of cells that can be placed on the list of available space. In addition to this problem, there is the cost of the additional space of the cell marks, which need only be maintained on the last block allocated to a cell. However, these problems are not as serious as they sound, for two related reasons. First, memory is plentiful on most contemporary computers. Second, all larger computers use virtual memory, which makes their large memories appear much larger than they actually are.

Both the marking algorithms for the garbage collection method and the processes required by the reference counter method can be made more efficient by use of the pointer rotation and slide operations that are described by Sutherland (1982).

**Variable-Size Cells** Managing a heap from which variable-size cells are allocated has all the difficulties of managing one for single-size cells, but also has additional problems. Unfortunately, variable-size cells are required by most programming languages. The additional problems posed by variable-size cell management depend on the method used. If garbage collection is used, the following additional problems occur:

- The initial setting of the indicators of all cells in the heap to indicate that they are garbage is difficult. Because the cells are different sizes, scanning them becomes a problem. One solution to this is to require each cell to have the cell size as its first field. Then the scanning can be done, although it takes slightly more space and somewhat more time than its counterpart for fixed-size cells.
- The marking process is nontrivial. How can a chain be followed from a pointer if there is no predefined location for the pointer in the pointed-to cell? Cells that do not contain pointers at all are also a problem. Adding a system pointer to each cell will work, but it must be maintained in parallel with the user-defined pointers. This adds both space and execution time overhead to the cost of running the program.
- Maintaining the list of available space is another source of overhead. The list can begin with a single cell consisting of all available space. Requests for segments simply reduce the size of this block. Reclaimed cells are added to the list. The problem is that before long, the list becomes a long list of various-size segments, or blocks. This slows allocation because requests cause the list to be searched for sufficient large blocks. Eventually, the list may consist of a large number of very small blocks, which are not large enough for most requests. At this point, adjacent blocks may need to be collapsed into larger blocks. Alternatively to using the first sufficiently large block on the list can shorten the search but require the list to be ordered by block size. In either case, maintaining the list is additional overhead.

If reference counters are used, the first two problems are avoided, but the available-space list maintenance problem remains.

### SUMMARY

The data types of a language are a large part of what determines that language's style and use. Along with control structures, they form the heart of a language.

The primitive data types of most imperative languages include numeric, character, and Boolean types. The numeric types are often directly supported by hardware.

The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs.

Arrays are part of most programming languages. The relationship between a reference to an array element and the address of that element is given in an access function, which is an implementation of a mapping. Arrays can be either static, as in Fortran 77; fixed stack-dynamic, as in C functions; stack-dynamic, as in Ada blocks; fixed heap-dynamic, as with Java's objects; or heap-dynamic, as in Perl's arrays. Most languages allow only a few operations on complete arrays.

Records are now included in most languages. Fields of records are specified in a variety of ways. In the case of COBOL, they can be referenced without naming all of the enclosing records, although this is messy to implement and harmful to readability. In Java, records are supported in the class construct.

Unions are locations that can store different type values at different times. Discriminated unions include a tag to record the current type value. A free union is one without the tag. Most languages with unions do not have safe designs for them, the exception being Ada.

Pointers are used for addressing flexibility and to control dynamic storage management. Pointers have some inherent dangers: Dangling pointers are difficult to avoid, and garbage is difficult to collect.

Reference types, such as those in Java, provide heap management without the dangers of pointers.

The level of difficulty in implementing a data type has a strong influence on whether the type will be included in a language. Enumeration types, subrange types, and record types are all relatively easy to implement. Arrays are also straightforward, although array element access is an expensive process when the array has several subscripts. The access function requires one addition and one multiplication for each subscript.

Pointers are costly to implement if they are used for dynamic storage management and if steps are taken to avoid dangling pointers. Heap management is relatively easy if all cells have the same size, but becomes more complicated with variable-size cell allocation and deallocation.
14. Define row major order and column major order.
15. What is an access function for an array?
16. What are the required entries in a Java array descriptor, and when must they be stored (at compile time or run time)?
17. What is the purpose of level numbers in COBOL records?
18. Define fully qualified and elliptical reference to fields in records.
19. Define union, free union, and discriminated union.
20. What are the design issues for unions?
21. Are the unions of Ada always type checked?
22. What are the design issues for pointer types?
23. What are the two common problems with pointers?
24. Why are the pointers of most languages restricted to pointing at a single type object?
25. What is a C++ reference type and what is its common use?
26. Why are reference variables in C++ better than pointers for formal parameters?
27. What advantages do Java and C# reference type variables have over the pointers in other languages?
28. Describe the lazy and eager approaches to reclaiming garbage.
29. Why wouldn’t arithmetic on Java and C# references make sense?

1. What is a descriptor?
2. What are the advantages and disadvantages of decimal data types?
3. What are the design issues for character string types?
4. Describe the three string length options.
5. Define ordinal, enumeration, and subrange types.
6. What are the advantages of user-defined enumeration types?
7. In what ways are the user-defined enumeration types of C# more reliable than those of C++?
8. What are the design issues for arrays?
9. Define static, fixed stack-dynamic, stack-dynamic, fixed heap-dynamic, and heap-dynamic array. What are the advantages of each?
10. What array initialization feature is available in Ada that is not available in other common imperative languages?
11. What is an aggregate constant?
12. What array operations are provided specifically for single-dimensioned arrays in Ada?
13. What are the differences between the slices of Fortran 95 and those of Ada?

1. What are the arguments for and against representing Boolean values as single bits in memory?
2. Why does a decimal value waste memory space?
3. COBOL uses several different methods of storing decimal numbers. Explain the format and purpose of each.
4. VAX minicomputers use a format for floating-point numbers that is not the same as the IEEE standard. What is this format, and why was it chosen by the designers of the VAX computers? A reference for VAX floating-point representations is Sebesta (1991).
5. Compare the tombstones and locks-and-keys methods of avoiding dangling pointers, from the points of view of safety and implementation cost.
6. What disadvantages are there in implicit dereferencing of pointers, but only in certain contexts? For example, consider the implicit dereference of a pointer to a record in Ada when it is used to reference a record field.
7. What significant justification is there for the → operator in C and C++?
8. The unions in C and C++ are separate from the records of those languages, rather than combined as they are in Ada. What are the advantages and disadvantages to these two choices?
9. Multidimensional arrays can be stored in row major order, as in C++, or in column major order, as in Fortran. Develop the access functions for both of these arrangements for three-dimensional arrays.
10. In the Burroughs Extended ALGOL language, matrices are stored as a single-dimensional array of pointers to the rows of the matrix, which are treated as single-dimensioned arrays of values. What are the advantages and disadvantages of such a scheme?
11. Analyze and write a comparison of C's malloc and free functions with C++'s new and delete operators. Use safety as the primary consideration in the comparison.
12. Analyze and write a comparison of using C++ pointers and Java reference variables to refer to fixed heap-dynamic variables. Use safety and convenience as the primary considerations in the comparison.
13. Write a short discussion of what was lost and what was gained in Java's designers' decision to not include the pointers of C++.
14. What are the arguments for and against Java's implicit heap storage recovery, when compared with the explicit heap storage recovery required in C++?
15. What are the arguments for the inclusion of enumeration types in C# although they are not in Java?
16. What would you expect to be the level of use of pointers in C#? How often will they be used when it is not absolutely necessary?
17. Make two lists of applications of matrices, one for those that require jagged matrices and one for those that require rectangular matrices. Now, argue whether just jagged, just rectangular, or both should be included in a programming language.
18. Compare the string manipulation capabilities of the class libraries of C++, Java, and C#.

3. Write a program that does matrix multiplication in some language that does subscript range checking and for which you can obtain an assembly language or machine language version from the compiler. Determine the number of instructions required for the subscript range checking and compare it with the total number of instructions for the matrix multiplication process.
4. If you have access to a compiler in which the user can specify whether subscript range checking is desired, write a program that does a large number of matrix accesses and times their execution. Run the program with subscript range checking and without it, and compare the times.
5. Write a simple program in C++ to investigate the safety of its enumeration types. Include at least 10 different operations on enumeration types to determine what incorrect or just silly things are legal. Now, write a C# program that does the same thing and run it to determine how many of the incorrect or silly things are legal. Compare your results.
6. Write a program in C++ or C# that includes two different enumeration types and has a significant number of operations using the enumeration types. Also write the same program using only integer variables. Compare the readability and predict the reliability differences between the two programs.
7. Write a C program that does a large number of references to elements of two-dimensioned arrays, using only subscripting. Write a second program that does the same operations but uses pointers and pointer arithmetic for the storage mapping function to do the array references. Compare the time efficiency of the two programs. Which of the two programs is likely to be more reliable? Why?
8. Write a Perl program that uses a hash and a large number of operations on the hash. For example, the hash could store people's names and their ages. A random number generator could be used to create three-character names and ages, which could be added to the hash. When a duplicate name was generated, it would cause an access to the hash, but not add a new element. Rewrite the same program without using hashes. Compare the execution efficiency of the two. Compare the ease of programming and readability of the two.

PROGRAMMING EXERCISES

1. Design a set of simple test programs to determine the type compatibility rules of a C compiler to which you have access. Write a report of your findings.
2. Determine whether some C compiler to which you have access implements the free function.