The disciplines of structured programming and programming for virtual storage are examined to show how they affect each other.

Considerations and techniques are proposed that, when applied during the process of structured design and coding, produce programs that place fewer demands on the computer storage resources.

The techniques are illustrated by example programs.

Structured programming for virtual storage systems

by J. G. Rogers

The design and coding of computer programs have been affected in recent years by two widely differing events. The introduction of virtual storage operating systems has reshaped our thinking about storage size and occupancy. At the same time, the discipline generally known as *structured programming* has caused great change in the process of design and coding. A body of literature has grown in both these areas, but little has been written that relates the two. This paper compares virtual storage systems and structured programming, and shows the constraints that each places on the other.

The concepts involved in structured programming began emerging in the late 1960s, when it became apparent that the complexity of very large programs was causing increasing problems in programming and maintenance. Structured programming has been very successful in reducing that complexity. Structured programming, however, has developed largely without regard to virtual storage. This has been primarily because virtual storage operating systems were not in general use during much of the development of the structured programming concept and because the concepts of structured programming transcend particular operating systems.

Implementations of virtual storage have proved not to be transparent to programs running in such systems. Because of this, techniques have evolved for coping with this lack of transparency. These techniques have developed without regard to structured programming and some suggestions that are quite alien to good structured programming practices have been made.

The orientation of this paper is toward preserving the concepts of structured programming. The paper also shows that, in most cases, the concepts of structured programming are quite suitable to the virtual storage environment. Presented first are brief descriptions of structured programming and virtual storage concepts, which are followed by a discussion of ways in which each affects the other.

Structured programming

The discipline of structured programming has evolved as a means of increasing program reliability and reducing programming and maintenance costs by making programs much less complex and thus more understandable. Some of the work in this field has been concerned primarily with program design,[1-5] whereas other work is more general.[6-9]

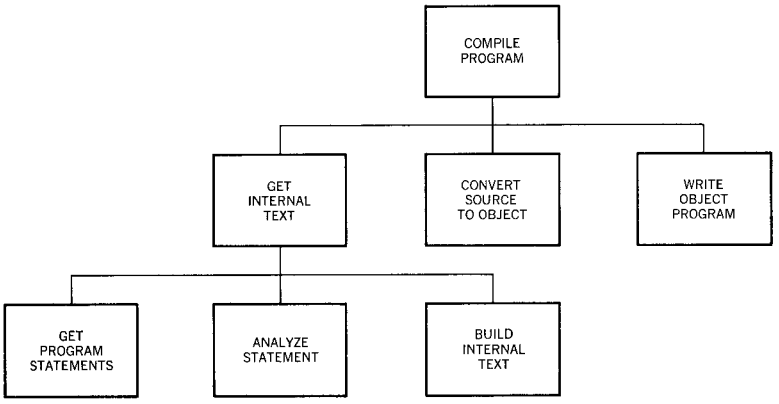
structured program design

The primary goal in the design of structured programs is to produce modular program structure through successive functional decompositions. An example of this might be for a compiler to create a top-level module whose function is to compile the program, as shown in Figure 1. The first level of decomposition might result in the following three modules: GET INTERNAL TEXT, CONVERT SOURCE TO OBJECT, and WRITE OBJECT PROGRAM. The top-level module, COMPILE PROGRAM, calls the other three modules. A further decomposition of the GET INTERNAL TEXT module might result in the following modules: GET PROGRAM STATEMENT, ANALYZE STATEMENT, and BUILD INTERNAL TEXT.

A *module* in the context of Figure 1 is a compilable unit of source code (e.g., a PL/I external procedure or a FORTRAN subprogram). Modules created in this manner should be made highly independent of each other, which can be done by minimizing the relationships among modules (called *module coupling*), and by maximizing the relationships among parts of each module (called *module strength*).[4]

The measure of the relationships among the parts of a module is module strength. Module strength is maximized if the parts of the module join forces to perform a single specific well-defined function.[4] In this context, if a module calls another module, the

Figure 1 Modular structure by successive decomposition of function



function of the called module is considered part of the calling module's function. One test of a module's strength is to attempt to write a sentence about the module's function. The module in Figure 1 called ANALYZE STATEMENT is a strong module because it can be described in a simple sentence with one verb and no words such as "if," "then," "next," "first," or "after."

The measure of the relationships among modules is module coupling. In this sense, a relationship exists between two modules if they refer to the same data. Some of the more common techniques used for referencing data among modules are externally declared data (such as data with an EXTERNAL attribute in PL/I programs), global data areas (for example, control blocks or data in a FORTRAN COMMON area), and parameters or arguments.[4] Externally declared data and global data areas create high coupling, which means that the modification of one module may require the modification of many more modules. To minimize coupling, data should be explicitly passed as parameters or arguments.[4]

The nature of the data passed between modules should also be considered. The data are either control information—in which the calling module tells the called module what to do—or pure data. Control information should be avoided because it increases coupling.[4] If a module tells another what to do, the second module is not considered to be a "black box" module. The classification of data used in this paper depends on how the sending module sees the data.

Some other characteristics that should be considered are decision structure, module size, and predictability. Whenever possible, it is desirable to arrange modules in such a way that modules directly affected by a decision are located beneath the

module that contains the decision step.[4] Very small modules tend to create excessive overhead, whereas very large modules may be difficult to understand. Experience indicates that modules of between ten and one hundred statements are the optimum size.[3] A module is said to be *predictable* if, for each time it is called with the same input, it produces the same output. In other words, the module does not "remember" what it has previously done and thus perform a slightly different function on subsequent calls. Such a module is independent of its environment, and is more likely to be usable in several contexts.

structured program writing

The programming phase of structured programming is concerned with reducing proposed modules to source language statements. The primary concept involved in structured programming is the use of structures such as DO-WHILE and IF-THEN-ELSE to control a program's flow rather than using GOTO statements. Another concept is the possible decomposition of modules into *segments*.[10]

The process of segmentation is similar in many ways to the decomposition of a program into modules, each of which performs a single function. A module is segmented by decomposing its function into more rudimentary subfunctions, each of which becomes a segment. Each segment is a separate entity which exists in external storage and is included in the module in a preprocessor phase of the compiler. Just as there are many levels of modules, each of which calls others at deeper levels, there are typically many levels of segments, each of which contains INCLUDE statements for other segments.

Each segment created consists of source language statements and perhaps INCLUDE statements for other segments. The source language statements in a segment define the control structure of the segment and perform part of the segment's function. Each INCLUDE statement for a segment represents some distinct function whose source statements are to be included later.

The concept of segmentation may be seen in Example 1, a PL/I program fragment:

Example 1

```
IF COMMAND = 'START' THEN
     DO;
          %INCLUDE STOPCMND:
     END;
ELSE
     IF COMMAND = 'STOP' THEN
          DO;
               %INCLUDE STRTCMND;
          END;
```

In Example 1, the programmer has written the control structure in the current segment. Because the intervening processing code has been left to other segments, the control structure is easily visible and understandable. The segments STRTCMND and STOPCMND are to be coded later. These segments are expected to be more easily understood because they are small independent pieces of code, each of which performs a single function.

Mills[9] describes the programming process as follows:

". . . one can write the first segment which serves as a skeleton for the whole program, using segment names, where appropriate, to refer to code that will be written later. In fact, by simply taking the precaution of inserting dummy members into a library with those segment names, one can compile or assemble, and even possibly execute this skeleton program while the remaining coding is continued.

"Now the segments at the next level can be written in the same way, referring, as appropriate, to segments to be later written. . . . As each dummy segment becomes filled in with its code in the library, the recompilation of the segment that includes it will automatically produce new updated expanded versions of the developing program."[11]

There are several characteristics that segments should have in a structured program. The segment should be small enough to be listed on a single page, i.e., have no more than fifty statements. Each segment should represent a single function, it should be entered only at the first statement of the segment, and it should exit only at the last statement.

The control logic contains no GOTO statements. In PL/I, the branching control can be defined entirely in terms of DO loops, IF-THEN-ELSE, and ON statements. The resulting code can be read strictly from top to bottom, with greater understanding than would otherwise be possible.[9]

Programming for virtual storage systems

Some programs tend to cause excessive paging when run in virtual storage systems. As these programs have been observed and the causes of paging analyzed, a set of techniques has evolved. In contrast to the concepts of structured programming, which are applied systematically during the process of program design and coding, the concepts of programming for virtual storage[12-14] are a loose body of individual techniques that may be applied during the design and coding phases of a project. These techniques are generally presented as "things to keep in mind"

because the present state of programming for virtual storage systems is much more of an art than a science.

virtual storage

The virtual storage systems refered to in this paper are those implemented in the IBM System/370 series of computers. In these systems, virtual address space is divided into fixed-length *pages* that map into page-size units in the computer's *real storage*, which is generally smaller than its virtual storage. These page-size units are called *page frames*. When a program refers to an address in virtual storage that is not currently in real storage, that occurrence is called a *page fault*. The operating system is designed to bring the referenced page into real storage through an action called *paging*.

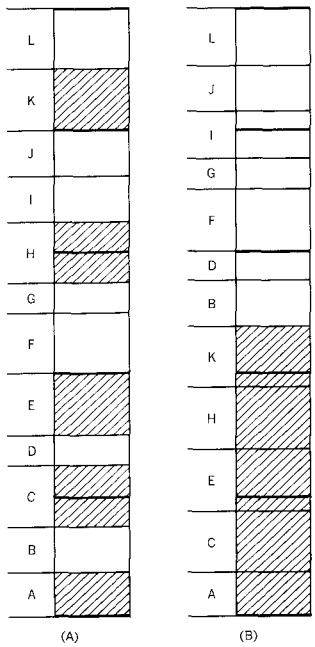
Each program that runs on a virtual storage system has, at any particular instant, a *working set* of pages that belong to it. The working set is simply the pages in real storage that the program is predicted to use in order to run for a given interval of time without incurring a page fault. The actual working set typically changes with the time intervals of the program's execution. If the set of programs that is contending for real storage space has cumulative working sets larger than the real storage available, then paging also occurs for that reason. If the rate of paging becomes so high that the system resources are used as much or more for paging than for productive work, a condition called *thrashing* is said to prevail.

When considering means for correcting an existing program that appears to be causing thrashing, one may hastily conclude that such a program may be incurring too frequent page faults. This, however, may not be the correct deduction from the facts. In the first place, not all page faults are incurred by a given program; other programs are incurring page faults as well. Further, if the given program were to run alone, it would probably not incur page faults. Thus a more useful deduction may be that the given program requires too much real storage to run in the prevailing multiprogramming environment. Rather than seeking ways of reducing page faults, it may be preferable to reduce the real storage necessary for the program by reducing the working-set size during periods in which the working set size may be a problem.

To determine the measures necessary to reduce a program's working set size, a useful approach is to begin with the working set and proceed backward into the program. This procedure has the desirable characteristic of beginning with large elements and considering successively smaller elements with each step. In taking this approach, one assumes that the working set size is too large, and at each step he looks at what can be done to reduce the working set size.

If a program's working set consists of too many pages, one's first action is to analyze the contents of the pages. Next in the programming hierarchy below the working set is the *control section*, which is the smallest element of a program that is identifiable by the linkage editor. Rearrangement of the most frequently referenced control sections may be possible so as to occupy fewer pages, as shown in Figure 2. Here the heavy black lines represent page boundaries, and the shaded portions represent control sections currently being referenced. Figure 2A shows that control sections A, C, E, H, and K are all that are currently needed in the working set. Because of the way in which the program is packaged, however, the working set consists of five pages. Figure 2B shows the control sections after rearrangement so that the current working set requires only three pages. Since the working set varies from time to time, the overall objective is to optimize the total execution of the program. Myers[3] gives a good description of the process of reordering control sections.

Figure 2 *The recording of control sections, (A) Given working set, (B) Reduced working set*



With the ability to reorder control sections, another consideration is to create control sections that lend themselves better to reordering. The easiest way to do this is to generate small control sections that have desirable characteristics. Such small control sections lend themselves to external arrangement to build pages that have desirable characteristics. The ease with which this can be done varies considerably among the different languages in use today.[14]

At this point, the paradigm program has been decomposed into modules. The program is thus at the terminal point of virtual storage design: a collection of proposed modules ready to be coded.

designing *for* virtual storage

It should be kept in mind during the design of program modules that are to be executed together with other program modules in a virtual storage environment that we are concerned with conserving and making the most efficient use of real storage. We need not be concerned about virtual storage by itself. Thus, those parts of programs that are executed only rarely need not be of concern.[14] This indicates that for any program there is a set of modules—perhaps entire branches of trees—for which storage is of no concern either during the design process or during the coding of the modules.

The modules must contain code, all of which is needed at the same time. Further, the larger a module that is written, the greater the distance the data needed by a particular instruction may be from the instruction itself, a condition that might well increase the working set size and possibly cause page faults. In essence, modularity should be by proximity of usage rather than by similarity of function.[14]

All code (if more than just a few bytes long) that is designed to handle exceptional conditions such as error recovery, permanent diagnostic capability, etc., should be apart from mainline code, preferably in separate control sections (CSECTS).[15] This gives the programmer the ability to remove such code from the program's working set, which thus becomes smaller.[14]

At this time, it is necessary to consider data that are referenced by more than one module. In other words, algorithms internal to some modules must be taken into account. As an example, suppose a module must build two 64 by 64 double-precision matrices. Each of these matrices requires 32K bytes or eight pages if the page size is 4K bytes. The algorithmic decision to be made at this time is whether these matrices are suitable for sparse matrix techniques. If they are, it is possible that a great savings in real storage can be accomplished. If they are not, then some decisions must be made. If another module must multiply these two matrices a decision must be made at this level as to whether one of them should be stored as its transpose.[16] The coding of multiplication does not require more complexity by one method than by another. There may be, however, a tremendous difference in the storage reference patterns. Figure 3A shows the corresponding elements to be multiplied for two such matrices, **A** and **B**, and Figure 3B shows the corresponding elements for **A** and the transpose of **B** (written $\mathbf{B}^t$).

Similar cases may exist whenever two modules must pass large masses of data between them. It is impossible to list all such possibilities. However, if one understands the principles well enough, then he will be aware of such occasions.

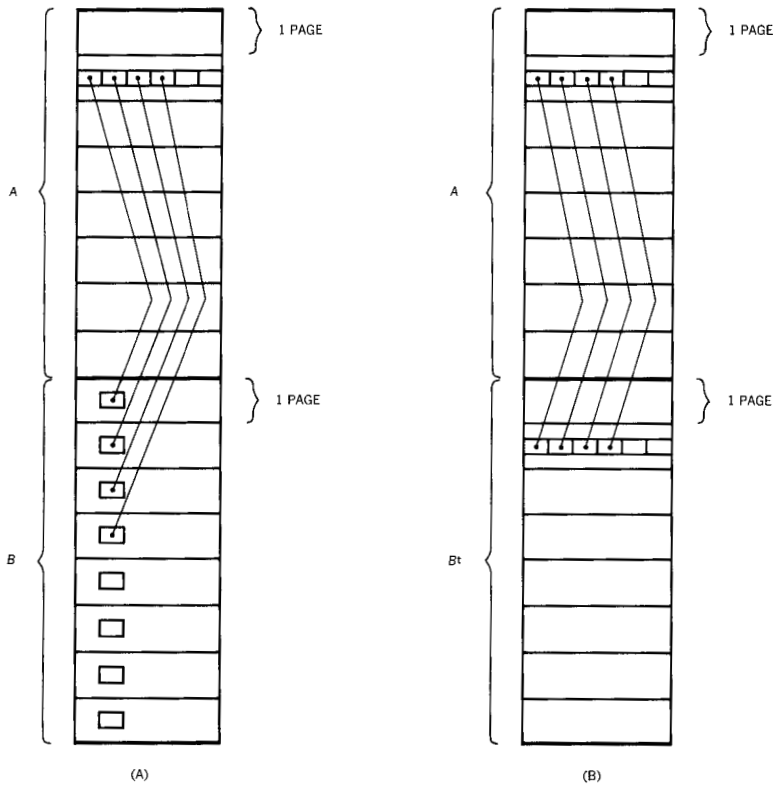
writing programs for virtual storage

Whereas design for virtual storage is considered here in terms of general principles, writing programs for virtual storage involves a somewhat more solid technique. There is still, however, an aura of art about such programming, and it is difficult to state general principles that are always valid and can be applied in "cook-book" fashion. Many of the techniques also depend on a specific language or a specific compiler implementation.

Some further rules that are generally applicable to programming for virtual storage systems are the following:

- Reference the data in the order in which they are stored, and/or store the data in the order in which they are referenced.[13] This is not always possible, of course, and the order in which an array is referenced is of no consequence if the array fits into a single page.[14]
- If possible, separate read-only data from areas that are to be changed.[14] This is of lesser importance than good locality of reference.

Figure 3 Module design for matrix multiplication, (A) Multiplication AB, (B) Multiplication AB^t



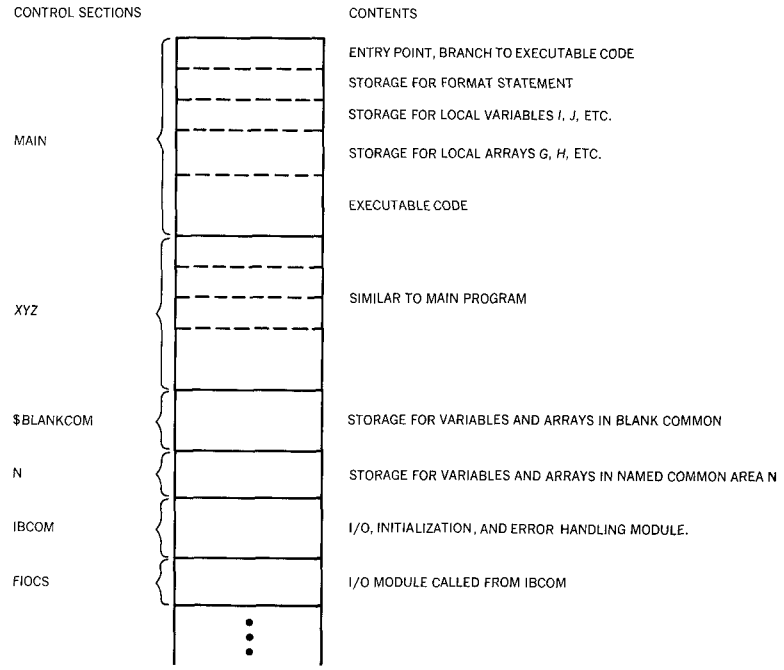
- Avoid the use of elaborate search strategies for large data areas, and avoid the use of large linked lists if these techniques cause a wide range of addresses to be referenced.[13]
- Virtual storage coding techniques vary by language and particular language implementation.
- In PL/I, the AREA variable is of particular interest. Based storage can be allocated inside an AREA variable. Thus, based variables that are to be used together can be localized.[14] The code fragment in Example 2 shows that based variables ALPHA and BETA can be made close together.

Example 2

```
DCL GAMMA AREA;
DCL (ALPHA, BETA) BASED;
. . .
ALLOCATE ALPHA IN (GAMMA);
. . .
ALLOCATE BETA IN (GAMMA);
```

- PL/I allows considerable flexibility in declaring data aggregates. Also in PL/I, arrays of structures and structures of ar-

Figure 4 Example FORTRAN storage areas



rays should be declared in the order in which they are to be referenced most frequently. Multiply dimensioned arrays in PL/I are stored rowwise.[14]

- FORTRAN stores arrays by column.[13]
- Avoid implied FORTRAN DO loops in I/O statements because they cause repeated return to the calling program.[14]
- In COBOL, data within the working storage section are allocated in the order in which they are declared. Therefore, declare data that are to be used together in consecutive statements.[14]
- Also in COBOL, files that are used together should be opened in the same statement. This causes their buffers to be close together and improves the locality of references as buffers are processed.[13]

Source language and compiler implementations

During the virtual storage program designing and coding processes, a choice of source language must be made. It is possible that a single language may be used throughout the program, but a mixture of languages may also be used. In any case, the characteristics of the language and the implementation must be understood if the program is to make efficient use of virtual storage.

The four primary languages used by IBM virtual storage systems are COBOL, FORTRAN, PL/I, and Assembler Language. For the first three, several different compilers are available, of which we are concerned only with the optimizing compilers because they tend to produce less code and reduce data references. The module structure of Assembler Language programs is under the control of the programmer, and it is not considered here. COBOL, FORTRAN, and PL/I and their optimizing compilers are discussed in the following three sections.

ANS COBOL version 4

The ANS COBOL version 4 compiler generates one control section for a main program and one for each separate subprogram. Data referenced within a program module are stored in the same control section. The data and executable statements appear in the control section in almost exactly the same order in which they appear in the source language statements. A large main program may be divided into several control sections by using the segmentation feature. When using segmentation, all data remain with the root segment; only executable code appears in the other control sections along with a small amount of control information.

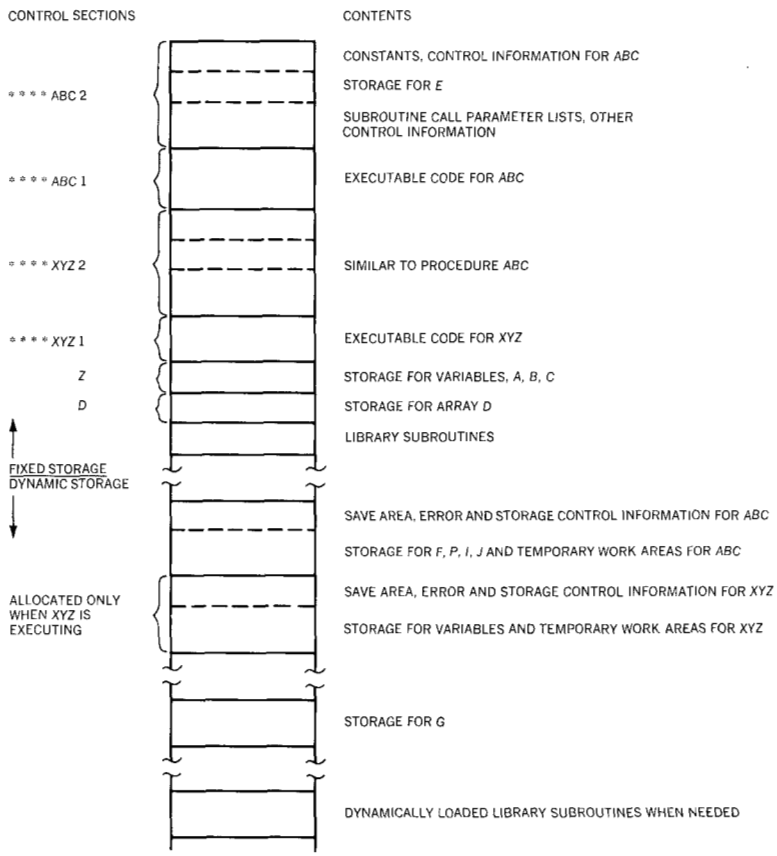
FORTRAN H extended compiler

The FORTRAN H extended compiler produces one control section for a main program and one control section for each subprogram. Blank COMMON is a control section. Each named COMMON area is a separate control section. Library subprograms are control sections. Most library subroutines are explicitly called, but some subroutines such as exponentiation routines and I/O routines are implicitly included. Figure 4 shows the control section structure for the code in Example 3.

Example 3

```
      COMMON A, B, C···
      COMMON /N/D, E, F···
      DIMENSION G(10), H(20),···
      ···
      J = 1
      DO 20 I = 1,100
      ···
20    CONTINUE
      CALL XYZ(J)
      WRITE (6,30)J
30    FORMAT (I2)
      STOP
      END
      SUBROUTINE XYZ(K)
      ···
      RETURN
      END
```

Figure 5 Example PL/1 storage areas



The most notable characteristic of the module structure is that local variables and arrays are in the same control sections as executable code, and thus they are not separable at linkage editing time. The IBCOM module, which is fairly large, is used during execution only for I/O statements and PAUSE and STOP statements. FIOCS is used for all I/O statements.

PL/I optimizing compiler

The PL/I optimizing compiler produces two or more control sections for each external procedure. One control section is created that contains only executable code, and another contains control information, constants, and internal variables. Other control sections are included as closed subroutines for actions such as character string assignments. Each variable, array, or structure declared as STATIC EXTERNAL is a separate control section. PL/I also uses dynamically acquired storage. Space for such procedures as register save areas, error handling information, AUTOMATIC variables, and work areas are in a dynamically acquired section of storage. These sections are obtained and freed dynamically when a procedure is entered. If procedure A calls

procedure B, then the dynamic storage area for procedure B is generally (and can be made to be) appended to the end of the dynamic storage area for procedure A. Variables that are allocated dynamically by the user with ALLOCATE statements are in a different dynamically acquired section of storage. Figure 5 shows the storage structure for the PL/I program in Example 4.

Example 4

```
ABC: PROCEDURE OPTIONS(MAIN);
     DECLARE
          1 S STATIC EXTERNAL,
          2 (A,B,C) FIXED BINARY(15);
     DECLARE D(10) FLOAT DECIMAL(16) STATIC
          EXTERNAL;
     DECLARE E(20) FLOAT DECIMAL(16) STATIC
          INTERNAL;
     DECLARE F(10) FIXED DECIMAL(4,2);
     DECLARE G(5) FIXED BINARY(31) BASED(P);
     DECLARE P POINTER;
          . . .
          J = 1;
          DO I = 1 TO 100;
          . . .
          END;
          ALLOCATE G;
          CALL XYZ(J);
     END;
XYZ:PROCEDURE(K);
     . . .
     END;
```

Virtual storage constraints on structured programming

The disciplines involved in structured programming offer, as was noted earlier, specific techniques that—when applied in a systematic manner—produce programs that are more easily understood than would otherwise be the case. The techniques involved in these disciplines take no notice of the virtual storage environment. In this section, the process of structured programming is analyzed and the principles of programming for virtual storage are applied against it. Thus a picture should emerge that shows the proper way to think about virtual storage during the structured programming process.

The first and one of the most important of all considerations is that of whether real storage constraints are important. Many programs have very little impact on the user's system. The concern becomes considerably less for large storage configurations and considerably greater for the smaller systems. Small programs, short-running programs, and one-time programs are not

typical subjects for programming for virtual storage. Large, long-running, and frequently-run programs benefit most from virtual storage programming techniques. Of course, a program that is large and long-running on a System/370 Model 145 with 512K bytes of real storage may be a small, short-running program on a System/370 Model 168 with 4096K bytes of real storage. This is one of the factors to be considered.[14] It cannot be emphasized too strongly that the designer must determine his degree of concern for real storage as early as possible in the design phase.

Just as there are entire classes of programs that are insensitive to virtual storage (when measured by system throughput) there are also parts of many programs that are similarly insensitive. A flurry of paging that occurs once every few minutes is of little concern. Thus there are things such as human interaction, error correction, and so forth that may be designed and coded as completely and elaborately as is convenient, simply because they occur infrequently.

design constraints

The process of structured programming begins in the design phase. The culmination of the design phase is a set of proposed modules that exhibit certain desirable characteristics. Among these are high module strength, low module coupling, predictability, size, and decision structure.

Some of these characteristics are quite easily dealt with. Predictability and decision structure are usually insensitive to virtual storage. Module strength tends to be generally compatible with virtual storage programming principles because it tends to localize code that is used at the same time.

Informational strength modules (modules that perform more than one function with an entry point for each function) may be sensitive to virtual storage. If not all of the included functions are needed at the same points in the program, then code may be dragged into the working set at times when it will not be used.

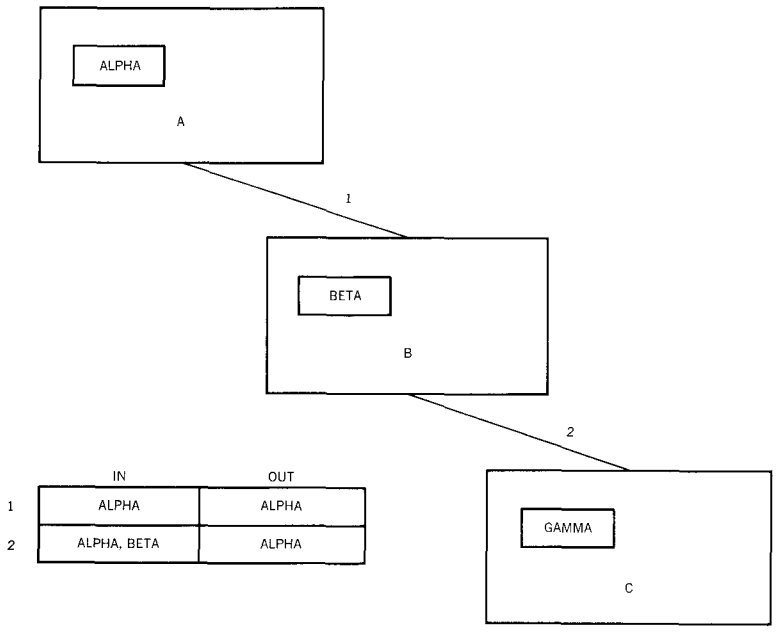
large data aggregates

Module size is a somewhat more complicated factor. Size, in this context, refers to number of source language statements. If this number is kept small, then the amount of executable code generated is also small. On the other hand, the addition of a statement such as

```
DECLARE X(1000) FLOAT DECIMAL(16);
```

increases the object module size by 8000 bytes, while adding only one source statement to the module. Although the module size—in terms of source statements—is small, the generated object module (which depends on the language implementation)

Figure 6 Large aggregate storage problem illustrated by matrix multiplication



	IN	OUT
1	ALPHA	ALPHA
2	ALPHA, BETA	ALPHA

becomes very large. Further, in high-level language implementations, there tends to be a separation of data from the code that references it.

The preceding problem of large data aggregates is merely one manifestation of an even greater problem of large data aggregates. The suggestion usually given is to make large data aggregates into independent modules (STATIC EXTERNAL in PL/I or COMMON in FORTRAN). This procedure, however, in the structured designer's view, causes a potential COMMON coupling that is undesirable. The purpose of making these data aggregates externally known is so that their location can be chosen at linkage editing time, and the user can arrange the aggregates so that real storage is conserved.

In the case of COBOL, all data must be internal. FORTRAN offers COMMON, which creates an undesirable COMMON coupling condition. PL/I offers STATIC EXTERNAL, which also creates COMMON coupling, but also offers the AREA variable. The AREA variable reserves a section of storage, either statically or dynamically acquired. BASED variables may then be allocated to space within the AREA. Under some circumstances, this mechanism can be used to preserve locality of reference.

An example of this problem is illustrated by Figure 6. Module A contains the storage for and creates matrix ALPHA. Module A calls Module B, and passes ALPHA as a parameter. Module B

contains the storage for and creates matrix BETA. Module B calls module C, and passes ALPHA and BETA as parameters. Module C multiplies ALPHA and BETA, and stores the result in GAMMA. Module C then sets ALPHA equal to GAMMA and returns. The FORTRAN program fragment in Example 5 accomplishes these operations.

Example 5

```
  SUBROUTINE A
  DIMENSION ALPHA(16,16)
  . . .
  CALL B(ALPHA)
  . . .
  STOP
  END
  SUBROUTINE B (ALPHA)
  DIMENSION ALPHA(16,16), BETA(16,16)
  . . .
  CALL C(ALPHA,BETA)
  . . .
  RETURN
  END
  SUBROUTINE C(ALPHA,BETA)
  DIMENSION ALPHA(16,16), BETA (16,16), GAMMA(16,16)
  . . .
  DO 10 I = 1,16
  DO 10 J = 1,16
  GAMMA(I,J) = 0.0
  DO 10 K = 1,16
10 GAMMA (I,J) = GAMMA(I,J) + ALPHA(I,K)*BETA(K,J)
  . . .
  DO 20 I = 1,16
  DO 20 J = 1,16
20 ALPHA(I,J) = GAMMA(I,J)
  . . .
  RETURN
  END
```

In this example, the user has little control over the areas that ALPHA, BETA, and GAMMA occupy in storage. Packageability of A, B, and C are hampered by the 1024 byte arrays inside each module. It is easily possible that the three modules can require three pages of real storage to execute, and it is possible that they may require six pages of real storage. If these arrays were put in COMMON—the only alternative in FORTRAN—they could all be put into one page, which would be present only when the arrays are being used. What appears to be desirable in this case in designing for virtual storage is clearly undesirable in structured design.

Data aggregates are not easy to cope with. On the one hand, making arrays internal is not a good practice for virtual storage. On the other hand, making the arrays COMMON opens them up for use by entirely different parts of the program, thereby creating a potential reliability problem. A closer look at the design process may help in finding a solution to the problem.

In some mathematical applications, the designer may be aware of alternative algorithms before the design process begins, or may at least become aware of them at some point during the design phase. There are, for example, at least two ways of calculating eigenvectors, one of which is highly inefficient in virtual storage and the other quite efficient. The designer should maintain an awareness of cases where large tables of data items are likely to be needed. Instead of filling a large table sequentially with values and then sequentially using the values one by one, it might be better to use each value as it is generated, and thus avoid the use of a table entirely. A large, sparsely used table that occupies several pages might be better utilized by storing table entries in sequential locations and keeping an index to the table rather than storing entries in locations of the table that correspond to item numbers.

Choice of implementation language may also be affected by some of the above design considerations. PL/I offers a better solution to the problem illustrated by FORTRAN in Example 5. The PL/I solution, which is shown later in Example 6, allows the matrices to be kept together and separate from the code, while it preserves DATA coupling between the modules.

constraints on writing structured programs

All the items discussed here thus far have been directed toward the design of structured programs. These principles also apply to coding within a module. If the designer realizes that a particular technique may be applicable within a module, he should point that fact out to the programmer, who may not be aware of it. Many techniques of programming for virtual storage are also available to the practitioner of structured programming.

Some general considerations in programming for virtual storage that also apply to structured programming are the following:

- Exceptional-condition code should be in a separate module that is called from the module in which the condition (say an error) occurs.
- Data should be referenced in the order in which they are stored, if possible, especially for large data aggregates that occupy several pages.
- Store data as closely as possible to other data that are to be used at the same time.

- Avoid the use of elaborate search strategies for large data areas.[13]

Many virtual storage programming techniques are valid for particular languages and compiler implementations only. Some virtual storage programming techniques that are usable with structured programming are now discussed.

ANS COBOL compiler version 4

Data within the working storage section are allocated in the order in which they are declared. Therefore, declare data that are to be used together in consecutive statements.[14] Declare files that are to be used together in consecutive declarations.[14] Files that are used together should be opened in the same statement. This causes their buffers to be closer together and improves the locality of reference as buffers are processed.[13] Do not use alternate areas unless they are needed for a special reason. The net effect of alternate buffers is to separate data areas.[13]

PL/I optimizing compiler

The AREA variable is of particular interest because based storage can be allocated within an AREA variable. In this way, based variables that are to be used together can be localized.[14] The localization of based variables is illustrated in Example 6, which accomplishes the same function as FORTRAN, as illustrated in Example 5.

Example 6

```
A: PROCEDURE;
     DECLARE OMEGA AREA(4000);
     DECLARE ALPHA(16,16) BASED(APTR);
       . . .
       ALLOCATE ALPHA IN(OMEGA);
       . . .
       CALL B(OMEGA,ALPHA);
       . . .
   END A;

B: PROCEDURE(OMEGA,ALPHA);
     DECLARE OMEGA AREA(*);
     DECLARE ALPHA(16,16);
     DECLARE BETA(16,16) BASED(BPTR);
       . . .
       ALLOCATE BETA IN(OMEGA);
       . . .
       CALL C(OMEGA,ALPHA,BETA);
       . . .
   END B;
```

```
C: PROCEDURE(OMEGA,ALPHA,BETA);
     DECLARE OMEGA AREA(*);
     DECLARE (ALPHA,BETA)(16,16);
     DECLARE GAMMA(16,16) BASED(GPTR);
       . . .
       ALLOCATE GAMMA IN(OMEGA);
       . . .
       DO I = 1 TO 16;
         DO J = 1 TO 16;
           GAMMA(I,J) = 0.0;
           DO K = 1 TO 16;
             GAMMA(I,J) = GAMMA(I,J) + ALPHA(I,K)*BETA(K,J);
           END;
         END;
       END;
       ALPHA = GAMMA;
       . . .
   END C;
```

In Example 6, the AREA, OMEGA, has been made large enough to hold ALPHA, BETA, and GAMMA, so that, when they are allocated to the area, ALPHA, BETA, and GAMMA are together in storage and create a better locality of reference.

Avoid, if possible, putting variables with the initial attribute in automatic storage.[14]

Be especially careful of arrays of structures and structures of arrays. Be sure to declare them in the order in which they are expected to be referenced most frequently.

PL/I passes all arguments in CALL statements by location rather than by value. Consider the program fragment in Example 7.

Example 7

```
X:  PROCEDURE OPTIONS(MAIN);
     DECLARE I STATIC INTERNAL;
       . . .
       CALL Y(I);
       . . .
    END X;
Y:  PROCEDURE(I);
     DECLARE J STATIC INTERNAL;
     . . .
     CALL Z(I,J);
     . . .
   END Y;
```

```
Z:  PROCEDURE(I,J);
      ...
      K = I + J;
    ...
    END Z;
```

general guidelines

Whenever modules Y or Z use the variable I, they reference the STATIC INTERNAL control section of X. Further, whenever Z references J, it references the STATIC INTERNAL control section of Y. Thus the working set, when Z is executing, may be larger than need be. To reduce the working set, assign the parameters to internal variables and then use those variables.

The purpose of special programming for virtual storage is to reduce a program's real storage requirement, an objective toward which the following guides may prove to be helpful:

- Separate unused code and data space from code that is frequently used.
- Seek algorithms or techniques that use small data areas.
- Remember that the modules created during the design and coding phases can be reordered during linkage editing to create better reference patterns.

Structured programming constraints on programming for virtual storage

Because the available literature on programming for virtual storage consists mainly of isolated techniques, this section attempts to bring together such practices and point out ways in which they do not fit within the constraints of structured programming. These practices may be well worth avoiding when they conflict with structured programming for virtual storage systems.

- Grouping high-use buffers and data areas together in common storage[13] causes a potential COMMON coupling and further forces an artificial structure on data.
- Making common data areas more productive by using the same area for different data in different phases of the program[13] creates even more severe COMMON coupling than grouping high-use buffers, because program changes in one area may cause errors in unrelated areas.
- Setting up initial conditions at the beginning of the program[14] causes poor module strength and makes later modification more difficult.
- Declaring structures and aggregates EXTERNAL[14] also causes COMMON coupling.
- Defining every variable as STATIC, particularly for arrays and data structures,[13] is not always good practice, as Example 7 shows.

The following are general guidelines that should be observed when writing structured programs for virtual storage systems:

- Do not create code that is impossible to comprehend, just to save real storage. In many cases, there is a simple way to accomplish the same thing.
- Do not create modules that have poor coupling or strength, to save real storage. Look for another way to do it. If it is impossible, perhaps another language is better suited to the application.
- If it is necessary to make modifications to achieve further savings in real storage, those modifications may be made more easily if structured programming techniques are used.

Concluding remarks

The concepts and techniques of virtual storage programming and those of structured programming have grown along diverse paths. Virtual storage programming has evolved largely by looking retrospectively at programs designed and coded by means other than structured programming, and thus has included techniques that are not applicable to structured programming. Structured programming has evolved with little regard to virtual storage, and thus includes techniques alien to virtual storage systems.

The two disciplines are not, however, at odds with each other. Very few of the concepts involved in structured programming cause problems in virtual storage. The few that do cause trouble can usually be avoided during the design and coding phases, and alternate methods may be found that more easily accommodate virtual storage. The few techniques of virtual storage programming that do not fit well within structured programming are usually recognizable by those who are engaged in structured programming.

The combination of these two disciplines requires more effort on the part of the designer and the programmer than either one does alone, but together they produce programs that are more easily understood and maintained, and put less strain on computer storage resources.

CITED REFERENCES AND FOOTNOTES

1. "Structured design" is also referred to as "composite design" and "functional design."
2. G. J. Myers, *Reliable Software Through Composite Design,* Mason and Charter, Publishers, Inc., New York, New York (1975).

3. G. J. Myers, *Composite Design: The Design of Modular Programs,* Technical Report TR00.2406, IBM Corporation, Poughkeepsie New York 12602 (January 29, 1973).
4. G. J. Myers, "Characteristics of composite design," *Datamation* **19,** No. 9, 100–102 (September 1973).
5. W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal* **13,** No. 2, 115–139 (1974).
6. E. W. Dijkstra, *Notes on Structured Programming,* T. H. Report WSK-03, Second Edition, Technical University Eindhoven, The Netherlands (April 1970).
7. E. W. Dijkstra, "GOTO statement considered harmful," *Communications of the ACM* **11,** No. 3, 147–148 (March 1968).
8. H. D. Mills, *Mathematical Foundations for Structured Programming,* FSC 72-6012, IBM Corporation, Gaithersburg, Maryland 20760 (February 1972).
9. H. D. Mills, *Structured Programming,* FSC 70-1070, IBM Corporation, Gaithersburg, Maryland 20760 (October 1970).
10. Meyers' limits on module size in Reference 2 largely preclude segmentation.
11. In this passage, Mills uses the word "program" synonymously with the word "module," as used in the present paper.
12. D. J. Hatfield and J. Gerald, "Program restructuring for virtual memory," *IBM Systems Journal* **10,** No. 3, 168–192 (1971).
13. J. E. Morrison, "User program performance in virtual storage systems," *IBM Systems Journal* **12,** No. 3, 216–237 (1973).
14. J. G. Rogers, "OS/VS programming considerations," *IBM Installation Newsletter,* Issue No. 73-01, 2G-11G, IBM Corporation, Data Processing Division, White Plains, New York 10604 (January 26, 1973).
15. CSECT is an acronym for control section.
16. If the value in the ith row and jth column of a matrix **A** is written as $A(i, j)$, then for its transpose $\mathbf{A}^t$, $A^t(j, i) = A(i, j)$. In other words, the matrix is flipped across its diagonal.
17. A. A. Dubrulle, "Solution of the complete symmetric eigenproblem in a virtual memory environment," *IBM Journal of Research and Development* **16,** No. 6, 612–616 (November 1972).

Reprint Order No. G321-5023