Storage Management in Ada Three Reports

David Auty

SofTech, Inc.

January 29, 1988

Cooperative Agreement NCC 9-16
Research Activity No. SE.9

ricis

Research Institute for Computing and Information Systems
University of Houston - Clear Lake

TECHNICAL REPORT

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information systems in 1986 to encourage NASA Johnson Space Center and local industry to actively support research in the computing and information sciences. As part of this endeavor, UH-Clear Lake proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a three-year cooperative agreement with UH-Clear Lake beginning in May, 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The mission of RICIS is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from UH-Clear Lake, NASA/JSC, and other research organizations. Within UH-Clear Lake, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business, Education, Human Sciences and Humanities, and Natural and Applied Sciences.

Other research organizations are involved via the "gateway" concept. UH-Clear Lake establishes relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research.

A major role of RICIS is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. Working jointly with NASA/JSC, RICIS advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research, and integrates technical results into the cooperative goals of UH-Clear Lake and NASA/JSC.

Storage Management in Ada Three Reports

Preface

This research was conducted under the auspices of the Research Institute for Computing and Information Systems by David Auty of SofTech, Inc. under the technical direction of Charles W. McKay, Director of the Software Engineering Research Center (SERC) at the University of Houston-Clear Lake.

Funding has been provided by the Avionic Systems Division, within the Engineering Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA Technical Monitor for this activity was Terry Humphrey, Data Management Section, Flight Data Systems Branch, NASA/JSC.

The views and conclusions contained in this report are those of the author and should not be interpreted as representative of the official policies, either express or implied, of NASA or the United States Government.

Storage Management in Ada
Three Reports

January 29, 1988
(SofTech Document WO-123)

A UHCL/RICIS Report, Contract SE.9

Vol. 1: Storage Management in Ada as a Risk to the Development of Reliable Software

Vol. 2: Relevant Aspects of the Language

Vol. 3: Requirements of the Language Versus Manifestations of Current Implementations

Prepared for

NASA Avionics Systems Division, Research and Engineering
Johnson Space Center

Prepared by

SofTech, Inc.
1300 Hercules Drive, Suite 105
Houston, TX 77058-2747

Volume 1

Storage Management in Ada as a Risk to the Development of Reliable Software

Storage Management in Ada as a Risk to the Developoment of Reliable Software

This report addresses a particular concern associated with the use of Ada for the development of Space Station software. The general concern is in the assurance of proper functionality of software which is to be depended upon for life and property. With this as the highest priority, the possibility of non-deterministic or difficult to verify software must be addressed; risks to the development of reliable software need to be identified and approaches outlined for reducing the risk.

In particular, this report will address storage management as one of these risks. The project of which this is a part is concerned with identifying such risks, clarifying their nature, and investigating and comparing alternative approaches. As a first step, this report addresses only the identification and clarification of the risk.

Storage management is not a new concern for NASA. As one of two limited resources in computing (CPU time being the other), and computing resources being in critical demand in most real-time computing applications, it is often addressed in budgeting and allocation decisions central to the systems software architecture and design. The space station requirements, however, pose new complexities in size and distributed system interactions. Ada and dynamic storage management are seen as tools to address this complexity, but use of these tools while maintaining and demonstrating a high standard for reliability poses a significant challenge.

Dynamic storage management in any form introduces a complexity of processing which, without some control and careful design, must be considered risky for critical software. In the general case, storage use which is dependent upon program execution implies that, in order to assure no storage use faults (e.g., out of storage), exhaustive testing must be applied. In complex systems such testing is impractical. Fortunately many forms of dynamic storage management offer built-in limitations and additional analysis techniques may offer assurances against storage errors.

TABLE OF CONTENTS

Section 1

ELEMENTS OF RISK

In simple terms, the risk to program reliability is derived from the use of a new language and from the potential use of new storage management techniques. With the novelty of Ada and associated support software, there is a lack of established guidelines and procedures, drawn from experience and common usage, which assure reliable behavior.

1.1 A New Language and its Support Software

The first source of risk to look at is simply the introduction of a new language. In the case of Ada, consideration also has to be given to all of the new support software which must accompany the language (compilers and runtime support systems, etc.). Ada dictates consideration of the support software because the language does not address the details of storage utilization, thus much freedom is given to different approaches provided by the support software.

Consider the concerns of an application writer when faced with a new language. In the case of storage management some of these are:

- How is storage utilization tied to the various features of the language?
- How do different constructs in the language imply different storage management requirements?
- What control is there over various options in storage management techniques?

Ada's lack of detail in storage management complicates the issue. Some aspects of storage management are implied by the definition of the language while other aspects clearly must be defined by the implementation.

To ease these problems, a coherent model of storage management must be presented which answers these concerns, the simpler the better. Development of reliable software depends on such a coherent model and upon the reliable implementation of that model in the support software.

1.2 New Storage Management Techniques

The second source of risk is new storage management techniques. In previous systems, storage budgeting and allocation have always been able to depend upon a relatively static model of storage use. In some designs, storage lies unused, waiting for invocation of the code which would put it to actual use. This means a somewhat excessive allocation of storage, but the program and programmer (designers, system verification, etc.) can depend upon it being available immediately and without question at the point of its use.

Different techniques have been introduced to provide a more effective use of storage. Common blocks and equivalence statements in FORTRAN provide a way to set aside an area of storage and put it to use in different ways at different times in the program's execution. Procedural languages which use a program stack for procedure-related data (local data, parameters, etc.) provide a mechanism for associating storage with procedure execution on an as needed basis only, eliminating much of the excessive allocation problem, but introducing a more complex dynamic model of storage use. A more extreme step in dynamic storage allocation is the use of a dynamic storage heap, where storage is not allocated for any particular use until it is needed at runtime. With a storage heap, storage is available on demand for any purpose, in any amount provided a suitable contiguous chunk remains in the heap, and is returned to the heap when it is no longer to be accessed for that particular use. Variations of these techniques exist in numerous forms, each with differing degrees of complexity and dynamic behavior patterns.

Ada provides the possibility for just about any of these storage management techniques, although equivalence and renaming of storage are specifically outside the intended use of the language. Thus significant freedom exists to utilize storage in whatever way best suits the application.

This freedom however is at the expense of a simple model of storage use and the implicit assurance of storage availability when it is needed. While storage management in Ada need not be fundamentally any different in the approach required for resource budgeting and allocation, unfamiliarity and new storage management models can add significant complexity to the process, and thus increase the risk of a lack-of-resources error during program operation.

1.3 A Summary List

Based on these sources of risk, the following elements of risk can be identified:

- lack of knowledge of how storage management is tied to the use of the language,
- lack of control over how storage is managed,
- lack of guidelines for establishing practical storage use budgets, and
- lack of techniques to demonstrate proper behavior and assure a lack of errors in actual use.

Section 2

A FRAMEWORK FOR RELIABLE STORAGE MANAGEMENT

Dynamic storage management will require some changes to storage budgeting and allocation policies. Storage allocation will become temporal and associated with execution patterns. Similarly, the use of Ada introduces new linguistic and compilation aspects to storage utilization. However, the principles of storage budgeting and storage management are not changed. Given sufficient control over storage utilization, the same principles of budgeting can be applied.

Reliable storage management is based on two critical aspects: a detailed model (budget) of how storage will be allocated and sufficient control over storage utilization to ensure that actual storage use adheres to the budget. Confidence in storage management comes from a well understood model of storage use and confidence in the support software to implement that model. The introduction of a new language and new storage management techniques need not destroy this confidence so long as reliable support software implements a manageable model of storage use.

Storage allocation in Ada will be covered in more detail in subsequent reports, but for now note that budgets can be assigned to individual procedure execution, task execution stacks, the number of active tasks and dynamic allocation pools. Through the use of various language features, and if necessary with specialized support software, various models of storage use, including dynamic allocation, can be reliably managed.

An associated problem of reliable storage management is that of verifying that storage errors will not occur. Again, a new language and new storage management techniques do not change the principles of verification and program testing. With a well designed model of storage use, well understood techniques of test and verification can be applied. For critical applications, the requirements for test and verification will impose the requirement for deterministic behavior of storage management. This may rule out some forms of dynamic storage use, but not necessarily all.

The key to reliable storage utilization is the establishment of storage use budgets and the understanding and control over storage use to fit within that budget. In the case of dynamic storage management, it is necessary to match the storage use budgets to the dynamic behavior of the system, and possibly design the dynamic behavior of the system around the established storage budgets. In the case of Ada storage management, it is necessary to understand how storage use is tied to the language, what runtime variances exist and what controls are available. In either case, the approach to storage use and management rests on a basis of budgets and control.

Section 3

COMPROMISES AND TRADEOFFS, THE BASIS FOR RECOMMENDATIONS

Unfortunately, no one answer will necessarily fit all situations. There are conflicting goals with different priorities within different applications. The factors which affect the selection of approach for storage management include:

- o The critical nature of the software; the extent to which software faults can be tolerated.
- o The cost of program development, in the sense that detailed attention to storage use will require additional development effort.
- o The cost of support software to provide specific storage management support and control over storage utilization.
- o The balance between risk due to the presence of dynamic storage management with risk due to the adoption of a static storage management scheme.

The last factor is of particular interest. One perspective is that static memory management unnecessarily limits the expressiveness of the language. Algorithms which are naturally stated in recursive form or using dynamically allocated objects become burdened with additional code to manage a static pool of data. Thus, while a static storage model presents a simple model to the application developer, it can force complexities into the application itself.

This implies a tradeoff which must be recognized between the risk of storage errors introduced by dynamic storage management and the risk of errors introduced by its avoidance. Dynamic storage management introduces a dependency on an implementation's storage model and upon the reliability of the support software, but is part of Ada because it offers the power to reduce complexity and total storage requirements. The elegance and simplicity of a recursive or multi-processing algorithm must be balanced against system dependencies which may be difficult to impossible to verify for correctness.

For this project, a range of solutions and an evaluation of the tradeoffs will be provided. In critical applications, it can be expected that certain programming forms will be restricted. On the other hand, it may be anticipated that given enough resources to provide tools and adapt compilers and runtime systems, the risks of dynamic storage management can be reduced to a level approaching that of static storage management. As a compromise, there may be an approximation to this ideal which is less taxing and considered acceptable. It is hoped that a clearer picture of the alternatives and dependencies will be the result of our effort.

Volume 2

Relevant Aspects of the Language

Relevant Aspects of the Language

In order to provide a framework for future consideration of dynamic storage management in Ada, this paper will present a description of the relevant aspects of the language. This description will be organized in two main sections: Program Data Sources, and Declaration and Allocation in Ada.

TABLE OF CONTENTS

PRECEDING PAGE BLANK NOT FILMED

Section 1

Program Data Sources

Normally, programmers view the principle source of program data to be the declaration or use of variables within the text of the program. Indeed this is the principle source of large data blocks, but for completeness, two other sources must be considered, making the list of data sources as follows:

- o Program Variables
- o Compiler Generated Objects
- o Runtime-Support Data (generated by runtime support library)

These three sources will each be considered in turn.

1.1 Program Variables

Program variables in Ada come in many flavors, some very familiar and others, perhaps less so. To understand the storage use of a program requires an understanding of the storage required for the various types of program variables. What follows is a quick overview of data typing in Ada and the storage requirements for each. Special mention will be given to the options for representation and storage use which programmers may find in different implementations of the language.

Program Variables in Ada are either:

- o scalar data objects,
- o composites of scalar data objects, or
- o references to one of a collection of such data objects (called access variables, to be discussed later)

To further divide the world of Ada variables, scalar data objects consist of:

- o numeric values, or
- o enumerated values (one of an enumerated list of legal values, e.g., True or False, etc.)

Enumeration variables provide a convenience for the programmer, allowing mnemonic naming of values which are often represented as unsigned integers. Enumeration variables, particularly the character enumeration type, provide the basic mapping from non-numeric values (symbols) to the numeric representations available on the hardware. For both enumeration variables and numeric variables, if supported by the compiler, the user may control the representation and therefore the amount of storage required. Generally scalar values are represented as bytes, words or double-words in memory.

Composite variables are either organized as collections of similarly typed objects, called arrays, or as collections of dissimilar objects called records. Again, provisions in the compiler may allow for control over the layout of components within a composite object. Generally composite variables require a block of contiguous storage of roughly the sum of the component object sizes. Some increase in storage may be required for padding and alignment to improve access to the components.

These aspects to the storage requirements of the language are not significantly different from those of other languages. While other primitive types may be found in other languages, similar typing of data may be formed from the flexible typing rules of Ada.

Perhaps more unusual among programming languages, Ada includes specifications for the precision and range of numeric values, allowing the programmer to explicitly specify these characteristics within the declaration of numeric objects. Most compilers will select an appropriate size according to these specifications from among a small number of options supported by the hardware, typically the byte, word or double-word mentioned above. Because of this it may not be explicitly clear what size object has been created. This size, however, is usually not too difficult to derive for each machine. One would expect familiarity to be established within each project group for a particular target implementation.

A further complication in Ada has to do with the declaration of records. Ada allows for the named declaration of a variable record type, within which the components and sizes of components may vary from one object to another. Under certain circumstances Ada provides the capability to declare a variable which may be assigned different variants of the record. In order to accommodate such a situation, implementations of Ada must either allocate enough storage to accept the largest possible variant, or turn to dynamic storage allocation. In the later case, if an assignment exceeds the current storage size, new storage is allocated and the previous storage is left for some other use. Clearly, such declarations need special attention when planning and reviewing storage requirements.

The third category of program variable is the access variable. It is used for referencing one of a collection of objects which may be allocated dynamically by the program. These objects are all of the same type, but may be of any of the above program variable types, i.e., an access variable may reference a collection of scalar objects or a collection of composite objects of any type. The collection of objects accessible from a given access variable is called a "collection" in the LRM (Language Reference Manual). The identification of these objects as a distinct body of storage is an important aspect to storage management in the language.

Access variables themselves require a small amount of storage, typically one or two words. The collection of objects accessible from an access variable, however, is potentially unbounded. Ada allows an optional clause which provides an upper bound on the storage allocated for such a collection. Exceeding this limit raises an exception (error condition) even while storage remains for other allocations.

The final category of program variables, that of task identifiers, is the most unusual in terms of its association with storage requirements. There are in fact several items of storage which can be associated with tasks, however, none may be directly referenced by reference to the task variable. For our purposes these may be listed as:

- o any variables declared in the task body,
- o the task execution stack (for storage required by procedures called by the task),
- o any objects allocated dynamically by the execution of the task,
- o the task control block (for each task object), and
- o the task descriptor block (for each task type).

Only the first three of these will be considered here as program variable storage which can be associated with a task. The task control block and task descriptor block will be considered as runtime support data and compiler generated objects respectively. Ada allows an optional clause specifying the amount of storage to be reserved for the execution of a task. This typically specifies the size of only the task execution stack, but which objects are allocated from this reserved storage and which are allocated separately is not prescribed by the LRM. Use of this clause is necessarily implementation dependent.

1.2 Compiler Generated Objects

The second source of data for an executable program is the compiler. For many features of the language it is necessary for the compiler to generate data objects for reference at runtime. This is of course dependent on the program text, but may be independent of any variables declared in the program.

Perhaps the most common use of compiler generated objects is for expression evaluation requiring intermediate values to be held in storage. These temporaries may be needed within the execution of a single statement or across a range of statements. Another use is for runtime type descriptors. Sometimes, but not always attached to data objects, these descriptors are associated with the type declarations of the program. Of these, the most common use is for arrays, giving the number of dimensions, and upper and lower bounds of indices for each dimension. A third common use of compiler generated objects is the runtime storage of enumeration literals. This supports the language features of I/O and image attributes.

In a different category are objects for use in program control. Very often it is necessary to save such information as return addresses and exception raised flags. A similar example would be indexed-jump tables for case statements, although different implementations may treat this as part of the object code. This introduces a complicating factor in the discussion, that some implementations may require certain objects to be treated as data while others will treat them as code. In general, we will treat static, compiler-generated data which is part of program control as part of the object code. Compiler generated objects can then refer to storage for data required by the compiler but generated at runtime.

There are numerous situations which may require the use of compiler generated variables. It is, however, highly dependent on the implementation approach. Users will have to become familiar with each implementation's use of compiler generated objects to fully anticipate the storage utilization of a given program.

1.3 Runtime Support Data

The last category of data sources is the runtime support library itself. Most implementations of Ada will have a number of support routines which can be called upon to implement some of the more complex aspects of the language, tasking and I/O being the principle candidates. The task control block has already been mentioned as one product of the runtime support library. This object is used to help control the execution, waiting and rendezvous of active tasks in the system. Similarly, to help manage a dynamic storage heap and exceptions properly, other objects and additions to program declared objects may be generated by the runtime support library. Again, this is a very individual characteristic of each implementation which will require familiarity on the part of the user in order to fully appreciate the storage demands of the program.

A sampling of runtime support data includes:

- task control blocks
- heap management control records
- procedure activation records of the runtime library routines
- file control blocks (the entire I/O support requirements may be quite extensive)

Section 2

DECLARATION AND ALLOCATION IN Ada

Ada requires all program data to be explicitly declared and qualified in terms of the type of information and the operations which will be allowed (the Ada type specification). This is a starting point for consideration of storage management in the language. The program data is completely listed within a program unit specification or its declarative part. It is the nature of that program unit (the declaration's context) and the type of the data which determines the nature of storage management which will be utilized.

A second aspect of the language affecting storage management is the presence of "allocators" which provide for dynamic allocation of objects above and beyond the declared objects of the various program units. Allocators are always introduced by the keyword NEW.

These two aspects of allocation are covered in the next two subsections respectively.

2.1 Declared Objects

In order to understand the relationship between program declarations and storage allocation it is important to have a basic knowledge of elaboration. This is covered first, followed by a discussion of different declaration contexts and their differing requirements for storage management.

2.1.1 Elaboration of Program Declarations

The relationship between program data declarations and any implied storage allocations is intrinsically tied to the concept of elaboration. The Language Reference Manual refers frequently to the process of elaborating an object's declaration. The process of elaboration refers to the evaluation of the clauses which make up a declaration. It is not necessary to understand all of the subtleties which define the elaboration process, but is important to recognize that each declaration gets evaluated and processed in order to establish a variable prior to its use.

It is convenient to talk of the process of elaboration as a sequential processing of each declaration occurring prior to the execution of a given program unit. This is processing required to establish a variable's size (possibly dynamically determined), location and initial contents prior to the variable's use. The LRM has formalized the definition of this process and stated requirements for the nature and order of elaboration.

Elaboration can also be thought of as a sequential processing of declarations by the compiler in order to interpret the meaning of the program text, such as exactly which objects and operations are being referred to by the use of particular names. If an Ada program were being interpreted, these two processes would occur at the same time. It is in this context that the term refers to a single process.

In the case of compiled object code, many of the implications of elaboration have been already been established and others have been combined, so that individual actions at runtime are only infrequently associated with the elaboration of a specific declaration. For example, consider the following code segment:

```
Procedure XYZ ( N : in integer) is
     A : integer;
     B : float;
     vector : array ( 1..N ) of float;
Begin
  ...
End;
```

In an interpretive system, the local variables would be processed sequentially while reading the text during execution of the program. Local names would be established, storage allocated and initial values assigned, each at the time the declaration is encountered. In a compiled system, the compiler would process the text, reading all declarations and generating code to perform some steps at runtime as necessary. However, name association would require processing only at compile time, with the results being built into the object code produced. Thus some processing can be eliminated from runtime, and the opportunity exists to combine the sequential steps. In this case, only the size of vector would be left to runtime determination and allocation of storage for all of the local variables could be combined into the allocation of a single block. The allocation could be accomplished with a single increment of a stack pointer.

2.1.2 Declaration Context and Allocation

Storage allocation is stated as the last of several steps required for the elaboration of a declaration. As this allocation occurs logically in a sequential fashion as a part of elaboration, the static context of a declaration and the implications of dynamic execution of the program text define the requirements for storage management. This is the long way around for what is often a familiar and simple process, but it helps to understand the formal terminology and establish a consistent framework for discussion.

In the above example it was noted that allocation of storage would be a runtime activity. This is reflective of the language characteristic that all procedures and functions are potentially reentrant, i.e., may have multiple entries prior to any one exit. Reentrancy is introduced by the features of recursion and multi-tasking. Reentrancy prohibits the pre-runtime allocation of local data for procedures. It is an example of the significance of context to storage management requirements.

For declarations within subprogram units (procedures and functions, and begin blocks within them) a FIFO or stack based storage management scheme is required. Storage is allocated upon entry to the unit (when called) and deallocated upon exit. Because allocation and deallocation follow the discipline of subprogram entry and exit, a stack is a sufficient storage management scheme. It is necessarily dynamic, but more manageable than a generalized heap mechanism.

Program library unit packages (those which are not nested in any other program unit) are the one context allowing pre-runtime allocations. These packages define data and subprograms which are "global" in nature and available to all subprograms at all times. In this case there is no requirement for multiple copies or dynamically created copies. (This does not extend to declarations within subprograms declared in the package.) Alternatively, packages may appear within subprogram definitions, or in other contexts. In these cases, storage for the package would be allocated at the same time as other storage for the parent unit.

An implementation option exists which would extend static storage allocation to declarations within subprogram units. While the language specifies in general that subprogram units are reentrant, to support recursion and multi-tasking, if these features are not needed and are not present (a "promise" from the programmer) this may be indicated to the compiler via an implementation defined pragma (e.g., PRAGMA STATIC). In this case, static allocation for subprogram data may be adopted.

The declaration of tasks introduces the third distinct context for declarations. The declaration of a unique task object, e.g.,

```
Task buffer_task is ... End buffer_task;
```

is similar in some respects to a procedure or function declaration, different in others. If contained within a dynamic program context, then storage must be allocated dynamically as for other data in this context. If, however it is contained in a static context (library unit package), storage may optionally be allocated pre-runtime along with the other package data.

Note, however, that a task may be declared as a task type, e.g.,:

```
Task Type buffer_task is ... End buffer_task;.
```

In this case, the data is not allocated at the time the declaration is encountered, but rather at the time that a task object is allocated with the new operator.

The final context of special interest is that of Generic Unit declarations. The difficulty in considering generic units is the significant dependency on various implementation strategies. One approach to generic units is to treat them essentially as source macros. With this approach, the instantiation of a generic unit is treated just as if the generic declaration were expanded in place. Thus storage utilization is no different from other declarations at the point of instantiation. Unfortunately, this can lead to excessive and unnecessary redundancy in storage allocation.

Other more complicated implementation techniques are possible, but more difficult to describe in terms of their memory utilization. Storage requirements in these cases will be a combination of storage requirements derived from the specific instantiations and from the generic unit's definition itself.

2.2 New Allocations

The second source of allocation requirements are the presence of NEW clauses in the program text. These allocators require the dynamic allocation of storage at runtime. Such allocation, however, is always associated with the access type of the variable which will initially reference the object, and thus added to the implicit collection of storage for that access type. Such collections may be individually bounded by representation clauses or collectively bounded by the amount of free storage available in the target computer. The ability to set specific limits on storage allocated to individual collections is a principle source of storage management control in Ada.

Dynamic Allocation for NEW data objects generally requires the support of a heap allocation scheme. The exception to this rule involves the presence of a clause limiting storage allocations for a constrained access type (where the allocated size is known pre-runtime). In this case, the storage can be allocated at the time the access type is introduced, and linked into a queue of available storage blocks. Management for these storage blocks avoids the complications of a general heap allocation scheme, is consequently much simpler and more time and space efficient.

Volume 3

Requirements of the Language Versus Manifestations of Current Implementations

Requirements of the Language Versus Manifestations of Current Implementations

It is easy to confuse characteristics of a programming language with characteristics of an implementation of that language. For example, the Ada Language Reference Manual imposes no limit on the size of an array, but almost every implementation has such a limit. Similarly, currently available implementations of the rendezvous are typically slow, but the rendezvous is not an inherently inefficient language feature; several schemes for efficient implementation of the rendezvous have been proposed, though they have not generally been incorporated in current compilers.

This report distinguishes between storage-management characteristics of the Ada language and storage-management characteristics of Ada implementations. The distinction is relevant to this study for a number of reasons. First, it identifies criteria that may be important in the selection of a compiler. Second, it suggests aspects of program behavior that may vary when a program is ported from one compiler to another. Appropriate programming guidelines can limit the effects of this variance. Third, the distinction clarifies which Ada programming guidelines are universally appropriate and which guidelines are appropriate only for certain implementations.

The report is divided into three sections. Section 1 defines terms that will be used in this report in a narrow and precise sense. Section 2 describes the storage-management implications of the Ada language. Section 3 describes storage-management options available to the Ada implementor and the implications of the implementor's choice for the Ada programmer.

TABLE OF CONTENTS

Section 1

TERMINOLOGY

To avoid confusion, we adopt precise meanings for certain terms that are commonly used in a variety of senses:

- collection
- designated variable
- collection region
- task
- correspond
- task object
- declared variable
- task unit
- designate

All variables in an Ada program are created either by elaboration of a declaration or evaluation of an allocator. We use the term declared variable for the first kind of variable and the term designated variable for the second kind. (In the nomenclature of the Ada Language Reference Manual, an access value designates the variable it points to. All variables created by evaluation of an allocator, and no variables created by elaboration of a declaration, are pointed to by access values).

In informal discussions, the term task is often used interchangeably with task unit or task object. This can lead to confusion. The Ada Language Reference Manual defines precise and distinct meanings for these terms, which we adopt here:

- A task is a process, i.e., a set of actions performed in sequence, as if executed on its own virtual processor.

- A task object is a variable that corresponds to at most one task. (The Ada Language Reference Manual states that the task object designates the task, but we avoid that term to avoid confusion with the other sense in which the Reference Manual uses the term designates, to mean "points to"). Like any other Ada variable, a task object may be either declared or designated. A task object may come into existence before the corresponding task begins execution and remain in existence after that task has terminated.

- A task unit is program text consisting of a task declaration or task-type declaration and a task body.

The variables designated by the values of an access type are called a collection. In some implementations of the Ada language, a region of storage is set aside for the allocation of the variables in a given collection. We call this region of storage a collection region.

Section 2

REQUIREMENTS OF THE LANGUAGE

A number of storage management risks are inherent in the Ada language, regardless of the way in which a particular implementation manages storage:

- The number of simultaneously active invocations of a subprogram may not be known until runtime, but each invocation requires storage for its own instances of the variables declared in the subprogram body.
- The number of designated variables a program attempts to create may not be known until runtime, nor can it always be assumed that an attempt to create a new designated variable will succeed just because the total amount of storage available for that purpose is sufficient (fragmentation and internal management may interfere).
- The sizes of individual objects may not be known until runtime.
- The number of tasks in a program may not be known until runtime.

The sections below elaborate on each of these risks.

2.1 Procedure-Call Based Allocation of Storage

Storage for variables declared in an Ada subprogram must, in general, be allocated upon a call to that subprogram. While this is not an explicit requirement in the Ada Language Reference Manual, it is a consequence of certain language rules. These rules allow multiple invocations of a subprogram, each with independent instances of the variables declared in the subprogram, to be active at once. The number of simultaneous invocations that will occur cannot always be determined before the program runs. A subprogram call may raise Storage_Error if the amount of storage required for the new invocation is not available.

Multiple invocations may arise either from recursion or from multitasking. In the case of recursion, a subprogram, executing within a single task, is called by itself or some subprogram it has called before it has completed its original processing. In the case of multitasking, two tasks may independently execute the same subprogram concurrently and asynchronously. In either case, each invocation has its own copy of the variables declared in the subprogram body.

The variables created at the beginning of a subprogram invocation cease to exist upon return from that subprogram. Furthermore, within any one task, procedure invocations are properly nested. That is, if one procedure calls a second, the return from the second call occurs before the return from the first call. It follows that, if storage for variables declared in subprograms is deallocated as soon as the variables cease to exist, the subprogram-variable storage used by any one task follows a last-allocated first-deallocated discipline. The storage used by a compiler for internal bookkeeping related to a subprogram call--for example, storage for saving a return address--also follows this discipline.

The Ada language rules do not impose any one storage-management mechanism upon an implementation. However, the last-allocated first-deallocated discipline is conducive to the use of a last-in first-out stack by each task to allocate storage for variables declared in the bodies of subprograms invoked by that task. Storage locations that are on the stack would be considered allocated and storage locations that are above the top of the stack would be available for future allocation. Allocation would consist of incrementing the top-of-stack pointer, and deallocation would consist of decrementing the pointer.

2.2 Designated Variables

Designated variables are created by the evaluation of expressions called allocators. Evaluation of an allocator raises Storage_Error if sufficient storage for the object is not available. The number of times an allocator is evaluated depends on the paths taken through a program, including the number of times that loops containing allocators are repeated. The paths taken in a program, in turn, may depend upon input values. Thus the number of designated variables a program attempts to create cannot always be determined before the program runs.

While the storage allocated for declared variables follows a last-allocated first-deallocated discipline, the storage for designated variables does not. Storage for designated variables is allocated upon the evaluation of an allocator. It may be deallocated when the corresponding access type ceases to exist, when the programmer releases the storage by calling an instance of the predefined generic procedure Unchecked_Deallocation, or when the implementation determines that a designated variable has become inaccessible to the program because no accessible access value points to it.

A desirable property of a storage-management scheme is that there be no unusable storage, that is, that an attempt to allocate a block of a given size should fail only if the total amount of available storage is less than the size of the required block. Storage that obeys a last-allocated first-deallocated discipline can be implemented with a stack so that the entire amount of unallocated storage is available for allocation, even as a single block. Even if a region of storage does not obey a last-allocated first- deallocated discipline, if blocks of a uniform size are allocated and deallocated, then the total number of blocks that can be allocated at any one time is equal to the amount of available storage divided by the uniform block size.

If the designated type is an unconstrained composite subtype, different designated variables may be of different sizes. Unchecked deallocation of small variables may free small blocks of storage that are not contiguous, so that allocation of a large variable may be impossible even if the total amount of storage remaining in the collection region is sufficient. This phenomenon is called fragmentation of storage. Fragmentation is discussed further in Section 3.3.1.

2.3 Runtime Determination of Object Sizes

The size of an object in an Ada program may be determined at runtime and may be arbitrarily large. Thus the amount of storage needed by an object cannot always be determined, let alone allocated, before a program is run. Among the objects whose size may be determined at execution time are declared variables, designated variables, and compiler-generated objects.

Objects may be created by the elaboration of declarations like:

```
Data_List: array (1 .. N) of Float;
```

or

```
Matrix_Product: Matrix_Type (Left'Range(1), Right'Range(2)) of Float;
```

In the first of these declarations, N might be a subprogram parameter, for example, or a variable declared earlier and initialized by calling some function. In the second declaration, Left and Right might be subprogram parameters whose dimensions determine the size of the declared array Matrix_Product. The elaboration of an object declaration may raise Storage_Error if sufficient storage for the declared object is not available.

Similarly, the size of a designated variable may be determined by expressions inside the allocator whose evaluation creates the variable:

```
Data_List_Pointer := new Float_List_Type (1 .. N);

Matrix_Product_Pointer :=

    new Matrix_Type (Left'Range(1), Right'Range(2));
```

Different evaluations of the same allocator may create designated variables of different sizes. For example, the first statement above may be preceded by a procedure call to read the value of N from an input file. The size of the object specified in a particular evaluation of an allocator may determine whether or not that evaluation raises Storage_Error.

Array-valued expressions may specify array values whose length cannot be determined until the expression is evaluated during program execution. Often the compiler must generate new internal objects at runtime to hold the values of these expressions. Storage_Error may be raised if insufficient storage is available to create these objects. Examples include the aggregate

```
(1 .. N => 0.0)
```

whose length depends on the value of N and the catenation

```
A & B
```

whose length depends on the lengths of the arrays A and B. Sometimes the context in which such an expression appears determines the length of the result, but sometimes it does not. For example, the expression may appear in a return statement in a function whose result subtype is an unconstrained array type; or it may appear as a parameter to a subprogram whose corresponding parameter subtype is an unconstrained array type.

Though the above discussion concentrates on runtime determination of the size of an array, much of it also applies to the size of a record with discriminants. A discriminant may control which fields are present in the record, the size of record components that are themselves arrays, and the discriminants of record components that are themselves records (in some other type). Any of these properties may affect the size of the record. The declaration of an object in a record type or an allocator for a designated variable in such a type may contain a discriminant constraint, evaluated at runtime, determining the value of the record's discriminants and hence the required size of the record object. (Alternatively, an allocator may specify discriminant values by providing an expression, evaluated at runtime, giving the initial value of the entire record). An object in a record type with discriminants can sometimes be declared without a discriminant constraint. Such an object is unconstrained and must be large enough to accommodate any value in the record type. (Designated variables in a record type with discriminants are always constrained). Like array values, record values may be described by aggregates, but the discriminants in a record aggregate are always fixed and determinable before runtime. A function call may return a value of unknown size in a record type with discriminants.

2.4 Runtime Determination of the Number of Tasks

An Ada program may create new tasks as it executes. The number of tasks that will be created cannot always be determined before runtime. Thus it may be impossible to allocate in advance the storage areas that will be used by all tasks. The creation of a task object may raise Storage_Error if the implementation attempts to reserve some amount of storage for the corresponding task and that much storage is not available.

Tasks may be created by the elaboration of a task object declaration or the evaluation of an allocator for an object in a task type. If a recursive subprogram contains a declaration of a task object, a new task will be created at each level of recursion, but the depth of recursion may depend on runtime values. Similarly, each evaluation of an allocator may create a new designated task object, but the number of times an allocator is evaluated may depend on such factors as the number of time a loop is repeated. The number of tasks created when a declared or designated array of task objects is brought into existence depends on the size of the array. As explained in Section 2.3, this size may be determinable only at runtime.

Section 3

MANIFESTATIONS OF CURRENT IMPLEMENTATIONS

3.1 Global Storage-Management Strategies

This section discusses strategies for managing the storage space available to a main program. Each task in the program implicitly allocates and deallocates declared variables and compiler-generated working areas following a stack (i.e., last-allocated first-deallocated) discipline. In addition, the program may explicitly allocate and deallocate designated variables in an arbitrary order. The Ada language does not specify how or when an implementation assigns various regions of storage to play specific roles. There are a number of strategies available to implementations. We begin with a scheme in which all available memory is used as a stack, and then describe other schemes as variations on the stack scheme.

3.1.1 A Pure Stack Model

Firth [Fir85] describes a strategy based entirely on a stack. This strategy is facilitated by certain consequences of the Ada language definition:

- The maximum amount of storage available for a task's stack may be restricted to the amount reserved by a length clause of the form

  ```
  for task_type_name'Storage_Size use expression;
  ```

  or to a default maximum.

- The maximum amount of storage available for allocation of variables designated by an access type may be restricted to the amount reserved by a length clause of the form

  ```
  for access_type_name'Storage_Size use expression;
  ```

  or to a default maximum.

- If an access type is declared in a subprogram body or task body, then the access type ceases to exist upon return from the subprogram or termination of the task. All variables designated by that access type become inaccessible and their storage may be reclaimed.
- If a task object is declared in a subprogram body or a task body, then departure from that body cannot occur until the declared task terminates.
- If an access type designating a task type is declared in a subprogram body or task body, then departure form that body cannot occur until all allocated tasks pointed to by values in that access type have terminated.

In the stack model, all available storage is viewed as forming a single stack. Among the items pushed onto this stack, however, may be large blocks of storage used as "substacks" by tasks. Substacks may themselves contain substacks. Each activated task will have its own fixed-sized stack. A stack or substack may also contain large blocks of storage used as collection regions.

Certain storage can be allocated before the main program begins execution. This storage consists of:

- storage for each variable declared in a library package
- runtime information about each type and subtype declared in a library package
- for each access type declared in a library package, a fixed-size region for allocating variables that will be designated by that type
- for each task object declared in a library package, a fixed-size region for that task's stack

(By library package, we mean both the specification and body of a package that is not nested inside any other program unit, either physically or as a subunit). We think of this storage as being pushed onto an initially empty stack. This storage remains at the bottom of the stack for the duration of the program. The remainder of the stack (i.e., the remainder of free storage) is available for use by the implicit task that executes the main program.

(For the most part, the storage allocated before invocation of the main program can be allocated statically. In unusual cases, however, the size of a variable declared in a library package must be determined dynamically during the elaboration of library packages. For example, consider the following package:

```
with Calendar; pragma Elaborate (Calendar);

package Sales_History_Package is

     Sales_This_Year:

          array (1 .. Calendar.Month (Calendar.Clock)) of Float;

end Sales_History_Package;
```

The array Sales_This_Year will contain one component if the package is elaborated during January, two components if the package is elaborated during February, and so forth).

During the execution of any task, including the implicit task executing the main program, certain events may occur that entail allocation or deallocation of storage. Storage within a stack is allocated upon activation of the task or upon a subprogram call, and deallocated upon the return from a subprogram. Storage within a collection region is allocated upon the evaluation of an explicit allocator; it may be deallocated when an instance of the generic procedure Unchecked_Deallocation is called or when the implementation determines that a designated variable is no longer accessible.

An activation record is pushed onto a task's stack when the task calls a subprogram, and popped off the stack when the subprogram returns. This activation record includes the storage for any variables declared in the subprogram body and storage for runtime information about types and subtypes declared in the subprogram body. If the body of the called subprogram contains a task-object declaration, the activation record also includes a large block for use as a stack for that task. If the body of the called subprogram contains an access-type declaration, the activation record also includes a large block for the corresponding collection region. Upon the initial activation of a task, its stack contains an activation record corresponding to the declarations in the task body.

This scheme, together with constraints imposed by the Ada language, ensures that storage areas remain on the stack for as long as they are needed. Since a subprogram cannot complete its return until the tasks declared inside it have terminated, the storage allocated to serve as stacks for those tasks will not be popped off their containing stack before those tasks have terminated. Since designated variables become inaccessible upon return from the subprogram in which the corresponding access type is declared, the storage allocated for collection regions will not be popped off the containing stack until all variables in the corresponding collections are inaccessible.

Upon evaluation of an allocator, storage for the explicitly allocated object is taken from the collection region associated with the corresponding access type. If the allocated object is a task object, the storage allocated includes a large block of storage to be used as a stack for the corresponding task. If the corresponding access type was declared inside a subprogram body or task body, departure from that body cannot occur until the newly allocated task terminates. This guarantees that the activation record containing the collection region will remain in existence, providing a home for the task stack, as long as the task executes.

In the pure stack model, Storage_Error may be raised upon any of the following events:

- Elaboration of library packages before execution of the main program consumes all the available storage.

- When a task calls a subprogram, there is not enough room on the stack for the subprogram's activation record. The factors contributing to the size of the activation record are the amount of storage required by variables declared in the subprogram body, the amount of storage required for storing runtime information about types and subtypes declared in the subprogram body, the amount of storage required for collection regions associated with access types declared in the subprogram body, and the amount of storage required for the stacks associated with task objects declared in the subprogram body.

- When a task is activated, there may not be enough room on the task's stack for the first activation record. The size of this activation record is determined by the declarations in the task body in the same way that the size of an activation record for a subprogram call is determined by the declarations in the subprogram body.

- When an allocator is evaluated, there may not be a large enough block of storage in the corresponding collection region for the allocated object or, in the case of an allocated task, for the allocated task control block plus the new task's stack. Because of fragmentation, this may happen even if the total amount of storage remaining in the collection region exceeds the size of the block of storage to be allocated.

Given knowledge of the following factors, it may be possible to determine that an implementation based on a pure stack model will not raise Storage_Error:

- Total available storage

- The amount of storage allocated before execution of the main program

- The size of each subprogram's activation record, each access type's collection region, and each task's stack

- The subprogram calling graph

- The maximum number of designated variables to be allocated in each collection and the maximum size of each such variable

The size of each access type's collection region and each task's stack can be controlled by length clauses. The storage fragmentation problem could be avoided by additional implementation-defined pragmas dividing an unconstrained composite type's collection region to subregions for different subtypes. This would reduce flexibility in the use of the collection region's storage; but make it easier to guarantee the absence of Storage_Error without assuming a worst-case size for each designated variable.

3.1.2 A Global Heap

Rather than assigning all available storage to a stack, an implementation can use some storage as a stack and some as a heap. The stack is used for storage of relatively small objects that obey a last-allocated first-deallocated discipline. The heap may be used for larger regions of storage or for storage that may be deallocated in an arbitrary order.

It is not necessary to determine in advance how much storage is to be allocated to the stack and how much to the heap. The stack can be placed at one end of storage and the heap at the other. Both can then be allowed to grow towards the middle of storage as the need arises. Storage_Error is raised by an operation that would cause the stack and the heap to meet.

There are two obvious approaches for using a heap:

- o Rather than imposing a maximum collection-region size for each access type so that collection regions can be placed on a stack, an implementation may use a single heap for allocation of all designated variables.
- o Rather than pushing substacks for newly created tasks onto a larger stack, an implementation may allocate work space for newly created tasks from the heap.

Either of these variations on the pure stack approach might be adopted individually, or they could be combined in a number of ways:

- o Either a single heap or two separate heaps might be used for designated variables and for task work spaces. However, the use of two heaps plus a stack requires a priority partition of storage into at least two regions if runtime relocation of a heap or a stack is to be avoided. The first of these two regions might be for a fixed-size heap while the stack and the other heap grow towards the middle of the second region; or the first region might be for a bounded stack while the two heaps grow towards the middle of the second region.
- o For access-type declarations elaborated by a task (either in the task's body or in the body of a subprogram called by the task), storage for the corresponding collection might be taken from within the task's work space or from the "top level" heap for designated variables.
- o For designated task objects, the work space for the corresponding task might be taken from the storage allocated for the task object or from the "top level" heap for task work spaces.
- o For "dependent" task-object declarations elaborated by a "master" task, work spaces for the dependent tasks might be taken from the "top level" heap for task work spaces or from the work space of the master task.

It is not clear what benefit is derived by moving fixed-length substacks for declared task objects off the declaring task's stack. Whether placed in the heap or in an enclosing stack, the size of each individual substack remains bounded. Furthermore, substacks for declared task objects continue to be created and destroyed in synchronization with the expansion and contraction of the declaring task's stack. In Section 3.1.3, however, we consider implementations in which unbounded task stacks can be allocated. In such implementations, it is certainly worthwhile to allocate the storage for task stacks from someplace other than the main stack.

In contrast, there is an obvious benefit gained from the use of a single heap for all collections of designated variables: Rather than reserving specific regions of storage in advance for specific collections, we allow the runtime system to use available heap storage to satisfy current needs. There is a limit on the maximum amount of storage available for designated variables, but individual collections are allowed to grow and shrink within this limit. At points in a computation at which few variables are allocated within one collection, there is more storage available for other collections.

There are also disadvantages to the use of a single heap. One is the difficulty of exploiting the lack of an upper bound on each individual collection while still ensuring that the total amount of storage in use at any one time does not exceed the global maximum. The other is ensuring that, even if the global maximum is not exceeded, the available storage does not become so fragmented by allocation and deallocation of small designated variables that it becomes impossible to allocate a large designated variable.

To some extent, these disadvantages can be mitigated by length clauses. In conjunction with a single fixed-length heap, length clauses serve to place a lower bound, but not an upper bound, on the amount of storage reserved for a particular designated type. Let R_i be the amount of storage reserved by length clauses for collection i (zero if there is no length clause for collection i), let A_i be the amount of storage already allocated for collection i, and let H be the amount of storage in the heap. The amount of storage that must be reserved for future allocations to collection i, F_i, is then defined by:

$$F_i = \begin{cases} R_i - A_i & \text{if } A_i < R_i \\ 0 & \text{if } A_i > R_i \end{cases}$$

The invariant

$$(H - \Sigma A_i) > \Sigma F_i \qquad (1)$$

expresses the requirement that there always be enough unallocated storage available to allow each collection to grow to at least its reserved size. If we force an attempted allocation of size n to collection j to raise Storage_Error unless either

$$R_j - A_j > n \qquad (2)$$

or

$$(H - \Sigma A_i - \Sigma F_i) > n \qquad (3)$$

then every successful allocation will maintain the invariant (1). (Condition (2) asserts that the allocation can be performed using storage that has been reserved for the collection but is not currently allocated. Condition (3) asserts that, regardless of whether collection j has already been allocated more storage than was reserved for it, an allocation of size n can be performed while leaving enough storage for every collection to grow to its reserved size). The invariant (1) is true at the beginning of the program provided that

$$H > \Sigma R_i$$

(i.e., that the total amount of storage reserved does not exceed the size of the heap).

Though allocation is required to fail when both Condition (2) and Condition (3) are false, the truth of Condition (2) or Condition (3) does not guarantee that allocation will succeed. Rather, when both invariant (1) and either Condition (2) or Condition (3) holds, the total amount of unallocated storage is sufficient for the allocation to succeed and for invariant (1) to remain true afterwards; but this unallocated storage may be too fragmented for n contiguous units to be allocated.

There are two closely related approaches to resolving the fragmentation problem:

- Fragmentation can be prevented by avoiding deallocation, or least avoiding dependence on the availability of deallocated storage. The programmer would plan the use of storage as if deallocation had no effect. The implementation's allocation policy would not reduce the value of A upon deallocation from collection j. Consequently, the j implementation would raise Storage_Error if an allocation were attempted and not enough virgin storage remained to guarantee that all reserved storage could be allocated.

- An implementation might use length clauses to set aside contiguous blocks of storage for each collection, as in the pure stack model, but allow a collection to obtain storage from the shared heap once its own reserved storage has been exhausted. Then, even in the presence of deallocation, fragmentation will be restricted to the shared heap and to collection regions for unconstrained composite types in which variables of different sizes are to be allocated. Fragmentation in the shared heap is insignificant because the heap is used only for storage beyond the reserved amount. Fragmentation in a collection region for an unconstrained composite type is unavoidable unless the region is partitioned into separate subregions for each subtype of the type.

3.1.3 A Segmented Virtual Memory

Ideally, one would like to create an unbounded collection region each time an access-type declaration is elaborated and an unbounded task stack each time each time a task is activated. This ideal is, of course, impossible to achieve on a computer with finite memory. It can, however, be approximated on a paged, segmented architecture with a large virtual address space.

The approach is simply to create a new segment for each collection region and each task stack. Because the address space is paged, a large number of extremely large segments can be created without setting aside large blocks of physical memory: Physical memory is not assigned to a segment until storage on a page is actually allocated; contiguous pages in the segment can then be mapped to noncontiguous page frames in physical memory, making a priority partition of the physical address space unnecessary. Thus segments can grow gradually, independently of each other, as new designated variables are allocated and new activation records are pushed onto the stack. As physical address space is exhausted, some pages will be "paged out" onto a backing store and data references will be slowed down by an increasing number of page faults.

Storage_Error will not occur unless a segment expands to fill all its pages or the maximum number of segments is created. However, both the size of a segment and the maximum number of segments can be quite large: In the Multics operating system designed for the Honeywell 645 (originally General Electric 645) computer in the mid-to-late 1960's and early 1970's, virtual addresses consist of an 18-bit segment name and an 18-bit index into a segment, thus allowing 256K segments of 256K words [BCD72]. A segment used as a collection region is not likely to be exhausted unless several extremely large arrays are allocated. A segment used as a stack is not likely to be exhausted unless large arrays are declared in a deeply recursive subprogram (or in the case of infinite recursion, which reflects a programming error). The set of segments is unlikely to be exhausted unless extremely large numbers of tasks are created (e.g., by declaring a large array of tasks or by allocating designated task objects in a loop).

Today, this approach has limited applicability for flight-control applications. Current flight-control computers do not have paged, segmented memories with large virtual address spaces. Furthermore, the delays associated with page faults may be incompatible with real-time constraints. This problem can be mitigated by programs that make heavy use of multitasking, so that one task can continue useful work after another task encounters a page fault; indeed, it is precisely for programs with large numbers of tasks that the segment approach is most useful. The problem can be eliminated with backing stores based on large semiconductor memories rather than mechanical mass-storage devices.

3.1.4 A Linked Stack

In contrast to viewing all of storage as a stack and allocating small heaps (collection regions) on the stack, we can view all of storage as a large heap. A stack can be constructed within this heap as a linked list of dynamically allocated activation records. When an activation record is popped off a stack, the activation record's storage is returned to the heap. The stack for the task invoking the main program and the stacks for other tasks can be treated indistinguishably. No stack has bounded length and it is unnecessary to partition memory into stack space and heap space. So that an activation record of adequate size can be allocated before the corresponding declarative part is elaborated, variables whose size is determined at runtime can be allocated outside of the activation record, with fixed-size pointers to these variables placed inside the activation record.

When a program refers to entities declared in surrounding program units (e.g., to global variables), the relevant data is located in activation records other than the one on top of the stack. A linked stack does not provide direct access to data in lower activation records in terms of an offset from the top of the stack. However, a contiguous stack does not provide the required kind of direct access either, because the activation record for the relevant invocation of the surrounding unit may be arbitrarily deep in the stack. (Consider the following example:

```
procedure P is

    Global_Variable : integer;

    procedure Q is
    begin
       if Global_Variable > 0 then
          Global_Variable := Global_Variable - 1;
          Q;  -- recursive call
          ...
        else
          ...
        end if;
      end Q;

    begin  -- P
      Get (Global_Variable);
      Q;

     end P;
```

P calls the nested procedure Q which calls itself recursively some number of times that cannot be known beforehand. The stack contains an activation record for P below a number of activation records for Q, one for each recursive invocation. When an invocation of Q refers to Global_Variable, it must refer to the activation record for P).

A data structure called a display is used in contiguous stacks to provide direct access to activation records of interest lower in the stack. A display can also be used in a stack built as a linked list of activation records. A display is essentially an array of pointers to activation records for relevant invocations of textual surrounding program units. Displays can easily be maintained by placing a copy of the current display in each activation record, but another implementation, more efficient for deeply nested programs, requires only one display for each task stack.

When there are several parallel invocations of task units in the same surrounding program unit, each task stack's display contains a pointer to the activation record for the surrounding program unit. Thus each task stack can be seen as an independent extension of the stack containing that activation record, as if the stack, growing upward, grew several branches, each of which continued to grow on its own. This kind of stack is therefore sometimes called a cactus stack. [B&W73] discusses in depth a variation of the cactus stack, more general than is necessary to implement Ada.

The obvious drawback of a linked stack is that it induces fragmentation. Fragmentation occurs not only within the limited context of a single collection region, but in the entire space of available memory. The impact of fragmentation can be reduced by always trying to find allocation-record storage at one end of memory and other storage at the other end of memory. Then storage allocation and deallocation patterns will approximate those of the single stack/single heap implementation discussed in Section 3.1.2. Indeed, for one-task programs, in which all activation records follow a strict last-allocated first-deallocated discipline, the end of memory from which activation records are allocated will behave precisely as a stack.

The overhead of pushing and popping is likely to be higher for a linked stack than for a stack implemented with a stack pointer into a contiguous region. Upon pushing, code must be executed to search for a sufficiently large block of available storage. Upon popping, code must be executed to merge adjacent free areas. A general-purpose linked-stack allocation scheme may mimic a contiguous-stack scheme for single-task programs, but the added overhead will be incurred even when the blocks used to build a linked stack happen to reside contiguously in bottom-to-top order.

A length clause reserving a certain amount of storage for a collection or for a task stack may actually be counterproductive when a linked-stack implementation is used. The attempt to reserve storage may trigger a Storage_Error as soon as an access-type declaration is elaborated or a task is activated, simply because the amount of storage that has been reserved for the peak storage usage of the collection or the task is not currently available. In a pure-heap storage allocation scheme there is no reason to set specific storage aside until it is actually needed. Since storage currently in use by other collections or other tasks may be freed before this peak is reached, it may be possible to avert Storage_Error by not reserving storage ahead of time. Length clauses might help to control which tasks raise Storage_Error when the global storage resource is exhausted, but this capability is seldom useful.

It is not clear whether an implementation using linked stacks can choose to reject a length clause for 'Storage_Size, or nominally to accept it but to interpret in such a way that it has no practical effect. It is the intent of the Ada Language Reference Manual that an implementation accept all representation clauses that, in the words of Reference Manual Section 13.1, "can be handled simply by the underlying hardware." Furthermore, Reference Manual Section 13.2 clearly states that a length clause for the 'Storage_Size attribute "specifies the number of storage units to be reserved" for a collection or for the activation of a task. On the other hand, notes at the end of Section 13.2 (which are not technically part of the Ada standard but do serve to shed light on the intent of the language designers) suggest some flexibility:

What is considered to be part of the storage reserved for a collection or for an activation of a task is implementation-dependent. The control afforded by length clauses is therefore relative to the implementation conventions. For example, the language does not define whether the storage reserved for an activation of a task includes any storage needed for the collection associated with an access type declared within the task body. Neither does it define the method of allocation for objects denoted by values of an access type. For example, the space allocated could be on a stack; alternatively, a general dynamic allocation scheme or fixed storage could be used.

3.2 Treatment of Function Results in Unconstrained Subtypes

As noted in Section 2.3, the Ada language allows function results that are arrays of unknown size. Function results can also be records of unknown size. There are a number of ways to implement the return of such values to the function caller. To illustrate these alternatives, we shall consider the following contrived function:

```
function String_Plus_Reversal (S: String) return String is
     Reversal : String (S'Range);
begin
     for I in Reversal'Range loop
         Reversal (I) := S (S'Last + 1 - I);
     end loop;
     return S & Reversal;
end String_Plus_Reversal;
```

The function call String_Plus_Reversal ("ABCD") returns the string "ABCDDCBA", for example.

In languages like Pascal that do not allow composite function results, it is common to reserve a word at the very bottom of an activation record to hold the function result. Upon return from the function, all of the activation record except for its bottom word is popped from the stack, leaving the function result at the top of the stack, where it can be interpreted as part of the calling subprogram's activation record. An alternative approach for languages without composite function results is simply to place function results in a fixed register before returning. Neither of these approaches provides a complete solution for Ada function results.

For contiguous stacks, a variation that can accommodate variable-size function results is to place the function result at the top of the activation record. Once the length of the result-value expression S & Reversal is determined, the stack can be extended by that amount. The problem with this approach is that the calling subprogram cannot reuse the storage occupied by the returned function's activation record until it has moved the function result; otherwise the stack might expand past the top of the popped activation record and overwrite the function result. The calling subprogram could immediately copy the function result from the old top of the stack to the new top of the stack as soon as the returned function's activation record is popped, but this can be time-consuming if the function result is a large array.

Retaining the returned function's activation record on the stack even after the function has returned, but before the function result is used, can be expensive in terms of space. In the String_Plus_Reversal example, this activation record includes storage for the variable Reversal, which can be arbitrarily large. This problem can be magnified if the function call is part of a larger expression, like

```
String_Plus_Reversal (S1) &
   (String_Plus_Reversal (S2) & String_Plus_Reversal (S3))
```

because the results of the first two calls on String_Plus_Reversal cannot be disposed of until the result of the third call is available. Thus the "dead" activation records for the first two calls must remain on the stack until the third call is complete.

If a linked stack is used, the activation record must be allocated at the beginning of the function call and cannot simply be extended once the size of the function result becomes known. Rather, additional space for the function result must be allocated from the heap. A pointer to the function result might be placed in a fixed part of the activation record or in a register, as in typical Pascal implementations. (Pascal programmers sometimes write functions returning pointers to composite results because they cannot write functions returning composite values directly. An Ada compiler can perform the same transformation behind the scenes).

While the placement of a function result in the heap is essentially required for a linked stack, it is also an appropriate approach for a contiguous stack. A moderate storage-management overhead is incurred, but the penalties are not generally as severe as for sliding function results down the stack or retaining dead activation records on the stack. A compiler can easily distinguish cases in which indirect pointers to function results are required from those in which they are not, using pointers only for functions whose result subtypes are unconstrained composite types.

One strategy employed by optimizing compilers to minimize movement of data is to pass a target address for the function result as an implicit parameter to the function. The effect of the return statement is then to copy the result value directly to this address before returning control to the calling subprogram. In the assignment statement

```
Theta := Arc_Sin (R);
```

for example, the address of Theta would be passed along with the value of R to the invocation of Arc_Sin, and the code generated for the return statement in the body of Arc_Sin would copy the return value directly into Theta. In a context like

```
Theta := Arc_Sin (R) + Offset_Angle;
```

the address passed to Arc_Sin would be that of some compiler-generated temporary variable used in the subsequent addition. This approach can be adapted to the return of certain composite function results. Instead of passing just the address of the target location, the compiler would also pass its length. In the case of the assignment statement

```
Y := String_Plus_Reversal (X);
```

for example, the address and declared length of Y would be passed to the code for String_Plus_Reversal. The code for the return statement would use this length in copying the computed function result to its intended target, or raise Constraint_Error if the length of the computed result did not match the length of the intended target. (To the programmer, Constraint_Error would appear to be raised by the assignment statement containing the function call). Unfortunately, a function call can appear in a context where the length of the function result is unknown:

```
Put (String_Plus_Reversal (X));

return String_Plus_Reversal (S1) & String_Plus_Reversal (S2);
```

Since code for the return statement must be generated without knowledge of the context in which the function will be called, the encoding of the target length must include a special code meaning "unknown." In the body of a function with an unconstrained result subtype, the instructions generated for a return statement must check for this special code and return the function result in some other manner (e.g., by indirect reference to a location in the heap) when the code is encountered.

3.3 Non-FIFO Storage Allocation and Deallocation

Any Ada implementation must deal with the management of storage that does not obey a last-allocated first-deallocated discipline. If the pure stack implementation described in Section 3.1.1 is used, this problem arises in the management of individual collection regions. If the shared-heap approach of Section 3.1.2 is used, the problem arises in the management of the shared heap. If the linked-stack implementation of Section 3.1.4 is used, the problem arises in the management of all free storage.

Only in the case of an implementation for a segmented virtual memory can the problem be totally ignored: As physical memory space becomes scarce, pages containing only data eligible for deallocation will not have been recently used, and will be prime candidates for replacement. Virtual memory space becomes scarce only after extensive use, so it may be reasonable for some implementations to raise Storage_Error when this happens rather than trying to reuse virtual address space.

Algorithms for allocating storage and algorithms for deallocating it are closely intertwined, since both manipulate the same data structures. Section 3.3.1 explores several allocation/deallocation strategies. Section 3.3.2 explores implementation options in determining when allocation should take place.

3.3.1 Allocation and Deallocation

Allocation and deallocation of storage is a classic problem, explored in depth in Section 2.5 of [Knu73]. The fundamental difficulty that must be overcome is fragmentation. The amount of fragmentation resulting from a given sequence of allocations and deallocations depends on how available storage is selected and how freed blocks of storage are reunited with neighboring blocks of free storage.

Fragmentation is not a problem when all allocations from a given pool of storage are of the same size. Freed blocks can simply be placed on a linked list of available blocks, usually called the free list. Assuming that the uniform block size is large enough to hold a list link, the links in the free list can reside within the blocks themselves, so that no storage overhead is required. If the uniform block size is small (e.g., if an access type is declared for pointers to Boolean or Character values), the block size may be expanded to accommodate links. Alternatively, a list of bits, with one bit indicating the availability of each block, might be associated with each pool of storage.

A good Ada compiler can recognize pools of storage for which all allocations are of the same size, and use simple and efficient fixed-block-boundary methods for managing storage in those pools. For implementations based on collection regions, fixed lock boundaries are possible for any access type designating a noncomposite subtype or a constrained composite subtype. In a record type with variants, if there is not too much variance among the storage requirements of each variant, it may be worthwhile always to allocate the amount of storage required for the largest variant. The amount of storage wasted in this way may well be less than the amount that would be wasted by.

fragmentation or by complex storage-management data structures if different block sizes were used for different variants; time is certain to be saved by the resulting simplicity of allocation and deallocation. Since the space lost or saved and the desirability of various time-space tradeoffs will depend on the application, it would make sense for this option to be controlled by an implementation-defined pragma.

When blocks of various sizes are to be allocated or deallocated from the same pool, fragmentation can be overcome by compaction. If a situation arises in which the amount of available storage is sufficient to perform a requested allocation, but the amount of contiguous storage is not, blocks can be moved within the pool of storage, with pointers to the blocks adjusted accordingly, so that all available storage is united into one large block. Compaction is generally complex, expensive (often prohibitively so in applications with real-time constraints or limited storage space), and only marginally beneficial. The principal difficulty is in locating and changing all pointers. Since pointers may reside in various activation records on various stacks, or in a collection region or heap, location of the pointers is an awesome data-structure traversal problem. The problem is further complicated by the fact that some of the pointers to be updated may themselves be moved by the compaction. The time to locate and revise pointers can be reduced by implementing access values as pointers to pointers, with the direct pointers all stored together in a known location. Only the direct pointers need be updated, but extra storage space is required. Perhaps most importantly, Knuth [Knu73] observes, on the basis of experimental simulations, that

> ... the vast majority of cases in which the "first-fit" method runs out of room actually would soon thereafter run completely out of space anyway, no matter how much compacting and re-compacting is done. Therefore, it is generally not worth while to write a compacting program, except under special circumstances in connection with garbage collection....

(The "first-fit" method is a simple, noncompacting allocation strategy discussed below. Garbage collection is discussed in Section 3.3.2.).

Knuth discusses three allocation/deallocation schemes in Section 2.5 of [Knu73]. They are the first fit system, the best-fit system, and the buddy system.

The first-fit system maintains a linked list of free blocks of storage, in order of storage location. Each block on the free list contains a block size and a list link at the beginning of the block. Initially, the list contains one block consisting of the entire pool of storage. When a block of size n is needed, the list is searched for the first block whose size is at least n. If no such block can be found, Storage_Error is raised. Otherwise, n units of storage are allocated from the end of the block on the free list. The beginning of the block remains on the list with the size decremented by n, except that if n is decremented to zero, the block is removed from the list. Within the allocated block, a small amount of storage is reserved to hold the size of the block; this information will be required when the block is deallocated.

Two enhancements can reduce the time spent scanning past small free blocks in search of a block large enough to satisfy an allocation request. First, it is a good idea to make the free list circular and to begin each search for a sufficiently large free block just past the point on the list where the previous search left off. Otherwise, small blocks tend to accumulate near the front of the free list, slowing down subsequent searches. Second, if the blocks being allocated are relatively large (for example, if the items being allocated are task stacks rather than designated scalar variables), the search can be sped up by eliminating from the free list blocks of storage that are too small to be useful: If only a small amount of a free-list block would remain free after allocating the requested amount, the entire block is allocated and removed from the free list, sacrificing free storage space to reduce time.

When a block of storage is deallocated, it should be returned to the free list. First, however, it should be merged with any neighboring free blocks, so that previous fragmentation can be repaired to the extent possible. Knuth presents two schemes for returning storage to the free list, one of which is faster but requires the use of additional space within each reserved block.

The slower scheme is simply to search the free list (which is sorted in storage-location order) until the last block preceding the newly freed block is found. Using the length information in the freed block, it is possible to determine which if any of the two surrounding blocks on the free list are contiguous with it. The faster scheme uses an extra link field to maintain the free list as a doubly linked, but unsorted, list. It also reserves the first and last storage units of each block, whether it is free or allocated, for use as a boundary tag. The boundary tags are set to one code in free blocks and to another code in allocated blocks. When a block is freed, it is possible to check the storage units just before the beginning of the block and just after the end of the block--both of which are boundary tags of neighboring blocks--to determine which if any of the two neighboring blocks are free. Since the position of blocks in the free list need not be exploited to determine whether neighboring blocks are free, the freed block can be inserted arbitrarily at the beginning of the free list. The bi-directional list links can be used to remove any free neighboring blocks from the free list so they can be merged with the newly deallocated block before that block is returned to the free list.

The best-fit system is similar to the first-fit system. Rather than allocating storage from the first sufficiently large block that is found, however, the best-fit allocation algorithm always searches the entire free list and allocates storage from the smallest sufficiently large block. Either of the two deallocation schemes described above for first-fit allocation apply equally well to best-fit allocation. However, Knuth [Knu73] strongly recommends the first-fit scheme over the best-fit scheme:

> Historically, the best-fit method was widely use for several years; this naturally appears to be a good policy since it saves the larger available areas for a later time when they might be needed. But several objections to the best-fit technique can be raised: It is rather slow, since it involves a fairly long search; if "best fit" is not substantially better than "first fit" for other reasons, this extra searching time is not worth while. More importantly, the best-fit method tends to increase the number of very small blocks [because a block on the free list is chosen to minimize the amount of storage left over after allocation], and proliferation of small blocks is usually undesirable. There are certain situations in which the first-fit technique is demonstrably better than the best-fit method.... For these reasons the first-fit method can be recommended.

Knuth offers experimental evidence, based on simulation, to support this recommendation:

> In all experiments comparing the best-fit and first-fit methods, the latter always appeared to be superior. When memory size was exhausted, the first-fit method actually stayed in action longer than the best-fit method before memory overflow occurred, in most instances.

In the buddy system, the size of the available pool is a power of two and the size of an allocated block is always the lowest power of two greater than or equal to the requested size. If request sizes are evenly distributed, the ratio of storage allocated to storage requested tends to lie somewhere between 1.33 and 1.50. Different free lists are maintained for each power of two. Initially, all lists are empty except for the list corresponding to the size of the entire pool^{p-1}. If n^{p} units of storage are requested, where $2 \ <n<2$, and the free list for blocks of size 2^{p} is nonempty, then a block is taken from that list. If all free lists for block sizes greater than or equal to 2 are empty, Storage_Error is raised. Otherwise, a block is taken from the list for the smallest size larger than 2^{p} with a nonempty free list, the block is split in half, the two halves are placed on the free list for the next smaller size, and this process is repeated until the list for size 2^{p} becomes nonempty. When a block is split in two, the two resulting smaller blocks are called buddies. Any time a block with a free buddy is deallocated, the buddies are rejoined. The resulting larger block may itself have a free buddy, so this process is repeated as many times as possible. The block ultimately reconstructed is placed on the free list corresponding to its size. Adjacent free blocks of storage are never reunited unless they are buddies. Buddy-system allocation and deallocation algorithms can be made quite efficient by exploiting the binary representations of storage block addresses.

In most cases, the first-fit system using boundary tags is preferable to the buddy system, but in special circumstances the buddy system may be superior. A timing analysis, combined with a computer simulation, reveals that with random block sizes and random deallocation times, the execution time of the first-fit and buddy methods are comparable. However, the buddy system imposes an average overhead of from 33 to 50 percent on the amount of storage used, because of its requirement that the sizes of allocated blocks be powers of two. In applications for which allocation request sizes are naturally powers of two (or slightly smaller), however, this overhead disappears. Indeed, the buddy system tends to allocate a slightly larger portion of available storage before fragmentation causes it to fail (irrespective of how much of the allocated storage was actually requested). As allocation and deallocation patterns approach a last-allocated first-deallocated pattern, the performance of the first-fit system improves and the performance of the buddy system deteriorates. In a linked-stack implementation, for example, the allocations and deallocations of activation records for individual tasks each follow a last-allocated first-deallocated pattern, so the sequence of interleaved allocation and deallocation requests from all tasks may well approximate such a pattern.

In some applications, Storage_Error is undesirable and should be made extremely unlikely; in other applications, even a rare occurrence of Storage_Error is unacceptable, and should be made impossible. Unfortunately, Knuth indicates in Section 2.5 of [Knu73] that the necessary guarantees cannot be provided by algorithms for allocating and deallocating blocks of various sizes in arbitrary order from the same pool of storage:

> J.M. Robson has shown [JACM 18 (1971), 416-423] that dynamic storage allocation strategies which never relocate reserved blocks cannot possibly be guaranteed to use memory efficiently; there will always be pathological circumstances in which the method breaks down. For example, even when blocks are restricted to be of sizes 1 and 2, overflow might occur with memory only 2/3 full, no matter what allocation algorithm is used!

Neither is there much hope of proving an upper bound on the amount storage effectively lost through fragmentation. Knuth writes that "The mathematical analysis of these dynamic storage-allocation algorithms has proved to be quite difficult...," and adds, "...our knowledge of the performance of these algorithms is based almost entirely on Monte Carlo experiments."

Statistical results are useless in guaranteeing that a program will not exhaust storage. For applications for which Storage_Error is unacceptable, conservative restrictions must be imposed, involving both the implementation's global storage-management strategy and the program's use of designated variables. A strong restriction would be to regard non-FIFO deallocation as a null operation. That is, programs and implementations would be permitted to allocate blocks of different sizes from the same pool without following a last-allocated first-deallocated discipline only if the required amount of storage can be made available without returning any storage to the pool. This approach reflects the fact that fragmentation can render much of the returned storage useless. Somewhat weaker restrictions may be possible if we exploit publicly documented characteristics of an implementation, though this will reduce the portability of the program. For instance, most implementations could, in theory, guarantee that immediately after a block of size n is deallocated, a block of size n or less can be successfully allocated. Some programs might be written to exploit this property by repeatedly allocating and deallocating blocks of decreasing size.

For applications in which it is sufficient to make Storage_Error extremely unlikely, Knuth's Monte Carlo experiments provide some important insights:

- For the variety of distributions of block sizes and allocation lifetimes that Knuth explored, programs tend to reach a steady state in which the average amount of currently allocated storage is the product of the mean amount of storage requested per time unit and the mean number of time units between allocation and deallocation of a block.

- When the expected amount of storage allocated at steady state exceeds two-thirds of the available storage space, unfulfillable allocation requests usually arise, often before the full amount of available memory is actually needed.

- If all block sizes are small compared to the size of the pool from which they are allocated, the pool can become over 90 percent allocated without rejecting an allocation request; but if maximum block sizes exceed one-third of the pool size, allocation requests tend to become unfulfillable while less than 50 percent of the pool is allocated. Therefore, Knuth recommends a ratio of at least 10:1 between the pool size and the maximum block size.

- As the buddy system reaches a steady state, splitting and rejoining of buddies becomes rare. Free lists for all block sizes tend to remain nonempty.

- As allocation and deallocation patterns approach a last-allocated, first-deallocated pattern, the speed of the first-fit scheme improved considerably (with the free list containing just a few large items) but the performance of the buddy scheme deteriorated (with more need to split and rejoin buddies).

- When the buddy system was unable to honor an allocation request, storage was typically about 95 percent allocated (although the buddy system allocates between 1.33 and 1.50 times as much storage as is actually requested, on the average).

3.3.2 Control Over Deallocation

The Ada language makes three provisions for deallocating designated variables:

- A programmer may explicitly deallocate a designated variable by calling an instance of the predefined generic procedure Unchecked_Deallocation.

- The implementation may deallocate the entire collection of variables designated by values of a given access type when that access type ceases to exist.

- The implementation may deallocate an individual designated variable upon determining that that variable is no longer accessible.

The effect of calling an instance of Unchecked_Deallocation depends on the implementation's storage management scheme for the corresponding access type. We would typically expect the call to invoke one of the deallocation algorithms described in Section 3.3.1, so that the storage occupied by the deallocated variable can be used in later allocations. Depending on the implementation, the freed storage might or might not be united with neighboring free blocks of storage. An implementor might claim that one implementation of deallocation is simply to set the deallocated storage aside without making it available for reuse, i.e., to do nothing. However, Section 13.10.1 of the Ada Language Reference Manual seems explicit in its requirement that storage actually be recycled: "Unchecked storage deallocation of [a designated variable] is achieved by a call of [an instance of Unchecked_Deallocation]." Furthermore, such a call (with a pointer to some designated object) "is an indication that the object ... is no longer required, and that the storage it occupies is to be reclaimed."

An access type ceases to exist upon departure from the innermost subprogram body, task body, or block statement in which it is declared. If the implementation uses collection regions, this is equivalent to freeing the storage occupied by the corresponding collection region. If the collection region is stored on a stack, liberation of the storage is a side effect of popping the top activation record off the stack.

Because a designated variable can only be referred to in terms of the access value pointing to it, there is no way to refer to the variable once that access value is no longer stored in an accessible location. Accessible locations include not only declared variables, but also designated variables pointed to by access values in other accessible locations. Designated variables may be deallocated without any noticeable effect upon the program as soon as they are inaccessible and, in the case of designated variables that are task objects (or that contain task objects as subcomponents) the corresponding tasks have terminated. The identification and deallocation of inaccessible designated variables is called garbage collection. Section 4.8, paragraph 7, of the Ada Language Reference Manual allows, but does not require, garbage collection.

Garbage collection can be a complex process. In the case of a recursive type (e.g., a type whose objects contain pointers to other objects in that type), the determination that a given object has become inaccessible requires the determination that all other designated objects containing pointers to the given object have themselves become inaccessible. Thus garbage collection begins with a marking phase in which all chains of accessible pointers are traversed and accessible designated variables are marked by setting a special bit set aside for this purpose. This is followed by a second phase in which marked variables are deallocated. Free storage may be compacted during this phase by relocating those designated variables that remain accessible.

Garbage collection can be quite time consuming, bringing normal processing to a halt. A sustained cessation of processing can be avoided if garbage collection is performed by a background process whenever processing time becomes available. However, even this may be impractical in some real-time applications. Furthermore, the effectiveness of garbage collection in making more storage available cannot be guaranteed. Therefore, the Ada language includes a pragma, Controlled, that may be used to ensure that garbage collection will not take place for a given collection.

3.4 Efficient Data Representation

Implementations may vary not only in their strategies for placing objects in the available storage, but also in their determination of the size of an object. Some implementations are more space-efficient than others. Some implementations provide the programmer with more control than others over the amount of storage used for objects of a given type.

Lower bounds on object sizes for a scalar type are logical consequences of the type definition. For example, an enumeration type with n values cannot be represented in fewer than log n_2 bits; an integer type with range L .. R cannot be represented with fewer than $\log_2$ (R-L+1) bits. Similar lower bounds for representations of floating-point and fixed-point types can be derived in terms of the number of model numbers in the type, since each model number requires a unique representation.

Nonetheless, an implementation may use more than the logical minimum amount of storage for a type. For example, if a machine uses eight-bit, sixteen-bit, and 32-bit representations for integer data, then a compiler might use a full eight bits for an enumeration type logically requiring only five bits, or sixteen bits for an integer type logically requiring only nine bits. Use of a smaller amount of storage might require the introduction of loading, masking and shifting instructions that would considerably slow down the execution of the object code.

The programmer may influence the size of objects in a scalar type in two ways. First, the Optimize pragma advises the compiler to pursue code-generation strategies that save time at the cost of space, or vice versa. However, the compiler has complete freedom in determining how this advice is to be applied, if at all. Second, a representation clause of the form

```
    for type_name'Size use static_expression;
```

imposes a mandatory upper bound on the number of bits that may be used to represent objects in the specified type. Section 13.1, paragraph 10, of the Ada Language Reference Manual states, "An implementation may limit its acceptance of representation clauses to those that can be handled simply by the underlying hardware," and requires a compiler to issue an error message for any representation clause it cannot accept. Until now, the term "simply" has been liberally interpreted, giving compilers a considerable degree of freedom in rejecting representation clauses. This has been a source of dissatisfaction among developers of embedded-computer programs, and compiler writers face strong market pressures to provide full support for representation specifications.

The storage space required for a composite type depends not only on the size of the components, but also on the amount of unused storage between components and the amount of internal information stored with the composite object.

Unused storage may be used to align array or record components on appropriate storage-unit boundaries, so that access to them will be fast. Like the size of a scalar object, the amount of unused storage in a composite object may be influenced by an Optimize pragma. Another pragma, the Pack pragma, specifically advises the compiler to minimize unused storage in the representation of a particular composite type. (The Optimize pragma potentially affects all types). Representation clauses of the form

```
    for type_name'Size use static_expression;
```

--if accepted by the compiler--can specify a mandatory upper bound on the size of objects in a composite type. Constraints on the maximum size of a composite type may affect the amount of unused storage between components, but they do not impose constraints on the maximum size of the components themselves. Rather, the size of the components is determined beforehand and used to ascertain whether the representation clause for the composite type can be accepted.

The size of a composite object may also be affected by the presence of internal information. The representation of an array in an unconstrained subtype must, in the general case, include enough data to determine the lower and upper index bounds in each dimension of the array. Some optimizing compilers may be clever enough to discover that all the bounds information needed for certain individual arrays in the unconstrained subtype can be ascertained at compile time, and to remove this information from the runtime representations of those arrays. However, this condition is difficult to detect and its application greatly complicates code generation. No runtime information need be stored with objects in a constrained array or record subtype. In a record type with discriminants, all the required runtime information can be deduced from the values of the discriminants themselves, though an implementation might store redundant information in the record to speed up certain runtime checks. (There may be additional runtime information relating to the composite type itself, such as runtime constraints on component values. However, this information need be stored only once per type, most likely in the activation record corresponding to the unit containing the type declaration.)

One special case worth noting is the storage allocation for an unconstrained record, one of whose components is an array with bounds controlled by a discriminant. Here is an example:

```
type Varying_String_Type (Maximum_Length: Positive := 80) is
   record
      Current_Length : Natural := 0;
      Contents       : String (1 .. Maximum_Length);
   end record;

VS : Varying_String_Type;
```

Since the variable VS is declared without a discriminant constraint, it is an unconstrained record. Therefore, this variable must be capable of holding any value in Varying_String_Type. Since the discriminant VS.Maximum_Length may be any value of subtype Positive, the array VS.Contents may, at times, consist of as many as Positive'Last characters. (For typical implementations, Positive'Last is the largest number that can be represented in a signed word). The usual implementation of an unconstrained-record declaration is to allocate enough space in the current activation record for the largest value that the record can assume. In the case of VS, this is almost certain to raise Storage_Error. However, an alternative is to place only a pointer to the unconstrained record in the activation record and to allocate enough space in the heap for the record's current contents. (VS.Contents'Length is initially 80). When VS is assigned a larger Varying_String_Type record size, the heap storage holding the old value is deallocated and a new block, large enough to hold the record's new contents, is allocated. Alternatively, an implementation might allocate only the Contents component of VS in the heap; in the activation record, the implementation would place a Varying_String_Type record containing the values of the other components and a pointer to the Contents component.

In addition to scalar types and composite types, the Ada language has access types and task types. Access values are typically implemented as machine addresses, leaving no flexibility in the size of an access type's objects. However, one can also envision access values for uniformly-sized collections represented compactly as numbers that, when multiplied by the uniform size of each designated variable in the collection, would provide an offset from the start of the collection region. (Essentially, the collection region would be treated internally as an array of fixed-sized components, and access values would be indices into this array). The representation of a task object depends entirely on the implementation of multitasking; there is no lower bound on the size of a task object inherent in the Ada language. In some implementations all task objects may have a uniform size, while in other implementations task objects of different types might potentially have different sizes.

3.5 Compiler Storage-Management Aids

A compiler can help to lower the risk of Storage_Error not only by the way it generates code, but by the tools it provides to the programmer to analyze and control the use of storage. These tools include information to help predict the availability of sufficient storage, implementation-defined pragmas to select various storage-management options, and runtime subprograms to obtain information about current storage use.

A programmer can most easily guarantee that a program will not raise Storage_Error if the program uses a simple storage-management scheme with easily understood behavior. One characteristic of a storage-management scheme that makes it easily understood is early binding: Storage blocks for specific purposes are fixed in size and reserved as soon as possible, as in the allocation of collection regions and substacks in the pure stack model. Early binding guarantees that a known amount of storage is available for a given purpose, but it also guarantees that the storage will not be available for other purposes. Thus simplicity and predictability come at the expense of flexibility in storage use. Early binding causes Storage_Error to occur after fewer allocations than in the average case using late binding, but provides certainty that Storage_Error will not be raised before that many allocations have been performed. The following graph compares the probability of Storage_Error after some number of allocations, using early and late binding:

Early binding is only useful, of course, if the programmer understands the guarantees that it provides. Therefore, the compiler should come with complete documentation describing how storage is set aside. In particular, the documentation should explain the effect of length clauses for 'Storage_Size. According to the Ada Language Reference Manual, the meaning of such a length clause is implementation-dependent. In some cases the storage reserved for a collection region may include allocation control information and in other cases it may not, for example.

A compiler can help programmers determine the storage-management characteristics of their programs by providing information about storage representations. Invoked with appropriate options, a compiler might, for example, report the value of the 'Size attribute for each type declared, as well as the number of bits used to represent objects in anonymous types. For arrays in unconstrained array types, the reported length could be a formula in terms of the length of the array in each dimension. For record types with discriminants, different lengths could be reported for each variant; for a variant containing a component with its own index or discriminant constraint, the reported length could again be a formula. Even more helpful would be a report on the size of a subprogram's or task body's activation record, including both programmer-declared and compiler-generated objects. This size would be expressed as a formula in terms of the nonstatic values appearing in index and discriminant constraints. We are aware of no compiler that currently provides this information.

A compiler can help programmers control the storage-management characteristics of their programs by providing implementation-defined pragmas specifying elements of a storage-allocation strategy. We have already mentioned two implementation-defined pragmas that might be useful:

- a pragma to divide an unconstrained composite type's collection region to subregions for subtypes of different sizes, thus reducing the flexibility with which the collection region's storage can be used, but avoiding fragmentation
- a pragma stipulating that all designated variables in a record type with variants are to be stored in blocks of the same length, thus forcing unused storage to be allocated for shorter variants, but avoiding fragmentation and simplifying allocation and deallocation algorithms

Other implementation-defined storage-management pragmas can easily be envisioned.

In addition to compile-time support, an implementation might provide runtime support to determine and control the current state of the storage pool. This runtime support would take the form of a set of implementation-defined subprograms. These subprograms might include:

- a function returning the amount of free storage available in some storage pool such as a collection region or a global heap
- functions returning measures of the fragmentation in some storage pool, for example the average size of a free block, the standard deviation in the sizes of free blocks, or the number of contiguous blocks at or above a given size
- procedures explicitly invoking garbage collection in specified collections or compaction of specified storage pools.

- procedures expanding the size of the global heap, or of regions that have been allocated for specific purposes

SIGAda's Ada Runtime Environments Working Group (ARTEWG) is compiling a catalog of common interfaces through which Ada programs can control various aspects of the runtime environment. The first release of that catalog [ARTE86] does not include any storage-management interfaces, but storage management and garbage collection are both specifically listed as topics for future consideration.

REFERENCES

[ARTE86] A catalog of interface features and options for the Ada runtime environment. Release 1.0. Ada Runtime Environment Working Group, Interfaces Subgroup, Association for Computing Machinery Special Interest Group on Ada, October 1986

[B&W73] Bobrow, Daniel G., and Wegbreit, Ben. A model and stack implementation of multiple environments. Communications of the ACM 16, No. 16 (October 1973), 591-603

[BCD72] Bensoussan, A., Clingen, C.T., and Daley, R.C. The Multics virtual memory: concepts and design. Communications of the ACM 15, No. 5 (May 1972), 308-318

[Bra83] Bray, Gary. Implementation implications of Ada generics. Ada Letters 3, No 2 (September-October 1983), 62-71

[Fir85] Firth, Robert. Ada code generation strategies. ACM SIGAda Summer Meeting, Minneapolis, Minnesota, August 1985 (oral presentation.)

[Knu73] Knuth, Donald E. The Art of Computer Programming. Volume 1, Fundamental Algorithms, 2nd edition. Addison-Wesley, Reading, Massachusetts, 1973