example, changes in government regulations affect the information that a financial system stores and reports.

**Programming system complexity**

Early programs were written in assembly language, usually by a single programmer. As hardware capabilities increased, the tasks computers were applied to became more complex and varied. Higher-level languages provided facilities for larger, more structured programs, and the issue of dealing with complexity became crucial in software development. As the size of a software system increases, the total complexity increases faster than the number of components due to interdependencies among those components. For example, consider a system with ten modules. If another module is added to the system, the total complexity increases not only by the additional ten percent within that module, but also by the complexity of the new module's interactions with up to ten other modules.

For very large software systems consisting of millions of lines of code, it is impossible for one designer to understand the system in its entirety. Therefore, software engineers must also deal with stating the system's requirements and describing its structure, and of communication among the members of the development team. For example, Brooks observed that n programmers working on a project do not complete it in $\frac{1}{n}$ the time necessary for one programmer. In addition, large systems often include the complexity associated with providing fault tolerance, concurrent access, usability, and maintainability. Complexity is also increased because software systems are frequently built from the smallest possible units, namely individual lines of code, rather than from larger components.

**Managing complexity**

We can see from this discussion that two central issues in software design are deciding how much of the problem domain complexity must be reflected in the software system, and determining how to structure the system to reduce its complexity, while maintaining its correspondence with the problem domain system. Human cognition uses a number of strategies for dealing with the complexity found in the environment, and these techniques are just as helpful in controlling software complexity. They allow us to handle more information, given the limits of human short-term memory, by structuring and grouping information, hiding complexity, and allowing us to understand the system at a higher, more abstract level. We will see in this chapter that these processes are also essential components of the object-oriented model of computation.

*Decomposition* is the process of dividing a complex system into loosely coupled components that can be considered individually. This partitioning reduces comprehension of the system to the smaller, independent problems of understanding each component and the interactions among components. We also use the inverse process, *composition*, when we recognize that a group of components makes up a higher-level component. For example, to understand the operation of a car, a mechanic divides it into the drive train, the electrical system, the suspension system, the braking system, and so on, and also decomposes each system into its components when necessary. We often find it useful to decompose a system's components, and composite components are often components of other composites, so the structure of a system is described by a hierarchy of whole-part and aggregation relationships. Such composition hierarchies appear in all complex physical, biological, and social systems.

Software designers use decomposition and composition of both processing and objects in developing an application. For example, in a functional decomposition, a subprogram represents a particular subtask, and that subprogram is composed of individual statements. We will see that an object-oriented decomposition of an application organizes the system in terms of classes of objects and their behavior, rather than according to tasks and subtasks.

*Classification* is the process of recognizing commonalities and differences among individual entities to create categories or archetypes. Grouping entities into categories and then dealing with the categories controls complexity by reducing the number of concepts necessary to describe a situation. The resulting concepts can also be used to describe other systems with similar components. For example, we all have a concept of the category "chair" that helps us deal effectively with a wide range of such objects, including new kinds of chairs we have never encountered. We often find that categories have subcategories with additional or differing properties and behavior, resulting in a hierarchy of classifications. Like composition hierarchies, classification hierarchies appear in physical, biological, and social systems. For example, the taxonomy of plants and animals is a classification hierarchy. Software designers use classification when determining the subprograms and types necessary in an application. That is, a subprogram represents a category of tasks, and a type describes a class of objects.
Abstraction is the process of extracting the relevant information about a category, entity, or activity, and ignoring the inessential details. (We also use the term "abstraction" to refer to the constructs that result from this process.) By not paying attention to unimportant details, we simplify our view of the category or activity. For example, the driver of a car only deals with using the gas pedal to speed up and the brake pedal to slow down, rather than the complex chain of mechanics triggered by those devices. As we will see in this section, software designers use subprograms and modules to create abstractions of process, and types to define abstractions of structure.

There are many ways to determine the necessary categories and abstractions and use them to decompose a complex system. Doing so is not easy, and different people often create different representations for the same situation. The measure of whether one organization is better than another is its usability and predictive power for the task for which it was created. That is, we judge the quality of an abstraction according to whether it represents the appropriate properties from the perspective of using of the abstraction. For example, the classifications, decompositions and abstractions that are effective for the design of a car and for an auto mechanic are quite different. In human cognition, categories and abstractions are prototypes that are not fixed, but are developed as experience dictates.

4.2 Abstraction in computing

Programming abstractions

Some basic abstractions were created early in the development of programming languages. Assembly languages used symbolic names for operations and locations to hide the details of the machine language encoding of instructions and the actual storage addresses of data. FORTRAN's algebraic expression syntax hid the use of temporary registers and individual machine operations to implement expression evaluation. FORTRAN also allowed the programmer to ignore the details of the encoding of real numbers and the manipulation of that representation. ALGOL hid the details of control flow and branching with the conditional and iterative control structures (rather than using goto's and statement labels as in assembly language). As we discussed in section 1.1, control structures also make it easier to decompose a program, and to do so hierarchically. Like all programming abstractions, each of these language features decreases the amount of detail we must deal with, and each represents a step away from the machine architecture and toward the problem domain.

Higher-level programming languages have implemented a progression of constructs for programmer-defined abstractions, beginning with abstractions of operation such as subprograms and procedural modules. Later languages such as Pascal provided constructs for abstractions that structure information, such as enumeration and record types. Contemporary programming languages support abstract data types and classes that encapsulate both structure and processing, and provide information hiding. We will discuss each of these abstractions in more detail in this section.

Software designers use programming abstractions as components in the decomposition of the system, and these abstractions and their interactions provide the designer with a higher-level view of the structure and operation of the system. For example, in a functional decomposition, an application consists of a set of subprograms or procedural modules, and a diagram of the system structure is a graph with the subprograms as vertices and invocations (and argument passing) as edges. For the programmer, an abstraction such as a subprogram or type extends the language to reflect the problem domain or the design of the application. The use of abstractions simplifies programs, improves readability, and reduces the number of errors by revealing the logical structure of the program. A subprogram or abstract data type written by one programmer can be used by others, who do not need to understand its internal operation. In fact, a software development organization can maintain a library of abstractions to reuse in applications, thereby reducing the coding effort and increasing programmer productivity.

Determining abstractions

Designers recognize the necessity for an instance of a particular kind of abstraction, e.g. a subprogram or type, by either a top-down decomposition of the structure of the application or a bottom-up composition

---

28 Modules, e.g. in Modula-2 and C, were originally created to encapsulate a group of related subprograms. Although, as we will see in section 4.4, modules can be used to encapsulate the operations and structure of a type, we will use the term "module" in its original procedural meaning, and distinguish between "process-oriented" modules and "object-based" abstract data types.
of lower-level elements. Top-down design proceeds from the general to the specific, while bottom-up design leads from the specific to the general. These processes provide two complementary perspectives on the nature of the abstraction.

In top-down decomposition, we recognize that a computation or a data structure is a logical unit or that the application requires several instances of it, usually from analysis of the system requirements or problem domain. We determine the abstraction’s external interface from those uses, and then design an appropriate internal implementation. For example, a program that provides user commands for managing a hierarchical file system (like the UNIX or DOS command shell) will have several commands whose argument is a path in the directory structure. The designer recognizes that the system needs a subprogram that takes a path and returns a reference to the specified file or directory, and that each subprogram that must evaluate a path will invoke this subprogram.

In bottom-up composition, we recognize that there are multiple occurrences of a series of operations or of a data structure. (We have all had the experience of writing some code, and thinking we have written those lines somewhere before.) We factor these out to create an abstraction, and replace each occurrence by a name and arguments (for an abstraction of processing), or by an instance of a type (for a data abstraction). As in classification, we notice that instances of similar processing or structure are present in the system, and create an abstraction that represents those instances.

Encapsulation and information hiding

The two processes of determining abstractions reflect the two essential characteristics of abstractions, encapsulation and information hiding. An abstraction is an encapsulation that groups a set of related lower-level units of processing and/or structure. From the bottom-up perspective, the process of designing an abstraction involves deciding which units make up the abstraction. For example,

- A subprogram encapsulates a series of operations.
- A module encapsulates a group of subprograms and objects.
- A record encapsulates its fields (but does not provide information hiding).
- An abstract data type encapsulates both its operations and storage structure.

Like classification, this packaging decreases the amount of information that must be retained to understand the structure of an application.

An abstraction provides information hiding because it hides inessential details from its users so that they can be ignored. Information hiding reflects the top-down perspective that an abstraction is defined by its purpose and how the application uses it, rather than by how that processing or structure is implemented. An important aspect of designing a programming abstraction is the process of separating the logical use of a category of code or data objects from its physical algorithm or representation. The external appearance of an abstraction to its users is called its interface, and the internal structure not visible to users is its implementation. In particular, the code of a subprogram and the storage structure of an abstract data type are implementations that are not visible to their users. Information hiding implies that users of an abstraction do not need to know its implementation to use it, and also that users of an abstraction cannot access its implementation.

Information hiding provides a number of advantages for software design. It reduces the complexity that users of a subprogram or type must handle, and it allows the programmer to think about instances of the abstraction and their interactions with the rest of the application at a higher level. With information hiding, the components of a software system can interact only through their interfaces. This restriction simplifies the relationships among those components, reducing the opportunity for unintended or incorrect interactions and decreasing the number of coding errors. Since designers can implement program components independently, they can partition the development effort in a manner that reflects the structure of the system. Information hiding also allows the designer of an abstraction to change the abstraction’s implementation without affecting other program units that employ it.

We refer to a program component that uses an abstraction through its interface as a client of that abstraction. The clients of a subprogram are subprograms that call it. The clients of an abstract data type are subprograms that create and manipulate instances of the type, or types that use it as the type of

---

29 In fact, the enclosure around the components that an abstraction encapsulates may or may not be transparent. For example, a record encapsulates its fields but does not provide information hiding. However, many authors use the term encapsulation to also imply hiding of the components of the abstraction.

30 We will also occasionally refer to the programmers that use an abstraction as its clients.
components in their storage structure. The interface of an abstraction represents a contract between the designer of the abstraction and its clients that describes the capabilities and behavior that clients can expect from the abstraction.

Subprograms as abstractions

Subprograms were the first mechanism that programming languages supported for named, reusable, programmer-defined abstractions. As we saw in section 1.2, subprograms provide an abstraction that allows us to package a computation and reuse it via an invocation, and to view the structure of programs that call it at a higher level.

A subprogram is an encapsulation of operation whose interface consists of the subprogram name, the argument signature, and a description of its operation. The description of a subprogram's operation can be either an informal explanation of its behavior, such as a comment or reference manual entry, or a formal specification of the pre-conditions and post-conditions of the subprogram's operation. The interface of a subprogram also specifies any error conditions that it can signal (or exceptions in a programming language that supports them).

The hidden implementation of a subprogram consists of its code (i.e., its algorithm) and its local variables. For example, the C programmer can use the standard library function \texttt{sqrt()} without being concerned with how it calculates the square root of its argument. (The interface of this function specifies that its return value is undefined if the argument is negative.) The designer of a subprogram can change its implementation, e.g., to improve its efficiency, without affecting its callers. As we saw in section 2.4, block-structured languages such as Pascal permit the programmer to nest auxiliary subprograms within a subprogram. In this case, the nested subprograms are also part of the implementation of the enclosing subprogram, and are not visible outside its scope.

As mentioned in section 2.4, subprograms are the basis for the design methodology of top-down functional decomposition. The designer decomposes the system according to the tasks that it must perform, then breaks those tasks into subtasks, and so on. Each task is coded as a subprogram that calls the subprograms that implement its subtasks. The interface for each subprogram is specified first according to the task it must perform and the information necessary to perform that task. After determining the necessary subprograms, the programmers design each subprogram's algorithm and local variables. (In fact, some subprograms may be temporarily coded as "stubs" that simply print a message so that their callers can be tested before they are implemented.) This partitioning of the system organizes its processing in levels of increasing detail, beginning with the main program, and allows subproblems at each level of granularity to be solved independently.

4.3 Modules

Separate compilation and linking

For a large application, keeping the entire source code in a single file quickly becomes unmanageable, especially when many programmers are involved. We would also like to avoid recompiling the entire source code of an application each time we modify a particular subprogram, and want to allow several programmers to work on different parts of the application simultaneously. These problems can be handled with the module construct. A \textit{module} is a source code file that can be compiled individually to an object code file, even though it refers to objects, subprograms, and types defined in other modules.\textsuperscript{31} A language that supports modules must provide syntax for partitioning a program into modules, and for declaring the subprograms and objects used by a module that are defined in other modules. In addition, some mechanism must implement cross-referencing and type checking of names that are used in more than one module. It is also helpful if the language provides a facility for determining which source code files in an application must be recompiled when a file is modified.

The \textit{linker} combines the collection of object files corresponding to the modules in an application to create a complete executable program, and resolves references to identifiers that are used in one module but defined in another. It also links the object code for any standard library subprograms the application invokes into the executable. Figure 6 illustrates the action of the linker for an application with three modules containing five subprograms:

\textsuperscript{31} FORTRAN supports separate compilation of individual subroutines. However, this is not the same as supporting modules because each compilation unit can contain only a single subroutine and no objects.
Five procedures defined in three modules

The result of linking the three modules

Figure 6: The linker resolves external references among modules

To support linking separately compiled modules, the compiler must include information about externally visible names with each object file. This consists of an external symbol table of the names exported by and imported into that module. The entry for an exported object or subprogram gives its location in the object code, and the entry for an imported object or subprogram gives a list of the locations of uses (or calls) of that entity. The linker uses these tables to substitute the location of the referenced object or subprogram (adjusted to its location in the entire executable) for each use, as illustrated in figure 1.1.

The linker can be provided by the operating system or by the programming language system. If the programmer specifies the files necessary to build the executable (as in C), a linker supplied by the operating system (e.g., the UNIX linker) can link modules written in different languages. To relieve the programmer of this task, the language system can maintain the module library and deduce which modules are necessary for an executable from the import declarations in the modules (as in Ada).

With the static linking we have described, the entire executable is packaged and linked before executing the application. An alternative is dynamic linking, in which the operating system maintains object files in a run-time library, and then loads object files and links them with the application as necessary while execution proceeds. This approach is more efficient for system-provided functionality such as file
operations or graphic interface components. The system can link several executing applications to the same library object files (if they are re-entrant) rather than duplicating them within each program. In addition, if a particular execution of the application never invokes the library code, that object file is not linked and loaded.

**Interface and implementation**

Languages that support modules allow us to divide the text of a program into modules (or, alternatively, to compose a program from a group of modules). Originally, modules were invented to allow a large program to be broken into separately compiled units. More important for managing a large project, different programmers can write and debug different modules. To do so independently, the interface of each module must be clearly defined so that a programmer working on one module does not need knowledge of the internal details of the other modules. That is, it became apparent that modules provide an abstraction.

In languages that support modules, we divide the code for a module into a public interface and a private implementation. (We will examine examples of module interface and implementation code in Modula-2 and C later in this section.) The interface of a module consists of forward declarations for the subprograms and objects it defines that are visible to its clients, and documentation describing how to use the module's facilities. If a module exports a constant, its definition must appear in the interface so that the value of the constant is available to the compiler when translating client modules. If a module exports a type name, e.g. a record or enumeration, the definition of the type must be given in the interface so that the compiler has the structure of the type, e.g. the size of an instance and the component names, when translating client modules.

The implementation of a module defines the subprograms and objects declared in its interface. In this way, each name is defined once in the entire application and each object or subprogram is allocated once in the application's executable (i.e., in the object code for its module). The implementation may also define subprograms, objects, constants, and types that are used only within the implementation of the module, and are not visible to clients. For example, the implementation may define auxiliary subprograms that are used by the module's subprograms, but are not intended to be invoked by clients.

The identifiers declared in a module's interface are *exported* to its clients. A language that supports modules must also provide syntax for *importing* module interfaces so that modules can use other modules' facilities. (The implementation of a module implicitly imports its interface.) A module cannot import another module's implementation. Depending on the language, an import declaration may specify an entire module interface (as in C and Ada), or a particular identifier in a module (as in Modula-2). Importing the declaration of an external object or subprogram permits the compiler to perform type checking of uses of that identifier in the client module. Note that an imported interface is located in a different code file than the client code, so the compiler must have a way of finding the interface file, given the module name. A name qualification or renaming feature is also helpful for resolving name conflicts among identifiers in the interfaces a module imports. Both the interface and the implementation of a module may contain import declarations for modules whose facilities they use, and the import declarations in a module interface are also visible in its implementation.

It is helpful if the language system allows the interface and implementation of a module to be written and compiled separately. This facilitates top-down programming because the interface can be specified and imported by other modules before the implementation has been designed and coded. Compiling a module implementation produces an object code file containing the code for its subprograms and allocations for any objects it defines. However, "compiling" an interface does not actually generate an object code unit in the traditional sense. It performs syntactic analysis of the declarations, and creates a symbol table that contains information allowing the compiler to perform type checking of uses of the exported identifiers in client modules. When compiling the corresponding implementation, that table is updated with references to the objects and subprograms in the implementation's object code. The compiler also ensures that the declarations in a module's interface match the definitions in its implementation.

**Module scope**

Information hiding is provided by the scope rules of the language. An identifier declared at module scope (i.e., not as a local variable of a subprogram in the module) is visible from its point of declaration throughout the module, and is not visible outside the module if it is not exported. That is, modules provide a level of scope structure that encloses the scopes of individual subprograms. Names declared in both the

---

32 C++ makes a distinction between access control and scope (unlike any other language).
interface and the implementation of a module contribute to the name space of the module's scope, as do imported names. The scope of a declaration at module scope is that module, and if it is exported, any modules that import that name. Within the implementation of a module, block-structured scope is usually supported so that all the identifiers declared at module scope are implicitly visible within the code of the module's subprograms.

Objects declared at module scope are statically allocated, and exist during and between invocations of the module's subprograms. That is, each module has a state that persists even when no subprograms in the module are executing. The language must provide a mechanism for initializing these objects. Since the names of module scope objects might or might not be exported to clients of the module, using module scope objects rather than global variables allows the designer to control access to these objects.

Languages that support separate compilation have used two different approaches to defining the name space of exported names. C uses static scope with one global scope for the entire application in which all exported names are defined. The scope of each module is nested within this scope, and exported names are visible implicitly within all modules. Modula-2 and Ada support the import/export scope relationship we discussed in section 2.5, and permit us to qualify an imported name with its module name to resolve name conflicts. Recall that import/export scope provides programmer control over the visibility of names (rather than the implicit name sharing that occurs with block-structured scope), so that a name is visible exactly where necessary. The designer of the module has explicit control over names visible in a module, and a name can only be shared via an explicit contract between the designer and the client. We will discuss both approaches in detail in this section (C++ includes both).

**System structure**

The code for a large system can consist of hundreds or even thousands of subprograms. Attempting to comprehend the structure of such a system by considering a list or invocation graph of these subprograms makes it difficult to "see the forest for the trees" because the representation is too fine-grained. An additional level of structure that groups related subprograms and the data objects they manipulate would facilitate understanding the structure of the system. Modules provide this higher-level abstraction.

Since each subprogram has its own scope, several programmers can use the same names for local variables without conflict. However, in a language without modules, such as standard Pascal, if more than one subprogram needs access to a data object, the object's name must be relatively global to those subprograms, and to all subprograms at that level of nesting. When subprograms communicate by manipulating global data structures, they are tightly coupled. An error or a modification to a data structure in one part of a program can have effects throughout the system, and coordination of the use of that data structure is complicated. Such systems are difficult to revise for the same reason.

Unlike the programming language features discussed in the first three sections, the module construct was introduced specifically to support the demands of large-scale programming projects. In a functional decomposition of an application, the designers partition the code by using a module for each major task. Each module consists of a group of related subprograms and the data structures that they manipulate. A data structure that must be visible to several modules is imported in only those modules. With modules as the unit of system decomposition, the focus of design is on dividing the application into groups of related subprograms, and associating the necessary data with each module. Like all programming abstractions, modules reduce complexity by decreasing both the interdependency among code units, and the amount of communication and coordination necessary among the programmers working on a system.

The use of modules for structuring programming systems was championed by Parnas, who formulated "Parnas's rule" for information hiding. This rule states that the designer of a module should provide a client with all the information necessary to use the module effectively and nothing more, and should provide the implementor with the information needed to code that module and nothing more. In this way, the client cannot write code that depends on the implementation. In addition, the implementor does not know what programs will use the module, and cannot write code that depends on characteristics of the module's clients. The clear distinction between the interface and implementation allows the modules that make up an application to be written, tested, and debugged independently.

Deciding on the right set of modules for the system is often a difficult problem, separate from determining the subprograms. However, it must be resolved because only very small systems can be contained in a single module. A good decomposition of the system's processing resulting from analysis of the requirements specification and the problem domain manages complexity by producing modules that we can design, understand, and code independently. It localizes the inevitable changes during the lifetime of the application to a small number of modules, so only those modules need to be considered and modified.
The granularity of the modules is also an issue. If modules are too small, organization and documentation becomes intractable, programmers have a difficult time finding the subprograms they need, and changes to the system can involve reworking numerous modules. On the other hand, if modules are too large, they cannot be understood as a unit and more recompilation may be necessary when making modifications.

Once a set of modules has been developed, it provides a library of abstractions that can be used in other applications with similar requirements to reduce the coding effort. For example, we can write and debug a module containing statistical functions once, and then use it in many such applications. Similarly, the C standard library includes modules containing functions that perform input/output and file operations, mathematical operations, string and character operations, error handling, memory allocation, time and date functions, and various other tasks. No C programmer ever has to write these functions. The disadvantage of grouping library subprograms into modules is that the linker (usually) includes the object code for the entire module in the executable, even if the application only invokes a single subprogram in that module.

**Coupling and cohesion**

Various dependencies can exist among the modules in an application, and among the components of a module. *Coupling* describes the degree of interdependence between the modules in a system, and *cohesion* describes the degree of interconnection among the elements encapsulated within a module. Software engineers use these concepts as criteria for evaluating a system design. In particular, we attempt to minimize the coupling among modules, and to maximize the cohesion within each module.

Coupling is reduced when modules communicate only through their interfaces, rather than by using global variables or accessing each other's data structures directly. Coupling is also reduced by decreasing the number of interactions between the modules in a system and the sizes of the modules' interfaces. When a module has a single clear purpose and implements a single well-defined task, it exhibits strong cohesion. This logical binding of the module's elements is expressed by the common sense saying that "things that belong together should be together", and things that do not should not. If a module has high cohesion, it should be possible to give a single sentence description of the purpose of the module. Cohesive modules furnish reusable components for applications that deal with the same problem domain or platform.

**Modules in Modula-2 and Ada**

Languages that support modules include constructs for forward declarations, for dividing a module into an interface and an implementation, and for importing interfaces. The compiler implements the import/export scope rules, and the language system must provide mechanisms for linking uses of imported names with the corresponding subprogram or object in another module, or use those of the operating system. Modula-2, Ada, and C (to a degree) provide these facilities. Most contemporary implementations of Pascal also support separately compiled units, although the language standard does not.

Modula-2 is a successor of Pascal that adds support for modules\(^{33}\) (as well as providing concurrency, low-level facilities, procedure types, and improvements to control structures). The module interface and implementation are called the *definition module* and the *implementation module*, respectively. The following example illustrates Modula-2 definition and implementation modules:

```plaintext
(* a Modula-2 module interface and implementation *)

DEFINITION MODULE trigMath;
  CONST
    pi = 3.1415926535;
    radiansPerDegree = pi / 180.0;
  PROCEDURE sin(x: REAL): REAL;
  PROCEDURE cos(x: REAL): REAL;
  (* ... other trigonometric function declarations ... *)
END trigMath.

IMPLEMENTATION MODULE trigMath;
  PROCEDURE sin(x: REAL): REAL;
    (* ... code for sin ... *)
```

\(^{33}\) In this section, we are discussing what Modula-2 refers to as "library modules". The language also supports "local modules" that define a name space with import/export scope within another module, but are not separately compiled.
A definition module may only contain declarations, not executable statements. Variables, procedures,\(^{34}\) constants, and types declared in the definition module are visible throughout the implementation module (as if the corresponding definition module is implicitly imported), as well as in any modules that import them. The implementation module must define any procedures declared in the corresponding definition module, and it may also define variables, procedures, constants, and types that the module does not export. The implementation may also contain an unnamed block of code, which usually initializes variables declared at module scope in the definition and implementation parts. This code is executed before any of the module's procedures are invoked.

The declaration **IMPORT** module; imports all the identifiers in the definition module module, and the client must qualify those names with the module name, i.e. as module.identifier. A client can also directly import individual names from a module's interface with the declaration **FROM** module IMPORT identifier1, identifier2, ..., in which case he or she can use those names without qualification (but those names cannot already be declared in the module). The **FROM** declaration essentially declares a local alias for the entity whose name is imported.\(^{35}\) If a module must import the same name from more than one module interface, it can include **IMPORT** declarations for each module and use qualified names so that there is no name conflict. Import declarations must appear at the beginning of the client module (after its header), and can be used in both interface and implementation modules. Identifiers imported into a definition module are visible in the corresponding implementation module. There is no global scope visible throughout the program in Modula-2 (although predefined identifiers such as CHAR and TRUNC are visible in any module).

Ada supplies the same capabilities and semantics for modules as Modula-2, but the language uses different terminology and syntax. In Ada, modules are called **packages**. A module interface is called a **package specification**, and all names declared in the specification are exported, except for those in the **private** section.\(^{36}\) A module implementation is called a **package body**, and it defines subprograms declared in the public part of the package specification, as well as private variables, subprograms, and so on. A client package imports the nonprivate names in a package specification with the **with** package; declaration, which makes those names visible as qualified names in the client. The **use** package; declaration declares external alias for the identifiers in the specification of package, so that the client package can use them without qualification. If a name conflict occurs between names imported from two modules, we can resolve a use with a qualified name. We will see examples of Ada packages in the next section.

Both Ada and Modula-2 allow compiling the interface of a module separately from its implementation. The language system maintains a database of module interfaces and implementations so that the compiler can find imported interfaces and match corresponding interfaces and implementations. (The system might or might not use the facilities of the operating system file system.) The language system ensures that we do not compile a module unless we have created and compiled all the interfaces that it imports. However, we can compile a module (but not link and execute it) even if we have not coded the implementation of an interface it imports. Linking compiled object code units can be done statically or dynamically, but must be done by a language-specific linker that uses information in the module database.

The language system also manages **compilation dependencies** among modules. An implementation depends on the corresponding interface in the sense that the implementation must be recompiled when the interface is modified, but not vice versa. Similarly, a module that imports module's interface depends on that interface, but does not depend on module's implementation. To relieve the programmer of determining which modules must be compiled when modifying a module, the language system recompiles all modules that depend on an interface when it is edited and compiled. Use of top-down design minimizes the amount

---

\(^{34}\) Both procedures and functions that return a value are called procedures in Modula-2.

\(^{35}\) The C++ using namespace directive corresponds to the IMPORT declaration (although name qualification is not necessary if a name is not declared locally), and the C++ using declaration corresponds to the FROM declaration.

\(^{36}\) Having a private section in a package specification also supports using packages to implement abstract data types.
of recompiling necessary during development because the interfaces stabilize early in a project, with most modifications being made in the implementations. Note that if mod1 imports the interface of mod2, compiling the interface of mod2 should trigger recompiling mod1. However, there is no indication that the dependency exists in the code of mod2, so the module database must represent these dependencies explicitly (i.e., the compiler cannot deduce a module’s dependents from its source code).

**Modules in C**

C support for modules and separate compilation is somewhat implicit, and is provided partly by the language, partly by the preprocessor, and partly by the operating system environment. C does not support the import/export scope relationship provided by Modula-2 and Ada. Instead, each source code file defines a level of scope nested within the global scope of the entire program, and all exported names are declared in this global scope. A C program has three levels of scope: global scope, the scope of each file, and the function scopes nested within the file scopes. The usual static scope rules apply among these levels, including implicit access of names in enclosing scopes and hiding of names by declarations in an intervening scope.

Both global scope and the scope of each file are referred to as file scope, and only declarations are allowed at file scope. All file scope declarations are global by default, and the reserved word static indicates that a file scope declaration is only visible within that file.37 That is, functions and objects that are private to a module are declared at file scope as static, and the compiler does not include those names in the module’s external symbol table. The extern specifier indicates that a file scope declaration is not a definition, i.e. that the referent is defined in another source code file to be linked with this file. Function declarations are extern by default.

Figure 7 illustrates C scope relationships in a program that consists of two modules containing three functions (dashed lines demarcate the text of the code files and solid lines indicate scope boundaries):

![Diagram of C scope relationships](image)

**Figure 7: The three levels of scope in a C program**
(dashed lines demarcate the code files, and solid lines indicate scope boundaries)

The names file1.c and file2.c are not declared in the global name space, and are not even known to the compiler. file1.c defines two global names, num1 and func1a(), and defines the identifiers num2 and func2().

---

37 This is a completely different use of the term static than a static local variable. All file scope objects are statically allocated, whether they are globally visible or not.
func1b() as local to itself. The scopes of the functions func1a() and func1b() are nested within the scope of the file (rather than the global scope): a static file scope identifier is implicitly visible in those function scopes if a local variable does not hide it, and static file scope identifiers hide global identifiers within the functions. (However, num2 is not visible in the code for func1a() because its declaration has not yet appeared at that point in the source code.) The extern declarations of num1 and func1a() in file2.c specify that those names are defined in another module, and permit the compiler to type-check uses of those names in file2.c. The two definitions of num2 in the two files do not clash because uses of that name in file1.c refer to its file scope variable, and similarly for uses in file2.c. The functions func1b() and func2() cannot be called outside their respective code files because their names are not entered into their modules' external symbol tables.

If we use an identifier in more than one code file, we define it in one file and declare in the global name space. Modules that import that name use an extern declaration. The convention in C is to place the forward declarations of objects and functions used in more than one module in header files. Header files may also define constants, types, and preprocessor macros. A small program may use one header file for all its global declarations. For a large application with a modular design, we code each module as two files, a header file containing the module interface and a code file containing its implementation. By convention, we name the corresponding header and code file module.h and module.c. The following example illustrates a C header file:

```c
/* a C header file, trigMath.h */
#define PI 3.1415926535
#define RADIANS_PER_DEGREE (PI / 180.0)
    double sin(double); /* pre-ANSI C does not specify the parameter types */
    double cos(double);
    /* ... other trigonometric function declarations ... */
```

Note that there is no indication in the language that this group of declarations make up the interface for a particular module.

Rather than using a language-supported import declaration, client files textually include the module's header file with the preprocessor directive #include. This operation inserts the text of the header file directly into the client code file before compilation, making those declarations available. (We also include the header file in the module's code file to ensure that its declarations are consistent with the definitions in the code file.) This command uses the file name of the header file, as follows:

```c
/* inclusion of interfaces coded in C header files */
#include <stdio.h>    /* a standard header file */
#include "trigMath.h" /* a programmer-defined header file */
```

In this way, the language system avoids the complexity of dealing with compiled interfaces and the requirement of managing a module library. However, an interface is recompiled every time a client file or the implementation file is compiled, increasing the total compilation time. In addition, the compiler cannot ensure that every client of a module has included the same version of its header file.

The C compiler includes an external symbol table with each object file. The table does not include type information, which is why you should include a header file in the corresponding code file. The operating system linker uses the symbol tables to resolve each external declaration to the corresponding definition when creating the executable. If more than one module defines a particular identifier in the global name space, the linker (not the compiler) signals a "multiple definition" error. If an identifier declared as global is not defined in any of the files linked to it, the linker signals an "undefined symbol" error.

Although C's approach to separate compilation and module scope simplifies implementation of the language, the compiler has no information about modules and the name importing relationships in the program. Name qualification is not possible, so a module cannot import the same name from more than one module. In fact, since there is a single global name space for all exported names, uses of the same name in different interfaces in an application conflict, even if those names are not used in the same module. For example, in a Modula-2 program with four modules mod1, mod2, mod3, and mod4, mod2 can import the name proc from mod1 while mod4 imports proc from mod3 without conflict. If we attempt to use the same pattern of import relationships in a C program, the linker would flag the name proc as multiply
defined because both \texttt{mod1} and \texttt{mod3} export that name to the global name space (and, therefore, \texttt{proc} is defined in the external symbol tables of both object files). Figure 8 illustrates these scope relationships:

Using module names in Modula-2 to select an imported identifier

The name conflict cannot be resolved in C

Figure 8: Exporting the same identifier twice in a program in Modula-2 and C

Since C has no knowledge of modules, the language system cannot automatically manage compilation dependencies among source code files. When a source file is modified, either the programmer or a programming tool must determine which additional modules to recompile before linking the application. UNIX programmers use the \texttt{make} utility for this purpose, and pass it a "makefile" that specifies the dependencies among the various header, code, and object files that make up an application. The makefile associates an operating system command with each dependent file, usually to compile or link, that "makes" that file. When \texttt{make} executes, it checks the times of modification of the involved files, and executes the commands to make any files that depend on files with later timestamps (doing so recursively). However, it is the programmer's responsibility to encode the dependencies, and to maintain them as the application is developed. Many C programming environments include a "project" facility that handles these details automatically.

4.4 Abstract data types

Definition and purpose

An \textit{abstract data type} is a programmer-defined type that packages the structure of the type's instances and the operations for those objects, and allows the designer to control access to those components.\footnote{Some authors use the term "abstract data type" to mean only the interface of the type, excluding its implementation (because it is "abstract").} Typically, the structure of the type's instances is hidden from its clients (i.e., from program units that create
and use instances), and clients may manipulate instances only with the operations defined for the type. That is, the definition of an abstract data type consists of

- an interface presenting the behavior of its instances
- a hidden implementation describing how an instance is represented, and how that representation is manipulated to provide the type's operations.

The type is "abstract" in the sense that clients cannot see its concrete implementation (unlike a record type).

As we saw in section 3.3, types model categories of entities in the problem domain, and the use of types improves readability and increases the correspondence between the structure of the application and that of the problem domain system. Abstract data types improve upon the type constructors of Pascal by encapsulating the type's operations directly with the type to supply a more complete package. The operations defined for a type should be motivated by the behavior of the real-world category that the type represents. For example, the abstract data type Bank in an ATM application includes operations for validating and executing transactions and for creating accounts. Like modules, we can design and code abstract data types separately, facilitating independent verification of the type's operations, isolation of program errors, and localization of modifications. Like modules, abstract data types can be stored as reusable components in a library.

Languages that support abstract data types provide constructs for

- packaging a type's operations with the definition of the type
- defining all the necessary operations so that the type is first-class
- specifying which components of the type are visible externally
- instantiating the type and using its operations on those objects

We will see in this section that an Ada package can export a type's name and operations and hide its implementation, so packages support abstract data types. Object-oriented languages also support abstract data types, and we will examine a small C++ example in this section.

**Interface and implementation**

We describe the external appearance of an abstract data type by stating what operations are valid for its instances and clearly describing the externally visible effects of each. For example, the client of a set can add, delete, and search for elements, obtain the number of elements, check for equality and subset relationships, copy sets, obtain the intersection or union of sets, and so on. Each operation in an abstract data type's interface is specified by a subprogram declaration that gives its name and argument signature, any exceptions it can signal, and a logical description of the operation.

An important component of the interface of the type is an operation for initializing an instance of the type with a valid value. For example, an instance of the type Set begins with a logically empty data structure (e.g., an empty list, tree, or hash table), and the number of elements set to 0. For many types, initialization takes arguments that give values for the information stored in the object, and these values are necessary to create a valid object. For example, an instance of Book must be initialized with a particular title, author, publisher, date, and so on (rather than having garbage in those fields as a new Pascal record would). Initializing a Date object requires a year, month, and day, and the initialization operation must do range checks on the arguments. No operations should be performed on an object unless it has been properly initialized.

By prohibiting direct access to a type's data components, the designer of an abstract data type provides representation independence. Like all information hiding, representation independence

- relieves clients from understanding the details of the type's implementation
- prevents clients from manipulating that implementation in ways that are not consistent with the type's semantics, either accidentally or intentionally
- allows the designer to change the storage structure of the type without affecting clients

These characteristics increase the reliability of programs that use the type. Representation independence is not a new concept in programming. For example, most languages provide floating-point types without supplying mechanisms for examining the actual bit string of an instance and the implementations of the arithmetic operations. However, application of representation independence to programmer-defined types is more recent.

Frequently, we can implement a particular logical entity in several different ways, each with its own storage structure and algorithms. For example, we can represent a set of elements as an array, a linked list, a binary search tree, or a hash table. We would implement the logical operations of searching for elements,
adding and deleting elements, determining the number of elements, and so on, differently for each representation. With representation independence, clients of the type manipulate set objects via its public operations only, allowing us to use any representation or change the implementation. This contrasts with languages such as Pascal in which the structure of a type is visible throughout the program, and modifying the type's implementation can break other code units.

If the information maintained by a particular abstract data type should be accessible to clients, you should provide accessor functions, rather than declaring the data components as public, to achieve representation independence. For example, we can represent a point in either Cartesian or polar coordinates. By providing operations that access and set a point object's \( x, y, r \) and \( \theta \) coordinates, we can use either representation, or change representations during development, by defining each of these operations correctly for the chosen representation. (For example, the operation that sets a point's \( x \) coordinate performs a single assignment with the Cartesian representation, but calculates and sets both coordinates in the polar representation.) Client code will use the accessor operations, and will not be affected by changes to the structure of the type. Another advantage of defining operations that set an instance's components rather than permitting clients to assign values directly to those components is that the type's operations can perform validation such as range checking so that an instance does not have an invalid state.

The implementation of an abstract data type consists of declarations that describe the storage structure of an instance, and subprograms that define the type's operations in terms of that structure. A record may be sufficient for encoding the information maintained by an instance, or the designer may use a dynamic linked data structure. For example, the storage structure of a date can be a record containing two integers giving the day number and year number, whereas that of a set may be a linked list, a binary search tree, or a hash table. For each subprogram in the type's interface, the implementation gives a subprogram definition that describes how to examine and modify an instance's storage structure to achieve the result defined for that operation. As with modules, the type's implementation may also define private subprograms, constants, objects, and types (e.g., for the nodes in a linked structure) that aid in coding the type's operations and structure.

**System structure**

As designers developed larger systems, they often found that the complexity of the data objects in the application contributes significantly to the complexity of the system, and that most of the operations necessary were specifically relevant to certain types of data objects. In addition, designers noticed that once they had created a type for a category of objects or data structure, they could frequently use it directly in other applications dealing with the same problem domain, design characteristics, or user interface. These realizations caused a shift in the process of design from process-oriented functional decomposition to **object-based decomposition**. With this design strategy, the focus is on determining the categories of entities that the system must represent and assigning operations to each, rather than on dividing the system structure according to the processing that it performs.

In the same way that subprograms and modules provide abstraction of operation, abstract data types provide abstraction of data. In moving from procedural modules to abstract data types as the basis for the structure of the application, there is a change of focus from a module and its data to a data object and its operations, i.e. from verbs to nouns. In object-based decomposition, the central task is determining the categories of entities that the system must represent, and their attributes and interactions. These interactions determine the interfaces necessary for each abstract data type. Rather than the system structure being a tree of tasks and the subtasks that they call, it is a graph of the interactions and relationships between abstract data types.

Parnas's rule and the principles of cohesion and coupling are as important for designing abstract data types as they are for designing modules. The interface of a type must include everything that a client needs to use instances of the type effectively, and the implementation should include exactly what is necessary to implement that interface. Clients should not be able to write code that depends on the representation of the type, and the designer should not make assumptions about code units that use instances of the type. The attributes and properties encapsulated within an abstract data type should be logically related, which will be true if each type is motivated by a distinct problem domain category. Instances of types should only

---

39 We use the term "object-based decomposition" to distinguish from "object-oriented decomposition", which also takes advantage of the additional properties of inheritance, dynamic binding, and class information that are available for classes, but not for abstract data types.
communicate through their interfaces, rather than by using global variables or accessing each other's storage structure directly.

**Dynamic subobjects and abstract data types in C++**

Since an abstract data type's implementation is hidden, the type can dynamically allocate substructures that are not visible to its clients. For example, we can define a list abstract data type implemented by a linked list. The type would be responsible for allocating and deallocating the list nodes, and must hide those details from clients. The distinction between instantiation and initialization is essential with such abstract data types. Instantiation creates an instance of the type, and initialization of that object creates and initializes the dynamic subobjects.

Let us define an abstract data type for a stack of characters in C++. To avoid having a limit on the number of elements in a stack (since it is not logically part of the stack concept), our stack type can allocate the array of elements dynamically. If the stack is full upon a push, we will allocate a larger array, copy the elements to it, and deallocate the old array, before pushing the new element. We will code this version using three members: the current size of the array, the index of the top element, and a pointer to the dynamic array. Figure 9 illustrates the storage structure of an instance allocated as a local variable of a client function `func()`:

![Stack Frame Diagram](image)

**Figure 9: Storage structure for the abstract data type `CharStack`**

The dashed line in the figure indicates that the three-member structure in the client function's stack frame and the dynamic array are logically a single object, although physically they are stored in different regions.

In C++, we define an abstract data type as a class, an expanded version of the C struct definition that includes

- *data members* that define the storage structure of the type
- *member functions* that provide the type's operations
- *access control* specifications, e.g. `public` and `private`, that indicate whether members are visible to clients

The syntactic unit that defines the type directly encapsulates the type's operations within the type itself, rather than grouping the type and its operations in a package as in Ada. A C++ class also does not define a separately compiled unit like an Ada package or a Modula-2 module. However, it does define a separate named scope, and the language supports name qualification with the scope operator :: and a class name. We define the class `CharStack` as follows:

```cpp
// a C++ class for a stack of characters
class CharStack
{
  public:
    void initialize() // initialize the object to an empty stack
    {
      size = INIT_SIZE;
    }

    // other member functions...

    // member data...

    // constructor...
    CharStack(size_t initialSize)
      : size(initialSize), top(-1)
    {
    }

    // destructor...
    ~CharStack()
    {
    }

    // other methods...
}
```

- 61 -
elements = new char[INIT_SIZE];
top = EMPTY;
}
void release()
{ delete [] elements; }
void push(char ch)
{
    if ( top == size - 1 )
        grow();
    elements[++top] = ch;
}
char pop()
{ return elements[top--]; }
bool isEmpty()
{ return top == EMPTY; }
private:
    char* elements;
    int top;
    int size;
    static const int INIT_SIZE;
    static const int EMPTY;
    void grow()
{
        char* newElems = new char[size + INIT_SIZE];
        for ( int index = 0; index < size; index++ )
            newElems[index] = elements[index];
        delete [] elements;
        size += INIT_SIZE;
        elements = newElems;
    }
);

const int CharStack::INIT_SIZE = 8;
const int CharStack::EMPTY = -1;

When a CharStack member function uses a data member name, it refers to that data member in the stack object whose operation is being performed. As we will see in section 9.1.5, C++ uses the new and delete operators for dynamic allocation and deallocation, rather than the C malloc() and free() functions. Unlike C, C++ includes a type bool for boolean values. The auxiliary member function CharStack::grow() is private to the class and clients cannot invoke it. We define the "static data members" CharStack::INIT_SIZE and CharStack::EMPTY within the scope of the class CharStack, but they are static objects that are not stored in each instance. To define and initialize them, we use qualified names.

When a client creates an instance of CharStack as an automatic or dynamic object, storage is allocated for its three data members elements, top and size, but they are not initialized. The member function initialize() provides these members with initial values, and allocates the array referred to by elements on the heap. We must define this function because the data members are private, and it relieves clients from having to know how to initialize a CharStack object. A client must invoke this function immediately after creating an instance of the type. If not, a run-time error or invalid result will eventually occur. With a data type such as CharStack that allocates substructures on the heap, we also need a function that deallocates those substructures when the object is no longer needed (since C++ uses programmer-controlled allocation). The member function release() frees the dynamic array referred to by elements, and clients must call it just before deallocating a CharStack so that a memory leak does not result. In general, it is necessary to define initialization and finalization functions for any type whose instances use system resources so that clients of the type do not need to know what resources it uses. In fact, we will see in section 10.2 that C++ supports type-specific initialization and finalization functions, which are called "constructors" and "destructors" respectively, that the compiler invokes implicitly whenever an object is created or destroyed so that these errors do not occur.
For a type such as CharStack that defines a dynamic subobject, we must also distinguish between initialization of a new object and assignment to an existing object. For initialization, the target object is just raw storage whose values must be set. With assignment, the dynamic subobject of the target instance has already been allocated. For example, consider the assignment chStk2 = chStk1. A C structure assignment would assign the values of the corresponding data members of chStk1 to those of chStk2, which is incorrect. If chStk2.size >= chStk1.size, we can copy the contents of chStk1->elements into chStk2->elements and set chStk2.top. If chStk2.size < chStk1.size, we deallocate chStk2's subobject, allocate a new array of size chStk1->size (or larger), then copy the elements. As we will see in section 11.1.3, C++ allows the designer of a type to overload the assignment operation with the required behavior.

In C++ and object-oriented languages, the syntax for invoking a type's operations distinguishes the type instance from the other argument of the call. We invoke a member function of an object using the component selection operator, like accessing a public data member of the object, as follows:

```cpp
// creating instances of the type CharStack and using their operations
#include "CharStack.h"  // import the interface

int main()
{
    CharStack chStk1, chStk2;
    chStk1.initialize();
    chStk2.initialize();
    chStk1.push(\"\$\" );
    if (!chStk1.isEmpty() )
        chStk2.push(chStk1.pop());
    chStk1.release();
    chStk2.release();
}
```

The C++ invocation syntax chStk1.push("\$") makes it more apparent that the function push() is directly associated with the chStk1 object than does the C call syntax, push(chStk1, \"\$\") because the C syntax makes no distinction between the arguments chStk1 and \"\$. In object-oriented terminology, the expression chStk1.push("\$") sends the message push() to the object chStk1. If you already know C++, you know that the member functions CharStack::initialize() and CharStack::release() would be defined as a constructor and destructor, so that the client would not have to call these functions explicitly.

**Generic abstract data types**

A common use of abstract data types is to represent a collection with particular characteristics such as a stack, a set, or a sorted list. We can design, code, and verify the collection type, and then reuse it in numerous applications. In a statically typed language, all the elements of a collection must be of the same type, or are converted to that type. Whether the collection is implemented by a sequential or linked organization, all elements will be bound to an identifier of the same type, either that of the array elements or the "element" field of the records in the linked structure. An advantage of this characteristic is that the compiler can ensure that an element of the wrong type is not added to the collection. A problem that occurs with collection types is that the interface, storage structure, and subprogram definitions for a stack are the same, no matter what the type of the elements is. However, in a statically typed language, a stack of integers and a stack of characters are different types because their operations have different argument signatures and the types of their elements components are different.

We would like a language feature that allows us to code the interface and implementation for a particular type of collection once, without specifying the actual type of an element. The compiler would then use that specification to create a type for a collection with a particular element type when indicated by the programmer. We refer to the construct that the compiler uses to generate an actual abstract data type as a *generic* type definition. The generic type is not an actual type that can be instantiated, but a template from which the compiler can generate a compilable type definition. It is similar to the Pascal array of type constructor, which is not a type itself, but can be used to define a new array type by giving an index type and a base type.

The Ada *generic package* construct allows us to code a package template with a *type parameter*, from which the compiler can generate actual packages. The client gives an actual type when specifying creation
of a package. The specification of a generic package is preceded by the reserved word generic and a list of parameters. These parameters may be objects (like subprogram parameters), types, or subprograms. For example, the following code defines the specification of a generic package for generating stack packages that define a fixed-size stack type with a given element type and capacity:

```ada
-- an Ada generic package specification for a fixed-size stack type
generic
  Stack_Size: Positive; -- the capacity
  type Elem_Type is private; -- the element type
package Stack_Types is
  type Stack is private;
  procedure Push(Stk: in out Stack; El: in Elem_Type);
  procedure Pop(Stk: in out Stack; out Elem_Type);
  function Is_Empty(Stk: in Stack) return Boolean;
  function Is_Full(Stk: in Stack) return Boolean;
private
  Empty: constant := 0;
  type Elem_Vector is array(Integer range <>) of Elem_Type;
  type Stack is
    record
      Top: Integer range Empty .. Stack_Size := Empty;
      Elements: Elem_Vector(Empty + 1 .. Stack_Size);
    end record;
end Stack_Types;
```

This generic package does not actually define a package containing a stack type and its operations. Instead, it supplies a template that the compiler uses to create a stack package with particular bindings for the parameters when specified by the programmer. The following example instructs the compiler to generate a package that defines a type for stacks of integers of capacity 100, and then creates an instance and uses its functions:

```ada
-- generating a package from a generic package with Stack_Types;
procedure Main is
  -- create the package for the type "stack of up to 100 integers"
  package Int_Stack is new Stack_Types(100, Integer);
  use Int_Stack;  
  Stk: Stack; -- create an instance of Int_Stack.Stack
  Int: Integer := 0;
begin
  Push(Stk, 10);
  if not Is_Empty(Stk) then
    Pop(Stk, Int);
  end if;
end Main;
```

If a program uses stacks with different element types or sizes, it must generate a separate package for each stack type. Since each stack type is different, the subprogram names in the generic package are overloaded for each stack type used, and the type of the stack argument determines which subprogram to call. However, since type names cannot be overloaded, a program that uses more than one stack type must qualify each use of a type name with the generated package name, e.g. Int_Stack.Stack.

The generic package Stack_Types does not make any demands on the actual type bound to the type parameter, other than that it can copy an instance to the Elements array (i.e., the type cannot be limited private). Other collection types may require the element type to provide certain operations. For example, a

---

40 Ada uses the generic package mechanism rather than a subprogram type to support subprogram parameters.
set type implemented as a binary search tree must compare instances of the element type, e.g. using the < operator. In Ada, these operations are passed as subprogram parameters to the generic package. Ada also provides many mechanisms for specifying constraints on the actual type bound to a type parameter, e.g. that it must be discrete.

The C++ template facility allows the class designer to define a collection class and its operations generically, together with a parameter that represents the type of the elements, by defining a class template. A client can specify the collection type and the element type when creating a collection object, with the compiler generating the definition of the collection class for the given element type automatically. That is, once we have defined a class template, the compiler instantiates the template whenever we use the class template name as a type specifier. A class template definition consists of the reserved word template, a type parameter list enclosed in angle brackets, and a standard class definition in which the type parameters may be used as type names. We define the class template Stack as follows:

```cpp
// a C++ class template for a stack
template <class ElemType> class Stack
{
    public:
        void initialize() // initialize the object to an empty stack
        {
            size = INIT_SIZE;
            elements = new ElemType[INIT_SIZE];
            top = EMPTY;
        }
        void release() // deallocate the hidden stack object
        { delete [] elements; }
        void push(ElemType elem) // insert elem on top of the stack
        { if (top == size - 1)
            grow();
            elements[++top] = elem;
        }
        ElemType pop() // remove and return the top element on the stack
        { return elements[top--]; }
        bool isEmpty() // return whether the stack is empty
        { return top == EMPTY; }
    private:
        ElemType* elements;
        int top;
        int size;
        static const int INIT_SIZE;
        static const int EMPTY;
        void grow()
        {
            ElemType* newElems = new ElemType[size + INIT_SIZE];
            for (int index = 0; index < size; index++)
                newElems[index] = elements[index];
            delete [] elements;
            size += INIT_SIZE;
            elements = newElems;
        }
};
```

template <class ElemType> const int Stack<ElemType>::INIT_SIZE = 8;
template <class ElemType> const int Stack::EMPTY = -1;

A class template name together with a set of actual types enclosed in angle brackets constitutes the name of a template class. For example, List<String> is the name of a class generated from the List<> class template that has instances of String as its elements. Like any class name, we can use a template class name
as a type specifier for object and pointer declarations, function parameters, data member declarations, class
derivation, and type parameter bindings. When we do so, the compiler automatically generates a type-
specific instantiation of the corresponding class template and its member function templates. It creates the
new class and member function definitions by substituting the actual type name given in the template class
name for each occurrence of the corresponding type parameter in those templates. The compiler then
translates this definition and creates a symbol table entry giving the template class name and the type object
for the class. If the template class is the type of a variable, that object is allocated and initialized, and we
use it in exactly the same fashion as any other class instance. The following example presents some uses of
template class names:

```c++
// instantiating two classes from the template class Stack
int main()
{
    Stack<char> charStack;
    Stack<Point> pointStack;
    charStack.initialize();
    pointStack.initialize();
    charStack.push($);
    charStack.release();
    pointStack.release();
}
```

### 4.5 Classes

#### Object terminology

In this section, we briefly describe how object-oriented languages extend the construct of the abstract
data type to that of the class, the abstraction that characterizes object-oriented programming. We will
cover classes, polymorphism and inheritance in detail in Chapters 2 and 3.

A class is an abstract data type that has the additional properties of:

- class information and operations
- inheritance among classes
- dynamic binding of class operations

Like a data type, a class is an abstraction that represents a category of entities in the problem domain which
is meaningful for the application, or a category of system components necessary in implementing the
application. A class is characterized by the activities that its instances are responsible for and the
information that they must maintain. Like the features of all programming abstractions, class information
and operations, inheritance, and dynamic binding are motivated by problem domain semantics and software
engineering concerns.

In the object model, an object is an encapsulation of state and behavior, i.e. it encompasses both "data"
and "code", and represents a problem domain entity or a system component. Each object is an instance of
a class that defines its behavior and properties. Each class defines a protocol that lists the messages its
instances can respond to, which gives the interface of the class. That is, the term "message" refers to an
operation of the class.

Each class defines its instances' internal state as a set of instance variables that represents the attributes
and properties of such objects. Each individual object has values for those variables that define its current
state, and those values may change over time. The instance variables are not visible to clients, so classes
support representation independence (although some object-oriented languages do not enforce it). For each
message, the class defines a method that describes how its instances respond to that message, namely by
accessing or modifying its state information or by sending messages to other objects. The instance variables

---

41 In fact, it has been suggested that object-oriented programming should be referred to as "class-oriented
programming".

42 Some authors use the term "object" to refer to classes as in the statement "all behavior is encapsulated in
objects", or the Object Pascal definition type = object ... end. I will always use "class" to refer to a category and
"object" to refer to an individual object.
and methods define the class's implementation, and are not visible externally. The terms "instance variable" and "method" correspond to the storage structure and operation definitions of an abstract data type.

Object-oriented programming makes extensive use of polymorphism. We say that an identifier, subprogram or object is polymorphic if it can be of more than one type. For example, an identifier in a dynamically typed language is polymorphic because it can refer to different types of objects at different times. An overloaded subprogram is polymorphic because it can be invoked with arguments of different types. In object-oriented languages, inheritance results in polymorphic objects and identifiers.

A pure object-oriented language is one that is designed specifically to support object-oriented features of data abstraction, class information, inheritance, and dynamic binding, and to enforce the object-oriented paradigm. Smalltalk, Java, and Eiffel are pure object-oriented languages. A hybrid object-oriented language is created by extending an existing procedural or functional language with the object-oriented features, and permits both object-oriented and procedural programming. Hybrid object-oriented languages include Object Pascal, Modula-3, Objective-C, C++, and CLOS.

Message sending
We initiate an operation in an object-oriented system by sending a message to an object, the receiver of the message. The receiver is responsible for performing the requested activity according to a method defined in its class (or, as we will see, inherited from its superclass). This concept of sending a message to an object rather than directly calling a subprogram in order to perform an activity completes the change of focus from operations to objects. In the object model, the emphasis is on requesting services from an object, rather than on directly manipulating a data structure. In a pure object-oriented language, all interactions are coded as messages passed to objects, and there are no subprograms that are not encapsulated within a class. In fact, a method consists of message sends and control flow statements. (In section 5.5, we will see that even control flow expressions are message sends in Smalltalk.)

A message may be overloaded in the protocols of different classes. For example, both sets and sorted lists implement insertion and deletion of elements, returning the size of the collection, and so on. Message passing is characterized by loose coupling because the class of the receiver and the method it uses might not be known to the object sending the message (due to identifier polymorphism), and different objects may use different methods for a given message transparently. In addition, an object may only access data within another object by sending a message to that object. For example, we might obtain a collection of objects from another object without knowing whether the collection is a set or a sorted list (again, due to identifier polymorphism). We can insert and delete elements or obtain the collection's size by sending messages to the collection, but we do not know what code is executed and cannot access the collection's structure directly.

Class variables and messages
Some classes contain information that is conceptually associated with the class itself, rather than with any particular instance. Examples include

- class-wide constants, e.g. the minimum balance and interest rate for a savings account class, or the default size for a window class
- information about a class's instances, e.g. an index for accessing the instances of an employee class, given the employee number

It is not appropriate for any particular instance of the class to be responsible for this information, nor should it be stored in each instance. In addition to defining the structure of its instances, a class may also define class variables that maintain classwide information. In a traditional programming language, this information would be kept in global variables. Class variables allow us to encapsulate such information together with the class so that it is available wherever the class is imported, but not elsewhere, increasing cohesion and locality and decreasing coupling. Class variables are accessible to all instances of the class but are hidden from other objects, and are independent of the number or even the existence of instances of the class. We also use class variables for constants that are needed in the class's implementation, such as the values INIT_SIZE and EMPTY for the C++ CharStack example in the previous section. Different object-oriented languages use various constructs for defining class variables.

For example, consider a class Date that represents calendar dates. Information associated with this category of entities includes the number of months, the number of days in each month, the names of the months, and the number and names of the days of the week. We encapsulate this information within the Date class as class variables so that it can be made available to applications that use dates, and because it is
required in some of the class's methods. Instances of Date have direct access to this information, e.g. if a
date needs to print itself or respond with the date \( n \) days after itself.

Like instance variables, class variables are not directly visible to the clients of a class. We could define
an instance method that provides access to a class variable for other objects, but this would require sending
that message to a particular instance of the class. Requesting the number of days in March from a particular
date object is somewhat counterintuitive, to say the least. That information exists and should be accessible
even if there are no instances of Date in existence. A class message provides client access to information
stored in class variables or other class-specific operations. For example, the class UnionEmployee would
declare a class message to access the monthly contribution, which might be used by a Payroll class. Class
messages can also provide class-wide computations, e.g. the class Date might define a message that returns
whether a particular year is a leap year. This is a class message rather than an instance message because
that operation is not relevant to a particular date object, and the information necessary to compute the
answer is relevant to dates as a whole. Class messages are passed to the class itself, rather than to a
particular instance of the class. Type operations (recall section 3.4) such as querying the class's superclass
or subclasses are also class messages. Each class message is implemented by a class method defined within
the class.

Inheritance

In addition to supporting encapsulation and information hiding, the object model extends abstract data
types by introducing inheritance among classes. Classes that represent problem domain categories that are
related by generalization/specialization may share common protocol, instance variables, and methods,
without duplicating the specifications of these items in the definitions of each class. The more general class
is referred to as a superclass of the specific class, and the specific class is called its subclass. A subclass
inherits the messages, methods, and instance variables of its superclass automatically. That is, instances of
the subclass respond to superclass messages with superclass methods, and have values for superclass
instance variables. The definition of a subclass does not repeat the specification of its superclass's
components, and may add instance variables and protocol to those it inherits from the superclass. The
subclass may also override methods inherited from the superclass for particular messages with methods that
are appropriate for its instances. Object-oriented languages support these mechanisms by including syntax
for specifying a superclass when defining a class, and the language provides both the superclass and
subclass behavior and structure for subclass instances.

For example, a manager object is an employee object with additional information, such as the
employees managed and the manager's level, and additional behavior associated with this information. We
can represent this problem domain relationship directly in an object-oriented language by defining the class
Manager as a subclass of Employee. We code the common behavior (i.e., messages and methods for
handling those messages) and state information (i.e., instance variables) in the definition of the superclass
Employee. When we define the class Manager, we specify it as a subclass of Employee, and only state
the messages, instance variables, and methods specific to Manager. An instance of Manager responds to
the messages defined in both classes, and has values for the instance variables defined in both classes. If
Manager specifies a method for a message in the protocol of Employee, its instances respond to that
message with the Manager method.

In an object-oriented design, a class hierarchy represents the generalization relationship among classes,
in which a more specific class inherits behavior and structure from its superclass. The class hierarchy
represents the type structure of the problem domain or application, and is usually diagrammed as a tree with
a superclass as the parent of its subclasses. For example, figure 10 illustrates a portion of the class
hierarchy for a motor vehicle registration system:
As in this example, the class hierarchy describes the taxonomy of the categories of entities in the problem domain. We also use class hierarchies to represent a set of system components or resources, e.g., a class hierarchy of input/output streams and files. In general, the structure of the class hierarchy may be a tree, a forest, or an acyclic directed graph (with "multiple inheritance", in which a class can be a subclass of several superclasses).

When an object receives a message, it executes the method for that message defined by its class. If its class does not define a method, its superclass's protocol is searched for a method, and so on up its chain of ancestors in class hierarchy. We refer to the process of determining the method to invoke for a message as method binding. Object-oriented languages are strongly typed because the method binding process signals an error if it does not find a method for a message in the receiver's class or any of its ancestors.

Binding methods according to superclass relationships allows a superclass to define a "default method" for instances of subclasses that do not require more specific behavior. For example, the class MotorVehicle in figure 1.6 can define a method tax that calculates the vehicle tax based on the general instance variables it defines (e.g., the value and age of the vehicle, its gas mileage, etc.). If a particular descendant such as SportsCar or Convertible needs to use a different method for calculating tax (e.g., with a "luxury car" surcharge), it is defined in that class. When the message tax is sent to an instance of a motor vehicle class that does not define a method for that message, it uses the method inherited from MotorVehicle. An instance of a class that does define a method uses its class's method, without encountering the conflicting behavior in the superclass.

Inheritance provides a number of advantages for software engineering. It reduces the amount of code in the system because we only need to code the common interface and implementation of a superclass and its subclasses once, in the superclass. Inheritance reduces the amount of effort necessary to create a new class. In many cases, an existing class provides much of the behavior and structure necessary for the new class. The class designer can use this class as a superclass, write only the code necessary for the additional behavior and structure of the new class, and inherit the superclass's behavior and structure. We can also use a superclass to express commonalities among a set of subclasses. Defining new subclasses reuses the superclass code, and we can use inheritance to extend an application with new capabilities. With inheritance, the design better reflects the structure of the problem in which specialization relates categories of entities.

**Dynamic method binding**

*Dynamic binding* of methods for overloaded messages is an important characteristic of object-oriented languages. Determining which method to execute must be done at run time because the compiler might not know the class of the receiver of a message send, due to either dynamic typing or inheritance. For example, with inheritance, a variable of type MotorVehicle can refer to a motorcycle, a car, or a pickup truck because they are all motor vehicles. Since the referent of that variable may depend on run-time conditions (e.g., a user selection), selecting the method according to the class of the receiver must be done at execution time. In order for the method binding process to select the correct method, each object must contain an indication of its class. That is, in object-oriented languages, each object maintains its type identity explicitly.

Suppose that we have a class Owner that maintains a list of the motor vehicles owned by a particular person or organization, and we want to define a method that calculates the total tax due for that owner. The method for totalTax in Owner sends the message tax to each object on the list and keeps a running total to return. The actual method executed in response to the message tax depends on the type of the list element.
that receives it at execution time. Some objects will use their own class’s method, and others will use the method defined in their ancestor MotorVehicle. The totalTax method does not need a case structure of tests, one for each of the different motor vehicle classes, to determine which subprogram to call (nor does any other method that sends an overridden message to a motor vehicle object). It also does not need to know which classes define a method for tax and which inherit the MotorVehicle method. That is, the responsibility for binding the code to the message send has been shifted from the sender to the receiver. A method that passes this message does not even need to know all the possible classes whose instances can receive that message. In addition, we might add new subclasses of MotorVehicle as we develop and maintain the application, for example, ElectricCar, which has an environmentally conscious tax method. We do not need to modify existing code in the system that uses the MotorVehicle interface, such as the totalTax method for Owner.

The disadvantage of the method binding process is its cost in terms of space and time. The run-time system that performs method binding requires tables that maintain the methods defined in each class, and a pointer in each object to its class’s method table. Dynamic binding requires extra time for the method search when sending a message.

5. Summary and review

5.1 Control structure

Sequential control structures
• Nested expressions are evaluated from the inside out (except in some functional languages), and operator precedence, associativity, and parenthesization determine the order of nesting.
• In early languages, programmers coded flow of control explicitly using goto statements and conditional branches, but programs using goto statements were found to be difficult to understand, debug, test comprehensively, and decompose into units.
• Modern languages include single-entry single-exit control structure statements that specify the order of execution of statements, which we can nest to compose a structured program. The sequence, conditional, and iterative control structures are sufficient to code any single-entry single-exit program.
• A conditional control structure selects one of a set of actions to execute, depending on the value of a test expression. It can be either a statement or an expression that returns a value. Examples include the if-else and case statements.
• An iterative control structure specifies repeated execution of a statement or group of statements.
• A conditional iteration repeats the enclosed statements until the test expression is true (or false for some forms). Examples include the Pascal while and repeat statements, which test the exit condition at the beginning or end of the loop body, respectively. The C break and Ada exit statements permit us to locate the loop test and exit at any point in the loop body, and code more than one exit test in a loop.
• A generalized iteration specifies a list of initializations, an exit test, a list of update expressions, and a loop body. The C for statement is an example.

Subprograms and parameter passing
• Subprograms provide an abstraction mechanism that extends the language to reflect the problem domain or application.
• A subprogram definition specifies the name of the subprogram, the parameters, the local variables, and the statements executed when the subprogram is invoked.
• Subprogram invocation is a control structure, and the state of the caller must persist during the execution of the called subprogram so that it can be restored. The run-time stack is a last-in first-out list of activation records, each representing the state of a subprogram invocation that has begun but is waiting for completion of a subprogram it called.
• An activation record contains the subprogram’s arguments, local variables, return address, and dynamic link to the caller’s activation record to its enclosing scope, and the contents of the processor registers.
• With *pass by value*, the parameter is treated as a local variable that is initialized with the value of the argument expression. Pass by value can only be used for input parameters, and is inefficient when passing large objects.

• A parameter *passed by reference* is an alias for the argument, and operations the subprogram performs on the parameter are performed on the argument, including assignments. We use pass by reference for output parameters, or to avoid the overhead of copying a large argument.

• The Ada programmer indicates the meaning of a parameter directly by specifying its *parameter mode* as *in*, *out*, or *in out*, and the compiler chooses the implementation.

**Exceptions**

• An *exception* is a run-time error that results in an invalid computation state. Exceptions include hardware errors, system errors, logical errors, and application-specific errors. A robust system must detect exceptions and respond to them, rather than continuing with invalid results.

• We often need to propagate the occurrence of an exception from the subprogram that detects the error to the subprogram that can handle the exception. Since different activation records are involved, a *goto* will not work. We can use parameters or return values to propagate the exception through a series of invocations, but this tightly binds those subprograms and clutters the intervening subprograms with code irrelevant to their purposes. In addition, the caller of a subprogram that returns an error condition might not check for the exception.

• Languages that support exceptions such as Java, Ada and C++ allow us to define exceptions, signal the occurrence of an exception, and specify *exception handlers* for various types of exceptions for a block.

• When an exception occurs, the block's handler for that exception is executed. If it doesn't define one, its activation record is popped and its caller's handler is executed. If the caller defines none, the run-time stack is unwound until encountering a subprogram that defines a handler for that exception, and that handler is executed in the context in which it is defined. If no matching handler is found, the program terminates. If the handler does not terminate execution, control continues at the statement after the block that specified the handler which was executed.

**Concurrency**

• Some programming languages permit the programmer to define an application as a collection of concurrent *tasks* or *processes*, which can execute independently.

• Programming languages that support concurrency include features for defining processes and specifying the code that they execute, for starting, suspending, re-activating, and terminating processes, and for synchronization and communication among processes. These operations are usually implemented using the facilities of the operating system.

• Processes can synchronize their operations using *semaphores*. A process signals that an event has occurred, and another process can wait for that signal. If a process performs a wait for a signal that has not occurred, it is suspended until the signal is performed.

• Processes can synchronize their operations using *critical regions*: Only one process can execute a critical region associated with a particular shared variable at a time.

### 5.2 Name structure

**Identifiers and declarations**

• An *identifier* is a name *bound* to an object, constant, subprogram, parameter, or type, which is its *referent*. An identifier can refer to different program entities in different parts of the program, and different identifiers can refer to the same entity.

• A *static binding* is fixed before execution (e.g., the location of a global variable), and a *dynamic binding* can change during execution (e.g., the location of a local variable).

• A *declaration* is a statement that introduces a name into a scope, and may associate attributes such as a type with that name. We must declare an identifier before using it so that the compiler can detect typographic errors and invalid uses of the identifier.
• A definition is a declaration that also creates the associated entity.
• A forward declaration introduces an identifier, but does not specify creation of its referent, so a
definition of that entity must occur elsewhere in the program. Forward declarations permit calling
a subprogram before defining it, using a program entity in more than one compilation unit, coding
mutually referential definitions, and designing with information hiding and top-down functional
decomposition.
• The symbol table contains the binding and attributes for each identifier. The symbol table is usually not
present at execution time in a compiled language, but is needed during execution in an interpreted
language.
• With static typing, a type is associated with every object, constant, and subprogram identifier. The type
of a subprogram is its argument signature, which lists its parameter and return types. The compiler uses
this information to catch type errors, perform implicit conversions, and allocate objects.
• With dynamic typing, types are not associated with identifiers, so identifiers are polymorphic. A
variable can refer to any object, and the type of the referent of an identifier can only be determined at
run time. The compiler cannot catch type errors, so objects must contain an indication of their types.
Object allocation, type checking, and binding of overloaded subprograms must be performed at run time.
Dynamic typing provides more flexibility, but reduces efficiency and safety.
• An alias declaration introduces a synonym for an existing entity, but does not create a new entity.

Scope
• A scope is a section of the program that defines a name space. We can use scope to structure a program
and partition its name space, allowing an identifier to refer to different objects, subprograms, or types in
different parts of a program without causing a name conflict.
• The scope of a declaration is the part of the program in which the declared identifier refers to that entity.
A global declaration or identifier is visible throughout the program, and a local declaration or identifier
is restricted to some scope in the program.

Nested scopes
• A compound statement (e.g., delimited by begin and end) that includes declarations is a block, and the
scope of those declarations is the block. In a block-structured language, blocks can be nested within
other blocks and the name space of a program has a hierarchical structure.
• A name declared in an enclosing block is visible in a block unless the block also declares it, which hides
declaration in the enclosing scope within the block. The scope of a declaration is the block in which it
occurs and all enclosed blocks that do not declare that name, and two declarations of the same identifier
have disjoint scopes.
• The process of identifier resolution determines which declaration an identifier use refers to, and selects
the declaration in the nearest enclosing scope. If there is no declaration of that name in any enclosing
scope, the use is an error.
• Nonlocal identifiers in a subprogram can be bound in the environment of the subprogram definition or in
the environment of the call. With static scope, nonlocals are resolved according to the environment of
the subprogram. The scope of an identifier and the referent of an identifier use are determined by the
nesting of scopes in the source code, and do not vary.
• Block-structured languages employ static scope. To implement block structure, the compiler creates an
activation record for each subprogram or block, and allocate its local variables within that activation
record. The compiler translates a local variable use as an offset in the activation record.
• When a subprogram is called in the scope in which it is defined, the static link points to the activation
record of its caller. If a subprogram is called in a scope nested within the scope of its definition, its
static link is set by following n static links beginning with that of the caller, where n is the static distance
between the call and the subprogram definition.

Programmer-defined visibility
• When a subprogram accesses a nonlocal identifier, a side effect that is not visible in an invocation
occurs, and the subprogram is not an independent unit.
• Block-structured languages provide a hierarchical name space structure and implicit access to names in enclosing scopes so that names can be shared. However, they do not permit controlling visibility explicitly for each name or for an individual scope.

• Languages that support modules allow us to divide the set of names in a module into an interface visible outside that unit and an implementation private to the unit, and to explicitly import names declared in the interfaces of other modules. Name visibility relationships are explicit, and hidden side effects, unintended access, and name capture cannot occur.

Overloading
• Most languages overload operators for their built-in types so that programmers can use the same operator symbol for conceptually similar operations on different types. Many contemporary languages support this capability for subprogram names, and we can define more than one subprogram with the same name in the same scope if each has a distinct argument signature.

• The compiler resolves an invocation of an overloaded subprogram name by matching the argument signature of the invocation with those of the subprogram's definitions.

• Implicit conversions can cause an invocation to be ambiguous because there can be conversions from the argument signature of the call to the argument signatures of more than one subprogram definition.

• A dynamically typed language can support overloading, but the process of binding a subprogram definition to a call occurs at execution time.

• The ability to overload subprogram names is essential for object-oriented languages.

5.3 Data structure

Data objects
• A data object consists of a region of storage in which a value is encoded, and represents a problem domain entity or a component of the system or application.

• Each data object is an instance of a data type, which defines the operations and storage structure of its instances.

• A variable is an association between a name and a data object consisting of a value stored at a location.

• A variable name use may refer to the object's value, or r-value, or its location, or l-value, depending on the context. An l-value is necessary to modify the object, e.g. for the left side of an assignment or a reference parameter.

• Named constants improve a program's readability. A constant differs from a variable because it cannot be modified, i.e. the identifier's l-value is not available.

• A symbolic constant declares a named constant whose value can be computed by the compiler, and usually must be an instance of a built-in type with a literal notation. A read-only variable denotes a constant object of any type, whose initialization may have to be calculated at execution time.

• We create an object with a variable definition or a dynamic storage operation, which instantiates a type and allocates storage for it. The lifetime of an object is the duration of execution from its creation to its destruction, when its storage is deallocated.

• The lifetime of an object at execution time and the scope of an identifier in the source code are different issues, and an object may exist when it is not accessible or have more than one name.

• Allocation of an object and initialization of that object, which places a valid value in that storage, are separate operations. No operations (besides assignment) are valid for an object that is not initialized. An assignment changes the value encoded in an existing object and requires an l-value.

Storage allocation policies
• The lifetime of a statically allocated object is the entire execution of the program.

• An automatically allocated object exists for a particular execution of the subprogram or block in which it is declared. The compiler allocates it in the subprogram's activation record, and accesses it by its offset, possibly after following a chain of static links.
• A *dynamically allocated* object is created when indicated by the programmer, and deallocation of
dynamic objects may be under programmer control or may be performed automatically, depending on
the language.

• Dynamic objects are allocated separately from the stack in the *heap* or *free store* by a run-time system
provided by the language system. Efficient management of heap storage is complex, and designers have
developed several algorithms and data structures to manage it.

• With *programmer-controlled deallocation*, three errors that are difficult to debug can occur: omitting
deallocation (a *memory leak*), using a *dangling pointer* to a deallocated object, or deallocating an object
more than once.

• *Automatic storage reclamation* detects when an object can no longer be referenced by the program and
reclaims its storage. Two common techniques are *reference counting*, which maintains a reference count
stored with each object, and *garbage collection*, which marks all accessible objects and then reclaims the
rest of storage. Automatic deallocation is more convenient and less error-prone for the programmer, but
results in less efficient executable code.

**Data types**

• A data type models a category of entities in the problem domain or components of the system or
application. The definition of a data type includes specification of both the valid values for instances,
and the operations that may be performed on instances. These should mirror those of the category the
type represents. The use of types extends the language so that the structure of the problem domain and
application is reflected in the program.

• Every language specifies a set of *built-in types* such as the logical, character and numeric types, which
are usually types built into the hardware or motivated by the intended problem domain. The language
provides constructs for creating instances, writing values, and performing basic operations and
input/output on instances.

• Numeric types include integer and floating-point types, which are usually represented in two's
complement and IEEE floating-point format, respectively. Some languages support more than one size
for numeric types.

• Most languages provide a logical type, e.g. *boolean*, for the values "true" and "false". We can use the
logical operations *and*, *or*, and *not* with booleans, comparisons and other tests return booleans, and
boolean values control conditional evaluation and iteration.

• In most languages, character is a built-in type, and strings are treated as fixed-size arrays of characters.
In Pascal, the size of a string object is fixed at compile time and different sizes are different types. In
Ada and C, the size is fixed at creation time, we can allocate strings dynamically, and string parameters
are the same type.

**Defining types**

• Modern languages support defining types and using them to create objects, specify parameters, and
define other types.

• The instances of a type are *first-class* objects if the language provides a complete set of features for
using those objects, e.g. initialization, assignment, equality comparisons, input and output, and a literal
notation for writing objects. The language must also support using of the type as a parameter or return
type to provide its operations, or as the base type of another type.

• A *type constructor* is a language feature that defines a kind of type by listing its values, restricting an
existing type, or describing the structure of composite objects.

• A *type definition* gives the name of the type, the kind of type, the structure of its instances, and the type's
operations if the language supports abstract data types. The translator creates a *type object* that
represents this information, which is a compile-time construct in compiled languages.

• Many languages provide *type operations* that return information about the type itself, rather than a
particular instance, such as the size of its instances or the smallest and largest value of an ordered type.
Programmer-defined types

- A type alias gives an alternate name for an existing type, and is used to give a name that reflects the intended semantics for instances of the type. In Pascal and C, the new type name is a synonym for the existing type.

- A pointer type specifies a base type, and the value of a pointer object is either a reference to an instance of the base type, or a special value that indicates that the pointer does not refer to an object. The type object includes the base type and its size for dynamic allocation.

- Pointer operations include dereference, which returns the referent and can be used as an l-value, allocation and deallocation of dynamic referents, assignment, and equality test.

- Pointer assignment causes both pointers to refer to the same object (rather than copying a referent), and pointer comparison tests whether the pointers refer to the same object (rather than comparing referents).

- A pointer object contains the address of its referent. C provides address arithmetic for pointers, which can be unsafe.

- An enumerated type models an entity or attribute for which there is a fixed set of values or states, and is defined by listing names for the values. The value names are symbolic constants declared in the scope in which the type definition appears (or in the enumeration scope in Ada), and provide a literal notation for instances. The type object specifies the size, the value names, and their encoding.

- The enumerated type is first-class in Pascal and Ada since they support initialization, assignment, equality tests, literals, enumeration parameters, and enumeration base types. Each enumeration in ANSI C is a separate type, but any integer can be assigned to an enumeration object, and enumeration values can be used as integers.

- A subrange type represents a category of entities with a restricted range of values, and its definition gives the type name and the value limits. The type object maintains the size, the base type, and the limits. The language ensures that a value assigned to a subrange instance is within the type's limits, possibly involving run-time checks. The subrange type provides safety and reliability, at the expense of execution-time efficiency. The operations and literal notation of the subrange type are those of its base type.

- An array is a fixed-size, homogenous, indexable sequence of elements that is treated as a composite object. The definition of an array type gives the type name, the base type, and the index type (or the subscript limits), and the type object specifies the base and index types.

- Subscription is the selector operation for arrays, and returns a reference to an element that may be used as an l-value or a r-value. Subscription is indicated by square brackets in Pascal and C, or parentheses in Fortran and Ada.

- Pascal introduced the array type with any discrete, ordered, finite type as the index type, but array limits are fixed at compile time, and arrays with different limits are different types. Ada introduced the unconstrained array type and array type attributes to facilitate defining dynamically sized arrays and subprograms that operate on arrays. C does not support an array type.

- Many languages support coherent operations such as assignment and equality comparison for arrays, and we can define additional operations by using an array parameter type.

- A record is a composite object consisting of a collection of components or fields, each with a name and a type. The components store values for the attributes and relationships of an instance of the category of entities the record type represents.

- A record type definition gives the type name and the names and types of its components. A record instance has a value for each component, and is typically stored as a sequence of the values for its fields. The type object includes the field names and types, the size, and the field offsets.

- The component selection operation accesses a field in a record object by the field name. The operation is usually indicated by record.field, and can be used as an l-value or a r-value.

- Support for coherent operations on records is incomplete in Pascal and C, while Ada supports record initialization, assignment, equality tests, literals, subprogram parameters, and operator overloading, so records are first-class.

- A union type specifies a set of alternate substructures of different types, and an instance can contain an object of one of these types. A Pascal variant record type specifies a set of fields and a variant part containing a tag component and a set of substructures, and only one of the substructures is stored in an
instance at any time. We access variant record fields using component selection. The type object contains the size of an instance, the nonvariant field names and types, and the type descriptions of the variant parts.

- The variant record type is unsafe in Pascal because we cannot assign a tag and the associated variant part to an instance coherently, so the tag and the variant part in an object might not be consistent. Ada variant records are type-safe because the language supports coherent assignments and specifies that the type tag must be checked when a field is accessed. The C union does not indicate which substructure is present, and is inherently unsafe.

- Abstract data types allow us to explicitly associate a type's operations with the definition of its storage structure, and to hide that structure from users of the type.

- Some languages support subprogram parameters to permit coding higher-level algorithms as subprograms, and reusing them via an invocation with a particular subprogram as an argument. However, function objects are not first-class in procedural languages.

**Type checking and conversions**

- A type specifies a domain over which operations on instances of the type apply, and a subprogram with a parameter of that type depends on its argument being an object in that domain for correct operation.

- A strongly typed language enforces the domain requirements of assignments, operator operands, and subprogram arguments, which provides safety and reliability. Type checking can be done by the compiler in a statically typed language, but must be done at execution time for a dynamically typed language.

- In a weakly typed language, type checking is incomplete or can be circumvented.

- Overlapping domains complicate type checking: No distinction is made between merged domains, instances of a constrained type can be used as instances of the base type (but not vice versa), and some overlapping domains require implicit conversions because instances of the types are represented differently.

- A type cast changes the type of an object from the point of view of the compiler, but does not create a new object.

- A type conversion creates a new object whose value is equivalent to that of the converted object.

- Casts and conversions can be invoked explicitly by the programmer or implicitly by the compiler, and the latter are referred to as coercions.

### 5.4 Abstraction mechanisms

**Software complexity**

- Software system complexity results from the characteristics of the problem domain system, from factors external to that system, and from changes in the problem domain or system requirements over time.

- Designing a software system involves determining how much problem domain complexity to represent, and how to reduce the inherent complexity of the system.

- With large systems, the total complexity increases faster than the number of components increases due to interdependencies among the components and communication among the design team.

- Like all humans, software designers use classification, decomposition, composition, and abstraction to manage complexity and structure information.

**Abstraction in computing**

- Early programming languages defined basic programming abstractions such as symbolic names, algebraic expression syntax, control structures, and subprograms.

- Higher-level languages have defined a progression of programming abstractions from abstraction of operation, to abstraction of structure, to abstractions that encapsulate both structure and operation.

- Abstractions provide a higher-level view of system structure, supply units for decomposing the system and organizing the development effort, allow independent design of individual components, and present opportunities for code reuse.
• Programming abstractions extend the language to mirror the problem domain, and therefore simplify programs, improve readability, and reduce errors.

• We can recognize abstractions either by top-down decomposition of the structure of the system, or by bottom-up recognition of multiple occurrences of processing or structure.

• A programming abstraction is an encapsulation that groups processing and/or structural units, and provides information hiding by separating its public interface from its implementation, which is not visible to clients.

• A subprogram is an abstraction whose interface consists of its name and argument signature, a description of the subprogram’s operation, restrictions on argument values, and the errors it can signal. Its implementation is its algorithm and local variables. Subprograms support the design methodology of top-down functional decomposition.

**Modules**

• A module is a source code unit that can be compiled individually, even though it refers to program entities defined in other modules.

• The linker combines an application’s object files into an executable program, and resolves references among identifiers that are used in one module but defined in another.

• The interface of a module is a set of forward declarations for the subprograms, objects, constants, and types exported by the module.

• The implementation of a module gives the definitions of the program entities whose names its exports, and possibly definitions of other auxiliary entities that are local to the module.

• An import declaration makes the interface of the named module visible in the module in which it appears. Implementations cannot be imported.

• A module defines a scope for the identifiers declared in the module. Module scope objects are statically allocated, but their names are hidden within the scope of the module (unless they are exported).

• Modules provide a level of structure between the level of the individual subprogram and that of the entire application, and allow the designer to group related subprograms and the data they manipulate as a unit.

• "Parnas's rule" states that the designer of a module should provide clients with exactly the information necessary to use the module, and should provide the implementer with exactly the information needed to code that module.

• Coupling describes the degree of interdependence between the modules in a system, and cohesion describes the degree of binding among the elements in a module. A good design minimizes the coupling among modules and maximizes the cohesion within each module, so that we can write and debug modules independently, and reuse them in similar applications.

• Modula-2 and Ada support module interfaces and implementations, import declarations, import/export scope rules for external names, and name qualification to resolve name conflicts among imported interfaces. A Modula-2 module is divided into a definition module giving its interface and an implementation module. In Ada, a module interface is called a package specification, and a module implementation is called a package body.

• Modula-2 and Ada support separate compilation of a module’s interface and implementation. The language system maintains a module database to translate import declarations and manage compilation dependencies.

• C supports separate compilation, but not import/export scope. Each source code file defines a level of scope enclosing function scopes, which is nested within the global scope of the entire program, in which all exported names are declared. Names that are private to a module are declared static, and forward declarations are indicated by extern.

• We code a C module as a header file containing its interface, and a code file giving its implementation. A client includes the text of a header file with a preprocessor directive. The C compiler has no knowledge of module names, import relationships, or compilation dependencies, and cannot support name qualification.

- 77 -
Abstract data types

- An abstract data type encapsulates the operations and storage structure of a type, and allows the
designer to control access to those components. Languages that provide first-class support for abstract
data types allow defining a complete set of operations for a type.
- The interface of an abstract data type consists of subprogram declarations for its public operations. The
type must provide initialization operations so that all objects have valid states, and no operations should
be performed on uninitialized objects.
- To provide representation independence, the designer of the type hides its storage structure. This
decouples the interface and the implementation of the type, and allows the designer to change the type's
structure without affecting clients.
- The implementation of an abstract data type consists of declarations that describe its storage structure
and subprograms that define the type's operations. It may also define private subprograms, constants,
objects, and types that aid in coding the type.
- With object-based decomposition, we structure a system in terms of categories of problem domain
entities or system components and their behavior, rather than according to the tasks it performs, and
represent each as an abstract data type. Parnas's rule and the principles of cohesion and coupling are
also important in designing abstract data types.
- Ada packages support abstract data types because a package specification can contain a private section
not visible to clients. This information is available to the compiler, and usually gives storage structure
details necessary to translate instantiation of the type. Object-oriented languages also support abstract
data types.
- Some abstract data types, e.g. for variable-sized collections, allocate dynamic subobjects and require
initialization and release operations to hide that implementation from clients.
- The interface, storage structure, and subprogram definitions for a particular kind of collection are the
same, irrespective of the element type. A generic abstract data type defines the interface and
implementation of a collection type together with a type parameter for the element type. The compiler
generates a collection type with an actual element type, e.g. a set of dates, when necessary. Ada generic
packages and C++ template classes support generic abstract data types.

Classes

- A class is an abstract data type with three additional features: class information and operations,
inheritance among classes, and dynamic binding of class operations.
- An object is an encapsulation of state and behavior that represents a problem domain entity or system
component.
- Every object is an instance of a class that defines its operations and state information.
- A message is an operation that the instances of a class can perform.
- The protocol of a class lists the messages that make up its interface.
- The instance variables of a class describe the storage structure of its instances.
- A method is the code that implements a message.
- An identifier, subprogram or object is polymorphic if it can have more than one type, e.g. an identifier in
a dynamically typed language or an overloaded subprogram.
- A pure object-oriented language supports object-oriented features and enforces the object-oriented
paradigm. A hybrid object-oriented language extends an existing procedural or functional language with
the object-oriented features, and permits both object-oriented and procedural programming.
- The fundamental operation in object-oriented programming is sending a message to another object, the
receiver. The receiver responds by executing the method defined by its class or inherited from an
ancestor.
- Message passing provides loose coupling because different objects may use different methods for a
message, the sender may not know the receiver's exact class, and an object can only access another
object's data by sending it a message.
- Class variables maintain information associated with the class as a whole, but not stored in each
instance. This encapsulates that information within the class, rather than using global variables.
• *Class messages* provide client access to class variables (including type information) or other class-specific operations, and are implemented by *class methods*.

• *Inheritance* is the relationship between a *subclass* representing a problem domain category which is a specialization of the general category represented by its *superclass*.

• A subclass *inherits* the protocol, instance variables, and methods of its superclass. The subclass may also introduce additional messages and instance variables, and *override* superclass methods with subclass-specific behavior.

• Inheritance defines a *class hierarchy* that represents the type structure of the problem domain, system resources, or application.

• Inheritance facilitates reuse of existing code and extendibility of the system, i.e. when a new subclass is defined.

• When an object receives a message, the *method binding* process searches for the method to execute beginning in its class's protocol, and continues up the class hierarchy if necessary.

• Since the compiler may not know type of the receiver of a message, object-oriented languages must employ *dynamic method binding*, in which the method to invoke is selected at execution time.