What Does PL/I Offer the Modern Programmer?

The answer to this question is, perhaps, contingent upon how we define the "Modern Programmer". For the purpose of this discussion, I will assume that our Modern Programmer is one who:

> Programs professionally, either as a corporate service or for/as a software vendor, and is, therefore, required to produce reliable ("industrial strength") code.
>
> Would like to minimise the number of programming languages required to develop any and all applications.
>
> Requires optimal, or near-optimal, execution speed of the finished program.
>
> Desires portability to multiple platforms, with consistent semantics for any given syntax.
>
> Requires the ability to create modular pieces of code that can be used and reused in multiple applications.

Given these criteria, the range of programming languages is reduced quite significantly on smaller platforms. In particular, the third criterion eliminates the interpretive and semi-interpretive languages from computationally intensive applications.

PL/I satisfies all of the above criteria:

> its mainframe background has generated a great deal of experience in the development of reliable compilers;
> its huge grammar makes it applicable to a wide variety of programming tasks;
> its grammar also makes its code amenable to heavy optimisation;
> there exist compilers for a multitude of platforms;
> its structured style makes modularity easy.

In addition to these, what makes PL/I so worthwhile?

The plenitude of virtues and dearth of vices make PL/I a favourite of nearly all who have used it extensively. I shall list the facilities and attributes of the language that I consider to be among the strongest attractants for programmers who have experience in other block-structured languages (i.e. those descended from ALGOL 60, such as ALGOL 68, Pascal, Oberon, Modula-2, and even C/C++).

Note that in the following list, words in all capital letters are lexical elements of the PL/I language and can be used to look up language constructs in a reference manual.

A rich macro language. The macro language of PL/I contains constructs for text substitution of source code, conditional compilation (%IF/%ELSE), case selection (%SELECT) and looping (%DO/%END); these provide comprehensive programmatic control over the compiler's processing of the source code.

What Does PL/I Offer the Modern Programmer?

Encapsulation. The language's ability to encapsulate data entities with code in "hermetically sealed" modules is unsurpassed. This modularity is, of course, one of the bases of Object Oriented programming, and is (to many practising programmers) the most significant basis.

Polymorphism. The polymorphic capabilities of PL/I are limited to binding instances to methods at compile-and-link time, using macro substitution and/or declaration of GENERIC names; there is no run-time binding. However, in the vast bulk of applications early binding is completely adequate, and there is the added benefit of no run-time penalty of resolving a polymorphic reference from some attribute-matching table.

Consistently structured and coherent syntax. There is a general style of PL/I statement construction that is used throughout the language. It is clear and English-like (if that is not a contradiction in itself); it permits great clarity of text in the source code and largely abominates terse or arcane code. That is not to say that one cannot write terse or arcane code; it is just so easy to write clear code that one can easily distinguish between "a bowl of spaghetti" and "a row of fish fingers" after one has seen well-written PL/I.

Extensible syntax. The PL/I macro preprocessor allows the programmer to define new statements to the language. This facility provides further encapsulation of code, since a substantial number of existing statements can be encapsulated into a "new" statement that has its own syntax. There is also a provision for generating compile-time error messages when the new syntax is transgressed.

Exception handling. The PL/I language defines a number of built-in CONDITIONs that represent run-time exceptions, and it allows the programmer to code "units" to handle any exception that occurs. Some of these CONDITIONs are very catchall in nature (e.g. ERROR and FINISH) and so a single ON-unit can catch multiple types of run-time exception. There are BUILTIN functions to determine which type of exception has been SIGNALled to trigger the ON-unit.

Furthermore, the programmer can define new CONDITIONs specific to the application; these can be used to throw and catch exception events peculiar to the given application.

In addition, each CONDITION has its own LIFO stack of ON-units, so when a lower level procedure overrides its caller's condition handling, the caller's overridden ON unit(s) is restored when that called procedure returns. This ensures that an application does not "break" because a subroutine disabled the main program's exception handler(s).

Pointers, pointers, pointers. In PL/I it is not necessary to restrict a pointer to pointing to a given type of data entity or aggregate. A pointer is simply an address of a location in memory; achieving valid access to that location is the programmer's responsibility - as it should be! By declaring multiple data items to be BASED on the same pointer, the programmer can use self-defining data structures that can share (or be considered to share) the same location in memory.

The language also has a portmanteau data type called an AREA, and specialised pointers, within an AREA, called OFFSETs. This allows independent data entities to be stored in contiguous (or nearly contiguous) storage and written to or read from auxiliary storage in a single transfer. If the OFFSET variables associated with a given AREA are also written when the AREA variable is written, their contents are valid if they are read back when the AREA is read.

Bits and bytes. The concepts of bit strings and character strings are native to the language; they are not derived from pointers or arrays or, worst of all, completely omitted. As a result, PL/I is quite replete with BUILTIN functions to manipulate bit strings and character strings, and the common operations of assignment [duplication], comparison and concatenation [also AND, OR and NOT for bit strings] are performed by simple operators. Also, bit strings of length 1 are used for Boolean values.

The library of BUILTINs includes functions to determine maximum and current length of strings (bit or character), extract substrings (bit or character), translate from one character set to another, scan for embedded strings (bit or character), scan for delimiters, scan for non-delimiters, generate repetitions - virtually everything you would ever want to do with a string. There are scan functions for both left-to-right and right-to-left scanning. The language definition permits the simpler ones to be expanded as in-line code, rather than incur the overhead of external calls.

Arrays as they were meant to be. Actually, APL implements arrays as they were meant to be, but PL/I comes very close. The array types are not derived from pointers, but instead are native to the language. A programmer can declare an array of any type of variable, including already aggregated variables and types, such as structures. The number of dimensions is limited only by the compiler implementation.

Array bounds can be any expression containing values that are "known" at the time the array is allocated. [See next bullet about memory allocation.] This permits arrays with variable bounds to be allocated on the stack instead of the heap. Good-bye memory leaks! It also means that subroutines do not have to over-allocate local array variables, to accommodate maximal parameters, during calls when smaller parameters are being passed.

The syntax also permits whole cross-sections, or whole arrays, to be referenced as single entities by using the asterisk as a subscript; it means "for every valid value of this subscript". In addition to being more succinct, it also ensures that the bounds are not transgressed and allows the compiler to generate more heavily optimised code for that reference. The BUILTIN library also includes functions (HBOUND, LBOUND and DIM) to inquire upon the bounds of an array's subscript(s). Since arrays are, by default, passed to subroutines and functions using a locator/descriptor that includes their bounds information, there is no need to pass array bounds to subroutines as separate parameters. This has two benefits: it shortens the calling sequence of the subroutine; and it eliminates the possibility of the caller passing the bounds in the wrong positions in the parameter list. Indeed, it further encapsulates the access

of the array within the subroutine.

Multiple classes of storage allocation. Like most other high-level languages, PL/I uses three areas for storage allocation: static, stack and heap. The methods for static allocation are exactly like those of FORTRAN or COBOL - the storage is reserved at compile time and the amount required must be known at compile time.

For stack allocation, PL/I differs from many other languages (except ALGOL 60 and ALGOL 68) in that the size of the data entities to be allocated in the stack frame do not need to be known at compile time; they can be calculated in the block prologue at run time. This permits adjustable array bounds, string lengths and area sizes in nested blocks and called subroutines; only the outermost main program, in which no variables are defined during its prologue, is limited to a compile-time-determined stack frame.

Heap allocation permits two subtly different approaches. These are CONTROLLED and BASED variables. The BASED approach is exactly analogous to C's **malloc()** function or C++'s **new** operator; a pointer is declared, the amount of storage required to map a structure is allocated from the heap and its address stored in the pointer - the amount can be calculated at run time, just as in C/C++. However, BASED structures in PL/I may contains "self-defining" fields, which are arithmetic fields that contain bounds or length values of other entities embedded in the structure. These allow the BUILTIN functions for determining array bounds, string lengths (current or maximum) and storage requirements that were not determinable at compile time, to execute accurately on arrays, strings and substructures embedded within a BASED structure.

The other class of variable allocated on the heap is called CONTROLLED. Each CONTROLLED variable is actually a LIFO stack in its own right; every time an ALLOCATE statement is executed a new generation is pushed onto the stack and every FREE statement pops off the current generation and reinstates the previous generation. There is a BUILTIN function to determine the number of generations currently on a CONTROLLED variable's stack. In a multi-threaded environment, different threads can be accessing different generations of a CONTROLLED variable concurrently. This also means that the programmer must ensure that a given generation of the variable remains allocated while ever the other threads need it.

Access methods. Although the concept of an "access method" is foreign to many programmers who have never worked on mainframes, it is one of great value on any platform.

An access method is a system-supplied, language-neutral mechanism for the storage and retrieval of records under program control. This differs from a database management system, where the storage and retrieval is under DBMS control. It also differs from hard-coded seek/read and seek/write methods in that the auxiliary storage addressing mechanism is supplied by system software rather than application software. This relieves the programmer from having to devise such a mechanism and it permits multiple programmers to access the same file(s) without having to thrash out a convention for storage of records.

Because of PL/I's mainframe background, it permits files to be declared with access method attributes. These are defined as part of the PL/I grammar, and a specific implementation of PL/I should have all the access methods defined for that platform as part of its grammar. For example, PL/I for OS/2 carries with it [almost] all of the mainframe access methods and its run-time library implements them under OS/2.

High precision arithmetic. Rounding errors are strongly despised in financial applications (they aren't popular in technical/scientific applications either), so floating-point arithmetic is not usable for truly "industrial strength" financial applications. Since unsigned binary integers overflow at just over 4 billion, this approach is too limited for larger values. PL/I offers packed decimal (or Binary Coded Decimal - BCD) as a native feature of the language. This provides 31-digit arithmetic without rounding errors (except when dividing - but such errors are unavoidable), thus permitting both the dynamic range and exact precision required for financial applications.

Mathematics. The library of BUILTIN functions for mathematical work is very comprehensive. The bulk of them are polymorphic, so one function name supports multiple levels of precision. Combined with the DEFAULT statement, this allows programs to be switched to/from single-precision to/from double-precision to/from extended-precision with ease.

Endian-independence. PL/I incorporates an attribute to allow mixed-endian programs, and client/server applications on different-endian platforms to exchange data. By declaring a variable to be NONNATIVE, the compiler will generate code to perform endian switching. The determination of whether NATIVE is big-endian or little-endian and, obviously, NON-NATIVE the reverse, can be done in a macro that enquires upon a compiler parameter. This allows one set of source to be compiled on both big-endian and little-endian platforms with correct results, including data interchange across platforms.

The foregoing list is not presented in any particular order of importance, since each programmer's priorities will differ. It is also not meant to be totally exhaustive, merely the non-trivial aspects of the language that are of significance to the author.

Although analogies were drawn with other languages, they were not really intended as competitive comparisons. This list is not intended to be "PL/I v. C/C++" or "PL/I v. Language X", but merely a statement of some of the virtues of PL/I.

Glossary of colloquial terms

bowl of spaghetti Jumbled, incoherent source code.

row of fish fingers Well-structured, coherent source code.