

The full implementation of the drill class is given in Fig. 5.3, and a main program to test the drill class is given in Fig. 5.4. When we wrote the manual constructor, `Drill_`, in this example we chose to

```

[ 1] interface
[ 2] ! type (Drill) :: x ; x = Drill (d, f, s) ! intrinsic constructor
[ 3]
[ 4] function Drill_ (d, f, s) result (x) ! default constructor
[ 5]   real, optional :: d, f, s          ! given diameter, feed, speed
[ 6]   type (Drill)   :: x                ! the Drill instance
[ 7] end function Drill_
[ 8]
[ 9] function get_mr_rate (x) result (r) ! material removal rate
[10]   type (Drill), intent(in) :: x      ! a given drill instance
[11]   real                     :: r      ! volume cut rate
[12] end function get_mr_rate
[13]
[14] function get_torque (x, unit_Power) result (t) ! torque from power
[15]   type (Drill), intent(in) :: x      ! given drill instance
[16]   real, intent(in) :: unit_Power    ! dissipated in cutting
[17]   real              :: t            ! resulting torque
[18] end function get_torque
[19]
[20] subroutine in (x)                      ! read a Drill instance
[21]   type (Drill), intent(out) :: x      ; end subroutine in
[22]
[23] subroutine out (x)                     ! output a Drill instance
[24]   type (Drill), intent(in) :: x       ! given drill instance
[25] end subroutine out
[26] end interface

```

**Figure 5.2:** Drill Object Contract Interface Prototype

```

[ 1] module class_Drill                      ! class name
[ 2]   implicit none                        ! enforce strong typing
[ 3]   real, parameter :: pi = 3.141592654 ! or use math_constants
[ 4]   public :: Drill, Drill_, get_mr_rate, get_torque
[ 5]   real, private :: diameter, feed, speed
[ 6]
[ 7]   type Drill                      ! defined type, private data
[ 8]     real :: diameter, feed, speed ; end type
[ 9]
[10] contains ! member functions, overloaded & new operators
[11]
[12] ! type (Drill) :: x ; x = Drill (d, f, s) ! intrinsic constructor
[13]
[14] function Drill_ (d, f, s) result (x) ! default constructor
[15]   real, optional :: d, f, s          ! given diameter, feed, speed
[16]   type (Drill)   :: x                ! the Drill instance
[17]   if ( present(d) .and. present(f) .and. present(s) ) then
[18]     x = Drill (d, f, s)              ! intrinsic constructor
[19]   else
[20]     if ( .not. ( present(d) ) ) then ! no diameter given
[21]       x = Drill (10., 0., 0.)        ! default 10mm, at rest zero
[22]     end if ! default form
[23]   end if ! full form
[24] end function Drill_
[25]
[26] function get_mr_rate (x) result (r) ! material removal rate, mm3/sec
[27]   type (Drill), intent(in) :: x      ! a given drill instance
[28]   real                     :: r      ! volume cut rate
[29]   r = 0.25 * pi * x%diameter * x%feed * x%speed/60.
[30] end function get_mr_rate
[31]
[32] function get_torque (x, unit_Power) result (t) ! torque from power
[33]   type (Drill), intent(in) :: x      ! given drill instance
[34]   real, intent(in) :: unit_Power    ! dissipated in cutting
[35]   real              :: t            ! resulting torque
[36]   real              :: rad_per_sec ! radians per second
[37]   rad_per_sec = 2 * pi * x%speed / 60.
[38]   t = get_mr_rate(x) * unit_Power / rad_per_sec ! torque
[39] end function get_torque
[40]
[41] subroutine in (x)                      ! input a Drill instance
[42]   type (Drill), intent(out) :: x      ! given drill instance
[43]   read *, x                          ! get intrinsic data
[44] end subroutine in
[45]
[46] subroutine out (x)                     ! output a Drill instance
[47]   type (Drill), intent(in) :: x      ! given drill instance
[48]   print *, "Drill"; print *, " Diameter: ", x % diameter
[49]   print *, " Feed      : ", x % feed; print *, " Speed      : ", x % speed
[50] end subroutine out
[51] end module class_Drill                  ! close class definition

```

**Figure 5.3:** A Electrical Drill Class

#### Attributes

<i>Name</i>	<i>Type</i>	<i>Private</i>	<i>Description</i>
diameter	real	Y	Bit diameter (mm)
feed	real	Y	Bit feed rate (mm/rev)
speed	real	Y	Bit rotational speed (rpm)

#### Behavior

<i>Name</i>	<i>Private</i>	<i>Description</i>
drill_	N	Default constructor using all attributes, or none
get_mr_rate	N	Material removal rate (mm <sup>3</sup> /sec)
get_torque	N	Required torque (N·m)
power	N	Required power (W)

#### Data

<i>Name</i>	<i>Description</i>
u	Material power description per unit volume (W s/mm <sup>3</sup> )

#### Interfaces

<i>Name</i>	<i>Description</i>
read	Input drill and material data
print	Output object results

**Table 5.2:** Electric Drill OO Design

utilize the intrinsic constructor `Drill` (in lines 18 and 21) rather than including lines to assign values to each of the components of our data type. If at some later time we add or delete a component in the type declaration then the number of required arguments for the intrinsic constructor would also change. That would require the revision of all members that used the intrinsic constructor. An advantage of the object-oriented approach to programming is that we know that all such routines (that can access the intrinsic constructor) are encapsulated within this class declaration module, and we can be sure that no other code segments must be changed to remain consistent with the new version. That is, OOP helps with code maintenance.

## 5.3 Global Positioning Satellite Distances

Consider the problem of traveling by ship or airplane between two points on the earth. Here we assume that there are no physical obstructions that prevent the vehicle from following the shortest path, which is an arc of a "great circle" on the earth's surface. We will neglect the altitude of the airplane in comparison to the earth's radius. The original and final positions are to be defined in terms of their angles of latitude (measured N or S from the equator) and longitude (measured E or W from Greenwich, England). These two attributes define an angular position from a defined reference point on the spherical surface of the earth. They are measured in terms of whole degrees, whole minutes (1 degree = 60 minutes), and seconds (1 minute = 60 seconds). Historically, whole seconds are usually used, but they give positions that are only accurate to about 300 meters. Thus, we will use a real variable for the seconds to allow for potential reuse for the software for applications that require more accuracy, such as those using Global Positioning Satellite (GPS) data. Recall that latitude and longitude have associated directional information of North or South, and East or West, respectively. Also in defining a global position point it seems logical to include a name for each position. Depending on the application the name may identify a city or port, or a "station number" in a land survey, or a "path point number" for a directed robot motion.

Eventually, we want to compute the great arc distance between given pairs of latitude and longitude. That solid geometry calculation requires that one use angles that are real numbers measured in radians ( $2\pi = 360$  degrees). Thus our problem description begins with an `Angle` class as its basic class. Both latitude and longitude will be defined to be of the `Position_Angle` class and we observe that a `Position_Angle` is a "Kind-Of" `Angle`, or a `Position_Angle` has an "Is-A" relationship to an `Angle`. The positions we seek are on a surface so only two measures (latitude and longitude) are needed

```

[ 1] program main                ! test the Drill class
[ 2]   use class_Drill          ! i.e., all public members and public data
[ 3]   implicit none
[ 4]   type (Drill) :: drill_A, drill_B, drill_C
[ 5]   real          :: unit_Power
[ 6]   print *, "Enter diameter (mm), feed (mm/rev), speed (rpm):"
[ 7]   call in (drill_A)
[ 8]   print *, "Enter average power unit for material ( W.s/mm**3):"
[ 9]   read *, unit_Power ; call out (drill_A) ! user input
[10]   print *, "Material removal rate is: ", get_mr_rate(drill_A), &
[11]         " mm**3/sec"
[12]   print *, "Torque in this material is: ", &
[13]         & get_torque (drill_A, unit_Power), " W.s"
[14]   drill_B = Drill_ (5., 4., 3.); call out (drill_B) ! manual
[15]   drill_C = Drill_ (); call out (drill_C) ! default
[16] end program                ! Running gives
[17] ! Enter diameter (mm), feed (mm/rev), speed (rpm): 10 0.2 800
[18] ! Enter average power unit for material ( W.s/mm**3): 0.5
[19] ! Drill
[20] ! Diameter: 10.
[21] ! Feed      : 0.200000003
[22] ! Speed      : 800.
[23] ! Material removal rate is: 209.439514 mm**3/sec
[24] ! Torque in this material is: 1.25 W.s
[25] ! Drill
[26] ! Diameter: 5.
[27] ! Feed      : 4.
[28] ! Speed      : 3.
[29] ! Drill
[30] ! Diameter: 10.
[31] ! Feed      : 0.E+0
[32] ! Speed      : 0.E+0

```

**Figure 5.4:** Testing a Electrical Drill Class

to uniquely define the location, which we will refer to as the `Global_Position`. Here we see that the two `Position_Angle` object values are a "Part-Of" the `Global_Position` class, or we can say that a `Global_Position` "Has-A" `Position_Angle`.

The sort of relationships between classes that we have noted above are quite common and relate to the concept of inheritance as a means to reuse code. In an "Is-A" relationship, the derived class is a variation of the base class. Here the derived class `Position_Angle` forms an "Is-A" relation to the base class, `Angle`. In a "Has-A" relationship, the derived class has an attribute or property of the base class. Here the derived class of `Global_Position` forms a Has-A relation to its base class of `Position_Angle`. Also, the `Great_Arc` class forms a "Has-A" relation to the `Global_Position` class.

Looking back at previous classes, in Chapter 3, we observe that the class `Student` "Is-A" variation of the class `Person` and the class `Person` forms at least one "Has-A" relationship with the class `Date`. In general we know that a graduate student is a "Kind-Of" student, but not every student is a graduate student. This subtyping, or "Is-A" relationship is also called interface inheritance. Likewise, complicated classes can be designed from simpler or composition inheritance.

The OO Analysis Tables for the classes of `Great_Arc`, `Global_Position`, `Position_Angle`, and `Angle` are given in Tables 5.3 through 5.6, respectively. Historically people have specified latitude and longitude mainly in terms of whole (integer) degrees, minutes, and seconds. Sometimes you find navigation charts that give positions in whole degrees and decimal minutes. Today GPS data are being used for various high accuracy positioning such as land surveys, or the control of robots as they move over distances of a few meters. The latter will clearly need decimal seconds values in their constructor. Thus, we will create a number of constructors for the position angles. In the next chapter we will review how to access any of them, based on the signature of their arguments, through the use of a single polymorphic routine name. These considerations and the OOA tables lead to the construction of the corresponding set of OO Design Tables given in Tables 5.7 through 5.10. Those OOD tables could lead to software interface contracts to be distributed to the programming groups. When combined and tested they yield the corresponding class modules which are shown for the classes `Angle`, `Position_Angle`, `Global_Position`, and `Great_Arc` in Figs. 5.6 to 5.12, respectively. They in turn are verified by the main program given in Fig. 5.13 along with its output.



#### Attributes

What knowledge does it possess or require?

- Global position 1 (latitude, longitude)
- Global position 2 (latitude, longitude)
- Smallest arc (km)
- Radius of the earth (km)

#### Behavior

What questions should it be able to answer?

- What is the (smallest) great arc between the points

What services should it provide?

- Default value (Greenwich, Greenwich, 0.0)
- Initialize for two positions
- Convert kilometers to miles

#### Relationships

What are its related classes?

- Has-A pair of Global \_ Positions

#### Interfaces

What entities need to be input or output?

- The distance between two positions.

**Table 5.3:** Great Arc OO Analysis

#### Attributes

What knowledge does it possess or require?

- Latitude (degrees, minutes, seconds, and direction)
- Longitude (degrees, minutes, seconds, and direction)

#### Behavior

What questions should it be able to answer?

- What is the latitude of the location
- What is the longitude of the location

What services should it provide?

- Default position (Greenwich)
- Initialize a position (latitude and longitude)

#### Relationships

What are its related classes?

- Part-Of GreatArc
- Has-A pair of Position \_ Angles

#### Interfaces

What entities need to be input or output?

- The latitude and longitude, and a position name.

**Table 5.4:** Global Position OO Analysis

#### Attributes

What knowledge does it possess or require?

- Magnitude (degrees, minutes, seconds)
- Direction (N or S or E or W)

#### Behavior

What questions should it be able to answer?

- What is its magnitude and direction

What services should it provide?

- Default value (0, 0, 0.0, N)
- Initialization to input value

#### Relationships

What are its related classes?

- Part-Of Global \_ Positions
- Is-A Angle

#### Interfaces

What entities need to be input or output?

- None

**Table 5.5:** Position Angle OO Analysis

#### Attributes

What knowledge does it possess or require?

- Signed value (radians)

#### Behavior

What questions should it be able to answer?

- What is the current value

What services should it provide?

- default values (0.0)
- Conversion to signed decimal degrees
- Conversion to signed degree, minutes, and decimal seconds
- Conversion from signed decimal degrees
- Conversion from signed degree, minutes, and decimal seconds

#### Relationships

What are its related classes?

- Base Class for Position \_ Angle

#### Interfaces

What entities need to be input or output?

- None

**Table 5.6:** Angle OO Analysis

#### Attributes

<i>Name</i>	<i>Type</i>	<i>Private</i>	<i>Description</i>
point _ 1	Global _ Position	Y	Lat-Long-Name of point 1
point _ 2	Global _ Position	Y	Lat-Long-Name of point 2
arc	real	Y	Arc distance between points

#### Behavior

<i>Name</i>	<i>Private</i>	<i>Description</i>
Great _ Arc _	N	Constructor for two position points
get _ Arc	N	Compute great arc between two points

#### Data

<i>Name</i>	<i>Description</i>
Earth _ Radius _ Mean	Conversion factor
m _ Per _ Mile	Conversion factor

#### Interfaces

<i>Name</i>	<i>Description</i>
List _ Great _ Arc	Print arc values (two positions and distance)
List _ Pt _ to _ Pt	Print distance and two points

**Table 5.7:** Class Great \_ Arc OO Design

#### Attributes

<i>Name</i>	<i>Type</i>	<i>Private</i>	<i>Description</i>
latitude	Position _ Angle	Y	Latitude
longitude	Position _ Angle	Y	Longitude
name	characters	Y	Point name

#### Behavior

<i>Name</i>	<i>Private</i>	<i>Description</i>
Global _ Position _	N	Constructor for d-m-s pairs and point name
set _ Lat _ and _ Long _ at	N	Constructor for lat-long-name set
get _ Latitude	N	Return latitude of a point
get _ Longitude	N	Return longitude of a point
set _ Latitude	N	Insert latitude of a point
set _ Longitude	N	Insert longitude of a point

#### Data

<i>Name</i>	<i>Description</i>
None	

#### Interfaces

<i>Name</i>	<i>Description</i>
List _ Position	Print name and latitude, longitude of a position

**Table 5.8:** Class Global \_ Position OO Design

**Attributes**

<i>Name</i>	<i>Type</i>	<i>Private</i>	<i>Description</i>
deg	integer	Y	Degrees of angle
min	integer	Y	Minutes of angle
sec	real	Y	Seconds of angle
dir	character	Y	Compass direction

**Behavior**

<i>Name</i>	<i>Private</i>	<i>Description</i>
Default_Angle	N	Default constructor
Decimal_min	N	Constructor for decimal minutes
Decimal_sec	N	Constructor for decimal seconds
Int_deg	N	Constructor for whole deg
Int_deg_min	N	Constructor for whole deg, min
Int_deg_min_sec	N	Constructor for whole deg, min, sec
to_Decimal_Degrees	N	Convert position angle values to decimal degree
to_Radians	N	Convert position angle values to decimal radian

**Data**

<i>Name</i>	<i>Description</i>
None	

**Interfaces**

<i>Name</i>	<i>Description</i>
List_Position_Angle	Print values for position angle
Read_Position_Angle	Read values for position angle

**Table 5.9:** Class Position\_Angle OO Design**Attributes**

<i>Name</i>	<i>Type</i>	<i>Private</i>	<i>Description</i>
rad	real	Y	Radian measure of the angle (rad)

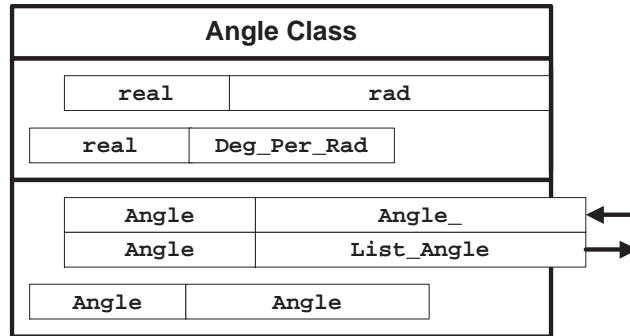
**Behavior**

<i>Name</i>	<i>Private</i>	<i>Description</i>
Angle_	N	A generic constructor
List_Angle	N	List angle value in radians and degrees

**Data**

<i>Name</i>	<i>Description</i>
Deg_per_Rad	Unit conversion parameter

**Table 5.10:** Class Angle OO Design



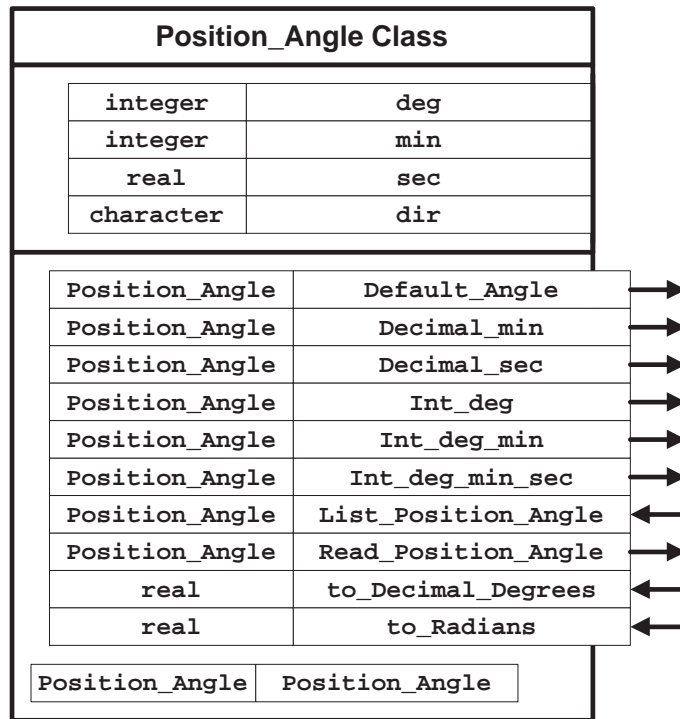
**Figure 5.5:** Graphical Representation of an Angle Class

```

[ 1] module class_Angle      ! file: class_Angle.f90
[ 2]   implicit none
[ 3]   type Angle           ! angle in (signed) radians
[ 4]     private
[ 5]       real :: rad      ! radians
[ 6]   end type
[ 7]   real, parameter:: Deg_Per_Rad = 57.2957795130823209d0
[ 8] contains
[ 9]
[10]   function Angle_ (r) result (ang) ! public constructor
[11]     real, optional :: r           ! radians
[12]     type (Angle)   :: ang
[13]     if ( present(r) ) then
[14]       ang = Angle (r)      ! intrinsic constructor
[15]     else ; ang = Angle (0.0) ! intrinsic constructor
[16]     end if ; end function Angle_
[17]
[18]   subroutine List_Angle (ang)
[19]     type (Angle), intent(in) :: ang
[20]     print *, 'Angle = ', ang % rad, ' radians (', &
[21]       Deg_Per_Rad * ang % rad, ' degrees)'
[22]   end subroutine List_Angle
[23] end module class_Angle

```

**Figure 5.6:** A Definition of the Class Angle



**Figure 5.7:** Graphical Representation of a Position Angle Class

```

[ 1] module class_Position_Angle ! file: class_Position_Angle.f90
[ 2]   use class_Angle
[ 3]   implicit none
[ 4]   type Position_Angle      ! angle in deg, min, sec
[ 5]     private
[ 6]     integer    :: deg, min  ! degrees, minutes
[ 7]     real       :: sec      ! seconds
[ 8]     character :: dir      ! N | S, E | W
[ 9]   end type
[10] contains
[11]
[12]   function Default_Angle () result (ang) ! default constructor
[13]     type (Position_Angle) :: ang
[14]     ang = Position_Angle (0, 0, 0., 'N') ! intrinsic
[15]   end function Default_Angle
[16]
[17]   function Decimal_min (d, m, news) result (ang) ! public
[18]     integer, intent(in) :: d      ! degrees
[19]     real,    intent(in) :: m      ! minutes
[20]     character, intent(in) :: news ! N | S, E | W
[21]     type (Position_Angle) :: ang ! angle out
[22]     integer    :: min            ! minutes
[23]     real       :: s              ! seconds
[24]     min = floor ( m ) ; s = (m - min)*60. ! convert
[25]     ang = Position_Angle (d, m, s, news) ! intrinsic
[26]   end function Decimal_min
[27]
[28]   function Decimal_sec (d, m, s, news) result (ang) ! public
[29]     integer, intent(in) :: d, m    ! degrees, minutes
[30]     real,    intent(in) :: s      ! seconds
[31]     character, intent(in) :: news ! N | S, E | W
[32]     type (Position_Angle) :: ang ! angle out
[33]     ang = Position_Angle (d, m, s, news) ! intrinsic
[34]   end function Decimal_sec
[35]

```

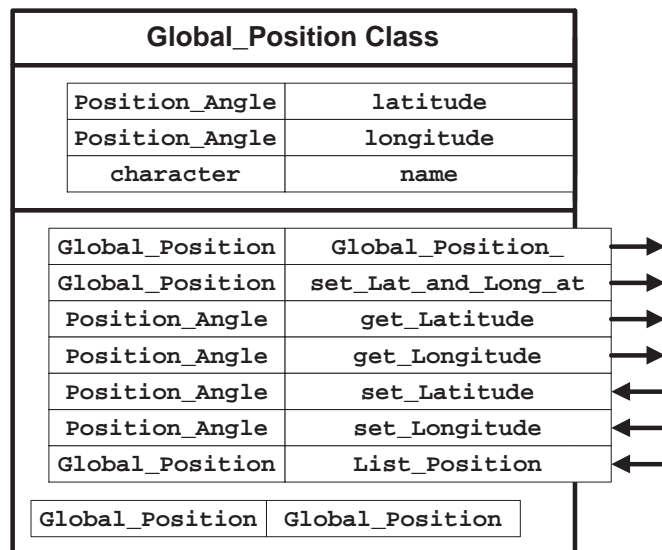
(Fig. 5.8, A Definition of the Class Position Angle (Continued))

```

[36] function Int_deg (d, news) result (ang) ! public
[37]   integer, intent(in) :: d ! degrees, minutes
[38]   character, intent(in) :: news ! N | S, E | W
[39]   type (Position_Angle) :: ang ! angle out
[40]   ang = Position_Angle (d, 0, 0.0, news) ! intrinsic
[41] end function Int_deg
[42]
[43] function Int_deg_min (d, m, news) result (ang) ! public
[44]   integer, intent(in) :: d, m ! degrees, minutes
[45]   character, intent(in) :: news ! N | S, E | W
[46]   type (Position_Angle) :: ang ! angle out
[47]   ang = Position_Angle (d, m, 0.0, news) ! intrinsic
[48] end function Int_deg_min
[49]
[50] function Int_deg_min_sec (d, m, s, news) result (ang) ! public
[51]   integer, intent(in) :: d, m, s ! deg, min, seconds
[52]   character, intent(in) :: news ! N | S, E | W
[53]   type (Position_Angle) :: ang ! angle out
[54]   ang = Position_Angle (d, m, real(s), news) ! intrinsic
[55] end function Int_deg_min_sec
[56]
[57] subroutine List_Position_Angle (a)
[58]   type (Position_Angle) :: a ! angle
[59]   print 5, a ; 5 format (i3, " ", i2, " ", f8.5, " ", a1)
[60] end subroutine
[61]
[62] subroutine Read_Position_Angle (a)
[63]   type (Position_Angle) :: a ! angle
[64]   read *, a%deg, a%min, a%sec, a%dir ; end subroutine
[65]
[66] function to_Decimal_Degrees (ang) result (degrees)
[67]   type (Position_Angle), intent(in) :: ang
[68]   real :: degrees
[69]   degrees = ang%deg + ang%min/60. + ang%sec/60.
[70]   if (ang%dir == "S" .or. ang%dir == "s" .or. &
[71]       ang%dir == "W" .or. ang%dir == "w") degrees = -degrees
[72] end function to_Decimal_Degrees
[73]
[74] function to_Radians (ang) result (radians)
[75]   type (Position_Angle), intent(in) :: ang
[76]   real :: radians
[77]   radians = (ang%deg + ang%min/60. + ang%sec/60.)/Deg_Per_Rad
[78]   if (ang%dir == "S" .or. ang%dir == "s" .or. &
[79]       ang%dir == "W" .or. ang%dir == "w") radians = -radians
[80] end function to_Radians
[81] end module class_Position_Angle

```

**Figure 5.8:** A Definition of the Class Position Angle



**Figure 5.9:** Graphical Representation of a Global Position Class

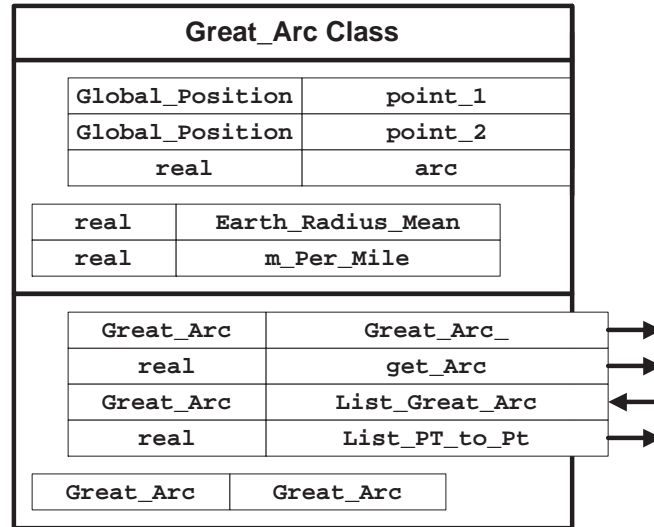
```

[ 1] module class_Global_Position
[ 2]   use class_Position_Angle
[ 3]   implicit none
[ 4]   type Global_Position
[ 5]     private
[ 6]     type (Position_Angle) :: latitude, longitude
[ 7]     character (len=31)    :: name
[ 8]   end type Global_Position
[ 9] contains
[10]
[11]   function Global_Position_ (d1, m1, s1, c1, & ! constructor
[12]                             d2, m2, s2, c2, n) result (GP)
[13]     integer, intent(in) :: d1, m1, s1 ! deg, min, sec
[14]     integer, intent(in) :: d2, m2, s2 ! deg, min, sec
[15]     character, intent(in) :: c1, c2 ! compass
[16]     character (len=*) :: n ! name
[17]     type (Global_Position) :: GP ! returned position
[18]     GP % latitude = Int_deg_min_sec (d1, m1, s1, c1)
[19]     GP % longitude = Int_deg_min_sec (d2, m2, s2, c2)
[20]     GP % name = n ; end function Global_Position_
[21]
[22]   function set_Lat_and_Long_at (lat, long, n) result (GP) ! cons
[23]     type (Position_Angle), intent(in) :: lat, long ! angles
[24]     character (len=*), intent(in) :: n ! name
[25]     type (Global_Position) :: GP ! position
[26]     GP % latitude = lat ; GP % longitude = long
[27]     GP % name = n ; end function set_Lat_and_Long_at
[28]
[29]   function get_Latitude (GP) result (lat)
[30]     type (Global_Position), intent(in) :: GP
[31]     type (Position_Angle) :: lat
[32]     lat = GP % latitude ; end function get_Latitude
[33]
[34]   function get_Longitude (GP) result (long)
[35]     type (Global_Position), intent(in) :: GP
[36]     type (Position_Angle) :: long
[37]     long = GP % longitude ; end function get_Longitude
[38]
[39]   subroutine set_Latitude (GP, lat)
[40]     type (Global_Position), intent(inout) :: GP
[41]     type (Position_Angle), intent(in) :: lat
[42]     GP % latitude = lat ; end subroutine set_Latitude
[43]
[44]   subroutine set_Longitude (GP, long)
[45]     type (Global_Position), intent(inout) :: GP
[46]     type (Position_Angle), intent(in) :: long
[47]     GP % longitude = long ; end subroutine set_Longitude
[48]
[49]   subroutine List_Position (GP)
[50]     type (Global_Position), intent(in) :: GP
[51]     print *, 'Position at ', GP % name
[52]     write (*, '( " Latitude: " )', advance = "no")
[53]     call List_Position_Angle (GP % latitude)
[54]     write (*, '( " Longitude: " )', advance = "no")
[55]     call List_Position_Angle (GP % longitude)
[56]   end subroutine List_Position
[57] end module class_Global_Position

```

**Figure 5.10:** A Definition of the Class Global Position





**Figure 5.11:** Graphical Representation of a Great Arc Class

```

[ 1] module class_Great_Arc
[ 2]   use class_Global_Position
[ 3]   implicit none
[ 4]   real, parameter :: Earth_Radius_Mean = 6.371d6 ! meters
[ 5]   real, parameter :: m_Per_Mile      = 1609.344
[ 6]   type Great_Arc
[ 7]     type (Global_Position) :: point_1, point_2
[ 8]     real :: arc
[ 9]   end type Great_Arc
[10] contains
[11]
[12]   function Great_Arc_ (GP1, GP2) result (G_A)      ! constructor
[13]     type (Global_Position), intent(in) :: GP1, GP2 ! points
[14]     type (Great_Arc) :: G_A ! earth arc
[15]     G_A = Great_Arc (GP1, GP2, get_Arc (GP1, GP2)) ! intrinsic
[16]   end function Great_Arc_
[17]
[18]   function get_Arc (GP1, GP2) result (dist)
[19]     type (Global_Position), intent(in) :: GP1, GP2
[20]     real :: dist
[21]     real :: lat1, lat2, long1, long2
[22]     ! convert latitude, longitude to radians
[23]     lat1 = to_Radians (get_Latitude (GP1))
[24]     lat2 = to_Radians (get_Latitude (GP2))
[25]     long1 = to_Radians (get_Longitude (GP1))
[26]     long2 = to_Radians (get_Longitude (GP2))
[27]     ! compute great circle arc of earth
[28]     dist = 2 * Earth_Radius_Mean &
[29]           * asin( sqrt ( (sin((lat1 - lat2)/2.))**2 &
[30]             + cos(lat1)*cos(lat2)*(sin((long1-long2)/2.))**2 ) ) &
[31]   end function get_Arc
[32]
[33]   subroutine List_Great_Arc (A_to_B)
[34]     type (Great_Arc), intent(in) :: A_to_B
[35]     real :: dist ! in meters
[36]     print * ; print *, "The great circle arc between"
[37]     call List_Position (A_to_B % point_1)
[38]     call List_Position (A_to_B % point_2)
[39]     dist = A_to_B % arc ! convert to km and miles
[40]     print *, "is ", dist/1000, " km (", dist/m_Per_Mile, "miles)."
[41]   end subroutine List_Great_Arc
[42]
[43]   subroutine List_PT_to_Pt (GP1, GP2) ! alternate
[44]     type (Global_Position), intent(in) :: GP1, GP2 ! points
[45]     real :: arc ! distance
[46]     print * ; print *, "The great circle arc between"
[47]     call List_Position (GP1) ; call List_Position (GP2)
[48]     arc = get_Arc (GP1, GP2) ! in meters
[49]     print *, "is ", arc/1000, " km (", arc/m_Per_Mile, "miles)"
[50]   end subroutine List_PT_to_Pt
[51] end module class_Great_Arc

```

**Figure 5.12:** Definition of the Class Great Arc

```

[ 1] program main
[ 2]   use class_Great_Arc
[ 3]   implicit none
[ 4]   type (Great_Arc)      :: arc
[ 5]   type (Global_Position) :: g1, g2
[ 6]   type (Position_Angle) :: a1, a2
[ 7]   type (Angle)          :: ang
[ 8]   real                  :: deg, rad
[ 9]   a1 = Decimal_sec      (10, 30, 0., "N"); call List_Position_Angle(a1
[10]   a1 = Int_deg_min_sec(10, 30, 0, "N"); call List_Position_
[11]   a1 = Int_deg_min      (10, 30, "N"); call List_Position_An
[12]   a1 = Int_deg          (20, "N"); call List_Position_Angle(a1
[13] !   call Read_Position_Angle (a2)
[14]   a2 = Decimal_sec      (30, 48, 0., "E"); call List_Position_Angle(a2
[15]   ang = Angle_ (1.0)      ; call List_Angle (ang)
[16]   deg = to_Decimal_Degrees (a1) ; print *, deg, deg/Deg_Per_Rad
[17]   rad = to_Radians (a1)      ; print *, rad
[18] !
[19]   g1 = set_Lat_and_Long_at (a1, a2, 'g1')
[20]   call List_Position (g1)
[21]   g2 = Global_Position_ (20, 5, 40, "S", 75, 0, 20, "E", 'g2')
[22]   call List_Position (g2)
[23]   print *, "Arc = ", get_Arc (g1, g2), " (meters)"
[24]   g1 = Global_Position_ ( 0, 0, 0, "N", 0, 0, 0, "E", 'equator')
[25]   g2 = Global_Position_ (90, 0, 0, "N", 0, 0, 0, "E", 'N-pole')
[26]   call List_Pt_to_Pt (g1, g2)
[27]   arc = Great_Arc_ (g1, g2) ; call List_Great_Arc (arc)
[28] end program main                                ! running gives:
[29] ! 10 30'  0.00000" N ; ! 10 30'  0.00000" N ; ! 10 30'  0.00000" N
[30] ! 20 0'   0.00000" N ; ! 30 48'  0.00000" N
[31] ! Angle =  1.0000000000 radians ( 57.29578018 degrees)
[32] ! 20.00000000 0.3490658402 ; ! 0.3490658402
[33] ! Position at g1 ; ! Position at g2
[34] ! Latitude:  20 0'  0.00000" N ; ! Latitude:  20 5' 40.00000" S
[35] ! Longitude: 30 48' 0.00000" E ; ! Longitude: 75 0' 20.00000" E
[36] ! Arc = 6633165.000 (meters)
[37] !
[38] ! The great circle arc between
[39] ! Position at equator ; ! Position at N-pole
[40] ! Latitude:  0 0'  0.00000" N ; ! Latitude:  90 0'  0.00000" N
[41] ! Longitude:  0 0'  0.00000" E ; ! Longitude:  0 0'  0.00000" E
[42] ! is 10007.54297 km ( 6218.398926 miles)
[43] !
[44] ! The great circle arc between
[45] ! Position at equator ; ! Position at N-pole
[46] ! Latitude:  0 0'  0.00000" N ; ! Latitude:  90 0'  0.00000" N
[47] ! Longitude:  0 0'  0.00000" E ; ! Longitude:  0 0'  0.00000" E
[48] ! is 10007.54297 km ( 6218.398926 miles)

```

**Figure 5.13:** Testing the Great Arc Class Interactions

## 5.4 Exercises

1. Referring to Chapter 3, develop OOA and OOD tables for the a) `Geometric` class, b) `Date` class, c) `Person` class, d) `Student` class.
2. Develop the graphical representations for the classes in the a) drill study, b) global position study.
3. Use the classes in the GPS study to develop a main program that will read a list (vector) of `Global_Position` types and use them to output a square table of great arc distances from one site to each of the others. That is, the table entry in row *j*, column *k* gives the arc from site *j* to site *k*. Such a table would be symmetric (with zeros along one diagonal) so you may want to give only half of it.
4. Modify the given `Class_Position_Angle` to provide a polymorphic interface for a constructor `Position_Angle_` that will accept decimal, integer or no data for the seconds value. Also allow for the omission of the minutes value.

## Chapter 6

# Inheritance and Polymorphism

### 6.1 Introduction

As we have seen earlier in our introduction to OOP *inheritance* is a mechanism for deriving a new class from an older *base class*. That is, the base class, sometimes called the *super class*, is supplemented or selectively altered to create the new *derived class*. Inheritance provides a powerful code reuse mechanism since a hierarchy of related classes can be created that share the same code. A class can be derived from an existing base class using the module construct illustrated in Fig. 6.1.

We note that the inheritance is invoked by the USE statement. Sometimes an inherited entity (attribute or member) needs to be slightly amended for the purposes of the new classes. Thus, at times one may want to selectively bring into the new class only certain entities from the base class. The modifier ONLY in a USE statement allows one to select the desired entities from the base class as illustrated below in Fig. 6.2. It is also common to develop name conflicts when combining entities from one or more related classes. Thus a rename modifier, =>, is also provided for a USE statement to allow the programmer to pick a new *local* name for an entity onherited from the base class. The form for that modifier is given in Fig. 6.3.

It is logical to extend any or all of the above inheritance mechanisms to produce multiple inheritance. *Multiple Inheritance* allows a derived class to be created by using inheritance from more than a single base class. While multiple inheritance may at first seem like a panacea for efficient code reuse, experience has shown that a heavy use of multiple inheritance can result in entity conflicts and be otherwise counterproductive. Nevertheless it is a useful tool in OOP. In F90 the module form for selective multiple inheritance would combine the above USE options in a single module as illustrated in Fig. 6.4.

---

```
module derived_class_name
    use base_class_name
    ! new attribute declarations, if any
    ...
contains
    ! new member definitions
    ...
end module derived_class_name
```

**Figure 6.1:** F90 Single Inheritance Form.

```

module derived_class_name
    use base_class_name, only: list_of_entities
    ! new attribute declarations, if any
    ...
contains

    ! new member definitions
    ...
end module derived_class_name

```

**Figure 6.2:** F90 Selective Single Inheritance Form.

```

module derived_class_name
    use base_class_name, local_name => base_entity_name
    ! new attribute declarations, if any
    ...
contains

    ! new member definitions
    ...
end module derived_class_name

```

**Figure 6.3:** F90 Single Inheritance Form, with Local Renaming.

```

module derived_class_name
    use base1_class_name
    use base2_class_name
    use base3_class_name, only: list_of_entities
    use base4_class_name, local_name => base_entity_name
    ! new attribute declarations, if any
    ...
contains

    ! new member definitions
    ...
end module derived_class_name

```

**Figure 6.4:** F90 Multiple Selective Inheritance with Renaming.

```

[ 1] module class_Professor      ! file: class_Professor.f90
[ 2]   implicit none
[ 3]   public :: print, name
[ 4]   private :: publs
[ 5]   type Professor
[ 6]     character (len=20) :: name
[ 7]     integer           :: publs ! publications
[ 8]   end type Professor
[ 9]   contains
[10]     function make_Professor (n, p) result (who)
[11]       character (len=*) optional, intent(in) :: n
[12]       integer, optional, intent(in) :: p
[13]       type (Professor) :: who ! out
[14]       who%name         = " " ! set defaults
[15]       who%publs        = 0.0
[16]       if ( present(n) ) who%name = n ! construct
[17]       if ( present(p) ) who%publs = p
[18]     end function make_Professor
[19]
[20]     function print (who)
[21]       type (Professor), intent(in) :: who
[22]       print *, "My name is ", who%name, &
[23]         ", and I have ", who%publs, " publications."
[24]     end function print
[25] end module class_Professor

```

**Figure 6.5:** A Professor Class

## 6.2 Example Applications of Inheritance

### 6.2.1 The Professor Class

In the introductory examples of OOP in Chapter 3 we introduced the concepts of inheritance and multiple inheritance by the use of the `Date` class, `Person` class, and `Student` class. To reinforce those concepts we will reuse those three classes and will have them be inherited by a `Professor` class. Acknowledging the common “publish or perish” aspect of academic life the professor class must keep up with the number of publications of the professor. The new class is given in Fig. 6.5 along with a small validation program in Fig. 6.6.

Note that the validation program brings in three different versions of the “print” member (lines 7-9) and renames two of them to allow a polymorphic print statement (lines 12-14) that selects the proper member based solely on the class of its argument. Observe that the previous `Date` class is brought into the main through the use of the `Person` class (in line 7). Of course, it is necessary to have an interface defined for the overloaded member name so that the compiler knows which candidate routines to search at run time. This example also serves to remind the reader that Fortran does *not* have *keywords* that are not allowed to be used by the programmer. In this case the print function (lines 19, 22, 25) has automatically replaced the intrinsic print function of Fortran. Most languages, including C++ do not allow one to do that.

### 6.2.2 The Employee and Manager Classes

Next we will begin the popular employee-manager classes as examples of common related classes that demonstrate the use of inheritance. Once again the idea behind encapsulating these data and their associated functionality is to model a pair of real world entities - an employee and a manager. As we go through possible relations between these two simple classes it becomes clear that there is no unique way to establish the classes and how they should interact. We begin with a minimal approach and then work through two alternate versions to reach the point where an experienced OO programmer might have begun. The first `Employee` class, shown in Fig. 6.7 has a name and pay rate as its attributes. Only the intrinsic constructor is used within the member `setDataE` to concatenate a first name and last name to form the complete name attribute and to accept the pay rate. To query members `getNameE` and `getRate` are provided to extract either of the desired attributes. Finally, member `payE` is provided to compute the pay earned by an employee. It assumes that an employee is paid by the hour. A simple testing main program is shown in Fig. 6.8 It simply defines two employees (`emp11` and `emp12`), assigns their names and pay rates, and then computes and displays their pay based on the respective number of hours worked.

```

[ 1] ! Multiple Inheritance and Polymorphism of the "print" function
[ 2] include 'class_Person.inc'           ! also brings in class_Date
[ 3] include 'class_Student.inc'
[ 4] include 'class_Professor.inc'
[ 5]
[ 6] program main
[ 7]   use class_Person                   ! no changes
[ 8]   use class_Student, print_S => print ! renamed to print_S
[ 9]   use class_Professor, print_F => print ! renamed to print_F
[10]   implicit none
[11]
[12] ! Interface to generic routine, print, for any type argument
[13] interface print ! using renamed type dependent functions
[14]   module procedure print_Name, print_S, print_F
[15] end interface
[16]
[17] type (Person) :: x; type (Student) :: y; type (Professor) :: z
[18]
[19]   x = Person ("Bob"); ! default constructor
[20]   call print(x);      ! print person type
[21]
[22]   y = Student ("Tom", 3.47); ! default constructor
[23]   call print(y);          ! print student type
[24]
[25]   z = Professor ("Ann", 7); ! default constructor
[26]   call print(z);          ! print professor type
[27]   ! alternate constructors not used
[28] end program main          ! Running gives:
[29] ! Bob
[30] ! My name is Tom, and my G.P.A. is 3.4700000.
[31] ! My name is Ann, and I have 7 publications.

```

**Figure 6.6:** Bringing Four Classes and Three Functions Together

```

[ 1] module class_Employee
[ 2] ! The module class_Employee contains both the
[ 3] ! data and functionality of an employee.
[ 4] !
[ 5]   implicit none
[ 6]   public :: setDataE, getNameE, payE ! the Functionality
[ 7]
[ 8]   type Employee                ! the Data
[ 9]   private
[10]     character(30) :: name
[11]     real          :: payRate ; end type Employee
[12]
[13] contains ! inherited internal variables and subprograms
[14]
[15]   function setDataE (lastName, firstName, newPayRate) result (E)
[16]     character(*), intent(in) :: lastName
[17]     character(*), intent(in) :: firstName
[18]     real,          intent(in) :: newPayRate
[19]     type (Employee) :: E                ! employee
[20]     ! use intrinsic constructor
[21]     E = Employee(trim(firstName)//" "/trim(lastName)),newPayRate)
[22]   end function setDataE
[23]
[24]   function getNameE ( Person ) result (n)
[25]     type (Employee), intent(in) :: Person
[26]     character(30)                :: n    ! name
[27]     n = Person % name ; end function getNameE
[28]
[29]   function getRate ( Person ) result ( r )
[30]     type (Employee), intent(in) :: Person
[31]     real                      :: r      ! rate
[32]     r = Person % payRate ; end function getRate
[33]
[34]   function payE ( Person, hoursWorked ) result ( amount )
[35]     type (Employee), intent(in) :: Person
[36]     real,          intent(in) :: hoursWorked
[37]     real                      :: amount
[38]     amount = Person % payRate * hoursWorked ; end function payE
[39] end module class_Employee

```

**Figure 6.7:** First Definition of an Employee Class

Note that both `empl1` and `empl2` are each an instance of a class, and therefore they are objects and thus distinctly different from a class.

```

[ 1]  program main
[ 2]  ! Example use of employees
[ 3]  use class_Employee
[ 4]  type (Employee)  empl1, empl2
[ 5]
[ 6]  ! Set up 1st employee and print out his name and pay
[ 7]  empl1 = setDataE ( "Jones", "Bill", 25.0 )
[ 8]  print *, "Name: ", getNameE ( empl1 )
[ 9]  print *, "Pay: ", payE ( empl1, 40.0 )
[10]
[11]  ! Set up 2nd employee and print out her name and pay
[12]  empl2 = setDataE ( "Doe", "Jane", 27.5 )
[13]  print *, "Name: ", getNameE ( empl2 )
[14]  print *, "Pay: ", payE ( empl2, 38.0 )
[15] end program main      ! Running produces:
[16] ! Name: Bill Jones    ! Pay:    1000.
[17] ! Name: Jane Doe     ! Pay:    1045.

```

**Figure 6.8:** First Test of an Employee Class

Next we deal with a manager which Is-A “kind of” employee. One difference is that some managers may be paid a salary rather than an hourly rate. Thus we have the `Manager` class inherit the attributes of the `Employee` class and add a new logical attribute `isSalaried` which is true when the manager is salary based. To support such a case we must add a new member `setSalaried` which can turn the new attribute on or off, and a corresponding member `payM` that uses the `isSalaried` flag when computing the pay. The `class_Manager` module is shown in Fig. 6.9 Note that the constructor `Manager_` defaults to an hourly worker (line 33) and it uses the inherited employee constructor (line 31). Figure 6.10 shows a test program to validate the manager class (and indirectly the employee class). It defines a salaried manager, `mgr1`, an hourly manager `mgr2`, and prints the name and weekly pay for both. (Verify these weekly pay amounts.)

With these two classes we have mainly used different program names for members that do similar things in each class (the author’s preference). However, many programmers prefer to use a single member name for a typical operation, regardless of the class of the operand. We also restricted all the attributes to `private` and allowed all the members to be `public`. We could use several alternate approaches to building our `Employee` and `Manager` classes. For example, assume we want a single member name called `pay` to be invoked for an employee, or manager (or executive). Furthermore we will allow the attributes to be `public` instead of `private`. Lowering the access restrictions to the attributes makes it easier to write an alternate program, but it is not a recommended procedure since it breaks the data hiding concept that has been shown to be important to OO software maintenance and reliability. The alternate `Employee` and `Manager` classes are shown in Figs. 6.11 and 6.12, respectively. Note that they both have a `pay` member but their arguments are of different classes and their internal calculations are different. Now we want a validation program that will create both classes of individuals, and use a single member name, `PrintPay`, to print the proper pay amount from the single member name `pay`. This can be done in different ways. One problem that arises in our plan to reuse the code in the two alternate class modules is that neither contained a pay printing member. We will need two new routines, `PrintPayEmployee` and `PrintPayManager`, and a generic or polymorphic interface to them. We have at least three ways to do this. One way is to place the two routines in an external file (or external to main if in the same file), leave the two class modules unchanged, and have the `main` program begin with (or `INCLUDE`) an external interface prototype. This first approach to `main` is shown in Fig. 6.13. Note that the two new external routines must each use their respective class module.

A second approach would be to have the two new routines become internal to the `main`, after line 30, and occur before `end program`. Another change would be that each routine would have to omit its `use` statement (such as lines 34 and 41). Why? Because they are now internal to `main` and it has already made use of the two classes (in line 2). That approach is shown in Figs. 6.13

A third approach would be the most logical and consistent with OOP principles. It is to make all the class attributes `private`, place the print members in each respective class, insert a single generic name interface in each class, and modify the `main` program to use the polymorphic name regardless of the class of the argument it acts upon. The improved version of the classes are given below in Figs. 6.14, 6.15, and 6.16. Observe that generic interfaces for `PrintPay` and `getName` have been added, but that we could



```

[ 1] module class_Manager
[ 2] ! Gets class_Employee and add additional functionality
[ 3] use class_Employee
[ 4] implicit none
[ 5] public :: setSalaried, payM
[ 6]
[ 7] type Manager          ! the Data
[ 8] private
[ 9]   type (Employee) :: Person
[10]   integer          :: isSalaried ! ( or logical )
[11] end type Manager
[12]
[13] contains ! inherited internal variables and subprograms
[14]
[15] function getEmployee ( M ) result (E)
[16]   type (Manager ), intent(in) :: M
[17]   type (Employee)           :: E
[18]   E = M % Person ; end function getEmployee
[19]
[20] function getNameM ( M ) result (n)
[21]   type (Manager ), intent(in) :: M
[22]   type (Employee)           :: E
[23]   character(30)             :: n ! name
[24]   n = getNameE(M % Person); end function getNameM
[25]
[26] function Manager_ (lastName, firstName, newPayRate) result (M)
[27]   character(*), intent(in) :: lastName
[28]   character(*), intent(in) :: firstName
[29]   real,          intent(in) :: newPayRate
[30]   type (Employee)           :: E ! employee
[31]   type (Manager )          :: M ! manager constructor
[32]   E = setDataE (lastName, firstName, newPayRate)
[33]   ! use intrinsic constructor
[34]   M = Manager(E, 0) ! default to no salary
[35] end function Manager_
[36]
[37] function setDataM (lastName, firstName, newPayRate) result (M)
[38]   character(*), intent(in) :: lastName
[39]   character(*), intent(in) :: firstName
[40]   real,          intent(in) :: newPayRate
[41]   type (Employee)           :: E ! employee
[42]   type (Manager )          :: M ! manager
[43]   E = setDataE (lastName, firstName, newPayRate)
[44]   M % Person = E
[45] end function setDataM
[46]
[47] subroutine setSalaried ( Who, salariedFlag )
[48]   type (Manager), intent(inout) :: Who
[49]   integer,          intent(in)   :: salariedFlag
[50]   Who % isSalaried = salariedFlag ; end subroutine setSalaried
[51]
[52] function payM ( Human, hoursWorked ) result ( amount )
[53]   type (Manager), intent(in) :: Human
[54]   real,          intent(in) :: hoursWorked
[55]   real,          :: amount, value
[56]   value = getRate( getEmployee(Human) )
[57]   if ( Human % isSalaried == 1 ) then ! (or use logical)
[58]     amount = value
[59]   else
[60]     amount = value * hoursWorked
[61]   end if ; end function payM
[62] end module class_Manager

```

**Figure 6.9:** A First Declaration of a Manager Class

not do that for a corresponding setData; do you know why? A final improvement will be given as an assignment.

## 6.3 Polymorphism

Fortran 90 and 95 do not include the full range of polymorphism abilities that one would like to have in an object-oriented language. It is expected that the Fortran 2000 standard will add those abilities.

Some of the code “re-use” features can be constructed through the concept of subprogram “templates,” which will be discussed below. The lack of a standard “Is\_A” polymorphism can be overcome in F90/95 by the use of the SELECT CASE feature to define “sub-types” of objects. This approach of sub-typing programming provides the desired additional functionality, but it is clearly not as easy to change or extend as an inheritance feature built into the language standard. A short example will be provided.

```

[ 1] program main ! Example use of managers
[ 2]   use class_Manager
[ 3]   implicit none
[ 4]   type (Manager) mgr1, mgr2
[ 5]
[ 6]   ! Set up 1st manager and print out her name and pay
[ 7]
[ 8]   mgr1 = setDataM ( "Smith", "Kimberly", 1900.0 )
[ 9]   call setSalaried ( mgr1, 1 ) ! Has a salary
[10]
[11]   print *, "Name: ", getNameM ( mgr1)
[12]   print *, "Pay: ", payM ( mgr1, 40.0 )
[13]
[14]   ! Set up 2nd manager and print out his name and pay
[15]
[16]   ! mgr2 = setDataM ( "Danish", "Tom", 46.5 )
[17]   ! call setSalaried ( mgr2, 0 ) ! Doesn't have a salary
[18]   ! or
[19]   mgr2 = Manager_ ( "Danish", "Tom", 46.5 )
[20]
[21]   print *, "Name: ", getNameM ( mgr2)
[22]   print *, "Pay: ", payM ( mgr2, 40.0 )
[23] end program main ! Running produces:
[24] ! Name: Kimberly Smith ! Pay: 1900.
[25] ! Name: Tom Danish ! Pay: 1860.

```

**Figure 6.10:** First Test of a Manager Class

```

[ 1] module class_Employee ! Alternate
[ 2]   implicit none
[ 3]   public :: setData, getName, pay ! the Functionality
[ 4]
[ 5]   type Employee ! the Data
[ 6]     character(30) :: name
[ 7]     real :: payRate
[ 8]   end type Employee
[ 9]
[10] contains ! inherited internal variables and subprograms
[11]
[12]   subroutine setData ( Person, lastName, firstName, newPayRate )
[13]     type (Employee) :: Person
[14]     character(*) :: lastName
[15]     character(*) :: firstName
[16]     real :: newPayRate
[17]     Person % name = trim (firstName) // " " // trim (lastName)
[18]     Person % payRate = newPayRate
[19]   end subroutine setData
[20]
[21]   function getName ( Person )
[22]     character(30) :: getName
[23]     type (Employee) :: Person
[24]     getName = Person % name
[25]   end function getName
[26]
[27]   function pay ( Person, hoursWorked )
[28]     real :: pay
[29]     type (Employee) :: Person
[30]     real :: hoursWorked
[31]     pay = Person % payRate * hoursWorked
[32]   end function pay
[33] end module class_Employee

```

**Figure 6.11:** Alternate Public Access Form of an Employee Class

### 6.3.1 Templates

One of our goals has been to develop software that can be reused for other applications. There are some algorithms that are effectively independent of the object type on which they operate. For example, in a sorting algorithm one often needs to interchange, or swap, two objects. A short routine for that purpose follows:

```

subroutine swap_integers (x, y)
  implicit none
  integer, intent(inout) :: x, y
  integer :: temp
  temp = x
  x = y
  y = temp
end subroutine swap_integers

```

```

[ 1] module class_Manager ! Alternate
[ 2]   use class_Employee, payEmployee => pay ! renamed
[ 3]   implicit none
[ 4]   public :: setSalaried, payManager
[ 5]
[ 6]   type Manager ! the Data
[ 7]     type (Employee) :: Person
[ 8]     integer :: isSalaried ! ( or logical )
[ 9]   end type Manager
[10]
[11] contains ! inherited internal variables and subprograms
[12]
[13]   subroutine setSalaried ( Who, salariedFlag )
[14]     type (Manager) :: Who
[15]     integer :: salariedFlag
[16]     Who % isSalaried = salariedFlag
[17]   end subroutine setSalaried
[18]
[19]   function pay ( Human, hoursWorked )
[20]     real :: pay
[21]     type (Manager) :: Human
[22]     real :: hoursWorked
[23]
[24]     if ( Human % isSalaried == 1 ) then ! (or use logical)
[25]       pay = Human % Person % payRate
[26]     else
[27]       pay = Human % Person % payRate * hoursWorked
[28]     end if
[29]   end function pay
[30] end module class_Manager

```

**Figure 6.12:** Alternate Public Access Form of a Manager Class

Observe that in this form it appears necessary to have one version for integer arguments and another for real arguments. Indeed we might need a different version of the routine for each type of argument that you may need to swap. A slightly different approach would be to write our swap algorithm as:

```

subroutine swap_objects (x, y)
  implicit none
  type (Object), intent(inout) :: x, y
  type (Object) :: temp
  temp = x
  x = y
  y = temp
end subroutine swap_objects

```

which would be a single routine that would work for any Object, but it has the disadvantage that one find a way to redefine the Object type for each application of the routine. That would not be an easy task. (While we will continue with this example with the algorithm in the above forms it should be noted that the above approaches would not be efficient if *x* and *y* were very large arrays or derived type objects. In that case we would modify the algorithm slightly to employ pointers to the large data items and simply swap the pointers for a significant increase in efficiency.)

Consider ways that we might be able to generalize the above routines so that they could accept and swap any specific type of arguments. For example, the first two versions could be re-written in a so called template form as:

```

subroutine swap_Template$ (x, y)
  implicit none
  Template$, intent(inout) :: x, y
  Template$ :: temp
  temp = x
  x = y
  y = temp
end subroutine swap_Template$

```

In the above template the dollar sign (\$) was included in the “wild card” because while it is a valid member of the F90 character set it is not a valid character for inclusion in the name of a variable, derived type, function, module, or subroutine. In other words, a template in the illustrated form would not compile, but such a name could serve as a reminder that its purpose is to produce a code that can be compiled after the “wild card” substitutions have been made.

With this type of template it would be very easy to use a modern text editor to do a global substitution of any one of the intrinsic types character, complex, double precision, integer, logical, or real for the “wild card” keyword Template\$ to produce a source code to swap any or all of

```

[ 1] program main ! Alternate employee and manager classes
[ 2]   use class_Manager ! and thus Employee
[ 3]   implicit none
[ 4]   ! supply interface for external code not in classes
[ 5]   interface PrintPay ! For TYPE dependent arguments
[ 6]     subroutine PrintPayManager ( Human, hoursWorked )
[ 7]       use class_Manager
[ 8]       type (Manager) :: Human
[ 9]       real            :: hoursWorked
[10]     end subroutine
[11]     subroutine PrintPayEmployee ( Person, hoursWorked )
[12]       use class_Employee
[13]       type (Employee) :: Person
[14]       real            :: hoursWorked
[15]     end subroutine
[16]   end interface
[17]
[18]   type (Employee) empl ; type (Manager) mgr
[19]
[20]   ! Set up an employee and print out his name and pay
[21]   call setData ( empl, "Burke", "John", 25.0 )
[22]
[23]   print *, "Name: ", getName ( empl )
[24]   call PrintPay ( empl, 40.0 )
[25]
[26]   ! Set up a manager and print out her name and pay
[27]   call setData ( mgr % Person, "Kovacs", "Jan", 1200.0 )
[28]   call setSalaried ( mgr, 1 ) ! Has a salary
[29]
[30]   print *, "Name: ", getName ( mgr % Person )
[31]   call PrintPay ( mgr, 40.0 )
[32] end program
[33]
[34] subroutine PrintPayEmployee ( Person, hoursWorked )
[35]   use class_Employee
[36]   type (Employee) :: Person
[37]   real            :: hoursWorked
[38]   print *, "Pay: ", pay ( Person, hoursWorked )
[39] end subroutine
[40]
[41] subroutine PrintPayManager ( Human, hoursWorked )
[42]   use class_Manager
[43]   type (Manager) :: Human
[44]   real            :: hoursWorked
[45]   print *, "Pay: ", pay ( Human, hoursWorked )
[46] end subroutine
[47] ! Running produces;
[48] ! Name: John Burke
[49] ! Pay: 1000.
[50] ! Name: Jan Kovacs
[51] ! Pay: 1200.

```

**Figure 6.13:** Testing the Alternate Employee and Manager Classes

the intrinsic data types. There would be no need to keep up with all the different routine names if we placed all of them in a single module and also created a generic interface to them such as:

```

module swap_library
  implicit none
  interface swap ! the generic name
    module procedure swap_character, swap_complex
    module procedure swap_double_precision, swap_integer
    module procedure swap_logical, swap_real
  end interface
contains
  subroutine swap_characters (x, y)
    . . .
  end subroutine swap_characters
  subroutine swap_ . . .
  . . .
end module swap_library

```

The use of a text editor to make such substitutions is not very elegant and we expect that there may be a better way to pursue the concept of developing a reusable software template. The concept of a text editor substitution also fails when we go to the next logical step and try to use a derived type argument instead of any of the intrinsic data types. For example, if we were to replace the “wild card” with our previous type (chemical\_element) that would create:

```

subroutine swap_type (chemical_element) (x,y)
  implicit none

```

```

[ 1] module class_Employee                ! the base class
[ 2] implicit none                      ! strong typing
[ 3] private :: PrintPayEmployee, payE ! private members
[ 4] type Employee                      ! the Data
[ 5]   private                          ! all attributes private
[ 6]   character(30) :: name
[ 7]   real          :: payRate ; end type Employee
[ 8]
[ 9]   interface PrintPay                ! a polymorphic member
[10]     module procedure PrintPayEmployee ; end interface
[11]   interface getName                ! a polymorphic member
[12]     module procedure getNameE      ; end interface
[13] ! NOTE: can not have polymorphic setData. Why ?
[14]
[15] contains ! inherited internal variables and subprograms
[16]
[17]   function setDataE (lastName, firstName, newPayRate) result (E)
[18]     character(*), intent(in) :: lastName
[19]     character(*), intent(in) :: firstName
[20]     real,          intent(in) :: newPayRate ! amount per period
[21]     type (Employee) :: E ! employee
[22]     ! use intrinsic constructor
[23]     E = Employee(trim(firstName)//" " //trim(lastName)),newPayRate)
[24]   end function setDataE
[25]
[26]   function getNameE ( Person ) result (n)
[27]     type (Employee), intent(in) :: Person
[28]     character(30)                :: n ! name
[29]     n = Person % name ; end function getNameE
[30]
[31]   function getRate ( Person ) result ( r )
[32]     type (Employee), intent(in) :: Person
[33]     real                        :: r ! rate of pay
[34]     r = Person % payRate ; end function getRate
[35]
[36]   function payE ( Person, hoursWorked ) result ( amount )
[37]     type (Employee), intent(in) :: Person
[38]     real,          intent(in) :: hoursWorked
[39]     real                        :: amount
[40]     amount = Person % payRate * hoursWorked ; end function payE
[41]
[42]   subroutine PrintPayEmployee ( Person, hoursWorked )
[43]     type (Employee) :: Person
[44]     real            :: hoursWorked
[45]     print *, "Pay: ", payE ( Person, hoursWorked )
[46]   end subroutine
[47] end module class_Employee

```

**Figure 6.14:** A Better Private Access Form of an Employee Class

```

[ 1] module class_Manager                ! the derived class
[ 2] ! Get class_Employee, add additional attribute & members
[ 3] use class_Employee                  ! inherited base class
[ 4] implicit none                      ! strong typing
[ 5] private :: PrintPayManager, payM, getNameM ! private members
[ 6]
[ 7] type Manager                        ! the Data
[ 8]   private                          ! all attributes private
[ 9]   type (Employee) :: Person
[10]   integer          :: isSalaried ! 1 if true (or use logical)
[11] end type Manager
[12]
[13]   interface PrintPay                ! a polymorphic member
[14]     module procedure PrintPayManager ; end interface
[15]   interface getName                ! a polymorphic member
[16]     module procedure getNameM      ; end interface

```

*Fig. 6.15: A Better Private Access Form of a Manager Class (continued)*

```

      type (chemical_element), intent (inout)::x,y
      type (chemical_element)                ::temp
      temp = x
      x    = y
      y    = temp
end subroutine swap_type (chemical_element)

```

This would fail to compile because it violates the syntax for a valid function or subroutine name, as well as the end function or end subroutine syntax. Except for the first and last line syntax errors this would be a valid code. To correct the problem we simply need to add a little logic and omit the characters type

```

[17] contains ! inherited internal variables and subprograms
[18]
[19]
[20] function getEmployee ( M ) result (E)
[21]   type (Manager ), intent(in) :: M
[22]   type (Employee)           :: E
[23]   E = M % Person ; end function getEmployee
[24]
[25] function getNameM ( M ) result (n)
[26]   type (Manager ), intent(in) :: M
[27]   type (Employee)           :: E
[28]   character(30)             :: n          ! name
[29]   n = getNameE(M % Person); end function getNameM
[30]
[31] function Manager_ (lastName, firstName, newPayRate) result (M)
[32]   character(*), intent(in) :: lastName
[33]   character(*), intent(in) :: firstName
[34]   real,          intent(in) :: newPayRate
[35]   type (Employee)           :: E          ! employee
[36]   type (Manager )          :: M          ! manager constructed
[37]   E = setDataE (lastName, firstName, newPayRate)
[38]                                     ! use intrinsic constructor
[39]   M = Manager(E, 0)                 ! default to hourly
[40] end function Manager_
[41]
[42] function setDataM (lastName, firstName, newPayRate) result (M)
[43]   character(*), intent(in) :: lastName
[44]   character(*), intent(in) :: firstName
[45]   real,          intent(in) :: newPayRate ! hourly OR weekly
[46]   type (Employee)           :: E          ! employee
[47]   type (Manager )          :: M          ! manager constructed
[48]   E = setDataE (lastName, firstName, newPayRate)
[49]   M % Person = E ; M % isSalaried = 0 ! default to hourly
[50] end function setDataM
[51]
[52] subroutine setSalaried ( Who, salariedFlag ) ! 0=hourly, 1=weekly
[53]   type (Manager), intent(inout) :: Who
[54]   integer,          intent(in)   :: salariedFlag ! 0 OR 1
[55]   Who % isSalaried = salariedFlag ; end subroutine setSalaried
[56]
[57] function payM ( Human, hoursWorked ) result ( amount )
[58]   type (Manager), intent(in) :: Human
[59]   real,          intent(in) :: hoursWorked
[60]   real,          :: amount, value
[61]   value = getRate( getEmployee(Human) )
[62]   if ( Human % isSalaried == 1 ) then
[63]     amount = value ! for weekly person
[64]   else
[65]     amount = value * hoursWorked ! for hourly person
[66]   end if ; end function payM
[67]
[68] subroutine PrintPayManager ( Human, hoursWorked )
[69]   type (Manager) :: Human
[70]   real           :: hoursWorked
[71]   print *, "Pay: ", payM ( Human , hoursworked )
[72] end subroutine
[73] end module class_Manager

```

**Figure 6.15:** A Better Private Access Form of a Manager Class

( ) when we create a function, module, or subroutine name that is based on a derived type data entity. Then we obtain

```

subroutine swap_chemical_element (x,y)
  implicit none
  type (chemical_element), intent (inout)::x,y
  type (chemical_element)           ::temp
  temp = x
  x = y
  y = temp
end subroutine swap_chemical_element

```

which yields a completely valid routine.

Unfortunately, text editors do not offer us such logic capabilities. However, as we have seen, high level programming languages like C++ and F90 do have those abilities. At this point you should be able to envision writing a pre-processor program that would accept a file of template routines, replace the template “wildcard” words with the desired generic forms to produce a module or header file containing the expanded source files that can then be brought into the desired program with an include or use statement. The C++ language includes a template pre-processor to expand template files as needed.

```

[ 1] program main ! Final employee and manager classes
[ 2]   use class_Manager ! and thus class_Employee
[ 3]   implicit none
[ 4]
[ 5]   type (Employee) empl ; type (Manager) mgr
[ 6]
[ 7]   ! Set up a hourly employee and print out his name and pay
[ 8]   empl = setDataE ( "Burke", "John", 25.0 )
[ 9]
[10]   print *, "Name: ", getName ( empl )
[11]   call PrintPay ( empl, 40.0 ) ! polymorphic
[12]
[13]   ! Set up a weekly manager and print out her name and pay
[14]   mgr = setDataM ( "Kovacs", "Jan", 1200.0 )
[15]   call setSalaried ( mgr, 1 ) ! rate is weekly
[16]
[17]   print *, "Name: ", getName ( mgr )
[18]   call PrintPay ( mgr, 40.0 ) ! polymorphic
[19] end program ! Running produces;
[20] ! Name: John Burke
[21] ! Pay: 1000.
[22] ! Name: Jan Kovacs
[23] ! Pay: 1200.

```

**Figure 6.16:** Testing the Better Employee-Manager Forms

Some programmers criticize F90/95 for not offering this ability as part of the standard. A few C++ programmers criticize templates and advise against their use. Regardless of the merits of including template pre-processors in a language standard it should be clear that it is desirable to plan software for its efficient reuse.

With F90 if one wants to take advantage of the concepts of templates then the only choices are to carry out a little text editing or develop a pre-processor with the outlined capabilities. The former is clearly the simplest and for many projects may take less time than developing such a template pre-processor. However, if one makes the time investment to produce a template pre-processor one would have a tool that could be applied to basically any coding project.

### 6.3.2 Subtyping Objects (Dynamic Dispatching)

One polymorphic feature missing from the Fortran 90 standard (but expected in Fortran 2000) that is common to most object oriented languages is called run-time polymorphism or dynamic dispatching. In the C++ language this ability is introduced in the so-called “virtual function.” To emulate this ability is quite straightforward in F90 but is not elegant since it usually requires a group of if-elseif statements or other selection processes. It is only tedious if the inheritance hierarchy contains many unmodified subroutines and functions. The importance of the lack of a standardized dynamic dispatching depends on the problem domain to which it must be applied. For several applications demonstrated in the literature the alternate use of subtyping has worked quite well and resulted in programs that have been shown to run several times faster than equivalent C++ versions.

We implement dynamic dispatching in F90 by a process often called subtyping. Two features must be constructed to do this. First, a pointer object, which can point to any subtype member in an inheritance hierarchy, must be developed. Second, an if-elseif or other selection method is developed to serve as a dispatch mechanism to select the unique appropriate procedure to be executed based on the actual class referenced in the controlling pointer object. This subtyping process is also referred to as implementing a polymorphic class. Of course, the details of the actual dispatching process can be hidden from the procedures that utilize the polymorphic class.

This process will be illustrated by creating a specific polymorphic class, called `Is_A_Member_Class`, which has polymorphic procedures named `new`, `assign`, and `display`. They will construct a new instance of the object, assign it a value, and list its components. The minimum example of such a process requires two members and is easily extended to any number of member classes. We begin by defining each of the subtype classes of interest.

The first is a class, `Member_1_Class`, which has two real components and the encapsulated functionality to construct a new instance and another to accept a pointer to such a subtype and display related information. It is shown in Fig. 6.17. The next class, `Member_2_Class`, has three components: two reals and one of type `Member_1`. It has the same sort of functionality, but clearly must act on more

```

[ 1] Module Member_1_Class
[ 2]   implicit none
[ 3]   type member_1
[ 4]     real :: real_1, real_2
[ 5]   end type member_1
[ 6]
[ 7]   contains
[ 8]
[ 9]     subroutine new_member_1 (member, a, b)
[10]       real, intent(in) :: a, b
[11]       type (member_1) :: member
[12]       member%real_1 = a ; member%real_2 = b
[13]     end subroutine new_member_1
[14]
[15]     subroutine display_memb_1 (pt_to_memb_1, c)
[16]       type (member_1), pointer :: pt_to_memb_1
[17]       character(len=1), intent(in) :: c
[18]       print *, 'display_memb_1 ', c
[19]     end subroutine display_memb_1
[20]
[21] End Module Member_1_Class

```

**Figure 6.17: Defining Subtype\_1**

```

[ 1] Module Member_2_Class
[ 2]   Use Member_1_Class
[ 3]   implicit none
[ 4]   type member_2
[ 5]     type (member_1) :: r_1_2
[ 6]     real :: real_3, real_4
[ 7]   end type member_2
[ 8]
[ 9]   contains
[10]
[11]     subroutine new_member_2 (member, a, b, c, d)
[12]       real, intent(in) :: a, b, c, d
[13]       type (member_2) :: member
[14]       call new_member_1 (member%r_1_2, a, b)
[15]       member%real_3 = c ; member%real_4 = d
[16]     end subroutine new_member_2
[17]
[18]     subroutine display_memb_2 (pt_to_memb_2, c)
[19]       type (member_2), pointer :: pt_to_memb_2
[20]       character(len=1), intent(in) :: c
[21]       print *, 'display_memb_2 ', c
[22]     end subroutine display_memb_2
[23]
[24] End Module Member_2_Class

```

**Figure 6.18: Defining Subtype\_2**

components. It has also inherited the functionality from the Member\_1\_Class; as displayed in Fig. 6.18.

The polymorphic class is called the Is\_A\_Member\_Class and is shown in Fig. 6.19. It includes all of the encapsulated data and function's of the above two subtypes by including their use statements. The necessary pointer object is defined as an Is\_A\_Member type that has a unique pointer for each subtype member (two in this case). It also defines a polymorphic interface to each of the common procedures to be applied to the various subtype objects. In the polymorphic function assignment the dispatching is done very simply. First, all pointers to the family of subtypes are nullified, and then the unique pointer component to the subtype of interest is set to point to the desired member. The dispatching process for the display procedure is different. It requires an if-elseif construct that contains calls to all of the possible subtype members (two here) and a failsafe default state to abort the process or undertake the necessary exception handling. Since all but one of the subtype pointer objects have been nullified it employs the associated intrinsic function to select the one, and only, procedure to call and passes the pointer object on to that procedure. The validation of this dynamic dispatching through a polymorphic class is shown in Fig. 6.20. There a target is declared for each possible subtype and then each of them is constructed and sent on to the other polymorphic functions. The results clearly show that different display procedures were used depending on the class of object supplied as an argument. It is expected that the new Fortran 2000 standard will allow such dynamic dispatching in a much simpler fashion.



```

[ 1] Module Is_A_Member_Class
[ 2] Use Member_1_Class ; Use Member_2_Class
[ 3]   implicit none
[ 4]
[ 5]   type Is_A_Member
[ 6]     private
[ 7]       type (member_1), pointer :: pt_to_memb_1
[ 8]       type (member_2), pointer :: pt_to_memb_2
[ 9]   end type Is_A_Member
[10]
[11]   interface new
[12]     module procedure new_member_1
[13]     module procedure new_member_2
[14]   end interface
[15]
[16]   interface assign
[17]     module procedure assign_memb_1
[18]     module procedure assign_memb_2
[19]   end interface
[20]
[21]   interface display
[22]     module procedure display_memb_1
[23]     module procedure display_memb_2
[24]   end interface
[25]
[26] contains
[27]
[28]   subroutine assign_memb_1 (Family, member)
[29]     type (member_1), target, intent(in) :: member
[30]     type (Is_A_Member),          intent(out) :: Family
[31]     call nullify_Is_A_Member (Family)
[32]     Family%pt_to_memb_1 => member
[33]   end subroutine assign_memb_1
[34]
[35]   subroutine assign_memb_2 (Family, member)
[36]     type (member_2), target, intent(in) :: member
[37]     type (Is_A_Member),          intent(out) :: Family
[38]     call nullify_Is_A_Member (Family)
[39]     Family%pt_to_memb_2 => member
[40]   end subroutine assign_memb_2
[41]
[42]   subroutine nullify_Is_A_Member (Family)
[43]     type (Is_A_Member), intent(inout) :: Family
[44]     nullify (Family%pt_to_memb_1)
[45]     nullify (Family%pt_to_memb_2)
[46]   end subroutine nullify_Is_A_Member
[47]
[48]   subroutine display_members (A_Member, c)
[49]     type (Is_A_Member), intent(in) :: A_Member
[50]     character(len=1),   intent(in) :: c
[51]
[52]     ! select the proper member
[53]     if ( associated (A_Member%pt_to_memb_1) ) then
[54]       call display (A_Member%pt_to_memb_1, c)
[55]     else if ( associated (A_Member%pt_to_memb_2) ) then
[56]       call display (A_Member%pt_to_memb_2, c)
[57]     else ! default case
[58]       stop 'Error, no member defined in Is_A_Member_Class'
[59]     end if
[60]   end subroutine display_members
[61] End Module Is_A_Member_Class

```

**Figure 6.19:** Combining Subtypes in an Is\_A Class

```

[ 1] program main
[ 2]   use Is_A_Member_Class
[ 3]   implicit none
[ 4]
[ 5]   type (Is_A_Member)      :: generic_member
[ 6]   type (member_1), target :: pt_to_memb_1
[ 7]   type (member_2), target :: pt_to_memb_2
[ 8]   character(len=1) :: c
[ 9]
[10]   c = 'A'
[11]   call new (pt_to_memb_1, 1.0, 2.0)
[12]   call assign (generic_member, pt_to_memb_1)
[13]   call display_members (generic_member, c)
[14]
[15]   c = 'B'
[16]   call new (pt_to_memb_2, 1.0, 2.0, 3.0, 4.0)
[17]   call assign (generic_member, pt_to_memb_2)
[18]   call display_members (generic_member, c)
[19]
[20] end program main
[21] ! running gives
[22] ! display_memb_1 A
[23] ! display_memb_2 B

```

**Figure 6.20:** Testing the Is\_A Subtypes

## 6.4 Exercises

1. Write a main program that will use the Class\_X and Class\_Y, given below, to invoke each of the f(v) routines and assign a value of 66 to the integer component in X, and 44 to the integer component in Y. (Solution given.)

```

module class_X
  public :: f
  type X_; integer a; end type X_
contains ! functionality
  subroutine f(v); type (X_), intent(in) :: v
    print *, "X_ f() executing"; end subroutine
end module class_X

module class_Y
  use class_X, X_f => f ! renamed
  public :: f
  type Y_; integer a; end type Y_ ! dominates X_ a
contains ! functionality, overrides X_ f()
  subroutine f(v); type (Y_), intent(in) :: v
    print *, "Y_ f() executing"; end subroutine
end module class_Y

```

2. Create the generic interface that would allow a single constructor name, Position\_Angle\_, to be used for all the constructors given in the previous chapter for the class Position\_Angle. Note that this is possible because they all had unique argument signatures. Also provide a new main program to test this polymorphic version.

3. Modify the last Manager class by deleting the member setDataM and replace its appearance in the last main with an existing constructor (but not used) in that class. Also provide a generic setData interface in the class Employee as a nicer name and to allow for other employees, like executives, that may have different kinds of attributes that may need to be set in the future. Explain why we could not use setDataM in the generic setData.

4. The final member setDataE in Employee is actually a constructor and the name is misleading since it does not just set data values, it also builds the name. Rename setDataE to the constructor notation Employee\_ and provide a new member in Employee called setRateE that only sets the employee pay rate.



## Chapter 7

# OO Data Structures

### 7.1 Data Structures

We have seen that F90 has a very strong intrinsic base for supporting the use of subscripted arrays. Fortran arrays can contain intrinsic data types as well as user defined types (i.e., ADT's). One can not directly have an array of pointers but you are allowed to have an array contain defined types that are pointers or that have components that are pointers. Arrays offer an efficient way to contain information and to insert and extract information. However, there are many times when creating an efficient algorithm dictates that we use some specialized storage method, or *container*, and a set of operations to act with that storage mode. The storage representation and the set of operations that are allowed for it are known as a *data structure*. How you store and retrieve an item from a container is often independent of the nature of the item itself. Thus, different instances of a data structure may produce containers for different types of objects. Data structures have the potential for a large amount of code reuse, which is a basic goal of OOP methods. In the following sections we will consider some of the more commonly used containers.

### 7.2 Stacks

A stack is a data structure where access is restricted to the last inserted object. It is referred to as a *last-in first-out* (LIFO) container. In other words, a stack is a container to which elements may only be inserted or removed at one end of the container, called the *top* of the stack. It behaves much like a pile of dinner plates. You can place a new element on the pile (widely known as a *push*), remove the top element from the pile (widely known as a *pop*), and identify the element on the top of the pile. You can also have the general concept of an empty pile, and possibly a full pile if it is associated with some type of restrictive container. Since at this point we only know about using arrays as containers we will construct a stack container by using an array.

Assume that we have defined the attributes of the "Object" that is to use our container by building a module called `object_type`. Then we could declare the array implementation of a stack type to be:

```
module stack_type
  use object_type ! to define objects in the stack
  implicit none

  integer, parameter :: limit = 999 ! stack size limit

  type stack
  private
    integer      :: size      ! size of array
    integer      :: top       ! top of stack
    type (Object) :: a(limit) ! stack items array
  end type stack
end module stack_type
```

The interface contract to develop one such stack support system (or ADT) is given as:

```

module stack_of_objects
implicit none
  public :: stack, push_on_Stack, pop_from_Stack, &
           is_Stack_Empty, is_Stack_Full

interface ! for a class_Stack contract

  function make_Stack (n) result (s) ! constructor
    use stack_type ! to define stack structure
    integer, optional :: n ! size of stack
    type (stack) :: s ! the new stack
  end function make_Stack

  subroutine push_on_Stack (s, item) ! push item on top of stack
    use stack_type ! for stack structure
    type (stack), intent(inout) :: s
    type (Object), intent(in) :: item
  end subroutine push_on_Stack

  function pop_from_Stack (s) result (item) ! pop item from top
    use stack_type ! for stack structure
    type (stack), intent(inout) :: s
    type (Object) :: item
  end function pop_from_Stack

  function is_Stack_Empty (s) result (b) ! test stack
    use stack_type ! for stack structure
    type (stack), intent(in) :: s
    logical :: b
  end function is_Stack_Empty

  function is_Stack_Full (s) result (b) ! test stack
    use stack_type ! for stack structure
    type (stack), intent(in) :: s
    logical :: b
  end function is_Stack_Full

end interface
end module stack_of_objects

```

In the interface we see that some of the member services (`is_Stack_Empty` and `is_Stack_Full`) are independent of the contained objects. Others (`pop_from_Stack` and `push_on_Stack`) explicitly depend on the `Object` utilizing the container. Of course, the constructor (here `make_Stack`) always indirectly relates to the `Object` being contained in the array. The full details of a `Stack` class are given in Fig. 7.1.

For a specific implementation test we will simply utilize objects that have a single integer attribute. That is, we define the object of interest by a code segment like:

```

module object_type
  type Object
    integer :: data ; end type ! one integer attribute
end module object_type

```

Obviously, there are many other types of objects that one may want to create and place in a container like a stack. At the present one would have to edit the above segment to define all the attributes of the object. (Begin to think about how you might seek to automate such a process.) The new `Stack` class is tested in Fig. 7.2, while a history of the example stack is sketched in Fig. 7.3. The only part of that code that depends on a specific object is in line 7 where the (public) intrinsic constructor, `Object`, was utilized rather than some more general constructor, say `Object_`.

In Fig. 7.1 note that we have used an alternate syntax and specified the type of function result (logical, `Object`, or `stack`) as a prefix to the function name (lines 16, 28, 36, 40). The author thinks that the form used in the interface contract is easier to read and understand since it requires an extra line of code, however some programmers prefer the condensed style of Fig. 7.1. Later we will examine an alternate implementation of a stack by using a linked list.

The stack implementation shown here is not complete. For example, some programmers like to include a member, say `show_Stack_top`, to display the top element on the container without removing it from the stack. Also we need to be concerned about *pre-conditions* that need to be satisfied for a member and may require that we throw an exception message. You can not pop an item off of an empty stack, nor can you push an item onto the top of a full stack. Only the member `pop_from_Stack` does such pre-condition checking in the sample code. Note that members `is_Stack_Empty` and `is_Stack_Full`

are called *accessors*, as would be `show_Stack_top`, since they query the container but do not change it.

```
[ 1] module class_Stack
[ 2]   implicit none
[ 3]   use exceptions ! to warn of errors
[ 4]   use object_type
[ 5]   public :: stack, push_on_Stack, pop_from_Stack, &
[ 6]           is_Stack_Empty, is_Stack_Full
[ 7]   integer, parameter :: limit = 999 ! stack size limit
[ 8]
[ 9]   type stack
[10]   private
[11]     integer      :: size      ! size of array
[12]     integer      :: top       ! top of stack
[13]     type (Object) :: a(limit) ! stack items array
[14]   end type
[15] contains ! encapsulated functionality
[16]
[17]   type (stack) function make_Stack (n) result (s)      ! constructor
[18]     integer, optional :: n ! size of stack
[19]     s%size = limit ; if ( present (n) ) s%size = n
[20]     s%top = 0      ! object array not initialized
[21]   end function make_Stack
[22]
[23]   subroutine push_on_Stack (s, item) ! push item on top of stack
[24]     type (stack), intent(inout) :: s
[25]     type (Object), intent(in)   :: item
[26]     s%top = s%top + 1 ; s%a(s%top) = item
[27]   end subroutine push_on_Stack
[28]
[29]   type (Object) function pop_from_Stack (s) result (item) ! off top
[30]     type (stack), intent(inout) :: s
[31]     if ( s%top < 1 ) then
[32]       call exception ("pop_from_Stack","stack is empty")
[33]     else
[34]       item = s%a(s%top) ; s%top = s%top - 1
[35]     end if ; end function pop_from_Stack
[36]
[37]   logical function is_Stack_Empty (s) result (b)
[38]     type (stack), intent(in) :: s
[39]     b = ( s%top == 0 ) ; end function is_Stack_Empty
[40]
[41]   logical function is_Stack_Full (s) result (b)
[42]     type (stack), intent(in) :: s
[43]     b = ( s%top == s%size ) ; end function is_Stack_Full
[44]
[45] end module class_Stack
```

**Figure 7.1:** A Typical Stack Class

```

[ 1] include 'class_stack.f'      ! previous figure
[ 2] program main
[ 3] use class_stack
[ 4] implicit none
[ 5] type (stack) :: b
[ 6] type (object) :: value, four, five, six
[ 7]
[ 8]     four = Object(4) ; five = Object(5) ; six = Object(6) ! initialize
[ 9]
[10]     b = make_stack(3)          ! private constructor
[11]     print *, is_stack_empty(b), is_stack_full(b) ! b = [], empty
[12]
[13]     call push_on_stack (b, four)          ! b = [4]
[14]     call push_on_stack (b, five)         ! b = [5,4]
[15]     call push_on_stack (b, six)          ! b = [6,5,4], full
[16]     print *, is_stack_empty(b), is_stack_full(b) ! F T
[17]
[18]     value = pop_from_stack (b) ; print *, value ! b = [5,4]
[19]     print *, is_stack_empty(b), is_stack_full(b) ! F F
[20]
[21]     value = pop_from_stack (b) ; print *, value ! b = [4]
[22]     print *, is_stack_empty(b), is_stack_full(b) ! F F
[23]
[24]     value = pop_from_stack (b) ; print *, value ! b = [], empty
[25]     print *, is_stack_empty(b), is_stack_full(b) ! T F
[26]
[27]     value = pop_from_stack (b)             ! nothing to pop
[28] end program main ! running gives:
[29] ! T F ! F T
[30] ! 6   ! F F
[31] ! 5   ! F F
[32] ! 4   ! T F
[33] ! Exception occurred in subprogram pop_from_stack
[34] ! With message: stack is empty

```

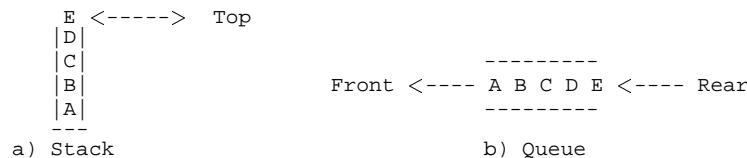
**Figure 7.2:** Testing a Stack of Objects

Full ?	F	F	F	T	F	F	F	F
Empty ?	T	F	F	F	F	F	T	T
Error ?	N	N	N	N	N	N	N	Y

Stack:		4	5	6	5	4		
			4	5	4			
			4	4				
	---	---	---	---	---	---	---	---
(Line)	9	12	13	14	17	20	23	26

**Figure 7.3:** Steps in the Stack Testing



**Figure 7.4:** Simple Containers

## 7.3 Queues

A comparison of a stack and another simple container, a *queue*, is given in Fig. 7.4. Its name queue comes from the British word which means waiting in a line for service. A queue is a container into which elements may be inserted at one end, called the *rear*, and leave only from the other end, called the *front*. The first element in the queue expects to be the first serviced and, thus, be the first out of line. A queue is a *first-in first-out* (FIFO) container system. In planning our first queue container we will again make use of an array of objects. Doing so one quickly finds that you are much less likely to encounter a full queue if it is stored as a so-called fixed circular array with a total of `Q_Size_Limit` storage slots. At this point we define the structure of our queue to be:

```
module Queue_type
! A queue stored as a so-called fixed circular array with a total
! of Q_Size_Limit storage slots; requires remainder function, mod.
! (version 1, i.e., without allocatable arrays and pointers)
  use object_type ! to define objects in the Container
  implicit none

  integer, parameter :: Q_Size_Limit = 999

  type Queue
  private
    integer :: head ! index of first element
    integer :: tail ! index of last element
    integer :: length ! size of used storage
    type (Object) :: store (Q_Size_Limit) ! a circular array
  end type Queue
end module Queue_type
```

An interface contract that will allow us to build a typical queue is:

```
module Queue_of_Objects
implicit none
  public :: Queue, Add_to_Q, Create_Q, Get_Front_of_Q, Is_Q_Empty, &
    Is_Q_Full, Get_Length_of_Q, Remove_from_Q

  interface ! for a class_Queue contract

    subroutine Add_to_Q (Q, item) ! add to tail of queue
      use Queue_type ! for Queue structure
      type (Queue), intent(inout) :: Q
      type (Object), intent(in) :: item ; end Subroutine Add_to_Q

    function Create_Q (N) result (Q) ! manual constructor
      use Queue_type ! for Queue structure
      integer, intent(in) :: N ! size of the new array
      type (Queue) :: Q ; end function Create_Q

    function Get_Capacity_of_Q (Q) result (item)
      use Queue_type ! for Queue structure
      type (Queue), intent(in) :: Q
      type (Object) :: item ; end function Get_Capacity_of_Q

    function Get_Front_of_Q (Q) result (item)
      use Queue_type ! for Queue structure
      type (Queue), intent(in) :: Q
      type (Object) :: item ; end function Get_Front_of_Q

    function Is_Q_Empty (Q) result(B)
      use Queue_type ! for Queue structure
      type (Queue), intent(in) :: Q
      logical :: B ; end function Is_Q_Empty

    function Is_Q_Full (Q) result(B)
      use Queue_type ! for Queue structure
      type (Queue), intent(in) :: Q
      logical :: B ; end function Is_Q_Full

    function Get_Length_of_Q (Q) result (N)
      use Queue_type ! for Queue structure
      type (Queue), intent(in) :: Q
      integer :: N ; end function Get_Length_of_Q

    subroutine Remove_from_Q (Q) ! remove from head of queue
      use Queue_type ! for Queue structure
      type (Queue), intent(inout) :: Q ; end subroutine Remove_from_Q

  end interface
end module Queue_of_Objects
```



For a specific version we provide full details for objects containing an integer in Fig. 7.5, and test and display the validity of the implementation in Fig. 7.6, where again the objects are taken to be integers (lines 15, 19, 20).

```
[ 1] module class_Queue                                ! file: class_Queue.f90
[ 2]
[ 3] ! A queue stored as a so-called fixed circular array with a total of
[ 4] ! Q_Size_Limit storage slots; requires remainder function, mod.
[ 5] ! (i.e., without allocatable arrays and pointers)
[ 6]
[ 7] use exceptions                                    ! inherit exception handler
[ 8] implicit none
[ 9]
[10] public :: Queue, Add_to_Q, Create_Q, Get_Front_of_Q
[11]         Is_Q_Full, Get_Length_of_Q, Remove_from
[12]
[13] integer, parameter :: Q_Size_Limit = 3
[14]
[15] type Queue
[16]   private
[17]     integer :: head          ! index of first element
[18]     integer :: tail          ! index of last element
[19]     integer :: length        ! size of used storage
[20]     integer :: store (Q_Size_Limit) ! a circular array of elements
[21] end type Queue
[22]
[23] contains                                           ! member functionality
[24]
[25] Subroutine Add_to_Q (Q, item)                      ! add to tail of queue
[26]   type (Queue), intent(inout) :: Q
[27]   integer,          intent(in)  :: item
[28]
[29]   if ( Is_Q_Full(Q) ) call exception ("Add_to_Q","full Q")
[30]   Q%store (Q%tail) = item
[31]   Q%tail      = 1 + mod (Q%tail, Q_Size_Limit)
[32]   Q%length    = Q%length + 1 ; end Subroutine Add_to_Q
[33]
[34] type (Queue) function Create_Q (N) result (Q)      ! manual constructor
[35]   integer, intent(in) :: N ! size of the new array
[36]   integer              :: k ! implied loop
[37]
[38]   if (N > Q_Size_Limit) call exception("Create_Q","increase size")
[39]   Q = Queue (1, 1, 0, (/ (0, k=1,N) /)) ! intrinsic constructor
[40] end function Create_Q
[41]
[42] integer function Get_Capacity_of_Q (Q) result (item)
[43]   type (Queue), intent(in) :: Q
[44]
[45]   item = Q_size_Limit - Q%length ; end function Get_Capacity_
[46]
[47] integer function Get_Front_of_Q (Q) result (item)
[48]   type (Queue), intent(in) :: Q
[49]
[50]   if (Is_Q_Empty(Q)) call exception("Get_Front_of_Q","em
[51]   item = Q%store (Q%head) ; end function Get_Front_of_Q
[52]
[53] logical function Is_Q_Empty (Q) result(B)
[54]   type (Queue), intent(in) :: Q
[55]
[56]   B = (Q%length == 0) ; end function Is_Q_Empty
[57]
[58] logical function Is_Q_Full (Q) result(B)
[59]   type (Queue), intent(in) :: Q
[60]
[61]   B = (Q%length == Q_Size_Limit) ; end function Is_Q_Full
[62]
[63] integer function Get_Length_of_Q (Q) result (N)
[64]   type (Queue), intent(in) :: Q
[65]   N = Q%length ; end function Get_Length_of_Q
[66]
[67] subroutine Remove_from_Q (Q)                      ! remove from head of queue
[68]   type (Queue), intent(inout) :: Q
[69]
[70]   if (Is_Q_Empty(Q)) call exception("Remove_from_Q","empty Q")
[71]   Q%head = 1 + mod (Q%head, Q_Size_Limit)
[72]   Q%length = Q%length - 1 ; end subroutine Remove_from_Q
[73]
[74] end module class_Queue                                ! file: class_Queue.f
```

**Figure 7.5:** A Typical Queue Class

```

[ 1] program main
[ 2]   use class_Queue ! inherit its methods & class global constants
[ 3]   implicit none
[ 4]
[ 5]   type (Queue) :: C, B ! not used, used
[ 6]   integer :: value, limit = 3 ! work items
[ 7]
[ 8]   C = Create_Q (limit) ! private constructor
[ 9]   print *, "Length of C = ", Get_Length_of_Q (C)
[10]   print *, "Capacity of C = ", Get_Capacity_of_Q (C)
[11]   print *, "C empty? full? ", is_Q_Empty (C), is_Q_Full (C) !
[12]
[13]   B = Create_Q (3) ! private constructor
[14]   print *, "B empty? full? ", is_Q_Empty (B), is_Q_Full (B) !
[15]
[16]   call Add_to_Q (B, 4); print *, "B = [4]"
[17]   print *, "Length of B = ", Get_Length_of_Q (B)
[18]   print *, "B empty? full? ", is_Q_Empty (B), is_Q_Full (B) !
[19]
[20]   call Add_to_Q (B, 5); print *, " B = [4,5]"
[21]   call Add_to_Q (B, 6); print *, " B = [4,5,6], full"
[22]   print *, "Length of B = ", Get_Length_of_Q (B)
[23]   print *, "B empty? full? ", is_Q_Empty (B), is_Q_Full (B) !
[24]   print *, "Capacity of B = ", Get_Capacity_of_Q (B)
[25]
[26]   value = Get_Front_of_Q (B); print *, "Front Q value = ", value
[27]
[28]   call Remove_from_Q (B); print *, "Removing from B"
[29]   print *, "Length of B = ", Get_Length_of_Q (B)
[30]   print *, "B empty? full? ", is_Q_Empty (B), is_Q_Full (B) !
[31]   value = Get_Front_of_Q (B); print *, "Front Q value = ", value
[32]
[33]   call Remove_from_Q (B); print *, "Removing from B"
[34]   print *, "Length of B = ", Get_Length_of_Q (B)
[35]   print *, "B empty? full? ", is_Q_Empty (B), is_Q_Full (B) !
[36]
[37]   call Remove_from_Q (B); print *, "Removing from B"
[38]   print *, "Length of B = ", Get_Length_of_Q (B)
[39]   print *, "B empty? full? ", is_Q_Empty (B), is_Q_Full (B) !
[40]
[41]   print *, "Removing from B"; call Remove_from_Q (B)
[42]   call exception_status
[43] end program main ! running gives:
[44] ! Length of C = 0 ! Capacity of C = 3 ! C empty? full? T, F
[45] ! B empty? full? T, F
[46] ! B = [4] ! Length of B = 1 ! B empty? full? F, F
[47] ! B = [4,5]
[48] ! B = [4,5,6], full ! Length of B = 3 ! B empty? full? F, T
[49] ! Capacity of B = 0 ! Front Q value = 4 ! Removing from B
[50] ! Length of B = 2 ! B empty? full? F, F ! Front Q value = 5
[51] ! Removing from B ! Length of B = 1 ! B empty? full? F, F
[52] ! Removing from B ! Length of B = 0 ! B empty? full? T, F
[53] ! Removing from B
[54] ! Exception Status Thrown
[55] ! Program :Remove_from_Q
[56] ! Message :empty Q
[57] ! Level : 5
[58] !
[59] ! Exception Summary:
[60] ! Exception count = 1
[61] ! Highest level = 5

```

**Figure 7.6:** Testing of the Queue Class

## 7.4 Linked Lists

From our limited discussion of stacks and queues it should be easy to see that to try to insert or remove an object at the middle of a stack or queue is not an efficient process. *Linked lists* are containers which make it easy to perform the operations of insertion and deletion. A linked list of objects can be thought of as a group of boxes, usually called *nodes*, each containing an object to stored and a *pointer*, or reference, to the box containing the next object in the list. In most of our applications a list is referenced by a special box, called the *header* or *root* node, which does not store an object but serves mainly to point to the first linkable box, and thereby produces a condition where the list is never truly empty. This simplifies the insertion scheme by removing an algorithmic special case. We will begin our introduction of these topics with a *singly linked list*, also known as a simple list. It is capable of being traversed in only one direction, from the beginning of the list to the end, or vice versa.

As we have seen, arrays of data objects work well so long as we know, or can compute, in advance the amount of data to be stored. The data structures (linked lists and trees) to be considered here employ *pointers* to store and change data objects when we do not know the required amount of storage in advance. During program execution linked lists and trees allow separate memory allocations for each individual data object. However, they do not permit direct access to an arbitrary object in the container. Instead some searching must be performed and thus they incur an execution time penalty for such an access operation. That penalty is smaller in tree structures than in linked lists (but is smallest of all in arrays).

Linked lists and trees must use pointer (reference) variables. Fortran pointers can simply be thought of as an alias for other variables of the same type. We are beginning to see that pointers give a programmer more power. However, that includes more power to “shoot yourself in the foot”; they make it hard to find some errors; and can lead to new types of errors such as the so called *memory leaks*. Recall that each pointer must be in one of three states: undefined, null, or associated. As dummy arguments within routines pointer variables cannot be assigned the INTENT attribute. That means they have a greater potential for undesired *side effects*. To avoid accidentally changing a pointer it is good programming practice to clearly state in comments the INTENT of all dummy pointer arguments and to immediately copy those with an INTENT IN attribute. Thereafter working with the copied pointer guarantees that an error or later modification of the routine can not produce a side effect on the pointer. We also want to avoid a *dangling pointer* which is caused by a deallocation that leaves its target object forever inaccessible. A related problem is a memory leak or *unreferenced storage* such as the program segment:

```
real, pointer :: X_ptr (:)
  allocate ( X_ptr(Big_number) )
  . . . ! use X_ptr
  nullify ( X_ptr ) ! dangling pointer
```

because now there is no way to release memory for X\_ptr. To avoid this we need to free the memory before the pointer is nullified, so the segment becomes:

```
real, pointer :: X_ptr (:)
  allocate ( X_ptr(Big_number) )
  . . . ! use X_ptr
  deallocate ( X_ptr ) ! memory released
  nullify ( X_ptr )
```

Remember that in F95 the memory is automatically deallocated at the end of the scope of the variable, unless one retains the variable with a SAVE statement (and formally deallocates it elsewhere).

### 7.4.1 Singly Linked Lists

We begin the study of the singly linked list by showing the notations employed in Fig. 7.7. From experience we have chosen to have a dummy first node, called `first`, to simplify our algorithms so that a list is never truly empty. Also as we scan through a list we will use one pointer, called `current`, to point to the current object in the list and a companion, called `previous`, to point to the directly preceding object (if any). If no objects have been placed in the list then both of these simply point to the `first` node. The end of the list is denoted by the `next` pointer attribute taking on the `null` value. To insert or delete objects one must be able to rank two objects. This means that in order to have a generic linked list one must overload the relational operators, (`<` and `==`) when the object to be placed in the container is defined. Since most objects have different types of attributes the overloading process is clearly application

```

Type (Singly_Linked_Node) ::
|-----|
| Object | An Object instance attribute
| instance | and one pointer attribute.
|-----|
| Next   |
|-----|
| pointer |----->
|-----|

```

a) Singly linked node

```

|-----|
| First |<--- Singly_Linked_List % First (An empty object instance)
|-----|
| Next  --+----->| Obj_1 |<--- Previous_pointer
|-----|
| Next  --+----->| Obj_m |<--- Current_pointer
|-----|
| Next  --+----->| Obj_n |
|-----|
| Next  --+-----> Null
|-----|
|-----|
|-----|

```

b) List of singly linked nodes

```

|-----|
| First |<--- Singly_Linked_List % First
|-----|
| First |<--- Previous_pointer
|-----|
| First |<--- Current_pointer
|-----|
| Next  --+-----> Null
|-----|
|-----|
|-----|

```

c) An ``empty'' (one node) singly linked list

**Figure 7.7:** Singly linked list terminology

dependent. The process for inserting an object is sketched in Fig. 7.8 while that for deleting an object is in Fig. 7.9.

The `Singly_Linked_List` class is given in Fig. 7.10. It starts with the definition of a singly linked node (lines 4-8) that has an object attribute and a pointer attribute to locate the next node. Then a list is begun (lines 10-13) by creating the dummy first node that is consider to represent an empty list. The object deletion member must employ an overloaded operator (line 28), as must the insertion member (line 52). Observe that a list never gets “full”, unless the system runs out of memory. The empty list test member (line 62) depends on the pointer status, but is independent of the objects stored. The constructor for a list (line 68) simply creates the first node and nullifies it. The printing member (line 74) is called an *iterator* since it runs through all objects in the list. The testing program for this container type and its output results are given in Fig. 7.11. In order to test such a container it is necessary to have an object type defined. Here an object with a single integer value was selected, and thus it was easy to overload the relational operators with a clear meaning as shown in Fig. 7.12.

INSERT

**Figure 7.8:** Inserting an Object in a Singly Linked List

DELETE

**Figure 7.9:** Deleting an Object from a Singly Linked List

```

[ 1] module singly_linked_list
[ 2]   use class_Object
[ 3]   implicit none
[ 4]
[ 5]   type S_L_node                                ! Singly Linked Node
[ 6]   private
[ 7]     type (Object)                               :: value      ! Object attribute
[ 8]     type (S_L_node), pointer :: next            ! Pointer to next node
[ 9]   end type S_L_node
[10]
[11]   type S_L_list                                ! Singly Linked List of Nodes
[12]   private
[13]     type (S_L_node), pointer :: first ! Dummy first object in list
[14]   end type S_L_list
[15]
[16] contains
[17]   subroutine S_L_delete (links, Obj, found)
[18]     type (S_L_list), intent (inout) :: links
[19]     type (Object),   intent (in)    :: Obj
[20]     logical,         intent (out)    :: found
[21]     type (S_L_node), pointer        :: previous, current
[22]
[23]     ! find location of Obj
[24]     previous => links%first      ! begin at top of list
[25]     current  => previous%next    ! begin at top of list
[26]     found = .false.             ! initialize
[27]     do
[28]       if ( found .or. (.not. associated (current))) return ! list end
[29]       if ( Obj == current%value ) then ! *** OVERLOADED ***
[30]         found = .true. ; exit ! this location search
[31]       else ! move the next node in list
[32]         previous => previous%next
[33]         current  => current%next
[34]       end if
[35]     end do ! to find location of node with Obj
[36]     ! delete if found
[37]     if ( found ) then
[38]       previous%next => current%next ! redirect pointer
[39]       deallocate ( current )        ! free space for node
[40]     end if
[41]   end subroutine S_L_delete
[42]

```

Fig. 8.5, A Typical Singly Linked List Class of Objects (continued)

```

[43]  subroutine S_L_insert (links, Obj )
[44]      type (S_L_list), intent (inout) :: links
[45]      type (Object),  intent (in)    :: Obj
[46]      type (S_L_node), pointer       :: previous, current
[47]
[48]      ! Find location to insert a new object
[49]      previous => links%first           ! initialize
[50]      current  => previous%next         ! initialize
[51]      do
[52]          if ( .not. associated (current) ) exit ! insert at end
[53]          if ( Obj < current%value ) exit      ! *** OVERLOADED ***
[54]          previous => current               ! insert before current
[55]          current  => current%next          ! move to next node
[56]      end do ! to locate insert node
[57]      ! Insert before current (duplicates allowed)
[58]      allocate ( previous%next )          ! get new node space
[59]      previous%next%value = Obj           ! new object inserted
[60]      previous%next%next => current       ! new next pointer
[61]  end subroutine S_L_insert
[62]
[63]  function is_S_L_empty (links) result (t_or_f)
[64]      type (S_L_list), intent (in) :: links
[65]      logical                  :: t_or_f
[66]      t_or_f = .not. associated ( links%first%next )
[67]  end function is_S_L_empty
[68]
[69]  function S_L_new () result (new_list)
[70]      type (S_L_list) :: new_list
[71]      allocate ( new_list%first )          ! get memory for the object
[72]      nullify ( new_list%first%next )      ! begin with empty list
[73]  end function S_L_new
[74]
[75]  subroutine print_S_L_list (links)
[76]      type (S_L_list), intent (in) :: links
[77]      type ( S_L_node), pointer    :: current
[78]      integer                      :: counter
[79]      current => links%first%next
[80]      counter = 0 ; print *, 'Link   Object Value'
[81]      do
[82]          if ( .not. associated (current) ) exit ! list end
[83]          counter = counter + 1
[84]          print *, counter, '      ', current%value
[85]          current => current%next
[86]      end do
[87]  end subroutine print_S_L_list
[88] end module singly_linked_list

```

**Figure 7.10:** A Typical Singly Linked List Class of Objects

```

[ 1] program main      ! test a singly linked object list
[ 2] use singly_linked_list
[ 3] implicit none
[ 4]   type (S_L_list) :: container
[ 5]   type (Object)   :: Obj_1, Obj_2, Obj_3, Obj_4
[ 6]   logical         :: delete_ok
[ 7]
[ 8]   Obj_1 = Object(15) ; Obj_2 = Object(25) ! constructor
[ 9]   Obj_3 = Object(35) ; Obj_4 = Object(45) ! constructor
[10]   container = S_L_new()
[11]   print *, 'Empty status is ', is_S_L_empty (container)
[12]   call S_L_insert (container, Obj_4) ! insert object
[13]   call S_L_insert (container, Obj_2) ! insert object
[14]   call S_L_insert (container, Obj_1) ! insert object
[15]   call print_S_L_list (container)
[16]
[17]   call S_L_delete (container, obj_2, delete_ok)
[18]   print *, 'Object: ', Obj_2, ' deleted status is ', delete_ok
[19]   call print_S_L_list (container)
[20]   print *, 'Empty status is ', is_S_L_empty (container)
[21]
[22]   call S_L_insert (container, Obj_3) ! insert object
[23]   call print_S_L_list (container)
[24]   call S_L_delete (container, obj_1, delete_ok)
[25]   print *, 'Object: ', Obj_1, ' deleted status is ', delete_ok
[26]   call S_L_delete (container, obj_4, delete_ok)
[27]   print *, 'Object: ', Obj_4, ' deleted status is ', delete_ok
[28]   call print_S_L_list (container)
[29]   print *, 'Empty status is ', is_S_L_empty (container)
[30]
[31]   call S_L_delete (container, obj_3, delete_ok)
[32]   print *, 'Object: ', Obj_3, ' deleted status is ', delete_ok
[33]   print *, 'Empty status is ', is_S_L_empty (container)
[34]   call print_S_L_list (container)
[35] end program          ! running yields
[36] ! Empty status is  T
[37] ! Link      Object Value
[38] ! 1         15
[39] ! 2         25
[40] ! 3         45
[41] ! Object:  25  deleted status is  T
[42] ! Link      Object Value
[43] ! 1         15
[44] ! 2         45
[45] ! Empty status is  F
[46] ! Link      Object Value
[47] ! 1         15
[48] ! 2         35
[49] ! 3         45
[50] ! Object:  15  deleted status is  T
[51] ! Object:  45  deleted status is  T
[52] ! Link      Object Value
[53] ! 1         35
[54] ! Empty status is  F
[55] ! Object:  35  deleted status is  T
[56] ! Empty status is  T
[57] ! Link      Object Value

```

**Figure 7.11:** Testing the singly linked list with integers

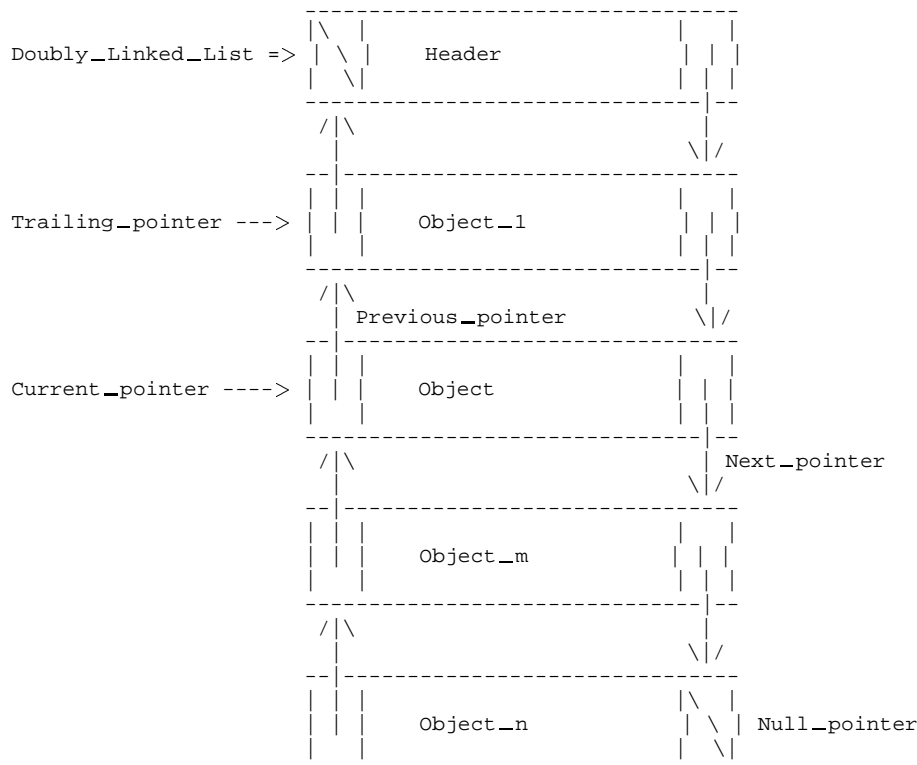
```

[ 1] module class_Object
[ 2] implicit none
[ 3]   type Object          ! An integer object for testing lists
[ 4]   integer :: data ; end type Object
[ 5]
[ 6]   interface operator (<)          ! for sorting or insert
[ 7]   module procedure less_than_Object ; end interface
[ 8]   interface operator (==)         ! for sorting or delete
[ 9]   module procedure equal_to_Object ; end interface
[10]
[11] contains                          ! overload definitions only
[12]   function less_than_Object (Obj_1, Obj_2) result (Boolean)
[13]   type (Object), intent(in) :: Obj_1, Obj_2
[14]   logical                     :: Boolean
[15]   Boolean = Obj_1%data < Obj_2%data ! standard (<) here
[16] end function less_than_Object
[17]   function equal_to_Object (Obj_1, Obj_2) result (Boolean)
[18]   type (Object), intent(in) :: Obj_1, Obj_2
[19]   logical                     :: Boolean
[20]   Boolean = Obj_1%data == Obj_2%data ! standard (==) here
[21] end function equal_to_Object
[22] end module class_Object

```

**Figure 7.12:** Typical object definition to test a singly linked list





**Figure 7.13:** Notations for a Doubly Linked List

#### 7.4.1.1 Example: A List of Sparse Vectors

In this example we want to create a linked list to hold sparse vectors (singly subscripted arrays) where the length of each vector is specified. We will do simple operations on all the vectors like input them, normalize them, add them (if their sizes are the same), etc. In doing this we will make use of some of the efficiencies that F90 provides for arrays, such as using the subscript array triplet to avoid serial loops, and operating on arrays by name alone. This is an example where a similar C++ implementation would be much longer in length because of the need to provide all the serial loops.

### 7.4.2 Doubly Linked Lists

The notations of the doubly linked list are shown in Fig. 7.13. Again we have chosen to have a dummy first node, called `header`, to simplify our algorithms so that a list is never truly empty. Also as we scan through a list we will use one pointer, called `current`, to point to the current object in the list and a companion, called `previous`, to point to the directly preceding object (if any). If no objects have been placed in the list then both of these simply point to the `header` node. The end of the list is denoted by the `next` pointer attribute taking on the `null` value. To insert or delete objects one must be able to rank two objects. This means that in order to have a generic linked list one must again overload the relational operators, (`<` and `==`) when the object to be placed in the container is defined.

An incomplete, but illustrative `Doubly_Linked_List` class is given in Fig. 7.14. It starts with the definition of a doubly linked node (lines 4-8) that has an object attribute and a pair of pointer attributes to locate the nodes on either side of the object. Then a list is begun (lines 10-13) by creating the dummy first node that is considered to represent an empty list. The object insertion member must employ an overloaded operator (line 53), as before. Observe that a list never gets “full”, unless the system runs out of memory. The constructor for a list (line 17) simply creates the first node and nullifies its pointers. A corresponding destructor (line 24) has been provided to delete every thing associated with the list when we are done

```

[ 1] module doubly_linked_list
[ 2] use class_Object
[ 3] implicit none
[ 4] type D_LL_node
[ 5]   private
[ 6]     type (Object) :: Obj
[ 7]     type (D_LL_node), pointer :: previous
[ 8]     type (D_LL_node), pointer :: next
[ 9]   end type D_LL_node
[10]
[11] type D_LL_list
[12]   private
[13]     type (D_LL_node), pointer :: header
[14]   end type D_LL_list
[15]
[16] contains
[17]
[18] function D_LL_new () result (new_list) ! constructor
[19]   type (D_LL_list) :: new_list
[20]   allocate (new_list % header)
[21]   nullify (new_list % header % previous)
[22]   nullify (new_list % header % next)
[23] end function D_LL_new
[24]
[25] subroutine destroy_D_LL_List (links) ! destructor
[26]   type (D_LL_list), intent (in) :: links
[27]   type (D_LL_node), pointer :: current
[28]   do
[29]     current => links % header % next
[30]     if ( .not. associated ( current ) ) exit
[31]     current % previous % next => current % next
[32]     if ( associated ( current % next ) ) then
[33]       current % next % previous => current % previous
[34]     end if
[35]     nullify ( current % previous )
[36]     nullify ( current % next )
[37]     print *, 'Destroying object ', current % Obj
[38]     deallocate ( current )
[39]   end do
[40]   deallocate ( links % header )
[41]   print *, 'D_LL_List destroyed'
[42] end subroutine destroy_D_LL_List
[43]

```

Fig. 7.14, A Typical Doubly Linked List Class of Objects (continued)

with it. The printing member (line 90) is called an iterator since it runs through all objects in the list. The testing program for this container type and its output results are given in Fig. 7.15. Here an object with a single integer value was selected, and thus it was easy to overload the relational operators with a clear meaning as shown in Fig. 7.12.

## 7.5 Direct (Random) Access Files

Often it may not be necessary to create special object data structures such as those outlined above. From its beginning Fortran has had the ability to create a sophisticated random access data structure where the implementation details are hidden from its user. This was necessary originally since the language was utilized on computers with memory sizes that are considered tiny by today's standard (e.g., 16 Kb), but it was still necessary to efficiently create and modify large amounts of data. The standard left the actual implementation details to the compiler writers. That data structure is known as a "direct access file". It behaves like a single subscript array in that the object at any position can be read, modified, or written at random so long as the user keeps up with the position of interest. The user simply supplies the position, known as the record number, as additional information in the read and write statements. With today's hardware, if the file is stored on a virtual disk (stored in random access memory) there is practically no difference in access times for arrays and direct files.

It should be noted here that since pointers are addresses in memory they can not be written to any type of file. That, of course, means that no object having a pointer as an attribute can be written either. Thus in some cases one must employ the other types of data structures illustrated earlier in the chapter.

To illustrate the basic concepts of a random access file consider the program called `random_access_file` which is given in Fig. 7.16. In this case the object is simply a character string, as

```

[ 43] subroutine D_L_insert_before (links, values)
[ 44]   type (D_L_list), intent (in) :: links
[ 45]   type (Object),   intent (in) :: values
[ 46]   type (D_L_node), pointer      :: current    ! Temp traversal pointer
[ 47]   type (D_L_node), pointer      :: trailing   ! Preceding node pointer
[ 48]   ! Find location to insert new node, in ascending order
[ 49]   trailing => links % header                ! initialize
[ 50]   current  => trailing % next                ! initialize
[ 51]   do
[ 52]     if (.not. associated (current)) exit      ! insert at end
[ 53]     if (values < current % Obj) exit          ! insert before current
[ 54]     trailing => current                      ! move to next node
[ 55]     current  => current % next              ! move to next node
[ 56]   end do
[ 57]   ! Insert before current (duplicates allowed)
[ 58]   allocate (trailing % next)                ! get new node space
[ 59]   trailing % next % Obj = values             ! new object inserted
[ 60]   ! Insert the new pointers
[ 61]   if (.not. associated (current)) then      ! End of list (special)
[ 62]     nullify (trailing % next % next)
[ 63]     trailing % next % previous => trailing
[ 64]   else
[ 65]     trailing % next % next => current        ! Not the end of the list
[ 66]     trailing % next % previous => trailing
[ 67]     current % previous => trailing % next
[ 68]   end if
[ 69] end subroutine D_L_insert_before
[ 70]
[ 71] function Get_Obj_at_Ptr (ptr_to_Obj) result ( values)
[ 72]   type (D_L_node), intent (in) :: ptr_to_Obj
[ 73]   type (Object)                :: values    ! intent out
[ 74]   values = ptr_to_Obj % Obj
[ 75] end function Get_Obj_at_Ptr
[ 76]
[ 77] function Get_Ptr_to_Obj (links, values) result (ptr_to_Obj)
[ 78]   type (D_L_list), intent (in) :: links    ! D_L_list header
[ 79]   type (Object),   intent (in) :: values    ! Node identifier Object
[ 80]   type (D_L_node), pointer      :: ptr_to_Obj ! Pointer to the Object
[ 81]   type (D_L_node), pointer      :: current   ! list traversal pointer
[ 82]   current => links % header % next
[ 83]   do ! Search list, WARNING: runs forever if values not in list
[ 84]     if (current % Obj == values) exit      ! *** OVERLOADED ***
[ 85]     current => current % next
[ 86]   end do
[ 87]   ptr_to_Obj => current                    ! Return pointer to node
[ 88] end function Get_Ptr_to_Obj
[ 89]
[ 90] subroutine print_D_L_list ( links )
[ 91]   type (D_L_list), intent (in) :: links
[ 92]   type (D_L_node), pointer      :: current ! Node traversal pointer
[ 93]   integer                      :: counter ! Link position
[ 94]   ! Traverse the list and print its contents to standard output
[ 95]   current => links % header % next
[ 96]   counter = 0 ; print *, 'Link   Object Value'
[ 97]   do
[ 98]     if (.not. associated (current)) exit
[ 99]     counter = counter + 1
[100]     print *, counter, ' ', current % Obj
[101]     current => current % next
[102]   end do
[103] end subroutine print_D_L_list
[104] end module doubly_linked_list

```

**Figure 7.14:** A Typical Doubly Linked List Class of Objects

defined in line 4. The hardware transportability of this code is assured by establishing the required constant record with the intrinsic given in line 10. It is then used in opening the file, which is designated as a direct file in line 12. Lines 16–24 create the object record numbers in a sequential fashion. They also define the new object to be stored with each record. In lines 27–32 the records are accessed in a backwards order, but could have been accessed in any random or partial order. In line 35 a random object is given a new value. Finally, the changes are output in a sequential order in lines 37–42. Sample input data and program outputs are included as comments at the end of the program.

```

[ 1] program main
[ 2] use doubly_linked_list
[ 3] implicit none
[ 4]   type (D_L_list)           :: container
[ 5]   type (Object)             :: Obj_1, Obj_2, Obj_3, Obj_4
[ 6]   type (Object)             :: value_at_pointer
[ 7]   type (D_L_node), pointer :: point_to_Obj_3
[ 8]
[ 9]   Obj_1 = Object(15) ; Obj_2 = Object(25)
[10]   Obj_3 = Object(35) ; Obj_4 = Object(45)
[11]   container = D_L_new()
[12]   ! print *, 'Empty status is ', is_D_L_empty (container)
[13]   call D_L_insert_before (container, Obj_4)
[14]   call D_L_insert_before (container, Obj_2)
[15]   call D_L_insert_before (container, Obj_1)
[16]   call D_L_insert_before (container, Obj_3)
[17]   call print_D_L_list (container)
[18]
[19]   ! find and get Obj_3
[20]   point_to_Obj_3 = Get_Ptr_to_Obj (container, Obj_3)
[21]   value_at_pointer = Get_Obj_at_Ptr (point_to_Obj_3)
[22]   print *, 'Object: ', Obj_3, ' has a value of ', value_at_pointer
[23]   call destroy_D_L_List (container)
[24] end program main
[25] ! Link      Object Value      ! Running gives:
[26] ! 1         15
[27] ! 2         25
[28] ! 3         35
[29] ! 4         45
[30] ! Object:   35   has a value of   35
[31] ! Destroying object 15
[32] ! Destroying object 25
[33] ! Destroying object 35
[34] ! Destroying object 45
[35] ! D_L_List destroyed

```

**Figure 7.15:** Testing a Partial Doubly Linked List

```

[ 1] program random_access_file
[ 2] ! create a file and access or modify it randomly
[ 3] implicit none
[ 4] character(len=10) :: name
[ 5] integer :: j, rec_len, no_name, no_open
[ 6] integer :: names = 0, unit = 1
[ 7]
[ 8] ! find the hardware dependent record length of the object
[ 9] ! to be stored and modified. Then open a binary file.
[10] inquire (iolen = rec_len) name
[11] open (unit, file = "random_list", status = "replace",
[12]       access = "direct", recl = rec_len,
[13]       form = "unformatted", iostat = no_open)
[14] if ( no_open > 0 ) stop 'open failed for random_list'
[15]
[16] ! read and store the names sequentially
[17] print *, ' '; print *, 'Original order'
[18] do ! forever from standard input
[19]   read (*, '(a)', iostat = no_name) name
[20]   if ( no_name < 0 ) exit ! the read loop
[21]   names = names + 1 ! record number
[22]   write (unit, rec = names) name ! save record
[23]   print *, name ! echo
[24] end do
[25] if ( names == 0 ) stop 'no records read'
[26]
[27] ! list names in reverse order
[28] print *, ' '; print *, 'Reverse order'
[29] do j = names, 1, -1
[30]   read (unit, rec = j) name
[31]   print *, name
[32] end do ! of random read
[33]
[34] ! change the middle name in random file
[35] write (unit, rec = (names + 1)/2) 'New_Name'
[36]
[37] ! list names in original order
[38] print *, ' '; print *, 'Modified data'
[39] do j = 1, names
[40]   read (unit, rec = j) name
[41]   print *, name
[42] end do ! of random read
[43]
[44] close (unit) ! replace previous records and save
[45] end program random_access_file
[46] ! Running with input of:   Name_1
[47] !                           B_name
[48] !                           3_name
[49] !                           name_4
[50] !                           Fifth
[51] ! Yields:
[52] ! Original order  Reverse order  Modified data
[53] ! Name_1          Fifth          Name_1
[54] ! B_name          name_4         B_name
[55] ! 3_name          3_name         New_Name
[56] ! name_4          B_name         name_4
[57] ! Fifth           Name_1         Fifth

```

**Figure 7.16:** Utilizing a Random Access File as a Data Structure

## 7.6 Exercises



## Chapter 8

# Arrays and Matrices

### 8.1 Subscripted Variables: Arrays

It is common in engineering and mathematics to employ a notation where one or more subscripts are appended to a variable which is a member of some larger set. Such a variable may be a member of a list of scalars, or it may represent an element in a vector, matrix, or Cartesian tensor.<sup>†</sup> In engineering computation, we usually refer to subscripted variables as *arrays*. Since programming languages do not have a convenient way to append the subscripts, we actually denote them by placing them in parentheses or square brackets. Thus, an element usually written as  $A_{jk}$  becomes  $A(j, k)$  in Fortran and MATLAB, and  $A[j][k]$  in C++.

Arrays have properties that need to be understood in order to utilize them correctly in any programming language. The primary feature of an array is that it must have at least one subscript. The “rank” of an array is the number of subscripts, or dimensions, it has. Fortran allows an array to have up to seven subscripts, C++ allows four, and MATLAB allows only two since it deals only with matrices. An array with two subscripts is called a rank-two array, one with a single subscript is called a rank-one array, or a vector. Matrices are rank-two arrays that obey special mathematical operations. A scalar variable has no subscripts and is sometimes called a rank zero array. Rank-one arrays with an extent of one are also viewed as a scalar.

The “extent” of a subscript or dimension is the number of elements allowed for that subscript. That is, the extent is an integer that ranges from the lower bound of the subscript to its upper bound. The lower bound of a subscript is zero in C++, and it defaults to unity in Fortran. However, Fortran allows the programmer to assign any integer value to the lower and upper bounds of a subscript.

The “size” of an array is the number of elements in it. That is, the size is the product of the extents of all of its subscripts. Most languages require the extent of each subscript be provided in order to allocate memory storage for the array.

The “shape” of an array is defined by its rank and extents. The shape is a rank-one array where each of its elements is the extent of the corresponding subscript of the array whose shape is being determined. Both Fortran and MATLAB have statements that return the shape and size of an array as well as statements for defining a new array by re-shaping an existing array.

It is also important to know which of two “storage mode” options a language employs to store and access array elements. This knowledge is especially useful when reading or writing full arrays. Arrays are stored by either varying their leftmost subscript first or by varying the rightmost subscript first. These are referred to as “column-wise” and “row-wise” access, respectively. Clearly, they are the same for rank-one arrays and differ for arrays of higher rank. Column-wise storage is used by Fortran and C++, while MATLAB uses row-wise storage.

Matrices are arrays that usually have only two subscripts: the first represents the row number, and the second the column number where the element is located. Matrix algebra places certain restrictions on the subscripts of two matrices when they are added or multiplied, etc. The fundamentals of matrices are covered in detail in this chapter.

---

<sup>†</sup>An  $n$ -th order tensor has  $n$  subscripts and transforms to different coordinate systems by a special law. The most common uses are scalars ( $n = 0$ ) and vectors ( $n = 1$ ).



Action	C++ <sup>a</sup>	F90	F77	MATLAB
Pre-allocate Initialize	<pre>integer A[100] for j=0,99   A[j]=12 end</pre>	<pre>INTEGER A(100) A=12</pre>	<pre>INTEGER A(100) do 5 J=1,100   A(J)=12 5 continue</pre>	<pre>A(100)=0 for j=1:100   A(j)=12 end</pre>

<sup>a</sup>Arrays in C++ have a starting index of zero.

**Table 8.1:** Typical Vector Initialization

Purpose	F90	MATLAB
Form subscripts	( )	( )
Separates subscripts & elements	,	,
Generates elements & subscripts	:	:
Separate commands	;	;
Forms arrays	(/ /)	[ ]
Continue to new line	&	...
Indicate comment	!	%
Suppress printing	default	;

**Table 8.2:** Special Array Characters

Both Fortran and C++ require you to specify the maximum range of each subscript of an array before the array or its elements are used. MATLAB does not have this as a requirement, but pre-allocating the array space can drastically improve the speed of MATLAB, as well as making much more efficient use of the available memory. If you do not pre-allocate MATLAB arrays, then the interpreter must check at each step to see if a position larger than the current maximum has been reached. If so, the maximum value is increased and memory is found to store the new element. Thus, failure to pre-allocate MATLAB arrays is permissible but inefficient.

For example, assume we want to set a vector *A* having 100 elements, to an initial value of 12. The procedures are compared in Table 8.1. This example could have also been done efficiently in F90 and MATLAB by using the colon operator: *A*(1:100) = 12. The programmer should be alert for the chance to replace loops with the colon operator: it's more concise while retaining readability and executes more quickly. The joys of the colon operator are described more fully in §8.1.3 (page 159).

Array operations often use special characters and operators. Fortran has “implied” DO loops associated with its array operations (see §4.3.2, page 60). Similar features in MATLAB and F90 are listed in Table 8.2.

Fortran has always had efficient array handling features, but until the release of F90 it was not easy to dynamically create and release the memory space needed to store arrays. That is a useful feature for arrays that require large amounts of space but are not needed for the entire life of the program. F90 has several types of arrays, with the most recent types being added to allow the use of array operations, and intrinsic functions similar to those in MATLAB. Without getting into the details of the F90 standards and terminology we will introduce the most common array usages in a historical order:

F77: Constant Arrays, Dummy Dimension Arrays, Variable Rank Arrays  
F90: Automatic Arrays, Allocatable Arrays.

These different approaches all have the common feature that memory space needed for an array must be set aside (allocated) before any element in the array is utilized.

The new F90 array features include the so-called automatic arrays. An automatic array is one that appears in a subroutine, or function and has its size, but *not* its name, provided in the argument list of the subprogram. For example,

```
subroutine auto_A_B (M, N, Other_arguments)
  implicit none
  integer :: M, N
```

```

      real    :: A(M, N), B(M)    ! Automatic arrays
      ! Create arrays A & B and use them for some purpose
      ...
end subroutine auto_A_B

```

would automatically allocate space for the *M* rows and *N* columns of the array *A* and for the *M* rows of array *B*. When the purpose of the subroutine is finished and it “returns” to the calling program the array space is automatically released, and the arrays *A* and *B* cease to exist. This is a useful feature, especially in Object Oriented programs. If the system does not have enough space available to allocate for the array the program stops and gives an error message to that effect. With today’s large memory computers that is unlikely to occur except for the common user error where the dimension argument is undefined.

An extension of this concept that allows more flexibility and control is the allocatable array. An allocatable array is one that has a known rank (number of subscripts), but an initially unknown extent (range over each subscript). It can appear in any program, function, or subroutine. For example,

```

program make_A_B      ! Allocatable arrays
implicit none
real, allocatable :: A(:, :), B(:)    ! Declares rank of each
integer           :: M, N             ! Row and column sizes
integer           :: A_B_Status       ! Optional status check
print *, "Enter the number of rows and columns: "
read *, M, N    ! Now know the (default) extent of each subscript
allocate ( A(M, N), B(M), stat = A_B_Status ) ! dynamic storage
! Verify that the dynamic memory was available
if ( A_B_Status /= 0 ) stop "Memory not available in make_A_B"
! Create arrays A & B and use them for some purpose
...
deallocate (A, B) ! free the memory space
! Do other things
...
end program make_A_B

```

would specifically allocate space for the *M* rows and *N* columns of the array *A* and for the *M* rows of array *B*, and optionally verify that the space was available. When the purpose of the arrays are finished the space is specifically released, and the arrays *A* and *B* cease to exist. The optional status checking feature is useful in the unlikely event that the array is so large that the system does not have that much dynamic space available. Then the user has the option of closing down the program in some desirable way, or simply stopping on the spot.

The old F77 standard often encouraged the use of dummy dimension arrays. The dummy dimension array is one that appears in a subroutine, or function and has its size *and* its name provided in the argument list of the subprogram. For example,

```

subroutine dummy_A_B (M, N, A, B, Other_things)
implicit none
integer :: M, N
real    :: A(M, N), B(M)    ! dummy arrays
! Create arrays A & B and use them for some purpose
...
end subroutine dummy_A_B

```

would imply that *existing* space for the *M* rows and *N* columns of the array *A* and for the *M* rows of array *B* (*or more*) was declared or allocated in the calling program. When the purpose of the subroutine is finished and it “returns” to the calling program the space in the calling program for the arrays *A* and *B* continues to exist until the declaring program unit terminates.

Of course the use of constant dimensioned arrays is always allowed. The constant dimension array is one that appears in any program unit and has integer constants, or integer *parameter* variables (preferred) as given extents for each subscript of an array. For example,

```

program main
implicit none
integer, parameter :: M_max=20, N_max=40 ! Maximum expected
integer           :: Days_per_Month(12) ! Constant array
integer           :: M, N                ! User sizes
real             :: A(M_max, N_max)      ! Constant arrays
print *, "Enter the number of rows and columns: "
read *, M, N    ! The user extent of each subscript

! Verify that the constant memory is available
if ( M > M_max ) stop "Row size exceeded in main"
if ( N > N_max ) stop "Column size exceeded in main"
! Create arrays A & B and use them for some purpose
call dummy_A_B (M, N, A, B, Other_things) ! dummy arrays

```

Action	F90	MATLAB
Define size <sup>a</sup> Enter rows	integer :: A (2, 3) A(1,:)=(/1,7,-2/) A(2,:)=(/3,4,6/)	A(2,3)=0; A=[1,7,-2; 3,4,6];

<sup>a</sup>Optional in MATLAB, but improves efficiency.

**Table 8.3:** Example Array Definitions

F90	MATLAB	Result
data = (/ (k, k=1,6) /) M = reshape(data, (/3,2/))  N = reshape(data, (/2,3/))	data = [1 : 6] M = reshape(data,3,2)  N = reshape(data,2,3)	$M = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$  $N = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$

**Table 8.4:** Array Reshape Intrinsics

```

...
end program main

subroutine dummy_A_B (M, N, A, B, Other_things) ! dummy arrays
implicit none
integer :: M, N
real    :: A(M, N), B(M)
! Create arrays A & B and use them for some purpose
...
end subroutine main

```

In general it is considered *very bad style* to use integer constants, like 12, in a dimension, or in a DO loop control, except for the unusual case where its meaning is obvious, and where you never expect to have to change the number. In the example declaration:

```
integer :: Days_per_Month(12) ! Constant array
```

It is obvious that we are thinking about 12 months per year and that we do not expect the number of months per year to ever change in other potential applications of this program.

### 8.1.1 Initializing Array Elements

Explicit lists of the initial elements in an array are allowed by C++, Fortran, and MATLAB. MATLAB is oriented to enter element values in the way that we read, that is, row by row. Fortran and C also allow array input by rows, but the default procedure is to accept values by ranging over its subscripts from left to right. That is, both F90 and C++ read by columns as their default mode. For example, consider the  $2 \times 3$  array

$$A = \begin{bmatrix} 1 & 7 & -2 \\ 3 & 4 & 6 \end{bmatrix}.$$

This array could be typed as explicit input with the commands shown in Table 8.3. An alternative for F90 and MATLAB is to define the full array by column order as a vector that is then reshaped into a matrix with a specified number of rows and columns. The use of the RESHAPE operator is shown in Table 8.4.

Returning to the previous example, we see that the matrix A could have also been defined as

F90	A = reshape((/1,3,7,4, -2,6/), (/2,3/)) A = reshape((/1,3,7,4, -2,6/), shape(A))
MATLAB	A = reshape([1,3,7,4, -2,6], 2,3)

To initialize the elements of an array to zero or unity, F90 and MATLAB have special constructs or functions that fill the bill. For example, for  $A$  to be zero and  $B$  to have unity elements, we could use the following commands.

Action	F90	MATLAB
Define size	integer :: A ( 2, 3 ) integer :: B ( 3 )	A(2,3)=0; B(3)=0;
Zero A	A=0	A=zeros(2,3);
Initialize B	B=1	B=ones(3);

If we want to create a new array  $B$  with the first three even numbers, we would use *implied loops*.

Action	F90	MATLAB
Even set	B=(/(2*k,k=1,3)/) B=(/(k,k=2,6,2)/)	B=2*[1:1:3]; B=[2:2:6];

Arrays can also be initialized by reading their element values from a stored data file. The two most common types of files are ASCII (standard characters) and binary (machine language) files. ASCII files are easy to read and edit, but binary files make more efficient use of storage, and are read or written much faster than ASCII files. ASCII files are often denoted by the name extension of “dat”. Binary files are denoted by the name extension “mat” in MATLAB, while in Fortran the extension “bin” is commonly employed.

For example, assume that the above  $A(2,3)$  array is to be initialized by reading its values from an ASCII file created by a text editor and given the name of `A.dat`. Further, assume that we wish to multiply all elements by 3 and store it as a new ASCII file. Then we could use `read` procedures like those in Table 8.5 where the last MATLAB command associated a file name and a file type with the desired input/output (I/O) action. Fortran requires an `OPEN` statement to do this if the default I/O files (unit 5 to read and unit 6 to write) are not used in the read or write.

## 8.1.2 Intrinsic Array Functions

Note that MATLAB has intrinsic functions `ones` and `zeros` to carry out a task that F90 does with an operator. Often the reverse is true. MATLAB has several operators that in Fortran correspond to an intrinsic function or a `CALLED` function. A comparison of the similar F90 and MATLAB array mathematical operators are given in Table 8.5. They generally only differ slightly in syntax. For example, to transpose the matrix  $A$ , the F90 construct is `transpose(A)` while in MATLAB it's simply  $A'$ .<sup>†</sup> In F90, the `*` operator means, for matrices, term by term multiplication: when  $A = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$  and  $B = \begin{bmatrix} 1 & 2 & 4 \\ 3 & 5 & 6 \end{bmatrix}$ ,  $A*B$  yields  $\begin{bmatrix} 1 & 6 & 20 \\ 6 & 20 & 36 \end{bmatrix}$ . In MATLAB, the same operation is expressed as  $A.*B$ . To multiply the matrices  $A$  and  $B$ , Fortran requires the use of the intrinsic function `matmul` (i.e., `matmul(A,B)`) while MATLAB uses the `*` operator ( $A*B$ ).

Another group of commonly used functions that operate on arrays in Fortran90 and MATLAB are briefly described in Table 8.6. Both languages have several other functions of a more specialized nature, but those in Table 8.6 are probably the most commonly used in programs.

Often one needs to truncate a real number in some special fashion. Table 8.7 illustrates how to do that using some of the functions common to the languages of interest. That table also implies how one can convert reals to integers and vice versa.

## 8.1.3 Colon Operations on Arrays (Subscript Triplet)

The syntax of the colon operator, which is available in MATLAB and F90, is detailed in Table 4.6. What the colon operator concisely expresses is a sequence of numbers in an arithmetic progression. As shown in the table, the MATLAB expression  $B:I:E$  expresses the sequence  $B, B+I, B+2*I, \dots, B + \lfloor \frac{E-B}{I} \rfloor I$ . The complicated expression for the sequence's last term simply means that the last value of the sequence does not exceed (in magnitude) the end value  $E$ .

<sup>†</sup>In MATLAB,  $A'$  actually means conjugate transpose. If  $A$  is real, this operator performs the transpose as desired. If  $A$  is complex and we want its transpose, the MATLAB construct is  $A.'$

<i>Description</i>	<i>Equation</i>	<i>Fortran90 Operator</i>	<i>Matlab Operator</i>	<i>Original Sizes</i>	<i>Result Size</i>
Scalar plus scalar	$c = a \pm b$	$c = a \pm b$	$c = a \pm b;$	1, 1	1, 1
Element plus scalar	$c_{jk} = a_{jk} \pm b$	$c = a \pm b$	$c = a \pm b;$	$m, n$ and 1, 1	$m, n$
Element plus element	$c_{jk} = a_{jk} \pm b_{jk}$	$c = a \pm b$	$c = a \pm b;$	$m, n$ and $m, n$	$m, n$
Scalar times scalar	$c = a \times b$	$c = a * b$	$c = a * b;$	1, 1	1, 1
Element times scalar	$c_{jk} = a_{jk} \times b$	$c = a * b$	$c = a * b;$	$m, n$ and 1, 1	$m, n$
Element times element	$c_{jk} = a_{jk} \times b_{jk}$	$c = a * b$	$c = a. * b;$	$m, n$ and $m, n$	$m, n$
Scalar divide scalar	$c = a/b$	$c = a/b$	$c = a/b;$	1, 1	1, 1
Scalar divide element	$c_{jk} = a_{jk}/b$	$c = a/b$	$c = a/b;$	$m, n$ and 1, 1	$m, n$
Element divide element	$c_{jk} = a_{jk}/b_{jk}$	$c = a/b$	$c = a./b;$	$m, n$ and $m, n$	$m, n$
Scalar power scalar	$c = a^b$	$c = a**b$	$c = a \wedge b;$	1, 1	1, 1
Element power scalar	$c_{jk} = a_{jk}^b$	$c = a**b$	$c = a \wedge b;$	$m, n$ and 1, 1	$m, n$
Element power element	$c_{jk} = a_{jk}^{b_{jk}}$	$c = a**b$	$c = a. \wedge b;$	$m, n$ and $m, n$	$m, n$
Matrix transpose	$C_{kj} = A_{jk}$	$C = \text{transpose}(A)$	$C = A';$	$m, n$	$n, m$
Matrix times matrix	$C_{ij} = \sum_k A_{ik} B_{kj}$	$C = \text{matmul}(A, B)$	$C = A * B;$	$m, r$ and $r, n$	$m, n$
Vector dot vector	$c = \sum_k A_k B_k$	$c = \text{sum}(A * B)$ $c = \text{dot\_product}(A, B)$	$c = \text{sum}(A. * B);$ $c = A * B';$	$m, 1$ and $m, 1$ $m, 1$ and $m, 1$	1, 1 1, 1

**Table 8.5:** Array Operations in Programming Constructs. Lower case letters denote scalars or scalar elements of arrays. Matlab arrays are allowed a maximum of two subscripts while Fortran allows seven. Upper case letters denote matrices or scalar elements of matrices.

You can also use the colon operator to extract smaller arrays from larger ones. If we wanted to extract the second row and third column of the array,  $A = \begin{bmatrix} 1 & 7 & -2 \\ 3 & 4 & 6 \end{bmatrix}$ , to get, respectively,

$$G = [3 \ 4 \ 6], \quad C = \begin{Bmatrix} -2 \\ 6 \end{Bmatrix},$$

we could use the colon operator as follows.

Action	F90	MATLAB
Define size	integer :: B (3) integer :: C (2)	B(3)=0; C(2)=0;
Extract row	B=A(2, :)	B=A(2, :);
Extract columns	C=A(:, 3)	C=A(:, 3);

**Table 8.6:** Equivalent Fortran90 and MATLAB Intrinsic Functions.

The following KEY symbols are utilized to denote the TYPE of the intrinsic function, or subroutine, and its arguments: A-complex, integer, or real; I-integer; L-logical; M-mask (logical); R-real; X-real; Y-real; V-vector (rank 1 array); and Z-complex. Optional arguments are not shown. Fortran90 and MATLAB also have very similar array operations and colon operators.

Type	Fortran90	MATLAB	Brief Description
A	ABS(A)	abs(a)	Absolute value of A.
R	ACOS(X)	acos(x)	Arc cosine function of real X.
R	AIMAG(Z)	imag(z)	Imaginary part of complex number.
R	AINIT(X)	real(fix(x))	Truncate X to a real whole number.
L	ALL(M)	all(m)	True if all mask elements, M, are true.
R	ANINT(X)	real(round(x))	Real whole number nearest to X.
L	ANY(M)	any(m)	True if any mask element, M, is true.
R	ASIN(X)	asin(x)	Arcsine function of real X.
R	ATAN(X)	atan(x)	Arctangent function of real X.
R	ATAN2(Y,X)	atan2(y,x)	Arctangent for complex number(X, Y).
I	CEILING(X)	ceil(x)	Least integer $\geq$ real X.
Z	CMPLX(X,Y)	(x+yi)	Convert real(s) to complex type.
Z	CONJG(Z)	conj(z)	Conjugate of complex number Z.
R	COS(R_Z)	cos(r_z)	Cosine of real or complex argument.
R	COSH(X)	cosh(x)	Hyperbolic cosine function of real X.
I	COUNT(M)	sum(m==1)	Number of true mask, M, elements.
R,L	DOT_PRODUCT(X,Y)	x'*y	Dot product of vectors X and Y.
R	EPSILON(X)	eps	Number, like X, $\ll 1$ .
R,Z	EXP(R_Z)	exp(r_z)	Exponential of real or complex number.
I	FLOOR(X)	floor	Greatest integer $\leq$ X.
R	HUGE(X)	realmax	Largest number like X.
I	INT(A)	fix(a)	Convert A to integer type.
R	LOG(R_Z)	log(r_z)	Logarithm of real or complex number.
R	LOG10(X)	log10(x)	Base 10 logarithm function of real X.
R	MATMUL(X,Y)	x*y	Conformable matrix multiplication, X*Y.
I,V	I=MAXLOC(X)	[y,i]=max(x)	Location(s) of maximum array element.
R	Y=MAXVAL(X)	y=max(x)	Value of maximum array element.
I,V	I=MINLOC(X)	[y,i]=min(x)	Location(s) of minimum array element.
R	Y=MINVAL(X)	y=min(x)	Value of minimum array element.

(continued)

Type	Fortran90	MATLAB	Brief Description
I	NINT(X)	round(x)	Integer nearest to real X.
A	PRODUCT(A)	prod(a)	Product of array elements.
call	RANDOM_NUMBER(X)	x=rand	Pseudo-random numbers in (0, 1).
call	RANDOM_SEED	rand('seed')	Initialize random number generator.
R	REAL (A)	real(a)	Convert A to real type.
R	RESHAPE(X, (/ I, I2 /))	reshape(x,i,i2)	Reshape array X into I×I2 array.
I,V	SHAPE(X)	size(x)	Array (or scalar) shape vector.
R	SIGN(X,Y)		Absolute value of X times sign of Y.
R	SIGN(0.5,X)-SIGN(0.5,-X)	sign(x)	Signum, normalized sign, -1, 0, or 1.
R,Z	SIN(R_Z)	sin(r_z)	Sine of real or complex number.
R	SINH(X)	sinh(x)	Hyperbolic sine function of real X.
I	SIZE(X)	length(x)	Total number of elements in array X.
R,Z	SQRT(R_Z)	sqrt(r_z)	Square root, of real or complex number.
R	SUM(X)	sum(x)	Sum of array elements.
R	TAN(X)	tan(x)	Tangent function of real X.
R	TANH(X)	tanh(x)	Hyperbolic tangent function of real X.
R	TINY(X)	realmin	Smallest positive number like X.
R	TRANPOSE(X)	x'	Matrix transpose of any type matrix.
R	X=1	x=ones(length(x))	Set all elements to 1.
R	X=0	x=zero(length(x))	Set all elements to 0.

For more detailed descriptions and example uses of these intrinsic functions see Adams, J.C., *et al.*, *Fortran 90 Handbook*, McGraw-Hill, New York, 1992, ISBN 0-07-000406-4.

C++	-	int	-	-	floor	ceil
F90	aint	int	aint	nint	floor	ceiling
MATLAB	real (fix)	fix	real (round)	round	floor	ceil
Argument	Value of Result					
-2.000	-2.0	-2	-2.0	-2	-2	-2
-1.999	-1.0	-1	-2.0	-2	-2	-1
-1.500	-1.0	-1	-2.0	-2	-2	-1
-1.499	-1.0	-1	-1.0	-1	-2	-1
-1.000	-1.0	-1	-1.0	-1	-1	-1
-0.999	0.0	0	-1.0	-1	-1	0
-0.500	0.0	0	-1.0	-1	-1	0
-0.499	0.0	0	0.0	0	-1	0
0.000	0.0	0	0.0	0	0	0
0.499	0.0	0	0.0	0	0	1
0.500	0.0	0	1.0	1	0	1
0.999	0.0	0	1.0	1	0	1
1.000	1.0	1	1.0	1	1	1
1.499	1.0	1	1.0	1	1	2
1.500	1.0	1	2.0	2	1	2
1.999	1.0	1	2.0	2	1	2
2.000	2.0	2	2.0	2	2	2

**Table 8.7:** Truncating Numbers

WHERE (logical_array_expression)
true_array_assignments
ELSEWHERE
false_array_assignments
END WHERE
WHERE (logical_array_expression)
true_array_assignment

**Table 8.8:** F90 WHERE Constructs

One can often use colon operators to avoid loops acting on arrays to define new arrays. For example, consider a square matrix

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}.$$

We can flip it left to right to create a new matrix (in F90 syntax)

$$B = A(:, n:1:-1) = \begin{bmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \\ 9 & 8 & 7 \end{bmatrix}$$

or flip it up to down

$$C = A(n:1:-1, :) = \begin{bmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{bmatrix}$$

or flip it up to down, then left to right

$$D = A(n:1:-1, n:1:-1) = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix},$$

where  $n = 3$  is the number of rows in the matrix  $A$ . In the MATLAB syntax, the second and third numbers would be interchanged in the colon operator. Actually, MATLAB has intrinsic operators to flip the matrices so that one could simply write

```
B = fliplr(A); C = flipud(A); D = rot90(A);
```

### 8.1.4 Array Logical Mask Operators

By default most MATLAB commands are designed to operate on arrays. Fortran77 and C++ have no built in array operations and it is necessary to program each loop. The Fortran90 standard has many of the MATLAB array commands and often with the identical syntax as shown in Table 8.5 and 8.6. Often the F90 versions of these functions have optional features (arguments) that give the user more control than MATLAB does by including a logical control mask to be defined shortly.

To emphasize that an IF type of relational operator is to act on all elements of an array, Fortran90 also includes an array WHERE block or statement control (that is, an IF statement acting on all array elements) which is outlined in Table 8.8.

Note that the necessary loops are implied and need not be written. As an example, if

$$A = \begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$



<i>Function</i>	<i>Description</i>	<i>Opt</i>	<i>Example</i>
all	Find if all values are true, for a fixed dimension.	d	all(B = A, DIM = 1) (true, false, false)
any	Find if any value is true, for a fixed dimension.	d	any (B > 2, DIM = 1) (false, true, true)
count	Count number of true elements for a fixed dimension.	d	count(A = B, DIM = 2) (1, 2)
maxloc	Locate first element with maximum value given by mask.	m	maxloc(A, A < 9) (2, 3)
maxval	Max element, for fixed dimension, given by mask.	b	maxval (B, DIM=1, B > 0) (2, 4, 6)
merge	Pick true array, A, or false array, B, according to mask, L.	—	merge(A, B, L) $\begin{bmatrix} 0 & 3 & 5 \\ 2 & 4 & 8 \end{bmatrix}$
minloc	Locate first element with minimum value given by mask.	m	minloc(A, A > 3) (2, 2)
minval	Min element, for fixed dimension, given by mask.	b	minval(B, DIM = 2) (1, 2)
pack	Pack array, A, into a vector under control of mask.	v	pack(A, B < 4) (0, 7, 3)
product	Product of all elements, for fixed dimension, controlled by mask.	b	product(B) ; (720) product(B, DIM = 1, T) (2, 12, 30)
sum	Sum all elements, for fixed dimension, controlled by mask.	b	sum(B) ; (21) sum(B, DIM = 2, T) (9, 12)
unpack	Replace the true locations in array B controlled by mask L with elements from the vector U.	—	unpack(U, L, B) $\begin{bmatrix} 7 & 3 & 8 \\ 2 & 4 & 9 \end{bmatrix}$

$$A = \begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}, \quad L = \begin{bmatrix} T & F & T \\ F & F & T \end{bmatrix}, \quad U = (7, 8, 9)$$

**Table 8.9:** F90 Array Operators with Logic Mask Control. *T* and *F* denote true and false, respectively. Optional arguments: b -- DIM & MASK, d -- DIM, m -- MASK, v -- VECTOR and DIM = 1 implies for any rows, DIM = 2 for any columns, and DIM = 3 for any plane.

then, WHERE (A > B) B = A gives a new  $B = \begin{bmatrix} 1 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$ . By default, MATLAB always acts on matrices and considers scalars a special case. Thus, it would employ the standard syntax, if A > B, B=A, to do the same task.

A more sophisticated way to selectively pick subscripts of an array is to use a *mask* array. A mask array is the same size and shape as the array on which it will act. It is a Boolean array: All its elements have either true or false values. When associated with an operator, the operator will only act on those elements in the original array whose corresponding mask location is true (i.e., .true. in Fortran, true in C++ and 1 in MATLAB and C). Fortran90 has several operations that allow or require masks (Table 8.9). MATLAB functions with the same name exist in some cases, as seen in Table 8.6. Usually, they correspond to the F90 operator where the mask is true everywhere.

ALL	ANY	COUNT
CSHIFT	DOT_PRODUCT	EOSHIFT
MATMUL	MAXLOC	MAXVAL
MINLOC	MINVAL	PACK
PRODUCT	REPEAT	RESHAPE
SPREAD	SUM	TRANSFER
TRANSPOSE	TRIM	UNPACK

**Table 8.10:** Intrinsic Functions Allowing Logical Mask Control

A general Fortran principle underlies the fact that the array mentioned in the `WHERE` mask may be changed within the `WHERE` construct. When an array appears in the `WHERE` statement mask, the logical test is executed **first** and the host system retains the result independent of whatever happens later inside the `WHERE` construct. Thus, in the program fragment

```
integer, parameter :: n = 5
real :: x(n) = (/ (k, k = 1, n) /)
where (x > 0.0)
  x = -x
end where
```

the sign is reversed for all elements of `x` because they all pass the initial logical mask. It is as if a classic `DO` sequence had been programmed

```
do i = 1, n, 1
  if (x(i) > 0.0) x(i) = -x(i)
end do
```

instead of the `WHERE` construct.

A more ominous and subtle issue surrounds the use of other transformational intrinsic functions listed in Table 8.10. The danger is that when these intrinsics appear inside the body of a `WHERE` construct, the `WHERE` statement's initial mask may no longer apply. Hence, in the following example the transformational intrinsic function `SUM` operates over all five elements of `x` rather than just the two elements of `x` that exceed six.

```
integer, parameter :: n = 5
real :: x(n) = (/ 2, 4, 6, 8, 10 /)
where (x > 6.0)
  x = x / sum(x)
end where
```

Thus, the new values for `x` are  $\{2, 4, 6, 8/30, 10/30\}$  rather than  $\{2, 4, 6, 8/18, 10/18\}$ . This standard-conforming, but otherwise “unexpected”, result should raise a caution for the programmer. If one did not want the above illustrated result, then it would be necessary to use the same mask of the `WHERE` as an optional argument to `SUM`: `sum(x, mask = x > 6.0)`. A lot of care needs to be taken to assure that transformational intrinsics that appear in a `WHERE` construct use exactly the same mask.

### 8.1.5 User Defined Operators

In addition to the many intrinsic operators and functions we have seen so far, the F90 user can also define new operators or extend existing ones. User defined operators can employ intrinsic data types and/or user defined data types. The user defined operators, or extensions, can be unary or binary (i.e., have one or two arguments). The operator symbol must be included between two periods, such as `‘.op.’`. As an example, consider a program to be used to create a shorthand notation to replace the standard F90 matrix transpose and matrix multiplication functions so that we could write

	<code>B = .t. A</code>
	<code>C = B .x. D</code>
or	<code>C = (.t.A) .x. D</code>
instead of	<code>B = TRANSPOSE(A)</code>
	<code>C = MATMUL (B, D)</code>
or	<code>C = MATMUL(TRANSPOSE (A), D)</code>

Operator	Action	Use	Algebra
.t.	transpose	.t.A	$A^T$
.x.	multiplication	A.x.B	$AB$
.i.	inverse of matrix	.i.A	$A^{-1}$
.ix.	solution	A.ix.B	$A^{-1}B$
.tx.	transpose times matrix	A.tx.B	$A^T B$
.xt.	matrix times transpose	A.xt.B	$AB^T$
.eye.	identity matrix	.eye.N	$I, N \times N$

**Table 8.11:** Definitions in Matrix Operators.

To do this, one must have a `MODULE PROCEDURE` to define the operator actions for all envisioned (and incorrect) inputs and an `INTERFACE OPERATOR` that informs F90 what your operation symbol is.

Fig. 8.1 illustrates the code that would partially define the operator `.t.`. Note that while `TRANSPOSE` accepts any type of matrix of any rank, our operator works only for real or integer rectangular arrays (of rank 2). It would not transpose `LOGICAL` arrays or vectors. That oversight can be extended by adding more functions to the interface.

If one works with matrices often, then one may want to define your own library of matrix operators. Such operators are not standard in F90 as they are in `MATLAB`, but can be easily added. To provide a foundation for such a library, we provide a `Matrix_Operators` module with the operators defined in Table 8.11. The reader is encouraged to expand the initial support provided in that module.

### 8.1.6 Connectivity Lists and Vector Subscripts

When using an array with constant increments in its subscripts, we usually provide its subscript in the form of a colon operator or a control variable in a `DO` or `FOR` loop. In either case, the array subscripts are integers. There are several practical programming applications where the required subscripts are not known in advance. Typically, this occurs when we are dealing with an assemblage of components that can be connected together in an arbitrary fashion by the user (e.g., electric circuits, truss structures, volume elements in a solid model). To get the subscripts necessary to build the assemblage we must read an integer data file that lists the junction numbers to which each component is attached. We call those data a *connectivity file*. If we assume each component has the same number of junction points, then the list can be input as a two-dimensional array. One subscript will range over the number of components and the other will range over the number of possible junctions per component. For ease of typing these data, we usually assume that the  $k^{\text{th}}$  row of the array contains the integer junction, or connection, points of that component. Such a row of connectivity data is often used in two related operations: `gather` and `scatter`. A `gather` operation uses the lists of connections to gather or collect information from the assembly necessary to describe the component or its action. The `scatter` operation has the reverse effect. It takes information about the component and sends it back to the assembly. Usually, values from the component are added into corresponding junction points of the assembly.

The main point of this discussion is that another way to define a non-sequential set of subscripts is to use an integer vector array that contains the set. Then one can use the array name as a way to range over the subscripts. This is a compact way to avoid an additional `FOR` or `DO` loop. The connectivity list for a component is often employed to select the subscripts needed for that component.

To illustrate the concept of vector subscripts, we will repeat the array flip example shown in §8.1.3 via the colon operators. Here we will define an integer vector called `Reverse` that has constant increments to be used in operating on the original array `A`. By using the vector name as a subscript, it automatically invokes an implied loop over the contents of that vector. As shown in Figure 8.2, this has the same effect as employing the colon operator directly.

The real power of the vector subscripts comes in the case where it has integers in a random, or user input, order rather than in an order that has a uniform increment. For example, if we repeat the above example using a vector `Random=[ 3 1 2 ]`, then both `MATLAB` and F90 would give the result

```

[ 1] MODULE Ops_Example    ! User defined matrix transpose example
[ 2]
[ 3] IMPLICIT NONE
[ 4] INTERFACE OPERATOR (.t.)    ! transpose operator
[ 5]   MODULE PROCEDURE Trans_R, Trans_I ! for real or integer matrix
[ 6]   ! Remember to add logicals and vectors later
[ 7] END INTERFACE              ! defining .t.
[ 8]
[ 9] CONTAINS ! the actual operator actions for argument types
[10]
[11] FUNCTION Trans_R ( A ) ! defines .t. for real rank 2 matrix
[12]   REAL, DIMENSION(:,:), INTENT(IN)    :: A
[13]   REAL, DIMENSION(SIZE(A,2), SIZE(A,1)) :: Trans_R
[14]   Trans_R = TRANSPOSE (A)
[15] END FUNCTION Trans_R ! for real rank 2 transpose via .t.
[16]
[17] FUNCTION Trans_I ( A ) ! defines .t. for integer rank 2 matrix
[18]   INTEGER, DIMENSION(:,:), INTENT(IN)  :: A
[19]   INTEGER, DIMENSION(SIZE(A,2), SIZE(A,1)) :: Trans_I
[20]   Trans_I = TRANSPOSE (A)
[21] END FUNCTION Trans_I ! for integer rank 2 transpose via .t.
[22]
[23] END MODULE Ops_Example    ! User defined matrix transpose example
[24]
[25] PROGRAM Demo_Trans      ! illustrate the .t. operator
[26]   USE Ops_Example        ! module with user definitions
[27]   IMPLICIT NONE
[28]   INTEGER, PARAMETER    :: M = 3, N = 2 ! rows, columns
[29]   REAL, DIMENSION(M,N) :: A ; REAL, DIMENSION(N,M) :: B
[30]
[31]   ! define A, test operator, print results
[32]   A = RESHAPE ( ( / ((I*J , I=1,M), J=1,N) / ), SHAPE(A) )
[33]   B = .t. A
[34]   PRINT *, 'MATRIX A' ; CALL M_print (A, M, N)
[35]   PRINT *, 'MATRIX B' ; CALL M_print (B, N, M)
[36]   ! Produces the result:
[37]   ! MATRIX A
[38]   ! RC      1      2
[39]   ! 1      1.000      2.000
[40]   ! 2      2.000      4.000
[41]   ! 3      3.000      6.000
[42]   !
[43]   ! MATRIX B
[44]   ! RC      1      2      3
[45]   ! 1      1.000      2.000      3.000
[46]   ! 2      2.000      4.000      6.000
[47] END PROGRAM Demo_Trans

```

**Figure 8.1:** Creating and applying user defined operators

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad \text{Reverse} = [321]$$

Flip left to right:

$$B = A(:, \text{Reverse}) = \begin{bmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \\ 9 & 8 & 7 \end{bmatrix}$$

Flip up to down:

$$C = A(\text{Reverse}, :) = \begin{bmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{bmatrix}$$

Flip up to down, left to right:

$$D = A(\text{Reverse}, \text{Reverse}) = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

**Figure 8.2:** F90 and MATLAB Vector Subscripts and Array Shifts.

```

five = (/ 1 2 3 4 5 /)
! without a pad
three = eoshift(five,2) != (/ 3 4 5 0 0 /)
three = eoshift(five,-2) != (/ 0 0 1 2 3 /)
! with a pad
pad = eoshift(five,2,9) != (/ 3 4 5 9 9 /)
pad = eoshift(five,-2,9) != (/ 9 9 1 2 3 /)

```

**Figure 8.3:** F90 end-off shift (eoshift) intrinsic.

```

five = (/ 1 2 3 4 5 /)
left_3 = cshift(five,3) != (/ 3 4 5 1 2 /)
right_3 = cshift(five,-3) != (/ 4 5 1 2 3 /)

```

**Figure 8.4:** F90 Circular shift (cshift) intrinsic.

$$E = A(:, \text{Random}) = \begin{bmatrix} 3 & 1 & 2 \\ 6 & 4 & 5 \\ 9 & 7 & 8 \end{bmatrix}.$$

While the `reshape` option of F90 and MATLAB allows the array elements to change from one rectangular storage mode to another, one can also move elements around in the fixed shape array by utilizing the colon operators, or by the use of “shift operators.” The latter accept an integer to specify how many locations to move or shift an element. A positive number moves an element up a column, a negative value moves it down the column, and a zero leaves it unchanged. The elements that are moved out of the array either move from the head of the queue to the tail of the queue (called a “circular shift”) or are replaced by a user specified “pad” value (called an “end off shift”). If no pad is given, its value defaults to zero. These concepts are illustrated for F90 in Figures 8.3 and 8.4.

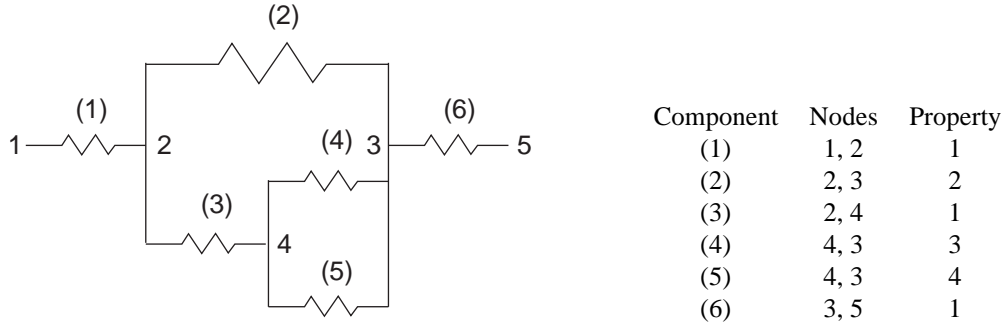
### 8.1.7 Component Gather and Scatter

Often the equations governing a system balance principle are assembled from the relative contributions of each component. When the answers for a complete system have been obtained, it is then possible to recover the response of each component. The automation of these processes has six basic requirements:

1. a component balance principle written in matrix form,
2. a joint connectivity data list that defines where a given component type connects into the system,
3. a definition of a `scatter` operator that scatters the coefficients of the component matrices into corresponding locations in the governing system equations,
4. an efficient system equation solver,
5. a `gather` operator to gather the answers from the system for those joints connected to a component, and
6. a recovery of the internal results in the component.

The first of these is discipline-dependent. We are primarily interested in the gather-scatter operations. These are opposites that both depend on the component connectivity list, which is often utilized as a vector subscript. The number of rows in the component equations is less than the number of rows in the assembled system, except for the special case where the system has only a single component. Thus, it is the purpose of the gather-scatter operators to define the relation between a system row number and a particular component row number. That is, they define the relation that defines the subset of component unknowns, say  $\mathbf{V}^e$  for component  $e$ , in terms of all the system unknowns, say  $\mathbf{V}$ :  $\mathbf{V}^e \subset_e \mathbf{V}$ . Here the containment  $\subset$  is defined by the component’s connection list and the number of unknowns per joint. If there is only one unknown per joint, then the subset involves only the connection list. The above process gathers the subset of component unknowns from the full set of system unknowns.

Let the list of joints or nodes connected to the component be called  $\mathbf{L}^e$ . The  $k^{\text{th}}$  member in this list contains the corresponding system node number,  $\kappa$ : i.e.  $\kappa = \mathbf{L}_e(k)$ . Thus, for a single unknown per



**Figure 8.5:** Example Circuit or Axial Spring System

joint, one simply has  $\mathbf{V}^e = \mathbf{V}(\mathbf{L}_e) \subset_e \mathbf{V}$ . Written in full loop form, the component gather operation would be

```
DO k = 1, size(L_e)
  V_e(k) = V(L_e(k))
END DO ! OVER LOCAL JOINTS
```

while in F90 or MATLAB vector subscript form, it is simply  $\mathbf{v}_e = \mathbf{v}(\mathbf{L}_e)$ , for a single unknown per joint. When there is more than one unknown per joint, the relation can be written in two ways.

We pick the one that counts (assigns equation numbers to) all unknowns at a joint before going on to the next joint. Let the number of unknowns per joint be  $N$ . Then by deduction, one finds that the equation number for the  $j$ -th unknown at the  $K^{\text{th}}$  system node is

$$E(K, j) = N * (K - 1) + j, \quad 1 \leq j \leq N.$$

But to find which equation numbers go with a particular component, we must use the connection list  $\mathbf{L}_e$ . For the  $k^{\text{th}}$  local node,  $K = \mathbf{L}_e(k)$  and

$$E(k, j) = N * (\mathbf{L}_e(k) - 1) + j, \quad 1 \leq j \leq N.$$

If we loop over all nodes on a component, we can build an index list, say  $\mathbf{I}_e$ , that tells which equations relate to the component.

```
INTEGER, ALLOCATABLE :: I_e(:), V_e(:)
ALLOCATE(I_e(N * SIZE(L_e)), V_e(N*SIZE(L_e)))
DO k = 1, SIZE(L_e) ! component nodes
  DO j = 1, N ! unknowns per node
    LOCAL = N * (k-1) + j
    SYSTEM = N * (L_e(k) - 1) + j
    I_e(LOCAL) = SYSTEM
  END DO ! on unknowns
END DO ! on local nodes.
```

Therefore, the generalization of the component gather process is

```
DO m = 1, SIZE(I_e)
  V_e(m) = V(I_e(m))
END DO ! over local unknowns
```

or in vector subscript form  $\mathbf{v}_e = \mathbf{v}(\mathbf{I}_e)$  for an arbitrary number of unknowns per joint.

To illustrate the scatter concept, consider a system shown in Figure 8.5, which has six components and five nodes. If there is only one unknown at each joint (like voltage or axial displacement), then the system equations will have five rows. Since each component is connected to two nodes, each will contribute to (scatter to) two of the system equation rows. Which two rows? That is determined by the connection list shown in the figure. For example, component (4) is joined to nodes 4 and 3. Thus, the coefficients in the first row of the local component balance law would scatter into (be added to) the fourth row of the system, while the second row of the component would scatter to the third system equation row. If the component balance law is symmetric, then the columns locations scatter in the same fashion.

## 8.2 Matrices

Matrices are very commonly used in many areas of applied mathematics and engineering. While they can be considered a special case of the subscripted arrays given above they have their own special algebra and calculus notations that are useful to know. In the following sections we will describe matrices and the intrinsic operations on them that are included in F90 and MATLAB. Neither C nor C++ have such useful intrinsics, but require the programmer to develop them or extract them from a special library.

A *matrix* is defined as a rectangular array of quantities arranged in rows and columns. The array is enclosed in brackets, and thus if there are  $m$  rows and  $n$  columns, the matrix can be represented by

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1j} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2j} & \cdots & a_{2n} \\ \vdots & & & & & & \vdots \\ a_{i1} & a_{i2} & a_{i3} & \cdots & a_{ij} & \cdots & a_{in} \\ \vdots & & & & & & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mj} & \cdots & a_{mn} \end{bmatrix} = [\mathbf{A}] \quad (8.1)$$

where the typical element  $a_{ij}$  has two subscripts, of which the first denotes the row ( $i^{\text{th}}$ ) and where the second denotes the column ( $j^{\text{th}}$ ) which the element occupies in the matrix. A matrix with  $m$  rows and  $n$  columns is defined as a matrix of order  $m \times n$ , or simply an  $m \times n$  matrix. The number of rows is always specified first. In Equation 8.1, the symbol  $\mathbf{A}$  stands for the matrix of  $m$  rows and  $n$  columns, and it is usually printed in **boldface** type. If  $m = n = 1$ , then the matrix is equivalent to a scalar. If  $m = 1$ , the matrix  $\mathbf{A}$  reduces to the single row

$$\mathbf{A} = [a_{11} \ a_{12} \ a_{13} \ \cdots \ a_{1j} \ \cdots \ a_{1n}] = (\mathbf{A})$$

which is called a *row matrix*. Similarly, if  $n = 1$ , the matrix  $\mathbf{A}$  reduces to the single column

$$\mathbf{A} = \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} = \text{col}[a_{11} \ a_{21} \ \cdots \ a_{m1}] = \{\mathbf{A}\}$$

which is called a *column matrix*, or vector. When all the elements of matrix are equal to zero, the matrix is called *null* or *zero* and is indicated by  $\mathbf{0}$ . A null matrix serves the same function as zero does in ordinary algebra. To set all the elements of  $\mathbf{A}$  to zero, one writes  $A = 0$  in F90, and  $A = \text{zeros}[m, n]$  in MATLAB.

If  $m = n$ , the matrix is said to be *square*.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

Before considering some of the matrix algebra implied by the above equation, a few other matrix types need definition. A *diagonal matrix* is a square matrix which has zero elements outside the principal diagonal. It follows, therefore, that for a diagonal matrix  $a_{ij} = 0$  when  $i \neq j$ , and not all  $a_{ii}$  are zero. A typical diagonal matrix may be represented by

$$\mathbf{A} = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & & & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix},$$

or more concisely as  $\mathbf{A} = \text{diag}[a_{11} a_{22} \cdots a_{nn}]$ .

A *unit* or *identity* matrix is a diagonal matrix whose elements are equal to 0 except those located on its main diagonal, which are equal to 1. That is,  $a_{ij} = 1$  if  $i = j$ , and  $a_{ij} = 0$  if  $i \neq j$ . The unit matrix will be given the symbol **I** throughout these notes. An example of a  $3 \times 3$  unit matrix is

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \text{diag}[1 \ 1 \ 1].$$

A *Toeplitz* matrix has constant-valued diagonals. An identity matrix is Toeplitz as is the following matrix.

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 3 & 5 \\ 4 & 1 & -2 & 3 \\ -1 & 4 & 1 & -2 \\ 10 & -1 & 4 & 1 \end{bmatrix}$$

Note how the values of a Toeplitz matrix's elements are determined by the first row and the first column. MATLAB uses the Toeplitz function to create this unusual matrix.

A *symmetric matrix* is a square matrix whose elements  $a_{ij} = a_{ji}$  for all  $i, j$ . For example,

$$\mathbf{A} = \begin{bmatrix} 12 & 2 & -1 \\ 2 & 33 & 0 \\ -1 & 0 & 15 \end{bmatrix}$$

is symmetric: The first row equals the first column, the second row the second column, etc.

An *antisymmetric* or *skew symmetric* matrix is a square matrix whose elements  $a_{ij} = -a_{ji}$  for all  $i, j$ . Note that this condition means that the diagonal values of an antisymmetric matrix must equal zero. An example of such a matrix is

$$\mathbf{A} = \begin{bmatrix} 0 & 2 & -1 \\ -2 & 0 & 10 \\ 1 & -10 & 0 \end{bmatrix}$$

The *transpose* of a matrix **A**, denoted by  $\mathbf{A}^T$ , is obtained by interchanging the rows and columns. Thus, the transpose of an  $m \times n$  matrix is an  $n \times m$  matrix. For example,

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 3 & 5 \\ 0 & 1 \end{bmatrix} \quad \mathbf{A}^T = \begin{bmatrix} 2 & 3 & 0 \\ 1 & 5 & 1 \end{bmatrix}.$$

In MATLAB an appended prime is used to denote the transpose of any matrix, such as  $\mathbf{B} = \mathbf{A}'$ , whereas in F90 we employ the intrinsic function  $\mathbf{B} = \text{transpose}(\mathbf{A})$ , or a user defined operator like  $\mathbf{B} = .\text{t}.\mathbf{A}$  which we defined earlier.

If all the elements on one side of the diagonal of a square matrix are zero, the matrix is called a *triangular* matrix. There are two types of triangular matrices: (1) an upper triangular **U**, whose elements below the diagonal are all zero, and (2) a lower triangular **L**, whose elements above the diagonal are all zero. An example of a lower triangular matrix is

$$\mathbf{L} = \begin{bmatrix} 10 & 0 & 0 \\ 1 & 3 & 0 \\ 5 & 1 & 2 \end{bmatrix}.$$

A matrix may be divided into smaller arrays by horizontal and vertical lines. Such a matrix is then referred to as a *partitioned matrix*, and the smaller arrays are called *submatrices*. For example, we can partition a  $3 \times 3$  matrix into four submatrices as shown:

$$\mathbf{A} = \left[ \begin{array}{cc|c} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ \hline a_{31} & a_{32} & a_{33} \end{array} \right] = \left[ \begin{array}{cc} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{array} \right] = \left[ \begin{array}{cc|c} 2 & 1 & 3 \\ 10 & 5 & 0 \\ \hline 4 & 6 & 10 \end{array} \right]$$



where, in the F90 and MATLAB colon notation;

$$\begin{aligned}\mathbf{A}_{11} &= \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 10 & 5 \end{bmatrix} = \mathbf{A}(1:2, 1:2) \\ \mathbf{A}_{12} &= \begin{bmatrix} a_{13} \\ a_{23} \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \end{bmatrix} = \mathbf{A}(1:2, 3) \\ \mathbf{A}_{21} &= \begin{bmatrix} a_{31} & a_{32} \end{bmatrix} = \begin{bmatrix} 4 & 6 \end{bmatrix} = \mathbf{A}(3, 1:2) \\ \mathbf{A}_{22} &= [a_{33}] = [10] = \mathbf{A}(3, 3)\end{aligned}$$

It should be noted that the elements of a partitioned matrix must be so ordered that they are compatible with the whole matrix  $\mathbf{A}$  and with each other. That is,  $\mathbf{A}_{11}$  and  $\mathbf{A}_{12}$  must have an equal number of rows. Likewise,  $\mathbf{A}_{21}$  and  $\mathbf{A}_{22}$  must have an equal number of rows. Matrices  $\mathbf{A}_{11}$  and  $\mathbf{A}_{21}$  must have an equal number of columns. Likewise, for  $\mathbf{A}_{12}$  and  $\mathbf{A}_{22}$ . Note that  $\mathbf{A}_{22}$  is a matrix even though it consists of only one element. Provided the general rules for matrix algebra are observed, the submatrices can be treated as if they were ordinary matrix elements.

### 8.2.1 Matrix Algebra

To define what addition and multiplication means for matrices, we need to define an *algebra* for arrays of numbers so that they become useful to us. Without an algebra, all we have is a sequence of definitions without the ability to manipulate what they mean!

Addition of two matrices of the same order is accomplished by adding corresponding elements of each matrix. The matrix addition  $\mathbf{C} = \mathbf{A} + \mathbf{B}$  (as we write it in F90 and MATLAB), where  $\mathbf{A}$ ,  $\mathbf{B}$  and  $\mathbf{C}$  are matrices of the *same* order  $m \times n$  can be indicated by the equation

$$c_{ij} = a_{ij} + b_{ij}, \quad 1 \leq i \leq m, \quad 1 \leq j \leq n$$

where  $c_{ij}$ ,  $a_{ij}$ , and  $b_{ij}$  are typical elements of the  $\mathbf{C}$ ,  $\mathbf{A}$ , and  $\mathbf{B}$  matrices, respectively. An example of matrix addition is

$$\begin{bmatrix} 3 & 0 & 1 \\ 2 & -1 & 2 \\ 1 & 1 & 1 \end{bmatrix} + \begin{bmatrix} -1 & 1 & -1 \\ 2 & 5 & 6 \\ -3 & 4 & 9 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 0 \\ 4 & -4 & 8 \\ -2 & 5 & 10 \end{bmatrix}.$$

Matrix subtraction,  $\mathbf{C} = \mathbf{A} - \mathbf{B}$ , is performed in a similar manner.

Matrix addition and subtraction are *associative* and *commutative*. That is, with the previous definitions for matrix addition and subtraction, grouping and ordering with respect to these operations does not affect the result.

$$\mathbf{A} \pm (\mathbf{B} \pm \mathbf{C}) = (\mathbf{A} \pm \mathbf{B}) \pm \mathbf{C} \quad \text{and} \quad \mathbf{C} \pm \mathbf{B} \pm \mathbf{A}$$

Multiplication of the matrix  $\mathbf{A}$  by a scalar  $c$  is defined as the multiplication of every element of the matrix by the scalar  $c$ . Thus, the elements of the product  $\mathbf{B} = c\mathbf{A}$  are given by  $b_{ij} = ca_{ij}$ , and is written as  $\mathbf{B} = \mathbf{C} * \mathbf{A}$  in both F90 and MATLAB. Clearly, scalar multiplication distributes over matrix addition.

We could define special multiplication in the somewhat boring way as the term by term product of two identical sized matrices:  $\mathbf{C} = \mathbf{AB} \implies c_{ij} = a_{ij}b_{ij}$ . This feature is allowed in both F90 and MATLAB where it is written as  $\mathbf{C} = \mathbf{A} * \mathbf{B}$ , and  $\mathbf{C} = \mathbf{A} . * \mathbf{B}$ , respectively. Although this definition might be useful in some applications, this choice for what multiplication means in our algebra does not give us much power. Instead, we define the matrix product  $\mathbf{C} = \mathbf{AB}$  to mean

$$c_{ij} = \sum_{k=1}^p a_{ik} b_{kj}, \quad 1 \leq i \leq m, \quad 1 \leq j \leq n.$$

$\mathbf{A}$  and  $\mathbf{B}$  can be multiplied together as *only* when the number of columns in  $\mathbf{A}$ ,  $p$ , equals the number of rows in  $\mathbf{B}$ . When this condition is fulfilled, the matrices  $\mathbf{A}$  and  $\mathbf{B}$  are said to be *conformable* for multiplication. Otherwise, matrix multiplication of two matrices cannot be defined. The product of two

conformable matrices  $\mathbf{A}$  and  $\mathbf{B}$  having orders  $m \times p$  and  $p \times n$ , respectively, yields an  $m \times n$  matrix  $\mathbf{C}$ . In MATLAB this is simply written as  $\mathbf{C} = \mathbf{A} * \mathbf{B}$ , where as in F90 one would use the intrinsic function  $\mathbf{C} = \text{matmul}(\mathbf{A}, \mathbf{B})$ , or a user defined operator such as  $\mathbf{C} = \mathbf{A} . \times . \mathbf{B}$  which we defined earlier.

The reason why this definition for matrix multiplication was chosen so that we can concisely represent a system of linear equations. The verbose form explicitly lists the equations.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= c_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n &= c_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3n}x_n &= c_3 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n &= c_n \end{aligned}$$

where the  $a_{ij}$ 's and  $c_i$ 's usually represent known coefficients and the  $x_i$ 's unknowns. To express these equations more precisely, we define matrices for each of these arrays of numbers and lay them out as a matrix-vector product equaling a vector.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ & & \vdots & & \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_n \end{bmatrix}$$

We thus obtain the more compact matrix form  $\mathbf{AX} = \mathbf{C}$ .  $\mathbf{A}$  represents the square matrix of coefficients,  $\mathbf{X}$  the vector (column matrix) of unknowns, and  $\mathbf{C}$  the vector of known quantities.

Matrix multiplication is associative and distributive. For example,

$$\begin{aligned} (\mathbf{AB})\mathbf{C} &= \mathbf{A}(\mathbf{BC}) \\ \mathbf{A}(\mathbf{B} + \mathbf{C}) &= \mathbf{AB} + \mathbf{AC} \end{aligned}$$

However, matrix multiplication is *not* commutative. In general,  $\mathbf{AB} \neq \mathbf{BA}$ . Consequently, the order in which matrix multiplication is specified is by no means arbitrary. Clearly, if the two matrices are not conformable, attempting to commute the product makes no sense (the matrix multiplication  $\mathbf{BA}$  is not defined). In addition, when the matrices are conformable so that either product makes sense (the matrices are both square and have the same dimensions, for example), the product cannot be guaranteed to commute. You should try finding a simple example that illustrates this point. When two matrices  $\mathbf{A}$  and  $\mathbf{B}$  are multiplied, the product  $\mathbf{AB}$  is referred to either as  $\mathbf{B}$  *premultiplied* by  $\mathbf{A}$ , or as  $\mathbf{A}$  *postmultiplied* by  $\mathbf{B}$ . When  $\mathbf{AB} = \mathbf{BA}$ , the matrices  $\mathbf{A}$  and  $\mathbf{B}$  are then said to be *commutable*. For example, the unit matrix  $\mathbf{I}$  commutes with any square matrix of the same order:  $\mathbf{AI} = \mathbf{IA} = \mathbf{A}$ .<sup>†</sup>

The process of matrix multiplication can also be extended to partitioned matrices, provided the individual products of submatrices are conformable for multiplication. For example, the multiplication

$$\mathbf{AB} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{bmatrix}$$

is possible provided the products  $\mathbf{A}_{11}\mathbf{B}_{11}$ ,  $\mathbf{A}_{12}\mathbf{B}_{21}$ , etc. are conformable. For this condition to be fulfilled, it is only necessary for the vertical partitions in  $\mathbf{A}$  to include a number of columns equal to the number of rows in the corresponding horizontal partitions in  $\mathbf{B}$ .

The transpose of a product of matrices equals  $(\mathbf{AB} \cdots \mathbf{YZ})^T = \mathbf{Z}^T \mathbf{Y}^T \cdots \mathbf{B}^T \mathbf{A}^T$ . As an example of matrix multiplication, let  $\mathbf{B} = \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix}$  and  $\mathbf{A} = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$ ; then

$$\mathbf{AB} = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 7 \\ 6 \end{bmatrix}$$

<sup>†</sup>This result is why  $\mathbf{I}$  is called the identity matrix: It is the identity element with respect to matrix multiplication.

$$\mathbf{B}^T \mathbf{A}^T = \begin{bmatrix} 3 & 1 & 2 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 7 & 6 \end{bmatrix}$$

### 8.2.2 Inversion

Every (non-singular) square matrix  $\mathbf{A}$  has an *inverse*, indicated by  $\mathbf{A}^{-1}$ , such that by definition the product  $\mathbf{A}\mathbf{A}^{-1}$  is a unit matrix  $\mathbf{I}$ . The reverse is also true:  $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ . Inverse matrices are very useful in the solution of simultaneous equations  $\mathbf{A}\mathbf{X} = \mathbf{C}$  such as above where  $\mathbf{A}$  and  $\mathbf{C}$  are known and  $\mathbf{X}$  is unknown. If the inverse of  $\mathbf{A}$  is known, the unknowns of the  $\mathbf{X}$  matrix can be (symbolically) found by premultiplying both sides of the equation by the inverse  $\mathbf{A}^{-1}\mathbf{A}\mathbf{X} = \mathbf{A}^{-1}\mathbf{C}$  so that

$$\mathbf{X} = \mathbf{A}^{-1}\mathbf{C}.$$

In this way, in theory we have “solved” our system of linear equations. To employ this approach, we must find the inverse of the matrix  $\mathbf{A}$ , which is not any easy task. Despite this computational difficulty, using matrix algebra to concisely express complicated linear combinations of quantities often provides much insight into a problem and its solution techniques.

Various methods can be used to determine the inverse of a given matrix. For very large systems of equations it is probably more practical to avoid the calculation of the inverse and solve the equations by a procedure called *factorization*. Various procedures for computing an inverse matrix can be found in texts on numerical analysis. The inverse of  $2 \times 2$  or  $3 \times 3$  matrices can easily be written in closed form by using *Cramer’s rule*. For a  $2 \times 2$  matrix, we have the classic formula, which *no* engineering student should forget.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}}{ad - bc}$$

However, finding the inverse of larger arrays using Cramer’s rule is very inefficient computationally. In MATLAB an inverse matrix of  $\mathbf{A}$  is computed as `inv(A)`, but this is only practical for matrices of a small size, say  $< 100$ . F90 does not have an intrinsic matrix inversion function but we provide such a function, named `inv`, in our operator library.

### 8.2.3 Factorizations

We have indicated that we will frequently employ matrices to solve linear equation systems like  $\mathbf{A} * \mathbf{x} = \mathbf{b}$ , where  $\mathbf{A}$  is a known square matrix,  $\mathbf{B}$  is a known vector, and  $\mathbf{X}$  is an unknown vector. While in theory the solution is simply the inverse of  $\mathbf{A}$  times the vector  $\mathbf{B}$ ,  $\mathbf{x} = \mathbf{A}^{(-1)} * \mathbf{b}$ , that is computationally the least efficient way to find the vector  $\mathbf{X}$ . In practice, one usually uses some form of factorization of the matrix  $\mathbf{A}$ . A very common method is to define  $\mathbf{A}$  to be the product of two triangular matrices, defined above, say  $\mathbf{L} * \mathbf{U} = \mathbf{A}$ , where  $\mathbf{L}$  is a square lower triangular matrix and  $\mathbf{U}$  is a square upper triangular matrix. Skipping the details of this “LU-factorization” we could rewrite the original matrix system as  $\mathbf{L} * \mathbf{U} * \mathbf{x} = \mathbf{b}$ , which can be viewed as two matrix identities:

$$\mathbf{L} * \mathbf{h} = \mathbf{b}$$

$$\mathbf{U} * \mathbf{x} = \mathbf{h},$$

where  $\mathbf{h}$  is a new temporary vector, and where both  $\mathbf{L}$  and  $\mathbf{U}$  are much cheaper to compute than the inverse of  $\mathbf{A}$ . We do not need the inverse of  $\mathbf{L}$  or  $\mathbf{U}$  since, as triangular matrices, their first or last row contains only one non-zero term. That allows us to find one term in the unknown vector from one scalar equation. The processes of recovering the vectors from these two identities is called substitution.

We illustrate this process with an example set of four equations with  $\mathbf{A}$  and  $\mathbf{b}$  given as:

$$\mathbf{A} = \begin{bmatrix} 1800 & 600 & -360 & 900 \\ 0 & 4500 & -2700 & 2250 \\ 0 & -2700 & 2700 & -1890 \\ 6300 & 5250 & -1890 & 3795 \end{bmatrix}$$

$$\mathbf{b}^T = [6300 \quad -2250 \quad 1890 \quad 21405].$$

The LU-factorization process mentioned above gives the first of two lower triangular systems;  $L * h = b$ :

$$\begin{bmatrix} 60 & 0 & 0 & 0 \\ 0 & 150 & 0 & 0 \\ 0 & -90 & 36 & 0 \\ 210 & 105 & 42 & -10 \end{bmatrix} \begin{Bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{Bmatrix} = \begin{Bmatrix} 6300 \\ -2250 \\ 1890 \\ 21405 \end{Bmatrix}.$$

Observe that the significant difference from  $A * x = b$  is that the first row of this identity has one equation and one unknown:

$$60 * h_1 = 6300$$

which yields  $h_1 = 105$ . This process continues through all the rows solving for one unknown,  $h_k$  in row  $k$ , because all the above  $h$  values are known. For example, the next row gives  $0 * 105 + 150 * h_2 = -2250$ , which yields  $h_2 = -15$ . This process is known as “forward substitution.” When completed the substitution yields the intermediate answer:

$$h^T = [105 \quad -15 \quad 15 \quad -30].$$

Now that  $h$  is known we can write the upper triangular identity,  $U * x = h$ , as:

$$\begin{bmatrix} 30 & 10 & -6 & 15 \\ 0 & 30 & -18 & 15 \\ 0 & 0 & 30 & -15 \\ 0 & 0 & 0 & 30 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 105 \\ -15 \\ 15 \\ -30 \end{Bmatrix}.$$

This time the bottom row has only one unknown,  $30 * x_4 = -30$ , so the last unknown is  $x_4 = -1$ . Working backward up to the next row again there is only one unknown:

$$30 * x_3 + -15 * (-1) = 15$$

so that  $x_3 = 0$ . Proceeding back up through the remaining rows to get all the unknowns is called “back substitution.” It yields

$$x^T = [4 \quad 0 \quad 0 \quad -1].$$

By inspection you can verify that this satisfies the original system of linear equations,  $A * x = b$ . With a little more work one can employ matrix multiplication to verify that  $L * U = A$ . While we have not given the simple algorithm for computing  $L$  and  $U$  from  $A$ , it is widely known as the “LU Factorization,” and is in many texts on numerical analysis. Other common factorizations are the “QR Factorization,” the “Cholesky Factorization” for a symmetric positive definite  $A$ , and the “SVD Factorization” for the case where  $A$  is rectangular, or ill-conditioned and one is seeking a best approximation to  $X$ .

The factorization process is relatively expensive to compute but is much less expensive than an inversion. The forward and backward substitutions are very fast and cheap. In problems where you have many different  $b$  vectors (and corresponding  $x$  vectors, such as time dependent problems), one carries out the expensive factorization process only once and then executes the cheap forward and back substitution for each  $b$  vector supplied.

## 8.2.4 Determinant of a Matrix

Every square matrix, say  $A$ , has a single scalar quantity associated with it. That scalar is called the determinant,  $|A|$ , of the matrix. The determinant is important in solving equations and inverting matrices. A very important result is that the inverse  $A^{-1}$  exists if and only if  $|A| \neq 0$ . If the determinant is zero, the matrix  $A$  (and the equivalent set of equations) is said to be *singular*. Simple conditions on a matrix’s structure can be used to infer the determinant or its properties.

- If two rows or columns are equal, the determinant is zero.

- Interchanging two rows, or two columns, changes the sign of the determinant.
- The determinant is unchanged if any row, or column, is modified by adding to it a linear combination of any of the other rows, or columns.
- A singular square matrix may have nonsingular square partitions.

The last two items will become significant when we consider how to apply boundary conditions and how to solve a system of equations.

### 8.2.5 Matrix Calculus

At times you might find it necessary to differentiate or integrate matrices. These operations are simply carried out on each and every element of the matrix. Let the elements  $a_{ij}$  of  $\mathbf{A}$  be a function of a parameter  $t$ . Then, the derivative and integral of a matrix simply equals term-by-term differentiation and integration, respectively.

$$\begin{aligned}\mathbf{B} &= \frac{d\mathbf{A}}{dt} \longleftrightarrow b_{ij} = \frac{da_{ij}}{dt}, \quad 1 \leq i \leq m, 1 \leq j \leq n \\ \mathbf{C} &= \int \mathbf{A} dt \longleftrightarrow c_{ij} = \int a_{ij} dt, \quad 1 \leq i \leq m, 1 \leq j \leq n\end{aligned}$$

When dealing with functional relations the concept of rate of change is often very important. If we have a function  $f(\cdot)$  of a single independent variable, say  $x$ , then we call the rate of change the derivative with respect to  $x$ , which is written as  $df/dx$ . Generalizing this notion to functions of more than two variables, say  $z = f(x, y)$ , we may define two distinct rates of change. One is the function's rate of change with respect to one variable with the other held constant. We thus define *partial derivatives*. When  $x$  is allowed to vary, the derivative is called the *partial derivative with respect to  $x$* , and is denoted by  $\partial f/\partial x$ . By analogy with the usual definition of derivative, this partial derivative is mathematically defined as

$$f_x = \frac{\partial f}{\partial x} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x, y) - f(x, y)}{\Delta x}.$$

A similar definition describes the partial derivative with respect to  $y$ , denoted by  $\partial f/\partial y$ . The second notion of rate-of-change is the *total* derivative, which is expressed as  $df$ .

$$df = \frac{\partial f}{\partial x} dx + \frac{\partial f}{\partial y} dy$$

These definitions can be extended to include a function of any number of independent variables.

Often one encounters a scalar  $u$  defined by a symmetric square  $n \times n$  matrix,  $\mathbf{A}$ , a column vector  $\mathbf{B}$ , and a column vector  $\mathbf{X}$  of  $n$  parameters. The combination we have in mind has the form

$$u = \frac{1}{2} \mathbf{X}^T \mathbf{A} \mathbf{X} + \mathbf{X}^T \mathbf{B} + \mathbf{C} \quad (8.2)$$

If we calculate the derivative of the scalar  $u$  with respect to each  $x_i$ , the result is the column vector

$$\frac{\partial u}{\partial \mathbf{X}} = \mathbf{A} \mathbf{X} + \mathbf{B},$$

a result that can be verified by expanding Equation 8.2, differentiating with respect to every  $x_i$  in  $\mathbf{X}$ , and rewriting the result as a matrix product.

### 8.2.6 Computation with Matrices

Clearly, matrices are useful in representing systems of linear equations and expressing the solution. As said earlier, we need to be able to express linear equations in terms of matrix notation so that analytic manipulations become easy. Furthermore, calculations with linear equations become easy if we can *directly* express our matrix formulas in terms of programs. This section describes programming constructs for the simple matrix expressions and manipulations covered in this chapter.

	MATLAB	C++	F90
Pre-allocate linear array	A(100)=0	int A[100]; <sup>a</sup>	integer A(100)
Initialize to a constant value of 12	for j=1:100 % slow A(j)=12 end % better way A=12*ones(1,100)	for (j=0; j<100; j++) A[j]=12;	A=12
Pre-allocate two-dimensional array	A=ones(10,10)	int A[10][10];	integer A(10,10)

<sup>a</sup>C++ has a starting subscript of 0, but the argument in the allocation statement is the array's size.

**Table 8.12:** Array initialization constructs.

Action	MATLAB	C++	F90
Define size	A=zeros(2,3) <sup>a</sup>	int A[2][3];	integer, dimension (2,3)::A
Enter rows	A=[1, 7, -2; 3, 4, 6];	int A[2][3]={ {1, 7, 2}, {3, 4, 6} };	A(1,:)=(/1,7,-2/) A(2,:)=(/3,4,6/)

<sup>a</sup>Optional in MATLAB, but improves efficiency.

**Table 8.13:** Array initialization constructs

In most languages, we must express the fact that a variable is an ordered array of numbers—a matrix—rather than a scalar (or some other kind of variable). Such *declaration* statements usually occur at the beginning of the program or function. Table 8.12 shows the declaration of an integer array for our suite of programming languages. Both Fortran and C++ require you to specify the maximum range of each subscript of an array before the array or its elements are used. Such range specification is not *required* by MATLAB, but pre-allocating the array space can drastically improve the speed of MATLAB, as well as making much more efficient use of the available memory. If you do not pre-allocate MATLAB arrays, the interpreter must check at each step if a position in a row or column is larger than the current maximum. If so, the maximum value is increased and the memory found to store the new element. Thus, failure to pre-allocate MATLAB arrays is permissible but inefficient.

Array initialization is concisely expressed in both Fortran and MATLAB; in C++, you must write a small program to initialize an array to a nonzero value.<sup>†</sup> If an array contains a variety of different numbers, we can concisely express the initialization; again, in C++, we must explicitly write statements for each array element.

#### An Aside: Matrix Storage

Most computer languages do not make evident how matrices are stored. More frequently than you might think, it becomes necessary to know how an array is actually stored in the computer's memory and retrieved. The procedure both Fortran and MATLAB use to store the elements of an array is known as *column major order*: all the elements of the first column are stored sequentially, then all of the second, etc. Another way of saying this is that the first (left most) subscript ranges over all its values before the second is incremented. After the second subscript has been incremented then the first again ranges over all its values. In C++, *row major order* is used: The first row of an array is stored sequentially, then the second, etc. Clearly, translating programs from Fortran to C++ or vice versa must be done with care.

However, the above knowledge can be used to execute some operations more efficiently. For example, the matrix addition procedure could be written as  $c_k = a_k + b_k$ ,  $1 \leq k \leq m \times n$ . One circumstance

<sup>†</sup>Global arrays—those declared outside of any function definition—are initialized to zero in many versions of C++. Array declared within the scope of a function have no predefined values.

	MATLAB	C++	F90
Addition $C = A + B$	$C=A+B$	<pre>for (i=0; i&lt;n; i++){   for (j=0; j&lt;n; j++){     C[i][j]=A[i][j]+B[i][j];   } }</pre>	$C=A+B$
Multiplication $C = AB$	$C=A*B$	<pre>for (i=0; i&lt;n; i++){   for (j=0; j&lt;n; j++){     C[i][j] = 0;     for (k=0; k&lt;n; k++){       C[i][j] += A[i][k]*B[k][j];     }   } }</pre>	$C=matmul(A,B)$
Scalar multiplication $C = aB$	$C=a*B$	<pre>for (i=0; i&lt;n; i++){   for (j=0; j&lt;n; j++){     C[i][j] = a*B[i][j];   } }</pre>	$C=a*B$
Matrix inverse $B = A^{-1}$	$B=inv(A)$	$a$	$B=inv(A)^a$

<sup>a</sup>Neither C++ nor F90 have matrix inverse functions as part of their language definitions nor as part of standard collections of mathematical functions (like those listed in Table 4.7). Instead, a special function, usually drawn from a library of numerical functions, or a user defined operation, must be used.

**Table 8.14:** Elementary matrix computational routines (for  $n \times n$  matrices)

where knowing the storage format becomes crucial is extracting submatrices in partitioned arrays. Such a Fortran subroutine would have to `dimension` the arrays with a single subscript.

Expressing the addition, subtraction, or multiplication of arrays in Fortran or MATLAB is concise and natural. Explicit programs must be written in C++ to accomplish these calculations. Table 8.14 displays what these constructs are for the special case of square matrices with  $n$  rows.

## 8.3 Exercises

1. Often it is necessary to check computer programs that invert matrices. One approach is use test matrices for which the inverse is known analytically. Few such matrices are known, but one is the following  $n \times n$  matrix.

$$\begin{bmatrix} \frac{n+2}{2n+2} & -\frac{1}{2} & 0 & 0 & \cdots & 0 & \frac{1}{2n+2} \\ -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & \cdots & 0 & 0 \\ 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \cdots & -\frac{1}{2} & 1 & -\frac{1}{2} \\ \frac{1}{2n+2} & 0 & \cdots & \cdots & 0 & -\frac{1}{2} & \frac{n+2}{2n+2} \end{bmatrix}^{-1} = \begin{bmatrix} n & n-1 & n-2 & \cdots & 2 & 1 \\ n-1 & n & n-1 & \cdots & 3 & 2 \\ n-2 & n-1 & n & \cdots & 4 & 3 \\ \vdots & & & \ddots & & \vdots \\ 2 & 3 & 4 & \cdots & n & n-1 \\ 1 & 2 & 3 & \cdots & n-1 & n \end{bmatrix}$$

Develop two routines that will create each of these two matrices for a given  $n$  value, and test them with a main program that uses `matmul` to compute their matrix product. The result should be the identity matrix.

2. The numerical accuracy in calculating an inverse is always an issue: To what extent can you believe the accuracy of the numbers that computer programs calculate. Because of the finite precision used to represent floating point numbers, floating point calculations can only rarely yield exact answers. We want to empirically compute the difference between the inverse of the first matrix in the previous exercise by using a library inversion routine and compare its result with the exact answer. Because the error varies throughout the matrix, we need to summarize the error with a single quantity. Two measures are routinely used: the peak absolute error  $\max_{i,j} |a_{ij} - b_{ij}|$  and the root-mean-squared (rms) error  $\sqrt{\frac{1}{n^2} \sum_{i,j} (a_{ij} - b_{ij})^2}$ .<sup>†</sup> The first captures the biggest difference between the elements of two matrices, and the second summarizes the error throughout the entire difference. Clearly, the peak absolute error is always larger than the rms error. Comparing these two error measures provides some insight into the distribution of error: If the two are comparable, the errors have about the same size; if not, the errors deviate greatly throughout the matrix.

3. Combine the intrinsic array features of F90 with the concepts of OO classes to create a Vector Class that is built around a type that has attributes consisting of the integer length of a vector and an array of its real components. Provide members to construct vectors, delete the arrays, real vectors, list vectors, and carry out basic mathematics operations. Overload the operators +, -, \*, =, and ==. Avoid writing any serial loops.

4. Extend the above Vector Class concepts to a Sparse Vector Class where it is assumed that most of the values in the vector are zero and for efficiency only the non-zero entries are to be stored. This clearly exceeds the intrinsic array features of F90 and begins to show the usefulness of OOP. The defined type must be extended to include an integer array that contains the location (row number) of the non-zero values. In addition to changing the input and output routines to utilize the extra integer position list, all the mathematical member functions such as addition will have to be changed so that the resulting vector has non-zero terms in locations that are a union of the two given location sets (unless the operation creates new zero values). Use the concept of logical array masks in computing the dot product. Avoid writing any serial loops.

---

<sup>†</sup>The  $1/n^2$  term occurs in this expression because that equals the number of terms in the sum. The rms error is used frequently in the practice to measure error; you average the squared error across the dataset and evaluate the square-root of the result.





## Chapter 9

# Advanced Topics

### 9.1 Templates

One of our goals has been to develop software that can be reused for other applications. There are some algorithms that are effectively independent of the object type on which they operate. For example, in a sorting algorithm one often needs to interchange, or *swap*, two objects. A short routine for that purpose follows:

```
subroutine swap_integers (x, y)
implicit none
integer, intent(inout) :: x, y
integer                  :: temp
temp = x
x     = y
y     = temp
end subroutine swap_integers
```

Observe that in this form it appears necessary to have one version for integer arguments and another for real arguments. Indeed we might need a different version of the routine for each type of argument that you may need to swap. A slightly different approach would be to write our swap algorithm as:

```
subroutine swap_objects (x, y)
implicit none
type (Object), intent(inout) :: x, y
type (Object)                  :: temp
temp = x
x     = y
y     = temp
end subroutine swap_objects
```

which would be a single routine that would work for any `Object`, but it has the disadvantage that one find a way to redefine the `Object` type for each application of the routine. That would not be an easy task. (While we will continue with this example with the algorithm in the above forms it should be noted that the above approaches would not be efficient if `x` and `y` were very large arrays or derived type objects. In that case we would modify the algorithm slightly to employ pointers to the large data items and simply swap the pointers for a significant increase in efficiency.)

Consider ways that we might be able to generalize the above routines so that they could accept and swap any specific type of arguments. For example, the first two versions could be re-written in a so called template form as:

```
subroutine swap_Template$ (x, y)
implicit none
Template$, intent(inout) :: x, y
Template$                  :: temp
temp = x
x     = y
y     = temp
end subroutine swap_Template$
```

In the above template the dollar sign (\$) was included in the “wild card” because while it is a valid member of the F90 character set it is not a valid character for inclusion in the name of a variable, derived type, function, module, or subroutine. In other words, a template in the illustrated form would not compile, but

such a name could serve as a reminder that its purpose is to produce a code that can be compiled after the “wild card” substitutions have been made.

With this type of template it would be very easy to use a modern text editor to do a global substitution of any one of the intrinsic types `character`, `complex`, `double precision`, `integer`, `logical`, or `real` for the “wild card” keyword `Template$` to produce a source code to swap any or all of the intrinsic data types. There would be no need to keep up with all the different routine names if we placed all of them in a single module and also created a generic interface to them such as:

```
module swap_library
implicit none
interface swap      ! the generic name
  module procedure swap_character, swap_complex
  module procedure swap_double_precision, swap_integer
  module procedure swap_logical, swap_real
end interface
contains
  subroutine swap_characters (x, y)
    . . .
  end subroutine swap_characters
  subroutine swap_ . . .
    . . .
end module swap_library
```

The use of a text editor to make such substitutions is not very elegant and we expect that there may be a better way to pursue the concept of developing a re-useable software template. The concept of a text editor substitution also fails when we go to the next logical step and try to use a derived type argument instead of any of the intrinsic data types. For example, if we were to replace the “wild card” with our previous type (`chemical_element`) that would create:

```
subroutine swap_type (chemical_element) (x,y)
implicit none
type (chemical_element), intent (inout)::x,y
type (chemical_element)                ::temp
    temp = x
    x = y
    y = temp
end subroutine swap_type (chemical_element)
```

This would fail to compile because it violates the syntax for a valid function or subroutine name, as well as the end function or end subroutine syntax. Except for the first and last line syntax errors this would be a valid code. To correct the problem we simply need to add a little logic and omit the characters `type ( )` when we create a function, module, or subroutine name that is based on a derived type data entity. Then we obtain

```
subroutine swap_chemical_element (x,y)
implicit none
type (chemical_element), intent (inout)::x,y
type (chemical_element)                ::temp
    temp = x
    x = y
    y = temp
end subroutine swap_chemical_element
```

which yields a completely valid routine.

Unfortunately, text editors do not offer us such logic capabilities. However, as we have seen, high level programming languages like C++ and F90 do have those abilities. At this point you should be able to envision writing a pre-processor program that would accept a file of template routines, replace the template “wildcard” words with the desired generic forms to produce a module or header file containing the expanded source files that can then be brought into the desired program with an `include` or `use` statement. The C++ language includes a template pre-processor to expand template files as needed. Some programmers criticize F90/95 for not offering this ability as part of the standard. A few C++ programmers criticize templates and advise against their use. Regardless of the merits of including template pre-processors in a language standard it should be clear that it is desirable to plan software for its efficient reuse.

With F90 if one wants to take advantage of the concepts of templates then the only choices are to carry out a little text editing or develop a pre-processor with the outlined capabilities. The former is clearly the simplest and for many projects may take less time than developing such a template pre-processor. However, if one makes the time investment to produce a template pre-processor one would have a tool

that could be applied to basically any coding project. In the following sections we will give one example of an F90 template pre-processor and demonstrate its application. Reviewing this approach you will probably notice alternate ways to solve the same problem.

## 9.2 Subtyping Objects (Dynamic Dispatching)

One polymorphic feature missing from the Fortran 90 standard that is common to most object oriented languages is called run-time polymorphism or *dynamic dispatching*. (This feature is expected in Fortran 200X as an "extensible" function.) In the C++ language this ability is introduced in the so-called "virtual function". To emulate this ability is quite straightforward in F90 but is not elegant since it usually requires a group of if-elseif statements or other selection processes. It is only tedious if the inheritance hierarchy contains many unmodified subroutines and functions. The importance of the lack of a standardized dynamic dispatching depends on the problem domain to which it must be applied. For several applications demonstrated in the literature the alternate use of subtyping has worked quite well and resulted in programs that have been shown to run several times faster than equivalent C++ versions.

We implement dynamic dispatching in F90 by a process often called subtyping. Two features must be constructed to do this. First, a pointer object, which can point to any subtype member in an inheritance hierarchy, must be developed. Remember that F90 uses the operator '`=>`' to assign pointers to objects, and any object to be pointed at must have the TARGET attribute. Second, we must construct a (dynamic) dispatching mechanism to select the single appropriate procedure to execute at any time during the dynamic execution of the program. This step is done by checking which of the pointers actually points to an object and then passing that (unique) pointer to the corresponding appropriate procedure. In F90 the necessary checking can be carried out by using the ASSOCIATED intrinsic. Here, an if-elseif or other selection method is developed to serve as a dispatch mechanism to select the unique appropriate procedure to be executed based on the actual class referenced in the controlling pointer object. This subtyping process is also referred to as implementing a *polymorphic class*. Of course, the details of the actual dispatching process can be hidden from the procedures that utilize the polymorphic class. The polymorphic class knows only about the interfaces and data types defined in the hierarchy and nothing about how those procedures are implemented.

This process will be illustrated by creating a specific polymorphic class, in this case called `Is_A_Member_Class`, which has polymorphic procedures named `new`, `assign`, and `display`. They will construct a new instance of the object, assign it a value, and list its components. The minimum example of such a process requires two members and is easily extended to any number of member classes. We begin by illustrating a short dynamic dispatching program and then defining each of the subtype classes of interest. The validation of this dynamic dispatching through a polymorphic class is shown in Fig. 9.1. There a target is declared for each possible subtype and then each of them is constructed and sent on to the other polymorphic functions. The results clearly show that different display procedures were used depending on the class of object supplied as an argument. It is expected that the new Fortran 200X standard will allow such dynamic dispatching in a much simpler fashion.

The first subtype is a class, `Member_1_Class`, which has two real components and the encapsulated functionality to construct a new instance and another to accept a pointer to such a subtype and display related information. It is shown in Fig. 9.2. The next subtype class, `Member_2_Class`, has three components: two reals and one of type `Member_1`. It has the same sort of functionality, but clearly must act on more components. It has also inherited the functionality from the `Member_1_Class`; as displayed in Fig. 9.3.

The polymorphic class, `Is_A_Member_Class`, is shown in Fig. 9.4. It includes all of the encapsulated data and function's of the above two subtypes by including their `use` statements. The necessary pointer object is defined as an `Is_A_Member` type that has a unique pointer for each subtype member (two in this case). That is, at any given time during execution it will associate only one of the pointers in this list with an actual pointer object, and the other pointers are nullified. That is why this dispatching is referred to as "dynamic". It also defines a polymorphic interface to each of the common procedures to be applied to the various subtype objects. In the polymorphic function `assign` the dispatching is done very simply. First, all pointers to the family of subtypes are nullified, and then the unique pointer component

```

[ 1] program main
[ 2] use Is_A_Member_Class
[ 3] implicit none
[ 4]
[ 5]   type (Is_A_Member)      :: generic_member
[ 6]   type (member_1), target :: pt_to_memb_1
[ 7]   type (member_2), target :: pt_to_memb_2
[ 8]   character(len=1) :: c
[ 9]
[10]   c = 'A'
[11]   call new (pt_to_memb_1, 1.0, 2.0)
[12]   call assign (generic_member, pt_to_memb_1)
[13]   call display_members (generic_member, c)
[14]
[15]   c = 'B'
[16]   call new (pt_to_memb_2, 1.0, 2.0, 3.0, 4.0)
[17]   call assign (generic_member, pt_to_memb_2)
[18]   call display_members (generic_member, c)
[19]
[20] end program main
[21] ! running gives
[22] ! display_memb_1 A
[23] ! display_memb_2 B

```

**Figure 9.1:** Test of Dynamic Dispatching

```

[ 1] Module Member_1_Class
[ 2]   implicit none
[ 3]   type member_1
[ 4]     real :: real_1, real_2
[ 5]   end type member_1
[ 6]
[ 7] contains
[ 8]
[ 9]   subroutine new_member_1 (member, a, b)
[10]     real, intent(in) :: a, b
[11]     type (member_1) :: member
[12]     member%real_1 = a ; member%real_2 = b
[13]   end subroutine new_member_1
[14]
[15]   subroutine display_memb_1 (pt_to_memb_1, c)
[16]     type (member_1), pointer :: pt_to_memb_1
[17]     character(len=1), intent(in) :: c
[18]     print *, 'display_memb_1 ', c
[19]   end subroutine display_memb_1
[20]
[21] End Module Member_1_Class

```

**Figure 9.2:** The First Subtype Class Member

to the subtype of interest is set to point to the desired member. The dispatching process for the display procedure is different. It requires an if-elseif construct that contains calls to all of the possible subtype members (two here) and a failsafe default state to abort the process or undertake the necessary exception handling. Since all but one of the subtype pointer objects have been nullified it employs the ASSOCIATED intrinsic function to select the one, and only, procedure to call and passes the pointer object on to that procedure. In F90 a pointer can be nullified by using the NULLIFY statement, while F95 allows the alternative of pointing at the intrinsic NULL function which returns a disassociated pointer. The NULL function can also be used to define the initial association status of a pointer at the point it is declared. That is a better programming style.

There are other approaches for implementing the dynamic dispatching concepts. Several examples are given in the publications by the group Decyk, Norton, and Szymanski (1995, 1997, 1999) and on Prof. Szymanski's Web site.

### 9.3 Non-standard Features

Elsewhere in this work only features of Fortran included in the 1995 standard have been utilized. It is common for compiler developers to provide additional enhancements, that are hardware or environment specific, and for the most useful of those features to appear in the next standard release. Compiler releases by Cray © Digital © and Silicon Graphics © computers are examples of versions with extensive enhancements. Some compilers, like the Digital © Visual Fortran © are designed to develop applications

```

[ 1] Module Member_2_Class
[ 2]   Use Member_1_Class
[ 3]   implicit none
[ 4]   type member_2
[ 5]     type (member_1)  :: r_1_2
[ 6]     real :: real_3, real_4
[ 7]   end type member_2
[ 8]
[ 9] contains
[10]
[11]   subroutine new_member_2 (member, a, b, c, d)
[12]     real, intent(in) :: a, b, c, d
[13]     type (member_2)  :: member
[14]     call new_member_1 (member%r_1_2, a, b)
[15]     member%real_3 = c ; member%real_4 = d
[16]   end subroutine new_member_2
[17]
[18]   subroutine display_memb_2 (pt_to_memb_2, c)
[19]     type (member_2), pointer :: pt_to_memb_2
[20]     character(len=1), intent(in) :: c
[21]     print *, 'display_memb_2 ', c
[22]   end subroutine display_memb_2
[23]
[24] End Module Member_2_Class

```

**Figure 9.3:** The Second Subtype Class Member

for the Microsoft © Windows © system and contain library modules for "standard" graphical displays via QuickWin © for dialog routines to the Graphical User Interface (GUI), for interfacing with multiple programming languages or the operation system, and for multiple "thread" operations. Threads are not currently in the F90 standard. They allow for response to the user interaction with any of a set of multiple buttons or dials in an active GUI.

Fortran 90 is a subset of the High Performance Fortran (HPF) standard that has been developed for use on massively parallel computers. We have not discussed those enhancements.

Even without these special enhancements the OOP abilities of F90 provide an important tool in engineering and scientific programming. In support of that position we close with a quote from computer scientist Professor Boleslaw K. Szymanski's Web page on High Performance Object-Oriented Programming in Fortran 90 where his group concludes: "All of our Fortran 90 programs execute more quickly than the equivalent C++ versions, yet the abstraction modeling capabilities that we needed were comparably powerful."

```

[ 1] Module Is_A_Member_Class
[ 2] Use Member_1_Class ; Use Member_2_Class
[ 3] implicit none
[ 4]
[ 5]   type Is_A_Member
[ 6]     private
[ 7]       type (member_1), pointer :: pt_to_memb_1
[ 8]       type (member_2), pointer :: pt_to_memb_2 ! etc for others
[ 9]   end type Is_A_Member
[10]
[11]   interface new
[12]     module procedure new_member_1
[13]     module procedure new_member_2 ! etc for others
[14]   end interface
[15]
[16]   interface assign
[17]     module procedure assign_memb_1
[18]     module procedure assign_memb_2 ! etc for others
[19]   end interface
[20]
[21]   interface display
[22]     module procedure display_memb_1
[23]     module procedure display_memb_2 ! etc for others
[24]   end interface
[25]
[26] contains
[27]
[28]   subroutine assign_memb_1 (Family, member)
[29]     type (member_1), target, intent(in) :: member
[30]     type (Is_A_Member),          intent(out) :: Family
[31]     call nullify_Is_A_Member (Family) ! nullify all
[32]     Family%pt_to_memb_1 => member
[33]   end subroutine assign_memb_1
[34]
[35]   subroutine assign_memb_2 (Family, member)
[36]     type (member_2), target, intent(in) :: member
[37]     type (Is_A_Member),          intent(out) :: Family
[38]     call nullify_Is_A_Member (Family) ! nullify all
[39]     Family%pt_to_memb_2 => member
[40]   end subroutine assign_memb_2 ! etc for others
[41]
[42]   subroutine nullify_Is_A_Member (Family)
[43]     type (Is_A_Member), intent(inout) :: Family
[44]     nullify (Family%pt_to_memb_1)
[45]     nullify (Family%pt_to_memb_2) ! etc for others
[46]   end subroutine nullify_Is_A_Member
[47]
[48]   subroutine display_members (A_Member, c)
[49]     type (Is_A_Member), intent(in) :: A_Member
[50]     character(len=1),   intent(in) :: c
[51]
[52]     ! select the one proper member
[53]     if ( associated (A_Member%pt_to_memb_1) ) then
[54]       call display (A_Member%pt_to_memb_1, c)
[55]     else if ( associated (A_Member%pt_to_memb_2) ) then
[56]       call display (A_Member%pt_to_memb_2, c) ! etc for others
[57]     else ! default case
[58]       stop 'Error, no member defined in Is_A_Member_Class'
[59]     end if
[60]   end subroutine display_members
[61] End Module Is_A_Member_Class

```

**Figure 9.4:** The Polymorphic Class for Subtypes

## Appendix A

# Bibliography

1. Adams, J.C., Brainerd, W.S., Martin, J.T., Smith, B.T. and Wagener, J.L., *Fortran 90 Handbook: Complete ANSI/ISO Reference*, Intertext Publications, McGraw-Hill Book Company, New York, 1992.
2. Akin, J.E. "Object-oriented Programming via Fortran 90," *Engineering Computations*, 16(1) 26-48, 1999.
3. Angell, I.O. and Griffith, G., *High Resolution Computer Graphics Using Fortran 77*, Macmillan, London, 1987.
4. Bar-David, T., *Object-Oriented Design for C++*, Prentice Hall, 1993.
5. Barton, J.J. and L.R. Nackman, *Scientific and Engineering C++*, Addison Wesley, 1994.
6. Cary, J.R., S.G. Shasharina, J.C. Cummings, J.V.W. Reynders, and P.J. Hinker, "A Comparison of C++ and Fortran 90 for Object-Oriented Scientific Programming", *Computer Phys. Comm.*, 105, 20, 1997.
7. Coad, P. and E. Yourdon, *Object Oriented Design*, Prentice Hall, 1991.
8. Decyk, V.K., Norton, C.D. and B.K. Szymanski, "Expressing Object-Oriented Concepts in Fortran90," *ACM Fortran Forum*, 16,(1), April 1997.
9. Decyk, V.K., Norton, C.D. and B.K. Szymanski, "How to Express C++ Concepts in Fortran90," *Scientific Programming*, 6, 363–390, 1997.
10. Dubois-Pèlerin, Y. and T. Zimmermann, "Object-oriented finite element programming: III. An efficient implementation in C++," *Comp. Meth. Appl. Mech. Engr.*, 108, 165–183, 1993.
11. Dubois-Pèlerin, Y. and P. Pegon, "Improving Modularity in Object-Oriented Finite Element Programming," *Communications in Numerical Methods in Engineering*, 13, 193–198, 1997.
12. Filho, J.S.R.A. and P.R.B. Devloo, "Object Oriented Programming in Scientific Computations," *Engineering Computations*, 8(1), 81–87, 1991.
13. Gray, M.G., and R.M. Roberts, "Object-Based Programming in Fortran 90", *Computers in Physics*, 11, 355, 1997.
14. Gehrke, W., *Fortran 90 Language Guide*, Springer, London, 1995.
15. George, A. and J. Liu "An Object-Oriented Approach to the Design of a User Interface for a Sparse Matrix Package", *SIAM J. Matrix Anal. Appl.*, 20(4), 953–969, 1999.
16. Graham, I., *Object Oriented Methods*, Addison-Wesley, 1991.



17. Hahn, B.D., *Fortran 90 for Scientists and Engineers*, Edward Arnold, London, 1994.
  18. Hanly, J.R., *Essential C++ for Engineers and Scientists*, Addison-Wesley, 1997.
  19. Hanselman, D. and Littlefield, B., *Mastering Matlab 5*, Prentice Hall, 1998.
  20. Hubbard, J.R., *Programming with C++*, McGraw Hill, 1994.
  21. Kerrigan, J., *Migrating to Fortran 90*, O'Reilly & Associates, Sebastopol, CA, 1993.
  22. Koelbel, C.H., Loveman, D.B., Schreiber, R.S., Steele, G.L., Jr. and Zosel, M.E., *The High Performance Fortran Handbook*, MIT Press, Cambridge, MA, 1994.
  23. Machiels, L. and M.O. Deville, "Fortran 90: On Entry to Object Oriented Programming for the Solution of Partial Differential Equations," *ACM Trans. Math. Software*, 23(1), 32–49, Mar. 1997.
  24. Mossberg, E. K. Otto, and M. Thune, "Object-Oriented Software Tools for the Construction of Preconditioners" *Scientific Programming*, 6, 285–295, 1997.
  25. Nielsen, K., *Object-Oriented Development with C++*, International Thomson Computer Press, 1997.
  26. Norton, C.D., B.K. Szymanski, and V.K. Decyk, "Object Oriented Parallel Computation for Plasma Simulation", *em Comm. ACM*, 38(10), 88, 1995.
  27. Norton, C.D., V.K. Decyk, and B.K. Szymanski, "High Performance Object-Oriented Scientific Programming in Fortran 90", *em Proc. Eighth SIAM Conf. on Parallel Processing for Scientific Programming*, (Ed. Heath et. al.), March 1997.
  28. Pratap, R., *Getting Started with Matlab*, Saunders College Publishing, Ft. Worth, TX, 1996.
  29. Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P., *Numerical Recipes in Fortran 77*, Cambridge University Press, 1989.
  30. Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P., *Numerical Recipes in Fortran 90*, 2nd ed., Cambridge University Press, 1996.
  31. Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object Oriented Modeling and Design*, Prentice Hall, 1991.
  32. Thomas, P. and Weedon, R., *Object-Oriented Programming in Eiffel*, Addison-Wesley, 1995.
- Links to World Wide Web sites (as of 2001, subject to change):
33. <http://blas.mcmaster.ca/fred/oo.html>
  34. <http://citeseer.nj.nec.com/242268.html>
  35. <http://csep1.phy.ornl.gov/CSEP/PL/PL.html>
  36. <http://epubs.siam.org/sam-bin/dbq/article/31773>
  37. <http://kanaima.ciens.ucv.ve/hpf/HTMLNotesnode29.html>
  38. <http://mis.ucd.ie/courses/mms402/prentice.htm>
  39. <http://webserv.gsfc.nasa.gov/ESS/annual.reports/ess98/cdn.html>
  40. [http://www.amath.washington.edu/lf/software/CompCPP\\_F90SciOOP.html](http://www.amath.washington.edu/lf/software/CompCPP_F90SciOOP.html)
  41. <http://www.arithmetica.ch/Oberon/CFORTRANOberon.nhtml>
  42. <http://www.cs.rpi.edu/szymansk/OOF90>

43. <http://www.nasatech.com/Briefs/Mar98/NPO20180.html>
44. [http://www.owl.net.rice.edu/mech517/F90\\_docs/EC\\_oop\\_f90.pdf](http://www.owl.net.rice.edu/mech517/F90_docs/EC_oop_f90.pdf)
45. <http://www.ssec.wisc.edu/robert/Software/F90-ObjOrientProg.html>
46. <http://www.tdb.uu.se/ngssc/OOP00/module2/>
47. <http://www.ticra.dk/ooa.htm>



## Appendix B

# Fortran 90 Overview

This overview of Fortran 90 (F90) features is presented as a series of tables that illustrate the syntax and abilities of F90. Frequently comparisons are made to similar features in the C++ and F77 languages and to the Matlab environment.

These tables show that F90 has significant improvements over F77 and matches or exceeds newer software capabilities found in C++ and Matlab for dynamic memory management, user defined data structures, matrix operations, operator definition and overloading, intrinsics for vector and parallel processors and the basic requirements for object-oriented programming.

They are intended to serve as a condensed quick reference guide for programming in F90 and for understanding programs developed by others.

## B.1 List of Language Tables

1.1	.....	9
1.2	.....	13
4.1	.....	52
4.2	.....	53
4.3	.....	53
4.4	.....	54
4.5	.....	54
4.6	.....	56
4.7	.....	56
4.8	.....	57
4.9	.....	59
4.10	.....	62
4.11	.....	62
4.12	.....	62
4.13	.....	63
4.14	.....	64
4.15	.....	65
4.16	.....	65
4.17	.....	65
4.18	.....	66
4.19	.....	66
4.20	.....	68
4.21	.....	68
4.22	.....	69
4.23	.....	72
4.24	.....	74

4.25	.....	78
4.26	.....	78
4.27	.....	79
4.28	.....	82
4.29	.....	82
4.30	.....	82
4.31	.....	83
4.32	.....	83
4.33	.....	87
4.34	.....	88
5.1	.....	104
5.2	.....	106
5.3	.....	108
5.4	.....	108
5.5	.....	109
5.6	.....	109
5.7	.....	110
5.8	.....	110
5.9	.....	111
5.10	.....	111
8.1	.....	156
8.2	.....	156
8.3	.....	158
8.4	.....	158
8.5	.....	160
8.7	.....	162
8.8	.....	163
8.9	.....	164
8.10	Intrinsic Functions Allowing Logical Mask Control . . . . .	165
8.11	.....	166
8.12	.....	177
8.13	.....	177
8.14	.....	178
B.1	Comment syntax . . . . .	3
B.2	Intrinsic data types of variables . . . . .	3
B.3	Arithmetic operators . . . . .	3
B.4	Relational operators (arithmetic and logical) . . . . .	4
B.5	Precedence pecking order . . . . .	4
B.6	Colon Operator Syntax and its Applications . . . . .	4
B.7	Mathematical functions . . . . .	5
B.8	Flow Control Statements . . . . .	6
B.9	Basic loop constructs . . . . .	6
B.10	IF Constructs . . . . .	7
B.11	Nested IF Constructs . . . . .	7
B.12	Logical IF-ELSE Constructs . . . . .	7
B.13	Logical IF-ELSE-IF Constructs . . . . .	7
B.14	Case Selection Constructs . . . . .	8
B.15	F90 Optional Logic Block Names . . . . .	8
B.16	GO TO Break-out of Nested Loops . . . . .	8
B.17	Skip a Single Loop Cycle . . . . .	8
B.18	Abort a Single Loop . . . . .	9

B.19 F90 DOs Named for Control . . . . .	9
B.20 Looping While a Condition is True . . . . .	9
B.21 Function definitions . . . . .	10
B.22 Arguments and return values of subprograms . . . . .	10
B.23 Defining and referring to global variables . . . . .	11
B.24 Bit Function Intrinsics . . . . .	11
B.25 The ASCII Character Set . . . . .	12
B.26 F90 Character Functions . . . . .	12
B.27 How to type non-printing characters . . . . .	12
B.28 Referencing Structure Components . . . . .	13
B.29 Defining New Types of Data Structure . . . . .	13
B.30 Nested Data Structure Definitions . . . . .	13
B.31 Declaring, initializing, and assigning components of user-defined datatypes . . . . .	13
B.32 F90 Derived Type Component Interpretation . . . . .	14
B.33 Definition of pointers and accessing their targets . . . . .	14
B.34 Nullifying a Pointer to Break Association with Target . . . . .	14
B.35 Special Array Characters . . . . .	14
B.36 Array Operations in Programming Constructs . . . . .	15
B.37 Equivalent Fortran 90 and MATLAB Intrinsic Functions . . . . .	16
B.38 Truncating Numbers . . . . .	17
B.39 F90 WHERE Constructs . . . . .	17
B.40 F90 Array Operators with Logic Mask Control . . . . .	18
B.41 Array initialization constructs . . . . .	30
B.42 Array initialization constructs . . . . .	30
B.43 Elementary matrix computational routines . . . . .	34
B.44 Dynamic allocation of arrays and pointers . . . . .	34
B.45 Automatic memory management of local scope arrays . . . . .	35
B.46 F90 Single Inheritance Form . . . . .	35
B.47 F90 Selective Single Inheritance Form . . . . .	35
B.48 F90 Single Inheritance Form, with Local Renaming . . . . .	35
B.49 F90 Multiple Selective Inheritance with Renaming . . . . .	36

Language	Syntax	Location
MATLAB	% comment (to end of line)	anywhere
C	/*comment*/	anywhere
F90	! comment (to end of line)	anywhere
F77	* comment (to end of line)	column 1

**Table B.1:** Comment syntax.

Storage	MATLAB <sup>a</sup>	C++	F90	F77
byte		char	character::	character
integer		int	integer::	integer
single precision		float	real::	real
double precision		double	real*8::	double precision
complex		<sup>b</sup>	complex::	complex
Boolean		bool	logical::	logical
argument			parameter::	parameter
pointer		*	pointer::	
structure		struct	type::	

<sup>a</sup>MATLAB4 requires no variable type declaration; the only two distinct types in MATLAB are strings and reals (which include complex). Booleans are just 0s and 1s treated as reals. MATLAB5 allows the user to select more types.

<sup>b</sup>There is no specific data type for a complex variable in C++; they must be created by the programmer.

**Table B.2:** Intrinsic data types of variables.

Description	MATLAB <sup>a</sup>	C++	Fortran <sup>b</sup>
addition	+	+	+
subtraction <sup>c</sup>	-	-	-
multiplication	* and .*	*	*
division	/ and ./	/	/
exponentiation	^ and .^	pow <sup>d</sup>	**
remainder		%	
increment		++	
decrement		--	
parentheses (expression grouping)	( )	( )	( )

<sup>a</sup>When doing arithmetic operations on matrices in MATLAB, a period ('.') must be put before the operator if scalar arithmetic is desired. Otherwise, MATLAB assumes matrix operations; figure out the difference between '\*\*' and '.\*'. Note that since matrix and scalar addition coincide, no '.+' operator exists (same holds for subtraction).

<sup>b</sup>Fortran 90 allows the user to change operators and to define new operator symbols.

<sup>c</sup>In all languages the minus sign is used for negation (i.e., changing sign).

<sup>d</sup>In C++ the exponentiation  $x^y$  is calculated by function *pow* ( $x, y$ ).

**Table B.3:** Arithmetic operators.

Description	MATLAB	C++	F90	F77
Equal to	==	==	==	.EQ.
Not equal to	~=	!=	/=	.NE.
Less than	<	<	<	.LT.
Less or equal	<=	<=	<=	.LE.
Greater than	>	>	>	.GT.
Greater or equal	>=	>=	>=	.GE.
Logical NOT	~	!	.NOT.	.NOT.
Logical AND	&	&&	.AND.	.AND.
Logical inclusive OR	!		.OR.	.OR.
Logical exclusive OR	xor		.XOR.	.XOR.
Logical equivalent	==	==	.EQV.	.EQV.
Logical not equivalent	~=	!=	.NEQV.	.NEQV.

**Table B.4:** Relational operators (arithmetic and logical).

MATLAB Operators	C++ Operators	F90 Operators <sup>a</sup>	F77 Operators
()	() [] -> .	()	()
+ -	! ++ -- + - * & (type) sizeof	**	**
* /	* / %	* /	* /
+ - <sup>b</sup>	+ - <sup>b</sup>	+ - <sup>b</sup>	+ - <sup>b</sup>
< <= > >=	<< >>	//	//
== ~=	< <= > >=	== /= < <= > >=	.EQ. .NE. .LT. .LE. .GT. .GE.
~	== !=	.NOT.	.NOT.
&	&&	.AND.	.AND.
		.OR.	.OR.
=		.EQV. .NEQV.	.EQV. .NEQV.
	?:		
	= += -= *= /= %= &= ^=  = <<= >>=		
	,		

<sup>a</sup>User-defined unary (binary) operators have the highest (lowest) precedence in F90.

<sup>b</sup>These are binary operators representing addition and subtraction. Unary operators + and - have higher precedence.

**Table B.5:** Precedence pecking order.

B = Beginning, E = Ending, I = Increment

Syntax	F90	MATLAB	Use	F90	MATLAB
Default	B:E:I	B:I:E	Array subscript ranges	yes	yes
≥ B	B:	B:	Character positions in a string	yes	yes
≤ E	:E	:E	Loop control	no	yes
Full range	:	:	Array element generation	no	yes

**Table B.6:** Colon Operator Syntax and its Applications.



Description	MATLAB	C++	F90	F77
exponential	<code>exp(x)</code>	<code>exp(x)</code>	<code>exp(x)</code>	<code>exp(x)</code>
natural log	<code>log(x)</code>	<code>log(x)</code>	<code>log(x)</code>	<code>log(x)</code>
base 10 log	<code>log10(x)</code>	<code>log10(x)</code>	<code>log10(x)</code>	<code>log10(x)</code>
square root	<code>sqrt(x)</code>	<code>sqrt(x)</code>	<code>sqrt(x)</code>	<code>sqrt(x)</code>
raise to power ( $x^r$ )	<code>x.^r</code>	<code>pow(x,r)</code>	<code>x**r</code>	<code>x**r</code>
absolute value	<code>abs(x)</code>	<code>fabs(x)</code>	<code>abs(x)</code>	<code>abs(x)</code>
smallest integer $\geq x$	<code>ceil(x)</code>	<code>ceil(x)</code>	<code>ceiling(x)</code>	
largest integer $\leq x$	<code>floor(x)</code>	<code>floor(x)</code>	<code>floor(x)</code>	
division remainder	<code>rem(x,y)</code>	<code>fmod(x,y)</code>	<code>mod(x,y)<sup>a</sup></code>	<code>mod(x,y)</code>
modulo			<code>modulo(x,y)<sup>a</sup></code>	
complex conjugate	<code>conj(z)</code>		<code>conjg(z)</code>	<code>conjg(z)</code>
imaginary part	<code>imag(z)</code>		<code>imag(z)</code>	<code>aimag(z)</code>
drop fraction	<code>fix(x)</code>		<code>aint(x)</code>	<code>aint(x)</code>
round number	<code>round(x)</code>		<code>nint(x)</code>	<code>nint(x)</code>
cosine	<code>cos(x)</code>	<code>cos(x)</code>	<code>cos(x)</code>	<code>cos(x)</code>
sine	<code>sin(x)</code>	<code>sin(x)</code>	<code>sin(x)</code>	<code>sin(x)</code>
tangent	<code>tan(x)</code>	<code>tan(x)</code>	<code>tan(x)</code>	<code>tan(x)</code>
arc cosine	<code>acos(x)</code>	<code>acos(x)</code>	<code>acos(x)</code>	<code>acos(x)</code>
arc sine	<code>asin(x)</code>	<code>asin(x)</code>	<code>asin(x)</code>	<code>asin(x)</code>
arc tangent	<code>atan(x)</code>	<code>atan(x)</code>	<code>atan(x)</code>	<code>atan(x)</code>
arc tangent <sup>b</sup>	<code>atan2(x,y)</code>	<code>atan2(x,y)</code>	<code>atan2(x,y)</code>	<code>atan2(x,y)</code>
hyperbolic cosine	<code>cosh(x)</code>	<code>cosh(x)</code>	<code>cosh(x)</code>	<code>cosh(x)</code>
hyperbolic sine	<code>sinh(x)</code>	<code>sinh(x)</code>	<code>sinh(x)</code>	<code>sinh(x)</code>
hyperbolic tangent	<code>tanh(x)</code>	<code>tanh(x)</code>	<code>tanh(x)</code>	<code>tanh(x)</code>
hyperbolic arc cosine	<code>acosh(x)</code>			
hyperbolic arc sine	<code>asinh(x)</code>			
hyperbolic arctan	<code>atanh(x)</code>			

<sup>a</sup>Differ for  $x < 0$ .

<sup>b</sup>`atan2(x,y)` is used to calculate the arc tangent of  $x/y$  in the range  $[-\pi, +\pi]$ . The one-argument function `atan(x)` computes the arc tangent of  $x$  in the range  $[-\pi/2, +\pi/2]$ .

**Table B.7:** Mathematical functions.

<i>Description</i>	<b>C++</b>	<b>F90</b>	<b>F77</b>	<b>MATLAB</b>
Conditionally execute statements	if { }	if end if	if end if	if end
Loop a specific number of times	for k=1:n { }	do k=1,n end do	do # k=1,n # continue	for k=1:n end
Loop an indefinite number of times	while { }	do while end do	— —	while end
Terminate and exit loop	break	exit	go to	break
Skip a cycle of loop	continue	cycle	go to	—
Display message and abort	error()	stop	stop	error
Return to invoking function	return	return	return	return
Conditional array action	—	where	—	if
Conditional alternate statements	else else if	else elseif	else elseif	else elseif
Conditional array alternatives	— —	elsewhere —	— —	else elseif
Conditional case selections	switch { }	select case end select	if end if	if end

**Table B.8:** Flow Control Statements.

<b>Loop</b>	<b>MATLAB</b>	<b>C++</b>	<b>Fortran</b>
Indexed loop	for index=matrix statements end	for (init;test;inc) { statements }	do index=b,e,i statements end do
Pre-test loop	while test statements end	while (test) { statements }	do while (test) statements end do
Post-test loop		do { statements } while (test)	do statements if (test) exit end do

**Table B.9:** Basic loop constructs.

MATLAB	Fortran	C++
if l_expression true group end	IF (l_expression) THEN true group END IF	if (l_expression) { true group; }
	IF (l_expression) true statement	if (l_expression) true statement;

**Table B.10:** IF Constructs. The quantity *l\_expression* means a logical expression having a value that is either TRUE or FALSE. The term *true statement* or *true group* means that the statement or group of statements, respectively, are executed if the conditional in the *if* statement evaluates to TRUE.

MATLAB	Fortran	C++
if l_expression1 true group A if l_expression2 true group B end true group C end statement group D	IF (l_expression1) THEN true group A IF (l_expression2) THEN true group B END IF true group C END IF statement group D	if (l_expression1) { true group A if (l_expression2) { true group B } true group C } statement group D

**Table B.11:** Nested IF Constructs.

MATLAB	Fortran	C++
if l_expression true group A else false group B end	IF (l_expression) THEN true group A ELSE false group B END IF	if (l_expression) { true group A } else { false group B }

**Table B.12:** Logical IF-ELSE Constructs.

MATLAB	Fortran	C++
if l_expression1 true group A elseif l_expression2 true group B elseif l_expression3 true group C else default group D end	IF (l_expression1) THEN true group A ELSE IF (l_expression2) THEN true group B ELSE IF (l_expression3) THEN true group C ELSE default group D END IF	if (l_expression1) { true group A } else if (l_expression2) { true group B } else if (l_expression3) { true group C } else { default group D }

**Table B.13:** Logical IF-ELSE-IF Constructs.

F90	C++
<pre> SELECT CASE (expression) CASE (value 1)   group 1 CASE (value 2)   group 2   : CASE (value n)   group n CASE DEFAULT   default group END SELECT </pre>	<pre> switch (expression) {   case value 1 :     group 1     break;   case value 2 :     group 2     break;   :   case value n :     group n     break;   default:     default group     break; } </pre>

**Table B.14:** Case Selection Constructs.

F90 Named IF	F90Named SELECT
<pre> name: IF (logical_1) THEN   true group A ELSE IF (logical_2) THEN   true group B ELSE   default group C ENDIF name </pre>	<pre> name: SELECT CASE (expression) CASE (value 1)   group 1 CASE (value 2)   group 2 CASE DEFAULT   default group END SELECT name </pre>

**Table B.15:** F90 Optional Logic Block Names.

Fortran	C++
<pre> DO 1 ... DO 2 ... ... IF (disaster) THEN   GO TO 3 END IF ... 2 END DO 1 END DO 3 next statement </pre>	<pre> for (...) {   for (...) {     ...     if (disaster)       go to error     ...   } } error: </pre>

**Table B.16:** GO TO Break-out of Nested Loops. This situation can be an exception to the general recommendation to avoid GO TO statements.

F77	F90	C++
<pre> DO 1 I = 1,N ... IF (skip condition) THEN   GO TO 1 ELSE   false group END IF 1 continue </pre>	<pre> DO I = 1,N ... IF (skip condition) THEN   CYCLE ! to next I ELSE   false group END IF END DO </pre>	<pre> for (i=1; i&lt;n; i++) {   if (skip condition)     continue; // to next   else if     false group   end } </pre>

**Table B.17:** Skip a Single Loop Cycle.

F77	F90	C++
<pre> DO 1 I = 1,N   IF (exit condition) THEN     GO TO 2   ELSE     false group   END IF 1 CONTINUE 2 next statement </pre>	<pre> DO I = 1,N   IF (exit condition) THEN     EXIT ! this do   ELSE     false group   END IF END DO next statement </pre>	<pre> for (i=1; i&lt;n; i++) {   if (exit condition)     break;// out of loop   else if     false group   end } next statement </pre>

**Table B.18:** Abort a Single Loop.

```

main: DO ! forever
  test: DO k=1,k_max
    third: DO m=m_max,m_min,-1
      IF (test condition) THEN
        CYCLE test ! loop on k
      END IF
    END DO third ! loop on m
    fourth: DO n=n_min,n_max,2
      IF (main condition) THEN
        EXIT main ! forever loop
      END DO fourth ! on n
    END DO test ! over k
  END DO main
next statement

```

**Table B.19:** F90 DOs Named for Control.

MATLAB	C++
<pre> initialize test while l_expression   true group   change test end </pre>	<pre> initialize test while (l_expression) {   true group   change test } </pre>
F77	F90
<pre> initialize test # continue IF (l_expression) THEN   true group   change test go to # END IF </pre>	<pre> initialize test do while (l_expression)   true group   change test end do </pre>

**Table B.20:** Looping While a Condition is True.

Function Type	MATLAB <sup>a</sup>	C++	Fortran
program	<i>statements</i> [y1...yn]=f(a1,...,am) [end of file]	main(argc, char **argv) { <i>statements</i> y = f(a1,I,am); }	program main type y type a1,...,type am <i>statements</i> y = f(a1,...,am) call s(a1,...,am) end program
subroutine		void f (type a1,...,type am) { <i>statements</i> }	subroutine s(a1,...,am) type a1,...,type am <i>statements</i> end
function	function [r1...rn] =f(a1,...,am) <i>statements</i>	type f (type a1,...,type am) { <i>statements</i> }	function f(a1,...,am) type f type a1,...,type am <i>statements</i> end

<sup>a</sup>Every function or program in MATLAB must be in separate files.

**Table B.21:** Function definitions. In each case, the function being defined is named *f* and is called with *m* arguments *a1*, ..., *am*.

One-Input, One-Result Procedures	
MATLAB	function out = name (in)
F90	function name (in) ! name = out
	function name (in) result (out)
C++	name (in, out)
Multiple-Input, Multiple-Result Procedures	
MATLAB	function [inout, out2] = name (in1, in2, inout)
F90	subroutine name (in1, in2, inout, out2)
C++	name(in1, in2, inout, out2)

**Table B.22:** Arguments and return values of subprograms.

Global Variable Declaration	
MATLAB	global list of variables
F77	common /set_name/ list of variables
F90	module set_name save type (type_tag) :: list of variables end module set_name
C++	extern list of variables

Access to Global Variables	
MATLAB	global list of variables
F77	common /set_name/ list of variables
F90	use set_name, only subset of variables use set_name2 list of variables
C++	extern list of variables

**Table B.23:** Defining and referring to global variables.

Action	C++	F90
Bitwise AND	&	iand
Bitwise exclusive OR	^	ieor
Bitwise exclusive OR		ior
Circular bit shift		ishftc
Clear bit		ibclr
Combination of bits		mvbits
Extract bit		ibits
Logical complement	~	not
Number of bits in integer	sizeof	bit_size
Set bit		ibset
Shift bit left	<<	ishft
Shift bit right	>>	ishft
Test on or off		btest
Transfer bits to integer		transfer

**Table B.24:** Bit Function Intrinsics.

0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	NL	11	VT	12	NP	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	"	35	#	36	\$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124		125	}	126	~	127	DEL

**Table B.25:** The ASCII Character Set.

ACHAR (I)	Character number I in ASCII collating set
ADJUSTL (STRING)	Adjust left
ADJUSTR (STRING)	Adjust right
CHAR (I) *	Character I in processor collating set
IACHAR (C)	Position of C in ASCII collating set
ICHAR (C)	Position of C in processor collating set
INDEX (STRING, SUBSTRING) <sup>a</sup>	Starting position of a substring
LEN (STRING)	Length of a character entity
LEN_TRIM (STRING)	Length without trailing blanks
LGE (STRING_A, STRING_B)	Lexically greater than or equal
LGT (STRING_A, STRING_B)	Lexically greater than
LLE (STRING_A, STRING_B)	Lexically less than or equal
LLT (STRING_A, STRING_B)	Lexically less than
REPEAT (STRING, NCOPIES)	Repeated concatenation
SCAN (STRING, SET) <sup>a</sup>	Scan a string for a character in a set
TRIM (STRING)	Remove trailing blank characters
VERIFY (STRING, SET) <sup>a</sup>	Verify the set of characters in a string
STRING_A//STRING_B	Concatenate two strings

<sup>a</sup>Optional arguments not shown.

**Table B.26:** F90 Character Functions.

Action	ASCII Character	F90 Input <sup>a</sup>	C++ Input
Alert (Bell)	7	Ctrl-G	\a
Backspace	8	Ctrl-H	\b
Carriage Return	13	Ctrl-M	\r
End of Transmission	4	Ctrl-D	Ctrl-D
Form Feed	12	Ctrl-L	\f
Horizontal Tab	9	Ctrl-I	\t
New Line	10	Ctrl-J	\n
Vertical Tab	11	Ctrl-K	\v

<sup>a</sup>“Ctrl-” denotes control action. That is, simultaneous pressing of the CONTROL key *and* the letter following.

**Table B.27:** How to type non-printing characters.



C, C++	Variable.component.sub_component
F90	Variable%component%sub_component

**Table B.28:** Referencing Structure Components.

C, C++	<pre>struct data_tag {     intrinsic_type_1 component_names;     intrinsic_type_2 component_names; } ;</pre>
F90	<pre>type data_tag     intrinsic_type_1 :: component_names;     intrinsic_type_2 :: component_names; end type data_tag</pre>

**Table B.29:** Defining New Types of Data Structure.

C, C++	<pre>struct data_tag {     intrinsic_type_1 component_names;     struct tag_2 component_names; } ;</pre>
F90	<pre>type data_tag     intrinsic_type :: component_names;     type (tag_2) :: component_names; end type data_tag</pre>

**Table B.30:** Nested Data Structure Definitions.

C, C++	<pre>struct data_tag variable_list; /* Definition */ struct data_tag variable = {component_values}; /* Initialization */ variable.component.sub_component = value; /* Assignment */</pre>
F90	<pre>type (data_tag) :: variable_list ! Definition variable = data_tag (component_values) ! Initialization variable%component%sub_component = value ! Assignment</pre>

**Table B.31:** Declaring, initializing, and assigning components of user-defined datatypes.

<pre> INTEGER, PARAMETER :: j_max = 6 TYPE meaning_demo   INTEGER, PARAMETER :: k_max = 9, word = 15   CHARACTER (LEN = word) :: name(k_max) END TYPE meaning_demo TYPE (meaning_demo) derived(j_max) </pre>	
Construct	Interpretation
derived	All components of all derived's elements
derived(j)	All components of j <sup>th</sup> element of derived
derived(j)%name	All k_max components of name within j <sup>th</sup> element of derived
derived%name(k)	Component k of the name array for all elements of derived
derived(j)%name(k)	Component k of the name array of j <sup>th</sup> element of derived

**Table B.32:** F90 Derived Type Component Interpretation.

	C++	F90
Declaration	<code>type_tag *pointer_name;</code>	<code>type (type_tag), pointer :: pointer_name</code>
Target	<code>&amp;target_name</code>	<code>type (type_tag), target :: target_name</code>
Examples	<pre> char *cp, c; int *ip, i; float *fp, f; cp = &amp; c; ip = &amp; i; fp = &amp; f; </pre>	<pre> character, pointer :: cp integer, pointer :: ip real, pointer :: fp cp =&gt; c ip =&gt; i fp =&gt; f </pre>

**Table B.33:** Definition of pointers and accessing their targets.

C, C++	<code>pointer_name = NULL</code>
F90	<code>nullify (list_of_pointer_names)</code>
F95	<code>pointer_name = NULL()</code>

**Table B.34:** Nullifying a Pointer to Break Association with Target.

Purpose	F90	MATLAB
Form subscripts	<code>()</code>	<code>()</code>
Separates subscripts & elements	<code>,</code>	<code>,</code>
Generates elements & subscripts	<code>:</code>	<code>:</code>
Separate commands	<code>;</code>	<code>;</code>
Forms arrays	<code>(/ /)</code>	<code>[ ]</code>
Continue to new line	<code>&amp;</code>	<code>...</code>
Indicate comment	<code>!</code>	<code>%</code>
Suppress printing	default	<code>;</code>

**Table B.35:** Special Array Characters.

<i>Description</i>	<i>Equation</i>	<i>Fortran90 Operator</i>	<i>Matlab Operator</i>	<i>Original Sizes</i>	<i>Result Size</i>
Scalar plus scalar	$c = a \pm b$	$c = a \pm b$	$c = a \pm b;$	1, 1	1, 1
Element plus scalar	$c_{jk} = a_{jk} \pm b$	$c = a \pm b$	$c = a \pm b;$	$m, n$ and 1, 1	$m, n$
Element plus element	$c_{jk} = a_{jk} \pm b_{jk}$	$c = a \pm b$	$c = a \pm b;$	$m, n$ and $m, n$	$m, n$
Scalar times scalar	$c = a \times b$	$c = a * b$	$c = a * b;$	1, 1	1, 1
Element times scalar	$c_{jk} = a_{jk} \times b$	$c = a * b$	$c = a * b;$	$m, n$ and 1, 1	$m, n$
Element times element	$c_{jk} = a_{jk} \times b_{jk}$	$c = a * b$	$c = a.*b;$	$m, n$ and $m, n$	$m, n$
Scalar divide scalar	$c = a/b$	$c = a/b$	$c = a/b;$	1, 1	1, 1
Scalar divide element	$c_{jk} = a_{jk}/b$	$c = a/b$	$c = a/b;$	$m, n$ and 1, 1	$m, n$
Element divide element	$c_{jk} = a_{jk}/b_{jk}$	$c = a/b$	$c = a./b;$	$m, n$ and $m, n$	$m, n$
Scalar power scalar	$c = a^b$	$c = a**b$	$c = a \wedge b;$	1, 1	1, 1
Element power scalar	$c_{jk} = a_{jk}^b$	$c = a**b$	$c = a \wedge b;$	$m, n$ and 1, 1	$m, n$
Element power element	$c_{jk} = a_{jk}^{b_{jk}}$	$c = a**b$	$c = a.^{\wedge} b;$	$m, n$ and $m, n$	$m, n$
Matrix transpose	$C_{kj} = A_{jk}$	$C = \text{transpose}(A)$	$C = A';$	$m, n$	$n, m$
Matrix times matrix	$C_{ij} = \sum_k A_{ik} B_{kj}$	$C = \text{matmul}(A, B)$	$C = A * B;$	$m, r$ and $r, n$	$m, n$
Vector dot vector	$c = \sum_k A_k B_k$	$c = \text{sum}(A * B)$ $c = \text{dot\_product}(A, B)$	$c = \text{sum}(A.*B);$ $c = A * B';$	$m, 1$ and $m, 1$ $m, 1$ and $m, 1$	1, 1 1, 1

**Table B.36:** Array Operations in Programming Constructs. Lower case letters denote scalars or scalar elements of arrays. Matlab arrays are allowed a maximum of two subscripts while Fortran allows seven. Upper case letters denote matrices or scalar elements of matrices.

**Table B.37:** Equivalent Fortran90 and MATLAB Intrinsic Functions.

The following KEY symbols are utilized to denote the TYPE of the intrinsic function, or subroutine, and its arguments: A-complex, integer, or real; I-integer; L-logical; M-mask (logical); R-real; X-real; Y-real; V-vector (rank 1 array); and Z-complex. Optional arguments are not shown. Fortran 90 and MATLAB also have very similar array operations and colon operators.

Type	Fortran90	MATLAB	Brief Description
A	ABS(A)	abs(a)	Absolute value of A.
R	ACOS(X)	acos(x)	Arc cosine function of real X.
R	AIMAG(Z)	imag(z)	Imaginary part of complex number.
R	AINIT(X)	real(fix(x))	Truncate X to a real whole number.
L	ALL(M)	all(m)	True if all mask elements, M, are true.
R	ANINT(X)	real(round(x))	Real whole number nearest to X.
L	ANY(M)	any(m)	True if any mask element, M, is true.
R	ASIN(X)	asin(x)	Arcsine function of real X.
R	ATAN(X)	atan(x)	Arctangent function of real X.
R	ATAN2(Y,X)	atan2(y,x)	Arctangent for complex number(X, Y).
I	CEILING(X)	ceil(x)	Least integer $\geq$ real X.
Z	CMPLX(X,Y)	(x+yi)	Convert real(s) to complex type.
Z	CONJG(Z)	conj(z)	Conjugate of complex number Z.
R	COS(R_Z)	cos(r_z)	Cosine of real or complex argument.
R	COSH(X)	cosh(x)	Hyperbolic cosine function of real X.
I	COUNT(M)	sum(m==1)	Number of true mask, M, elements.
R,L	DOT_PRODUCT(X,Y)	x'*y	Dot product of vectors X and Y.
R	EPSILON(X)	eps	Number, like X, $\ll 1$ .
R,Z	EXP(R_Z)	exp(r_z)	Exponential of real or complex number.
I	FLOOR(X)	floor	Greatest integer $\leq$ X.
R	HUGE(X)	realmax	Largest number like X.
I	INT(A)	fix(a)	Convert A to integer type.
R	LOG(R_Z)	log(r_z)	Logarithm of real or complex number.
R	LOG10(X)	log10(x)	Base 10 logarithm function of real X.
R	MATMUL(X,Y)	x*y	Conformable matrix multiplication, X*Y.
I,V	I=MAXLOC(X)	[y,i]=max(x)	Location(s) of maximum array element.
R	Y=MAXVAL(X)	y=max(x)	Value of maximum array element.
I,V	I=MINLOC(X)	[y,i]=min(x)	Location(s) of minimum array element.
R	Y=MINVAL(X)	y=min(x)	Value of minimum array element.
I	NINT(X)	round(x)	Integer nearest to real X.
A	PRODUCT(A)	prod(a)	Product of array elements.
call	RANDOM_NUMBER(X)	x=rand	Pseudo-random numbers in (0, 1).
call	RANDOM_SEED	rand('seed')	Initialize random number generator.
R	REAL (A)	real(a)	Convert A to real type.
R	RESHAPE(X, (/ I, I2 /))	reshape(x,i,i2)	Reshape array X into I $\times$ I2 array.
I,V	SHAPE(X)	size(x)	Array (or scalar) shape vector.
R	SIGN(X,Y)		Absolute value of X times sign of Y.
R	SIGN(0.5,X)-SIGN(0.5,-X)	sign(x)	Signum, normalized sign, -1, 0, or 1.
R,Z	SIN(R_Z)	sin(r_z)	Sine of real or complex number.
R	SINH(X)	sinh(x)	Hyperbolic sine function of real X.
I	SIZE(X)	length(x)	Total number of elements in array X.
R,Z	SQRT(R_Z)	sqrt(r_z)	Square root, of real or complex number.
R	SUM(X)	sum(x)	Sum of array elements.

(continued)

Type	Fortran90	MATLAB	Brief Description
R	TAN(X)	tan(x)	Tangent function of real X.
R	TANH(X)	tanh(x)	Hyperbolic tangent function of real X.
R	TINY(X)	realmin	Smallest positive number like X.
R	TRANPOSE(X)	x'	Matrix transpose of any type matrix.
R	X=1	x=ones(length(x))	Set all elements to 1.
R	X=0	x=zero(length(x))	Set all elements to 0.

For more detailed descriptions and example uses of these intrinsic functions see Adams, J.C., *et al.*, *Fortran 90 Handbook*, McGraw-Hill, New York, 1992, ISBN 0-07-000406-4.

## B.2 Alphabetical Table of Fortran 90 Intrinsic Routines

The following KEY symbols are utilized to denote the TYPE of the intrinsic function, or subroutine, and its arguments: A-complex, integer, or real; B-integer bit; C-character; D-dimension; I-integer; K-kind; L-logical; M-mask (logical); N-integer, or real; P-pointer; R-real; S-string; T-target; V-vector (rank one array); X-real; Y-real; Z-complex; and \*-any type. For more detailed descriptions and example uses of these intrinsic functions see Adams, J.C., *et al.*, *Fortran 90 Handbook*, McGraw-Hill, New York, 1992, ISBN 0-07-000406-4.

C++	–	int	–	–	floor	ceil
F90	aint	int	anint	nint	floor	ceiling
MATLAB	real (fix)	fix	real (round)	round	floor	ceil
Argument	Value of Result					
–2.000	–2.0	–2	–2.0	–2	–2	–2
–1.999	–1.0	–1	–2.0	–2	–2	–1
–1.500	–1.0	–1	–2.0	–2	–2	–1
–1.499	–1.0	–1	–1.0	–1	–2	–1
–1.000	–1.0	–1	–1.0	–1	–1	–1
–0.999	0.0	0	–1.0	–1	–1	0
–0.500	0.0	0	–1.0	–1	–1	0
–0.499	0.0	0	0.0	0	–1	0
0.000	0.0	0	0.0	0	0	0
0.499	0.0	0	0.0	0	0	1
0.500	0.0	0	1.0	1	0	1
0.999	0.0	0	1.0	1	0	1
1.000	1.0	1	1.0	1	1	1
1.499	1.0	1	1.0	1	1	2
1.500	1.0	1	2.0	2	1	2
1.999	1.0	1	2.0	2	1	2
2.000	2.0	2	2.0	2	2	2

**Table B.38:** Truncating Numbers.

WHERE (logical_array_expression) true_array_assignments ELSEWHERE false_array_assignments END WHERE
WHERE (logical_array_expression) true_array_assignment

**Table B.39:** F90 WHERE Constructs.

<i>Function</i>	<i>Description</i>	<i>Opt</i>	<i>Example</i>
all	Find if all values are true, for a fixed dimension.	d	all(B = A, DIM = 1) (true, false, false)
any	Find if any value is true, for a fixed dimension.	d	any (B > 2, DIM = 1) (false, true, true)
count	Count number of true elements for a fixed dimension.	d	count(A = B, DIM = 2) (1, 2)
maxloc	Locate first element with maximum value given by mask.	m	maxloc(A, A < 9) (2, 3)
maxval	Max element, for fixed dimension, given by mask.	b	maxval (B, DIM=1, B > 0) (2, 4, 6)
merge	Pick true array, A, or false array, B, according to mask, L.	–	merge(A, B, L) $\begin{bmatrix} 0 & 3 & 5 \\ 2 & 4 & 8 \end{bmatrix}$
minloc	Locate first element with minimum value given by mask.	m	minloc(A, A > 3) (2, 2)
minval	Min element, for fixed dimension, given by mask.	b	minval(B, DIM = 2) (1, 2)
pack	Pack array, A, into a vector under control of mask.	v	pack(A, B < 4) (0, 7, 3)
product	Product of all elements, for fixed dimension, controlled by mask.	b	product(B) ; (720) product(B, DIM = 1, T) (2, 12, 30)
sum	Sum all elements, for fixed dimension, controlled by mask.	b	sum(B) ; (21) sum(B, DIM = 2, T) (9, 12)
unpack	Replace the true locations in array B controlled by mask L with elements from the vector U.	–	unpack(U, L, B) $\begin{bmatrix} 7 & 3 & 8 \\ 2 & 4 & 9 \end{bmatrix}$

$$A = \begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}, \quad L = \begin{bmatrix} T & F & T \\ F & F & T \end{bmatrix}, \quad U = (7, 8, 9)$$

**Table B.40:** F90 Array Operators with Logic Mask Control. *T* and *F* denote true and false, respectively. Optional arguments: b -- DIM & MASK, d -- DIM, m -- MASK, v -- VECTOR and DIM = 1 implies for any rows, DIM = 2 for any columns, and DIM = 3 for any plane.

<b>Type</b>	<b>Intrinsic</b>	<b>Description</b>
A	ABS (A)	Absolute value of A.
C	ACHAR (I)	Character in position I of ASCII collating sequence.
R	ACOS (X)	Arc cosine (inverse cosine) function of real X.
C	ADJUSTL (S)	Adjust S left, move leading blanks to trailing blanks.
C	ADJUSTR (S)	Adjust S right, move trailing blanks to leading blanks.
R	AIMAG (Z)	Imaginary part of complex number, Z.
R	AINIT (X [,K])	Truncate X to a real whole number, of the given kind.
L	ALL (M [,D])	True if all mask, M, elements are true, in dimension D.
L	ALLOCATED (*_ARRAY_P)	True if the array or pointer is allocated.

(continued)

Alphabetic Table of Fortran90 Intrinsic Functions (continued)		
Type	Intrinsic	Description
R	ANINT (X [,K])	Real whole number nearest to X, of the given kind.
L	ANY (M [,D])	True if any mask, M, element is true, in dimension D.
R	ASIN (X)	Arcsine (inverse sine) function of real X.
L	ASSOCIATED (P [,T])	True if pointer, P, is associated with any target, or T.
R	ATAN (X)	Arctangent (inverse tangent) function of real X.
R	ATAN2 (Y,X)	Arctangent for argument of complex number (X, Y).
I	BIT_SIZE (I)	Maximum number of bits integer I can hold, e.g. 32.
L	BTEST (I,I_POS)	True if bit location I_POS of integer I has value 1.
I	CEILING (X)	Least integer $\geq$ real X, of the given kind.
C	CHAR (I [,K])	Character in position I of processor collating sequence.
Z	CMPLX (X [,Y][,K])	Convert real(s) to complex type, of given kind.
Z	CONJG (Z)	Conjugate of complex number Z.
R	COS (R_Z)	Cosine function of real or complex argument.
R	COSH (X)	Hyperbolic cosine function of real X.
I	COUNT (M [,D])	Number of true mask, M, elements, in dimension D.
*	CSHIFT (*_ARRAY,I_SHIF [,D])	Circular shift out and in for I_SHIF elements.
call	DATE_AND_TIME ([S_DATE] [,S_TIME] [,S_ZONE] [,I_V_VALUES])	Real-time clock date, time, zone, and vector with year, month, day, UTC, hour, minutes, seconds, and milliseconds.
R	DBLE (A)	Convert A to double precision real.
N	DIGITS (N)	Number of significant digits for N, e.g. 31.
R	DIM (X,Y)	The difference, MAX (X - Y, 0.0).
N,L	DOT_PRODUCT (V,V_2)	Dot product of vectors V and V_2.
R	DPROD (X,Y)	Double precision real product of two real scalars.
*	EOSHIFT (*_ARRAY, I_SHIFT [,*_FILL][,D])	Perform vector end-off shift by $\pm$ I_shift terms, and fill, in dimension D.
R	EPSILON (X)	Number $\ll$ 1, for numbers like X, e.g. 2** $-23$ .
R,Z	EXP (R_Z)	Exponential function of real or complex argument.
I	EXPONENT (X)	Exponent part of the model for real X.
I	FLOOR (X)	Greatest integer less than or equal to X.
R	FRACTION (X)	Fractional part of the model for real X.
N	HUGE (N)	Largest number for numbers like N, e.g. 2**128.
I	IACHAR (C)	Position of character C in ASCII collation.
B	IAND (I,I_2)	Logical AND on the bits of I and I_2.
B	IBCLR (I,I_POS)	Clear bit I_POS to zero in integer I.
B	IBITS (I,I_POS,I_LEN)	Extract an I_LEN sequence of bits at I_POS in I.
B	IBSET (I,I_POS)	Set bit I_POS to one in integer I.
I	ICHAR (C)	Position of character C in processor collation.
B	IEOR (I,I_2)	Exclusive OR on the bits of I and I_2.
I	INDEX (S,S_SUB [,L_BACK])	Left starting position of S_SUB within S (right).
I	INT (A [,K])	Convert A to integer type, of given kind.
B	IOR (I,I_2)	Inclusive OR on the bits of I and I_2.
B	ISHFT (I,I_SHIFT)	Logical shift of bits of I by I_SHIFT, pad with 0.
B	ISHFTC (I,I_SHIFT [,I_SIZE])	Logical circular shift of I_SIZE rightmost bits of I.
I	KIND (ANY)	Kind type integer parameter value for any argument.
I,V	LBOUND (*_ARRAY [,D])	ARRAY lower bound(s) vector, along dimension D.
I	LEN (S)	Total character string length.
I	LEN_TRIM (S)	Length of S without trailing blanks.
L	LGE (S,S_2)	True if S $>$ or equal to S_2 in ASCII sequence.
L	LGT (S,S_2)	True if S follows S_2 in ASCII collating sequence.

(continued)

Alphabetic Table of Fortran90 Intrinsic Functions (continued)		
Type	Intrinsic	Description
L	LLE (S,S_2)	True if S < or equal to S_2 in ASCII sequence.
L	LLT (S,S_2)	True if S precedes S_2 in ASCII collating sequence.
R	LOG (R_Z)	Natural (base e) logarithm of real or complex number.
L	LOGICAL (L [,K])	Convert L to logical of kind K.
R	LOG10 (X)	Common (base 10) logarithm function of real X.
N,L	MATMUL (MATRIX,MATRIX_2)	Conformable matrix multiplication.
N	MAX (N,N_2 [,N_3,...])	Maximum value of two or more numbers same type.
I	MAXEXPONENT (X)	Maximum exponent for real numbers like X, e.g. 128.
I,V	MAXLOC (N_ARRAY [,M])	Location(s) of maximum ARRAY element, passing M.
N	MAXVAL (N_ARRAY [,D] [,M])	Maximum ARRAY term, in dimension D, passing M.
*	MERGE (*_TRUE, *_FALSE,M)	Use *_TRUE when M is true; *_FALSE otherwise.
N	MIN (N,N_2 [,N_3,...])	Minimum value of two or more same type numbers.
I	MINEXPONENT (X)	Minimum exponent for real numbers like X, e.g. -125.
I,V	MINLOC (N_ARRAY [,M])	Location(s) of minimum ARRAY term, passing M.
N	MINVAL (N_ARRAY [,D] [,M])	Minimum ARRAY term, in dimension D, passing M.
N	MOD (N,N_2)	Remainder for N_2. That is, N-INT(N/N_2)*N_2.
N	MODULO (N,N_2)	Modulo, that is, N-FLOOR(N/N_2)*N_2.
call	MVBITS (I_FROM,I_LOC, I_LEN,I_TO,I_POS)	Copy I_LEN bits at I_LOC in I_FROM to I_TO at I_POS.
R	NEAREST (X,Y)	Nearest number at X in the direction of sign Y.
I	NINT (X [,K])	Integer nearest to real X, of the stated kind.
I	NOT (I)	Logical complement of the bits of integer I.
*,V	PACK (*_ARRAY,M [,V_PAD])	Pack ARRAY at true M into vector, using V_PAD.
I	PRECISION (R_Z)	Decimal precision for a real or complex R_Z, e.g. 6.
L	PRESENT (OPTIONAL)	True if optional argument is present in call.
A	PRODUCT (A_ARRAY [,D] [,M])	Product of ARRAY elements, along D, for mask M.
I	RADIX (N)	Base of the model for numbers like N, e.g. 2.
call	RANDOM_NUMBER (X)	Pseudo-random numbers in range $0 < X < 1$ .
call	RANDOM_SEED ([I_SIZE] [,I_V_PUT],[I_V_GET])	Initialize random number generator, defaults to processor initialization.
I	RANGE (A)	Decimal exponent range in the model for A, e.g. 37.
R	REAL (A [,K])	Convert A to real type, of type K.
S	REPEAT (S,I_COPIES)	Concatenates I_COPIES of string S.
*	RESHAPE (*_ARRAY,I_V_SHAP [,*_PAD] [,V_ORDER])	Reshape ARRAY, using vector SHAP, pad from an array, and re-order.
R	RRSPACING (X)	Relative spacing reciprocal of numbers near X.
R	SCALE (X,I)	Return X times b**I, for base of b = RADIX (X).
I	SCAN (S,S_SET [,L_BACK])	Leftmost character index in S found in S_SET; (rightmost).
I	SELECTED_INT_KIND (I_r)	Integer kind with range, $-(10**I_r)$ to $(10**I_r)$ .
I	SELECTED_REAL_KIND ([I] [,I_r])	Kind for real of decimal precision, I, and exponent range, I_r.
R	SET_EXPONENT (X,I)	Number with mantissa of X and exponent of I.
I,V	SHAPE (*_ARRAY)	ARRAY (or scalar) shape vector.
N	SIGN (N,N_2)	Absolute value of N times sign of same type N_2.
R,Z	SIN (R_Z)	Sine function of real or complex number.
R	SINH (X)	Hyperbolic sine function of real X.
I	SIZE (*_ARRAY [,D])	ARRAY size, along dimension D.
R	SPACING (X)	Absolute spacing of numbers near real X, e.g. $2**{-17}$ .
*	SPREAD (*_ARRAY,D,I_COPIES)	I_COPIES along dimension D of ARAY into an array of rank 1 greater.

(continued)



Alphabetic Table of Fortran90 Intrinsic Functions (continued)		
Type	Intrinsic	Description
R,Z	SQRT (R_Z)	Square root function, of real or complex number.
A	SUM (A_ARRAY [,D] [,M])	Sum of ARRAY elements, along D, passing mask M.
call	SYSTEM_CLOCK ([I_NOW] [,I_RATE] [,I_MAX])	Integer data from real-time clock. CPU time is (finish_now - start_now) / rate.
R	TAN (X)	Tangent function of real X.
R	TANH (X)	Hyperbolic tangent function of real X.
R	TINY (N)	Smallest positive number, like N, e.g. 2* $\epsilon$ -126.
*	TRANSFER (*_ARRAY, V_MOLD [,I_SIZE])	Same representation as ARRAY, but type of MOLD, in vector of length SIZE.
*	TRANSPOSE (MATRIX)	Matrix transpose of any type matrix.
S	TRIM (S)	Remove trailing blanks from a single string.
I,V	UBOUND (*_ARRAY [,D])	ARRAY upper bound(s) vector, along dimension D.
*	UNPACK (V,M,*_USE)	Unpack vector V at true elements of M, into USE.
I	VERIFY (S,S_SET [,L_BACK])	First position in S not found in S_SET (or last).

## Subject Table of Fortran 90 Intrinsic Routines

The following KEY symbols are utilized to denote the TYPE of the intrinsic function, or subroutine, and its arguments: A-complex, integer, or real; B-integer bit; C-character; D-dimension; I-integer; K-kind; L-logical; M-mask (logical); N-integer, or real; P-pointer; R-real; S-string; T-target; V-vector (rank one array); X-real; Y-real; Z-complex; and \*-any type. For more detailed descriptions and example uses of these intrinsic functions see Adams, J.C., et al., *Fortran 90 Handbook*, McGraw-Hill, New York, 1992, ISBN 0-07-000406-4.

Type	Intrinsic	Description
	ALLOCATION	
L	ALLOCATED (*_ARRAY)	True if the array is allocated.
	ARGUMENT	
L	PRESENT (OPTIONAL)	True if optional argument is present in the call.
	ARRAY: CONSTRUCTION	
*	MERGE (*_TRUE,*_FALSE,M)	Use *_TRUE if M is true; *_FALSE otherwise.
*,V	PACK (*_ARRAY,M [,V_PAD])	Pack ARRAY for true M into vector, pad from V_PAD.
*	RESHAPE (*_ARRAY,I_V_SHAPE [,*_PAD] [,V_ORDER])	Reshape ARRAY using vector SHAPE, pad from an array, and re-order.
*	SPREAD (*_ARRAY,D,I_COPIES)	I_COPIES along D of ARRAY to rank 1 greater array.
*	UNPACK (V,M,*_USE)	Unpack V at true elements of M, into USE.
	ARRAY: DIMENSIONS	
I,V	LBOUND (*_ARRAY [,D])	ARRAY lower bound(s) vector, along dimension D.
I,V	SHAPE (*_ARRAY)	ARRAY (or scalar) shape vector.
I	SIZE (*_ARRAY [,D])	ARRAY size, along dimension D.
I,V	UBOUND (*_ARRAY [,D])	ARRAY upper bound(s) vector, along dimension D.
	ARRAY: INQUIRY	
L	ALL (M [,D])	True if all mask, M, elements are true, along D.
L	ALLOCATED (*_ARRAY)	True if the array is allocated.
L	ANY (M [,D])	True if any mask, M, element is true, along D.
I,V	LBOUND (*_ARRAY [,D])	ARRAY lower bound(s) vector, along dimension D.
I,V	SHAPE (*_ARRAY)	ARRAY (or scalar) shape vector.

(continued)

<i>Subject Table of Fortran 90 Intrinsic Functions (continued)</i>		
Type	Intrinsic	Description
I,V	UBOUND (*_ARRAY [,D])	ARRAY upper bound(s) vector, along dimension D.
	ARRAY: LOCATION	
I,V	MAXLOC (N_ARRAY [,M])	Location(s) of maximum ARRAY term, passing M.
I,V	MINLOC (N_ARRAY [,M])	Location(s) of minimum ARRAY term, passing M.
	ARRAY: MANIPULATION	
*	CSHIFT (*_ARRAY,I_SHIFT [,D])	Circular shift out and in for I_SHIFT elements.
*	EOSHIFT (*_ARRAY,I_SHIFT [,*_FIL],[,D])	End-off shift ARRAY, and fill, in dimension D.
*	TRANPOSE (MATRIX)	Matrix transpose of any type matrix.
	ARRAY: MATHEMATICS	
N,L	DOT_PRODUCT (V,V_2)	Dot product of vectors V and V_2.
N,L	MATMUL (MATRIX,MATRIX_2)	Conformable matrix multiplication.
N	MAXVAL (N_ARRAY [,D] [,M])	Value of max ARRAY term, along D, passing M.
N	MINVAL (N_ARRAY [,D] [,M])	Value of min ARRAY term, along D, passing M.
A	PRODUCT (A_ARRAY [,D] [,M])	Product of ARRAY terms, along D, for mask M.
A	SUM (A_ARRAY [,D] [,M])	Sum of ARRAY terms, along D, passing mask M.
	ARRAY: PACKING	
*,V	PACK (*_ARRAY,M [,V_PAD])	Pack ARRAY for true M into vector, pad from V_PAD.
*	UNPACK (V,M,*_USE)	Unpack V at true elements of M, into USE.
	ARRAY: REDUCTION	
L	ALL (M [,D])	True if all mask, M, terms are true, along D.
L	ANY (M [,D])	True if any mask, M, term is true, along D.
I	COUNT (M [,D])	Number of true mask, M, terms, along dimension D.
N	MAXVAL (N_ARRAY [,D] [,M])	Value of max ARRAY term, along D, passing M.
N	MINVAL (N_ARRAY [,D] [,M])	Value of min ARRAY term, along D, passing M.
A	PRODUCT (A_ARRAY [,D] [,M])	Product of ARRAY terms, along D, for mask M.
A	SUM (A_ARRAY [,D] [,M])	Sum of ARRAY terms, along D, passing mask M.
	BACK SCAN	
I	INDEX (S,S_SUB [,L_BACK])	Left starting position of S_SUB within S (or right).
I	SCAN (S,S_SET [,L_BACK])	Left character index in S also in S_SET (or right).
I	VERIFY (S,S_SET [,L_BACK])	First position in S not belonging to S_SET (or last).
	BIT: INQUIRY	
I	BIT_SIZE (I)	Max number of bits possible in integer I, e.g. 32.
	BIT: MANIPULATION	
L	BTEST (I,I_POS)	True if bit location I_POS of integer I has value one.
B	IAND (I,I_2)	Logical AND on the bits of I and I_2.
B	IBCLR (I,I_POS)	Clear bit I_POS to zero in integer I.
B	IBITS (I,I_POS,I_LEN)	Extract I_LEN bits at I_POS in integer I.
B	IBSET (I,I_POS)	Set bit I_POS to one in integer I.
B	IEOR (I,I_2)	Exclusive OR on the bits of I and I_2.
B	IOR (I,I_2)	Inclusive OR on the bits of I and I_2.
B	ISHFT (I,I_SHIFT)	Logical shift of bits of I by I_SHIFT, pad with 0.
B	ISHFTC (I,I_SHIFT [,I_SIZE])	Logical circular shift of I_SIZE rightmost bits of I.
call	MVBITS (I_GET, I_LOC, I, I_TO, I_POS)	Copy I bits at I_LOC in I_GET to I_TO at I_POS.
I	NOT (I)	Logical complement of the bits of integer I.
*	TRANSFER (*_ARRAY,	

(continued)

Subject Table of Fortran 90 Intrinsic Functions (continued)		
Type	Intrinsic	Description
	V _MOLD [,I _SIZE])	Same representation as ARRAY, but type of MOLD.
<b>BOUNDS</b>		
I	CEILING (X)	Least integer greater than or equal to real X.
I	FLOOR (X)	Greatest integer less than or equal to X.
I,V	LBOUND (* _ARRAY [,D])	ARRAY lower bound(s) vector, along dimension D.
N	MAX (N,N _2 [,N _3,...])	Maximum value of two or more numbers same type.
N	MAXVAL (N _ARRAY [,D] [,M])	Value of max ARRAY term, along D, passing M.
N	MINVAL (N _ARRAY [,D] [,M])	Value of min ARRAY term, along D, passing M.
I,V	UBOUND (* _ARRAY [,D])	ARRAY upper bound(s) vector, along dimension D.
<b>CALLS</b>		
call	MVBITS (I _GET,I _LOC,I, I _TO,I _POS)	Copy I bits at I _LOC in I _GET to I _TO at I _POS.
call	DATE _AND _TIME ([S _DATE] [,S _TIME] [,S _ZONE] [,I _V _VALUES])	Real-time clock data.
call	RANDOM _NUMBER (X)	Pseudo-random numbers in range $0 < X < 1$ .
call	RANDOM _SEED ([I _SIZE] [,I _V _P] [,I _V _G])	Initialize random number generator.
call	SYSTEM _CLOCK ([I _NOW] [,I _RAT] [,I _MX])	Integer data from real-time clock.
<b>CHARACTERS</b>		
C	ACHAR (I)	Character in position I of ASCII collating sequence.
C	CHAR (I [,K])	Character in position I of processor collation.
I	IACHAR (C)	Position of character C in ASCII collating sequence.
I	ICHAR (C)	Position of character C in processor collation.
<b>CLOCK</b>		
call	SYSTEM _CLOCK ([I _NOW] [,I _RAT] [,I _MX])	Integer data from real-time clock.
<b>COMBINING</b>		
*	MERGE (* _TRUE,* _FALSE,M)	Use * _TRUE term if M is true or * _FALSE otherwise.
<b>COMPLEX</b>		
R	AIMAG (Z)	Imaginary part of complex number.
Z	CMPLX (X [,Y][,K])	Convert real(s) to complex type, of given kind.
Z	CONJG (Z)	Conjugate of complex number Z.
R	COS (R _Z)	Cosine function of real or complex argument.
R,Z	EXP (R _Z)	Exponential function of real or complex argument.
R	LOG (R _Z)	Natural (base e) logarithm of real or complex number.
I	PRECISION (R _Z)	Decimal precision of real or complex value, e.g. 6.
R,Z	SIN (R _Z)	Sine function of real or complex number.
R,Z	SQRT (R _Z)	Square root function, of real or complex number.
<b>CONVERSIONS</b>		
R	AIMAG (Z)	Imaginary part of complex number.
R	AINT (X [,K])	Truncate X to a real whole number.
Z	CMPLX (X [,Y][,K])	Convert real (s) to complex type, of given kind.
R	DBLE (A)	Convert A to double precision real.
R	DPROD (X,Y)	Double precision product of two default real scalars.

(continued)

<i>Subject Table of Fortran 90 Intrinsic Functions (continued)</i>		
Type	Intrinsic	Description
I	INT (A [,K])	Convert A to integer type, of given kind.
L	LOGICAL (L [,K])	Convert L to logical of kind K.
I	NINT (X [,K])	Integer nearest to real X, of the stated kind.
R	REAL (A [,K])	Convert A to real type, of type K.
N	SIGN (N,N_2)	Absolute value of N times sign of same type N_2.
*	TRANSFER (*_ARRAY, V_MOLD [,I_SIZ])	Same representation as ARRAY, but type of MOLD.
COPIES		
*	MERGE (*_TRUE,*_FALSE,M)	Use *_TRUE if M is true or *_FALSE otherwise.
call	MVBITS (I_FROM,I_LOC, I, I_TO,I_POS)	Copy I bits at I_LOC in I_FROM to I_TO at I_POS.
S	REPEAT (S,I_COPIES)	Concatenates I_COPIES of string S.
*	SPREAD (*_ARRAY,D,I_COPIES)	I_COPIES along D of ARRAY to rank 1 greater array.
COUNTING		
I	COUNT (M [,D])	Number of true mask, M, terms, along dimension D.
DATE		
call	DATE_AND_TIME ([S_DATE] [,S_TIME] [,S_ZONE] [,I_V_VALUES])	Real-time clock data.
DIMENSION OPTIONAL ARGUMENT		
L	ALL (M [,D])	True if all mask, M, terms are true, along D.
L	ANY (M [,D])	True if any mask, M, term is true, along D.
I	COUNT (M [,D])	Number of true mask, M, terms, along dimension D.
*	CSHIFT (*_ARRAY,I_SHIFT [,D])	Perform circular shift out and in for I_SHIFT terms.
*	EOSHIFT (*_ARRAY, I_SHIFT [,*_FIL][,D])	Perform end-off shift, and fill, in dimension D.
I,V	LBOUND (*_ARRAY [,D])	ARRAY lower bound(s) vector, along dimension D.
N	MAXVAL (N_ARRAY [,D] [,M])	Value of max ARRAY term, along D, passing M.
N	MINVAL (N_ARRAY [,D] [,M])	Value of min ARRAY term, along D, passing M.
A	PRODUCT (A_ARRAY [,D] [,M])	Product of ARRAY terms, along D, for mask M.
I	SIZE (*_ARRAY [,D])	ARRAY size, along dimension D.
A	SUM (A_ARRAY [,D] [,M])	Sum of ARRAY terms, along D, passing mask M.
I,V	UBOUND (*_ARRAY [,D])	ARRAY upper bound(s) vector, along dimension D.
DIMENSIONS		
I,V	LBOUND (*_ARRAY [,D])	ARRAY lower bound(s) vector, along dimension D.
I,V	SHAPE (*_ARRAY)	ARRAY (or scalar) shape vector.
I	SIZE (*_ARRAY [,D])	ARRAY size, along dimension D.
I,V	UBOUND (*_ARRAY [,D])	ARRAY upper bound(s) vector, along dimension D.
DOUBLE PRECISION		
R	DBLE (A)	(see SELECTED_REAL_KIND) Convert A to double precision real.
R	DPROD (X,Y)	Double precision product of two default real scalars.
EXISTENCE		
L	ALLOCATED (*_ARRAY)	True if the array is allocated.
L	ASSOCIATED (P [,T])	True if pointer, P, is associated with any target, or T.
L	PRESENT (OPTIONAL)	True if optional argument is present in call.
FILE		

(continued)

Subject Table of Fortran 90 Intrinsic Functions (continued)		
Type	Intrinsic	Description
	FILL IN	
*	EOSHIFT (*_ARRAY,I_SHIFT [,*_FIL][,D])	End-off shift ARRAY, and fill, in dimension D.
	INQUIRY: ARRAY	
L	ALL (M [,D])	True if all mask, M, terms are true, along D.
L	ALLOCATED (*_ARRAY)	True if the array is allocated.
L	ANY (M [,D])	True if any mask, M, term is true, along D.
I,V	LBOUND (*_ARRAY [,D])	ARRAY lower bound(s) vector, along dimension D.
I,V	SHAPE (*_ARRAY)	ARRAY (or scalar) shape vector.
I	SIZE (*_ARRAY [,D])	ARRAY size, along dimension D.
I,V	UBOUND (*_ARRAY [,D])	ARRAY upper bound(s) vector, along dimension D.
	INQUIRY: BIT	
I	BIT_SIZE (I)	Max number of bits possible in integer I, e.g. 32.
	INQUIRY: CHARACTER	
I	LEN (S)	Total character string length.
I	LEN_TRIM (S)	Length of S without trailing blanks.
	INQUIRY: NUMBER MODEL	
N	DIGITS (N)	Number of significant digits in number N, e.g. 31.
R	EPSILON (X)	Number $\ll 1$ , for numbers like X, e.g. $2^{*-23}$ .
N	HUGE (N)	Largest number for numbers like N, e.g. $2^{*128}$ .
I	MAXEXPONENT (X)	Max exponent for real numbers like X, e.g. 128.
I	MINEXPONENT (X)	Min exponent for real numbers like X, e.g. -125.
I	PRECISION (R_Z)	Decimal precision for real or complex value, e.g. 6.
I	RADIX (N)	Base of the model for numbers like N, e.g. 2.
I	RANGE (A)	Decimal exponent range for A, e.g. 37.
I,V	SHAPE (*_ARRAY)	ARRAY (or scalar) shape vector.
I	SIZE (*_ARRAY [,D])	ARRAY size, along dimension D.
R	TINY (N)	Smallest positive number, like N, e.g. $2^{*-126}$ .
	INQUIRY: MISCELLANEOUS	
I	COUNT (M [,D])	Number of true mask, M, elements, along D.
I	INDEX (S,S_SUB [,L_BACK])	Left starting position of S_SUB within S (or right).
I	SCAN (S,S_SET [,L_BACK])	Left character index in S also in S_SET; (or right).
I	VERIFY (S,S_SET [,L_BACK])	First position in S not belonging to S_SET, (or last).
	INTEGERS	
I	CEILING (X)	Least integer greater than or equal to real X.
I	FLOOR (X)	Greatest integer less than or equal to X.
I	MAX1 (X,X2 [,X3])	Maximum integer from list of reals
I	MIN1 (X,X2 [,X3])	Minimum integer from list of reals
N	MODULO (N,N_2)	Modulo, $N - \text{FLOOR}(N/N_2) * N_2$ .
I	SELECTED_INT_KIND (I_r)	Integer with exponent, $-(10^{*I_r})$ to $(10^{*I_r})$ .
	KIND: INQUIRY	
I	KIND (ANY)	Kind type integer parameter value for any argument.
	KIND: DEFINITION	
I	SELECTED_INT_KIND (I_r)	Integer with exponent, $-(10^{*I_r})$ to $(10^{*I_r})$ .
I	SELECTED_REAL_KIND ([I [,I_r])	Real with precision, I, and exponent range, I_r.
	KIND: USE OPTION	

(continued)

<i>Subject Table of Fortran 90 Intrinsic Functions (continued)</i>		
Type	Intrinsic	Description
R	AINT (X [,K])	Truncate X to a real whole number.
R	ANINT (X [,K])	Real whole number nearest to X.
C	CHAR (I [,K])	Character in position I of processor collation.
Z	CMPLX (X [,Y][,K])	Convert real(s) to complex type, of given kind.
I	INT (A [,K])	Convert A to integer type, of given kind.
L	LOGICAL (L [,K])	Convert L to logical of kind K.
I	NINT (X [,K])	Integer nearest to real X, of the stated kind.
R	REAL (A [,K])	Convert A to real type, of type K.
LOCATION		
I	IACHAR (C)	Position of character C in ASCII collating sequence.
I	ICHAR (C)	Position of character C in processor collation.
I	INDEX (S,S_SUB [,L_BACK])	Left starting position of S_SUB within S (or right).
I,V	MAXLOC (N_ARRAY [,M])	Vector location(s) of ARRAY maximum, passing M.
I,V	MINLOC (N_ARRAY [,M])	Vector location(s) of ARRAY minimum, passing M.
I	SCAN (S,S_SET [,L_BACK])	Left character index in S found in S_SET; (or right).
LOGICAL		
L	ALL (M [,D])	True if all mask, M, terms are true, along D.
L	ALLOCATED (*_ARRAY)	True if the array is allocated.
L	ANY (M [,D])	True if any mask, M, term is true, along D.
L	ASSOCIATED (P [,T])	True if pointer, P, is associated with any target, or T.
L	BTEST (I,I_POS)	True if bit location I_POS of integer I has value one.
N,L	DOT_PRODUCT (V,V_2)	Dot product of vectors V and V_2.
B	IAND (I,I_2)	Logical AND on the bits of I and I_2.
B	IEOR (I,I_2)	Exclusive OR on the bits of I and I_2.
B	IOR (I,I_2)	Inclusive OR on the bits of I and I_2.
B	ISHFT (I,I_SHIFT)	Logical shift of bits of I by I_SHIFT, pad with 0.
L	LGE (S,S_2)	True if S is $\geq$ S_2 in ASCII collating sequence.
L	LGT (S,S_2)	True if S follows S_2 in ASCII collating sequence.
L	LLE (S,S_2)	True if S is $\leq$ to S_2 in ASCII collating sequence.
L	LLT (S,S_2)	True if S precedes S_2 in ASCII collating sequence.
N,L	MATMUL (MATRIX,MATRIX_2)	Conformable matrix multiplication.
L	LOGICAL (L [,K])	Convert L to logical of kind K.
I	NOT (I)	Logical complement of the bits of integer I.
L	PRESENT (OPTIONAL)	True if optional argument is present in call.
MASK, or MASK OPTIONAL ARGUMENT		
L	ALL (M [,D])	True if all mask, M, terms are true, along D.
L	ANY (M [,D])	True if any mask, M, term is true, along D.
I	COUNT (M [,D])	Number of true mask, M, terms, along dimension D.
I,V	MAXLOC (N_ARRAY [,M])	Vector of location(s) of ARRAY max's, passing M.
N	MAXVAL (N_ARRAY [,D] [,M])	Value of ARRAY maximum, along D, passing M.
*	MERGE (*_TRUE,*_FALSE,M)	Use *_TRUE if M is true or *_FALSE otherwise.
I,V	MINLOC (N_ARRAY [,M])	Vector location(s) of ARRAY minimum, passing M.
N	MINVAL (N_ARRAY [,D] [,M])	Value of ARRAY minimum, along D, passing M.
*,V	PACK (*_ARRAY,M [,V_PAD])	Pack ARRAY for true M into vector, pad from V_PAD.
A	PRODUCT (A_ARRAY [,D] [,M])	Product of ARRAY terms, along D, for mask M.
A	SUM (A_ARRAY [,D] [,M])	Sum of ARRAY terms, along D, passing mask M.
MATHEMATICAL FUNCTIONS		
R	ACOS (X)	Arc cosine (inverse cosine) function of real X.

(continued)

Subject Table of Fortran 90 Intrinsic Functions (continued)		
Type	Intrinsic	Description
R	ASIN (X)	Arcsine (inverse sine) function of real X.
R	ATAN (X)	Arctangent (inverse tangent) function of real X.
R	ATAN2 (Y,X)	Arctangent for argument of complex number (X, Y).
R	COS (R_Z)	Cosine function of real or complex argument.
R	COSH (X)	Hyperbolic cosine function of real X.
R,Z	EXP (R_Z)	Exponential function of real or complex argument.
R	LOG (R_Z)	Natural logarithm of real or complex number.
R	LOG10 (X)	Common (base 10) logarithm function of real X.
R,Z	SIN (R_Z)	Sine function of real or complex number.
R	SINH (X)	Hyperbolic sine function of real X.
R	TAN (X)	Tangent function of real X.
R	TANH (X)	Hyperbolic tangent function of real X.
MATRICES (See ARRAYS)		
N,I	DOT_PRODUCT (V,V_2)	Dot product of vectors V and V_2.
N,I	MATMUL (MATRIX,MATRIX_2)	Conformable matrix multiplication.
*	TRANPOSE (MATRIX)	Matrix transpose of any type matrix.
NUMBER MODEL		
N	DIGITS (N)	Number of significant digits for N, e.g. 31.
R	EPSILON (X)	Number $\ll 1$ , for numbers like X, e.g. 2** $-23$ .
I	EXPONENT (X)	Exponent part of the model for real X.
R	FRACTION (X)	Fractional part of the model for real X.
N	HUGE (N)	Largest number for numbers like N, e.g. 2**128.
R	NEAREST (X,Y)	Nearest number at X in the direction of sign Y.
I	RADIX (N)	Base of the model for numbers like N, e.g. 2.
I	RANGE (A)	Decimal exponent range for A, e.g. 37.
R	RRSPACING (X)	Reciprocal of relative spacing of numbers near X.
R	SCALE (X,I)	Return X times b**I, where base b = RADIX (X).
R	SET_EXPONENT (X,I)	Real with mantissa part of X and exponent part of I.
R	SPACING (X)	Absolute spacing of numbers near X, e.g. 2** $-17$ .
R	TINY (N)	Smallest positive number, like N, e.g. 2** $-126$ .
NUMERIC FUNCTIONS		
A	ABS (A)	Absolute value of A.
R	AIMAG (Z)	Imaginary part of complex number.
R	ANINT (X [,K])	Real whole number nearest to X.
I	CEILING (X)	Least integer greater than or equal to real X.
Z	CMPLX (X [,Y][,K])	Convert real(s) to complex type, of given kind.
Z	CONJG (Z)	Conjugate of complex number Z.
R	DBLE (A)	Convert A to double precision real.
R	DPROD (X,Y)	Double precision real product of two real scalars.
I	FLOOR (X)	Greatest integer less than or equal to X.
I	INT (A [,K])	Convert A to integer type, of given kind.
N	MAX (N,N_2 [,N_3,...])	Maximum value of two or more numbers same type.
N	MIN (N,N_2 [,N_3,...])	Minimum value of two or more same type numbers.
N	MOD (N,N_2)	Remainder for N_2, i.e., N-INT(N/N_2)*N_2.
N	MODULO (N,N_2)	Modulo, N-FLOOR(N/N_2)*N_2.
R	REAL (A [,K])	Convert A to real type, of type K.
N	SIGN (N,N_2)	Absolute value of N times sign of same type N_2.
PADDING		
B	ISHFT (I,I_SHIFT)	Logical shift of bits of I by I_SHIFT, pad with 0.

(continued)

Subject Table of Fortran 90 Intrinsic Functions (continued)		
Type	Intrinsic	Description
*,V	PACK (*_ARRAY,M [,V _PAD])	Pack ARRAY for true M into vector, pad from V _PAD.
*	RESHAPE (*_ARRAY,I _V _SHAPE [,*_PAD] [,V _ORDER])	Reshape ARRAY to vector SHAPE, pad, re-order.
POINTER		
L	ASSOCIATED (P [,T])	True if pointer, P, is associated with any target, or T.
PRESENCE		
L	PRESENT (OPTIONAL)	True if optional argument is present in call.
RANDOM NUMBER		
call	RANDOM _NUMBER (X)	Pseudo-random numbers in range $0 < X < 1$ .
call	RANDOM _SEED ([I _SIZE] [,I _V _P][,I _V _G])	Initialize random number generator.
REALS		
R	AINT (X [,K])	Truncate X to a real whole number.
R	ANINT (X [,K])	Real whole number nearest to X.
R	AMAX0 (I,I2 [,I3])	Maximum real from list of integers.
R	AMIN0 (I,I2 [,I3])	Minimum real from list of integers.
R	REAL (A [,K])	Convert A to real type, of type K.
I	SELECTED _REAL _KIND ([I [,I _r])	Real with precision, I, and exponent range, I _r.
REDUCTION		
L	ALL (M [,D])	True if all mask, M, terms are true, along D.
L	ANY (M [,D])	True if any mask, M, term is true, along D.
I	COUNT (M [,D])	Number of true mask, M, terms, along dimension D.
N	MAXVAL (N _ARRAY [,D] [,M])	Value of max ARRAY term, along D, passing M.
N	MINVAL (N _ARRAY [,D] [,M])	Value of min ARRAY term, along D, passing M.
A	PRODUCT (A _ARRAY [,D] [,M])	Product of ARRAY terms, along D, for mask M.
A	SUM (A _ARRAY [,D] [,M])	Sum of ARRAY terms, along D, passing mask M.
RESHAPING ARRAYS		
*	CSHIFT (*_ARRAY,I _SHIFT [,D])	Perform circular shift out and in for I _SHIFT terms.
*	EOSHIFT (*_ARRAY,I _SHFT [,*_FIL] [,D])	End-off shift ARRAY, and fill, in dimension D.
*,V	PACK (*_ARRAY,M [,V _PAD])	Pack ARRAY for true M into vector, pad from V _PAD.
*	RESHAPE (*_ARRAY,I _V _SHAPE [,*_PAD] [,V _ORDER])	Reshape ARRAY to vector SHAPE, pad, re-order.
*	UNPACK (V,M,*_USE)	Unpack V for true elements of M, into USE.
REVERSE ORDER		
I	INDEX (S,S _SUB [,L _BACK])	Left starting position of S _SUB within S (right-most).
I	SCAN (S,S _SET [,L _BACK])	Left character index in S found in S _SET; (right-most).
I	VERIFY (S,S _SET [,L _BACK])	First position in S not found in S _SET, (or last).
SHIFTS		
*	CSHIFT (*_ARRAY,I _SHIFT [,D])	Perform circular shift out and in for I _SHIFT terms.
*	EOSHIFT (*_ARRAY,I _SHIFT [,*_FILL][,D])	Perform end-off shift, and fill, in dimension D.
B	ISHFT (I,I _SHIFT)	Logical shift of bits of I by I _SHIFT, pad with 0.
B	ISHFTC (I,I _SHIFT [,I _SIZE])	Logical circular shift of I _SIZE rightmost bits of I.

(continued)



Subject Table of Fortran 90 Intrinsic Functions (continued)		
Type	Intrinsic	Description
STRING		
C	ADJUSTL (S)	Adjust S left, move leading blanks to trailing blanks.
C	ADJUSTR (S)	Adjust S right, move trailing to leading blanks.
I	INDEX (S,S _SUB [,L _BACK])	Left starting position of S _SUB within S (or right).
I	LEN (S)	Total character string length.
I	LEN _TRIM (S)	Length of S without trailing blanks.
L	LGE (S,S _2)	True if S is $\geq$ to S _2 in ASCII collating sequence.
L	LGT (S,S _2)	True if S follows S _2 in ASCII collating sequence.
L	LLE (S,S _2)	True if S is $\leq$ to S _2 in ASCII collating sequence.
L	LLT (S,S _2)	True if S precedes S _2 in ASCII collating sequence.
S	REPEAT (S,I _COPIES)	Concatenates I _COPIES of string S.
I	SCAN (S,S _SET [,L _BACK])	Left character index in S found in S _SET; (or right).
S	TRIM (S)	Remove trailing blanks from a single string.
I	VERIFY (S,S _SET [,L _BACK])	First position in S not found in S _SET, (or last).
TARGET		
L	ASSOCIATED (P [,T])	True if pointer, P, is associated with any target, or T.
TIME		
call	DATE _AND _TIME ([S _DATE] [S _TIME] [S _ZONE] [I _V _VALUES])	Real-time clock data.
call	SYSTEM _CLOCK ([I _NOW] [I _RAT] [I _MX])	Integer data from real-time clock.
VECTOR (See ARRAYS)		
N,L	DOT _PRODUCT (V,V _2)	Dot product of vectors V and V _2.
I,V	LBOUND (* _ARRAY [,D])	ARRAY lower bound(s) vector, along D.
I,V	MAXLOC (N _ARRAY [,M])	Location(s) of maximum ARRAY term, passing M.
I,V	MINLOC (N _ARRAY [,M])	Location(s) of minimum ARRAY term, passing M.
*,V	PACK (* _ARRAY,M [,V _PAD])	Pack ARRAY for true M into vector, pad from V _PAD.
*	RESHAPE (* _ARRAY,I _V _SHAPE [,* _PAD] [,V _ORDER])	Reshape ARRAY to vector SHAPE, pad, re-order.
I,V	SHAPE (* _ARRAY)	ARRAY (or scalar) shape vector.
*	TRANSFER (* _ARRAY, V _MOLD [,I _SIZE])	Same representation as ARRAY, but type of MOLD.
I,V	UBOUND (* _ARRAY [,D])	ARRAY upper bound(s) vector, along dimension D.

## B.3 Syntax of Fortran 90 Statements

The following is a list of the recommended Fortran90 statements. Additional statements are allowed, but have been declared obsolete, and are expected to be deleted in future standards. Thus, they should not be utilized in new programs. They are appended to the end of this list. Below we list the standard syntax for the Fortran90 statements. In some cases the most common simple form of a statement is shown before it's more general options. Such optional features are shown included in brackets, [ ], and a vertical bar | means "or." Note that the new attribute terminator symbol :: is always optional, but its use is recommended.

The following abbreviations are employed: arg=argument, attr=attribute, exp=expression, i\_=integer, r\_=real, s\_=string, spec=specifier, and here [type] means CHARACTER | COMPLEX | INTEGER | LOGICAL | REAL, or a user defined name given in a TYPE statement. Recall that F90 allows variable names to be 31 characters long and they may include an underscore (but F77 allows only 6 characters and no underscore). F90 lines may contain up to 132 characters (but just 72 in F77). All

	MATLAB	C++	F90
Pre-allocate linear array	A(100)=0	int A[100]; <sup>a</sup>	integer A(100)
Initialize to a constant value of 12	for j=1:100 % slow A(j)=12 end % better way A=12*ones(1,100)	for (j=0; j<100; j++) A[j]=12;	A=12
Pre-allocate two-dimensional array	A=ones(10,10)	int A[10][10];	integer A(10,10)

<sup>a</sup>C++ has a starting subscript of 0, but the argument in the allocation statement is the array's size.

**Table B.41:** Array initialization constructs.

Action	MATLAB	C++	F90
Define size	A=zeros(2,3) <sup>a</sup>	int A[2][3];	integer,dimension(2,3)::A
Enter rows	A=[1,7,-2; 3, 4, 6];	int A[2][3]={ {1,7,2}, {3,4,6} };	A(1,:)=(/1,7,-2/ A(2,:)=(/3,4,6/)

<sup>a</sup>Optional in MATLAB, but improves efficiency.

**Table B.42:** Array initialization constructs.

standard F77 statements are a sub-set of F90. Attribute options, and their specifiers, for each statement are given in the companion table "Fortran 90 Attributes and Specifiers". The numerous options for the INQUIRE statement are given in the table entitled "Options for F90 INQUIRE."

In addition to the statements given below F90 offers intrinsic array operations, implied do loops, vector subscripts, and about 160 intrinsic functions. Those functions, with their arguments, are given in tables "Alphabetical Table of Fortran 90 Intrinsic Functions and Subroutines," and "Subject Table of Fortran 90 Intrinsic Functions and Subroutines."

F90 Syntax
<p>! precedes a comment in F90  in column one denotes a comment line in F77  &amp; continues a line in F90 (must be in column 6 for F77)  ; terminates a statement in F90 (allows multiple statements per line)  variable = expression_or_statement ! is an assignment (column 7 in F77)</p> <p>ALLOCATABLE (::) array_name[(extents)] [, array_name[(extents)]]  ALLOCATE (array_name)  ALLOCATE (array_name [, STAT=status] [,array_name [, STAT=status]])  BACKSPACE i_exp ! file unit number  BACKSPACE ([UNIT=i_value [, IOSTAT=i_variable] [, ERR=i_label])  C in column one denotes a comment line in F77  CALL subroutine_name [(args)]  CASE (range_list) [select_name] ! purpose  CASE DEFAULT [select_name] ! purpose  CHARACTER LEN=i_value (::) s_list  CHARACTER [(LEN=i_value   * [, KIND=i_kind)] [, attr_list] ::) s_list  CHARACTER [(i_value   *, [KIND=i_kind)] [, attr_list] ::) s_list</p>

(continued)

F90 Syntax (continued)

```

CHARACTER [(KIND=i_kind) [, LEN=i_value | *)] [, attr_list ::] s_list
CLOSE (i_value) ! unit number
CLOSE ([UNIT=j_value [, ERR=i_label] [, IOSTAT=i_variable] [, STATUS=exp])
COMPLEX :: variable_list
COMPLEX [(KIND=j_kind)] [, attr_list ::] variable_list
CONTAINS ! internal definitions follow
CYCLE ! current do only for a purpose
CYCLE [nested_do_name] ! and terminate its sub_do's for a purpose
DEALLOCATE (array_name)
DEALLOCATE (array_name [, STAT=status] [, array_name [, STAT=status]])
DIMENSION array_name(extents) [, array_name(extents)]
DO ! forever
DO i_variable = i_start, i_stop ! loop_name_or_purpose
DO [i_variable = i_start, i_stop [, i_inc]] ! loop_name_or_purpose
DO [i_label,] [i_variable = i_start, i_stop [, i_inc]] ! loop_name
[loop_name:] DO [i_variable = i_start, i_stop [, i_inc]] ! purpose
[loop_name:] DO [i_label,] [i_variable = i_start, i_stop [, i_inc]]
DO WHILE (logical_expression) ! obsolete, use DO-EXIT pair
DO [i_label,] WHILE (logical_expression) ! obsolete-obsolete
[name:] DO [i_label,] WHILE (logical_expression) ! obsolete
ELSE [if_name]
ELSE IF (logical_expression) THEN [if_name]
ELSE WHERE (logical_expression)
END [name] ! purpose
END DO [do_name] ! purpose
END FUNCTION [function_name] ! purpose
END IF [if_name] ! purpose
END INTERFACE ! purpose
END MODULE [module_name] ! purpose
END PROGRAM [program_name] ! purpose
END SELECT [select_name] ! purpose
END SUBROUTINE [name] ! purpose
END TYPE [type_name] ! purpose
END WHERE ! purpose
ENDFILE i_exp ! for file unit number
ENDFILE ([UNIT=j_value [, IOSTAT=i_variable] [, ERR=i_label])
ENTRY entry_name ([args]) [RESULT(variable_name)]
EXIT ! current do only for a purpose
EXIT [nested_do_name] ! and its sub_do's for a purpose
EXTERNAL program_list
i_label FORMAT (specification_and_edit_list)
FUNCTION name ([args]) ! purpose
FUNCTION name ([args]) [RESULT(variable_name)] ! purpose
[type] [RECURSIVE] FUNCTION name ([args]) [RESULT(variable_name)]
[RECURSIVE] [type] FUNCTION name ([args]) [RESULT(variable_name)]
GO TO i_label ! for_a_reason
IF (logical_expression) executable_statement
[name:] IF (logical_expression) THEN ! state_purpose
IMPLICIT type (letter_list) ! F77 (a-h,o-z) real, (i-n) integer
IMPLICIT NONE ! F90 recommended default
INCLUDE source_file_path_name ! purpose
INQUIRE ([FILE=]name_string [, see_INQUIRE_table]) ! re file

```

(continued)

F90 Syntax (continued)

INQUIRE ([NAME=]s \_variable [, see \_INQUIRE \_table]) ! re file  
 INQUIRE (IOLength=i \_variable [, see \_INQUIRE \_table]) ! re output  
 INQUIRE ([UNIT=]i \_value [, see \_INQUIRE \_table]) ! re unit  
 INTEGER [::] variable \_list  
 INTEGER [(KIND=]i \_kind)] [, attr \_list ::] variable \_list  
 INTENT ([IN | INOUT | OUT]) argument \_list  
 INTERFACE ASSIGNMENT (+ | - | \* | / | = | \*\*) ! user extension  
 INTERFACE OPERATOR (.operator.) ! user defined  
 INTERFACE [interface \_name]  
 INTRINSIC function \_list  
 LOGICAL [::] variable \_list  
 LOGICAL [(KIND=]i \_kind)] [, attr \_list ::] variable \_list  
 MODULE PROCEDURE program \_list  
 MODULE module \_name ! purpose  
 NULLIFY (pointer \_list)  
 OPEN (i \_value) ! unit number  
 OPEN ([UNIT=]i \_value [, ERR=i \_label] [, IOSTAT=i \_variable] [, other \_spec])  
 OPTIONAL [::] argument \_list  
 PARAMETER (variable=value [, variable=value])  
 POINTER [::] name[(extent)] [, name[(extent)]] ! purpose  
 PRINT \*, output \_list ! default free format  
 PRINT \*, (io \_implied \_do) ! default free format  
 PRINT '(formats)', output \_list ! formatted  
 PRINT '(formats)', (io \_implied \_do) ! formatted  
 PRIVATE [::] module \_variable \_list ! limit access  
 PROGRAM [program \_name] ! purpose  
 PUBLIC [::] module \_variable \_list ! default access  
 READ \*, input \_list ! default free format  
 READ \*, (io \_implied \_do) ! default free format  
 READ '(formats)', input \_list ! formatted  
 READ '(formats)', (io \_implied \_do) ! formatted  
 READ ([UNIT=]i \_value, [FMT=]i \_label [, io \_spec \_list]), input \_list ! formatted  
 READ ([UNIT=]i \_value, s \_variable [, io \_spec \_list]), input \_list ! formatted  
 READ ([UNIT=]i \_value, '(formats)' [, io \_spec \_list]), input \_list ! formatted  
 READ (i \_value), input \_list ! binary read  
 READ ([UNIT=]i \_value, [, io \_spec \_list]), input \_list ! binary read  
 READ (s \_variable, [FMT=]i \_label), input \_list ! internal file type change  
 READ ([UNIT=]s \_variable, [FMT=]i \_label [, io \_spec \_list]), input \_list ! internal file change  
 REAL [::] variable \_list  
 REAL [(KIND=]i \_kind)] [, attr \_list ::] variable \_list  
 RECURSIVE FUNCTION name ([args]) [RESULT(variable \_name)] ! purpose  
 [type] RECURSIVE FUNCTION name ([args]) [RESULT(variable \_name)] ! purpose  
 RECURSIVE SUBROUTINE name ([args]) ! purpose  
 RETURN ! from subroutine \_name  
 REWIND i \_exp ! file unit number  
 REWIND ([UNIT=]i \_value [, IOSTAT=i \_variable] [, ERR=i \_label])  
 SAVE [::] variable \_list  
 [name:] SELECT CASE (value)  
 SEQUENCE  
 STOP ['stop \_message \_string']  
 SUBROUTINE name [(args)] ! purpose  
 SUBROUTINE name [(args)] [args, optional \_args] ! purpose

(continued)

F90 Syntax (continued)
[RECURSIVE] SUBROUTINE name [(args)] ! purpose TARGET [::] name[(extent)] [, name[(extent)]] TYPE (type_name) [, attr_list] :: variable_list TYPE [, PRIVATE   PUBLIC] name USE module_name [, ONLY: list_in_module_name] ! purpose USE module_name [, new_var_or_sub=>old_name] ! purpose WHERE (logical_array_expression) ! then WHERE (logical_array_expression) array_variable = array_expression WRITE *, output_list ! default free format WRITE *, (io_implied_do) ! default free format WRITE '(formats)', output_list ! formatted write WRITE '(formats)', (io_implied_do) ! formatted write WRITE ([UNIT=j_i_value, [FMT=j_i_label [, io_spec_list]], output_list ! formatted write WRITE ([UNIT=j_i_value, s_variable [, io_spec_list]], output_list ! formatted write WRITE ([UNIT=j_i_value, '(formats)' [, io_spec_list]], output_list ! formatted write WRITE (i_value), output_list ! binary write WRITE (i_value), (io_implied_do) ! binary write WRITE ([UNIT=j_i_value, [, io_spec_list]], output_list ! binary write WRITE (s_variable, [FMT=j_i_label], output_list ! internal file type change WRITE ([UNIT=s_variable, [FMT=j_i_label [, io_spec_list]], output_list ! internal file change

Obsolescent statements are those from Fortran77 that are redundant and for which better methods are available in both Fortran77 and Fortran90.

Obsolete Syntax
ASSIGN i_label TO i_variable BLOCK DATA [block_data_name] COMMON [/common_block_name/] r_variable_list, i_variable_list [i_label] CONTINUE ! from do [do_name] DATA variable_list / value_list / DATA (array_implied_do) / value_list / DOUBLE PRECISION [, attr_list] :: variable_list DO [i_label,] [r_variable = r_start, r_stop [, r_inc]] ! real control DO _CONTINUE_pair [name:] DO [i_label,] WHILE (logical_expression) ! obsolete END BLOCK DATA [block_data_name] EQUIVALENCE (variable_1, variable_2) [, (variable_3, variable_4)] GO TO (i_label_1, i_label_2, ..., i_label_n)[, ] i_variable IF (arithmetic_exp) i_label_neg, i_label_zero, i_label_pos NAMELIST /group_name/ variable_list PAUSE ! for human action RETURN alternates statement_function (args) = expression

The attributes lists for the type declarations, e.g. REAL, are ALLOCATABLE, DIMENSION, INTENT, OPTIONAL, KIND, POINTER, PARAMETER, PRIVATE, PUBLIC, SAVE, and TARGET; those for OPEN and CLOSE are ACCESS, ACTION, BLANK, and DELIM; while those for READ and WRITE are ADVANCE, END, EOR, ERR, and FMT.

	MATLAB	C++	F90
Addition $C = A + B$	$C=A+B$	<pre>for (i=0; i&lt;10; i++){   for (j=0; j&lt;10; j++){     C[i][j]=A[i][j]+B[i][j];   } }</pre>	$C=A+B$
Multiplication $C = AB$	$C=A*B$	<pre>for (i=0; i&lt;10; i++){   for (j=0; j&lt;10; j++){     C[i][j] = 0;     for (k=0; k&lt;10; k++){       C[i][j] += A[i][k]*B[k][j];     }   } }</pre>	$C=matmul(A,B)$
Scalar multiplication $C = aB$	$C=a*B$	<pre>for (i=0; i&lt;10; i++){   for (j=0; j &lt; 10; j++){     C[i][j] = a*B[i][j];   } }</pre>	$C=a*B$
Matrix inverse $B = A^{-1}$	$B=inv(A)$	$a$	$B=inv(A)^a$

<sup>a</sup>Neither C++ nor F90 have matrix inverse functions as part of their language definitions nor as part of standard collections of mathematical functions (like those listed in Table 4.7). Instead, a special function, usually drawn from a library of numerical functions, or a user defined operation, must be used.

**Table B.43:** Elementary matrix computational routines.

C++	<pre>int* point, vector, matrix ... point = new type_tag  vector = new type_tag [space_1] if (vector == 0) {error_process} matrix = new type_tag [space_1 * space_2] ... delete matrix ... delete vector delete point</pre>
F90	<pre>type_tag, pointer, allocatable :: point type_tag, allocatable :: vector (:), matrix (:,:) ... allocate (point) allocate (vector (space_1), STAT = my_int) if (my_int /= 0) error_process allocate (matrix (space_1, space_2)) ... deallocate (matrix) if (associated (point, target_name)) pointer_action... if (allocated (matrix)) matrix_action... ... deallocate (vector) deallocate (point)</pre>

**Table B.44:** Dynamic allocation of arrays and pointers.

```

SUBROUTINE AUTO_ARRAYS (M,N, OTHER)
USE GLOBAL_CONSTANTS ! FOR INTEGER K
IMPLICIT NONE
INTEGER, INTENT (IN) :: M,N
type_tag, INTENT (OUT) :: OTHER (M,N) ! dummy array

! Automatic array allocations
type_tag :: FROM_USE (K)
type_tag :: FROM_ARG (M)
type_tag :: FROM_MIX (K,N)
...
! Automatic deallocation at end of scope
END SUBROUTINE AUTO_ARRAYS

```

**Table B.45:** Automatic memory management of local scope arrays.

```

module derived_class_name
  use base_class_name
  ! new attribute declarations, if any
  ...
contains

  ! new member definitions
  ...
end module derived_class_name

```

**Table B.46:** F90 Single Inheritance Form.

```

module derived_class_name
  use base_class_name, only: list_of_entities
  ! new attribute declarations, if any
  ...
contains

  ! new member definitions
  ...
end module derived_class_name

```

**Table B.47:** F90 Selective Single Inheritance Form.

```

module derived_class_name
  use base_class_name, local_name => base_entity_name
  ! new attribute declarations, if any
  ...
contains

  ! new member definitions
  ...
end module derived_class_name

```

**Table B.48:** F90 Single Inheritance Form, with Local Renaming.

```

module derived_class_name
    use base1_class_name
    use base2_class_name
    use base3_class_name, only: list_of_entities
    use base4_class_name, local_name => base_entity_name
    ! new attribute declarations, if any
    ...
contains

    ! new member definitions
    ...
end module derived_class_name

```

**Table B.49:** F90 Multiple Selective Inheritance with Renaming.



## Examples of F90 Statements

The following is a list of examples of the recommended Fortran90 statements. Some have been declared obsolete, and are expected to be deleted in future standards. Thus, they should not be utilized in new programs. They are noted in the comments. In some cases the most common simple form of a statement is shown along with it's more general options. Note that the new attribute terminator symbol `::` is always optional, but its use is recommended. While Fortran is not case-sensitive, this table employs upper case letters to denote standard features, and lower case letters for user supplied information. The following abbreviations are employed: arg=argument, attr=attribute, exp=expression, i\_=integer, l\_=logical, r\_=real, s\_=string, spec=specifier, z\_=complex.

Recall that F90 allows variable names to be 31 characters long and they may include an underscore (but F77 allows only six characters and no underscore). F90 lines may contain up to 132 characters (but just 72 in F77). All standard F77 statements are a sub-set of F90.

The attributes lists for the type declarations, e.g. REAL, are ALLOCATABLE, DIMENSION, INTENT, OPTIONAL, KIND, POINTER, PARAMETER, PRIVATE, PUBLIC, SAVE, and TARGET. Those optional attributes for OPEN are ACCESS = [DIRECT, SEQUENTIAL], ACTION = [READ, READWRITE, WRITE], BLANK = [NULL, ZERO], DELIM = [APOSTROPHE, NONE, QUOTE], ERR = i\_label, FILE = s\_name, FORM = [FORMATTED, UNFORMATTED], IOSTAT = i\_var, PAD = [NO, YES], POSITION = [APPEND, ASIS, REWIND], RECL = i\_len, STATUS = [NEW, OLD, REPLACE, SEARCH, UNKNOWN], and UNIT = i\_unit; while CLOSE utilizes only ERR, IOSTAT, STATUS, and UNIT.

The io\_spec\_list options for READ and WRITE are ADVANCE = [NO, YES], END = i\_label, EOR = i\_label, ERR = i\_label, FMT = [\* , i\_label, s\_var], IOSTAT = i\_var, NML = var\_list, REC = i\_exp, SIZE = i\_size, and UNIT = i\_unit.

Fortran Statement Examples		
Name	Examples	Comments
Allocatable	ALLOCATABLE :: force, stiffness ALLOCATABLE :: force(:), stiffness(:,:)	By name Ranks
Allocate	ALLOCATE (hyper_matrix(5, 10, 3)) ALLOCATE (force(m)) ALLOCATE (array_name(3, 3, 3, 3), STAT=i_err)	Error status
Assign	ASSIGN 9 TO k	Obsolete
Assignment	c = 'b' s = "abc" s = c // 'abc' s = string(j:m) s_fmt = '(2F5.1)'	Character String Concatenation Sub-string Stored format
	l = l_1 .OR. l_2 l = m <= 80 poor = (final >= 60) .AND. (final < 70) proceed = .TRUE.	Logical
	n = n + 1 x = b'1010' z = (0.0, 1.0) r = SQRT (5.) converged = ( ABS (x0 - x) < 2*SPACING (x) ) x = z'B' k = 123 x = o'12' r = 321. a = 23. ; j = 120 ; ans = .TRUE.;	Arithmetic Binary Complex Function  Hexadecimal Integer Octal Real Semicolon
	k = SELECTED_INTEGER_KIND (20) m = SELECTED_REAL_KIND (16, 30)	Kind

(continued)

Fortran Statement Examples (continued)		
Name	Examples	Comments
	long = SELECTED_REAL_KIND (9, 20) pi = 3.1459265_long	
	a = b + c d = MATMUL (a, b) e = TRANSPOSE (d) f = 0 ; g = (/ 2. , 4. , 6. /) B = A1:, n:(-1) x = (/ (k, k = 0, n) /) * d	Matrix add Matrix multiply Matrix transpose Matrix initialize Matrix flipped Implied do
	kth_row => a(k,:) corners => a(1:n:(n-1), 1:m:(m-1)) p_2 => r	Pointer
	student_record%rank = 51 patient_data%city = 'houston'	Defined type
	sqrt(x) = DSQRT(x) ! function statement	Obsolete
Backspace	BACKSPACE i_exp BACKSPACE 8 BACKSPACE (UNIT=9, IOSTAT=i, ERR=5) BACKSPACE (9, IOSTAT=io_ok, ERR=99) BACKSPACE (UNIT=9, IOSTAT=io_ok, ERR=99) BACKSPACE (8, IOSTAT=io_ok)	Compute unit Unit Error go to  I/O status
Block Data	BLOCK DATA ! Obsolete BLOCK DATA winter ! Obsolete	Named
C	C in column one denotes a comment line in F77 * in column one denotes a comment line in F77 ! anywhere starts a comment line in F90	Obsolete Obsolete
Call	CALL sub1 (a, b) CALL sub2 (a, b, *5) ! Obsolete, use CASE CALL sub3 CALL subroutine_name (args, optional_args)	Alt return to 5 No arguments Optional arg
Case	CASE (range_list) CASE (range_list) select_name	See SELECT Named
Case Default	CASE DEFAULT CASE DEFAULT select_name	See SELECT Named
Character	CHARACTER (80) s, s_2*3(4) CHARACTER *16 a, b, c CHARACTER * home_team CHARACTER (*), INTENT(IN) :: home_team CHARACTER (LEN=3) :: b = 'xyz' CHARACTER LEN=40 :: monday, wednesday, friday CHARACTER (LEN=40), attr_list :: last, first, middle CHARACTER (40), attr_list :: name, state CHARACTER (*), PARAMETER :: reply = "Invalid Data" CHARACTER (*, KIND=greek), attr_list :: s1_list CHARACTER (*, KIND=greek), attr_list :: last, first, middle CHARACTER (KIND=cyrillic, LEN=40) :: name, state CHARACTER (KIND=cyrillic, *), attr_list :: s_list	:: recommended Intent Initialize b   Kind
Close	CLOSE (7) CLOSE (UNIT=k) CLOSE (UNIT=8, ERR=90, IOSTAT=i) CLOSE (8, ERR=99, IOSTAT=io_ok, STATUS='KEEP')	Unit number  Error go to I/O status

(continued)

<i>Fortran Statement Examples (continued)</i>		
<i>Name</i>	<i>Examples</i>	<i>Comments</i>
	CLOSE (9, ERR=99, IOSTAT=io, STATUS='DELETE') CLOSE (UNIT=8, ERR=95, IOSTAT=io_ok)	File status
Common	COMMON / name / h, p, t ! Obsolete COMMON p, d, q(m,n) ! Obsolete	Named common Blank common
Complex	COMPLEX u, v, w(3, 6) COMPLEX :: u = (1.0,1.0), v = (1.0,10.0) COMPLEX :: variable_list COMPLEX attr_list :: variable_list COMPLEX (KIND=i2_kind), attr_list :: variable_list	:: recommended Initialize u and v  Kind
Contains	CONTAINS	Internal definitions
	CONTAINS FUNCTION mine (b) ... END FUNCTION mine	Or subroutines
Continuation	! any non-block character in column 6 flags continuation & at the end flags continuation to next line & at the beginning flags continuation from above line	F77 obsolete F90 standard
	a_long_name = a_constant_value* & another_value ! on following line	
	a_long_name_here_is_set_to = value & * another_value ! continued from above	
Continue	100 CONTINUE	Obsolete
Cycle	CYCLE CYCLE nested_do_name	Current do only Terminate sub_dos
Data	DATA a, s / 4.01, 'z' / DATA s_fmt / '(2F5.1)' / DATA (r(k), k=1,3) / 0.7, 0.8, 1.9 / DATA array (4,4) / 1.0 / DATA bit_val / b'0011111' /	Obsolete Stored format Implied do Single value Binary
Deallocate	DEALLOCATE (force) DEALLOCATE (force, STAT=i_err)	File name Error status
Dimension	DIMENSION array (4, 4) DIMENSION v(1000), w(3) = (/ 1., 2., 4. /) DIMENSION force(20), stiffness(:, :) DIMENSION (5,10,3) :: triplet	Initialize w  :: recommended
	INTEGER, DIMENSION (:, :) :: material, nodes_list REAL, DIMENSION(m, n) :: a, b REAL, DIMENSION (:, :) :: force, stiffness	Typed
	REAL, DIMENSION (5,10,3), INTENT( IN) :: triplet	Intent
Do	DO 100 j = init, last, incr ! Obsolete ... 100 CONTINUE	Labeled do  Obsolete
	DO j = init, last ... END DO	Unlabeled do
	DO ! forever ... END DO ! forever	Unlabeled do
	DO WHILE (diff <= delta) ... END DO	Unlabeled while
	DO 100 WHILE (diff <= delta) ! Obsolete	Labeled while

(continued)

<i>Fortran Statement Examples (continued)</i>		
<i>Name</i>	<i>Examples</i>	<i>Comments</i>
	...	
	100 CONTINUE	Obsolete
	DO	Forever
	DO k = i_start, i_stop	Integer range
	DO k = i_start, i_stop, i_inc	Increment
	DO 10, k = i_start, i_stop	Obsolete
	do_name: DO k = i_start, i_stop, i_inc	Named
	do_name: DO 10, k = i_start, i_stop, i_inc	Named label
	DO 10, r_variable = r_start, r_stop, r_inc ! Obsolete	Real range
Do While	DO WHILE (.NOT. converged) DO 10, WHILE (.NOT. converged) do_name: DO 10, WHILE (.NOT. converged)	Use DO-EXIT pair Obsolete Obsolete
Double Precision	DOUBLE PRECISION a, d, y(2) DOUBLE PRECISION :: a, d = 1.2D3, y(2) DOUBLE PRECISION, attr_list :: variable_list	Obsolete Initialize D Obsolete
Else	ELSE ELSE leap_year	Then Named
Else If	ELSE IF (k > 50) THEN ELSE IF (days_in_year == 364) THEN ELSE IF (days_in_year == 364) THEN leap_year	Named
Elsewhere	ELSEWHERE	See WHERE
End	END END name	Named
End Block Data	END BLOCK DATA END BLOCK DATA block_data_name	Obsolete Obsolete
End Do	END DO END DO do_name	Named
End Function	END FUNCTION function_name END FUNCTION	
End If	END IF leap_year END IF	Named
End Interface	END INTERFACE	
End Module	END MODULE my_matrix_operators END MODULE	
End Program	END PROGRAM program_name END PROGRAM	
End Select	END SELECT select_name END SELECT	Named
End Subroutine	END SUBROUTINE name END SUBROUTINE	
End Type	END TYPE type_name END TYPE	See TYPE
End Where	END WHERE	See WHERE
Endfile	ENDFILE i_exp ENDFILE (UNIT=k) ENDFILE k ENDFILE (UNIT=8, ERR=95) ENDFILE (7, IOSTAT=io_ok, ERR=99) ENDFILE (UNIT=8, IOSTAT=k, ERR=9) ENDFILE (UNIT=9, IOSTAT=io_ok, ERR=99)	Compute unit Unit number  Error go to I/O status

(continued)

<i>Fortran Statement Examples (continued)</i>		
<i>Name</i>	<i>Examples</i>	<i>Comments</i>
Entry	ENTRY sec1 (x, y) ENTRY sec2 (a1, a2, *4) ! Obsolete, use CASE ENTRY section ENTRY entry _name RESULT(variable _name)	Arguments Alternate return to 4 No arguments Result
Equivalence	EQUIVALENCE (v (1), a (1,1)) EQUIVALENCE (v, a) EQUIVALENCE (x, v(10)), (p, q, d)	Obsolete
Exit	EXIT EXIT nested _do _name	Current do only Current & sub-dos
External	EXTERNAL my _program	
Format	10 FORMAT (2X, 2I3, 3F6.1, 4E12.2, 2A6, 3L2 ) 10 FORMAT (// 2D6.1, 3G12.2) 10 FORMAT (2I3.3, 3G6.1E3, 4E12.2E3) 10 FORMAT ('a quoted string', "another", I2) 10 FORMAT (1X, T10, A1, T20, A1) 10 FORMAT (5X, TR10, A1, TR10, A1, TL5, A1) 10 FORMAT ("Init=", I2, :, 3X, "Last=", I2) 10 FORMAT ('Octal ', o6, ', Hex ' z6) 10 FORMAT (specification _and _edit _list)	X I F E A L D, G Exponent w Strings Tabs Tab right, left : stop if empty Octal, hex
Function	FUNCTION z (a, b) FUNCTION w (e, d) RESULT (a) FUNCTION name (args) FUNCTION name FUNCTION name (args) RESULT(variable _name)	Arguments Result  No argument
	INTEGER FUNCTION n (j, k) INTEGER FUNCTION name (args) COMPLEX RECURSIVE FUNCTION dat (args) RECURSIVE REAL FUNCTION name (args)	Type
Go To	GO TO 99	Unconditional
	GO TO (10,20,35,95), i _variable ! Obsolete	Computed
If	IF (arithmetic _exp) 95, 10, 20 ! Obsolete	Arithmetic
	IF (logic) RETURN IF (logic) n = n + 2	Logical if
	IF (logic) THEN n = n + 1 k = k + 1 END IF	if block
	leap _year: IF (logical _expression) THEN	Named
	IF (logic) THEN n = n + 1 ELSE k = k + 1 END IF	if else block
	IF (c == 'a') THEN na = na + 1 CALL sub _a ELSE IF (c == 'b') THEN nb = nb + 1 ELSE IF (c == 'c') THEN nc = nc + 1	if else-if block  (Use CASE)

(continued)

<i>Fortran Statement Examples (continued)</i>		
<i>Name</i>	<i>Examples</i>	<i>Comments</i>
	CALL sub _c END IF	
Implicit Type	IMPLICIT INTEGER (i-n) IMPLICIT REAL (a-h,o-z) IMPLICIT NONE IMPLICIT CHARACTER *10 (f,l) IMPLICIT COMPLEX (a-c,z) IMPLICIT TYPE (color) (b,g,r) IMPLICIT LOGICAL (KIND=bit) (m)	F77 default F77 default Recommended F90 Character Complex Derived type Logical
Include	INCLUDE 'path/source.f'	
Inquire	INQUIRE (UNIT=3, OPENED=t_or _f)	Opened
	INQUIRE (FILE='mydata', EXIST=t_or _f)	Exists
	INQUIRE (UNIT=3, OPENED=ok, IOSTAT=k)	I/O status
	INQUIRE (FILE='name _string', see _INQUIRE _table)	Re file
	INQUIRE (NAME=s _variable, see _INQUIRE _table)	Re file
Integer	INQUIRE (IOLENGTH=i _var, see _INQUIRE _table)	Re output
	INQUIRE (7, see _INQUIRE _table)	Re unit
	INQUIRE (UNIT=8, see _INQUIRE _table)	Re unit
Integer	INTEGER c, d(4) INTEGER (long), attr _list :: variable _list INTEGER, DIMENSION (4) :: a, d, e INTEGER, ALLOCATABLE, DIMENSION(:,:) :: a, b INTEGER :: a = 100, b, c = 9 INTEGER :: i, j, k, l, m, n, month, year = 1996 INTEGER, attr _list :: variable _list	:: Recommended  Allocatable Initialize a & c
	INTEGER (KIND=i2 _kind), attr _list :: variable _list	Kind
Intent	INTENT (IN) :: credit _card _owners INTENT (INOUT) :: amount _due INTENT (OUT) income _rank	
Interface	INTERFACE ASSIGNMENT (=) INTERFACE OPERATOR (+) INTERFACE OPERATOR (-) INTERFACE OPERATOR (/) INTERFACE OPERATOR (*) INTERFACE OPERATOR (**) INTERFACE OPERATOR (.operator.) INTERFACE INTERFACE interface _name	User extension User extension User extension User extension User extension User extension User defined
Intrinsic	INTRINSIC SQRT, EXP	Functions
Logical	LOGICAL c LOGICAL, ALLOCATABLE :: mask(:), mask _2(:,:) LOGICAL (KIND = byte) :: flag, status LOGICAL :: b = .FALSE., c	:: recommended Allocatable Kind Initialize b
Module	MODULE PROCEDURE mat _x _mat, mat _x _vec MODULE my _matrix _operators	Generics
Namelist	NAMelist /data/ s, n, d	Obsolete
Nullify	NULLIFY (pointer _list)	
Open	OPEN (7) OPEN (UNIT=3, FILE="data.test ") OPEN (UNIT=2, FILE="data ", STATUS = "old ")	Unit number Name File status

(continued)

<i>Fortran Statement Examples (continued)</i>		
<i>Name</i>	<i>Examples</i>	<i>Comments</i>
	OPEN (UNIT=3, IOSTAT=k) OPEN (9, ERR = 12, ACCESS = "direct") OPEN (8, ERR=99, IOSTAT=io_ok) OPEN (UNIT=8, ERR=99, IOSTAT=io_ok)	I/O status Access type Error go to
Optional	OPTIONAL slow, fast OPTIONAL :: argument_list	Argument list
Parameter	PARAMETER (a="xyz"), (pi=3.14159) PARAMETER (a="z", pi=3.14159) PARAMETER (x=11, y = x/3) PARAMETER, REAL :: weight = 245.6	Character Real Computed Type
Pause	PAUSE ! for human action	Obsolete
Pointer	POINTER current, last POINTER :: name(4,5) REAL, POINTER :: y(:), x(:,,:)	:: recommended Rank Type
Print	PRINT *, a, j PRINT *, output_list PRINT *, (io_implied_do) PRINT *, "The square root of", n, ' is ', SQRT(n) PRINT *, (4*k-1, k=1,10,3)	List-directed Default unformatted Implied do Function
	PRINT 10, a, j	Formatted
	PRINT 10, m_array	Array
	PRINT 10, (m(i), i = j,k)	Implied do
	PRINT 10, s(j:k)	Substring
	PRINT '(A6, I3)', a, j PRINT FMT='(A6, I3)', a, j	Character, integer Included format
	PRINT data_namelist ! Obsolete	Namelist
	PRINT '(formats)', output_list PRINT '(formats)', (io_implied_do) PRINT '(I4)', (2*k, k=1,5)	Formatted Implied do
	PRIVATE PRIVATE :: module_variable_list	Specific items
	PROGRAM my_job PROGRAM	
Public	PUBLIC PUBLIC :: module_variable_list	Specific items
Read	READ *, a, j	List-directed
	READ 1, a, j	Formatted
	READ 10, m_array	Formatted array
	READ 10, (m(i), i=j, k)	Implied do
	READ 10, s(i:k)	Substring
	READ '(A6, I3)', a, i	Character, integer
	READ (1, 2) x, y READ (UNIT=1, FMT=2) x, y READ (1, 2, ERR=8, END=9) x, y READ (UNIT=1, FMT=2, ERR=8, END=9) x, y	Formatted file End of file go to Error go to
	READ (*, 2) x, y	Formatted, std out
	READ (*, 10) m_array	Unformatted array
	READ (*, 10) (m(i), i=j, k)	Implied do
	READ (*, 10) s(i:k)	Substring
	READ (1, *) x, y	Unformatted file

(continued)

Fortran Statement Examples (continued)		
Name	Examples	Comments
	READ (*, *) x, y	Unformatted, std out
	READ (1, '(A6, I3)') x, y	Character, integer
	READ (1, FMT='(A6, I3)') x, y	Included format
	READ (1, s_fmt) x, y	Format in a string
	READ (1, FMT=s_fmt) x, y	
	READ (*, NML=data) ! Obsolete	Namelist read
	READ (1, NML=data) ! Obsolete	Namelist from a file
	READ (1, END=8, ERR=9) x, y	Unformatted
	READ (s2, 1, ERR=9) x	Internal, formatted
	READ (s2, *, ERR=9) x	Unformatted
	READ (s2, REC=4, END=8) x	Internal, direct
	READ (1, REC=3) v	Unformatted direct
	READ (1, 2, REC=3) v	Formatted direct
	READ *, input_list	Default unformatted
	READ *, (io_implied_do)	Implied do
	READ *, (a(j,:), j=1, rows)	
	READ '(formats)', input_list	Formatted read
	READ '(formats)', (io_implied_do)	Formatted read
	READ '(5I5, (5I5))', (num(k), k=1, n)	
	READ (8, FMT=20), input_list	Formatted
	READ (8, FMT=20, ADVANCE='NO'), input	Advance
	READ (9, FMT=20, io_spec_list), input_list	I/O Specification
	READ (UNIT=7, 20, io_spec_list), input_list	
	READ (UNIT=8, FMT=10, io_spec_list), input	
Real	READ (7, s_fmt, io_spec_list), input_list	Stored format
	READ (UNIT=7, s_fmt, io_spec_list), input	
	READ (9, '(formats)', io_spec_list), input_list	Inline format
	READ (UNIT=9, '(formats)', io_spec_list), input	
	READ (8), input_list	Binary read
	READ (UNIT=7), input_list	
	READ (8, io_spec_list), input_list	I/O Specification
	READ (UNIT=9, io_spec_list), input_list	
	READ (s_variable, FMT=20), input_list	Internal file,
	READ (UNIT=s_variable, 10, io_spec_list), input	type change
Recursive	REAL*4	:: recommended
	REAL :: r, m(9)	
	REAL*16 a, b, c	Quad Precision
	REAL*8, DIMENSION(n) :: a, b, c	Double Precision
Return	REAL :: a = 3.14, b, c = 100.0	Initialize a & c
	REAL :: variable_list	
	REAL, attr_list :: variable_list	
	REAL, POINTER :: a(:, :)	
	REAL (KIND=i2_kind), attr_list :: variable_list	Kind
	REAL (double), attr_list :: variable_list	
	RECURSIVE FUNCTION name	Function
	RECURSIVE FUNCTION a(n) RESULT(fac)	Result
	INTEGER RECURSIVE FUNCTION name (args)	
	RECURSIVE SUBROUTINE name (args)	Subroutine
	RECURSIVE SUBROUTINE name	
	RETURN	Standard return

(continued)



<i>Fortran Statement Examples (continued)</i>		
<i>Name</i>	<i>Examples</i>	<i>Comments</i>
Rewind	REWIND i_exp REWIND 2 REWIND k REWIND (UNIT=8, IOSTAT=k, ERR=9) REWIND (UNIT=8, ERR=95) REWIND (8, IOSTAT=io_ok, ERR=99)	Compute unit Unit number  Error go to  I/O status
Save	SAVE a, /name/, c SAVE SAVE :: variable_list	Scalars, common Everything
Select Case	SELECT CASE (value) name: SELECT CASE (value)	Named
	u_or_l SELECT CASE (letter) CASE ("a":"z") ! lower case lower = .TRUE. CASE ("A":"Z") ! upper case lower = .FALSE. CASE DEFAULT ! not a letter PRINT *, "Symbol is not a letter", letter lower = .FALSE. END SELECT u_or_l	Block
Sequence	SEQUENCE	Forced storage
Stop	STOP STOP "invalid data "	With message
Subroutine	SUBROUTINE sub1 (a, b) SUBROUTINE sub1 SUBROUTINE name (args, optional_args) SUBROUTINE sub3 (a, b, *9) ! Obsolete, use CASE RECURSIVE SUBROUTINE sub2 (a, b)	No arguments Optional arguments Return to 9 Recursive
Target	TARGET :: name, name_2 TARGET :: name(4,5), name_2(3)	See Pointer
Type Declaration	TYPE (person) car_pool(5) TYPE (color), DIMENSION(256) :: hues TYPE (type_name), attr_list :: variable_list TYPE (person), DIMENSION (n) :: address_book TYPE (type_name) :: variable_list	User defined type
	TYPE (student_record) CHARACTER (name_len) :: last, first INTEGER :: rank END TYPE student_record	Definition block
Type Statement	TYPE, PRIVATE name TYPE, PUBLIC :: name	Access
Use	USE module_name USE module_name, ONLY: list_in_module_name USE module_name, var_subr_fun_name => old_name	Only Rename
Where	WHERE (logical_array_mask)	Then
	WHERE ( a_array > 0.0 ) sqrt_a = SQRT(a_array) END WHERE	Where block
	WHERE ( mask > 0.0 ) a_array = mask	Elsewhere block

(continued)

Fortran Statement Examples (continued)		
Name	Examples	Comments
Write	ELSEWHERE a_array = 0.0 END WHERE	
	WHERE (a_array>0) b_array = SQRT(a_array)	Statement
	WRITE (*, 10) s(j:k)	Substring
	WRITE (1, *) x, y WRITE (*, *) x, y	Unformatted file Unformatted
	WRITE (1, '(A6, I3)') x, y WRITE (1, FMT='(A6, I3)') x, y	Character, integer Included format
	WRITE (1, s_fmt) x, y WRITE (1, FMT=s_fmt) x, y	Stored format string
	WRITE (*, NML=data) ! Obsolete WRITE (1, NML=data) ! Obsolete	Namelist to stdout Namelist to a file
	WRITE (1, END=8, ERR=9) x, y	Unformatted
	WRITE (1, REC=3) v	Unformatted direct
	WRITE (1, 2, REC=3) v	Formatted direct
	WRITE (s2, 1, ERR=9) x	Internal, format
	WRITE (s2, *, ERR=9) x	Unformatted
	WRITE (s2, REC=4, END=8) x	Internal, direct
	WRITE *, output_list WRITE *, (io_implied_do) WRITE *, ((a(i, j), j=1, cols), i=1, rows)	Unformatted Implied do
	WRITE '(formats)', output_list WRITE '(formats)', (io_implied_do)	Formatted write Implied do
	WRITE (7, 10, ADVANCE='NO'), output_list WRITE (8, 10, io_spec_list), output_list WRITE (9, FMT=20, io_spec_list), output_list WRITE (UNIT=7, 10, io_spec_list), output_list	Advance I/O specification
	WRITE (9, s_fmt, io_spec_list), output_list WRITE (UNIT=8, s_fmt, io_spec_list), output	Stored format
	WRITE (9, '(formats)', io_spec_list), output_list WRITE (UNIT=7, '(formats)', io_spec_list), output	Inline format
	WRITE (8), output_list WRITE (7), (io_implied_do) WRITE (8, ADVANCE='NO'), output_list WRITE (9, io_spec_list), output_list WRITE (UNIT=9, io_spec_list), output_list	Binary write Implied do Advance I/O specification
	WRITE (s_variable, FMT=20), output_list WRITE (UNIT=s_variable, FMT=20), output_list WRITE (s_variable, 20, io_spec_list), output_list WRITE (UNIT=s_var, FMT=20, io_spec), output	Internal file I/O specification

## Appendix C

# Selected Exercise Solutions

### C.1 Problem 1.8.1 : Checking trigonometric identities

The Fortran 90 program and output follow. The error levels are due to the fact that F90 defaults to single precision reals. F90 is easily extended to double precision, and in theory supports any level of user specified precision. For simplicity the F77 default naming convention for integers and reals is used. That is not a good practice since safety dictates declaring the type of each variable at the beginning of each program. (Try changing the reals to double precision to verify that the error is indeed reduced.)

```
[ 1] implicit none
[ 2] integer :: k,n = 16
[ 3] real, parameter :: pi = 3.141592654 ! set constant
[ 4] print *, ' Theta      sin^2+cos^2      error'
[ 4] do k = 0, n          ! Loop over (n+1) points
[ 5]   theta = k*pi/n
[ 6]   sint = sin( theta )
[ 7]   cost = cos( theta )
[ 8]   test = sint*sint + cost*cost
[ 9]   write (*, '( 3(1pe14.5) )') theta, test, 1.-test
[10] end do ! over k
```

Theta	sin^2+cos^2	error
0.00000E+00	1.00000E+00	0.00000E+00
1.96350E-01	1.00000E+00	5.96046E-08
3.92699E-01	1.00000E+00	0.00000E+00
5.89049E-01	1.00000E+00	0.00000E+00
7.85398E-01	1.00000E+00	5.96046E-08
9.81748E-01	1.00000E+00	0.00000E+00
1.17810E+00	1.00000E+00	5.96046E-08
1.37445E+00	1.00000E+00	0.00000E+00
1.57080E+00	1.00000E+00	0.00000E+00
1.76715E+00	1.00000E+00	5.96046E-08
1.96350E+00	1.00000E+00	0.00000E+00
2.15985E+00	1.00000E+00	0.00000E+00
2.35619E+00	1.00000E+00	5.96046E-08
2.55254E+00	1.00000E+00	0.00000E+00
2.74889E+00	1.00000E+00	0.00000E+00
2.94524E+00	1.00000E+00	0.00000E+00
3.14159E+00	1.00000E+00	0.00000E+00

### C.2 Problem 1.8.2 : Newton-Raphson algorithm

The most convenient form of loop is the post-test loop, which allows each iteration to be calculated and the error checked at the end.

```
xnew = x
do {
  x = xnew
  xnew = x - f(x)/fprime(x)
}
while (abs(xnew-x) < tolerance)
```

The alternate logic constructs employ tests at the end of the loop and transfer out the end of the loop when necessary. MATLAB and C++ transfer using the “break” command while F90 uses the “exit” command.

A F90 program with an infinite loop, named `testnewton.f90`, and its result is given below. Be warned that this version uses the `IMPLICIT` name styles for integers and reals instead of the better strong typing that results from the recommended use of `IMPLICIT NONE`.

```
[ 1] function f(x) result(y)
[ 2]   real, intent (in) :: x
[ 3]   real               :: y
[ 4]   y = exp(2*x) - 5*x - 1
[ 5] end function f
[ 6] !
[ 7] function fprime(x) result(y)
[ 8]   real, intent (in) :: x
[ 9]   real               :: y
[10]   y = 2*exp(2*x) - 5
[11] end function fprime
[12] !
[13] program main
[14] implicit none
[15] real, parameter :: tolerance = 1.e-6 ! set constant
[16] real :: x, xnew = 3. ! Initial value
[17] integer :: iteration
[18]   iteration = 0
[19] ! Iteration count
[20] do ! forever until true
[21]   iteration = iteration + 1
[22]   x = xnew
[23]   xnew = x - f(x)/fprime(x)
[24]   if ( abs(xnew - x) < tolerance ) exit ! converged is true
[25] end do ! forever
[26] print *, 'Solution: ', xnew, ', Iterations:', iteration
[27] end program main

>>f90 -o newton testnewton.f90
>>newton
Solution:      0.8093941 , Iterations: 10
```

### C.3 Problem 1.8.3 : Game of life

```
[ 1] program game_of_life ! procedural version
[ 2] implicit none
[ 3] integer, parameter :: boardsize = 10
[ 4] integer             :: board (boardsize, boardsize) = 0
[ 5] integer             :: newboard (boardsize, boardsize)
[ 6] character(len=1)    :: ok ! page prompt
[ 7] integer             :: k, number ! loops
[ 8]
[ 9] ! Initial life data, the "Glider"
[10] board (3, 3) = 1; board (4, 4) = 1; board (5, 4) = 1
[11] board (5, 3) = 1; board (5, 2) = 1
[12]
[13] print *, "Initial Life Display:"
[14] call spy (board) ! show initial lifeforms
[15] print *, "Initially alive = ", sum (board); print *, " "
[16]
[17] print *, "Enter number of generations to display:"
[18] read *, number
[19] do k = 1, number
[20]   newboard = next_generation (board)
[21]   board = newboard ! save current lifeforms
[22]   call spy (board) ! show current lifeforms
[23]   print *; print *, "Generation number = ", k
[24]   print *, "Currently alive = ", sum (newboard)
[25]
[26]   print *, 'continue? (y, n)'
[27]   read *, ok ! read any character to continue
[28]   if ( ok == 'n' ) exit ! this do loop only
[29] end do ! on k for number of generations
[30]
[31] contains ! internal (vs external) subprograms
[32]
[33] function next_generation (board) result (newboard)
[34] ! Compute the next generation of life
[35] integer, intent(in) :: board (:,:)
[36] integer             :: newboard (size(board, 1), size(board, 2))
[37] integer             :: i, j, neighbors ! loops
[38]
[39] newboard = 0 ! initialize next generation
[40] do i = 2, boardsize - 1
[41]   do j = 2, boardsize - 1
[42]     neighbors = sum (board (i - 1:i + 1, j - 1:j + 1)) %
[43]       - board (i, j)
[44]     if ( board (i, j) == 1 ) then ! life in the cell
```

```

[ 45]         if ( (neighbors > 3 .or. neighbors < 2) ) then
[ 46]             newboard (i, j) = 0          ! it died
[ 47]         else
[ 48]             newboard (i, j) = 1          ! newborn
[ 49]         end if ! on number of neighbors
[ 50]     else ! no life in the cell
[ 51]         if ( neighbors == 3 ) then
[ 52]             newboard (i, j) = 1          ! newborn
[ 53]         else
[ 54]             newboard (i, j) = 0          ! died
[ 55]         end if ! on number of neighbors
[ 56]     end if ! life status
[ 57] end do ! on column j
[ 58] end do ! on row i
[ 59] end function next_generation
[ 60]
[ 61] Subroutine spy (board)      ! model matlab spy function
[ 62] ! Show an X at each non-zero entry of board, else show -
[ 63] integer, intent(in) :: board (:, :)
[ 64] character (len=1)    :: line (size(board, 1))    ! a line on screen
[ 65] integer              :: i                        ! loops
[ 66]
[ 67]     line = ' '                                ! blank out the line
[ 68]     do i = 1, size (board, 1 )                ! loop over each row
[ 69]         line (1:size (board, 2 )) = '-'        ! current board width
[ 70]         where ( board (i, :) /= 0 ) line = 'X' ! mark non-zero columns
[ 71]         write (*, '(80a1)') line              ! print current row
[ 72]     end do ! over all rows
[ 73] end subroutine spy
[ 74] end program ! game_of_life
[ 75]
[ 76] ! Running gives:
[ 77] ! Initial Life Display:
[ 78] ! -----
[ 79] ! -----
[ 80] ! --X-----
[ 81] ! ---X-----
[ 82] ! -XXX-----
[ 83] ! -----
[ 84] ! -----
[ 85] ! -----
[ 86] ! -----
[ 87] ! -----
[ 88] ! Initially alive = 5
[ 89] !
[ 90] ! Enter number of generations to display: 4
[ 91] ! -----
[ 92] ! -----
[ 93] ! -----
[ 94] ! -X-X-----
[ 95] ! --XX-----
[ 96] ! --X-----
[ 97] ! -----
[ 98] ! -----
[ 99] ! -----
[100] ! -----
[101] !
[102] ! Generation number = 1
[103] ! Currently alive = 5
[104] ! continue? (y, n) n

```

## C.4 Problem 2.5.1 : Conversion factors

This code illustrates the type of global units conversion factors that you can define for your field of study. They can be accessed by any program that includes a use `Conversion_Constants` line and cites a parameter name, as shown on line 16.

```

[ 1] Module Conversion_Constants      ! DefineUnits Conversion
[ 2] ! Define selected precision
[ 3]     INTEGER, PARAMETER :: DP = KIND (1.d0) ! Alternate form
[ 4] ! ===== Metric Conversions =====
[ 5] real(DP), parameter:: cm_Per_Inch      = 2.54_DP
[ 6] real(DP), parameter:: kg_Per_Pound     = 0.45359237_DP
[ 7] real(DP), parameter:: kg_Per_Short_Ton = 907.18474_DP
[ 8] real(DP), parameter:: kg_Per_Long_Ton  = 1016.0469088_DP
[ 9] real(DP), parameter:: m_Per_Foot       = 3.048_DP
[10] real(DP), parameter:: m_Per_Mile       = 1609.344_DP
[11] real(DP), parameter:: m_Per_Naut_Mile  = 1852.0_DP
[12] real(DP), parameter:: m_Per_Yard      = 0.9144_DP
[13] end Module Conversion_Constants
[14] Program Test

```

```

[15] use Conversion_Constants
[16] print *, 'cm_Per_Inch = ', cm_Per_Inch ; End Program Test
[17] ! Running gives: cm_Per_Inch = 2.540000000000000004

```

This code illustrates the type of common physical constants that can be made available as global variables that you can define for your field of study. They can be accessed by any program that includes a use `Physical_Constants` line and cites a parameter name, as shown on line 60 below.

```

[ 1] Module Physical_Constants      ! Define Physical Constants
[ 2] ! Define selected precision
[ 3]   INTEGER, PARAMETER :: DP = KIND (1.d0) ! Alternate form
[ 4]
[ 5] ! ===== Physics Constants and units =====
[ 6] real(DP), parameter:: AMU_Value      = 1.6605402E-27_DP ! kg
[ 7] real(DP), parameter:: Atmosphere_Pres = 9.80665E+04_DP ! Pa
[ 8] real(DP), parameter:: Avogadro        = 6.0221367E+23_DP ! 1/mol
[ 9] real(DP), parameter:: Bohr_Magneton    = 9.2740154E-24_DP ! J/T
[10] real(DP), parameter:: Bohr_Radius      = 5.29177249E-11_DP ! m
[11] real(DP), parameter:: Boltzmann        = 1.380657E-23_DP ! J/K
[12] real(DP), parameter:: c_Light          = 2.997924580E+8_DP ! m/s
[13] real(DP), parameter:: Electron_Compton = 2.42631058E-12_DP ! m
[14] real(DP), parameter:: Electron_Angular = 5.2729E-35_DP ! J*s
[15] real(DP), parameter:: Electron_Charge  = -1.60217738E-19_DP ! coul
[16] real(DP), parameter:: Electron_Mass_Rest = 9.1093897E-31_DP ! kg
[17] real(DP), parameter:: Electron_Moment  = 9.2847700E-24_DP ! J/T
[18] real(DP), parameter:: Electron_Radius  = 2.81794092E-15_DP ! m
[19] real(DP), parameter:: Faraday          = 9.6485309E+04_DP ! C/mo
[20] real(DP), parameter:: G_Universal      = 6.67260E-11_DP ! m^3/(s^2*kg)
[21] real(DP), parameter:: Light_Year       = 9.46073E+15_DP ! m
[22] real(DP), parameter:: Mech_equiv_Heat  = 4.185E+3_DP ! J/kcal
[23] real(DP), parameter:: Molar_Volume      = 0.02241410_DP ! m^3/mol
[24] real(DP), parameter:: Neutron_Mass      = 1.6749286E-27_DP ! kg
[25] real(DP), parameter:: Permeability      = 1.25663706143E-06_DP ! H/m
[26] real(DP), parameter:: Permittivity      = 8.85418781762E-12_DP ! F/m
[27] real(DP), parameter:: Planck_Const      = 6.6260754E-34_DP ! J*s
[28] real(DP), parameter:: Proton_Mass       = 1.6726230E-27_DP ! kg
[29] real(DP), parameter:: Proton_Moment     = 1.41060761E-26_DP ! J/T
[30] real(DP), parameter:: Quantum_charge_r  = 4.13556E+12_DP ! J*s/C
[31] real(DP), parameter:: Rydberg_inf        = 1.0973731534E+07_DP ! 1/m
[32] real(DP), parameter:: Rydberg_Hydrogen  = 1.09678E+07_DP ! 1/m
[33] real(DP), parameter:: Std_Atmosphere     = 1.01325E+05_DP ! Pa
[34] real(DP), parameter:: Stefan_Boltzmann  = 5.67050E-08_DP ! W/(m^2*K^4)
[35] real(DP), parameter:: Thomson_cross_sect = 6.6516E-29_DP ! m^2
[36] real(DP), parameter:: Universal_Gas_C   = 8.314510_DP ! J/mol*K
[37]
[38] ! ===== Astronomy Constants and units =====
[39] real(DP), parameter:: AU_Earth_Sun      = 1.4959787E+11_DP ! m
[40] real(DP), parameter:: Anomal_Month      = 27.5546_DP ! days
[41] real(DP), parameter:: Anomal_Year       = 365.2596_DP ! days
[42] real(DP), parameter:: Dracon_Month      = 27.2122_DP ! days
[43] real(DP), parameter:: Earth_G           = 9.80665_DP ! m/s^2
[44] real(DP), parameter:: Earth_Mass        = 5.974E+24_DP ! kg
[45] real(DP), parameter:: Earth_Radius_Eq   = 6.37814E+6_DP ! m
[46] real(DP), parameter:: Earth_Radius_Mean = 6.371E+6_DP ! m
[47] real(DP), parameter:: Earth_Radius_Polar = 6.356755E+6_DP ! m
[48] real(DP), parameter:: Julian_Year       = 365.25_DP ! days
[49] real(DP), parameter:: Rotation_Day      = 23.93447222_DP ! hours
[50] real(DP), parameter:: Sidereal_Day      = 23.93446944_DP ! hours
[51] real(DP), parameter:: Sidereal_Month     = 27.3217_DP ! days
[52] real(DP), parameter:: Sidereal_Ratio     = 1.0027379092558_DP
[53] real(DP), parameter:: Sidereal_Year     = 365.2564_DP ! days
[54] real(DP), parameter:: Solar_Day         = 24.06571111_DP ! hours
[55] real(DP), parameter:: Synodic_Month      = 29.5306_DP ! days
[56] real(DP), parameter:: Tropical_Year     = 365.2422_DP ! days
[57] end Module Physical_Constants      ! Define Physical Constants
[58] Program Test
[59]   use Physical_Constants
[60]   print *, 'Avogadro = ', Avogadro ; End Program Test
[61] ! Running gives: Avogadro = 0.6022136699999999967E+24

```

## C.5 Problem 3.5.3 : Creating a vector class

We begin by defining the components to be included in our vector object. They include the length of each vector and a corresponding real array of pointers to the vector components:

```

[ 1] module class_Vector      ! filename: class_Vector.f90
[ 2] ! public, everything by default, but can specify any
[ 3]   implicit none
[ 4]   type Vector
[ 5]     private

```

```

[ 6]         integer                :: size    ! vector length
[ 7]         real, pointer, dimension(:) :: data  ! component values
[ 8]     end type Vector

```

For persons familiar with vectors the use of overloaded operators makes sense (but it often does not make sense). Thus we overload the addition, subtraction, multiplication, assignment, and logical equal to operators by defining the correct class members to be used for different argument types:

```

[ 9] ! Overload common operators
[10] interface operator (+) ! add others later
[11] module procedure add_Vector, add_Real_to_Vector; end interface
[12] interface operator (-) ! add unary versions later
[13] module procedure subtract_Vector, subtract_Real; end interface
[14] interface operator (*) ! overload *
[15] module procedure dot_Vector, real_mult_Vector, Vector_mult_real
[16] end interface
[17] interface assignment (=) ! overload =
[18] module procedure equal_Real; end interface
[19] interface operator (==) ! overload ==
[20] module procedure is_equal_to; end interface
[21]

```

Then we encapsulate the supporting member functions, beginning with two constructors, assign and make\_Vector:

```

[22] contains ! functions & operators
[23]
[24] function assign (values) result (name) ! array to vector constructor
[25]     real, intent(in) :: values(:) ! given rank 1 array
[26]     integer :: length ! array size
[27]     type (Vector) :: name ! Vector to create
[28]     length = size(values); allocate ( name%data(length) )
[29]     name % size = length; name % data = values; end function assign
[30]
[31] function make_Vector (len, values) result(v) ! Optional Constructor
[32]     integer, optional, intent(in) :: len ! number of values
[33]     real, optional, intent(in) :: values(:) ! given values
[34]     type (Vector) :: v
[35]     if ( present (len) ) then ! create vector data
[36]         v%size = len ; allocate ( v%data(len) )
[37]         if ( present (values) ) then ; v%data = values ! vector
[38]         else ; v%data = 0.d0 ! null vector
[39]         end if ! values present
[40]     else ! scalar constant
[41]         v%size = 1 ; allocate ( v%data(1) ) ! default
[42]         if ( present (values) ) then ; v%data(1) = values(1) ! scalar
[43]         else ; v%data(1) = 0.d0 ! null
[44]         end if ! value present
[45]     end if ! len present
[46] end function make_Vector
[47]

```

The remainder of the members are given in alphabetical order:

```

[48] function add_Real_to_Vector (v, r) result (new) ! overload +
[49]     type (Vector), intent(in) :: v
[50]     real, intent(in) :: r
[51]     type (Vector) :: new ! new = v + r
[52]     if ( v%size < 1 ) stop "No sizes in add_Real_to_Vector"
[53]     allocate ( new%data(v%size) ) ; new%size = v%size
[54]     ! new%data = v%data + r ! as array operation
[55]     new%data(1:v%size) = v%data(1:v%size) + r ; end function
[56]
[57] function add_Vector (a, b) result (new) ! vector + vector
[58]     type (Vector), intent(in) :: a, b
[59]     type (Vector) :: new ! new = a + b
[60]     if ( a%size /= b%size ) stop "Sizes differ in add_Vector"
[61]     allocate ( new%data(a%size) ) ; new%size = a%size
[62]     new%data = a%data + b%data ; end function add_Vector

```

Note that lines 55 and 62 above are similar ways to avoid writing serial loops that would have to be used in most languages. This keeps the code cleaner and shorter, and more importantly it lets the compiler carry out those operations in parallel on some machines.

While copy members are very important to C++ programmers the following copy\_Vector should probably be omitted since you would not usually pass big arrays as copies and F90 defaults to passing by reference unless forced to pass by value.

```

[63]
[64] function copy_Vector (name) result (new)
[65]     type (Vector), intent(in) :: name

```

```

[ 66]     type (Vector)                :: new
[ 67]     allocate ( new%data(name%size) ) ; new%size = name%size
[ 68]     new%data = name%data          ; end function copy_Vector

```

The routine `delete_Vector` is the destructor for this class. In some sense it is incomplete because it does not delete the `size` attribute. It was decided that while the actual array of data may take a huge amount of storage, the single integer was not important. To be more complete one would have to have to make `size` an integer pointer and allocate and deallocate it at numerous locations within this module.

```

[ 69]
[ 70] subroutine delete_Vector (name) ! deallocate allocated items
[ 71]   type (Vector), intent(inout) :: name
[ 72]   integer                       :: ok ! check deallocate status
[ 73]   deallocate (name%data, stat = ok )
[ 74]   if ( ok /= 0 ) stop "Vector not allocated in delete_Vector"
[ 75]   name%size = 0 ; end subroutine delete_Vector
[ 76]
[ 77] function dot_Vector (a, b) result (c)      ! overload *
[ 78]   type (Vector), intent(in) :: a, b
[ 79]   real                       :: c
[ 80]   if ( a%size /= b%size ) stop "Sizes differ in dot_Vector"
[ 81]   c = dot_product(a%data, b%data); end function dot_Vector
[ 82]
[ 83] subroutine equal_Real (new, R) ! overload =, real to vector
[ 84]   type (Vector), intent(inout) :: new
[ 85]   real, intent(in) :: R
[ 86]   if ( associated (new%data) ) deallocate (new%data)
[ 87]   allocate ( new%data(1) ); new%size = 1
[ 88]   new%data = R ; end subroutine equal_Real
[ 89]
[ 90] logical function is_equal_to (a, b) result (t_f) ! overload ==
[ 91]   type (Vector), intent(in) :: a, b ! left & right of ==
[ 92]   t_f = .false. ! initialize
[ 93]   if ( a%size /= b%size ) return ! same size ?
[ 94]   t_f = all ( a%data == b%data ) ! and all values match
[ 95] end function is_equal_to
[ 96]
[ 97] function length (name) result (n) ! accessor member
[ 98]   type (Vector), intent(in) :: name
[ 99]   integer :: n
[100]   n = name % size ; end function length
[101]
[102] subroutine list (name) ! accessor member, for prettier printing
[103]   type (Vector), intent(in) :: name
[104]   print *, "[", name % data(1:name%size), "]" ; end subroutine list
[105]
[106] function normalize_Vector (name) result (new)
[107]   type (Vector), intent(in) :: name
[108]   type (Vector) :: new
[109]   real :: total, nil = epsilon(1.0) ! tolerance
[110]   allocate ( new%data(name%size) ); new%size = name%size
[111]   total = sqrt ( sum ( name%data**2 ) ) ! intrinsic functions
[112]   if ( total < nil ) then ; new%data = 0.d0 ! avoid division by 0
[113]     else ; new%data = name%data/total
[114]   end if ; end function normalize_Vector
[115]
[116] subroutine read_Vector (name) ! read array, assign
[117]   type (Vector), intent(inout) :: name
[118]   integer, parameter :: max = 999
[119]   integer :: length
[120]   read (*, '(i1)', advance = 'no') length
[121]   if ( length <= 0 ) stop "Invalid length in read_Vector"
[122]   if ( length >= max ) stop "Maximum length in read_Vector"
[123]   allocate ( name % data(length) ) ; name % size = length
[124]   read *, name % data(1:length) ; end subroutine read_Vector
[125]
[126] function real_mult_Vector (r, v) result (new) ! overload *
[127]   real, intent(in) :: r
[128]   type (Vector), intent(in) :: v
[129]   type (Vector) :: new ! new = r * v
[130]   if ( v%size < 1 ) stop "Zero size in real_mult_Vector"
[131]   allocate ( new%data(v%size) ) ; new%size = v%size
[132]   new%data = r * v%data ; end function real_mult_Vector
[133]
[134] function size_Vector (name) result (n) ! accessor member
[135]   type (Vector), intent(in) :: name
[136]   integer :: n
[137]   n = name % size ; end function size_Vector
[138]
[139] function subtract_Real(v, r) result(new) ! vector-real, overload -
[140]   type (Vector), intent(in) :: v
[141]   real, intent(in) :: r
[142]   type (Vector) :: new ! new = v + r

```



```

[143]     if ( v%size < 1 ) stop "Zero length in subtract_Real"
[144]     allocate ( new%data(v%size) ) ; new%size = v%size
[145]     new%data = v%data - r           ; end function subtract_Real
[146]
[147] function subtract_Vector (a, b) result (new) ! overload -
[148] type (Vector), intent(in) :: a, b
[149] type (Vector)              :: new
[150] if ( a%size /= b%size ) stop "Sizes differ in subtract_Vector"
[151] allocate ( new%data(a%size) ) ; new%size = a%size
[152] new%data = a%data - b%data       ; end function subtract_Vector
[153]
[154] function values (name) result (array)      ! accessor member
[155] type (Vector), intent(in) :: name
[156] real                      :: array(name%size)
[157] array = name % data ; end function values

```

The routine `delete_Vector` is the manual constructor for this class. It has no optional arguments so both arguments must be supplied, and it duplicates the constructor on line 31, but it uses the naming convention preferred by the author.

```

[158]
[159] function Vector_ (length, values) result(name) ! constructor
[160] integer, intent(in) :: length           ! array size
[161] real, target, intent(in) :: values(length) ! given array
[162] real, pointer          :: pt_to_val(:) ! pointer to array
[163] type (Vector)          :: name          ! Vector to create
[164] integer                :: get_m        ! allocate flag
[165] allocate ( pt_to_val (length), stat = get_m ) ! allocate
[166] if ( get_m /= 0 ) stop 'allocate error'      ! check
[167] pt_to_val = values                          ! dereference values
[168] name = Vector(length, pt_to_val) ! intrinsic constructor
[169] end function Vector_
[170]
[171] function Vector_max_value (a) result (v)      ! accessor member
[172] type (Vector), intent(in) :: a
[173] real                      :: v
[174] v = maxval ( a%data(1:a%size) ) ; end function Vector_max_value
[175]
[176] function Vector_min_value (a) result (v)      ! accessor member
[177] type (Vector), intent(in) :: a
[178] real                      :: v
[179] v = minval ( a%data(1:a%size) ) ; end function Vector_min_value
[180]
[181] function Vector_mult_real(v, r) result(new) ! vec*real, overload *
[182] type (Vector), intent(in) :: v
[183] real, intent(in) :: r
[184] type (Vector) :: new ! new = v * r
[185] if ( v%size < 1 ) stop "Zero size in Vector_mult_real"
[186] new = Real_mult_Vector(r, v); end function Vector_mult_real
[187]
[188] end module class_Vector

```

A first test of this class is given below along with comments that give the verifications of the members.

```

[ 1] !           Testing Vector Class Constructors & Operators
[ 2] include 'class_Vector.f90'           ! see previous figure
[ 3] program check_vector_class
[ 4]   use class_Vector
[ 5]   implicit none
[ 6]
[ 7]   type (Vector) :: x, y, z
[ 8]
[ 9] !           test optional constructors: assign, and copy
[10] x = make_Vector ( ) ! single scalar zero
[11] write (*, '("made scalar x = ")', advance='no'); call list(x)
[12]
[13] call delete_Vector (x) ; y = make_Vector (4) ! 4 zeros
[14] write (*, '("made null y = ")', advance='no'); call list(y)
[15]
[16] z = make_Vector (4, (/11., 12., 13., 14./) ) ! 4 non-zeros
[17] write (*, '("made full z = ")', advance='no'); call list(z)
[18] write (*, '("assign [ 31., 32., 33., 34. ] to x")')
[19]
[20] x = assign( (/31., 32., 33., 34./) ) ! (4) non-zeros
[21] write (*, '("assigned x = ")', advance='no'); call list(x)
[22]
[23] x = Vector_(4, (/31., 32., 33., 34./) ) ! 4 non-zeros
[24] write (*, '("public x = ")', advance='no'); call list(x)
[25] write (*, '("copy x to y = ")', advance='no')
[26] y = copy_Vector (x) ; call list(y) ! copy
[27]
[28] !           test overloaded operators
[29] write (*, '("z * x gives ")', advance='no'); print *, z*x ! dot
[30] write (*, '("z + x gives ")', advance='no'); call list(z+x) ! add

```

```

[31] y = 25.6                                ! real to vector
[32] write (*, '("y = 25.6 gives ")', advance='no'); call list(y)
[33] y = z                                    ! equality
[34] write (*, '("y = z gives y as ")', advance='no'); call list(y)
[35] write (*, '("logic y == x gives ")', advance='no'); print *, y==x
[36] write (*, '("logic y == z gives ")', advance='no'); print *, y==z
[37]
[38] !                                     test destructor, accessors
[39] call delete_Vector (y)                  ! destructor
[40] write (*, '("deleting y gives y = ")', advance='no'); call list(y)
[41] print *, "size of x is ", length (x)    ! accessor
[42] print *, "data in x are [", values (x), "]" ! accessor
[43] write (*, '("2. times x is ")', advance='no'); call list(2.0*x)
[44] write (*, '("x times 2. is ")', advance='no'); call list(x*2.0)
[45] call delete_Vector (x); call delete_Vector (z) ! clean up
[46] end program check_vector_class
[47] ! Running gives the output: ! made scalar x = [0]
[48] ! made null y = [0, 0, 0, 0] ! made full z = [11, 12, 13, 14]
[49] ! assign [31, 32, 33, 34] to x ! assigned x = [31, 32, 33, 34]
[50] ! public x = [31, 32, 33, 34] ! copy x to y = [31, 32, 33, 34]
[51] ! z * x gives 1630           ! z + x gives [42, 44, 46, 48]
[52] ! y = 256 gives [256000004] ! y = z, y = [11, 12, 13, 14]
[53] ! logic y == x gives F       ! logic y == z gives T
[54] ! deleting y gives y = []    ! size of x is 4
[55] ! data in x : [31, 32, 33, 34] ! 2 times x is [62, 64, 66, 68]
[56] ! x times 2 is [62, 64, 66, 68]

```

Having tested the vector class we will now use it in some typical vector operations. We want a program that will work with arrays of vectors to read in the number of vectors. The array of vectors will use an automatic storage mode. That could be risky because if the system runs out of memory we get a fatal error message and the run aborts. If we made the alternate choice of allocatable arrays then we could check the allocation status and have a chance (but not a good chance) of closing down the code in some "friendly" manner. Once the code reads the number of vectors then for each one it reads the number of components and the the component values. After testing some simple vector math we compute a more complicated result known as the orthonormal basis for the given set of vectors:

```

[ 1] !      Test Vector Class Constructors, Operators and Basis
[ 2] include 'class_Vector.f'
[ 3]
[ 4] program check_basis ! demonstrate a typical Vector class
[ 5]   use class_Vector
[ 6]   implicit none
[ 7]
[ 8]   interface
[ 9]     subroutine testing_basis (N_V)
[10]       integer, intent(in) :: N_V
[11]     end subroutine testing_basis
[12]   end interface
[13]
[14]   print *, "Test automatic allocate, deallocate"
[15]   print *, " " ; read *, N_V
[16]   print *, "The number of vectors to be read is: ", N_V
[17]   call testing_basis ( N_V) ! to use automatic arrays
[18] end program check_basis
[19]
[20] subroutine testing_basis (N_V)
[21] ! test vectors AND demo automatic allocation/deallocation
[22] use class_Vector
[23]
[24] integer, intent(in) :: N_V
[25] type (Vector)      :: Input(N_V) ! automatic array
[26] type (Vector)      :: Ortho(N_V) ! automatic array
[27] integer            :: j
[28] real               :: norm
[29]
[30] interface
[31]   subroutine orthonormal_basis (Input, Ortho, N_given)
[32]     use class_Vector
[33]     type (Vector), intent(in) :: Input(N_given)
[34]     type (Vector), intent(out) :: Ortho(N_given)
[35]     integer, intent(in)       :: N_given
[36]   end subroutine orthonormal_basis
[37] end interface
[38]
[39] print *, " " ; print *, "The given ", N_V, " vectors:"
[40] do j = 1, N_V
[41]   call read_Vector ( Input(j) )
[42]   call list        ( Input(j) )
[43] end do ! for j
[44]
[45] print *, " "

```

```

[ 46] print *, "The Orthogonal Basis of the original set is:"
[ 47]
[ 48] call orthonormal_basis (Input, Ortho, N_V)
[ 49] do j = 1, N_V ! list new orthogonal basis
[ 50]   call list ( Ortho(j) )
[ 51] end do ! for j
[ 52]
[ 53] ! use vector class features & operators
[ 54] print *, ' ' ; print *, "vector 1 + vector 2 = "
[ 55] call list (Input(1)+Input(2))
[ 56] print *, "vector 1 - vector 2 = "
[ 57] call list (Input(1)-Input(2))
[ 58] print *, "vector 1 dot vector 2 = ", Input(1)*Input(2)
[ 59] print *, "vector 1 * 3.5 = "
[ 60] call list (3.5*Input(1))
[ 61] norm = sqrt ( dot_Vector( Input(1), Input(1) ))
[ 62] print *, "norm(vector 1) = ", norm
[ 63] print *, "normalized vector 1 = "
[ 64] call list (normalize_Vector(Input(1)))
[ 65] print *, "max(vector 1) = ", vector_max_value (Input(1))
[ 66] print *, "min(vector 1) = ", vector_min_value (Input(1))
[ 67] print *, "length of vector 1 = ", length ( Input(1) )
[ 68] end subroutine testing_basis
[ 69]
[ 70] subroutine orthonormal_basis (Input, Ortho, N_given)
[ 71] ! Find Orthonormal Basis of a Set of Vector Classes
[ 72] use class_Vector
[ 73] ! *****
[ 74] ! =, -, +, * are overloaded operators from class_Vector
[ 75] ! *****
[ 76]
[ 77] type (Vector), intent(in) :: Input(N_given)
[ 78] type (Vector), intent(out) :: Ortho(N_given)
[ 79] integer, intent(in) :: N_given
[ 80] integer :: i, j ! loops
[ 81] real :: dot
[ 82] do i = 1, N_given ! original set of vectors
[ 83]   Ortho(i) = Input(i) ! copy input vector class
[ 84]   do j = 1, i ! for previous copies
[ 85]     dot = dot_Vector(Ortho(i), Ortho(j))
[ 86]     Ortho(i) = Ortho(i) - (dot*Ortho(j))
[ 87]   end do ! for j
[ 88]   Ortho(i) = normalize_Vector ( Ortho(i) )
[ 89] end do ! over i
[ 90] end subroutine orthonormal_basis
[ 91]
[ 92] ! Compiling and inputting :
[ 93] ! 4
[ 94] ! 3 0.625 0 0
[ 95] ! 3 7.5 3.125 0
[ 96] ! 3 13.25 -7.8125 6.5
[ 97] ! 3 14.0 3.5 -7.5
[ 98] ! Gives:
[ 99] ! Test automatic allocate, deallocate
[100] !
[101] ! The number of vectors to be read is: 4
[102] ! The given 4 vectors:
[103] ! [ 0.6250 0.0000 0.0000 ]
[104] ! [ 7.5000 3.1250 0.0000 ]
[105] ! [ 13.2500 -7.8125 6.5000 ]
[106] ! [ 14.0000 3.5000 -7.5000 ]
[107] !
[108] ! The Orthogonal Basis of the original set is:
[109] ! [ 1.0000 0.0000 0.0000 ]
[110] ! [ 0.0000 -1.0000 0.0000 ]
[111] ! [ 0.0000 0.0000 -1.0000 ]
[112] ! [ 0.0000 0.0000 0.0000 ]
[113] !
[114] ! vector 1 + vector 2 = [ 8.1250 3.1250 0.0000 ]
[115] ! vector 1 - vector 2 = [-6.8750 -3.1250 0.0000 ]
[116] ! vector 1 dot vector 2 = 4.6875
[117] ! vector 1 * 3.5 = [ 2.1875 0.0000 0.0000 ]
[118] ! norm(vector 1) = 0.6250
[119] ! normalized vector 1 = [ 1.0000 0.0000 0.0000 ]
[120] ! max(vector 1) = 0.6250
[121] ! min(vector 1) = 0.0000
[122] ! length of vector 1 = 3

```

## C.6 Problem 3.5.4 : Creating a sparse vector class

This class begins like the previous Vector class except that we must add a row entry (line 4) for each data value entry (line 5). This is done for efficiency since we expect most values in sparse vectors to be zero (and hence their name). The attribute `non_zero` is the size of both rows and values.

```
[ 1] module class_sparse_Vector
[ 2]   implicit none
[ 3]   type sv          ! a sparse vector
[ 4]   integer          :: non_zeros
[ 5]   integer, pointer :: rows(:)
[ 6]   real,    pointer :: values(:)
[ 7] end type
[ 8]
```

The overloading process is similar, but now we will see that much more logic is required to deal with the zero entries and new zeros created by addition or multiplication.

```
[ 8]   interface assignment (=)
[ 9]     module procedure equal_Vector ; end interface
[10]   interface operator (.dot.) ! define dot product operator
[11]     module procedure dot_Vector ; end interface
[12]   interface operator (==) ! Boolean equal to
[13]     module procedure is_equal_to ; end interface
[14]   interface operator (*) ! term by term product
[15]     module procedure el_by_el_Mult, real_mult_Sparse
[16]     module procedure Sparse_mult_real
[17]   end interface
[18]   interface operator (-) ! for sparse vectors
[19]     module procedure Sub_Sparse_Vectors ; end interface
[20]   interface operator (+) ! for sparse vectors
[21]     module procedure Sum_Sparse_Vectors ; end interface
[22]
[23] contains ! operators and functionality
```

In the following constructor for the class note that both the pointer array attributes are allocated (line 32) the same amount of storage in memory. One should also include the allocation status flag here and checks its value to raise a possible exception (as seen in lines 41-46).

```
[24]   subroutine make_Sparse_Vector (s,n,r,v)
[25]     ! allows zero length vectors
[26]     type (sv) :: s ! name
[27]     integer, intent(in) :: n ! size
[28]     integer, intent(in) :: r(n) ! rows
[29]     real, intent(in) :: v(n) ! values
[30]     if ( n < 0 ) stop &
[31]       "Error, negative rows in make_Sparse_Vector"
[32]     allocate (s%rows(n), s%values(n))
[33]     s%non_zeros = n ! copy size
[34]     s%rows = r ! row array assignment
[35]     s%values = v ! value array assignment
[36]   end subroutine make_Sparse_Vector
[37]
```

This is really a destructor. Again, it is incomplete because the integer array size was not made allocatable for simplicity.

```
[38]   subroutine delete_Sparse_Vector (s)
[39]     type (sv) :: s ! name of sparse vector
[40]     integer :: error ! deallocate status flag, 0 no error
[41]     deallocate (s%rows, s%values, stat = error) ! memory released
[42]     if ( error == 0 ) then
[43]       s%non_zeros = 0 ! reset size
[44]     else ! never created
[45]       stop "Sparse vector to delete does not exist"
[46]     end if ; end subroutine delete_Sparse_Vector
[47]
```

This creates a user defined operator call `.dot.` to be applied to sparse vectors.

```
[48]   function dot_Vector (u, v) result (d) ! defines .dot.
[49]     ! dot product of sparse vectors
[50]     type (sv), intent(in) :: u, v ! sparse vectors
[51]     type (sv) :: w ! sparse vector, temporary
[52]     real :: d ! dot product value
[53]     d = 0.0 ! default
[54]     if ( u%non_zeros < 1 .or. v%non_zeros < 1 ) return ! null
[55]     w = el_by_el_Mult (u, v) ! element by element sparse product
[56]     if ( w%non_zeros > 0 ) &
[57]       d = sum( w%values(1:w%non_zeros) ) ! summed
[58]     call delete_Sparse_Vector (w) ! delete temp
```

```
[ 59]     end function dot_Vector
[ 60]
```

The above `dot_Vector` is more complicated in this format because it is likely that stored non-zero values will be multiplied by (unstored) zeros. Thus, the real work is done in the following member function that employs Boolean logic. The terms for the summation that creates the scalar dot product are first computed in a full vector equal in length to the minimum row number given. Observe that its size is established through the use of the `min` intrinsic, acting on the two given sizes, within the `dimension` attribute for the full array (lines 67,68). Three logical arrays (line 68) are used as “masks” which are true when a non-zero exists in the corresponding row of their associated sparse vector (down to the minimum row cited above). The three logical vectors are initialized in lines 77 to 92. That process ends with the third vector being created as a Boolean product (line 91) and the maximum possible number of non-zero products is found from the `count` intrinsic (line 92).

It is also important to note that the working space vector `full` is an automatic array and memory for it is automatically allocated for it each time the function is called. It could be an extremely long vector and thus it is possible (but not likely) that there would not be enough memory available. Then the system would abort with an error message. To avoid that possibility one could have declared `full` to be an allocatable vector and then allocate its memory by using a similar `min` construct. That allocation request should (always) include the `STAT` flag so that if the memory allocation fails it would be possible to issue an exception to try to avoid a fatal crash of the system (not likely).

```
[ 61]     function el_by_el_Mult (u, v) result (w) ! defines * operator
[ 62]     ! element by element product of sparse vectors: 0 * real ?
[ 63]     type (sv), intent(in) :: u, v           ! given vectors
[ 64]     type (sv)              :: w             ! new vector
[ 65]     real :: full( min( u%rows(u%non_zeros), & ! automatic
[ 66]     &                v%rows(v%non_zeros) ) ) ! workspace
[ 67]     logical, dimension( min( u%rows(u%non_zeros), &
[ 68]     v%rows(v%non_zeros))) :: u_m, v_m, w_m ! logical product masks
[ 69]     integer :: j, k, last, n, row
[ 70]     ! is either u or v null ?
[ 71]     if ( u%non_zeros < 1 .or. v%non_zeros < 1 ) then ! w is null
[ 72]         allocate ( w%rows(0), w%values(0) )
[ 73]         w%non_zeros = 0
[ 74]         return ! a null sparse vector
[ 75]     end if ! no calculation necessary
[ 76]
[ 77]     ! Initialize logic masks
[ 78]     last = min( u%rows(u%non_zeros), v%rows(v%non_zeros) ) ! max size
[ 79]     u_m = .false. ! assume no contributions
[ 80]     do j = 1, size(u%rows)
[ 81]         row = u%rows(j) ! get row number to flag
[ 82]         if ( row > last ) exit ! j loop
[ 83]         u_m(row) = .true. ! possible contribution
[ 84]     end do ! to initialize u mask
[ 85]     v_m = .false. ! assume no contributions
[ 86]     do j = 1, size(v%rows)
[ 87]         row = v%rows(j) ! get row number to flag
[ 88]         if ( row > last ) exit ! j loop
[ 89]         v_m(row) = .true. ! possible contribution
[ 90]     end do ! to initialize v mask
[ 91]     w_m = (u_m .and. v_m) ! Boolean product logic
[ 92]     n = count ( w_m ) ! count possible products
[ 93]     ! if ( n == 0 ) print *, "Warning: zero length sparse" ! debug
[ 94]
```

The vector `full` is set to zero (line 96) and comparison DO loops (lines 97,101) over the two given vectors are minimized (lines 100,103) by testing where the mask vector `w_m` is true (thereby indicating a non-zero product). When all the products are stored in the `full` vector it is converted to the sparse vector storage mode (line 109) for release as the return result. Because `full` is an automatic array its memory is automatically released when the function is exited.

```
[ 95]     ! Fill the product workspace, full
[ 96]     full = 0.0 ! initialize
[ 97]     do j = 1, size(u%rows) ! loop over u
[ 98]         row = u%rows(j) ! row in u
[ 99]         if ( row > last ) exit ! this loop in u ! past end of w
[100]         if ( .not. w_m(row) ) cycle ! to next j ! not in product
[101]         do k = 1, size(v%rows) ! loop over v
[102]             if ( v%rows(k) > last ) exit ! this loop ! past end of w
[103]             if ( .not. w_m(v%rows(k)) ) cycle ! to k+1 ! not in product
[104]             if ( row == v%rows(k) ) then ! same row, u & v
[105]                 full(row) = u%values(j)*v%values(k) ! get product
[106]
```

```

[106]         end if
[107]     end do ! on k in v
[108] end do ! on j in u
[109] w = Vector_To_Sparse (full) !delete any zeros
[110] end function el_by_el_Mult ! deletes full & 3 masks
[111]

```

The operator overloading members are given with the next function (line 112) as well as in lines 140, 231, and 320.

```

[112] subroutine equal_Vector (new, s) ! overload =
[113]     type (sv), intent(inout) :: new
[114]     type (sv), intent(in)    :: s
[115]     allocate ( new%rows(s%non_zeros) )
[116]     allocate ( new%values(s%non_zeros) )
[117]     new%non_zeros = s%non_zeros
[118]     if ( s%non_zeros > 0 ) then
[119]         new%rows(1:s%non_zeros) = s%rows(1:s%non_zeros) ! array copy
[120]         new%values(1:s%non_zeros) = s%values(1:s%non_zeros) ! copy
[121]     end if ; end subroutine equal_Vector
[122]
[123] function get_element (name, row) result (v)
[124]     type (sv), intent(in) :: name ! sparse vector
[125]     integer, intent(in) :: row ! row in sparse vector
[126]     integer :: j ! loops
[127]     real :: v ! value at row
[128]     v = 0.0 ! default
[129]     if ( row < 1 ) stop "Invalid row number, get_element"
[130]     if ( name%non_zeros < 1 ) return ! not here
[131]     if ( row > name%rows(name%non_zeros) ) return ! not here
[132]     do j = 1, name%non_zeros
[133]         if ( row == name%rows(j) ) then
[134]             v = name%values(j) ! found the value
[135]             return ! search done
[136]         end if ! in the vector
[137]     end do ! over possible values
[138] end function get_element
[139]
[140] function is_equal_to (a, b) result (t_or_f) ! define ==
[141]     type (sv), intent(in) :: a, b ! two sparse vectors
[142]     logical :: t_or_f
[143]     integer :: i ! loops
[144]     t_or_f = .true. ! default
[145]     if ( a%non_zeros == b%non_zeros ) then ! also check values
[146]         do i = 1, a%non_zeros ! or use count function for simplicity
[147]             if ( a%rows(i) /= b%rows(i) .or. &
[148]                 a%values(i) /= b%values(i) ) then
[149]                 t_or_f = .false. ! because rows and/or values differ
[150]                 return ! no additional checks needed
[151]             end if ! same values
[152]         end do ! over sparse rows
[153]     else ! sizes differ so vectors must be different
[154]         t_or_f = .false.
[155]     end if ! sizes match
[156] end function is_equal_to
[157]
[158] function largest_index (s) result(row)
[159]     type (sv), intent(in) :: s ! sparse vector
[160]     integer :: row ! last non-zero in full vector
[161]     integer :: j ! loops
[162]     row = 0 ! initialize
[163]     if ( s%non_zeros < 1 ) return ! null vector
[164]     do j = s%non_zeros, 1, -1 ! loop backward
[165]         if ( s%values(j) /= 0.0 ) then ! last non-zero term
[166]             row = s%rows(j) ! actual row number
[167]             return ! search done
[168]         end if
[169]     end do
[170] end function largest_index
[171]
[172] function length (name) result (n)
[173]     type (sv), intent(in) :: name
[174]     integer :: n
[175]     n = name % non_zeros ! read access to size, if private
[176] end function length
[177]

```

Once again we observe that the next two functions employ the colon operator (lines 185,196,199,201) to avoid explicit serial loops which would make them faster on certain vector and parallel computers.

```

[178] function norm (name) result (total)
[179]     type (sv), intent(in) :: name
[180]     real :: total
[181]     if ( name%non_zeros < 1 ) then

```

```

[182]         ! print *, "Warning: empty vector in norm"
[183]         total = 0.0
[184]     else
[185]         total = sqrt( sum( name%values(1:name%non_zeros)**2 ))
[186]     end if ! a null vector
[187] end function norm
[188]
[189] function normalize_Vector (s) result (new)
[190] type (sv), intent(in) :: s
[191] type (sv)              :: new
[192] real                   :: total, epsilon = 1.e-6
[193] allocate ( new%rows (s%non_zeros) )
[194] allocate ( new%values(s%non_zeros) )
[195] new%non_zeros = s%non_zeros ! copy size
[196] new%rows(1:s%non_zeros) = s%rows(1:s%non_zeros) ! copy rows
[197] total = sqrt( sum( s%values(1:s%non_zeros)**2 )) ! norm
[198] if ( total <= epsilon ) then ! divide by 0 ?
[199]     new%values(1:s%non_zeros) = 0.d0 ! set to zero
[200] else ! or real values
[201]     new%values(1:s%non_zeros) = s%values(1:s%non_zeros)/total
[202] end if ! division by zero
[203] end function normalize_Vector
[204]
[205] subroutine pretty (s) ! print all values if space allows
[206] type (sv), intent(in) :: s ! sparse vector
[207] integer, parameter    :: limit = 20 ! for print size
[208] integer               :: n
[209] real                  :: full( s%rows(s%non_zeros) ) ! temp
[210] n = s%non_zeros
[211] if ( s%non_zeros < 1 .or. s%rows(s%non_zeros) > limit ) then
[212]     print *, "Wrong size to pretty print"
[213] else
[214]     full = 0. ! initialize to zero
[215]     if ( n > 0 ) full(s%rows) = s%values ! array copy non zeros
[216]     print *, "[", full, "]" ! pretty print
[217] end if ; end subroutine pretty ! automatic deallocate of full
[218]
[219] subroutine read_Vector (name) ! sparse vector data on unit 1
[220] type (sv), intent(inout) :: name
[221] integer                   :: length, j
[222] read (1,'(i1)', advance = 'no') length
[223] if ( length <= 0 ) stop "Invalid length in read_Vector"
[224] name % non_zeros = length
[225] allocate ( name % rows (length) )
[226] allocate ( name % values(length) )
[227] read (1,*) ( name%rows(j), name%values(j), j = 1, length)
[228] name%rows = name%rows + 1 ! default to 1 not 0 in F90
[229] end subroutine read_Vector
[230]
[231] function real_mult_Sparse (a, b) result (new)
[232] ! scalar * vector
[233] real, intent(in) :: a
[234] type (sv), intent(in) :: b
[235] type (sv)              :: new
[236] allocate ( new%rows (b%non_zeros) )
[237] allocate ( new%values(b%non_zeros) )
[238] new%non_zeros = b%non_zeros
[239] if ( b%non_zeros < 1 ) then
[240]     print *, "Warning: zero size in real_mult_Sparse "
[241] else ! copy array components
[242]     new%rows (1:b%non_zeros) = b%rows (1:b%non_zeros)
[243]     new%values(1:b%non_zeros) = a * b%values(1:b%non_zeros)
[244] end if ! null vector
[245] end function real_mult_Sparse
[246]
[247] function rows_of (s) result(n) ! copy rows array of s
[248] type (sv) :: s ! sparse vector
[249] integer :: n(s%non_zeros) ! standard array
[250] if ( s%non_zeros < 1 ) stop "No rows to extract, rows_of"
[251] n = s%rows ! array copy
[252] end function rows_of
[253]
[254] subroutine set_element (s, row, value)
[255] ! Set, or insert, value into row of a sparse vector, s
[256] type (sv), intent(inout) :: s ! sparse vector
[257] integer, intent(in) :: row ! full vector row
[258] real, intent(in) :: value ! full vector value
[259] type (sv) :: new ! workspace
[260] logical :: found ! true if row exists
[261] integer :: j, where ! loops, locator
[262] found = .false. ! initialize
[263] where = 0 ! initialize
[264] do j = 1, s%non_zeros
[265]     if ( s%rows(j) == row ) then ! found it
[266]         s%values(j) = value ! value changed

```

```

[267]         return                                ! no insert needed
[268]     end if
[269]     if ( s%rows(j) > row ) then
[270]         where = j                                ! insert before j
[271]         exit ! the loop search
[272]     else ! s%rows(j) < row,                        may be next or last
[273]         where = j + 1
[274]     end if
[275] end do ! over current rows in s
[276] if ( .not. found ) then ! expand and insert at where
[277]     if ( where == 0 ) stop "Logic error, set_element"
[278]     new%non_zeros = s%non_zeros + 1
[279]     allocate ( new%rows (new%non_zeros) )
[280]     allocate ( new%values(new%non_zeros) )
[281]     ! copy preceeding rows
[282]     if ( where > 1 ) then ! copy to front of new
[283]         new%rows (1:where-1) = s%rows (1:where-1) ! array copy
[284]         new%values(1:where-1) = s%values(1:where-1) ! array copy
[285]     end if ! copy to front of new
[286]     ! insert, copy following rows of s
[287]     new%rows (where ) = row                        ! insert
[288]     new%values(where ) = value                     ! insert
[289]     new%rows (where+1:) = s%rows (where:)          ! array copy
[290]     new%values(where+1:) = s%values(where:)        ! array copy
[291]     ! deallocate s, move new to s, deallocate new
[292]     call delete_Sparse_Vector (s)                 ! delete s
[293]     call equal_Vector (s, new)                     ! s <- new
[294]     call delete_Sparse_Vector (new)                ! delete new
[295] end if ! an insert is required
[296] end subroutine set_element
[297]
[298] subroutine show (s) ! alternating row number and value
[299]     type (sv) :: s ! sparse vector
[300]     integer :: j, k ! implied loops
[301]     k = length (s)
[302]     if ( k == 0 ) then
[303]         print *, k ; else ;
[304]         print *, k, ( (s%rows(j)-1), s%values(j), j = 1, k )
[305]     end if ; end subroutine show
[306]
[307] subroutine show_r_v (s) ! all rows then all values
[308]     type (sv) :: s ! sparse vector
[309]     print *, "Vector has ", s%non_zeros, " non_zero terms."
[310]     if ( s%non_zeros > 0 ) then
[311]         print *, "Rows: ", s%rows - 1 ! to look like C++
[312]         print *, "Values: ", s%values
[313]     end if ; end subroutine show_r_v
[314]
[315] function size_of (s) result(n)
[316]     type (sv) :: s
[317]     integer :: n
[318]     n = s%non_zeros ; end function size_of
[319]
[320] function Sparse_mult_real (a, b) result (new)
[321]     ! vector * scalar
[322]     real, intent(in) :: b
[323]     type (sv), intent(in) :: a
[324]     type (sv) :: new
[325]     new = real_mult_Sparse ( b, a ) ! reverse the order
[326] end function Sparse_mult_real
[327]

```

In the following subtraction and addition functions we again note that sparse terms with the same values but opposite signs can result in new zero terms in the resulting vector. A temporary automatic workspace vector, `full`, is used to hold the preliminary results. In this case it must have a size that is the maximum of the two given vectors. Thus, the `max intrinsic` is employed in its dimension attribute (lines 331,344) which is opposite the earlier multiplication example (line 65).

```

[328] function Sub_Sparse_Vectors (u, v) result (w) ! defines -
[329]     type (sv), intent(in) :: u, v
[330]     type (sv) :: w
[331]     real :: full( max( u%rows(u%non_zeros), & ! automatic
[332]         & v%rows(v%non_zeros) ) ) ! workspace
[333]     if ( u%non_zeros <= 0 ) stop "First vector doesn't exist"
[334]     if ( v%non_zeros <= 0 ) stop "Second vector doesn't exist"
[335]     full = 0.0 ! set to zero
[336]     full(u%rows) = u%values ! copy first values
[337]     full(v%rows) = full(v%rows) - v%values ! less second values
[338]     w = Vector_To_Sparse (full) ! delete any zeros
[339] end function Sub_Sparse_Vectors ! automatically deletes full
[340]
[341] function Sum_Sparse_Vectors (u, v) result (w) ! defines +
[342]     type (sv), intent(in) :: u, v

```



```

[343]         type (sv)                                :: w
[344]         real :: full( max( u%rows(u%non_zeros), & ! automatic
[345]             & v%rows(v%non_zeros) ) ) ! workspace
[346]         if ( u%non_zeros <= 0 ) stop "First vector doesn't exist"
[347]         if ( v%non_zeros <= 0 ) stop "Second vector doesn't exist"
[348]         full = 0.                                ! set to zero
[349]         full(u%rows) = u%values                    ! copy first values
[350]         full(v%rows) = full(v%rows) + v%values    ! add second values
[351]         w = Vector_To_Sparse (full)                ! delete any zeros
[352]     end function Sum_Sparse_Vectors ! automatically deletes full
[353]
[354]     function values_of (s) result(v) ! copy values of s
[355]         type (sv) :: s                ! sparse vector
[356]         real      :: v(s%non_zeros)   ! standard array
[357]         if ( s%non_zeros < 1 ) &
[358]             stop "No values to extract, in values_of"
[359]         v = s%values                    ! array copy
[360]     end function values_of
[361]
[362]     function Vector_max_value (a) result (v)
[363]         type (sv), intent(in) :: a
[364]         real                  :: v
[365]         v = maxval (a%values(1:a%non_zeros)) ! intrinsic function
[366]         ! is it a sparse vector with a false negative maximum ?
[367]         if ( a%non_zeros < a%rows(a%non_zeros) .and. v < 0. ) v = 0.0
[368]     end function Vector_max_value
[369]
[370]     function Vector_min_value (a) result (v)
[371]         type (sv), intent(in) :: a
[372]         real                  :: v
[373]         v = minval ( a%values(1:a%non_zeros) ) ! intrinsic function
[374]         ! is it a sparse vector with a false positive minimum ?
[375]         if ( a%non_zeros < a%rows(a%non_zeros) &
[376]             .and. v > 0. ) v = 0.0
[377]     end function Vector_min_value
[378]

```

This function is invoked several times in other member functions. It simply accepts a standard (dense) vector and converts it to the sparse storage mode in the return result.

```

[379]     function Vector_To_Sparse (full) result (sparse)
[380]         real, intent(in) :: full(:) ! standard array
[381]         type (sv)       :: sparse   ! sparse vector copy
[382]         integer          :: j, n, number ! loops and counters
[383]         n = count ( full /= 0.0 ) ! count non_zeros
[384]         ! if ( n == 0 ) print *, "Warning: null full vector "
[385]         allocate ( sparse%rows(n), sparse%values(n) )
[386]         sparse%non_zeros = n ! sparse size
[387]         number = 0 ! non zeros inserted
[388]         do j = 1, size(full)
[389]             if ( full(j) == 0.0 ) cycle ! to next j value
[390]             number = number + 1 ! non zeros inserted
[391]             sparse%rows(number) = j ! row number in full
[392]             sparse%values(number) = full(j) ! value
[393]             if ( number == n ) exit ! all non_zeros found
[394]         end do ; end function Vector_To_Sparse
[395]
[396]     function zero_sparse () result (s)
[397]         type (sv) :: s ! create sparse null vector
[398]         s%non_zeros = 0
[399]         allocate (s%rows(0), s%values(0)); end function zero_sparse
[400] end module class_sparse_Vector

```

## C.7 Problem 4.11.1 : Count the lines in an external file

```

[ 1] function inputCount(unit) result(linesOfInput)
[ 2] !-----
[ 3] ! takes a file number, counts the number of lines in that
[ 4] ! file, and returns the number of lines.
[ 5] !-----
[ 6] implicit none
[ 7] integer, intent(in) :: unit ! file unit number
[ 8] integer              :: linesOfInput ! result
[ 9] integer ioResult      ! system I/O action error code
[10] character temp        ! place to hold the character read
[11]
[12] rewind (unit) ! go to the front of the file
[13] linesOfInput = 0 ! initially, there are 0 lines
[14]
[15] do ! Until iostat says we've hit the end_of_file
[16]     read (unit,'(A)', iostat = ioResult) temp ! one char

```

```

[17]
[18]     if ( ioResult == 0 ) then          ! there were no errors
[19]         linesOfInput = linesOfInput + 1 ! increment lines
[20]     else if ( ioResult < 0 ) then      ! we've hit end-of-file
[21]         exit                          ! so exit this loop.
[22]     else ! ioResult is positive, which is a user error
[23]         write (*,*) 'inputCount: no data at unit =', unit
[24]         stop 'user read error'
[25]     end if
[26] end do
[27] rewind(unit)                ! go to the front of the file
[28] end Function inputCount

```

## C.8 Problem 4.11.3 : Computing CPU time usage

While this is mainly designed to show the use of the module `tic_toc` you should note that the intrinsic way of printing a date or time is not “pretty” and could be easily improved.

```

[ 1] program watch
[ 2] ! -----
[ 3] ! Exercise DATE_AND_TIME and SYSTEM_CLOCK functions.
[ 4] ! -----
[ 5] use tic_toc
[ 6] implicit none
[ 7] character* 8 :: the_date
[ 8] character*10 :: the_time
[ 9] integer      :: j, k
[10] !
[11] call date_and_time ( DATE = the_date )
[12] call date_and_time ( TIME = the_time )
[13] print *, 'The date is ', the_date, &
[14] & ' and the time is now ', the_time
[15] ! Display facts about the system clock.
[16] print *, ' '
[17] call system_clock ( COUNT_RATE = rate )
[18] print *, 'System clock runs at ', rate,&
[19] & ' ticks per second'
[20] !
[21] ! Call the system clock to start an execution timer.
[22] call tic
[23] !
[24] ! call run_the_job, or test with next 3 lines
[25] do k = 1, 9999
[26]     j = sqrt ( real(k*k) )
[27] end do
[28] ! Stop the execution timer and report execution time.
[29] print *, ' '
[30] print *, 'Job took ', toc (), ' seconds to execute.'
[31] end program watch ! Running gives
[32] ! The date is 19980313 and the time is now 171837.792
[33] ! System clock runs at 100 ticks per second
[34] ! Job took 0.9999999776E-02 seconds to execute.

```

## C.9 Problem 4.11.4 : Converting a string to upper case

The change from the `to_lower` should be obvious here. It seems desirable to place these two routines, and others that deal with strings into a single strings utility module.

```

[ 1] function to_upper (string) result (new_string) ! like C
[ 2] ! -----
[ 3] ! Convert a string or character to upper case
[ 4] ! (valid for ASCII or EBCDIC processors)
[ 5] ! -----
[ 6] implicit none
[ 7] character (len = *) , intent(in) :: string ! unknown length
[ 8] character (len = len(string)) :: new_string ! same length
[ 9] character (len = 26), parameter :: &
[10] UPPER = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ', &
[11] lower = 'abcdefghijklmnopqrstuvwxyz'
[12] integer :: k ! loop counter
[13] integer :: loc ! position in alphabet
[14] new_string = string ! copy everything
[15] do k = 1, len(string) ! to change letters
[16]     loc = index ( lower, string(k:k)) ! locate
[17]     if (loc /= 0) new_string(k:k) = UPPER(loc:loc) ! convert
[18] end do ! over string characters
[19] end function to_upper

```

## C.10 Problem 4.11.8 : Read two values from each line of an external file

```
[ 1] subroutine readData (inFile, lines, x, y)
[ 2] ! -----
[ 3] !   Take a file number, the number of lines to be read,
[ 4] !   and put the data into the arrays x and y
[ 5] ! -----
[ 6] ! inFile      is unit number to be read
[ 7] ! lines       is number of lines in the file
[ 8] ! x           is independent data
[ 9] ! y           is dependent data
[10] implicit none
[11] integer, intent(in) :: inFile, lines
[12] real, intent(out) :: x(lines), y(lines)
[13] integer :: j
[14]
[15] rewind (inFile) ! go to front of the file
[16] do j = 1, lines ! for the entire file
[17]   read (inFile, *) x(j), y(j) ! get the x and y values
[18] end do ! over all lines
[19] end subroutine readData
```

## C.11 Problem 4.11.14 : Two line least square fits

The extension of the single-line least squares fit shown in Fig. 4.21 is rather straightforward in that we will call subroutine `lsq_fit` multiple times. In line 37 we first call it in case a single-line fit may be more accurate than the expected two-line fit.

```
[ 1] program two_line_lsq_fit
[ 2] ! -----
[ 3] ! Best two-line linear least-squares fit of data in
[ 4] ! file specified by the user, and split in two sets
[ 5] ! -----
[ 6] implicit none
[ 7] real, allocatable :: x(:) ! independent data
[ 8] real, allocatable :: y(:) ! dependent data
[ 9]
[10] real :: fit(3), fit1(3), fit2(3) ! error results
[11] real :: left(3), right(3) ! best results
[12] real :: error ! current error
[13] real :: error_min ! best error
[14] integer :: split ! best division
[15]
[16] integer, parameter :: filenumber = 1 ! input unit
[17] character (len = 64) :: filename ! input file
[18] integer :: lines ! of input
[19] integer :: inputCount, j ! loops
[20]
[21] ! Get the name of the file containing the data.
[22] write (*, *) 'Enter the data input filename:'
[23] read (*, *) filename
[24]
[25] ! Open that file for reading.
[26] open (unit = filenumber, file = filename)
[27]
[28] ! Find the number of lines in the file
[29] lines = inputCount (filenumber)
[30] write (*, *) 'There were ', lines, ' records read.'
[31]
[32] ! Allocate that many entries in the x and y array
[33] allocate (x(lines), y(lines))
[34] call read_xy_file (filenumber, lines, x, y) ! Read data
[35] close (filenumber)
[36]
[37] call lsq_fit (lines, x, y, fit) ! single line fit
[38] print *, "Single line fit"
[39] print *, "the slope is ", fit(1)
[40] print *, "the intercept is ", fit(2)
[41] print *, "the error is ", fit(3)
[42]
```

After that we want to try all the reasonable choices for breaching the data set into two adjacent regions that are each to be fit with a different straight line. Trial variables were defined in lines 10 and 12, while the best results found are in variables declared in lines 11, 13, and 14. Note that on line 48 we have required that at least three points be used to define an approximate straight line. If we allowed two points to be employed we would get a false (or misleading) indication of zero error for such a choice. Thus, in

line 48 we begin a loop over all possible sets of three or more data points and call `lsq_fit` for each of the two segments, as seen in lines 50 and 51.

```
[ 43] ! Loop to determine the mean squared error for each
[ 44] ! of the possible two divisions of the data
[ 45] !
[ 46]     error_min = HUGE(error_min)    ! initialize the error_min
[ 47]     split = 3                      ! initialize split point
[ 48]     do j = 3, lines-3              ! 3 pts to approximate a line
[ 49]         ! least-squares fit of two data subsets
[ 50]         call lsq_fit (j,           x(1:j),      y(1:j),      fit1)
[ 51]         call lsq_fit (lines-j, x(j+1:lines), y(j+1:lines), fit2)
[ 52]         error = fit1(3) + fit2(3)
[ 53]
```

In splitting up the two data regions not that it was not necessary to copy segments of the independent and dependent data. Instead the colon operator, or implied do loops, were used in lines 50 and 51 to pass vectors with  $j$  and  $(lines - j)$  entries, respectively to the two calls to `lsq_fit`. After combining the two errors, in line 52, we update the current best choice for the data set division point in lines 55 through 58.

```
[ 54] !           does this division gives you a smaller error ?
[ 55]     if ( error < error_min ) then
[ 56]         error_min = error ; split = j
[ 57]         left      = fit1 ; right = fit2
[ 58]     end if ! current best choice
[ 59] end do ! of split choices
```

After we exit the loop, at line 59, we simply list the best results obtained. In line 73 we have also deallocated the data arrays even though it is just a formality at this point since all memory is released at the program terminates immediately afterwards. Had this been a subroutine or function then we would need to be sure that allocated variables are released when their access scope has terminated. Later versions of Fortran will do that for you, but good programmers should keep up with memory allocations.

```
[ 60] !           Display the results
[ 61]     print *, "Two line best fit; combined error is ", error_min
[ 62]     print *, "Best division of the data is:"
[ 63]     print *, "data(:j), data(j+1:), where j = ", split
[ 64]     print *, "Left line fit:"
[ 65]     print *, "the slope is      ", left(1)
[ 66]     print *, "the intercept is ", left(2)
[ 67]     print *, "the error is      ", left(3)
[ 68]     print *, "Right line fit:"
[ 69]     print *, "the slope is      ", right(1)
[ 70]     print *, "the intercept is ", right(2)
[ 71]     print *, "the error is      ", right(3)
[ 72]
[ 73]     deallocate (y, x)
[ 74] end program two_line_lsq_fit
[ 75]
```

For completeness an input routine, `read_xy_file`, is illustrated. It is elementary since it does not check for any read errors, and thus does not allow for any exception control if the read somehow fails.

```
[ 76] subroutine read_xy_file (infile, lines, x, y)
[ 77] !-----
[ 78] ! Take a file number, the number of lines to be read,
[ 79] ! and put the data into the arrays x and y
[ 80] !-----
[ 81]     implicit none
[ 82]     integer, intent(in) :: inFile    ! unit to read
[ 83]     integer, intent(in) :: lines     ! length of the file
[ 84]     real,    intent(out) :: x(lines) ! independent data
[ 85]     real,    intent(out) :: y(lines) ! dependent data
[ 86]     integer j
[ 87]     rewind (inFile) ! go to front of the file
[ 88]     do j = 1, lines ! for the entire file
[ 89]         read (infile, *) x(j), y(j) ! get the x and y values
[ 90]     end do ! over all lines
[ 91] end subroutine read_xy_file
[ 92]
```

If the supplied data file was huge, say argument `lines` has a value of ten million, the such data would probably have been stored in a binary rather than a formatted file. In that case we would simply invoke a binary read by re-writing line 89 as

```
[ 89]         read (infile) x(j), y(j) ! binary read of x and y
```

Such a change would yield a much faster input, but would still be relatively slow due to being in the loop starting at line 88. To get the fastest possible input we would have had to have saved the binary data on the file such that all the x values were stored first, followed by all the corresponding y values. In that case, we avoid the loop and get the fastest possible input by replacing lines 88–90 with:

```
[ 88]      ! sequential binary read of x and y values
[ 89]      read (infile) x, y
[ 90]      ! input complete, add iostat for exceptions
```

Here we will not go into the details about how we would have to replace subroutine inputCount an equivalent one for binary files. To do that you will have to study the Fortran INQUIRE statement for files, and its IOLENGTH option to get a hardware independent record length of a real variable.

```
[ 93]      ! Given test data in file two_line.dat:
[ 94]      ! 0.0000000e+00  1.7348276e+01
[ 95]      ! 1.0000000e+00  6.5017349e+01
[ 96]      ! 2.0000000e+00  8.7237749e+01
[ 97]      ! 3.0000000e+00  1.2433478e+02
[ 98]      ! 4.0000000e+00  1.5456681e+02
[ 99]      ! 5.0000000e+00  1.8956219e+02
[100]      ! 6.0000000e+00  2.1740486e+02
[101]      ! 7.0000000e+00  2.3138619e+02
[102]      ! 8.0000000e+00  2.7995041e+02
[103]      ! 9.0000000e+00  3.1885162e+02
[104]      ! 1.0000000e+01  3.4628642e+02
[105]      ! 1.1000000e+01  3.3522546e+02
[106]      ! 1.2000000e+01  3.7626218e+02
[107]      ! 1.3000000e+01  3.9577060e+02
[108]      ! 1.4000000e+01  4.2217988e+02
[109]      ! 1.5000000e+01  4.3388828e+02
[110]      ! 1.6000000e+01  4.5897959e+02
[111]      ! 1.7000000e+01  4.9506511e+02
[112]      ! 1.8000000e+01  5.0747649e+02
[113]      ! 1.9000000e+01  5.2168101e+02
[114]      ! 2.0000000e+01  5.2976511e+02
```

Assuming the formatted data are stored in file two\_line.dat, as shown above we obtain the best two straight line fit.

```
[115]      ! Running the program gives:
[116]      !
[117]      ! Enter the data input filename: two_line.dat
[118]      ! There were 21 records read.
[119]      ! Single line fit
[120]      ! the slope is      25.6630135
[121]      ! the intercept is 53.2859993
[122]      ! the error is      343.854675
[123]      ! Two line best fit; combined error is 126.096634
[124]      ! Best division of the data is:
[125]      ! data(:j), data(j+1:), where j = 11
[126]      ! Left line fit:
[127]      ! the slope is      31.9555302
[128]      ! the intercept is 24.9447269
[129]      ! the error is      46.060421
[130]      ! Right line fit:
[131]      ! the slope is      21.6427555
[132]      ! the intercept is 112.166664
[133]      ! the error is      80.0362091
[134]
```

Check this out by plotting the data points and the three straight line segments. Just remember that the first line covers the whole domain, while the second goes only up to halfway between points 11 and 12 while the third line runs from there to the end of the independent data.

## C.12 Problem 4.11.15 : Find the next available file unit

The INQUIRE statement has a lot of very useful features that return information based on the unit number, or the file name. It can also tell you how much storage a particular type of record requires (like the sizeof function in C and C++). Here we use only the ability to determine if a unit number is currently open. To do that we begin by checking the unit number that follows the last one we utilized. Line 9 declares that variable, last\_unit and initializes it to 0. The save attribute in that line assures that the latest value of last\_unit will always be saved and available on each subsequent use of the function. Since the standard input/output units have numbers less than ten we allow the unit numbers to be used to range from 10 to 999, as seen in line 8. However, the upper limit could be changed.

Lines 14–18 determine if the unit after last\_unit is closed. If so that unit will be used and we are basically finished. We set the return value, next, update last\_unit, and return.

```
[ 1] function get_next_io_unit () result (next)
[ 2] ! * * * * *
[ 3] ! find a unit number available for i/o action
[ 4] ! * * * * *
[ 5] implicit none
[ 6] integer :: next ! the next available unit number
[ 7]
[ 8] integer, parameter :: min_unit = 10, max_unit = 999
[ 9] integer, save :: last_unit = 0 ! initialize
[10] integer :: count ! number of failures
[11] logical :: open ! file status
[12]
[13] count = 0 ; next = min_unit - 1
[14] if ( last_unit > 0 ) then ! check next in line
[15]     next = last_unit + 1
[16]     inquire (unit=next, opened=open)
[17]     if ( .not. open ) last_unit = next ! found it
[18]     return
```

Otherwise, if the unit after last\_unit is open we must loop over all the higher unit numbers in search of one that is closed. If we succeed then we update last\_unit and return by exiting the forever loop, as seen in lines 24 and 25.

```
[19] else ! loop through allowed units
[20]     do ! forever
[21]         next = next + 1
[22]         inquire (unit=next, opened=open)
[23]         if ( .not. open ) then
[24]             last_unit = next ! found it
[25]             exit ! the unit loop
[26]         end if
```

At this point it may be impossible to find a unit. However, with 999 units available it is likely that one that was previously in use has now been closed and is available again. Before aborting we reset the search and allow three cycles to find a unit that is now free. That is done in lines 27–31.

```
[27]         if ( next == max_unit ) then ! attempt reset 3 times
[28]             last_unit = 0
[29]             count = count + 1
[30]             if ( count <= 3 ) next = min_unit - 1
[31]         end if ! reset try
```

In the unlikely event that all allowed units are still in use we abort the function after giving some insight to why.

```
[32]         if ( next > max_unit ) then ! abort
[33]             print *, 'ERROR: max unit exceeded in get_next_io_unit'
[34]             stop 'ERROR: max unit exceeded in get_next_io_unit'
[35]         end if ! abort
[36]     end do ! over unit numbers
[37] end if ! last_unit
[38] end function get_next_io_unit
```

## C.13 Problem 5.4.4 : Polymorphic interface for the class 'Position\_Angle'

```
[ 1] module class_Position_Angle ! file: class_Position_Angle.f90
[ 2] use class_Angle
[ 3] implicit none
[ 4] type Position_Angle ! angle in deg, min, sec
[ 5]     private
[ 6]     integer :: deg, min ! degrees, minutes
[ 7]     real :: sec ! seconds
[ 8]     character :: dir ! N | S, E | W
[ 9] end type
```

The above type definitions are unchanged. The only new part of the module for this class is the INTERFACE given in the following four lines.

```
[10] interface Position_Angle_ ! generic constructor
[11]     module procedure Decimal_sec, Decimal_min
[12]     module procedure Int_deg, Int_deg_min, Int_deg_min_sec
[13] end interface
[14] contains . . .
```

Returning to the original main program:

```

[ 1] program main
[ 2]   use class_Great_Arc
[ 3]   implicit none
[ 4]   type (Great_Arc)      :: arc
[ 5]   type (Global_Position) :: g1, g2
[ 6]   type (Position_Angle) :: a1, a2
[ 7]   type (Angle)          :: ang
[ 8]   real                  :: deg, rad

```

We simply replace all the previous constructor calls with the generic function `Position_Angle_` as shown on lines 8 through 17 below.

```

[ 9]   a1 = Position_Angle_ (10, 30, 0., "N") ! note decimal point
[10]   call List_Position_Angle (a1)
[11]   a1 = Position_Angle_ (10, 30, 0, "N")
[12]   call List_Position_Angle (a1)
[13]   a1 = Position_Angle_ (10, 30, "N")
[14]   call List_Position_Angle (a1)
[15]   a1 = Position_Angle_ (20, "N")
[16]   call List_Position_Angle (a1)
[17]   a2 = Position_Angle_ (30, 48, 0., "N")
[18]   call List_Position_Angle (a2)

```

## C.14 Problem 6.4.1 : Using a function with the same name in two classes

```

[ 1] include 'class_X.f90'
[ 2] include 'class_Y.f90'
[ 3] program main ! modified from Fig. 4.6.2-3F
[ 4]   use class_Y, Y_f => f ! renamed in main
[ 5]   implicit none
[ 6]   type (X_) :: x, z ; type (Y_) :: y
[ 7]   x%a = 22 ! assigns 22 to the a defined in X
[ 8]   call X_f(x) ! invokes the f() defined in X
[ 9]   print *, "x%a = ", x%a ! lists the a defined in X
[10]   y%a = 44 ! assigns 44 to the a defined in Y
[11]   x%a = 66 ! assigns 66 to the a defined in X
[12]   call Y_f(y) ! invokes the f() defined in Y
[13]   call X_f(x) ! invokes the f() defined in X
[14]   print *, "y%a = ", y%a ! lists the a defined in X
[15]   print *, "x%a = ", x%a ! lists the a defined in X
[16]   z%a = y%a ! assign Y a to z in X
[17]   print *, "z%a = ", z%a ! lists the a defined in X
[18] end program main ! Running gives:
[19] ! X_ f() executing ! x%a = 22
[20] ! Y_ f() executing ! X_ f() executing
[21] ! y%a = 44 ! x%a = 66
[22] ! z%a = 44

```

## C.15 Problem 6.4.3 : Revising the employee-manager classes

The changes are relatively simple. First we add two lines in the `Employee` class:

```

interface setData ! a polymorphic member
module procedure setDataE ; end interface

```

Then we change two other lines:

```

[ 8]   empl = setData ( "Burke", "John", 25.0 )
[14]   mgr = Manager_ ( "Kovacs", "Jan", 1200.0 ) ! constructor

```

The generic `setData` could not also contain `setDataM` because it has the same argument signature as `setDataE` and the compiler would not be able to tell which dynamic binding to select.





## Appendix D

# Companion C++ Examples

### D.1 Introduction

It is necessary to be multilingual in computer languages today. Since C++ is often used in the OOP literature it should be useful to have C++ versions of the same code given earlier in F90. In most cases these examples have the same variable names and the line numbers are usually very close to each other. This appendix will allow you to flip from F90 examples in Chapter 4 of the main body of the text to see similar operations in C++.

---

```
[ 1] #include <iostream.h> // system i/o files
[ 2] #include <math.h>    // system math files
[ 3] main ()
[ 4] // Examples of simple arithmetic in C++
[ 5] {
[ 6]     int    Integer_Var_1, Integer_Var_2; // user inputs
[ 7]     int    Mult_Result, Div_Result, Add_Result
[ 8]     int    Sub_Result, Mod_Result;
[ 9]     double Pow_Result, Sqrt_Result;
[10]     cout << "Enter two integers: ";
[11]     cin >> Integer_Var_1, Integer_Var_2;
[12]
[13]     Add_Result = Integer_Var_1 + Integer_Var_2;
[14]     cout << Integer_Var_1 << " + " << Integer_Var_2 << " = "
[15]         << Add_Result << endl;
[16]     Sub_Result = Integer_Var_1 - Integer_Var_2 ;
[17]     cout << Integer_Var_1 << " - " << Integer_Var_2 << " = "
[18]         << Sub_Result << endl;
[19]     Mult_Result = Integer_Var_1 * Integer_Var_2 ;
[20]     cout << Integer_Var_1 << " * " << Integer_Var_2 << " = "
[21]         << Mult_Result << endl;
[22]     Div_Result = Integer_Var_1 / Integer_Var_2 ;
[23]     cout << Integer_Var_1 << " / " << Integer_Var_2 << " = "
[24]         << Div_Result << endl;
[25]     Mod_Result = Integer_Var_1 % Integer_Var_2; // remainder
[26]     cout << Integer_Var_1 << " % " << Integer_Var_2 << " = "
[27]         << Mod_Result << endl;
[28]     Pow_Result = pow ((double)Integer_Var_1, (double)Integer_Var_2);
[29]     cout << Integer_Var_1 << " ^ " << Integer_Var_2 << " = "
[30]         << Pow_Result << endl;
[31]     Sqrt_Result = sqrt( (double)Integer_Var_1 );
[32]     cout << "Square root of " << Integer_Var_1 << " is "
[33]         << Sqrt_Result << endl;
[34] } // end main, Running produces:
[35] // Enter two integers: 25 4
[36] // 25 + 4 = 29
[37] // 25 - 4 = 21
[38] // 25 * 4 = 100
[39] // 25 / 4 = 6, note integer
[41] // 25 % 4 = 1
[42] // 25 ^ 4 = 390625
[43] // Square root of 25 = 5
```

**Figure D.1:** Typical Math and Functions in C++

```

[ 1] #include <iostream.h> // system i/o files
[ 2] main ()
[ 3] // Examples of a simple loop in C++
[ 4] {
[ 5]     int Integer_Var;
[ 6]
[ 7]     for (Integer_Var = 0; Integer_Var < 5; Integer_Var ++ )
[ 8]     {
[ 9]         cout << "The loop variable is: " << Integer_Var << endl;
[10]     } // end for
[11]
[12]     cout << "The final loop variable is: " << Integer_Var << endl;
[13]
[14] } // end main // Running produces:
[15] // The loop variable is: 0
[16] // The loop variable is: 1
[17] // The loop variable is: 2
[18] // The loop variable is: 3
[19] // The loop variable is: 4
[20] // The final loop variable is: 5 <- NOTE

```

**Figure D.2:** Typical Looping Concepts in C++

```

[ 1] #include <iostream.h> // system i/o files
[ 2] main ()
[ 3] // Examples of simple array indexing in C++
[ 4] {
[ 5]     int MAX = 5, loopcount;
[ 6]     int Integer_Array[5] ;
[ 7]     // or, int Integer_Array[5] = {10, 20, 30, 40, 50 };
[ 8]
[ 9]     Integer_Array[0] = 10 ; // C arrays start at zero
[10]     Integer_Array[1] = 20 ; Integer_Array[2] = 30 ;
[11]     Integer_Array[3] = 40 ; Integer_Array[4] = 50 ;
[12]
[13]     for ( loopcount = 0; loopcount < MAX; loopcount ++ )
[14]     {
[15]         cout << "The loop counter is: " << loopcount
[16]         << " with an array value of: " << Integer
[17]         // end for loop
[18]         cout << "The final loop counter is: " << loopcount << endl ;
[19]     } // end main
[20]
[21] // Running produces:
[22] // The loop counter is: 0 with an array value of: 10
[23] // The loop counter is: 1 with an array value of: 20
[24] // The loop counter is: 2 with an array value of: 30
[25] // The loop counter is: 3 with an array value of: 40
[26] // The loop counter is: 4 with an array value of: 50
[27] // The final loop counter is: 5

```

**Figure D.3:** Simple Array Indexing in C++

```

[ 1] #include <iostream.h> // system i/o files
[ 2] main ()
[ 3] // Examples of relational "if" operator, via C++
[ 4] {
[ 5]     int Integer_Var_1, Integer_Var_2; // user inputs
[ 6]
[ 7]     cout << "\nEnter two integers: ";
[ 8]     cin >> Integer_Var_1, Integer_Var_2;
[ 9]
[10]     if ( Integer_Var_1 > Integer_Var_2 )
[11]         cout << Integer_Var_1 << " is greater than " << Integer_Var_2;
[12]
[13]     if ( Integer_Var_1 < Integer_Var_2 )
[14]         cout << Integer_Var_1 << " is less than " << Integer_Var_2;
[15]
[16]     if ( Integer_Var_1 == Integer_Var_2 )
[17]         cout << Integer_Var_1 << " is equal to " << Integer_Var_2;
[18]
[19] } // end main
[20]
[21] // Running with 25 and 4 produces: 25 4
[22] // Enter two integers:
[23] // 25 is greater than 4

```

**Figure D.4:** Typical Relational Operators in C++

```

[ 1] #include <iostream.h>
[ 2] main ()
[ 3] // Illustrate a simple if-else logic in C++
[ 4] {
[ 5]     int Integer_Var;
[ 6]
[ 7]     cout << "Enter an integer: ";
[ 8]     cin >> Integer_Var;
[ 9]
[10]     if ( Integer_Var > 5 && Integer_Var < 10 )
[11]     {
[12]         cout << Integer_Var << " is greater than 5 and less than 10"
[13]         << endl; }
[14]     else
[15]     {
[16]         cout << Integer_Var << " is not greater than 5 and less than 10"
[17]         << endl; } // end of range of input
[18] } // end program main
[19]
[20]
[21] // Running with 3 gives: 3 is not greater than 5 and less than 10
[22] // Running with 8 gives: 8 is greater than 5 and less than 10

```

**Figure D.5:** Typical If-Else Uses in C++

```

[ 1] #include <iostream.h>
[ 2] main ()
[ 3] // Examples of Logical operators in C++
[ 4] {
[ 5]     int Logic_Var_1, Logic_Var_2;
[ 6]
[ 7]     cout << "Enter logical value of A (1 or 0): ";
[ 8]     cin >> Logic_Var_1;
[ 9]
[10]     cout << "Enter logical value of B (1 or 0): ";
[11]     cin >> Logic_Var_2;
[12]
[13]     cout << "NOT A is " << !Logic_Var_1 << endl;
[14]
[15]     if ( Logic_Var_1 && Logic_Var_2 )
[16]     {
[17]         cout << "A ANDEd with B is true " << endl;
[18]     }
[19]     else
[20]     {
[21]         cout << "A ANDEd with B is false " << endl;
[22]     } // end if for AND
[23]
[24]     if ( Logic_Var_1 || Logic_Var_2 )
[25]     {
[26]         cout << "A ORed with B is true " << endl;
[27]     }
[28]     else
[29]     {
[30]         cout << "A ORed with B is false " << endl;
[31]     } // end if for OR
[32]
[33]     if ( Logic_Var_1 == Logic_Var_2 )
[34]     {
[35]         cout << "A EQiValent with B is true " << endl;
[36]     }
[37]     else
[38]     {
[39]         cout << "A EQiValent with B is false " << endl;
[40]     } // end if for EQV
[41]
[42]     if ( Logic_Var_1 != Logic_Var_2 )
[43]     {
[44]         cout << "A Not EQiValent with B is true " << endl;
[45]     }
[46]     else
[47]     {
[48]         cout << "A Not EQiValent with B is false " << endl;
[49]     } // end if for NEQV
[50]
[51] } // end main
[52] // Running with 1 and 0 produces:
[53] // Enter logical value of A (1 or 0): 1
[54] // Enter logical value of B (1 or 0): 0
[55] // NOT A is 0
[56] // A ANDEd with B is false
[57] // A ORed with B is true
[58] // A EQiValent with B is false
[59] // A Not EQiValent with B is true

```

**Figure D.6:** Typical Logical Operators in C++

```

[ 1] // Program to find the maximum of a set of integers
[ 2] #include <iostream.h>
[ 3] #include <stdlib.h> // for exit
[ 4] #define ARRAYLENGTH 100
[ 5] long integers[ARRAYLENGTH];
[ 6]
[ 7] // Function interface prototype
[ 8] long maxint(long [], long);
[ 9]
[10] // Main routine
[11]
[12] main() { // Read in the number of integers
[13] long i, n;
[14]
[15]     cout << "Find maximum; type n: "; cin >> n;
[16]     if ( n > ARRAYLENGTH || n < 0 ) {
[17]         cout << "Value you typed is too large or negative." << endl;
[18]         exit(1);
[19]     } // end if
[20]
[21]     for (i = 0; i < n; i++) { // Read in the user's integers
[22]         cout << "Integer " << (i+1) << ": "; cin >> integers[i]; cout
[23]         << endl; } // end for
[24]     cout << "Maximum: ", cout << maxint(integers, n); cout << endl;
[25] } // end main
[26]
[27] // Find the maximum of an array of integers
[28] long maxint(long input[], long input_length) {
[29] long i, max;
[30]
[31]     for (max = input[0], i = 1; i < input_length; i++) {
[32]         if ( input[i] > max ) {
[33]             max = input[i]; } // end if
[34]     } // end for
[35]     return(max);
[36] } // end maxint // produces this result
[37] // Find maximum; type n: 4
[38] // Integer 1: 9
[39] // Integer 2: 6
[40] // Integer 3: 4
[41] // Integer 4: -99
[42] // Maximum: 9

```

**Figure D.7:** Search for Largest Value in C++

```

[ 1] #include <iostream.h>
[ 2]
[ 3] // declare the interface prototypes
[ 4] void Change ( int& Input_Val);
[ 5] void No_Change ( int Input_Val);
[ 6]
[ 7] main ()
[ 8] // illustrate passing by reference and by value in C++
[ 9] {
[10]     int Input_Val;
[11]
[12]     cout << "Enter an integer: ";
[13]     cin >> Input_Val;
[14]     cout << "Input value was " << Input_Val << endl;
[15]
[16]     // pass by value
[17]     No_Change ( Input_Val ); // Use but do not change
[18]     cout << "After No_Change it is " << Input_Val << endl;
[19]
[20]     // pass by reference
[21]     Change ( Input_Val ); // Use and change
[22]     cout << "After Change it is " << Input_Val << endl;
[23] }
[24]
[25] void Change (int& Value)
[26] {
[27]     // changes Value in calling code IF passed by reference
[28]     Value = 100;
[29]     cout << "Inside Change it is set to " << Value << endl;
[30] }
[31]
[32] void No_Change (int Value)
[33] {
[34]     // does not change Value in calling code IF passed by value
[35]     Value = 100;
[36]     cout << "Inside No_Change it is set to " << Value << endl;
[37] }
[38] // Running gives:
[39] // Enter an integer: 12
[40] // Input value was 12
[41] // Inside No_Change it is set to 100
[42] // After No_Change it is 12
[43] // Inside Change it is set to 100
[44] // After Change it is 100

```

**Figure D.8:** Passing Arguments by Reference and by Value in C++

```

[ 1] #include <iostream.h>
[ 2] main ()
[ 3] // Compare two character strings in C++
[ 4] // Concatenate two character strings together
[ 5] {
[ 6]     char String1[40];
[ 7]     char String2[20];
[ 8]     int  length;
[ 9]
[10]     cout << "Enter first string (20 char max):";
[11]     cin >> String1;
[12]
[13]     cout << "Enter second string (20 char max):";
[14]     cin >> String2;
[15]
[16]     // Compare
[17]     if ( !strcmp(String1, String2) ) {
[18]         cout << "They are the same." << endl;
[19]     }
[20]     else {
[21]         cout << "They are different." << endl;
[22]     } // end if the same
[23]
[24]     // Concatenate
[25]     strcat(String1, String2) ; // add onto String1
[26]
[27]     cout << "The combined string is: " << String1 << endl;
[28]     length = strlen( String1 );
[29]     cout << "The combined length is: " << length << endl;
[30]     length = strlen( String1 );
[31]
[32] } // end main
[33] // Running with "red" and "bird" produces:
[34] // Enter first string (20 char max): red
[35] // Enter second string (20 char max): bird
[36] // They are different.
[37] // The combined string is: redbird
[38] // The combined length is: 7
[39] // But, "the red" and "bird" gives unexpected results

```

**Figure D.9:** Using Two Strings in C++

```

[ 1] #include <iostream.h>
[ 2] #include <stdlib.h>
[ 3] #include <math.h> // system math files
[ 4]
[ 5] main()
[ 6] // Convert a character string to an integer in C++
[ 7] {
[ 8]     char Age_Char[5];
[ 9]     int  age;
[10]
[11]     cout << "Enter your age: ";
[12]     cin >> Age_Char;
[13]
[14]     // convert with intrinsic function
[15]     age = atoi(Age_Char);
[16]
[17]     cout << "Your integer age is " << age << endl;
[18]     cout << "Your hexadecimal age is " << hex << age << endl;
[19]     cout << "Your octal age is " << oct << age << endl;
[20]
[21] } // end of main
[22]
[23] // Running gives:
[24] // Enter your age: 45
[25] // Your integer age is 45.
[26] // Your hexadecimal age is 2d.
[27] // Your octal age is 55.

```

**Figure D.10:** Converting a String to an Integer with C++

```

[ 1] #include <iostream.h>
[ 2]
[ 3] // Define structures and components in C++
[ 4]
[ 5] struct Person      // define a person structure type
[ 6] {
[ 7]     char    Name[20];
[ 8]     int     Age;
[ 9] };
[10]
[11] struct Who_Where   // use person type in a new structure
[12] {
[13]     struct Person Guest;
[14]     char    Address[40];
[15] };
[16]
[17] // Fill a record of the Who_Where type components
[18] main ()
[19] {
[20]     struct Who_Where Record;
[21]
[22]     cout << "Enter your name: ";
[23]     cin  >> Record.Guest.Name;
[24]
[25]     cout << "Enter your city: ";
[26]     cin  >> Record.Address;
[27]
[28]     cout << "Enter your age: ";
[29]     cin  >> Record.Guest.Age;
[30]
[31]     cout << "Hello " << Record.Guest.Age << " year old "
[32]           << Record.Guest.Name << " in " << Record.Address << endl;
[33] }
[34] // Running with input: Sammy, Houston, 104 gives
[35] // Hello 104 year old Sammy in Houston
[36] //
[37] // But try: Sammy Owl, Houston, 104 for a bug

```

**Figure D.11:** Using Multiple Structures in C++



## Appendix E

# Glossary of Object Oriented Terms

**abstract class:** A class primarily intended to define an instance, but can not be instantiated without additional methods.

**abstract data type:** An abstraction that describes a set of items in terms of a hidden data structure and operations on that structure.

**abstraction:** A mental facility that permits one to view problems with varying degrees of detail depending on the current context of the problem.

**accessor:** A public member subprogram that provides query access to a private data member.

**actor:** An object that initiates behavior in other objects, but cannot be acted upon itself.

**agent:** An object that can both initiate behavior in other objects, as well as be operated upon by other objects.

**ADT:** Abstract data type.

**AKO:** A Kind Of. The inheritance relationship between classes and their superclasses.

**allocatable array:** A named array having the ability to dynamically obtain memory. Only when space has been allocated for it does it have a shape and may it be referenced or defined.

**argument:** A value, variable, or expression that provides input to a subprogram.

**array:** An ordered collection that is indexed.

**array constructor:** A means of creating a part of an array by a single statement.

**array overflow:** An attempt to access an array element with a subscript outside the array size bounds.

**array pointer:** A pointer whose target is an array, or an array section.

**array section:** A subobject that is an array and is not a defined type component.

**assertion:** A programming means to cope with errors and exceptions.

**assignment operator:** The equal symbol, "=", which may be overloaded by a user.

**assignment statement:** A statement of the form "variable = expression".

**association:** Host association, name association, pointer association, or storage association.

**attribute:** A property of a variable that may be specified in a type declaration statement.

**automatic array:** An explicit-shape array in a procedure, which is not a dummy argument, some or all of whose bounds are provided when the procedure is invoked.

**base class:** A previously defined class whose public members can be inherited by another class. (Also called a super class.)

**behavior sharing:** A form of polymorphism, when multiple entities have the same generic interface. This is achieved by inheritance or operator overloading.

**binary operator:** An operator that takes two operands.

**bintree:** A tree structure where each node has two child nodes.

**browser:** A tool to find all occurrences of a variable, object, or component in a source code.

**call-by-reference:** A language mechanism that supplies an argument to a procedure by passing the address of the argument rather than its value. If it is modified, the new value will also take effect outside of the procedure.

**call-by-value:** A language mechanism that supplies an argument to a procedure by passing a copy of its data value. If it is modified, the new value will not take effect outside of the procedure that modifies it.

**class:** An abstraction of an object that specifies the static and behavioral characteristics of it, including their public and private nature. A class is an ADT with a constructor template from which object instances are created.

**class attribute:** An attribute whose value is common to a class of objects rather than a value peculiar to each instance of the class.

**class descriptor:** An object representing a class, containing a list of its attributes and methods as well as the values of any class attributes.

**class diagram:** A diagram depicting classes, their internal structure and operations, and the fixed relationships between them.

**class inheritance:** Defining a new derived class in terms of one or more base classes.

**client:** A software component that users services from another supplier class.

**concrete class:** A class having no abstract operations and can be instantiated.

**compiler:** Software that translates a high-level language into machine language.

**component:** A data member of a defined type within a class declaration

**constructor:** An operation, by a class member function, that initializes a newly created instance of a class. (See default and intrinsic constructor.)

**constructor operations:** Methods which create and initialize the state of an object.

**container class:** A class whose instances are container objects. Examples include sets, arrays, and stacks.

**container object:** An object that stores a collection of other objects and provides operations to access or iterate over them.

**control variable:** The variable which controls the number of loop executions.

**data abstraction:** The ability to create new data types, together with associated operators, and to hide the internal structure and operations from the user, thus allowing the new data type to be used in a fashion analogous to intrinsic data types.

**data hiding:** The concept that some variables and/or operations in a module may not be accessible to a user of that module; a key element of data abstraction.

**data member:** A public data attribute, or instance variable, in a class declaration.

**data type:** A named category of data that is characterized by a set of values. together with a way to denote these values and a collection of operations that interpret and manipulate the values. For an intrinsic type, the set of data values depends on the values of the type parameters.

**deallocation statement:** A statement which releases dynamic memory that has been previously allocated to an allocatable array or a pointer.

**debugger software:** A program that allows one to execute a program in segments up to selected break-points, and to observe the program variables.

**debugging:** The process of detecting, locating, and correcting errors in software.

**declaration statement:** A statement which specifies the type and, optionally, attributes of one or more variables or constants.

**default constructor:** A class member function with no arguments that assigns default initial values to all data members in a newly created instance of a class.

**defined operator:** An operator that is not an intrinsic operator and is defined by a subprogram that is associated with a generic identifier.

**deque:** A container that supports inserts or removals from either end of a queue.

**dereferencing:** The interpretation of a pointer as the target to which it is pointing.

**derived attribute:** An attribute that is determined from other attributes.

**derived class:** A class whose declaration indicates that it is to inherit the public members of a previously defined base class.

**derived type:** A user defined data type with components, each of which is either of intrinsic type or of another derived type.

**destructor:** An operation that cleans up an existing instance of a class that is no longer needed.

**destructor operations:** Methods which destroy objects and reclaim their dynamic memory.

**domain:** The set over which a function or relation is defined.

**dummy argument:** An argument in a procedure definition which will be associated with the actual (reference or value) argument when the procedure is invoked.

**dummy array:** A dummy argument that is an array.

**dummy pointer:** A dummy argument that is a pointer.

**dummy procedure:** A dummy argument that is specified or referenced as a procedure.

**dynamic binding:** The allocation of storage at run time rather than compile time, or the run time association of an object and one of its generic operations..

**edit descriptor:** An item in an input/output format which specifies the conversion between internal and external forms.

**encapsulation:** A modeling and implementation technique (information hiding) that separates the external aspects of an object from the internal, implementation details of the object.

**exception:** An unexpected error condition causing an interruption to the normal flow of program control.

**explicit interface:** For a procedure referenced in a scoping unit, the property of being an internal procedure, a module procedure, an external procedure that has an interface (prototype) block, a recursive procedure reference in its own scoping unit, or a dummy procedure that has an interface block.

**explicit shape array:** A named array that is declared with explicit bounds.

**external file:** A sequence of records that exists in a medium external to the program.

**external procedure:** A procedure that is defined by an external subprogram.

**FIFO:** First in, first out storage; a queue.

**friend:** A method, in C++, which is allowed privileged access to the private implementation of another object.

**function body:** A block of statements that manipulate parameters to accomplish the subprogram's purpose.

**function definition:** Program unit that associates with a subprogram name a return type, a list of arguments, and a sequence of statements that manipulate the arguments to accomplish the subprogram's purpose

**function header:** A line of code at the beginning of a function definition; includes the argument list, and the function return variable name.

**generic function:** A function which can be called with different types of arguments.

**generic identifier:** A lexical token that appears in an INTERFACE statement and is associated with all the procedures in the interface block.

**generic interface block:** A form of interface block which is used to define a generic name for a set of procedures.

**generic name:** A name used to identify two or more procedures, the required one being determined by the types of the non-optional arguments in the procedure invocation.

**generic operator:** An operator which can be invoked with different types of operands.

**Has-A:** A relationship in which the derived class has a property of the base class.

**hashing technique:** A technique used to create a hash table, in which the array element where an item is to be stored is determined by converting some item feature into an integer in the range of the size of the table.

**heap:** A region of memory used for data structures dynamically allocated and deallocated by a program.

**host:** The program unit containing a lower (hosted) internal procedure.

**host association:** Data, and variables automatically available to an internal procedure from its host.

**information hiding:** The principle that the state and implementation of an object should be private to that object and only accessible via its public interface.

**inheritance:** The relationship between classes whereby one class inherits part or all of the public description of another base class, and instances inherit all the properties and methods of the classes which they contain.

**instance:** A individual example of a class invoked via a class constructor.

**instance diagram:** A drawing showing the instance connection between two objects along with the number or range of mapping that may occur.

**instantiation:** The process of creating (giving a value to) instances from classes.

**intent:** An attribute of a dummy argument that which indicates whether it may be used to transfer data into the procedure, out of the procedure, or both.

**interaction diagram:** A diagram that shows the flow of requests, or messages between objects.

**interface:** The set of all signatures (public methods) defined for an object.

**internal file:** A character string that is used to transfer and/or convert data from one internal storage mode to a different internal storage mode.

**internal procedure:** A procedure contained within another program unit, or class, and which can only be invoked from within that program unit, or class.

**internal subprogram:** A subprogram contained in a main program or another subprogram.

**intrinsic constructor:** A class member function with the same name as the class which receives initial values of all the data members as arguments.

**Is-A:** A relationship in which the derived class is a variation of the base class.

**iterator:** A method that permits all parts of a data structure to be visited.

**keyword:** A programming language word already defined and reserved for a single special purpose.

**LIFO:** Last in, first out storage; a stack.

**link:** The process of combining compiled program units to form an executable program.

**linked list:** A data structure in which each element identifies its predecessor and/or successor by some form of pointer.

**linker:** Software that combines object files to create an executable machine language program.

**list:** An ordered collection that is not indexed.

**map:** An indexed collection that may be ordered.

**matrix:** A rank-two array.

**member data:** Variables declared as components of a defined type and encapsulated in a class.

**member function:** Subprograms encapsulated as members of a class.

**method:** A class member function encapsulated with its class data members.

**method resolution:** The process of matching a generic operation on an object to the unique method appropriate to the object's class.

**message:** A request, from another object, for an object to carry out one of its operations.

**message passing:** The philosophy that objects only interact by sending messages to each other that request some operations to be performed.

**module:** A program unit which allows other program units to access variables, derived type definitions, classes and procedures declared within it by USE association.

**module procedure:** A procedure which is contained within a module, and usually used to define generic interfaces, and/or to overload or define operators.

**nested:** Placement of a control structure inside another control structure.

**object:** A concept, or thing with crisp boundaries and meanings for the problem at hand; an instance of a class.

**object diagram:** A graphical representation of an object model showing relationships, attributes, and operations.

**object-oriented (OO):** A software development strategy that organizes software as a collection of objects that contain both data structure and behavior. (Abbreviated OO.)

**object-oriented programming (OOP):** Object-oriented programs are object-based, class-based, support inheritance between classes and base classes and allow objects to send and receive messages.

**object-oriented programming language:** A language that supports objects (encapsulating identity, data, and operations), method resolution, and inheritance.

**octree:** A tree structure where each node has eight child nodes.

**OO (acronym):** Object-oriented.

**operand:** An expression or variable that precedes or succeeds an operator.

**operation:** Manipulation of an object's data by its member function when it receives a request.

**operator overloading:** A special case of polymorphism; attaching more than one meaning to the same operator symbol. 'Overloading' is also sometimes used to indicate using the same name for different objects.

**overflow:** An error condition arising from an attempt to store a number which is too large for the storage location specified; typically caused by an attempt to divide by zero.

**overloading:** Using the same name for multiple functions or operators in a single scope.

**overriding:** The ability to change the definition of an inherited method or attribute in a subclass.

**parameterized classes:** A template for creating real classes that may differ in well-defined ways as specified by parameters at the time of creation. The parameters are often data types or classes, but may include other attributes, such as the size of a collection. (Also called generic classes.)

**pass-by-reference:** Method of passing an argument that permits the function to refer to the memory holding the original copy of the argument

**pass-by-value:** Method of passing an argument that evaluates the argument and stores this value in the corresponding formal argument, so the function has its own copy of the argument value

**pointer:** A single data object which stands for another (a "target"), which may be a compound object such as an array, or defined type.

**pointer array:** An array which is declared with the pointer attribute. Its shape and size may not be determined until they are created for the array by means of a memory allocation statement.

**pointer assignment statement:** A statement of the form "pointer-name  $\Rightarrow$  target".

**polymorphism:** The ability of an function/operator, with one name, to refer to arguments, or return types, of different classes at run time.

**post-condition:** Specifies what must be true after the execution of an operation.

**pre-condition:** Specifies the condition(s) that must be true before an operation can be executed.

**private:** That part of an class, methods or attributes, which may not be accessed by other classes, only by instances of that class.

**protected:** (Referring to an attribute or operation of a class in C++) accessible by methods of any descendent of the current class.

**prototype:** A statement declaring a function's return type, name, and list of argument types.

**pseudocode:** A language of structured English statements used in designing a step-by-step approach to solving a problem.

**public:** That part of an object, methods or attributes, which may be accessed by other objects, and thus constitutes its interface.

**quadtree:** A tree structure where each tree node has four child nodes.

**query operation:** An operation that returns a value without modifying any objects.

**rank:** Number of subscripted variables an array has. A scalar has rank zero, a vector has rank one, a matrix has rank two.

**scope:** That part of an executable program within which a lexical token (name) has a single interpretation.

**section:** Part of an array.

**sequential:** A kind of file in which each record is written (read) after the previously written (read) record.

**server:** An object that can only be operated upon by other objects.

**service:** A class member function encapsulated with its class data members.

**shape:** The rank of an array and the extent of each of its subscripts. Often stored in a rank-one array.

**side effect:** A change in a variable's value as a result of using it as an operand, or argument.

**signature:** The combination of a subprogram's (operator's) name and its argument (operand) types. Does not include function result types.

**size:** The total number of elements in an array.

**stack:** Region of memory used for allocation of function data areas; allocation of variables on the stack occurs automatically when a block is entered, and deallocation occurs when the block is exited

**stride:** The increment used in a subscript triplet.

**strong typing:** The property of a programming language such that the type of each variable must be declared.

**structure component:** The part of a data object of derived type corresponding to a component of its type.

**sub-object:** A portion of a data object that may be referenced or defined independently of other portions. It may be an array element, an array section, a structure component, or a substring.

**subprogram:** A function or subroutine subprogram.

**subprogram header:** A block of code at the beginning of a subprogram definition; includes the name, and the argument list, if any.

**subscript triplet:** A method of specifying an array section by means of the initial and final subscript integer values and an optional stride (or increment).

**super class:** A class from which another class inherits. (See base class.)

**supplier:** Software component that implements a new class with services to be used by a client software component.

**target:** The data object pointed to by a pointer, or reference variable.

**template:** An abstract recipe with parameters for producing concrete code for class definitions or sub-program definitions.

**thread:** The basic entity to which the operating system allocates CPU time.

**tree:** A form of linked list in which each node points to at least two other nodes, thus defining a dynamic data structure.

**unary operator:** An operator which has only one operand.

**undefined:** A data object which does not have a defined value.

**underflow:** An error condition where a number is too close to zero to be distinguished from zero in the floating-point representation being used.

**utility function:** A private subprogram that can only be used within its defining class.

**vector:** A rank-one array. An array with one subscript.

**vector subscript:** A method of specifying an array section by means of a vector containing the subscripts of the elements of the parent array that are to constitute the array section.

**virtual function:** A genetic function, with a specific return type, extended later for each new argument type.

**void subprogram:** A C++ subprogram with an empty argument list and/or a subroutine with no returned argument.

**work array:** A temporary array used for the storage of intermediate results during processing.



## Appendix F

# Subject Index

In the index the F90/95 intrinsic attributes, functions, subroutines, statements, etc. are shown in upper-case letters even though Fortran is not case sensitive. The page numbers are cited with the chapter (or appendix) number followed by a period, followed by the pages in that chapter separated by commas. Topics that occur frequently are only cited at their first few uses.

- A edit descriptor 4.34,35
- ABS intrinsic 4.24 B.1
- abstract data type 2.5,7
- access restriction 1.23 2.5 3.1
- ACCESS specifier
- accessor A.1
- accuracy of real arithmetic
- ACHAR intrinsic 4.33
- ACTION specifier
- actual argument
- actual array argument
- actual pointer argument
- ADVANCE specifier 3.8 4.49,50
- allocatable array 5.3
- ALLOCATABLE attribute 5.3 B.12
- ALLOCATE
  - statement 5.3 B.13
  - status 4.29,30 B.13
- ALLOCATED intrinsic B.1
- allocation statement B.4
- allocation status
- alphabetic sorting
- alternate RETURN statement B.18
- ampersand
- analysis
- .AND. 3.17 4.16,17,48
- angle class
- APOSTROPHE
- apostrophe edit descriptor
- argument
  - actual
  - association
  - dummy
  - function 4.23
  - list
  - presence
  - subroutine 4.23
- arguments 1.15
- arithmetic expression
- arithmetic IF statement
- arithmetic operators 4.4
- arithmetic unit
- array
  - allocatable
  - allocation
  - assumed-shape
  - assumed-size 4.31
  - automatic 5.2,3 A.1
  - bounds
  - conformable
  - constant
  - constructor 4.24 5.4,10,14 B.4
  - deferred-shape
  - dimensions B.4,7
  - dummy argument 5.3
  - element
  - explicit-shape
  - extent 5.1
  - extraction 5.9
  - flip 5.9
  - initialization 5.2,4,5
  - input
  - inquiry B.4
  - mask 5.10,11
  - name in an I/O list
  - output
  - overflow A.1
  - packing B.4
  - pointer
  - rank 4.31,53
  - reduction B.5
  - reshape 5.13,14
  - shape 5.1
  - shifts 5.14
  - size 5.1
  - subscripts 5.2
  - unknown size 4.31
- array element order
- array of pointers
- array processing intrinsics 5.6,7,11
- array section
- array specification
- array variable
- array-valued derived-type component

- array-valued function 4.31
- array-valued literal constant
- ASCII
  - character 2.1 4.33,35
  - collating sequence 4.32
- assembly language 1.18
- ASSIGN statement B.16,17
- assignment operator 1.11,12
- assignment statement A.1 B.17
- ASSOCIATED intrinsic 4.45
- assumed-length dummy argument
- assumed-shape array
- assumed-size array
- asterisk
  - format specifier 2.3,4,5
  - default input unit
  - default output unit
- attribute
  - ALLOCATABLE B.16
  - DIMENSION B.16
  - EXTERNAL
  - INTENT B.16
  - INTRINSIC
  - KIND B.16
  - object 1.22, 2.4, A.1
  - OPTIONAL B.16
  - PARAMETER B.16
  - POINTER B.16
  - PRIVATE B.16
  - PUBLIC B.16
  - SAVE B.16
  - SEQUENCE
  - TARGET B.16
- attributes 1.23
- automatic array 4.48 5.3
- automatic character length
- B edit descriptor 4.36
- back substitution
- back-up B.5
- BACKSPACE statement 4.29 B.17
- base class 7.1 A.2
- behaviour 1.23
- binary digit 4.29
- binary file 5.5
- binary number 4.36
- binary operator 4.29
- binary tree
- bit intrinsic functions 4.29 B.5
- blank character 4.33
- BLANK specifier
- BLOCK DATA statement B.16
- block IF construct
- block WHERE construct
- BN edit descriptor
- Boolean 2.1
- bottom-up design 1.4
- bounds
  - array B.5
  - lower
  - scalar B.5
  - upper
- bubble sort 4.52
- bug 1.8
- BZ edit descriptor
- C language 1.1
- C++ language 1.1,11,15,21 2.2
  - 4.4,5,8,9,10,14,17,19,22
  - 4.23,27,29,35,38,39,40,41
  - 4.45,47 5.2,4,9,24,25
- call by reference 4.31
- call by value 4.31
- CALL statement
- CASE construct
- CASE DEFAULT statement 4.17,18
- case expression
- case selector 4.17,18
- CASE statement 4.17,18
- character
  - argument
  - array
  - assignment
  - constant 3.8
  - control

- data I/O
- expression
- length
- pointer 4.45
- substring
- character edit descriptor 3.8
- character set
  - default 4.32
  - Fortran
- CHARACTER statement 4.3 B.13
- CHARACTER type 2.1 4.31
- characteristics
  - dummy argument
  - result variable
- chemical element 2.4,7
- Circle class 3.2,4,19
- class
  - base
  - defined
  - derived 4.38
  - hierarchies 3.2
- classes 1.18,23 2.8 3.1
- CLOSE statement 4.29
- closing a file
- CMPLX intrinsic 5.7
- collating sequence
- colon edit descriptor
- colon operator 4.7,25 5.8
- column extraction 5.9
- comment
  - fixed source
  - free source
  - ! statement
- comments 1.1,6 4.1
- COMMON block
- COMMON block name
- COMMON statement 4.27 B.16
- comparing character strings 4.32
- comparison of two real
- compiler 1.19 3.6
- COMPLEX statement 4.3 B.13,21
- COMPLEX type 2.1, B.6
- component
  - derived type 2.4
- composition
- computed GO TO B.16,21
- concatenation
  - operator
- condition
  - end-of-file 4.29
  - end-of-record 4.29
  - error
- conditionals 1.6,7,14 4.13
- conformable arrays
- connectivity 5.12
- constant
  - character
  - derived type
  - integer
  - literal
  - named
  - real
- constant expression
- constructor
  - default 1.23
  - intrinsic 2.5 3.2
  - manual 2.8 3.7
  - structure
- constructors 1.23 3.2
- containers 8.1
- CONTAINS statement 2.9 3.1 4.25,43
- continuation 1.11 B.18
- CONTINUE statement B.16
- control characters 4.32,35
- conversion constants
- copies B.6
- count-controlled DO 1.13 4.11
- counting B.6
- CPU time 4.28
- curve-fitting 4.49,50
- CYCLE
  - named 4.20
  - statement 4.9 B.13
- D edit descriptor

- data abstraction 1.23
- data hiding 3.1
- DATA statement B.16,19
- data member 2.9
- data structure
  - defining 4.39
  - initializing 4.39
  - interpretation 4.40
  - nested 4.38
- data types 2.1
- Date class 3.5
- DEALLOCATE
  - statement 5.3 B.13
  - status 4.29
- deallocation
- debugging 1.19,20
- decimal exponent range
- decimal precision
- default
  - accessibility
  - character set
  - constructor 1.23
  - input unit 5.5
  - kind 4.3
  - output unit 5.5
  - precision
  - private accessibility
  - public accessibility
- deferred-shape array
- defined operation 4.31
- DELIM specifier B.16
- delimiter
- dereferencing 4.8
- derived class 7.1
- derived type
  - argument
  - component
  - constant
  - definition in a module
- destructor 3.2
- dimension
  - attribute
- DIMENSION statement 4.25 B.19
- direct access
  - READ statement
  - WRITE statement
- DIRECT specifier B.16
- DO
  - abort 4.10,20
  - construct 4.10
  - cycle
  - forever 4.10
  - loop 2.9
  - named 4.9,20
  - nested 4.19
  - termination
  - until 4.10,20
  - variable B.16
- DO statement 2.9 4.9 B.13
- DO WHILE statement 4.9,16,20 B.19
- documentation 1.21
- DOT\_PRODUCT intrinsic 1.14
- DOUBLE PRECISION attribute 2.1 4.3
- DOUBLE PRECISION statement B.16,17
- double precision 2.3
- doubly linked list 8.15
- Drill class 6.1
- dummy argument
- dummy array argument 5.3
- dummy pointer argument
- dynamic binding
- dynamic character 4.31
- dynamic data structure
- dynamically allocated array 5.3
- dynamically allocated memory
- E edit descriptor
- edit descriptor
  - A 4.34,35
  - B 4.36
  - BN
  - BZ
  - D B.20
  - E B.20
  - EN

- ES 3.4 B.20
- F B.20
- G B.20
- I 4.36,38 B.20
- L
- O 4.36 B.20
- P
- S
- SP
- SS
- T B.20
- TL B.20
- TR B.20
- X B.20
- Z 4.36 B.20
- / B.20
- :
- ELEMENTAL prefix
- ELSE IF statement 4.9,16
- ELSE statement 3.16 4.9
- ELSE WHERE statement 4.9
- embedded format
- Employee class 7.5,9,12,15
- EN edit descriptor
- encapsulation 3.1
- END DO statement 2.9
- END FUNCTION statement
- END IF statement 3.16 4.9
- END INTERFACE statement 4.37
- END MODULE statement 2.3
- END PROGRAM statement 2.3 4.2
- END SELECT statement
- END statement
- END SUBROUTINE statement
- end-of-file condition 4.29
- end-of-record condition 4.29
- end-of-transmission 4.32
- END= 4.30
- ENDFILE statement 4.29 B.20
- ENTRY statement B.13,20
- EOSHIFT intrinsic 5.11,14
- .EQ., see ==
- equality of two reals
- EQUIVALENCE statement B.16,20
- .EQV. 4.17
- EM specifier
- error
  - checking for
  - compilation
  - condition
  - execution
  - I/O
  - logical
  - semantic
  - syntactic
- ES edit descriptor
- exception 4.29,30
- exception descriptor
  - IOSTAT 4.13,29
  - STAT 4.29 5.3
- executable statement
- execution error
- existence B.7
- EXISTS specifier 4.30
- EXIT
  - named 4.20
  - statement 4.9,25
- explicit
  - interface 4.31
  - loops 4.9
- explicit-shape array
- exponent range
- exponential fit 4.50
- exponential format
- expression
  - arithmetic
  - constant
  - evaluation
  - in an output list
  - mixed-mode expression
- expressions 1.12 4.1
- extending an operator
- extent 5.1
- EXTERNAL attribute
- external file 4.13,37,47

- external procedure 4.47
- EXTERNAL statement B.13
- F edit descriptor B.20
- Fibonacci number
  - ADT 2.7
  - class 2.8
- file
  - access
  - connection
  - creation
  - disconnection
  - existence
  - external
  - inquiry
  - internal
  - position
- FILE specifier
- fill in B.7
- fixed source form
- floating-point numbers
- flow control 1.13 4.1,9
- FMT specifier
- FORALL construct
- FORM specifier
- format
  - embedded
  - list-directed
  - user input
- FORMAT statement 3.4 B.14
- formatted file
- formatted I/O statement
- formatted record
- FORMATTED specifier
- Fortran Character Set
- fraction 4.43,44
- free source form
- function
  - elemental
  - length
  - name
  - pure
  - reference
  - result
  - type
  - with no arguments
- function actual argument
- function dummy argument
- FUNCTION statement 2.9 4.22
- functions 1.6,15 4.22
- G edit descriptor
- Game of Life 1.4,9,16,20 4.23,25
- gather 5.12,14,15
- Gaussian elimination
- .GE., see >=
- generic
  - defined operator
  - function 3.2,4
  - identifier
  - interface 3.4,7,15
  - interface block
  - name 4.31
  - operator
  - procedure 4.31
- geometry module
- global variables 1.16 4.27
- Global Position class 6.7
- GO TO statement 4.9,16,19 B.14
- Great Arc class 6.7
- greatest common divisor 3.16 4.49
- .GT., see >
- hash table
- hexadecimal number 4.36
- host association
- host program unit
- host scoping unit
- I edit descriptor 4.36,38 B.20
- IF
  - construct 3.16 4.14
  - named 4.18
  - nested 4.14
- IF statement 3.5 4.15 B.14

IF ELSE	3.16 4.14	DEALLOCATE	5.3
imaginary part	4.8	DIGITS	
implicit declaration	4.3	DOT_PRODUCT	1.14 5.6,7,11
implicit interface	4.25	EOSHIFT	5.11,14
implicit loop	4.13,24 5.5	EPSILON	5.7
IMPLICIT NONE statement	4.3,28 B.14	EXP	4.8 5.7
IMPLICIT statement	4.3	FLOAT	
INCLUDE line	3.6	FLOOR	4.8 5.8
INDEX intrinsic	4.33,36	HUGE	4.7
index array		IACHAR	4.33,35
index bounds		IAND	4.29
infinite loop	4.10	IBCLR	4.29
information hiding	3.1	IBITS	4.29
inheritance	3.1,10 4.27 7.1	IBSET	4.29
initial statement		ICHAR	4.33
initial value		IEOR	4.29
initialization expression		IMAG	4.8
input device		INDEX	4.32
input editing		INT	5.7,8
input list		IOR	4.29
input record		ISHFT	4.29
input statement		ISHFTC	4.29
inquiry	B.7	KIND	2.3
list-directed	4.13	LBOUND	
input unit		LEN	4.33,36
I/O statement		LEN_TRIM	4.33
INQUIRE statement	4.30 B.14	LGE	4.33
inquire-by-file	4.30	LGT	4.33
inquire-by-output-list		LLE	4.33
inquire-by-unit	4.30	LLT	4.33
instance		LOG	4.8 5.7
INT intrinsic	5.7,8	LOG10	4.8 5.7
integer		LOGICAL	2.1
argument		MATXUL	5.5,6,7,11
constant		MAXLOC	4.24 5.7,11
division		MAXVAL	4.24 5.7,11
expression		MERGE	5.11
kind		MINLOC	4.24 5.7,11
literal constant		MINVAL	4.24 5.7,11
numbers		MOD	4.6,8
pointer	4.45	MODULO	3.16 4.8
INTEGER type	2.1,9	MVBITS	4.29
INTENT attribute	3.4 4.9,23	NINT	4.8 5.7,8
INTENT statement	B.14	NOT	4.29
interface	1.2,32 3.2 4.30	NULL	
INTERFACE ASSIGNMENT state-		PACK	5.11
ment	3.16 4.44	PRESENT	
B.14		PRODUCT	5.7,11
interface block	3.16 6.4	RANDOM_NUMBER	5.7
interface body	4.31	RANDOM_SEED	5.7
INTERFACE OPERATOR statement	3.16 4.44	REAL	5.7
5.12,13 B.14		REPEAT	4.33 5.11
INTERFACE statement	3.4 4.37 B.14	RESHAPE	5.4,7,11
internal file	4.35	SCAN	4.33
internal procedure	4.24,28	SELECTED_INT_KIND	2.1
internal variable		SELECTED_REAL_KIND	2.1,3
INTRINSIC attribute	B.22	SHAPE	5.7
intrinsic constructor	4.43	SIGN	4.24 5.7
intrinsic data type	4.4	SIN	1.13 4.8 5.7
intrinsic procedures and calls		SINH	4.8 5.7
ABS	4.8,24 5.7	SIZE	4.23 5.7,13
ACHAR	4.33,35	SPREAD	5.11
ACOS	4.8 5.7	SQRT	4.6,8 5.7
ADJUSTL		SUM	1.14 4.23 5.6,11
AIMAG	5.7	TAN	5.7
AINT	4.8 5.7,8	TANH	4.8 5.8
ALL	5.7,11	TINY	5.8
ALLOCATE	5.3	TRANSFER	4.29 5.11
ALLOCATED	5.3	TRANSPOSE	5.6,8,11
ANINT	5.7,8	TRIM	4.33
ANY	5.7,11	UBOUND	
ASSOCIATED	4.45,46	UNPACK	5.11
ASIN	4.8 5.7	VERIFY	4.33
ATAN	4.8 5.7	WHERE	4.25
ATAN2	1.15 4.8 5.7	INTRINSIC statement	B.14,22
BIT_SIZE	4.29	inverse of a matrix	4.48
BTEST	4.29	IOLength specifier	B.14
CEILING	4.8 5.7,8	IOSTAT specifier	B.13
CHAR	4.32	iteration count	
CMPLX	5.7	iterative methods	
CONJG	4.8 5.7	iterator	
COS	4.8 5.7		
COSH	4.8 5.7	keyword	
COUNT	5.7,11	argument	
CSHIFT	5.11,14		

KIND intrinsic 2.2,3  
 kind  
     default  
     inquiry  
     selector  
     type B.8,13  
 kind type parameter  
     of an expression  
  
 L edit descriptor  
 label 4.16  
 latitude 6.7  
 LBOUND intrinsic  
     .LE., see <=  
 leading blanks  
 least squares fit 4.49,52  
 LEN intrinsic 4.33  
 LEN\_TRIM intrinsic 4.34  
 length  
     of a character argument  
     of a character variable 4.35  
     specification  
 lexical comparison intrinsic 4.33  
 LGE intrinsic 4.33  
 LGT intrinsic 4.33  
 library function 1.19  
 line  
     continuation 1.11 B.18  
     maximum length of  
     multiple statements on 4.48  
 linked list  
     circular  
     double 8.15  
     pointer 4.45,47  
     single 8.10  
 list-directed  
     data value termination on input  
     format specifier  
     formatting  
     input  
     output  
     PRINT statement  
     READ statement  
 literal constant  
     array-valued  
 LLE intrinsic 4.33  
 LLT intrinsic 4.33  
 local variable  
 location in an array B.8  
 logical  
     expression 1.13  
     function  
     literal constant  
     value  
     variable  
 logical IF statement  
 LOGICAL intrinsic  
 logical operator 4.17  
 LOGICAL statement B.14  
 LOGICAL type 2.1 4.17 B.8  
 longitude 6.7  
 loop  
     abort 4.19  
     cycle 4.19  
     counter 4.10,11  
     implied 4.12,24  
     indexed 1.13 4.11,48  
     infinite 4.10  
     named 4.9  
     nested 4.12,13  
     post-test 4.10,20  
     pre-test 4.10,20  
     variable 1.13  
 loops 1.6,7,13 4.12  
 loss of precision  
 lower bound  
 lower case letters 4.36  
     .LT., see <  
  
 main program  
 maintainability  
 Manager class 3.1 7.5,8,10,12,13,15  
 mantissa  
 many-one array section  
 masked array assignment  
  
 masks 4.13 5.10,12,25 B.1,2,3,9  
 massively parallel computer  
 mathematical constants 2.3  
 Matlab 4.4,5,7,8,9,10,12,14  
     4.22,23,27 5.2,4,5,6,7  
     5.9,9,24,25  
 matrix  
     addition 5.18,24  
     column 5.11,16  
     diagonal 5.17  
     factorization 5.21  
     inverse 5.12,20,24  
     multiplication 5.12,19,24  
     operations 5.12  
     partition 5.17  
     row 5.16  
     shifts 5.14  
     square 5.16  
     symmetric 5.17  
     transpose 5.12,17  
 mean 4.23  
 memory  
     allocation  
     deallocation  
     leak 8.9  
 message  
 mixed kind expressions  
 mixed-mode expression  
 model number  
     bit  
     integer  
     real  
 modular design 1.5,6  
 modular program development 1.2  
 module 1.18 2.3 3.1 4.27,28  
 module procedure  
 MODULE PROCEDURE state-  
 ment 3.4,16 5.12,13 7.1  
 MODULE statement 2.9 7.1 B.14  
 module variable 2.3 4.28  
 multiple inheritance 7.1  
  
 name length  
 NAME specifier  
 named  
     DO construct 4.9  
     IF construct 4.18  
     CASE construct  
     SELECT construct 4.18  
 named constant  
 NAMED specifier  
 NAMELIST statement B.16,22  
     .NE., see /=  
 negative iteration count  
 negative subscript value  
     .NEQV. 4.17  
 nested  
     data structures 4.38  
     DO loops 4.13  
     IF blocks 4.14  
     implied loops 4.12  
     scoping unit  
 Newton-Raphson method 1.13,25  
 NEXTREC specifier  
 NML specifier B.23  
 node  
 non-advancing I/O 3.18  
 non-advancing READ statement  
 non-counting DO loop  
 non-default  
     character  
     character set  
     complex number  
     integer  
     kind  
     logical  
     real  
     .NOT. 4.17,49  
 NULL 4.46  
 null character 4.33  
 NULLIFY statement 4.46 B.14,22  
 NUMBER specifier  
 number B.9  
 numerical sorting

- O edit descriptor 4.36
- object 3.1
- object-oriented
  - analysis 1.21,23
  - design 1.21,24
  - languages 1.21,24
  - programming 1.1,21 3.1 6.1
- obsolescent statements 2.1,2 B.17
- octal number 4.36
- ONLY qualifier 4.27 7.1 B.25
- OPEN statement B.14,25
- OPENED specifier 4.30
- operator
  - definition 4.43
  - overloading 3.14 4.43
- operators
  - arithmetic
  - binary
  - concatenation
  - unary
- optional argument 2.9 3.7
- OPTIONAL attribute 2.9 3.7 4.30,31
- OPTIONAL statement B.14,22
- .OR. 3.7 4.49
- order of evaluation
  - effect of parentheses
- output
- output device
- output editing
- output format
- output list
  - expression in
- output statement
  - list-directed
- overflow
- overloading 3.14,16 4.43
- P edit descriptor
- PACK intrinsic procedure
- PAD specifier B.16
- padding 5.14
- parallel computers 3.19
- PARAMETER attribute 2.3
- PARAMETER statement B.14,22
- parameterized constants
- parameterized data types
- parameterized real variables
- parameterized variables
- parentheses to set order
- pass by reference 4.7,31
- pass by value 4.8,31
- PAUSE statement B.16,22
- peripheral device
- Person class 3.6 4.47
- physical constants
- pointer
  - allocation
  - array
  - assignment 4.45
  - association status
  - component of derived type
  - deallocation
  - dummy argument
  - input and output
  - inquiry B.10
  - linked list
  - nullification
  - return value 4.31
  - target 4.45
  - variable
- POINTER attribute 4.31,45
- POINTER statement B.14
- pointer-valued function
- polymorphism 3.1 7.1
- POSITION specifier
- post-condition
- power law fit 4.50
- precedence orders 4.5
- precision 2.2
- pre-condition 3.7 4.30
- PRESENT intrinsic
- PRINT statement
- PRIVATE attribute
- private components 2.6
- prive members 2.6

- PRIVATE qualifier
- PRIVATE statement 2.9 B.14
- procedure
  - argument
  - interface
  - intrinsic
  - libraries
  - size
- PRODUCT intrinsic
- Professor class 7.3
- program
  - design 1.3,11
  - errors
  - name
  - structure
  - testing 1.18
- PROGRAM statement 2.3 4.2 B.14
- program unit
  - external routine
  - function
  - internal routine
  - main
  - module
  - subroutine
- projectile 4.49
- prototype 1.6,23 4.30
- pseudocode 1.5,8,9,14,16 4.10
- PUBLIC attribute
- public attributes 2.6
- public member
- PUBLIC statement 2.9 B.14
- PURE prefix
- quadratic equation 1.8
- random access
- random number
- range
- range of a DO loop
- rank
- rational number
  - arithmetic 3.15
  - class 3.14
  - derived type 3.16
- READ specifier
- READ statement 2.4 4.12,29 B.14
- read-only file
- READWRITE specifier
- real
  - argument
  - arithmetic operation
  - constant
  - DO variable
  - exponential literal constant
  - expression
  - literal constant
  - number
  - part (of a complex number)
  - variable
- REAL intrinsic
- REAL statement 2.3 B.14
- REAL type 2.1
- REC specifier B.16
- RECL specifier B.16
- Rectangle class 3.2,3,4,19
- recursion
- recursive
  - algorithms 4.49
  - bisection method
  - data structure
  - function B.15,21,24
  - procedure
  - process
  - reference
  - subroutine B.15,24,25
- RECURSIVE qualifier 3.16 B.21,24
- reduction B.11
- relational expression
- relational operator 4.4
- rename modifier 7.1,4 B.25
- renaming of module entities
- repeat count
- repeatable edit descriptor
- repeated format
- RESHAPE intrinsic

reshaping an array B.11  
 restricting access to module  
 restrictions on a logical IF  
 restrictions on a DO loop  
 result length  
 RESULT specification 2.9 3.4 B.13  
 result variable  
 return from a procedure  
 RETURN statement 4.9 B.15,16  
 returns 1.15  
 reverse order B.11  
 REWIND statement 4.29 B.15,24  
 root  
 roots of a quadratic equation  
 round-off error  
 row extraction 5.9  
  
 S edit descriptor  
 SAVE attribute  
 SAVE statement B.15,24  
     in a module 4.27  
 scalar  
     conformable with an array  
 scalar product of two vectors  
 scalar variable  
 scale factor  
 scatter 5.12,14  
 scope  
 scoping unit  
 scratch file  
 SELECT CASE statement 4.9,17,18 B.15  
 SELECTED\_INT\_KIND intrinsic  
 SELECTED\_REAL\_KIND intrinsic  
 semantic error  
 SEQUENCE attribute B.24  
 sequential access  
 sequential file  
 sequential I/O statement  
 SEQUENTIAL specifier  
 shape  
 side effects 8.9  
 SHAPE intrinsic procedure  
 shifts B.11  
 simultaneous linear equations  
 singly linked list 8.10  
 size  
 SIZE intrinsic 5.13  
 SIZE specifier  
 solution of linear equations  
 sorting, bubble  
 source form  
     fixed form  
     free form  
 SP edit descriptor  
 space character  
 SPACING intrinsic B.17  
 sparse matrix  
 Sparse Vector class  
 specification  
 specification expression  
 specification statement  
 specifier  
     ACCESS B.16  
     ACTION  
     ADVANCE B.16,23  
     APPEND B.16  
     ASIS B.16  
     BLANK B.16  
     DELIM B.16  
     DIRECT B.16  
     END B.16,22  
     EOR B.16  
     ERR B.13  
     EXIST B.21  
     FILE B.21  
     FMT B.14,16,23  
     FORM B.16  
     FORMATTED B.16  
     IOLENGTH B.14,21  
     IOSTAT B.13,16,21  
     NAME B.14,21  
     NAMED  
     NEXTREC  
     NEW B.16  
     NML B.23,25  
     NONE B.16  
     NULL B.16  
     NUMBER  
     OLD B.16  
     OPENED B.21  
     PAD B.16  
     POSITION B.16  
     QUOTE B.16  
     READ B.16  
     READWRITE B.16  
     REC B.16,23,25  
     RECL B.16  
     REPLACE B.16  
     REWIND B.16  
     SEARCH B.16  
     SEQUENTIAL B.16  
     SIZE B.16  
     STAT B.13  
     STATUS B.16  
     UNFORMATTED B.16  
     UNIT B.14,16  
     UNKNOWN B.16  
     WRITE B.16  
     ZERO B.16  
 SS edit descriptor  
 statement entity  
 statement function B.16  
 statement label 4.16  
 statement order  
 statements 1.1,2,11  
 STATUS specifier  
 STOP statement 3.7 4.9 5.3 B.15  
 storage unit  
     character  
     numeric  
 string 2.1  
 strong typing 4.3,28  
 structure  
     components 4.38  
     constructor 4.43  
 structured programming 1.15  
 Student class 3.7,11  
 subprograms 4.21  
 subprogram interface 3.16 4.31 6.4  
 subroutine  
     actual argument  
     arguments  
     call  
     dummy argument  
     interface  
     library  
     name  
     result  
 SUBROUTINE statement 2.9 4.22 B.15  
 subscript 2.9 5.1  
 subscript expression  
 subscript triplet 4.7 5.2,8  
 substring  
 SUM intrinsic 1.14  
 swap 9.1  
 syntactic error 1.1,19  
 SYSTEM\_CLOCK call 4.28  
  
 tabs 4.50  
 T edit descriptor  
 target B.11  
 TARGET attribute 4.45  
 TARGET statement B.15  
 template 9.1  
 TL edit descriptor  
 top-down design  
 TR edit descriptor  
 TRANSPOSE intrinsic 5.4  
 TRIM intrinsic 4.33  
 trailing blanks  
 tree-structured data 4.45 8.1  
 truncation error  
 type conversion intrinsic  
 TYPE declaration statement 2.4,9  
 type parameter  
 TYPE statement 2.4,9 B.15  
  
 UBOUND intrinsic  
 unary operator  
 undefined array



```

undefined pointer status
underflow
unformatted
    file
    I/O statements
    record
UNFORMATTED specifier
unit number
unit specifier
until construct
upper bound
upper case letters 4.36
US Military Standard 4.29
USE association
USE statement 2.3 3.2,4,16 4.27
    7.1 B.15

variable
    character
    declaration
    initial value
    internal
    local
    name 4.2
variables 1.12 4.1
Vector class 5.25 B.12
vector subscript 5.12

WHERE construct 5.10 B.25
WHERE statement 3.4 4.9 5.9 B.14,25
while loop 4.20,222
WRITE specifier 3.16 B.14
WRITE statement

X edit descriptor

Z edit descriptor
zero-sized array

! comment
    continuation marker
    namelist data initiator
( )
    implied loop bounds
    subscript bounds
(/ /) array constructor 5.2
** exponentiation 4.8
+ overloaded 3.16
- overloaded
/
    edit descriptor
    list-directed data terminator
    namelist data terminator
    value separator
// concatenation 4.32
/=
    not equal
    overloaded
:
    edit descriptor
    subscript triplet
:: attribute terminator 2.3
; statement terminator
<
    less than
    overloaded
<=
    less than or equal to
    overloaded
=
    assignment
    overloaded 3.16
==
    equal to
    overloaded 3.16
>=
    greater than or equal to
    overloaded
=> rename option
>
    greater than
    overloaded
- character in a name

```

## Appendix G

# Program Index

add_Rational	3.16	Int_deg_min	6.11,13,16
Add_to_Q	8.5,7,8	Int_deg_min_sec	6.11,13,16
Angle_	6.11,12,16	invert	3.17,18
array_indexing	4.11	is_equal_to	3.17
assign	4.44	Is_Q_Empty	8.5,7,8
Change	4.32	Is_Q_Full	8.5,7,8
check_basis	D.12	is_Stack_Empty	8.3,4
circle_area	3.4	is_Stack_Full	8.3,4
class_Angle	6.12	is_S_L_empty	8.12,13
class_Circle	3.4,19	less_than_Object	8.14
class_Date	3.7,10,136.17	linear_fit	4.51
class_Drill	6.5	list	3.17
class_Employee	7.5,6,8,9,11,12 D.21	list	4.44
class_Fibonacci_Numbers	2.9	List_Angle	6.11,12
class_Global_Position	6.14	List_Great_Arc	6.10,15
class_Great_Arc	6.15 D.21	List_Positions	6.10,14,16
class_Manager	7.8,9,10,11,12,14	List_Position_Angle	6.11,13
class_Object	8.13,14,15	List_Pt_to_Pt	6.10,15,16
class_Person	3.9,10,12,13 6.17	logical_operators	4.17
class_Position_Angle	6.12 D.21	lsq_fit	4.52
class_Professor	7.3	make_Person	3.9,10,13
class_Queue	8.6	make_Professor	7.3
class_Rational	3.16,18	make_Rational	3.17
class_Rectangle	3.4,19	make_Rectangle	3.5
class_Sparse_Vector	D.14	make_Stack	8.3,4
class_Stack	8.3	make_Student	3.12
class_Student	3.12,13	Manager_	7.8,9,13
class_Vector	D.7	Math_constants	2.3
clip	4.24	maximum	4.26
clip_an_array	4.24	maxint	4.26
compare_strings	4.34	mean	4.23
Conversion_Constants	D.3	mult_Fraction	4.43,44
convert	3.16	mult_Rational	3.17,18
copy_Rational	3.16,18	next_generation	4.25
create_a_type	2.4	No_Change	4.32
Create_Q	8.5,7,8	Object	8.4,13,14,18
Date_	3.7	Ops_example	5.13
Decimal_min	6.11,12	out	6.4,5
Decimal_sec	6.11,12,16	passing_arguments	4.32
Default_Angle	6.11,12	pay	7.9,10
define_structures	4.42	payE	7.5,6,12
delete_Rational	3.16,18	payM	7.8,9,13
derived_class_name	7.1	Person_	3.9,10
destroy_D_L_List	8.16	Physical_Constants	D.4
doubly_linked_list	8.16	pop_from_Stack	8.3,4
Drill_	6.3,4,5,6	print	7.3
D_L_insert_before	8.17,18	PrintPay	7.11,12,14
D_L_new	8.16,18	PrintPayEmployee	7.11,12
equal_Fraction	4.44	PrintPayManager	7.11,13
equal_integer	3.15	print_Date	3.7
equal_to_Object	8.14	print_DOB	3.9
exception	4.30	print_DOD	3.9
exceptions	4.30	print_DOM	3.12
exception_status	4.30	print_D_L_list	8.17,18
Fibonacci	2.9	print_GPA	3.12,13
Fraction	4.44	print_name	3.9,10,13
Fractions	4.44	print_Nationality	3.9
game_of_life	4.25	print_sex	3.9
gcd	3.16 4.54	print_S_L_list	8.12,13
geometry	3.4	pt_expression	4.46
getEmployee	7.8,13	push_on_Stack	8.3,4
getName	7.9,12	Rational_	3.17,18
getNameE	7.5,6,12	readData	4.37 D.6
getNameM	7.8,9,13	read_Date	3.7
getRate	7.5,12	Read_Position_Angle	6.11,13
get_Arc	6.10,15	rectangle_area	3.4
Get_Capacity_of_Q	8.5,7,8	reduce	3.17
get_Denominator	3.16	relational_if	4.15
Get_Front_of_Q	8.5,7,8	Remove_from_Q	8.5,7,8
get_Latitude	6.10,14	setData	7.9,11
Get_Length_of_Q	8.5,7,8	setDataE	7.5,6,12,14
get_Longitude	6.10,14	setDataM	7.8,9,13,14
get_mr_rate	6.3,5,6	setSalaried	7.8,9,10,11,13,14
get_Numerator	3.17	set_Date	3.7
Get_Obj_at_Ptr	8.17,18	set_DOB	3.9,10,13
get_person	3.12,13	set_DOD	3.9,10
Get_Ptr_to_Obj	8.17,18	set_DOM	3.12,13
get_torque	6.3,5,6	set_Latitude	6.10,14
Global_Position_	6.10,14,16	set_Lat_and_Long_at	6.10,14,16
Great_Arc_	6.10,15,16	set_Longitude	6.10,14
hello_world	4.2,53	simple_arithmetic	4.6
in	6.4,5	simple_if_else	4.16
inputCount	4.37 D.6	simple_loop	4.11
Int_deg	6.11,12,16	singly_linked_list	8.11
		spy	4.25

```

string_to_integer  4.36
Student_          3.12
swap_chemical_element  9.2
swap_integer      9.1
swap_library      9.2
swap_objects      9.1
S_L_delete        8.11,13
S_L_insert        8.12,13
S_L_new           8.12,13
test_Arc          6.16
test_Drill        6.6
test_D_L_L        8.18
test_matrix       4.48
test_Professor    7.4
test_Queue        8.8
test_Stack        8.4
test_S_L_L        8.13
test_Vector       D.11
tic               4.28
tic_toc           4.28
toc               4.28
to_Decimal_Degrees 6.11,13,16
to_lower          4.36,37
to_Radians        6.11,13,16
to_upper          4.36,37 D.5
up_down           4.37
watch             D.5

```

# Index

- abstract data type, 15, 23, 27
- abstraction, 19, 27
- access, 36
- access restriction, 19
- accessibility, 19
- accessor, 18
- actual argument, 56
- Ada, 33
- addition, 56
- ADT, *see* abstract data type
- ADVANCE specifier, 42, 103
- agent, 18
- algorithm, 51
- ALLOCATABLE, 15
- allocatable array, 160, 161
- ALLOCATE, 15
- allocate, 42
- ALLOCATE statement, 75, 93
- ALLOCATED, 15
- allocation status, 75
- AND operand, 42
- area, 34
- argument
  - inout, 71
  - input, 71
  - interface, 76
  - none, 71
  - number of, 76
  - optional, 76, 77
  - order, 76
  - output, 71
  - rank, 76
  - returned value, 76
  - type, 76
- array, 26, 60, 67, 83
  - allocatable, 160
  - assumed shape, 77
  - automatic, 90, 160
  - Boolean, 168
  - constant, 160
  - dummy dimension, 160
  - flip, 170
  - mask, 168, 183
  - rank, 77, 159, 161, 170
  - rectangular, 170
  - reshape, 159
  - shape, 159
  - shift, 172
  - size, 159
  - unknown size, 77
  - variable rank, 160
- array operations, 163
- ASCII, 23
- ASCII character set, 77, 78, 99, 163
- assembly language, 15
- assignment operator, 10, 39
- ASSOCIATED, 15
- ASSOCIATED function, 76, 89
- ASSOCIATED intrinsic, 132, 134
- associative, 176, 177
- asterisk (\*), 58
- ATAN2, 13
- attribute, 105, 106, 109, 121, 125
  - private, 27, 125
  - public, 27
  - terminator, 25
- attribute terminator, 25
- attributes, 19, 27
- automatic array, 90, 160, 161
- automatic deallocation, 29
- BACKSPACE statement, 76
- bad style, 162
- base class, 121
- behavior, 106, 109
- binary file, 163
- bit
  - clear, 75
  - extract, 75
  - set, 75
  - shift, 75
  - test, 75
- bit manipulation, 75
- blanks
  - all, 78
  - leading, 78
  - trailing, 78
- Boolean, 53
- Boolean value, 23

- bottom-up, 4
- bounds, 159
- bubble sort, 93, 95
  - ordered, 96
- bug, 9
- C, 1, 33, 52
- C++, 1, 10, 14, 24, 33, 52, 58, 60, 77, 82, 103, 123
- CALL statement, 42
- CASE DEFAULT statement, 64
- CASE statement, 64
- cases, 62
- central processor unit, 73
- character, 82
  - case change, 81
  - control, 77
  - from number, 81
  - functions, 78
  - non-print, 103
  - non-printable, 77
  - strings, 77
  - to number, 81
- character set, 23
- CHARACTER type, 23, 26, 53
- chemical element, 25
- circuits, 170
- circular shift, 172
- class, 15, 19, 33
  - base, 18
  - Date, 120, 123
  - derived, 18
  - Drill, 105
  - Employee, 125
  - Geometric, 120
  - Global\_Position, 114
  - Great\_Arc, 114
  - hierarchy, 33
  - instance, 33
  - Manager, 125, 135
  - Person, 120, 123
  - polymorphic, 133
  - Position\_Angle, 109, 114
  - Professor, 123
  - Student, 120, 123
- class code
  - class\_Angle, 114
  - class\_Circle, 34
  - class\_Date, 37
  - class\_Fibonacci\_Number, 29
  - class\_Person, 37
  - class\_Rational, 42
  - class\_Rectangle, 34
  - class\_Student, 37
  - Drill, 106
  - Global\_Position, 114
  - Great\_Arc, 114
  - Position\_Angle, 114
- clipping function, 14, 71
- CLOSE statement, 75
- Coad/Yourdon method, 18
- colon operator, 56, 61, 62, 78, 160, 163, 167, 170
- column major order, 181
- column matrix, 174
- column order, 162
- comma, 99
- comment, 1, 2, 7, 9, 12, 52
- commutative, 101, 176, 177
- compiler, 10, 15, 91
- complex, 10, 82, 165
- COMPLEX type, 23, 53
- COMPLEX type, 24
- composition, 34, 36
- conditional, 7–9, 11, 51, 58
- conformable, 176
- connectivity, 170
- constant array, 160
- constructor, 18, 29, 34, 125, 134, 135
  - default, 18
  - intrinsic, 18, 26, 34, 39
  - manual, 36
  - public, 37
  - structure, 26
- CONTAINS statement, 29, 33, 34, 73, 76, 86
- continuation marker, 10
- control key, 79
- conversion factors, 29
- count-controlled DO, 12, 13
- CPU, *see* central processor unit
- curve fit, 91
- CYCLE statement, 66
- data abstraction, 19
- data hiding, 36
- data types, 10
  - intrinsic, 23
  - user defined, 23
- date, 101
- DEALLOCATE, 15
- deallocate, 18, 42
- DEALLOCATE statement, 75
- debugger, 17
- debugging, 16
- default case, 64
- default value, 29
- dereference, 58
- derived class, 121

- derived type, 15, 23
  - component, 83
  - nested, 83
  - print, 85
  - read, 85
- destructor, 29, 34, 41, 48
- determinant, 179
- diagonal matrix, 174
- dimension
  - constant, 161
  - extent, 159
  - lower bound, 159
  - upper bound, 159
- distributive, 177
- division, 56
- DO statement, 29, 60, 62
- DO WHILE statement, 67
- DO-EXIT pair, 68, 69
- documentation, 17
- domain, 19
- dot\_product, 12
- double, 24
- DOUBLE PRECISION type, 23, 24, 53
- dummy argument, 56, 73
- dummy dimension, 161
- dummy dimension array, 160
- dummy variable, 73
- dynamic binding, 18
- dynamic data structures, 38
- dynamic dispatching, 132
- dynamic memory, 75
  - allocation, 15
  - de-allocation, 15
  - management, 15
- dynamic memory management, 89
- e, 25
- EBCDIC, 23
- EBCDIC character set, 77
- Eiffel, 18
- electric drill, 105
- ELSE statement, 42, 63, 67
- encapsulate, 15
- encapsulation, 27, 33
- end off shift, 172
- end-of-file, 76
- end-of-record, 76
- end-of-transmission, 78
- EOF, *see* end-of-file
- EOR, *see* end-of-record
- EOT, *see* end of transmission
- equation
  - number, 173
- error checking, 18
- exception, 75
- exception handler, 75
- exception handling, 18
- exercises, 48
- EXIT statement, 66, 67
- explicit loop, 11
- exponent range, 24
- exponentiation, 56
- expression, 10, 51, 52, 89
- external subprogram, 77
- factorization, 178, 179
- FALSE result, 63
- Fibonacci number, 29
- file, 75
  - column count, 100
  - internal, 81
  - line count, 100
  - read status, 100
  - unit number, 101
- finite element, 43
- flip, 167, 170
- float, 53
- floating point, *see* real, 23, 24, 183
- flow control, 11, 51, 58
- FORMAT statement, 34
- function, 7, 9, 51, 69, 70
  - argument, 13, 15
  - extensible, 132
  - recursive, 42, 102
  - result, 70
  - return, 13
  - variable, 15
- function code
  - Add, 29
  - add\_Rational, 42
  - Angle\_, 114
  - circle\_area, 34
  - clip, 71
  - convert, 42
  - copy\_Rational, 42
  - Date\_, 37
  - Decimal\_min, 114
  - Decimal\_sec, 114
  - Default\_Angle, 114
  - Drill\_, 106
  - gcd, 42, 102
  - get\_Arc, 114
  - get\_Denominator, 42
  - get\_Latitude, 114
  - get\_Longitude, 114
  - get\_mr\_rate, 106
  - get\_next\_io\_unit, 103
  - Get\_Next\_Unit, 99

- get \_Numerator, 42
- get \_person, 37
- get \_torque, 106
- Great \_Arc \_, 114
- inputCount, 93
- Int \_deg, 114
- Int \_deg \_min, 114
- Int \_deg \_min \_sec, 114
- is \_equal \_to, 42
- make \_Person, 37
- make \_Rational, 42
- make \_Rectangle, 36
- make \_Student, 37
- mid \_value, 70
- mult \_Fraction, 87
- mult \_Rational, 42
- new \_Fibonacci \_Number, 29
- Person \_, 37
- Rational \_, 42
- rectangle \_area, 34
- set \_Date, 37
- set \_Lat \_and \_Long \_at, 114
- Student \_, 37
- toc, 73
- to \_Decimal \_Degrees, 114
- to \_lower, 81
- to \_Radians, 114
- to \_upper, 81, 101
- FUNCTION statement, 29
- Game of Life, 4
- Gamma, 25
- gather-scatter, 172
- gcd, *see* greatest common divisor
- generic function, 33, 34
- generic interface, 134
- generic name, 34
- generic object, 42
- generic routine, 123
- generic subprogram, 77
- geometric shape, 34
- global positioning satellite, 108
- global variable, 14, 73
- GO TO statement, 65, 66
- GPS, *see* global positioning satellite
- Graham method, 18
- graphical representation, 27, 120
- greatest common divisor, 42, 102
- Has-A, 109
- header file, 131
- hello world, 52, 101
- hierarchy
  - kind of, 18

- part of, 18
- horizontal tab, 78
- Hubbard, J.R., 36
- hyperbolic tangent, 103
- identity matrix, 182
- if, 12
- IF ELSE statement, 62
- IF statement, 29, 37, 42, 62
- if-else, 12
- IF-ELSE pair, 63
- IF-ELSEIF, 132
- IMPLICIT COMPLEX, 53
- IMPLICIT DOUBLE PRECISION, 53
- IMPLICIT INTEGER, 52
- implicit loop, 12
- IMPLICIT NONE, 26, 29
- IMPLICIT REAL, 52
- implied loop, 61, 62, 160, 170
- INCLUDE line, 37, 42, 90
- INDEX intrinsic, 81
- indexed loop, 11
- infinite loop, 9, 68, 69
- inheritance, 18, 33, 34, 73, 121
- inherited, 37
- inner loop, 62
- INQUIRE intrinsic, 93, 98, 103
- INQUIRE statement, 76
- instance, 33, 124
- integer, 10, 82, 165
- INTEGER type, 23, 24, 53
- intent
  - in, 29
  - inout, 29
  - statement, 29
- INTENT statement, 29, 58, 71, 94
- interface, 2, 6, 9, 13, 15, 27, 34, 76, 93, 106, 109, 123
  - general form, 77
  - human, 18
  - input/output, 18
  - prototype, 18
- INTERFACE ASSIGNMENT (=) block, 87
- interface block, 34, 77
- interface body, 77
- interface operator (\*), 39
- INTERFACE OPERATOR block, 86, 87
- INTERFACE OPERATOR statement, 170
- interface prototype, 105, 106, 125
- INTERFACE statement, 34
- internal file, 81
- internal sub-programs, 73
- interpreter, 10, 15
- intrinsic, 170



- intrinsic constructor, 86, 99, 108
- intrinsic function, 12, 70
- inverse, 182
- IOSTAT= variable, 75, 76
- Is-A, 108, 109
- ISO\_VARIABLE\_LENGTH\_STRING, 23
- Is\_A, 126
- keyword, 123
- KIND intrinsic, 24
- Kind-Of, 109, 125
- latitude, 108
- least squares, 91
- LEN intrinsic, 78, 81
- length
  - line, 52
  - name, 52
- LEN\_TRIM intrinsic, 78
- lexical operator, 95
- lexically
  - greater than, 78
  - less than, 78
  - less than or equal, 78
- library function, 16
- line continuation, 101
- linear equations, 177, 178
- linked list, 38, 88, 89
- linker, 16, 90
- list
  - doubly-linked, 89
  - singly-linked, 89
- logarithm, 70, 92
- logical, 82
  - AND, 63
  - equal to, 63
  - EQV, 63
  - greater than, 63
  - less than, 63
  - NEQV, 63
  - NOT, 63
  - operator, 63
  - OR, 63
- logical expression, 11
- logical mask, 62
- logical operator, 63
- LOGICAL type, 23, 42
- long, 24
- long double, 24
- long int, 24
- longitude, 108
- loop, 5, 7–9, 11, 51, 58, 183
  - abort, 68
  - breakout, 66
  - counter, 60
  - cycle, 66
  - exit, 60, 66
  - explicit, 59
  - implied, 61
  - index, 101
  - infinite, 60, 68
  - nested, 62, 66
  - pseudocode, 59
  - skip, 66
  - until, 67, 68
  - variable, 60
  - while, 67
- loop control, 61, 162
- loop index, 101
- loop variable, 11
- lower triangle, 175, 178
- manual constructor, 86, 106
- manual page, 17
- mask, 165, 168, 169, 183
- masks, 62
- Mathematica, 51
- mathematical constants, 25
- Matlab, 1, 10, 14, 52, 61, 70, 100, 103
- MATMUL intrinsic, 177
- matrix, 159, 174
  - addition, 176
  - algebra, 159
  - column, 174
  - compatible, 176
  - determinant, 179
  - diagonal, 174
  - factorization, 178
  - flip, 167
  - identity, 178
  - inverse, 90, 178
  - multiplication, 163, 176
  - non-singular, 178
  - null, 174
  - skew symmetric, 175
  - solve, 90
  - square, 174, 175
  - symmetric, 175
  - Toeplitz, 175
  - transpose, 163, 175
  - triangular, 175, 178
- matrix addition, 181, 182
- matrix algebra, 159, 176
- matrix multiplication, 169, 177, 182
- matrix operator, 38
- matrix transpose, 169
- maximum values, 71
- MAXLOC intrinsic, 71

- MAXVAL intrinsic, 71
- mean, 70
- member, 121
- message, 27
- methods, 3
  - private, 27
  - public, 27
- military standards, 75
- minimum values, 71
- MINLOC intrinsic, 71
- MINVAL intrinsic, 71
- modular design, 6
- module, 15, 25, 33, 69
- module code
  - class \_Angle, 114
  - class \_Circle, 34
  - class \_Date, 37
  - class \_Fibonacci \_Number, 29
  - class \_Global \_Position, 114
  - class \_Great \_Arc, 114
  - class \_Person, 37
  - class \_Position \_Angle, 114
  - class \_Rational, 42
  - class \_Rectangle, 34
  - class \_Student, 37
  - exceptions, 76
  - Fractions, 87
  - Math \_Constants, 25
  - record \_Module, 97
  - tic \_toc, 73, 101
- MODULE PROCEDURE statement, 34, 39, 86, 87, 170
- MODULE statement, 29
- module variable, 29
- modulo function, 56
- multiple inheritanc, 121
- multiplication, 56
- Myer, B., 18
- NAG, *see* National Algorithms Group
- named
  - CYCLE, 66, 67
  - DO, 67
  - DO loop, 60
  - EXIT, 66, 67
  - IF, 64
  - SELECT CASE, 64
- National Algorithms Group, 91
- nested
  - DO, 67
  - IF, 62
- new line, 79, 103
- Newton-Raphson method, 11
- non-advancing I/O, 42
- NULL function (f95), 89
- NULLIFY, 15
- nullify, 134
- NULLIFY statement, 89
- number
  - bit width, 24
  - common range, 24
  - label, 60
  - significant digits, 24
  - truncating, 166
  - type, 24
- numeric types, 23
- numerical computation, 38
- object, 15, 19, 33
- object oriented
  - analysis, 18, 43, 105, 109, 120
  - approach, 18
  - design, 18, 43, 105, 109, 120
  - language, 18
  - programming, 18, 105
  - representation, 18
- Object Pascal, 18
- OOA, *see* object oriented analysis
- OOD, *see* object oriented design
- OOP, *see* object oriented programming
- OPEN statement, 75, 163
- operator, 27
  - .op., 87, 169
  - .solve., 90, 91
  - .t., 170
  - .x., 170
  - assignment, 39
  - binary, 87
  - defined, 18, 87
  - extended, 87
  - overloaded, 18
  - overloading, 39, 86
  - symbol, 87
  - unary, 87
  - user defined, 77, 169
- operator overloading, 10
- operator precedence, 52
- operator symbol, 169
- optional argument, 29, 37, 76
- OPTIONAL attribute, 29, 36, 106
- OR operand, 37
- ordering array, 96
- outer loop, 62
- overloaded member, 123
- overloading, 39, 48, 86
  - testing, 87
- package, 15

- parallel computer, 43
- PARAMETER attribute, 25
- Part-Of, 109
- partial derivative, 180
- partitioned matrix, 175
- pass by reference, 57, 77, 88
- pass by value, 57, 58, 77
- path name, 37
- pi, 25
- pointer, 10, 23, 76, 87
  - allocatable, 15
  - arithmetic, 88
  - assignment, 89
  - association, 88
  - declaration, 88
  - dereference, 58
  - detrimental effect, 88
  - in expression, 89
  - inquiry, 89
  - nullify, 89
  - status, 15, 88
  - target, 88
- pointer object, 133
- pointer variable, 87
- polymorphic class, 133
- polymorphic interface, 120
- polymorphism, 18, 33, 34, 121, 126
- portability, 15
- pre-processor, 131
- precedence rules, 11
- precision, 183
  - double, 82
  - kind, 24
  - portable, 82
  - single, 82
  - specified, 82
  - underscore, 24
  - user defined, 24
- precision kind, 24
- PRESENT function, 76
- PRESENT intrinsic, 29, 36
- PRINT \* statement, 29
- private, 33, 106
- PRIVATE attribute, 29, 36
- private attributes, 37
- PRIVATE statement, 27
- procedural programming, 18
- procedure, 69
- program
  - documentation, 17
  - executable, 17
  - scope, 14
- program code, 114
- array \_ indexing, 60
- clip \_ an \_ array, 71
- create \_ a \_ type, 26
- declare \_ interface, 77
- Fibonacci, 29
- geometry, 34
- if \_ else \_ logic, 63
- linear \_ fit, 93
- Logical \_ operators, 63
- main, 37, 42
- operate \_ on \_ strings, 79
- relational \_ operators, 63
- simple \_ loop, 60
- string \_ to \_ numbers, 81
- structure \_ components, 85
- test \_ bubble, 98
- test \_ Drill, 108
- test \_ Fractions, 87
- test \_ Great \_ Arc, 114
- program keyword, 56
- PROGRAM statement, 26, 29
- projectile, 102
- prototype, 6, 76
- pseudo-pointer, 96
- pseudocode, 5, 14, 51, 71, 102
- public, 33, 125
- PUBLIC attribute, 29
- public constructor, 37
- public method, 27
- PUBLIC statement, 27
- quadratic equation, 3
- queue, 89
- rank, 161
- rational number, 38, 39
- read error, 103
- READ statement, 29, 62, 76
- real, 10, 82, 165
- REAL type, 23, 24, 53
- recursive algorithm, 88
- RECURSIVE qualifier, 42, 102
- reference, 10
- relational operator, 52, 63, 78
- remainder, 56
- rename modifier, 121
- reshape, 162
- RESULT option, 29
- result value, 70
- return, 161
- RETURN statement, 66
- REWIND statement, 76
- sample data, 99

- scatter, 173
- scope, 14
- SELECT CASE statement1, 64
- SELECTED \_INT \_KIND, 23, 24
- SELECTED \_REAL \_KIND, 23, 24
- selector symbol, 26, 29, 34
- server, 18
- short, 24
- size, 12
- SIZE intrinsic, 70, 90, 93, 159
- Smalltalk, 18
- sort, 87, 91, 93, 96, 127
  - bubble, 93
  - characters, 95
  - object, 97
  - objects, 95
  - strings, 95
- sorting, 42
- sparse vector, 49
- sparse vector class, 183
- specification, 4
- SQRT intrinsic, 27
- square root, 27, 56, 70
- stack, 89
- STAT = variable, 75
- statement, 2, 9
- statement block, 12, 58
- statements, 1
- status
  - FILE, 76
  - IOSTAT =, 76
  - MODE, 76
  - OPENED =, 76
- status checking, 161
- STOP statement, 37
- storage
  - column wise, 159
  - row wise, 159
- string, 23, 56
  - adjust, 78
  - case change, 81
  - character number, 78
  - collating sets, 78
  - colon operator, 78
  - concatenate, 78
  - copy, 78
  - dynamic length, 77
  - from number, 81
  - functions, 78
  - length, 78
  - logic, 78
  - repeat, 78
  - scan, 78

- to number, 81
- trim, 78
- verify, 78
- strings, 77
- strong typing, 53
- struct, 53
- structure, 23, 25, 33, 85
- structure constructor, 26
- structured programming, 13
- submatrix, 175
- subprogram, 69
  - recursive, 102
- subroutine, 69, 70
- subroutine code, 114
  - assign, 87
  - Change, 77
  - delete \_Rational, 42
  - equal \_Fraction, 87
  - equal \_Integer, 42
  - exception \_status, 76
  - in, 106
  - Integer \_Sort, 96, 99
  - invert, 42
  - list, 42
  - List \_Angle, 114
  - List \_Great \_Arc, 114
  - List \_Position, 114
  - List \_Position \_Angle, 114
  - List \_Pt \_to \_Pt, 114
  - lsq \_fit, 93
  - mult \_Fraction, 87
  - No \_Change, 77
  - out, 106
  - Print, 29
  - print \_Date, 37
  - print \_DOB, 37
  - print \_DOD, 37
  - print \_DOM, 37
  - print \_GPA, 37
  - print \_Name, 37
  - print \_Sex, 37
  - readData, 93, 101
  - read \_Date, 37
  - Read \_Position \_Angle, 114
  - reduce, 42
  - set \_DOB, 37
  - set \_DOD, 37
  - set \_DOM, 37
  - set \_Latitude, 114
  - set \_Longitude, 114
  - simple \_arithmetic, 56
  - Sort \_Reals, 94
  - Sort \_String, 95

- String \_ Sort, 99
- test \_ matrix, 90
- tic, 73
- SUBROUTINE statement, 29
- subroutines, 33
- subscrip, 159
- subscript, 26, 60
  - bounds, 159
  - range, 181
  - vector, 170
- subtraction, 56
- subtype, 133
- subtyping, 126, 132
- sum, 12
- SUM intrinsic, 70, 93, 169
- super class, 121
- syntactic error, 17
- SYSTEM\_CLOCK intrinsic, 73
- tab, 79, 99, 103
- TARGET, 15
- target, 23, 76, 88, 89
- template, 43, 126, 128
- tensor, 159
- testing, 15
- time of day, 101
- Toeplitz matrix, 175
- top-down, 4
- transformational functions, 169
- transpose, 163, 175, 177
- TRANSPOSE intrinsic, 170
- tree structure, 38, 88, 89
- triplet, *see* colon operator
- true, 12
- TRUE result, 63
- truss, 170
- type
  - conversion, 81
  - default, 52
  - implicit, 52
- TYPE declaration, 26, 29
- TYPE statement, 27, 34
- unexpected result, 169
- upper triangle, 175, 178
- USE association, 121, 125
- USE statement, 29, 33, 34, 37, 86, 90
- user defined operator, 169
- user interface, 2
- validation, 29
- variable, 8, 10, 23, 51
  - global, 14
  - name, 10

- type, 10
- variable rank array, 160
- vector, 159
- vector class, 48, 183
- vector subscript, 62, 170
- volume, 48
- WHERE construct, 169
- WHERE statement, 62, 67, 169
- while-true, 68
- wildcard, 128
- WRITE statement, 34, 62, 76