The semantics of a language construct or program describes its meaning. We can formulate this description in terms of a formal specification of a model of computation, or an informal model based on the operation of a typical computer. We will discuss programming language features from the latter point of view. We will not consider lexical elements, syntax, or the related issues of parsing and readability, but will concentrate on the semantics of and purpose for various constructs. Furthermore, we consider each language feature for what it means and why we use it, rather than how it is manifested in a particular language. We will divide our discussion of programming language features into three sections: flow of control, the name space of a program, and the structure of data. We will also survey the abstraction mechanisms used to express programs and partition large software systems, following the progression from abstraction of operation in early programming languages to contemporary abstractions that include both structure and operation.

A grasp of programming language semantics is essential for the correct use of a programming language. You can avoid programming errors or recognize them more quickly with this knowledge. For example, you must understand the distinction between issues within the name space of a program, and those that have to do with objects and their interactions during execution. Similarly, if you do not know how storage is allocated on the stack and the heap, you can make errors that the compiler does not catch that result in effects whose causes are difficult to determine (especially in C and C++). An understanding of data types, encapsulation, and first class support for instances of types is necessary to comprehend the object-oriented concept of a class. Concepts such as scope, binding time, static and dynamic typing, and their interaction with inheritance are intrinsic to the use of object-oriented languages.

This hadnout covers the following topics:
- sequential control structures (1.1)
- subprograms and parameter passing (1.2)
- exception handling (1.3) and concurrency (1.4)
- identifiers (2.1), and declarations and definitions (2.2)
- static scope, and the visibility of identifiers (2.3 and 2.4)
- programmer-defined visibility (2.5)
- overloading (2.6)
- data objects (3.1) and object allocation policies (3.2)
- the purpose and semantics of data types, and primitive types (3.3)
- defining types, and the type object (3.4)
- programmer-defined types, including pointers, enumerations, arrays, records, unions, abstract data types, and subprogram types (3.5)
- type checking, casts, conversions and coercions (3.6)
- sources of complexity in software systems, and strategies for dealing with it (4.1)
- abstraction, encapsulation and information hiding, interface and implementation, and subprograms as abstractions (4.2)
- modules, the linker, the interface and implementation, scope, procedural design (4.3)
- abstract data types, representation independence, and object-based design (4.4)
- an introduction to objects, classes, messages, inheritance, and dynamic binding (4.5)
1. Control Structure

1.1 Sequential control structures

Sequence control within expressions and among statements

Generally, a program is not just a set of independent operations that can be performed in any order. A language must provide constructs for indicating the order in which actions are performed, or for specifying that the order is unimportant. In procedural languages such as Pascal and C, we indicate the order of execution by nesting of expressions and sequencing of statements. Some languages also provide mechanisms for specifying synchronization with concurrent processes.

A nested expression applies a function or operator to some arguments which, in turn, can be the results of other function or operator calls. For example:

```c
/* a nested expression in C */
abs(cos(x + 3 * y) / z)
```

The order of nesting is affected by operator precedence, associativity, and parenthesization. For example, in the subexpression `x + 3 * y` in the previous example, the multiplication operation is nested within the addition, not vice versa. In traditional languages, the innermost function or operator invocations in an expression are evaluated first to provide the arguments to the function or operator call in which they are nested, e.g. `cos()` is executed before `abs()` in the preceding example. The subprogram that contains the expression is suspended, the inner function is invoked, and its return value is passed back to the caller, which uses that value as an argument to the next most nested function call in the expression, and so on.

The order of evaluation of the arguments of a particular function call might or might not be guaranteed by the language definition. It is not specified and is left to the implementation in Pascal. In C, the `&&`, `||` and `,` (comma) operators always evaluate their left operand first, but the order is undefined for all other operators and for functions. The order is irrelevant if evaluating an argument expression does not cause a side effect that would affect evaluation of another argument expression, such as modifying a global used as an argument. A Pascal expression consisting entirely of operator applications performs no side effects, and can be executed in any order as long as each argument is evaluated before the call that uses it. For example, in the expression `a * b + c mod d`, it doesn't matter whether the multiplication or modulus operation is executed first. However, if a Pascal function `func` modifies its argument (i.e., the argument is passed by reference), then the result of the expression `func(x) + x` is indeterminate. The programmer must be particularly aware of this situation in C because many of its operators such as `*=` and `++` cause side effects.

A statement is executed for its side effect, i.e. to modify a program variable or perform input/output. Since statements do not return a value, we cannot use nesting to indicate the desired order of execution. In procedural languages, a program is a series of statements to be executed in sequence. The use of statement sequencing as the default flow of control reflects the fetch-execute cycle implemented in the hardware.

In contrast, functional languages such as LISP and ML do not include statements, and the entire program is written as a nested expression. In contemporary functional languages, no functions can cause side effects, so expressions can be evaluated from the outside in. An argument expression is evaluated only when its value is needed (often termed pass by need), and the resulting value is saved locally for further uses in the function. This is termed lazy evaluation, in contrast to the strict evaluation used in most procedural languages, which always evaluates every argument. In fact, a conditional in a traditional language is also evaluated lazily from the outside in, since either the `then` or the `else` clause will not be evaluated. Some procedural languages also use lazy evaluation for certain operations, such as "short circuit" logical operators, such as the `&&` and `||` operators in C, and the `and` and `or` operators in Ada.¹ This characteristic is useful for "guard conditions", for example,

```c
/* using lazy evaluation for a "guard condition" in C */
while (index < size && arr[index] != key)
    index++;
```

¹ Lazy evaluation could also be used in arithmetic expressions in certain cases, e.g. consider evaluation of the expression `a * (b + c + d)` when `a` is 0.
The **goto** statement

The **goto** statement is the simplest control flow statement, and mirrors the unconditional branch instruction built into all computers. Early higher-level languages included this feature because programmers were familiar with using it to control flow of execution in assembly language, and because the concept of structured control constructs had not been developed. To support the **goto** statement, the language provides a way of labeling statements, and a **goto** statement transfers control to the specified statement directly. The programmer can use a conditional branch (an if ... **goto**) to direct control flow, as in assembly language. Backward jumps are used to create loops, and forward jumps implement conditional execution.

The **goto** statement has a number of disadvantages. It has a negative effect on readability because the structure of the program is obscured by the way control flow is expressed. There is no syntactic indication of the type of control pattern used, i.e. whether an if ... **goto** is the beginning of a conditional or the end of a loop. The **goto** statement encourages the programmer to write code with tangled nonlocal flow of control, and such a program cannot be read from beginning to end to determine its behavior. Programs that use goto often cannot be divided into independent units, and a program may have a large number of possible control paths to be analyzed. In particular, there may be several ways to reach a given point in the program, making it difficult to describe the state of the computation at that point. For these reasons, programs using **goto**s are much more difficult to understand, debug and maintain, and are difficult to test comprehensively or prove correct.

**Control structures**

A **control structure** is a language construct that defines the order of execution of expressions, statements or groups of statements, usually based on execution-time values of program variables, and has a single entry point and a single exit point. Control structures are typically statements in procedural languages, but can also be expressions. In a classic paper, Bohm and Jacopini [Boh66] showed that all single-entry single-exit control flow patterns can be expressed as nested applications of three basic control structures. These are:

- the **sequence** structure, i.e. a series of operations
- the **conditional** structure, e.g. an if ... then ... else statement
- the **iterative** structure, e.g. a while loop

This result showed that the **goto** statement is not essential in a programming language.

A conditional branch is sufficient for constructing any pattern of control flow, as is done in assembly language. However with **goto**s, there is no restriction on the location of the target statement, so locality is not preserved. On the other hand, each control structure indicates the control flow pattern used, defines a delimited scope within the program, and has a single entry point and a single exit point. Languages that support control structures allow us to compose a program as a group of modular units that are sequenced or nested. In this way, the structure of the program reflects the dynamic flow of control. Such structured programs are also easier to decompose and understand than programs that use arbitrary transfers of control with **goto**s.

A major contribution of Algol (inherited by all its descendants) is support for these control structures, and for nesting them to any depth. To support nesting a sequence of statements within a conditional or iterative control structure, Algol introduced the **compound statement**, which encloses a group of statements within the delimiters **begin** and **end**, and is treated as a single statement. This organization of the code gives Algol programs a hierarchical structure, rather than the linear structure of Fortran or Basic programs.

The **conditional control structure**

A **conditional control structure** includes a test expression and two statements or expressions, one of which will be executed, depending on the value of the test expression. With conditional branches, the target of the branch can be anywhere in the program. A structured conditional has exactly one exit point, so we can consider the conditional structure and the enclosed statements or expressions as a unit. The test expression is usually a boolean-valued expression. The exception is C, in which we can use any arithmetic or pointer expression, and a nonzero or non-NULL result is treated as "true".

The conditional construct can be either an expression or a statement. All procedural languages since Algol provide a conditional statement. For example, in Pascal,
{ a conditional and a sequence nested within a conditional in Pascal }
if num < 0 then begin
  if num <> 0
    writeln('number is negative')
  end
else begin
  total := total + num;
  count := count + 1
end

In this example, the begin and end in the then clause of the conditional are necessary so that the else clause is associated with the outer conditional, and the begin and end in the else clause allow nesting the sequence of statements within the conditional. For a conditional statement to be useful, the enclosed statements must cause side effects, i.e. assignments, data transfers or transfers of control, since the conditional returns no value.

In functional languages, there are no statements, and the conditional control structure is an expression that returns the result of evaluation of the selected subexpression. The simplest form is an if-then-else that evaluates either the then or else expression, and returns that result as the value of the conditional. Two examples are the C ?: operator and the Algol if-then-else:

/* the if-else conditional expression in C and in Algol */
x = y + (x < 0 ? 0 : x);
x := y + (if x < 0 then 0 else x)

We can also regard "short circuit" logical operators, such as && and || in C and and and or in Modula-2, as conditional expressions because the right operand might or might not be evaluated. Since such a conditional is an expression, it can be embedded in another expression, and can communicate with other operations via its return value.

Many languages also support a conditional construct that contains an arbitrary number of conditions, each associated with an expression or statement to execute if that condition is true. Examples include the Lisp cond, and the Modula-2 and Ada if statement with elsifs, each of which evaluates the conditions in the order given. For example,

-- the Ada if statement
if OverallAvg > 90 or ExamAvg > 85 then
  Grade := 'A';
elsif OverallAvg > 80 then
  Grade := 'B';
elsif OverallAvg > 70 then
  Grade := 'C';
elsif OverallAvg > 60 and ExamAvg > 50 then
  Grade := 'D';
else
  Grade := 'E';
end if;

In an Ada or Modula-2 if statement, any number of statements may appear after then or else without a begin because there is a closing delimiter for the group of statements in the statement syntax, namely elsif, else, or end if.

We sometimes need to use the value of a variable or expression to select one of a number of actions, each associated with a particular value. We can use a series of nested ifs or an if with elsifs, but this does not reflect the logic as clearly as an explicit switch on the value because it imposes an ordering on the tests and does not indicate that the values are mutually exclusive. Pascal introduced the single-entry single-exit case statement, which gives a discrete-valued expression and labels each action with the value(s) that specify its execution. In many languages, the programmer can associate an action with several values (e.g., separated by , in Pascal or | in Ada), or with a range of values (e.g., using .. in Modula-2 and Ada). A statement to execute when the expression value does not appear in any of the cases is essential for safety.
and convenience (i.e., so that the programmer does not have to give actions for every possible value). The lack of this feature is a major defect of the Pascal case statement, and is the cause of a number of incompatible extensions to the language (using else, otherwise, etc.). Ada improves on the safety of the Pascal case by providing the others clause, and by requiring that the case statement specify actions for all possible values of the switch expression if there is no others clause. The C switch is unusual and is less safe because each action "falls through" to the next, which is rarely desired, so the programmer must remember to place a break at the end of each action.

The iterative control structure

Instead of creating loops with gotos, modern programming languages provide a single-entry single-exit iterative control structure, which specifies repeated execution of a statement or group of statements, and whose beginning and end is delimited in the code. Even when an iterative control structure can specify more than one exit test within the loop body, each exit always transfers control to the statement directly after the control structure so that the entire control structure has a single exit. Most languages do not permit a transfer of control into an iterative control structure with a goto (C is an exception). The iterative control structure is translated to the same machine code as a backwards branch, but it is safer and easier to understand and maintain because it is more structured. Like all control structures, we can nest an iteration within a sequence or a conditional, and vice versa. Two kinds of iterative control structures are common, conditional iteration and counter-controlled iteration.

A conditional iteration control structure repeats execution of a statement or series of statements until a given condition is satisfied. A loop with a preloop test tests the loop exit condition before executing the loop body (like an if-goto at the top of the loop), and does not execute the body if the test is not satisfied initially. The Pascal and C while statements employ a preloop test. For example:

{ a Pascal while statement }
while num > 0 do begin
  num := num div 5;
  count := count + 1
end

With a postloop test, the condition is evaluated after execution of the body, and the loop is always executed once. Examples include the Pascal repeat statement and the C do while statement. The while loop is more common because we usually need to handle empty lists or files, or errors in accessing or setting up the data. The loop test may be satisfied when iteration is to continue (e.g., the Pascal while do), or when control is to exit (e.g., the Pascal repeat until). Clearly, the loop body must modify a variable used in the loop test so that the loop will terminate.

Restricting the loop test to the beginning or end of the loop as in Pascal can cause awkward code idioms, and is not necessary to maintaining the locality and structuredness of the program. For example, a read loop that exits upon reading a sentinel value or reaching the end of the input file cannot test for the exit condition until a read has been performed. Without a loop exit construct (or a goto), the programmer must unwind the loop such that part of the first iteration precedes the iteration control structure, or use an extra test within the loop. In addition, many loops have more than one logical exit. For example, an array search loop exits if it finds the key or encounters the end of the array. In some cases, the conditions can be combined with a logical and, but in others the validity of one test determines whether another should even be performed (i.e., lazy evaluation is called for). If so, the Pascal programmer must set a boolean flag in the loop, and then test it in the loop condition. If several conditions must be tested, a number of boolean variables are necessary, and the logic of the code becomes obscured.

C provides the break statement, which exits the enclosing control structure. Ada supports a generalized loop construct enclosed in the keywords loop and end loop, in which an exit can appear at any point within the loop body. For example:

-- an Ada loop with a loop exit
loop
  Put("Enter a number:");
  Get(Num);
  exit when Num = 0;
  Total := Total + Num;

- 5 -
Count := Count + 1;
end loop

An Ada loop can also contain multiple exits, each of which transfers control to the statement after the loop. We can also use an Ada exit to exit from several nested loop by specifying the loop label of the outermost loop to be terminated, which is useful for handling error conditions. In any case, exiting a loop still transfers control to the statement immediately following the loop. C also includes the continue statement that transfers control directly to the loop test of the nearest enclosing loop.

Some languages provide a more general form of loop specification that we can use to express both conditional iteration and counter-controlled iteration. A generalized iteration gives initialization expressions, a loop test, and update expressions. The loop test is evaluated before the loop body, and the update expressions are executed after the loop body. This construct is provided by the C for statement, in which we can give more than one initialization or update expression separated by commas. For example,

```c
/* the C generalized for loop */
/* reversing the order of a linked list */
for (pNewlist = NULL, pCurr = pList; pCurr != NULL; pCurr = pSucc)
{
    pSucc = pCurr->next;
    pCurr->next = pNewlist;
    pNewlist = pCurr;
}
```

Each of the iteration expressions is optional, and an omitted loop test results in an infinite loop requiring another way to exit (e.g., a conditional break or a return). The flexibility provided by this construct can be useful (e.g., with loops over linked lists such as the previous example) or confusing, because there are many ways to write a particular computation, some overly condensed.

### 1.2 Subprograms

**Purpose and definition**

Subprograms were originally devised as a way of reducing the number of lines of code and the compilation time for a program by factoring out multiple occurrences of code segments. They permit us to package and parameterize a computation, code it once, and reuse it in other situations via an invocation. Since a subprogram definition essentially creates a new statement or operator, subprograms provide a mechanism for extending the language to reflect the structure of the problem domain or the design of the application, although there is execution time overhead. That is, a subprogram is an abstraction that permits the programmer to view the structure of the program at a higher level.

The definition of a subprogram describes its inputs and outputs, and the actions performed when it is called. A subprogram definition begins with a header that marks the beginning of that syntactic unit, gives a name for the subprogram, and specifies the names, types, and modes of the parameters. The parameter mode describes the parameter passing mechanism or whether the parameter is an input, an output or both. The header is followed by a body that gives the series of statements executed when the subprogram is invoked. In most languages, a subprogram definition is identified by a keyword, e.g. procedure or function in Pascal. This aids the parser and the reader of the program in identifying the construct, and facilitates searching for subprogram definitions with an editor. For example,

```pascal
{ a Pascal procedure definition }
procedure swap(var left: integer; var right: integer);
var
    temp: integer;
begin
    temp := left;
```

---

2 In addition, the Ada loop statement allows prefixes which specify conditional iteration (while) or counted iteration (for).
left := right;
right := temp
end

In many languages such as Pascal and Ada, a distinction is made between procedures that do not return a value, and functions that do. A procedure call is a statement and a function call is an expression. To support functions, the language defines a statement or operation for indicating the return value that can be used in a function body.

**Parameters, local variables, and scope**

A subprogram can communicate with its caller either through parameters or nonlocal variables. Manipulating nonlocal variables tightly binds the subprogram to those variables, and prevents it from being an independent unit. Using parameters provides more flexibility because we can use the subprogram in a variety of contexts with different inputs and outputs. In this text, we refer to the "formal parameters" in the subprogram definition as parameters, and the corresponding "actual arguments" passed in an invocation as arguments.

In a statically typed language, the subprogram definition gives the type of each parameter. In a strongly typed language, the type of the argument must be the same as the type of the parameter, or must be convertible to it. If not, a compiler error results. Strong typing provides much more reliable code. For example, in an invocation sqrt(7) for which the parameter type of sqrt is real, a Fortran or pre-ANSI C compiler might not perform the necessary conversion, with unpredictable results. Pascal, Ada, ANSI C (almost), and Fortran 90 are strongly typed languages.

Subprograms can define local variables that maintain information needed only within an invocation of that subprogram. Their names are visible only within the subprogram code, and they must exist for the duration of an invocation of the subprogram.

Each subprogram defines a scope in which a new set of names can be declared, and the subprogram's local variables and parameters are declared within that scope only. When those identifiers are used elsewhere in the program, they are bound to other program entities or are undeclared. This allows the programmer to choose local names for a subprogram freely. (We will discuss scope in detail in section 2.)

**Subprogram call and return**

A subprogram defines a parameterized sequence of operations that can be treated as an individual operation from the perspectives of syntax and program design. When a procedure is invoked, its code is executed and then control returns to the statement after the invocation. For a function call, control returns to the point in the expression at which it is called, and its return value is used in continuing the evaluation of that expression. That is, subprogram invocation and return is also a single-entry single-exit control structure. If a subprogram proc1 contains an invocation of another subprogram proc2, then proc2 must complete execution before proc1 can continue. Similarly, if proc2 then calls proc3, proc3 must complete execution before proc2 can continue, and so on a sequence of invocations. Due to the possibility of nested invocations, subprogram invocation is sometimes referred to as a "hierarchical" control structure.

Unlike other control structures, a subprogram defines a lexical context, and an invocation can create objects whose lifetimes are exactly the duration of that invocation, namely the parameters and local variables. When one subprogram invokes another, the invoked subprogram must be completed before the calling subprogram can continue processing. Therefore, the state of the caller must be saved upon invocation, and restored upon return. In addition, the environment of the caller should not be accessible to the subprogram (except for arguments passed by reference). The structure that maintains the state of a subprogram is referred to as the activation record for that invocation. An activation record contains values for the subprogram's arguments and local variables (except in Fortran), and the return address, a program counter value indicating the point in its code at which execution will continue upon reactivation. For implementation purposes, the activation record also includes the dynamic link, a pointer to the caller's activation record (since the sizes of activation records vary), and the contents of the processor registers, which are used for restoring the state of the calling subprogram. The activation record may also include temporary storage needed for expression evaluation.

Since a subprogram can invoke another subprogram before completing, a sequence of these state structures may have to be retained at any one time. Since the subprograms must return in the opposite order

---

3 The subprogram's code is also part of its state but is not stored on the stack.
from that in which they were called, a last-in-first-out stack is used to maintain this information. The run-time stack is a list of activation records, often called stack frames, that contain state information about subprograms that have been entered but not exited, i.e. those that are awaiting completion. For example, if subprogram proc1 calls subprogram proc2 which calls subprogram proc3, we have the run-time stack structure illustrated in figure 1 during proc3’s execution:

![Graphical representation of the run-time stack]

**A typical stack frame**  **The stack after proc1 calls proc2 and proc2 calls proc3**

**Figure 1: The run-time stack**

If subprogram proc3 calls another subprogram, that subprogram’s activation record is pushed on the stack above the record for proc3. When proc3’s execution completes, its activation record is popped from the stack, and execution proceeds at the point given by the return address in proc3’s stack frame. With this scheme, storage for a subprogram’s local variables is only allocated during an activation of that subprogram. Similarly, since a subprogram can only access its own stack frame, it can only access its own variables and nonlocal variables accessed via the static link, but not those of the caller.

**Recursion**

It is often necessary or convenient to define a function recursively, as in the following example:

\[
\text{fibonacci}(n) = \begin{cases} 
1 & \text{if } n = 0 \text{ or } n = 1 \\
\text{fibonacci}(n-1) + \text{fibonacci}(n-2) & \text{otherwise}
\end{cases}
\]

We refer to the nonrecursive case as the base case of the definition. Although the definition appears circular, it is not. The function’s value is always well-defined (for the domain of natural numbers), since we do not define the value for a particular element in terms of the value for that same element. The function is defined in terms of its value for smaller elements of its domain, together with a definition of its value for the smallest elements.

The value of the function for any element can be computed by computing the value for all smaller elements back to the base case. The solution for \text{fibonacci}(n) involves setting up the subproblems of solving \text{fibonacci}(n-1) and \text{fibonacci}(n-2), and then combining those solutions. The program directly mirrors the recursive definition of the function, which requires that the function calls itself:

```c
/* A C function for computing fibonacci(n) */
long fibonacci(long n)
{
    if (n == 0 || n == 1)
        return 1;
```

\(^4\) You are probably aware that this is not an efficient way of performing this computation.
else
    return fibonacci(n-1) + fibonacci(n-2);
}

We define a recursive subprograms by first testing for the base case whose occurrence causes the recursive calls to stop. (We test the base case before the recursive calls to prevent "infinite recursion"). When that test is not satisfied, the subprogram calls itself, usually with a "smaller" argument. When the base case is satisfied, the recursive calls stop, and the succession of deferred computations is completed in reverse order. Each recursive call has its own activation record on the run-time stack, and each activation record contains a different value for the variable used to control the recursion. That is, with recursion, there can be several activation records on the run-time stack corresponding to invocations of the same subprogram. Each of these represents a computation "on hold" waiting for a result from a subprogram that it called. Fortran, Cobol and Basic cannot support recursion because they allocate all variables statically, so there is only one copy of each local variable.

Recursive definitions of subprograms are often more succinct than the corresponding iterative code because no loop variable is necessary to control the repetition. There are many dynamic data structures such as trees and nested lists that are defined recursively. For these structures, it is more natural to define subprograms that operate on them recursively, in the same manner as the definition of the data structure. Recursion is also appropriate for a "divide and conquer" problem that can be solved by reducing it to instances of the same problem for "smaller" inputs, such as fibonacci(n) or quicksort. Recursion is essential for problem solutions that require backtracking such as bin packing, graph search, or artificial intelligence problems.

Parameter passing

When we use pass by value for a parameter, it is a local variable initialized with the value of the argument expression. The argument expression is evaluated before the call, and the resulting value is copied into the subprogram's stack frame. Changes to the parameter within the subprogram only modify its local copy, so pass by value can only be used for input parameters. This is safer than other parameter modes because the subprogram cannot affect the environment of the caller. (However, if a pointer value is passed, the subprogram has access to its referent, and may modify that object.) Parameters are passed by value in Pascal by default, i.e. when the var keyword is not given. All parameters in C are passed by value except for arrays, for which a pointer to the first element is passed. A disadvantage of pass by value occurs when the parameter type represents a large object such as an array or record. Copying the argument to the environment of the subprogram upon invocation is costly in both time and space.

With pass by reference, the invoked subprogram receives a reference to the actual argument object. Essentially, the parameter is an "alias" for the argument variable. When the parameter is encountered upon executing the subprogram, that use refers to the location or value of the associated data object, depending on the context, e.g. on the left or right side of an assignment. For this reason, a subprogram can return an output by modifying a parameter passed by reference. (Usually, constants and expressions cannot appear as an argument corresponding to a parameter passed by reference.) We can also use pass by reference to avoid the overhead of copying an argument that is a large structure. This has the disadvantage that the subprogram cannot be prevented from modifying that object. C++ supports pass by constant reference, which passes the argument by reference but prevents the function from modifying the argument, and permits a constant argument. The compiler can implement pass by reference by allocating a pointer to the argument object in the subprogram's stack frame and automatically dereferencing the pointer in translating the subprogram code. Although this avoids copying the argument value, pass by reference has the additional cost of indirection each time the subprogram uses the parameter.

Fortran uses pass by reference for all parameters, and Pascal indicates pass by reference with the var reserved word. Pass by reference is more convenient than passing a pointer directly because the parameter need not be dereferenced in the subprogram. It also allows passing a local variable, array element, or record field as the argument (in Pascal, a pointer cannot refer to any of these). C does not support pass by reference, so to modify an argument or avoid copying a large object, we use a pointer parameter passed by value, and dereference the pointer within the function when referring to the object. (C allows the programmer to obtain a pointer to any object or component with the & operator.)

One difficulty with parameter passing in Pascal is that the only efficient mechanism for passing large objects is pass by reference, whose intended semantics is returning information by modifying a variable in the caller. That is, it merges a performance issue with a logical issue. In Ada, the programmer specifies the
intended semantics by using the parameter modes in for an input parameter, out for an output parameter, and inout for an input/output parameter, and the translator selects the appropriate mechanism. For example, the compiler can implicitly use a pointer to the argument for an in parameter if it is a large object, and the compiler decides when an object is large enough to use indirection for efficiency. In this way, the appearance of the code better matches the intent of the programmer, and the implementation is left to the translator. The programmer cannot assign a new value to an in parameter since it is regarded as an input to the subprogram. Similarly, the subprogram cannot obtain the value of an out parameter.5

1.3 Exceptions

Execution-time errors

Attempting to perform an operation sometimes results in a state in which correct processing cannot proceed. This can occur because an argument has an invalid value for the operation, because required system resources are unavailable, or because of a hardware error. Execution-time errors include

- hardware-detected errors such as division by zero, arithmetic overflow, memory violations, and device errors
- system errors such as failure of a file operation or a full message queue
- logical errors such as an out of bounds array index or removing an element from an empty queue
- application-specific errors such as an invalid input format

In these cases, we say that an exception has occurred.

Exceptions occur infrequently during execution, but a robust system must be prepared to deal with them. The worst response is to simply continue execution with no notification that the results produced are invalid. The Pascal standard requires that a program terminate upon a subscript error, a subrange error, or an arithmetic error such as division by zero. However, immediate termination complicates implementation of the language, and is not a proper response for a production system that may need to deallocate system resources or restore the state of files. In addition, some indication of the nature of the error should be provided. In many cases, such as application-specific exceptions, it may be possible to limit the effect of the problem and proceed with execution. That is, the program can recover from the error and continue.

Error propagation

In many cases, the subprogram that detects an exception is not the one that has the information necessary to deal with it. For example, if a stack function signals stackUnderflow because the caller popped an empty stack, then it is more appropriate for the user of that object to specify the action to be taken than the designer of the stack module. Similarly, the module that detects an invalid input may not be the same module that communicates with the user. In addition, the subprogram that can detect the error condition might be called by many other subprograms in the system (like the stack pop function), and may have been written and compiled separately.

A direct transfer of control using a goto does not suffice because the subprogram that detects the error has no way of knowing where to transfer control (unless it is only called by one subprogram). In addition, most languages restrict the location of the target of a goto to the same scope or subprogram as the goto statement, so that the same static and dynamic context (i.e., activation record) can be maintained. A goto also cannot pass any information to the handler about the situation that occurred.

If the language supports error detection, the programmer can explicitly check for the error condition, and pass the error notification from one subprogram to another in a sequence of invocations. For example, C file operations and system calls return an error code upon failure. Each invocation of such an operation becomes a conditional statement that tests for a potential error condition, and specifies the action to be taken if an error occurs. To propagate the error notification, each subprogram must pass that information back to its caller, and ultimately to the subprogram that can handle the error condition. However, this tightly binds those subprograms, and is inconvenient if subprograms must detect, propagate or handle more than one exception. It also clutters the logic of all the intervening subprograms, especially since the error information and its propagation is irrelevant to the purpose of those subprograms. In addition, there is no guarantee that the caller of a function that may signal an error will check for the error condition.

5 Ada95 allows accessing an output parameter once a value has been assigned to it.
Exception handling

Error handling is an essential part of programming so it is appropriate that it is reflected by a language construct. The designers of many contemporary programming languages such as ML, Ada, C++ and Java recognized that exceptions are semantically different from loop exits and other control structures, so they included a special mechanism for handling these situations in the language. Languages that support exception handling provide constructs for

- defining exceptions (or exception types)
- signaling the occurrence of an exception
- specifying handlers for different exceptions for a code unit

A subprogram or block can specify an exception handler that will be invoked when a particular error condition occurs in any subprograms it calls, or calls indirectly. It can also specify different handlers for different types of exceptions. The translator must implement mechanisms for transferring control from a signaler to a handler. Unlike the use of error return codes, the programmer does not have to specify error handling separately for each operation, or for each subprogram in a sequence of invocations. In addition, if no subprogram in the sequence of invocations defines a handler for an exception that occurs, the program is terminated. The exception handler is intended to perform any clean-up of the program state necessary before proceeding or terminating the program. This feature allows the programmer to separate application logic from error handling code, making the operation of each clearer.

When an exception is raised, the rest of that block is skipped and any handler the block defines for that exception is invoked. If the subprogram that signaled the exception does not define an applicable handler, that of its caller is invoked. If the caller does not define a handler, the process of locating a handler for the exception continues back through the sequence of pending subprogram invocations on the run-time stack. That is, a block defines an exception handler for itself and for all the subprograms it calls, even those that are invoked indirectly.

We can think of the expression or statement that signals an exception as a non-local goto statement that transfers control to an exception handler, if one is defined. The transfer of control is safe because it returns to an existing environment. Signaling an exception is unlike a subprogram call because control does not return to the subprogram that signaled the exception, and is unlike a return statement because it might not transfer control to the subprogram's caller.

An exception handler must execute in the environment in which it is defined, not the environment of the signaling subprogram. Identifiers in the handler refer to the objects and functions in the scope of the handler's definition, not at the point of the error signal. Therefore, the environment of the subprogram that defines the handler must be restored as the active stack frame before executing the handler. When a subprogram does not define a handler for an exception it raises, its activation record is popped from the run-time stack and its caller is examined for an applicable handler. This process of "unwinding" the stack continues back through the sequence of invocations that lead up to the subprogram that signaled the exception. If no handler is found for the exception, the program terminates.

The language must also specify the behavior of a program if the handler does not terminate execution. Some languages, such as CLU, Ada and C++, only support the termination model, in which execution always continues after the block whose handler was triggered by the exception. If it is necessary to retry the operation after an error has occurred, the programmer encloses that block in a loop that exits upon successful completion of the operation. Other languages, such as PL/I, support the resumption model, in which the handler may directly retry the operation that resulted in an exception after it performs some clean-up activity. This involves restoring the state of the program at that point (after having unwound the stack to the activation record for the handler). In Eiffel, the handler may only retry the operation or terminate execution, rather than continue without performing an operation in the program.

1.4 Concurrency

Purpose

In some cases, a computation can be partitioned into subtasks that can be performed in parallel, or a system consists of a number of independent, cooperating programs that can execute concurrently. For example, a word processor may include a program that presents a user interface and permits editing, and a program that prints a document, and these tasks can be performed concurrently on a multiprocessing
system. Similarly, many windowing systems create a process for each application to respond to events in its window.

Some programming languages such as Simula67, Modula-2 and Ada support features that allow the programmer to specify a concurrent program as a collection of tasks, each of which is a sequential program, i.e. a thread of control. A particular execution of one of these tasks is a process (or thread) scheduled by the operating system on the platform's processor(s). These processes execute asynchronously in the sense that any possible interleaving of program steps can occur, as long as the steps in a particular process are performed in the given order. The processes may be executed concurrently on separate processors, or may be scheduled in an interleaved fashion on the same processor.

**Language support**

A programming language that supports concurrency must define mechanisms for:

- defining processes and specifying the code that they execute
- starting processes, suspending an active process, re-activating a suspended process, and terminating processes
- synchronization among processes to prevent undesirable interleavings of program actions, e.g. so that one process will wait until another performs some action or produces some result
- communication among processes

Some languages permit the programmer to give priority levels for processes, which are used to determine which process to activate when the active process terminates or suspends itself. Some languages also provide an operation that suspends a process for a particular amount of time and then re-activates it.

Programming languages have used two mechanisms for specifying process synchronization. A **semaphore** is an object visible to more than one process that a process can set to indicate that some condition of interest to other processes has occurred. This operation is usually called a **signal**. A process can indicate that it is waiting for the corresponding event by executing a **wait** operation on that semaphore. When a process performs a wait, it proceeds if the semaphore has received a signal, but is suspended if it has not. A process that is waiting for a semaphore to be signaled will be resumed when that signal occurs. Semaphores are often used to implement mutually exclusive use of shared resources. When the resource is created, a signal is sent to the associated semaphore to indicate that the resource is available. When a process wants to use the resource, it executes a wait on this semaphore. After obtaining the resource and using it, the process sends a signal to indicate that the resource is available to other processes.

Another mechanism for synchronization is to group program statements that access a shared variable into **critical regions**. Only one process can execute a critical region associated with a particular shared variable at a time. If a process is executing in a critical region, another process attempting to execute a critical region associated with the same variable must wait until the first process finishes its critical region. That is, executions of the critical regions associated with a variable are mutually exclusive. A similar construct is the **monitor**. A **monitor** defines a shared variable together with a set of operations for that object. The language system ensures that only one process can execute an operation for a particular monitor object at a time.

2. Name structure

2.1 Identifiers

**Naming program entities**

In writing a program, we manipulate various kinds of program entities such as objects, constants, subprograms, and types. In order to refer to an entity in a program, it must be given a name. An **identifier** is a name in the source code used to refer to an object, constant, subprogram, parameter, or type in a program. We say that an identifier is bound to that program entity, and that the entity is the referent of the identifier and is its meaning. We use identifiers to refer both to execution-time entities such as objects and subprograms (i.e., code objects), and to compile-time entities such as data types and symbolic constants.

---

6 Some objects such as objects on the heap and array or record components may only be named indirectly by expressions.
In contemporary functional languages, each name has a unique meaning and cannot be rebound. A name can be declared to stand for an expression, and a parameter name acquires a meaning, i.e. the corresponding argument, during an invocation. The expression can always be substituted for the name and vice versa, without changing the meaning of the expression in which it occurs. However, in most programming languages the correspondence between identifiers and program entities is not one-to-one, and is different in different parts of the program. The same name can refer to several objects, e.g. the local variable of a recursive subprogram. An object can have no name, one name, or more than one name, e.g. when it is passed by reference.

Identifiers have semantic value for the programmer (but not for the translator). Choosing identifier names is a common task, and is often one of the more difficult decisions in programming. To facilitate understanding and maintaining the program, an identifier name should always be suggestive of the meaning of the entity to which it refers. The rules for valid programmer-defined identifiers vary widely among languages. These include the permissible characters, the maximum length of a name, case sensitivity, and so on. For example, a Pascal identifier must be begin with a letter, which can be followed by any number of letters and digits, and case is ignored. Allowing a non-alphabetic "word separator" character, such as - in Cobol or _ in C, is helpful for expressing multiple-word identifiers.

Reserved words

Each language defines a number of names that are part of the syntax of the language, such as control structure keywords, operators, and scope delimiters. In most modern languages, language-defined names are reserved words that the programmer may not use as identifiers. Many languages also define names for primitive types and for particular functions, constants, and objects that a language system must provide. The programmer usually cannot redefine these names.

Binding

The binding of an identifier gives the correspondence between the identifier and the program entity it refers to. A binding maps from the name space of the program to the set of entities manipulated by the program. In general, the process of binding may involve determining other attributes of an identifier as well, such as the type or storage policy used for the referent object, or the correct subprogram definition with overloading. The environment of a statement or subprogram is the set of bindings in effect at that point, and it determines the referent for an identifier use. The environment provides the context in which names acquire meanings. As we will see in this section, the environment is modified by a declaration or by entry into a new level of scope.

The referent associated with an identifier and its attributes can be bound at various times during the translation and execution of a program. A binding is static if the association between an identifier and an entity or its attributes is made before execution, and cannot change during execution, e.g. the type of an identifier in a statically typed language. If a binding is determined at run time or can change during execution, it is dynamic. Examples include the location of a local variable, and the type of the referent of an identifier in a dynamically typed language.

For example, binding a location to an object may occur at various times during the process of translating and executing a program. It is done at compile time for "absolute" addresses explicitly given in the code (e.g., with "memory mapped" device registers), and at load time for static variables (including all variables in Fortran, Basic and Cobol). Binding of addresses for objects is done at execution time for both automatic variables allocated at block or subprogram entry, and for dynamically allocated objects that are assigned storage when the allocation operation is executed. With process swapping in a multiprocessing system, all location bindings may be recomputed during program execution.

2.2 Declarations

Purpose and definition

In Fortran, the programmer can use a variable name in an executable statement without declaring it beforehand. The type of the variable is determined by default according to the first letter of the identifier: names that begin with a letter between I and N are of type INTEGER and others are REAL. This was originally intended as a convenience, but it was found to be a major source of unreliability. If a programmer mistypes a variable name (at any time during development and maintenance), it creates a new
uninitialized variable and the compiler does not catch the error. This situation can be very difficult to debug.

In modern languages, we must declare an identifier before using it so that the translator can associate properties with the name, identify misuses of the identifier, and catch typographic errors. A declaration is a nonexecutable statement that is an instruction to the language translator introducing a name into some scope in a program. We say that the name is visible in that scope. A declaration usually associates various attributes with the name, such as what kind of program entity its referent is (object, subprogram, type, etc.). In statically typed languages, an object declaration also gives a type for the referent of the identifier, and the compiler can verify that subsequent uses of that identifier are valid.

In most languages, all the declarations for a scope (i.e., a block or subprogram) must appear at the beginning of the scope, which provides a dictionary that the programmer can examine to determine the meaning of an identifier. In C++, a declaration can appear at any point, so that the declaration of an entity can be placed nearer to its first use and an object need not be defined before the information necessary to initialize it is available.

Definitions

Most often, the language statement that declares an identifier also specifies that its referent is to be created and defines that object, subprogram, type or constant. A definition is a declaration that also specifies that a program entity is to be created, and binds the name with that entity.\(^7\) For example,

```plaintext
{ example object definitions in Pascal }
var
  count, limit: integer;
  total: real;

/* example object definitions in C */
int count, limit;
double total;
```

The entity created is a code object for a subprogram definition, or a compiler data structure for a constant or type definition.

Forward declarations

There are times when we need to introduce a name to the translator without specifying creation of its referent. Examples include

- when we call a subprogram before defining it
- when we use an object or subprogram in more than one compilation unit, since it must be defined exactly once
- when two definitions are mutually referential, e.g. mutually recursive subprograms or two record types that contain pointers to instances of each other

A declaration that is not a definition is referred to as a forward declaration in Pascal or a specification in Ada. A forward declaration indicates that the programmer wants to use an identifier that is defined elsewhere in the program, either later in the source code or in another compilation unit. Uses of the identifier refer to the program entity specified in that definition. Forward declarations are necessary with information hiding because the interface (i.e., the declaration) of a subprogram or type should be visible to users of the subprogram or type, but its implementation (i.e., its definition) should not be. They are also used with top-down functional decomposition in which a subprogram is written before those it calls.

In languages such as C and Ada that support separately-compiled modules, the association between a declaration and the corresponding definition may be external to the compilation unit, so the linker must create it. An ANSI C function declaration (often called a "prototype") declares the name and argument types of the function to the compiler so that it can determine whether invocations in that module are syntactically correct. For example:

---

\(^7\) Our use of the terms "declaration" and "definition" is that of the C literature. Different terminology is sometimes used with other languages.
/* an ANSI C function declaration gives the function name and argument types */
int min(int arr[], int size); /* min() is defined in another file to be linked with this one */

int main()
{
    const int NUM_SCORES = 5;
    int scores[] = {70, 85, 75, 55, 90};
    int worst;
    /* the compiler uses the function declaration to determine whether the invocation is 
syntactically correct */
    worst = min(scores, NUM_SCORES);
}

The declaration does not define the operation of the function, so the function definition (i.e., the body of the 
function) is expected to be given elsewhere, either later in the program or in a module linked to the resulting 
object code file. If a definition of that function name does not appear in the module or in a module linked 
to it, a linkage error results. The C programmer declares a variable name without defining it by using the 
extern reserved word, as follows:

/* a declaration which is not a definition in C */
extern int arrSize;

Generally, a name may be declared more than once in the same scope (in the same way), but may not
be redefined within a given scope (except when the language supports subprogram name overloading, as
described in section 2.7).

The symbol table

The language translator maintains a data structure called the symbol table that stores the attributes and
bindings of the valid identifiers in the program. Initially, the symbol table contains entries for language-
defined identifiers, such as the names of built-in types. A declaration enters the identifier into this table,
together with information about the meaning of the referent. For example, the entry for an object name
specifies its type and location, e.g., a stack frame offset for a local variable. The form of this information
depends on whether the referent is an object, subprogram, type or constant.

In a compiled language, identifiers exist only during translation (except for symbols external to the
compilation unit), and the symbol table is a compile-time structure (unless it is retained to provide symbolic
debugging). In an interpreted language, it must be present at execution time.

Static and dynamic typing

In general, types may be associated with either identifiers or with the entities they are bound to. In
statically typed languages, the declaration of each object, constant or subprogram name gives a type, and
this information is entered in the symbol table. The type of a subprogram is its argument signature, which
consists of its parameter types and return type. As we will see in section 3.6, the compiler for a strongly
typed language can use this information to ensure that an object identifier only refers to objects of its type,
and that the arguments of a subprogram invocation are of the correct type. Strong static typing provides
safety and reliability because many typographical and logical errors are caught earlier in the process of
coding, and the code is easier to understand. It also informs the compiler how much storage is required for
the referent of an object identifier so that it can use static or automatic allocation. FORTRAN, Pascal, C
and Ada are statically typed languages.

In a dynamically typed language, identifier declarations (if any) do not specify a type. An identifier
may refer to objects of different types during the execution of the program, and there are no invalid
assignments. A variable cannot be said to "be" a real, but only to "refer to" a real at some point during
execution, and the type of the object bound to an identifier can only be determined at run time. That is, we
have dynamic binding of the type of an identifier, and identifiers are polymorphic because they can refer to
objects of different types. In addition, binding of overloaded subprograms (see section 2.6) is also

---

8 A function declaration in C is extern by default.
9 The return type is not considered part of the argument signature in C++.
dynamic, which is essential for object-oriented languages. Dynamically typed languages provide increased flexibility in programming, at the expense of efficiency, and include Lisp, APL and Smalltalk.

With dynamic typing, type information must be stored with each data object to determine whether an operation is meaningful, and to determine which operation to perform with overloading. In addition, it is not possible to associate a fixed size region of storage with an identifier because the size of its referent cannot be determined statically. In fact, the meaning of an assignment is quite different: an assignment binds the name on the left side to a different object (like an implicit pointer assignment), rather than copying the right side value into the storage bound to the identifier. Dynamically typed languages often provide built-in functions that the programmer can use to test the type of an object to ensure that the referent is an instance of the correct type, or to perform different operations for different types of objects. For example, Lisp provides the "predicates" numberp, symbolp, atom, listp and null. We will discuss dynamic typing in more detail in section 5.4.

Alias declarations

Many languages support alias declarations that introduce an alternate name for an existing entity. We can obtain the effect of having two names that refer to the same storage by using a variant record in Pascal or a union in C. C++ includes the reference type, which declares an alternate name for an existing object. For example:

```c++
// A C++ reference
int num;
int& refNum = num; // refNum is an alias for num
```

Any expressions that use the name refNum actually affect num. Such a declaration does not create an object or allocate storage, but enters the new name into the symbol table with the existing object as its referent. Many languages also allow the programmer to define aliases for types, e.g. the C typedef statement.

Generally the use of aliases for objects is questionable practice for several reasons. A hidden side effect can occur: a modification to the object through one identifier changes the referent of another name without mentioning it. Aliasing can break type safety when identifiers with different types alias the same storage as occurs with Pascal variant records and C unions. In addition, aliasing can be confusing for the reader.

2.3 Scope

We often want to use the same name for different purposes in different parts of a program. For example, several subprograms may need a local variable called index. If all names are visible throughout the program (e.g., as in Basic or assembly language), then each subprogram could use a distinguishing prefix, e.g. avgIndex and sortIndex. However, this lengthens identifiers without adding any useful semantic information. Similarily, in a large system composed of code units written by several programmers, we would like each programmer to be able to choose his or her own variable names without being concerned with whether other programmers are using those names.

What we want is the ability to restrict the scope of a declaration so that there is no name conflict with other declarations of the same identifier. In a mathematics text, the author writes "let x be ..." to declare an identifier to the reader, and the scope of the declaration is implied, e.g. the duration of a proof or an example. Unfortunately, compilers do not have the understanding of context necessary for such an approach, so programmers must specify scopes explicitly in the code.

A scope is a lexically delimited region of a program that specifies a name space in which a set of names is visible. The scope of a declaration is the portion of the source code in which occurrences of the declared identifier refer to the associated program entity. Scope allows us to use the same name to refer to different entities in different parts of a program, e.g. in different subprograms. You must be aware of the rules by which identifiers are bound to data objects to write correct programs. An identifier has global scope if it is visible throughout the program. If the scope of an identifier declaration is restricted to some portion of the code, e.g. a local variable of a subprogram, we say that its scope is local to that section of the program.
2.4 Nested scopes

Blocks and nested scopes
As we saw in section 1.1, Algol introduced the compound statement delimited by begin and end to permit nesting the sequence control structure within conditional and iterative structures. Any compound statement can also include declarations, in which case it is a block. Those declarations are visible throughout that block but not elsewhere, like the local variables of a subprogram. Like statements, blocks can be nested, which gives the name space of an Algol program a hierarchical structure.

A block introduces a new scope for its declarations which is nested within the scope of the block in which it is defined. In a block-structured language, we can nest block scopes within other block scopes to an arbitrary level, and the names declared in enclosing blocks are visible in a block if the block does not redeclare them. That is, a block implicitly inherits access to the names declared in enclosing blocks. The body of a subprogram definition is also a statement in Algol, and can be a block that includes declarations, namely of parameter and local variable names. The name of the subprogram is declared in the enclosing scope.

Block structure allows us to organize the name space of the program by restricting the declarations of a set of identifiers to the region of the code in which they are needed. For example, if several functions must access the same data structure, we can declare the structure in a block, and nest the subprograms that manipulate that data within that block. Nested name spaces also help reduce the amount of context the reader must remember. Nested subprogram definitions are supported in Algol and its descendants to support program design by top-down functional decomposition, also called stepwise refinement. A program is broken into main tasks, each of which is coded as a subprogram nested within the main program. If the task handled by a subprogram is large, it contains further subtasks and the corresponding subprograms are nested within it.

The symbol table must also reflect the hierarchical name structure of block-structured scope. Typically, the symbol table has a stack structure, in which the compiler pushes a suitable upon entering a block and pops it upon completing translation of the block. When the compiler searches for the binding of a name used in a statement, it examines the top suitable for the name, then the next suitable, and so on.

All descendants of Algol, including Pascal, Modula-2, C, and Ada, are block-structured languages. However, the generality of Algol's block structure has been restricted in Pascal and C. In Pascal, declarations can only accompany a subprogram, but not an arbitrary compound statement. In C, function definitions cannot be nested, but any block within a function definition can contain local declarations (although C programmers rarely use this feature).

Identifier declarations and uses
The scope of an identifier declaration consists of the block or subprogram in which it is declared, and all contained blocks that do not include a redeclaration of that identifier. The declarations is not visible in enclosing scopes. An identifier declared in a block is local to that block, and is relatively global or nonlocal to all enclosed blocks that do not redeclare it. The declaration is hidden in those enclosed blocks that declare the name, i.e. the entity it refers to is not accessible. All uses of a hidden name in an enclosed scope that redeclares the name (and all scopes nested within that scope that do not redefine the name) refer to the program entity corresponding to the enclosed scope declaration. Although the entity denoted by the outer scope declaration becomes inaccessible when the inner scope is entered, it continues to exist, and when the inner scope is exited, it becomes the referent of that identifier again. Two declarations of the same identifier always have disjoint scopes, and occurrences of that identifier in each scope refer to different entities. These rules apply to the parameter names and local variables of a subprogram as well.

A use of an identifier in an expression or statement (other than its declaration) must be within the scope of some declaration of that identifier. The process of identifier resolution determines which declaration of the identifier a use refers to. A use of an identifier is resolved to the declaration in the scope in which the use appears if there is one, or in the nearest enclosing scope that declares it. If no declaration of that name is encountered in any enclosing scope, the identifier use is an error. The declaration found by the search of enclosing scopes determines the entity denoted by that use of the identifier. In a statically typed language, it also specifies the referent's type, which the compiler uses to determine whether the use of the identifier is valid. The corresponding declaration is found first, and then the type is checked. The programmer cannot use the type demanded by the context in which the identifier occurs to indicate that the use refers to another, more global, declaration of that identifier.
To distinguish two different declarations of the same identifier or to access an identifier that is not visible, some languages name scopes and allow the programmer to use a qualified name that specifies both the scope name and the identifier name. A qualified name identifies the entity to which it is bound uniquely. In some languages, we can use a qualified name to refer to an identifier that is hidden by a declaration in an intervening scope, or to refer to a name declared in a scope that does not enclose the scope of the use. For example, qualified names take the form scope.name in Java and Ada and scope::name in C++.

Lexical scope

A block that is not a subprogram definition can only be entered when control flows directly into it from the previous statement. When this occurs, its declarations become visible, and any declarations in the enclosing scope that are hidden in the block are no longer visible. Every time a statement is executed, the identifiers in it are bound statically according to the hierarchy of scopes defined in the source code.

For subprograms, the situation is more complex because the subprogram name is visible in any scope nested within its enclosing scope, and subprograms can be invoked by name. When a subprogram is called, nonlocal names in the subprogram could be bound either in the environment of the subprogram definition or in the environment of the invocation. We say that languages that use the former use static scope, also called lexical scope, because the structure of the source code determines the scope structure, and the bindings for nonlocal names are fixed for all invocations. Static scope is used in all block-structured languages. If identifiers are bound in the environment of the invocation, the language employs dynamic scope (which is uncommon in contemporary languages).

The Pascal example in figure 2 illustrates the rules of static scope (lines demarcate the scopes in the example):
The interpretation of the diagram is that identifier resolution can see outward from a scope boundary, but not into a scope. Note that the scope boundaries cannot overlap, and two scopes are either disjoint or one encloses the other. Within the code for proc1, the identifiers i, c, x, proc2 and proc1 are visible. The first two are local to proc1, and the last three are relatively global to the block. Any use of i in proc1 refers to its parameter, not to the global variable, which is hidden. The procedure proc2 is defined in the same scope as proc1, but the compiler has not yet seen its definition when it is translating proc1's body. The forward declaration of proc2 allows proc1 to call proc2. (The parameter k in the forward declaration is a place holder and does not actually declare a name.) Within the code for proc2, the identifiers k, y, x, proc2a, i, proc1, and proc2 are visible. Any use of x in proc2 refers to its local variable, and an occurrence of i in proc2 refers to the global program variable. Within the code for proc2a, the same identifiers are visible, but a use of k refers to proc2a's local variable. Note that if proc2a uses the name x in a context in which a real is required, such as an assignment k := x, it is an error because the x in proc2 is a boolean, rather than a reference to the global x defined in nestedScopes. Within the code for nestedScopes, the identifiers x, i, proc1, and proc2 are visible, and that code cannot access any of the local variables or parameters nor invoke the procedure proc2a. The same scope rules illustrated here also apply to the identifiers in any const or type definitions in the program or its subprograms. Note that an invocation of proc1 within the body of proc2a invokes a scope that does not enclose the scope of the call. With static scope, any occurrence of x in the code for proc1 is bound to the global declaration of x, not to the declaration in proc2a.
2.5 Programmer-defined visibility

Problems with use of nonlocal identifiers

With block-structured scope, a subprogram can access any identifier relatively global to it that is not hidden, e.g. a subprogram can modify the value of a nonlocal variable. Any such interaction between the subprogram and its context that does not occur through its parameter list is called a side effect, because it is not visible in the subprogram call. Since the side effect is not explicitly represented in the subprogram invocation, code with side effects is very difficult to understand. Side effects also present a problem for maintenance, e.g. if the programmer needs to determine all uses of the nonlocal variable. Side effects do not occur when the subprogram manipulates nonlocal variables that are passed as arguments.

Direct use of a nonlocal identifier within a subprogram prevents using the subprogram outside the context in which it was originally written, so the subprogram is not a self-contained unit. In another context, a nonlocal identifier of that name might or might not exist, and might or might not have the same type and meaning as in the context in which the subprogram is defined. The subprogram is tightly coupled to the nonlocal identifier, and cannot be used independently of it. Of course, this is not an issue if the subprogram performs a subtask specific to the subprogram in which it is nested.

There are other problems associated with the direct use of nonlocal variables in subprograms. If a nonlocal variable accessed by a subprogram is also passed to it by reference, there are two names for the same object within the subprogram code. The object can be modified either directly or through the parameter, and changes through one name affect the referent of the other. When the programmer makes an identifier nonlocal to provide access by several subprograms, he or she cannot hide it from other subprograms at that level of nesting. If a nonlocal name is accessed in a deeply nested program, it is possible that reorganizing the program or introducing additional identifiers in intervening scopes will cause the use of the nonlocal to refer to a different entity than that intended. There is also a chance that a typing error will accidentally match a name defined in an enclosing scope. All of these situations are problems for reading, debugging and testing the code.

Limitations of hierarchical scope

The hierarchical scope structure rules provide a mechanism for separating name spaces that should be disjoint and for sharing names. However, they are a hindrance when we need to control shared access to individual identifiers and their referents explicitly on a name-by-name or scope-by-scope basis. The only way to share access to an identifier is to declare it global to the scopes in which it must be visible. However, this causes it to be accessible within other scopes at the same level of nesting. For example, the subprograms that search for, insert and delete elements in a table must be nested within the scope in which the table is declared so that they can access it. (Even if the table is passed as a parameter, both the table and the subprogram names must be visible in the scope of a caller.) However, the subprograms that use the table must be defined in the same scope so that they can call those subprograms, and therefore they can access the table data structure directly (which they might do for efficiency). These subprograms might not manipulate the structure consistently with its semantics, and even if they do, modifying the structure of the table might break them.

The problem is more severe in large programs that define several data structures, each intended to be accessible to a particular set of subprograms. For example, suppose table 1 is used by subprograms proc1 and proc2, table2 is used by subprograms proc1, proc3 and proc4, and table3 is used by subprograms proc3, proc5 and proc6. To provide each subprogram with exactly the access it needs via static scope nesting, a nonhierarchical scope structure is necessary, as illustrated in figure 3:
Figure 3: Three tables accessed by six subprograms

With hierarchical scope structure, it is not possible for scopes to overlap in this way. Two scopes either are disjoint or one wholly encloses the other. To implement the access necessary for this example in a block-structured language, we must declare `table2` in a scope visible to both `proc1` and `proc3`, `table1` must be visible to `proc1`, and `table3` must be visible to `proc3`, so all three tables will be visible to all six subprograms.

Exporting, importing, and hiding names

Instead of the implicit hierarchical sharing defined for block-structured scope, we would like a mechanism that allows the designer of a program unit to specify what names are visible within that unit. That unit will define some of these identifiers, and others will be defined in other units and only declared in the unit. The programmer should also be able to specify that some names are not visible outside the unit by dividing the name space into an interface visible to users of the unit, and an implementation which is private to the unit. This allows the designer to decouple the logical use of the unit from the physical implementation of that functionality.

The module construct supports division of a program unit into an interface and implementation. A module consists of a set of subprograms, objects, types, and constants with a single logical purpose. The designer of the module specifies which names the module exports to the rest of the program. A module may import identifiers exported by other modules with a forward declaration. In this way, all name visibility is explicit, and access to an entity through its name occurs only through mutual consent of both the designer exporting the name and the module importing the name. To support this feature, it must be possible to decouple the declaration of a name from the creation of the referent (i.e., to code a forward declaration), so that modules can import the name of an entity defined elsewhere.

This export/import relationship for name sharing involves disjoint scopes, rather than accessing names in a textually enclosing scope. An exported name can be made visible in exactly the modules necessary (rather than in all units at a given level of nesting), and a module can import the interfaces of more than one module (rather than importing access to all the names in a single enclosing scope). If all names are explicitly declared in this fashion, the problems that can occur when using nonlocal names cannot occur. Hidden side effects, unintended access, and name capture all result from implicit access of nonlocal names.

2.6 Overloading

In most languages, operators such as `+` and `<` are overloaded for built-in types, which means that the same operator symbol refers to different operations for different types of operands. For example, the `+` operator can mean either integer or floating-point addition. (It can also denote set union in Pascal and string concatenation in Java or Basic.) The compiler uses the types of the operands in an expression to determine which operation to perform.

Many contemporary programming languages such as Ada and C++ allow the programmer to overload subprogram names. (Operators are just a syntactic variant of functions, although operators for built-in types do not involve the same invocation overhead.) With overloading, there can be more than one subprogram with the same name defined within a particular scope. In this way, we do not have to use different names for the same conceptual operation, e.g. `insertLinkedList` and `insertHashTable` (to say nothing of `insertListOfIntegers`, `insertListOfStrings`, etc.). Ada and C++ also support operator overloading, which allows the programmer to use the standard expression syntax for programmer-defined types. For example, we can code definitions for
• equality and inequality tests for any types
• arithmetic operators for complex numbers
• <, <=, > and >= for strings and dates
• [] (indexed access) for linked lists
Identifiers that refer to data objects cannot be overloaded.

Since more than one definition of the subprogram name or operator is visible in the scope in which the overloads are defined, there must be a way of determining which definition to invoke. Each overloading of a particular subprogram name or operator must have a different argument signature. In translating an invocation of overloaded subprogram or operator name, the compiler chooses a subprogram to invoke by matching the argument signature of the call with those of the subprogram definitions for that name. If there is no definition with the argument signature of the invocation, it is an error. For example, in C++, we can define several print() functions, each for a different type of object:

```c++
/* example of function overloading in C++ */
/* each of the following functions are declared here and are defined elsewhere */
void print(Date);
void print(int arr[], int size);
void print(Employee);

int main()
{
    Date birthday;
    const int NUM_SCORES = 30;
    int scores[NUM_SCORES];
    Graph routes;
    /* ... other operations ... */
    /* the compiler determines which function to call by the argument type(s) */
    print(birthday);  /* print(Date) invoked */
    print(scores, NUM_SCORES);  /* print(int[], int) invoked */
    print(routes);  /* error: print(Graph) not declared */
}
```

Implicit conversions of arguments complicates resolution of overloaded subprogram invocations, and can result in ambiguity. For example, if we have two overloads func(long) and func(double) in C++, the invocation func('c') is ambiguous because a char can be converted to either a long or a double. That invocation (but not the existence of the two overloads) causes a compiler error. The programmer can resolve the ambiguous call by explicitly converting the argument to the desired parameter type, in this case func((long) 'c').

Languages with dynamic typing can also support subprogram and operator overloading. The programmer specifies the parameter types for each definition, and the type information in the argument objects is used to select the subprogram to invoke at run time. That is, overloaded subprogram invocations are dynamically bound. We will see that this is an essential aspect of object-oriented programming languages.

### 3. Data Structure

#### 3.1 Data objects

**Purpose and definition**

A data object maintains a package of information that the application must represent. Generally, we use an object to model an entity or relationship in the problem domain. Examples include the dimensions and weight of a package, the information associated with a transaction, and the description of a design component. We also use data objects to represent components of the computer system or the application itself, such as files, input/output devices, network connections, and user interaction components. The object is an abstraction that represents the relevant properties of an entity, but is not that entity.
We can also distinguish between the abstract "program object" and the "storage object" that implements it. The program object is a collection of values defined by the language, which is accessible via an identifier. The language also defines the valid operations for the program object. At the lowest physical level, an object consists of a region of storage and a collection of values encoded in that region. Typically, the storage allocated to an object is a contiguous collection of memory units (bits, bytes, words, etc.) defined by a starting location and a size. Storage objects can be adjacent in memory, so indexing beyond the region allocated to an object will access another object, while program objects are semantically independent (unless they are components of a composite object such as an array). Languages that support abstract data types provide facilities that allow the programmer to treat a data structure such as a linked list that consists of a discontinuous group of regions as a single program object.

It is also instructive to draw a distinction between "pure values" as used in mathematics, and the objects used in programming [Mac82]. A value such as the number 2 is a timeless abstraction that cannot be modified. For example, functional programming languages deal with expressions whose evaluation results in values, rather than commands that modify storage locations. In contrast, since an object is a data structure that represents a problem domain entity, it can be modified as a program execution that models the problem domain proceeds. That is, an object has a state that can change over time. Unlike values, objects can be created and destroyed, and can be shared so that changes to an object through one reference modify the object state accessible through another reference.

Each data object is an instance of a data type, which may be a built-in or programmer-defined type. The type of an object defines the valid values for that object and the encoding of those values, i.e. the meaning of the stored data. The data type also specifies the set of operations that a program can perform on the object.

**Variables**

A *variable* is an association between a name and a data object that consists of a value stored at some location. (In fact, programmers use the term "variable" to refer to the object, its value, or its location.) Figure 4 diagrams this relationship.

![Figure 4: The components of a variable](image)

Usually, when we use a variable name e.g. if we use it as an operator operand or as the argument of a subprogram call, we are referring to its value. However, to modify an object (e.g., to assign a new value to a variable), we must have a reference to the object, which is the location for a storage object. Whether an identifier use refers to the object's location or its value depends implicitly on the context of its appearance. In an assignment, evaluation of the right-hand side expression must result in a value, and identifiers within that expression refer to their values. The resulting value is copied into the location bound to the identifier on the left-hand side, i.e. the identifier on the left-hand side refers to a location. We refer to the location and value of the data object to which an identifier is bound as the identifier's *l-value* and *r-value*, respectively (the *l* and *r* standing for "left" and "right", reflecting use of the identifier on the left or right side of an assignment.)

There are other contexts in which an identifier or expression can refer to either a reference to an object or the object's value. When we invoke a subprogram, it receives the argument's *r*-value for parameters that are passed by value, and its *l*-value for parameters passed by reference. In addition, three kinds of expressions can be either *l*-values or *r*-values, namely

- an array subscription expression, e.g. `array[index]`
- a record component selection expression, e.g. `record.field`
- a pointer dereference expression, e.g. `pointer^` in Pascal or `*pointer` in C
These expressions are *l*-values when we use them on the left side of an assignment or as arguments corresponding to reference parameters, or *r*-values elsewhere. The reader must take into account the context when considering an identifier or expression (in contrast to functional languages, in which an expression always denotes a value).

**Constants**

We often need to use a particular value in a program, e.g. \( \pi \) or the number of months in the year. We can improve the readability of the program by using a symbolic name rather than the literal value, e.g. `numMonths` is more descriptive than 12. Named constants are also useful when we need to change the value of the constant during development. For example, a constant value that gives the size of several arrays might be used for the array declarations and for the limits of counter-controlled loops over the arrays.

With a named constant, we only need to change the initialization in the constant declaration, rather than finding and changing each use of that value.

The programmer could assign the value to a variable, then use the variable in expressions and statements. However, a constant has different semantics than a variable, because a constant cannot be modified by an assignment or data transfer. If a language supports named constants, the programmer can use the symbolic name and the compiler will ensure that the identifier's *l*-value cannot be used.

To support named constants, Pascal includes *symbolic constants*, which declare a named constant with a value given by a literal. The type of a `const` definition must be a discrete type (i.e., integer, enumeration, character or boolean), for example:

```pascal
{ Pascal symbolic constants }
const
  numScores = 20;
  fileDelimiter = "\n";
```

The type of the literal value gives the type of the constant. The restriction that the value cannot be an expression is inconvenient when the value of one constant depends on that of another. In Java and C++, the value of a symbolic constant can be an expression using previously-defined constants. The compiler evaluates the expression and stores its value in the symbol table. The compiler can often avoid allocating storage for a symbolic constant by using an "immediate" or "direct" addressing mode for operations that use the identifier.

In pre-ANSI C, the programmer uses a *preprocessor macro* to specify a constant, for example:

```c
/* specifying a constant with a preprocessor macro in C */
#define NUM_MONTHS 12
```

The preprocessor substitutes the text that is the "value" of the macro name directly for the constant name before compilation. When the macro value is an expression, this has three disadvantages: the expression must be fully parenthesized, it is evaluated each time the macro is used, and any names in the expression are bound in the environment of the use. In addition, like Pascal symbolic constants, we cannot use a C macro for a constant array or structure object.

*Read-only variables* provide somewhat different semantics for constants, and are supported by Ada and C++ (and have been incorporated into ANSI C). A variable of any type can be declared as a constant, which indicates that it cannot be modified after initialization. Clearly, the constant definition must also specify an initial value, which may be any expression of the correct type. For example (assuming that we have defined a structure `Point` with two members for the \( x \) and \( y \) coordinates),

```c
/* read-only variables in ANSI C */
const int NUM_MONTHS = 12;
const struct Point origin = { 0, 0 };
```

A read-only variable can also be a local variable of a subprogram that is initialized differently each time the subprogram is called, e.g. using argument values. In this case, the value of the object is dynamically bound, and the compiler must allocate space for that object and evaluate the initialization expression for each call.

If the compiler can determine that a particular read-only variable's value is static, it can use immediate addressing modes and avoid allocation, as described for symbolic constants.
In C++ and ANSI C, we can also specify a function parameter as `const`, which indicates that the function does not modify that argument and the caller can pass a constant object. This is particularly useful with C++, which supports pass by reference. We can declare a parameter as a `const` reference to avoid copying the argument and to indicate that the function does not modify the argument, which allows the caller to pass a constant object.

**The lifetime of an object**

We specify creation of a new object by a variable definition or by an explicit storage allocation operation, such as `new` in Pascal. Creation of an object `instantiates` the given data type, and causes unused storage for that object to be located and `allocated` to it. The object's type determines the amount of storage required, and allocation can be managed by the compiler, by a run-time system, or by both. (In the next section, we will discuss storage allocation policies for data objects.) An object is destroyed or `deallocated` when storage is no longer assigned to the object. It is a semantic error to access an object after it has been deallocated. The result of this error is unpredictable because that storage might already be allocated to another object.

We refer to the duration of program execution during which an object exists as its *lifetime*. To use programming languages effectively, we must be clear on the distinction between the lifetime of an object at run time and the scope of an identifier in the source code. Clearly, an object must exist during execution of any code that includes an identifier bound to it. On the other hand, an object may exist when it is inaccessible to the subprogram executing. For example, a local variable name becomes invisible when that subprogram invokes another, or can be hidden when an enclosed scope is entered. With pass by reference, the period of time that the parameter name is associated with the argument object is shorter than the object's existence. Pass by reference also causes an object to be referred to by two different names with different scopes. Dynamic objects can outlive pointers that refer to them. In the next section, we discuss static local variables (e.g., in C), which also illustrate the distinction between scope and lifetime.

**Initialization and assignment**

*Initialization* of an object stores a meaningful value for its type in the region of storage allocated for the object immediately after it is created. Allocation of storage for an object and initialization of the value encoded in that region are separate operations, and are often specified by different language constructs. When storage is allocated for an object, that storage initially contains some arbitrary string of bits, often referred to as "garbage". Any operations performed using this undefined value will return invalid results, which is a frequent cause of programming errors that are difficult to debug. In general, the translator cannot detect this error.

Pascal does not support initialization. A `var` definition specifies the allocation of an object, but a value can only be placed in the object's storage by an assignment or input statement. Many languages such as Fortran, C, Modula-2, and Ada, provide syntax for specifying an initial value for an object when defining it. For example,

```c
/* an object definition with initialization in C */
int num = 1;
```

This statement declares an identifier `num` with type `int`, defines an object (since `extern` is not specified), and initializes that storage with the value 1. C also permits initializing static and automatic arrays and structures by enclosing a list of values in braces. In addition, all static variables in C are initialized to 0 by default. Support for initialization adds reliability to the language because there is less chance of forgetting the initialization, or of using the variable before it is initialized. It also makes it easier to find the initial value of the variable. In C++, all objects must be initialized when they are defined (except instances of C built-in types).

In procedural languages, we can use the same variable to store different values at various times during execution. An *assignment* replaces the current value of a variable by a new value, losing the old value.\(^\text{10}\) The meaning of the variable has now changed, either because the entity it represents has changed, or because the variable represents a different entity. Assignment is a unique operation because it requires an *l* value and modifies an existing object, and is a statement in most languages. In fact, we can view a program

---

\(^{10}\) The distinction between initialization and assignment is important when defining the operations for abstract data types implemented by complex data structures.
in a procedural language as a series of assignments and control statements that direct the flow of execution among them. Contemporary functional languages do not use assignments: When a value is computed, it is passed as an argument rather than stored in a variable.

In some languages such as C, assignment is an expression rather than a statement. This allows cascading assignments (e.g., \( i = j = k = 1 \) in C), and embedding assignments within other expressions. C also includes assignments such as the \( += \) and \( ++ \) operators that require both the \( l \)-value and the \( r \)-value of the target object.

### 3.2 Storage allocation policies

**Memory management**

Modern programming languages provide three storage allocation policies, static, automatic and dynamic allocation. Each policy has its own purpose, and each defines the lifetime of objects allocated using that policy. With static allocation, an object occupies the same location for the duration of program execution. With automatic and dynamic allocation, storage for the object is allocated and deallocated at execution time, so the language must provide memory management. The memory management process assigns storage to objects, attempting to be as efficient as possible, while ensuring that only objects with disjoint lifetimes use the same storage locations. The compiler can implement management of automatic objects, but allocation and deallocation of dynamic objects must be performed by a run-time system. As we will see, managing dynamic memory is a complex task.

The storage allocated to a program execution typically is addressed contiguously, and has a format similar to that diagrammed in figure 5:

![Diagram of storage layout](image)

**Figure 5 A typical storage layout**

Static storage is fixed in size for the duration of program execution. In the layout illustrated in figure 0.6, the stack grows "upward" as functions are called and "downward" as they exit, and the heap grows "downward" as objects are allocated dynamically.

**Static allocation**

With static allocation, an object is allocated when the program is loaded and execution begins, and remains allocated until the program ceases execution. The lifetime of a static object is the entire execution of the program. The object is static in the sense that it always has the same address (or virtual address).

All objects in Fortran, Cobol and Basic are statically allocated, and there are the same number of objects as identifiers. This simplifies compilation because there is a one-to-one correspondence between qualified names and objects, addresses are fixed, and no run-time allocation and deallocation overhead is
necessary. However, it does not use storage efficiently because storage is allocated for objects whose lifetimes have ended, i.e. the local variables of subprograms that are not active. As we stated in section 1.2, static allocation of local variables also does not permit recursion. Main program variables in Pascal and file scope variables in C are statically allocated. This is necessary because the lifetime of those objects is the entire execution of the program.

**Automatic stack allocation**

As we discussed in sections 1.2, storage for the objects bound to the local variables and arguments of a subprogram is allocated in its stack frame. We refer to this as automatic allocation because the storage is allocated and deallocated automatically without explicit specification by the programmer.\(^{11}\) It is also sometimes referred to as **dynamic stack allocation** or **semidynamic allocation** because memory is allocated and deallocated at execution time. The corresponding data objects do not exist until the block is entered or the subprogram is called, and continue to exist until its execution is complete. After exiting the block or subprogram, that physical storage may be used for other purposes. Automatic objects have nested lifetimes: An automatic object that is created after another automatic object is always destroyed before that object.

The compiler translates an access of a local variable as an offset into its stack frame. Stack allocation of local variables is simple to implement (the compiler generates code that adds the total amount of automatic storage for the subprogram to the stack pointer upon entering the subprogram), and is usually supported by the processor hardware. It provides efficient use of the storage available because storage is only allocated for those data objects that need to exist at any point during execution. It also permits recursion because each invocation of a recursive subprogram has its own stack frame containing the values of the local variables for that invocation.

**Dynamic heap allocation**

Contemporary programming languages support dynamic allocation of objects under the direct control of the programmer. This is useful when the need for an object depends on execution-time conditions, or when an object must outlive the subprogram invocation that creates it. By dynamically allocating records that contain pointer fields, we can construct dynamically varying data structures such as linked lists, trees, and graphs.

A language that supports dynamic allocation includes an operation that allocates a new object and sets a pointer argument or return value to refer to it (e.g., the new procedure in Pascal). The type, and therefore the size, of the object is determined by the type of the pointer. The new object does not have a name of its own and can only be referenced through a pointer. The malloc() function in C takes the size of the object as an argument,\(^{12}\) and allocates the indicated amount of storage (rather than an object). It returns a pointer to the new region of storage, which can then be cast to the correct type and assigned to a pointer variable. Since the argument specifies only the size, the programmer can use malloc() to allocate an array dynamically by multiplying the sizeof a type by the number of objects desired. (In Pascal and Ada, we use a pointer to an array type for this purpose.) In Ada and C++, we can give an initialization for the object in the dynamic allocation operation. For example:

```plaintext
-- initializing a dynamically allocated object in Ada
  type Integer_Ptr is access Integer;       -- a type for pointers to dynamic integers
  Int_Ptr: Integer_Ptr := null;             -- a pointer definition and initialization
  Int_Ptr := new Integer'(1);               -- create and initialize a dynamic integer
```

With this capability, the programmer can create an object at any time, not just at subprogram or block entry. The storage requirements of a subprogram cannot be determined at compile time because a data object created during an invocation can persist when that block of code is exited. For these reasons, dynamic allocation requires a storage management facility separate from the run-time stack. We refer to this storage as the heap or the free store, and it is administered by a run-time system included with the language system that cooperates with the operating system, which has the ultimate responsibility for storage allocation.

Initially, the heap is an empty region of storage and objects can be allocated contiguously. However, as objects are deallocated, the heap will contain empty regions of various sizes scattered throughout its area of

\(^{11}\) This specification can be made explicit in C by using the auto storage class.

\(^{12}\) The sizeof operator returns the size of a type or object.
storage, reusable for other allocations. This situation is referred to as fragmentation. To provide efficient use of storage, the run-time memory management system must keep track of these free regions, coalesce adjacent free regions, and implement algorithms for choosing a free region for a new allocation (when possible). (Usually, it is not possible to relocate live objects because pointer variables in the program contain their addresses.) The efficiency of this system can have a large effect on the execution time and space requirements of programs that use that language system.

Management of the free store is a complex task involving many tradeoffs because objects of any size can be allocated or deallocated at any time. Typically, the list of free regions is maintained as a doubly-linked list ordered by address to facilitate coalescing adjacent regions. There are three basic policies for selecting a region from which to allocate a new object. With first fit, the first free region that is large enough is selected, which requires less searching. Often, the system uses a "roving pointer" into the free list so that the "left over" free regions are not concentrated in one part of the free store. Best fit selects the free region closest to the object's size, leaving the least extra space. With worst fit, the largest free region is selected, which leaves behind the largest possible free region, making it easier to use. Most systems have a minimum size allocation to reduce fragmentation (say, enough space for the two pointers and a size field). Another technique is to use separate free lists for each size of allocation necessary, or for commonly needed sizes. This is more efficient because the free regions can be treated equally, and all insertions and deletions in a particular free list can be done by pointer assignments at the front of the linked list with no search overhead.

**Deallocation of dynamic storage**

Deallocation of dynamically allocated objects when they are no longer needed can be handled either explicitly by the programmer or implicitly by the language's run-time storage management system. Many languages use *programmer-controlled deallocation* because the programmer is (presumably) in the best position to know when an object is no longer needed. This policy results in a more efficient executable because it avoids the execution-time overhead necessary for the system to determine when an object can no longer be referenced. The language defines a deallocation operation that takes a pointer argument and returns the storage allocated to the object referenced by the pointer to the heap. Examples include *dispose* in Pascal and the *free()* function in C. Note that there will still be at least one pointer whose value refers to the deallocated region, namely the argument of the deallocation operation.

With explicit deallocation, the programmer must keep track of all dynamic objects and references to those objects. Three types of errors can occur with programmer-controlled deallocation, which are notoriously difficult to recognize and debug. A programmer may forget to release storage before reassigning or destroying the last pointer that refers to it, causing the region to be unusable. This situation is often referred to as a *memory leak* because some storage capacity has effectively "leaked away". If this occurs often enough (e.g., in an input loop), an allocation operation may fail because the free store is empty. Another error is to refer to previously released storage through a *dangling pointer* that still contains the address of the region, e.g. the pointer used in the deallocation operation. This is especially an issue in an application in which there can be several pointers to the same object. At some point later in the program, another object may be allocated in that region, possibly of another type. An operation through the dangling pointer will be performed on the new object's representation using the implementation for the pointer's type. A third error is to release the same storage more than once, e.g. in different subprograms. This will corrupt the data structures that keep track of the free regions in the heap. In all three cases, the effects of the error occur some time after the actual error has happened. In addition, the symptoms exhibited by the program will not be directly related to the cause of the problem, and may not be repeatable.

To avoid these problems, many languages include a run-time facility that reclaims unreferenceable storage without programmer intervention. Languages that support *automatic storage reclamation* include Lisp, Ada, Smalltalk, and Java. The storage management system can deallocate an object when there are no pointers referring to it. One simple technique is *reference counting*, in which each dynamic object contains a field hidden to the programmer that indicates the number of pointers that refer to it. These fields

---

13 In fact, the Pascal language definition states that all pointers that refer to an object should be set to nil when the object is deallocated with *dispose* to prevent dangling pointers. However, no implementations of the language keep track of all pointers to each object so that they can do this.

14 Ada does not actually require language systems to provide automatic storage reclamation, and some do not. If there is none, the programmer must use the "generic procedure" Unchecked_Deallocation to deallocate a dynamic object.
are updated whenever a pointer is initialized, assigned, or destroyed. When the reference count for an object is zero, it is returned to the free store. If the deallocated object also contains pointers, the reference counts for their references are decremented, which may cause deallocation of those objects, and so on.

The disadvantages of reference counting are the additional overhead for pointer operations and the fact that it cannot detect when to deallocate a cyclically linked structure. A technique that can also handle cyclic structures is garbage collection. When very little free storage remains or when the application is inactive, the application is suspended and the garbage collection process marks all dynamically allocated regions that the program can access. It does so by performing a search that traces all linked structures beginning with live pointers in the program, marking each region it encounters. It then examines the entire heap area and returns any unmarked regions to the free store. Finally, execution of the program proceeds. Since suspending the application at an arbitrary time is a problem for interactive or time-critical applications, system designers have developed techniques for "incremental" garbage collection that can be interrupted and restarted. For example, the garbage collection process starts up when the application is inactive (e.g., during data transfer operations or while the user is thinking), and is suspended when the application begins an activity.

3.3 Data types

Purpose and definition

The properties of data objects are described by data types. Each data object is an instance of one data type, and its type defines the set of values that are valid for such objects. Programmers often think of an object's type as specifying only the interpretation of an instance's value since this is all that is stored in the object. In addition, a type definition in Pascal or C only gives the structure of its instances. However, a value is not useful without operations that can be performed with it. For example, the operations for built-in types such as integers and booleans are defined by the language, even though they are not directly associated with instances of those types. To define a type completely, we must specify the valid operations for its instances, as well as the representation of the legal values. That is, the definition of a data type describes what its instances can "do", not just what they "are". For example, a phone number or a social security number might be more accurately modeled as a string than as an integer because a program never performs arithmetic on such objects.

As stated in section 3.1, a data object represents the relevant properties of an entity in the problem domain or a component of the system or application. A data type is a model that describes the common structure and operations of a category of such entities. That is, a data type is not just a set of values and operations, but an abstraction that also has semantic intent. For example, the software for an automatic teller machine will have types that represent problem domain categories such as banks, customers, accounts and transactions, as well as types for system devices such as the keypad, command buttons, and cash dispenser.

Early programming languages supplied a restricted set of types motivated by the hardware or a particular problem domain, and did not support programmer-defined types. However, if the types provided by the language do not mirror the structure and operations of the entities in the program's problem domain, the form of the code does not reflect the problem domain process, and the programmer must make mental associations between data objects and the corresponding entities. For example, Fortran does not support records, so programmers model a rectangle as an array of four numbers representing the x coordinate, the y coordinate, the width, and the height of an instance. Unfortunately, they must remember which subscripts correspond to which values when writing, reading, or maintaining that code.

In modern languages, we can define a new data type to represent a category of entities in the problem domain. The values and operations that we provide for the type should correspond clearly to states and behaviors of the corresponding problem domain entities. The better a data type mirrors the properties of the entities it represents, the easier it is to code, understand, and maintain programs using the type. Like subprograms, types provide a mechanism for extending the language to a dialect that reflects the structure of the problem domain and the application. This concept of computation as simulation of a problem domain process is an essential part of the object-oriented paradigm.

We will see that in object-oriented languages, inheritance results in an object being regarded as an instance of more than one type.
There are a number of alternate views of what a data type is. For the compiler, types impose syntactic constraints on expressions, allow it to associate semantic features with expressions and statements, and indicate the amount of storage to allocate for an object. From an implementation point of view, a type specifies a mapping between a value for the type and an encoding of an instance in a region of storage, and vice versa. The type also defines how to manipulate those bit strings to implement the type's operations. Formally, a type can be considered a set consisting of the objects that are its instances, or a predicate which is true for those values. In this sense, a type defines a domain for the functions that operate on instances of the type. Similarly, types partition the set of possible values into classes with common attributes and operations. We can also think of a type as defining a finite state machine whose states are defined by the valid states of an instance, and whose transitions are defined by the type's operations.

**Types and instances**

It is important that you understand the difference between a type and a data object that is an instance of that type. In compiled procedural languages such as Pascal and C, a type is a compile-time construct that does not create an object, and instances of the type are created and manipulated during execution. In Pascal, there is a strict syntactic distinction between the definition of a subprogram's types and its variables. The former are introduced by the keyword type and the latter follow the keyword var. Type definitions must precede the variable definitions so that the types are available for creating instances. For example,

```pascal
{ a record type and three instances in Pascal }

type
  Point =
    record
      x: integer;
      y: integer
    end

var
  pt1, pt2, pt3: Point;
```

In C, the distinction is less clear in the structure of the code, because we can use the same statement to define an array, enumeration, structure or union type, and also to create instances of that type. For example, the following statement creates a type struct Point, and allocates three instances pt1, pt2, and pt3:

```c
/* a struct type and instances in C */
struct Point /* data type "struct Point" */
{
  int x;
  int y;
} pt1, pt2, pt3; /* three instances */
```

In addition, C allows the programmer to define composite objects without explicitly defining and naming a type for them, for example, if the previous statement did not include the identifier Point.

**Primitive and built-in types**

*Primitive types* describe atomic values that have no accessible substructure, and often mirror types that are built into the instruction set of the hardware. Some typical examples include boolean, character, and various numeric types such as integer, fixed point, and floating point. Every language specifies a set of *built-in types* that every implementation must provide, which are primitive types or types motivated by the intended problem domain or type of application. For each built-in type, the language must provide constructs for creating instances, specifying values (e.g., by a literal notation), inputting and outputting values, and performing basic operations on such objects. Operations on these types are often indicated by infix operators such as > and +, or by reserved words such as and or mod. The programmer can define additional operations for a primitive type by using it as the type of a subprogram parameter.

The built-in types of Pascal are integer, real, char, and boolean. Ada elaborates on these, especially in its treatment of numeric types. The primitive types for ANSI C are char, short, int, long, float, double, and long double, and any of the integral types (including char) may be specified as signed or unsigned. The C built-in type void indicates an unknown type of value, and is mainly used as the return type of a
function that does not return a value (or as the parameter type for a function that does not take arguments in ANSI C).

**Numeric types**

Programming languages have used three kinds of numeric types: integers, floating point numbers, and binary coded decimal numbers,\(^{16}\) and each is typically supported by the hardware. Beginning with FORTRAN, languages have overloaded arithmetic operators for all numeric types, and most have provided implicit conversions from less general types to more general types in mixed-mode expressions.

Typically, positive integers are encoded as binary integers, and negative integers are encoded using the two's complement representation because it permits using the same arithmetic operations for positive and negative values. The number of bytes the hardware uses to encode an integer affects the range of numbers that can be represented, so an integer type cannot model the corresponding mathematical abstraction exactly. The hardware signals an error if an operation causes an overflow, but most higher-level languages do not provide a construct for detecting and responding to this error. Many machines provide instructions that implement operations on more than one size of integers, e.g. on two, four and eight byte representations.

Fortran, Algol, and Pascal support a single integer type, which is mapped onto whatever representation the target platform implements. C provides the types short, int and long, and a language system can assign them to different sizes if they are available on the target platform. C also allows the programmer to declare an integer variable as unsigned, which allows values with twice as much magnitude for a given storage size. Ada also includes the types Short_Integer, Integer and Long_Integer.

In C, an instance of any of the integral types char, short, int and long can also be treated as a bit string. The language provides the bitwise logical operators & | ^ and ~, and the shift operators >> and << for instances of these types. That is, the values of these types are not regarded as atomic. These operations provide additional flexibility which is necessary for system programming, but break the integrity of the integral types. Many implementations of Pascal provide similar functionality because it is useful for these applications.

Computer designers have devised a number of ways of encoding real values in bit strings, each supporting a different range and precision. The IEEE has developed two standard formats for representation of reals as normalized "floating point" values, which are used in most contemporary computers. The encoding uses bit fields for the sign, the mantissa (the leading 1 is not stored explicitly), and the exponent (offset by 127 or 1023). Two sizes of floating point numbers are defined because many machines support two representations, and are referred to as single and double precision. To take advantage of multiple representations, Fortran includes the types REAL and DOUBLE PRECISION, C supports float and double, and ANSI C adds long double. Algol and Pascal only provide the type real since the designers of those languages considered the distinction between single and double precision to be a machine dependency.

The numeric types in programming languages are an approximation of the mathematical values that they represent. In practice, only a finite range of integer values can be represented, depending on the number of bytes used to represent an integer on the machine. Real numbers can only be represented to a finite precision, and real arithmetic is approximate. For these reasons, the specifics of the implementation of numeric values can affect the accuracy and even the correctness of the results obtained. These details are not accounted for in the definition of most programming languages because they depend on the hardware. This relieves the programmer from having to describe these characteristics, but does not permit specifying the size or precision of values to ensure consistent results when porting a program to another system.

**Logical values**

Due to its importance in reflecting the problem domain and program logic, most languages provide a built-in type for the logical values "true" and "false". For example, ALGOL and Pascal supply the type boolean, and Fortran 77 provides LOGICAL. Comparisons and other tests return instances of this type, and its values are used to control conditional evaluation and iteration. We can use the logical operations and, or, and not on its values. Only a bit is required to represent a logical value, but since most machines are not bit-addressable, a logical value is represented by the smallest addressable unit, typically a byte.

\(^{16}\) Binary coded decimal numbers were implemented by early business-oriented machines and languages, but are rarely supported directly by modern languages so we will not discuss them.
There is no boolean type in C. C programmers use an int for logical values, with the value 0 denoting "false" and nonzero values denoting "true" in conditionals such as if or while statements.

Text data
Hardware vendors have used several different encodings of character data. Currently, most machines use the ASCII code, which defines the values 0 through 127 for printable characters and various "control codes". These include codes for controlling output devices such as TAB and FORMFEED, and for byte-oriented communications protocols such as ACKnowledge. Most languages use the characters' ASCII values when comparing characters for ordering. For example, Pascal defines a built-in type char with the conversions ord, which returns the character's ASCII value, and chr, which creates a char with a given ASCII value. Characters can be tested for equality and for ASCII ordering. C permits implicit conversions between a character and an integer that gives its ASCII value.

Strings represent natural language text, and are different from primitive types because they are not atomic and are not supported by the hardware. They also differ from other composite types like arrays and records because they are inherently of variable length, and operations can involve instances with different lengths. Most languages support fixed-length arrays of characters only, because of the storage management overhead necessary for variable-length strings. In Pascal, strings are represented by arrays of characters, and there are no predefined string operations, so they must be coded. Strings of different lengths are instances of different types, so the programmer usually "pads" all string instances to the same size with blank characters so that there is one type for string variables and parameters. In Ada, the type String is predefined as an array of characters whose length is fixed when the object is created, as follows:17

```
-- the Ada String type, a fixed-size array with dynamically bound size
type String is array(Positive range <=) of Character;
```

Ada supports string literals enclosed in double quotes, initialization, assignment, comparison for equality or ordering, concatenation (using the infix & operator), and substring operations. For example, Str(3:6) denotes the string consisting of the third through sixth characters of Str.

To support the variable-length nature of strings implicitly in a built-in string type, a language must allocate and deallocate strings dynamically and implicitly, since the size of an instance cannot be known at compile time. Some languages such as Basic and Snobol handle allocation automatically using a "string pool" managed by a run-time system that employs reference counting or garbage collection. Support for variable-length strings provides convenience and writability for the programmer, at the expense of additional compiler complexity and the execution time cost of managing allocation. Two common representations for variable-length strings are the counted string, which gives the length in the first byte, and the terminated string, in which a delimiter character marks the end of the string. Both representations allow efficient processing loops, but use of a delimiter has poorer error behavior, i.e. when the delimiter is missing, and prohibits using the delimiter as a character.

C provides some support for variable-length strings. In C, strings are null-terminated character arrays, and the language supports string literals enclosed in double-quotes. The programmer can allocate strings statically, automatically or dynamically. Since arrays are passed to functions by passing a pointer to the first element, the programmer can use character pointer parameters to define functions that operate on strings, irrespective of their lengths. The standard library string.h defines numerous string functions. However, the programmer must explicitly manage allocation of dynamic strings, and must be aware of the size of static and automatic strings. This approach matches C's design goals of efficiency and flexibility rather than convenience and safety.

3.4 Defining types

Development of the type construct
In first generation programming languages, aggregate objects such as arrays (Fortran) and records (Cobol) could be defined, but the concept of a structured type as a separate construct had not been developed. For example, a Fortran programmer can create an array object, but the language does not

---

17 As we will see in section 3.5, the <> notation defines an "unconstrained array type" whose instances can be different sizes.
include a construct to define the type "array of 100 reals, indexed from 1 to 100". Similarly, a Cobol programmer can define a record variable, but each record object is specified separately and is not related to other objects with the same structure.

By the late 1960s, language designers recognized that a type is an abstraction independent of its instances, and that a language should include constructs for defining types and using them to create objects, specify parameters, and define further types. The designers of Algol68 developed a comprehensive set of type constructors and selectors and clearly defined type compatibility rules. The programmer could define types based on both built-in types and programmer-defined types, e.g. to define a type for an array of records. Most of these constructs were incorporated into Pascal and clarified further in Ada. Ada also extended the notion of types to include generic packages in which a type can be a parameter to a type definition. Object-oriented programming languages extend the concept of type further by providing inheritance among type definitions, and programmer access to type objects and operations.

First-class objects
Throughout the development of programming languages, the kinds of objects that programs can refer to and process have increased. We say that a type is first-class in a language, and that its instances are first-class objects if the language provides complete support for manipulating that type of object. To illustrate this concept, let us consider the kinds of capabilities that are usually available for built-in types. We can perform initializations, assignments and equality comparisons. We can use the type as a parameter or return type to provide additional operations for its instances, or as the base type of another type, e.g. as the type of a record field. The language defines a literal notation for writing values of the type, and provides input and output operations, and ordering comparisons and infix arithmetic operators, if appropriate. As we discuss programmer-defined types in section 3.5, we will consider to what extent each kind of type is first-class in various languages. This will help us understand what is necessary to give a complete definition of a class in an object-oriented language.

Early programming languages do not provide the same status for arrays as for built-in types, e.g. they do not support array assignments, equality tests and return values. In programming languages such as Pascal and C, support for treating aggregate objects such as arrays and records as objects in their own right is much better, but is still not complete. In languages that support abstract data types such as CLU and Ada, we can manipulate complex data structures coherently in the same fashion as data objects. An important design goal of object-oriented languages is that programmer-defined types be first-class. Many languages provide some support for treating functions as objects, e.g. a function can be a parameter of another function or the referent of a variable. In functional languages, this support is complete, and function objects can be created during execution and manipulated. Some languages such as Smalltalk and Java also support treating data types as first-class objects. A program can query the type of any object, and use the result as the value of a variable or an argument to a subprogram, and the language (actually, the class library) provides many operations for type objects.

Language support and type constructors
To support programmer-defined types, a language must provide constructs for naming and describing a new type, usually based on existing types. We can define the structure of a new type by

- listing its values, e.g. an enumeration type
- restricting an existing type, e.g. a subrange type
- describing the structure of composite objects in terms of the number and types of their components, e.g. an array or record type

We can then use the type to create instances, declare parameters, and define other types. The language must also provide constructs for writing and initializing instances of the type, and for defining the type's operations, e.g. for selecting the components of a composite object.

Pascal was the first widely used language to include type constructors for defining new types. The type declaration statement describes the structure of a pointer, enumeration, subrange, array, record, or set type. Any of these may be based on existing programmer-defined types, so the programmer can create a type for an array of records that contains a set and an enumeration, and so on. However, these types are not first-class in Pascal. For example, we can only write instances of enumerations and subranges as literals, and functions cannot return arrays or records.

The Pascal programmer does not give the type's operations as part of the type definition, but defines them separately as subprograms that take a parameter of that type. Languages that support abstract data
types such as Ada and object-oriented languages provide a construct for grouping a type's operations together with the definition of the structure of its instances.

**Type definitions and the type object**

A type definition gives the name of the type, the kind of type (e.g., array, record, etc.), and the structure of its instances. If the language supports abstract data types, the type definition also specifies the type's operations. Unlike an object definition, a type definition does not create any data objects or allocate storage during execution (in compiled languages). Instead, the translator creates a symbol table entry associated with the type name. This type object maintains information about the type such as:

- the type's name
- the kind of type
- the structure of the type's instances
- the size of an instance of the type
- the operations that are valid for the type

The symbol table initially contains such entries for the built-in types of the language. The compiler uses this information to allocate instances and check the validity of operations using instances of the type. The type object is usually not present at run time in compiled languages, but must exist at execution time in an interpreted language.

The information present in a type object depends on the kind of type it represents. It specifies:

- the base type for a pointer type
- the constant names for an enumeration type
- the limits and base type for a subrange type
- the number of dimensions, index type or range, and element type for an array type
- the components names and types for a record or union type
- the argument signature for a subprogram type

In object-oriented languages, the information in the type object is more extensive (e.g., it includes the type's operations and class hierarchy relationships), and some of it must present at execution time to support dynamic binding of methods.

**Type declarations**

Like objects and subprograms, types can be declared without being defined. Such a declaration informs the compiler that the identifier is a type name, but does not define the structure of the type. (Of course, a type definition must be given elsewhere in the program.) For example, a forward type declaration is necessary in Pascal when two types contain pointers to instances of each other. We can use a type that is declared but not defined as the base type of a pointer since all pointers are represented in the same way, or as the type of a reference parameter. However, we cannot define instances of the type since the compiler does not know how much space to allocate for that object.

**Type operations**

We sometimes want to code a general-purpose algorithm that varies only in the type of a parameter, e.g. a subprogram that sorts the elements of an array of any orderable type. However, in a statically typed language, the subprogram that sorts an array of integers is different from the one that sorts an array of reals. If the language supports overloading, both functions can have the same name, but we must still code each separately, even though the code is the same except for the type name. Some languages provide a construct for defining a generic subprogram which includes a type parameter that is bound according to the type of an argument in an invocation. The programmer can also define a generic type, e.g. to define a list type with a type parameter for the type of an element. This capability is provided by the Ada generic package and the C++ template facility.

If a type can be a parameter, details of the code may depend on attributes of the actual type used. For example, a sort program needs the subscript bounds of the array type actually passed to the subprogram. It would be useful for the language to provide constructs for obtaining this information, i.e. for making information in the type object available to the programmer. Similarly, a robust program may wish to check

---

18 In practice, the translator may store the information the type object represents in a number of different tables.
the range of values representable by particular implementation for a built-in numeric type. If we think of
the type as an object, these operations are type operations on the type object.

C defines just one type operation, sizeof, which returns the number of bytes required for an instance of
the type. In addition, the standard library "header file" limits.h defines constants that give the lower and
upper limits for an implementation's representation of the built-in numeric types. Pascal also defines
constants such as maxint for this purpose. Ada provides access to a number of type attributes, which we
refer to by the type name, a single quote, and the attribute name. Examples include

- the size in bits of an instance of the type, e.g. Point'Size
- the smallest and largest values for a numeric or subrange type, e.g. Integer'Last
- the number of elements of an array type, e.g. Scores'Length
- the resolution of a fixed point type, e.g. Dollars'Delta

Note that all these values are constants that the compiler can substitute for a use.

In object-oriented languages, a type is referred to as a class. In many languages, a class is an object
that is present at execution time. For example, in Smalltalk, the type object maintains

- the operations of the class
- the "methods" that define the class's operations
- the superclass and subclasses of the class
- "class variables" that maintain information relevant to the class but not its instances
- a list of the class's instances

The interface of the class object includes operations to access this information. We can ask any object for
its class at execution time, and can also use the resulting type object as the referent of a variable. That is,
type objects are first-class and type operations are coded in the same manner as operations on data objects.
Creating an instance of a type is performed by invoking an operation of the type object. We will discuss the
Java reflection API which provides operations dealing with type identity.

3.5 Programmer-defined types

Type aliases

In Pascal, we can define a type alias for a built-in or programmer-defined type to indicate semantic
intent for instances of the type. The new type has the same representation and operations as the existing
type. For example,

\{ a type alias in Pascal \}

```
  type
      Length = real;
```

We can obtain the same effect in C with the typedef statement. However, these languages make no
distinction between the existing type and the new type, and the types' operations can be performed on
instances of the new type and the existing type together. For example, we can pass instances of real to
subprograms that take a Length and vice versa, and can multiply a real and a Length or compare them for
ordering.

Pointers

In most contemporary languages, we can declare a variable of type pointer or reference to a particular
type of data object. (Ada uses the reserved word access.) Its value specifies a reference to another object
elsewhere in storage, i.e. its value is an l-value. Its value is not the object itself, nor does the pointer contain
the object. Defining a pointer variable does not create or initialize the object it refers to. The language
supplies a special value to indicate that the pointer does not point to any object, e.g. nil in Pascal, NULL or
0 in C, and null in Ada. All pointers in Ada are initialized to null by default for safety. In most languages,
pointers can only refer to dynamically allocated objects. (C is an exception.)

The first higher-level language to support pointers was PL/I. However, a particular pointer could refer
to any type of object, so it was not possible for the compiler to check the validity of operations using the
referent of a pointer. In all succeeding statically typed languages, a pointer type specifies a base type, and
instances of the pointer type can only refer objects of the base type. The type object for a pointer type includes the base type to ensure that uses of the pointer's referent are valid, and to indicate how much storage is needed when allocating a dynamic object. A pointer value is usually implemented as the address of its referent object. The storage required for a pointer object depends on the size of the address space, but not on the base type of the pointer type.

Several operations for pointers are typically supported. The dereference or indirect operation accesses the object the pointer refers to. It is denoted by the postfix * operator in Pascal and the prefix * operator in C, and the type of that expression is the pointer's base type. We can use the result of this operation, i.e. ptr^ in Pascal or "ptr" in C, as an r-value to obtain the value of the indirectly referenced object, or as an l-value, e.g. on the left side of an assignment that modifies that object. Attempting to dereference a null pointer is a run-time error that typically causes program termination. We may also consider the dynamic allocation and deallocation operations described in section 3.2 as pointer operations.

A pointer variable can be assigned the value of another pointer variable of the same type, and pointers can be compared for equality, i.e. to determine whether they point to the same object.

You must be clear on the difference between pointer assignment and object assignment. A pointer assignment causes both pointers to refer to the object referred to by the right-side pointer, while an object assignment involves copying the object referred to by the right-side pointer into the region referred to by the left-side pointer. For example, in C,

```c
/* object and pointer assignment in C */
int main()
{
    int* plnt1;
    int* plnt2;
    /* other operations ... */
    /* object assignment: plnt1's referent is assigned the value of plnt2's referent */
    *plnt1 = *plnt2;
    /* other operations ... */
    /* pointer assignment: plnt1 now refers to plnt2's referent */
    plnt1 = plnt2;
}
```

Special care is necessary after a pointer assignment because both names now refer to the same object. If we change that object's value through one pointer, the value referred to by the other pointer changes, even though the assignment statement does not mention the latter. If we deallocate the object through one pointer, both pointers become dangling pointers. In addition, if the former referent of the reassigned pointer is no longer accessible, it should be deallocated before the assignment so that a memory leak does not occur. The same distinction occurs with respect to equality tests. An equality test between pointers, i.e. pointer equivalence, tests whether they point to the same object. An equality test between dereferenced pointers tests whether they refer to equal objects, possibly stored in different regions.

C treats pointers as addresses in a linearly numbered storage unit, as in assembly language. In addition to referring to dynamic objects, a pointer may refer to an automatically allocated object, an element of an array, or a component of a structure. The prefix & operator returns the address of the object referred to by an l-value, which may then be assigned to a pointer variable. Care is necessary when using a pointer to an automatic object because the pointer becomes a dangling pointer when the object's scope is exited. The type void* is used to declare a "generic pointer". A void* pointer cannot be dereferenced directly, but it may be cast to any pointer type, and it is the return type of the malloc() allocation function.

C also supports arithmetic operations on pointers. An integer num may be added to or subtracted from a pointer that refers to an array element, and the result is a pointer to the element num positions after or before the element originally referenced. Pointer arithmetic is performed in units of the size of the array

---

19 Pointer dereferencing is implicit in Ada, Fortran 90 and many object-oriented languages.
20 The & operator is not used with the name of an array, and is optional when assigning a function to a pointer function.
21 For this reason, Pascal does not permit the programmer to obtain a pointer to a local variable. Algol68 avoids this problem by requiring that the scope of the referent of a pointer be at least as large as the scope of the pointer variable.
elements, whose type is the base type of the pointer. The programmer can also use the increment and decrement operators ++ and -- with pointers, and the pointer is incremented or decremented by the element size. Two pointers that point to the same array can be subtracted, and the result is the number of elements between their referents. The language also supports ordering comparisons for pointers that point to elements of the same array.

Unfortunately, C makes no distinction between a pointer to an object and a pointer to an array of objects. Therefore, pointer arithmetic can be used with any pointer (although it is meaningless for pointers to non-array objects), and the programmer cannot restrict a function parameter to being a pointer to an object or a pointer to an array only. In addition, no range checking is performed for any of the operations that modify pointer values. The use of pointer arithmetic in C is very unsafe, and errors are notoriously difficult to debug.

**Enumerated types**

We often need to represent an entity or attribute for which there is a fixed set of values, such as the days of the week or the colleges in a university. Similarly, many entities can be modeled by a state machine, in which case the corresponding object is in one of a fixed set of states. In Fortran or Algol, the programmer uses numeric constants for each value, and then uses an integer variable to model the object. For example:

```
C USING INTEGERS TO ENCODE A SET OF VALUES IN FORTRAN
C 5 MEANS FRIDAY
PAYDAY = 5
```

This technique is error-prone because the programmer must remember the codes, so a named variable might be used for each value. For example, the programmer could define `FRIDAY = 5`, and then use the assignment `PAYDAY = FRIDAY`. Unfortunately, the compiler cannot verify that a meaningful value is assigned to the variable, to prevent assignments such as `PAYDAY = -100` and `PAYDAY = ENGINEERING`. Integer operations are also permitted on the variable, which is meaningless. This technique does not retain the semantic intent of the type.

To avoid this lack of security, Pascal provides the enumerated type. An enumerated type allows the programmer to specify a set of mnemonic names for the values possible for an instance of the type. The use of an enumerated type is an improvement over the use of integer variables because it enhances the readability of the code, and allows type checking that prevents meaningless operations. In addition, the compiler can use a condensed representation whose size is determined by the number of values. We define an enumerated type by listing the value names, as in the following examples:

```pascal
{ an enumerated type in Pascal }
type
  Day = (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);

/* an enumerated type in C */
enum Day { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
```

The value names are treated as symbolic constants, and must be unique within the scope in which the enumeration is defined (in Pascal and C). These names can be used as literal values for an instance of the type, e.g. in an assignment or equality comparison. The value names are typically mapped onto integers internally. The type object for an enumerated type specifies the size of an instance, the value names, and their encoding.

In Pascal, each enumerated type is a separate type that may be used for variable declarations and parameter types, and its instances are first-class objects. Assignments and equality tests are supported for instances of enumerated types. The constants are considered to be ordered according their order in the type definition, and the operations `pred, succ, ord` (conversion to integer), and the relational operators are supported for these values. We can use an enumerated type as the type of an array index, and can use an instance as the control variable in a for loop that iterates over the array. Input and output operations are not supported for enumerated types, and must be coded by the programmer.22

---

22 To support input/output operations, the value names would have to be present at execution time.
Ada extends the Pascal enumerated type by allowing two enumerated types in the same scope to use the same constant name, i.e. enumeration values can be overloaded. For example:

```pascal
-- overloaded enumeration values in Ada
type Screen_Color is (Red, Green, Blue);
type Traffic_Light is (Red, Yellow, Green);
```

If the value Red or Green is assigned to a variable, the type of the variable determines which value is used. If the usage is ambiguous, e.g. as an argument to a subprogram which is overloaded for both types, the type of the value must be specified by a cast, e.g. `Screen_Color(Red).

In ANSI C, each enumeration is considered a distinct type that has integer values, the value identifiers are regarded as int constants, and integer values can be given for the constants. However, any integer value can be assigned to a variable of an `enum` type, and the enumeration constants can be used as integers in any context, e.g. they can be multiplied.

**Constrained types**

In many languages, the programmer can define a new data type as a subrange of an existing type to represent a kind of entity whose values are restricted to a subrange of the values of the existing type. The subrange type definition gives the name of the new type, the base type, and the limits for the values for an instance. In Pascal, the base type must be a discrete ordered type, i.e. integer, character, an enumerated type, or another subrange type. The following are example Pascal subrange type definitions (assuming that the enumerated type `Day` is defined as in the previous subsection):

```pascal
{ subrange types in Pascal }
type
  Year = 0 .. 2200;
  Weekday = Monday .. Friday;
  Uppercase = 'A' .. 'Z';
```

The base type of the subrange type is implied by the type of the limits given in the type definition, which must be literals. The use of a subrange type permits the compiler to check whether a constant value being assigned to a variable of that type is within the valid range. The Pascal standard also requires execution-time checking of assignments. Subrange types provide safety and reliability because it is much easier to debug a subrange violation than the erroneous behavior that results some time later if that error is not caught. Since the subrange type has fewer valid values than the base type, the compiler can use a more compact representation to save space. The type object for a subrange type specifies the size, the base type, and the limits. Pascal programmers often use subrange types to specify the index type of an array.

A subrange type is compatible with its base type in the sense that a subrange instance can be assigned to a base type variable or passed as the argument corresponding to a base type parameter. The operations of the subrange type are those of the base type, and the base type's literal notation is used for specifying values. However, the base type is not compatible with the subrange type since the actual value of a base type object at execution time may be out of range (like integer which is compatible with real but not vice versa in Pascal).

**Arrays**

The need to manage and process collections of like objects is common in programming. An array is a fixed-size, homogenous, indexable sequence of elements that is treated as an composite object. Almost every language permits definition of array objects or types. Formally, an array specifies a mapping from a contiguous range of integers (or of other types in Pascal or Ada) to a set of elements of the same type, called the base type. The definition of an array object or type gives the name, the base type, and the index type or the subscript limits (often called the subscript "bounds"). Informally, programmers usually consider an array to be a sequence of elements of the same type, which is the usual storage layout.

- There are a number of characteristics of arrays that a language supporting arrays must address. In some languages, array indices must be integers, and the language may specify a fixed lower limit, e.g. 1 in Fortran or 0 in C. In other languages such as Pascal and Ada, the index type may be any discrete, ordered, finite type. Fortran only permits statically allocated arrays, Algol allows automatic allocation, and C and Ada support dynamic allocation on the heap. With automatic allocation, the limit expressions may be evaluated
at compile time (as in Pascal) or at execution time (as in Ada). The language must support accessing and modifying elements, and may provide syntax for initialization and for array literals. More recent languages provide better support for coherent operations such as assignment, equality tests, and mapped operations so array objects are first-class. The type object for an array type specifies the base type and the index type, or the upper and lower limits if the index type must be integer. It may also include the size of the array and the base type, for convenience.

The primary operation for arrays is subscription, which takes an array name and a subscript or index value as arguments, and returns the referenced element. We enclose the index in square brackets [ ] following the array name in Algol, Pascal, and C. (Fortran and Ada use parentheses.) The type of a subscription expression is the base type of the array. A subscription expression can be used as an r-value or as an l-value, and we use it on the left side of an assignment to store an element in the array. That is, subscription is a shorthand for the operations retrieve(Array, Index): Element and store(Array, Index, Element). Since each element is individually accessible, an array is a “random access” structure.

Subscription is undefined for an index argument outside the subscript bounds for that array object or type, and the language should address the result of this error. A subscript range error causes a run-time error in Pascal, but is undefined in C and usually accesses an unintended region of storage. These options reflect the tradeoff between safety and efficiency. Ada requires execution time range checking, but it can be disabled with the directive Pragma Suppress(Index_Check). This allows the programmer to include the range checking during development, and then remove it in the final version of the program, for efficiency.

Pascal supports an array type constructor that gives the base type and the index type, which may be any discrete, ordered, finite type, including char, boolean, subrange and enumerations. (Programmers use a subrange for an array with integer indices.) The base type may be any type, including programmer-defined types such as records and arrays. For example (again, assuming that the enumerated type Day is defined as previously in this section),

```pascal
{ array type definitions in Pascal }

type
  TotalPerDay = array[Day] of integer;
  GradeDistribution = array['A' .. 'E'] of integer;

All array types are one-dimensional, but multidimensional array types can be defined as arrays of arrays of reals, and so on.

In Pascal, the array bounds are considered to be part of the array type, and must be determinable at compile time. However, this prevents dynamically-sized arrays, and implies that arrays with different bounds are instances of different types. Since Pascal is strongly typed, two arrays with different subscript ranges cannot be arguments of the same subprogram. To avoid rewriting each array subprogram for every array type, ISO Pascal introduced "conformant array parameters", which specify identifiers that can be used as the argument's subscript limits within the subprogram code. The limit identifiers are bound according to the limits of the index type of the type of the array actually passed to the subprogram. The type of the argument must have the same base type as that given for the parameter. For example,

```pascal
{ an ISO Pascal conformant array parameter }

function sum(arr: array[low .. high: IndexType] of real): real;
var
  total: real;
  index: IndexType;
begin
  total := 0.0;
  for index := low to high do
    total := total + arr[index];
  sum := total
end

In Ada, we can define an "unconstrained" array type that gives the index and base type, but does not specify the actual array limits. We then give the limits for an array object when creating it with an index constraint. For example,
-- An Ada unconstrained array type, and two instances with different bounds
type Vector is array(integer range < >) of Float;
Arr1: Vector(-100 .. 100);
Arr2: Vector(0 .. 50);

The limits in the definition of an array instance can be expressions including variables, so Ada supports
dynamically sized arrays. Ada uses parentheses for subscription because an array is a mapping like a
function. A subprogram can use an unconstrained array type as a parameter type, and can access the
argument's limits within the subprogram using the type attributes 'First and 'Last. For example,

-- an Ada function with an array parameter
function Sum(Vec: Vector) return Float is
    Total: Float := 0.0;
begin
    for Index in Vec'First .. Vec'Last loop
        Total := Total + Vec(Index);
    end loop;
    return Total;
end Sum;

Both the objects Arr1 and Arr2 can be passed to the function Sum. Ada arrays also support the type
attribute 'Range, which gives the array's index range, so we can also write the loop as for Index in
Vec'Range loop ... end loop;

C does not define an array type, but we can create array objects statically, automatically or
dynamically. The index type is always int, and array indices always begin with 0. Arrays in C have a
different interpretation than in most higher-level languages that mirrors the storage format. The array name
is regarded as a pointer to the first element of the array, and succeeding elements can be accessed via
pointer arithmetic and dereferencing. An array can be passed as the argument of a function, but a pointer
to the first element is actually passed, and an array argument can always be modified by a function. In
addition, arrays cannot be assigned or compared, so there is essentially no array type in C. The programmer
creates a two-dimensional array by defining an array of pointers, each of which points to an array, and
higher-rank arrays are defined similarly. We can allocate an array of objects dynamically by calling
malloc() with the product of the size of the element type and the number of elements as the argument. For
example:

/* creating a dynamic array in C */
pArrDbI = (double*) malloc(sizeof(double) * ARR_SIZE);

Many languages provide a feature for initializing an array object. In C, we can give a list of values
enclosed in braces, or use a string literal to initialize a char array. If the compiler can infer the number of
elements from the initializer, it need not be given. For example:

/* initialization of arrays in C */
int arr[5] = { 1, 2, 4, 8, 16 };
char greeting[] = "hello";

Unfortunately, we cannot use the braces notation to specify an array literal in an assignment or as a function
argument. As with other objects, Pascal does not provide initialization, so an array initially contains
garbage, and its values are set by a series of assignments or input operations to individual elements. Ada
supports specification of a list of initial values for an array enclosed in parentheses.

In addition to defining an array type, many languages allow us to treat the entire array as an object in its
own right by supporting coherent operations on array objects. For example, the language may permit

23 In fact, the expression arr[index] is identical to *(arr + index) (as is index[arr]), and may be used with any
pointer, not just an array name. Using a pointer to access an array sequentially allows more efficient iteration over the
elements of an array because the address of each element does not have to be calculated from the base address of the
array and the index value for each access. However, most compilers for other languages can perform this optimization.
assignment or equality test operations for array objects. We can provide additional operations by defining subprograms with parameters of array type, e.g. `isElement(Array, Element): boolean`. Some languages such as PL/I, Fortran 90 and APL provide memberwise operations on array instances. Fortran 90 and APL also include predefined operations for matrix multiplication and transposition, and for vector dot products. Ada supports the infix concatenation operator `&` for arrays.

**Records**

The most important programmer-defined type for modeling problem domain categories is the record type. A **record** ("structure" in C) is a composite object made up of a collection of components or **fields** ("members" in C) of various types. The components store values for the attributes and relationships the application maintains for an instance of the category the type represents. Each component has a name and a type, and a record instance has a value for each component.

Cobol introduced the record variable to permit processing files of information about customers, inventory items, and so on. Algol 68, Pascal, and succeeding languages support a type constructor for records. We define a record type by listing the names and types of its components, which can be any previously defined types, including other record types. The type object for the record type maintains this information, and also contains the size of an instance and the field offsets for component selection operations. We saw examples of record type definitions in Pascal and C in section 3.3. Ada permits the programmer to specify default initial values for the fields of a record, as follows:

```pascal
-- an Ada record type with default values for the fields
type Point is
  record
    X: Integer := 0;
    Y: Integer := 0;
  end record;
```

If the programmer creates an instance of the type `Point` without giving initial values for its fields, they are set to the default initial values.

The storage structure for a record type typically consists of a sequence of the storage structures for the individual fields' types. (The order of the fields in the storage structure of a record is not usually visible to the programmer.) If certain field types must be aligned on word or long word boundaries on the target hardware, "padding" within record objects may be necessary.

The basic selector operation for records is **component selection**, which takes a record instance name and a component name, and is usually denoted by an `infix .` (period). (Alternately, we can think of each component name as a selector function.) The type of a component selection expression is the type of the named field. Like an array subscription or pointer dereference expression, we can use a component selection expression as either an `l-value` or an `r-value` to set or obtain the value of a field in a record object.\(^{24}\) For example (assuming that the record type `Point` is defined as in section 3.3),

```pascal
{ component selection in Pascal }
var
  pt1, pt2: Point;
beg
  pt1.x := 1;
  pt2.y := pt1.x + 3
end
```

If a record contains a field of record type, selection operations can be composed to access fields of the nested record, e.g. `employee.birthdate.year`. Such an expression can appear in any context in which a simple variable of the last component's type can appear. The compiler translates a component selection expression as an offset into the record instance that is the sum of the sizes of the preceding components (plus any padding necessary for alignment).

\(^{24}\) Languages that support records permit assignment of individual components, even though it may not be semantically meaningful for the problem domain entity represented.
Since we refer to the components of a record by names specific to the record type, a record type introduces a new level of scope, and the component names are declared within that scope. In this way, two record types may use the same name for a component, and a variable name can also be used as a component name without conflict. The component names (and their offsets) are stored in the type object for the record type, rather than directly in the symbol table.

Many languages support treating a record instance as an individual object rather than as a group of related objects, and allow assignments or equality tests for records. We can also define subprogram parameters and return values of record type to define the operations required for instances. The language may also provide syntax for specifying each component value for an initialization or record literal. Support for such coherent operations on record instances is incomplete in the languages of the 1970s. Pascal allows record assignment, but not equality comparison, initialization, or record literals.\(^{25}\) It also permits record parameters, but not record return values. Pre-ANSI C supports initialization for static structures and structure parameters, but not structure assignment or return values. ANSI C supports initialization and assignment of structures, but not equality tests. The definition of a C structure variable can include a list of initial values for the members in braces, for example,

```c
/* structure initialization in C */
struct Point origin = { 0, 0 };
```

Initializers enclosed in braces can be nested to initialize a structure that contains a structure or array member. As with arrays, we cannot use the braces notation as a function argument or as the right side of an assignment. Records are first-class in Ada. The language supports record assignment, equality comparison, and initialization by a list of values enclosed in parentheses. We can use a record type name to construct an instance by giving values for each of the object's fields, e.g. `Point(1, 1)`, which provides a literal notation for records. Ada also supports operator overloading for record types.

**Unions**

A union type allows an instance to represent different types of entities at various points during execution of the program.\(^{26}\) Algol68 introduced the discriminated union, which contains an indication of the type currently stored in the union object, and Pascal integrated this construct with the record type. A Pascal variant record type specifies a set of fields and a "variant part" introduced by the reserved word case that contains a tag component and a set of substructures. Only one substructure is present in a particular instance at any time, and its tag field indicates which it is. For example, we can define the following variant record type whose instances represent one of three different kinds of shapes:

```pascal
{ a variant record in Pascal }

type
  ShapeType = (Circle, Rectangle, Spiral);

  record
    center: Point; \{ Point defined in section 0.3.3 \}
  case typeTag: ShapeType of
    Circle: (radius: integer);
    Rectangle: (height, width, tilt: integer);
    Spiral: (radius, spacing: integer)

A particular Shape object can contain a Circle at some point in execution, and a Rectangle at another. The fields center and typeTag are present in all instances, and the typeTag indicates which of the other substructures is stored in an instance. We access the fields in a variant record using component selection, and usually process instances of the type using a case statement that tests the tag and specifies different

---

\(^{25}\) To initialize a record instance as a unit, the programmer can define a "constructor" procedure that takes a record reference parameter and values for all the fields, and sets the fields in the object.

\(^{26}\) We will see that object-oriented languages support this capability in a safer, more controlled manner via inheritance.
operations for each variant. The type object for a variant record contains the size of an instance, the
nonvariant field names and types, and a list of the type descriptions of the variant parts.

The variant record type avoids the storage inefficiency of including all possible fields in every instance.
The storage structure includes the nonvariant fields and the amount of storage necessary for the largest
of the variant substructures. In an instance of the Shape type, there is space for a Point, the tag field, and
three integers. The height and width fields occupy the same storage as the radius and spacing fields in an
object.

To ensure type-correct operation, you should always test the type tag of a variant record before
manipulating the other fields in an instance. Unfortunately, Pascal defines no syntax for coherently
assigning the type tag and the corresponding variant fields of an instance, so each component must be
assigned independently. Since each assignment is a separate statement, the compiler cannot ensure that a
complete set of assignments has been performed. The safety of a component access cannot even be checked
at execution time since the type tag and the contents of the variant part may not be consistent. For example,
the programmer can define a record with integer and real variants, store an integer, change the tag field to
the real indicator, and access the integer's representation as a real (even if the implementation provides run-
time checking of tags). That is, a union defines aliases for the same storage that might not be the same type.
Since a variant record can be used in semantically unsound ways, you should use the type with care.

The lack of type safety for variant records has been fixed in Ada. The language supports coherent
assignment to a record object from a literal, disallows independent assignment of the tag field, and checks
the validity of assignments to the variant parts at execution time. For example (assuming that we have
defined a Shape variant record type as in the previous example),

```plaintext
-- type safety for variant records in Ada
Graphic: Shape;
Graphic := (Circle, (2, 2), 5);              -- a Circle literal specification
Graphic.Type := Spiral;                     -- illegal, not a complete specification
Graphic.Height := 5;                        -- illegal, Graphic is a Circle
Graphic.Radius := 3;                        -- OK
```

In addition, we can define a subtype or derived type whose instances can only contain one of the possible
variants with a discriminant constraint, e.g. the type Shape(Circle).

An undiscriminated union does not explicitly store an indication of which substructure is present in an
instance, and is inherently unsafe. For example, C supports the union construct that does not require a tag.
The union in C is regarded as a structure that can store any one of a set of members in an instance. Each
instance is allocated the maximum storage of the requirements of its members' types, and the offset for each
member is 0. This construct allows using objects of different types in the same context. For example, we
can define a union that can take any of three types of objects and two instances of the type, as follows:

```plaintext
/* an undiscriminated union in C */
union Data
{
  int in;
  char ch;
  double dbl;
} data1, data2;
```

We access the members of a union with component selection, e.g. data1.ch = 'n'; It is the programmer's
responsibility to keep track of what type of object is stored in an instance. For example, the language does
not prevent us from assigning an int to data1.in, and then retrieving a double from data1.db. For safety,
the C programmer should nest a union within a struct that includes a member that specifies the type of
object currently stored in an instance.

Abstract data types

We can use a record type to group the information that the program stores about a category of entities,
with the individual attributes and relationships of an entity stored as field values for the corresponding
instance. We must also define the operations required for instances of the type. In languages that do not
support abstract data types such as Pascal and C, the programmer accesses the fields within a record directly
via component selection, and codes the operations for the record type as subprograms that take instances of the type as arguments. These languages provide no direct support for packaging the storage structure for instances of a type and the operations for the type as a unit, which can then be reused in other applications that deal with the same problem domain.

An abstract data type encapsulates the definition of the storage structure for a type (i.e., the components and their types) and the type's operations in a separate package, and allows the programmer to control access to the components of the type. These types are "abstract" in the sense that a programmer using the type need not know its internal structure, just as a programmer does not need to know how floating-point numbers are represented and how that storage structure is manipulated to provide arithmetic operations. Languages such as Ada and object-oriented languages that support abstract data types provide syntactic structures for this encapsulation, and for specifying which components of the abstract data type are visible externally.

Subprogram types

Many procedural languages allow a subprogram to take another subprogram as an argument. Such a subprogram encodes a higher-level algorithm that uses specific functionality supplied by the subprogram argument. Examples include a sort function that takes the function used to compare elements for ordering as an argument, and a "mapping" function that applies an arbitrary function to every element of an array or list. The use of subprogram parameters allows the general procedure to be coded once, then reused for particular subprograms as needed. For example, Pascal allows procedure and function parameters. The following function returns the slope of the chord between the points \((x1, f(x1))\) and \((x2, f(x2))\) for a function \(f\):

```
{ a function parameter in Pascal }
f
function slope(function f(x: real): real; x1, x2: real): real;
begin
  if x1 = x2 then
    slope := 0
  else
    slope := (f(x2) - f(x1)) / (x2 - x1)
end
```

The identifier \(x\) in the parameter declaration is just a place-holder. We invoke the function \(slope\) by using the name of a function with the correct argument signature as the first argument. For example,

```
{ passing a function object as an argument }
m := slope(sin, 3.0, 5.0);
```

C supplies the type `pointer to function`, which may be used for both function parameters and function variables.

It is also convenient for a language to support a variable whose value is a subprogram, e.g. to assign an operation selected by a user interactively as its value. That is, we want to treat a subprogram as an object, and define parameters and variables of subprogram type. The basic operation on this type is invocation of the referent. Subprogram objects are not first-class in procedural languages such as Pascal and C. Pascal does not permit subprogram variables, and does not support writing anonymous subprograms or creating a new subprogram within a subprogram and returning it.

3.6 The semantics of types

Domains

In mathematics, the domain of a function is the set of objects for which the function is defined. As stated in section 3.3, a type specifies the structure of its instances and the operations that can be performed on those objects. That is, a type specifies a domain over which its operations apply. A subprogram with a parameter of that type depends on its argument being an object in that domain so that it can use the domain's selectors and other operations. These dependencies might or might not be enforced by the language.
Early programming languages specified a fixed set of domains that were built into the language. Languages designed since Algol68 provide type constructors that we can use to introduce new domains. Domains represented by types can be independent, or can overlap in the sense that an instance of one type can be regarded as an instance of the other. Two overlapping domains can be intersecting or merged, or one can be a subset of another. The domains boolean and integer are regarded as independent in Pascal. The domains integer and real intersect (rather than there being a subset relationship) because some integers cannot be represented exactly as reals due to the limitation on the number of significant digits. We can create merged domains in Pascal by defining type aliases. A subrange definition creates a new domain that is a subset of the base type domain. In C, the domains logical value, character, and integer are merged since the translator makes no distinctions among them. This can be convenient in some circumstances, but it prevents the compiler from catching some errors. Merged domains can also be created in C using the typedef statement.

Type checking

In statically typed languages, the type of every identifier is known at compile time via its declaration. In a strongly typed language, the compiler enforces the domain identity of objects and the domain requirements of operators and subprograms for their arguments. It prevents the program from assigning values of one type to variables of another, and ensures that the type of an identifier in an expression is correct for the operation being performed on its referent. If the type of the object and the type expected do not match and the language defines a conversion that obtains the correct type, the compiler performs it implicitly. Type checking prevents programs from performing meaningless operations since such errors are usually typographic or logic errors. Ada is strongly typed, and Pascal is strongly typed except when variant records are used.

In a weakly typed language, type checking is incomplete or can be circumvented, and the resulting errors are not caught until incorrect results are obtained or the program aborts unexpectedly. Fortran is weakly typed due to the EQUIVALENCE and COMMON statements, and the use of "Hollerith" (string) constants stored in integer variables. In Fortran and pre-ANSI C, the types of the arguments of a function call are not checked, so these languages are weakly typed. For example, the C standard library function sqrt() expects a double argument. For the invocation sqrt(7), a pre-ANSI compiler might simply push the integer value 7 on the stack, rather than creating and pushing a double with that value. The function code will interpret that argument's bytes (and probably succeeding bytes!) directly as a double value, resulting in an incorrect return value.

Dynamically typed languages may employ either strong or weak typing. As stated in section 2.2, with strong typing, objects must contain a representation of their type, and type checking occurs when an operation is applied to the object, i.e. at execution time rather than at compile time. If an object's type is inappropriate for the operation, an error occurs and execution is halted (if the programmer has not specified actions to be taken in that case). The language provides this checking for built-in operations. For programmer-defined subprograms, the built-in type checking is performed when an operation is applied to an argument or component, since parameter types are not specified. As stated in section 2.2, most dynamically typed languages provide type predicates so that a subprogram can perform type checking explicitly.

The process of type checking is complicated by overlapping domains, which indicate the compatibility or convertibility of some types. In Pascal, type aliases do not define new domains, so operations for one type can be applied to instances of the other. For some subset or intersection relationships, the representation of equivalent objects is different for each domain, e.g. integer and real in Pascal, so the compiler must perform an implicit conversion. In some languages, domains that are represented in the same manner are considered compatible, e.g. enums and ints in C.

Type checking provides increased safety and reliability, but programmers occasionally need to break type safety. A system programmer may need to manipulate a region of storage irrespective of the type of object contained, or a programmer may wish to use the binary encoding of an object as the basis for a hash function. Some strongly typed languages provide mechanisms for evading type checking, or for interpreting the same region of storage as an instance of different types for these purposes. For example, in Pascal, the programmer can use a variant record with pointer and integer variants to provide address arithmetic. In C, the programmer can use a union, or access an object through pointers with different base types via a pointer cast. ANSI C's type checking can be circumvented because we can cast a pointer to any object to char*, and then access that object as a sequence of bytes. Ada specifically provides no loopholes in its type system so that programs are safe, readable and portable.
Explicit casts and conversions

We will draw a distinction between "casts" and " conversions" because they are different operations.\textsuperscript{27} A type cast is a change in the type label on an object that does not change the object's representation. Examples include interpreting an instance of a subrange as an instance of the base type in Pascal, and using an instance of unsigned int as an int in C. The purpose of a cast is to indicate to the compiler that it is meaningful to use the object in a context that is not normally valid for its defined type. The compiler generates no code for a cast, nor does it create a new object. A cast is inherently unsafe because it changes the semantics of an object, and should be used with good reason and care. In C, we may cast a pointer to a pointer of a different type (and in pre-ANSI C, this can occur implicitly).

In many cases, it is valid to convert a value for one type to a corresponding value for another type. A type conversion creates a new object whose value is equivalent to that of the converted object. The conversion creates a new object because instances of the types have a different size or encoding. Some numeric conversions preserve all the information in the object, e.g. converting an integer to a real, or a conversion from char to double in C. Such an operation is termed a "promotion" or a "widening" conversion because the size of the new object is larger than the original. Other conversions are "demotions" or "narrowing" conversions that lose information, e.g. a conversion from a double to an int in C. Each language defines a set of conversions among built-in types that can be invoked explicitly using function call syntax, e.g. the Pascal functions trunc and round. (In C, we enclose the type name in parentheses preceding the object to be converted, e.g. (long double) num.) We can code conversions among programmer-defined types or between programmer-defined and built-in types as functions.

Implicit coercions

Casts and conversions can be invoked explicitly by the programmer or implicitly by the compiler. A cast or conversion that is performed implicitly is called a coercion. The compiler can use coercions in an initialization, assignment, expression evaluation, or argument passing to obtain the type required for the operation. For example, when we add an integer and a real in most languages, the integer is automatically converted to a real, and real addition is performed. In C, when adding an int and a long int, the int is promoted to a long. In fact, either conversion could be done in these cases, but most languages perform the promotion so that no information is lost implicitly. Each language defines the coercions that the compiler can use. For example, reals can be coerced to integers in C, but not in Pascal. PL/I will even implicitly convert a string variable that contains a sequence of characters representing a numeric literal to a number in an arithmetic expression. For safety and portability, Ada does not support implicit coercions, but the programmer can overload operators and subprograms to provide any argument signatures necessary.

Some languages that support abstract data types, such as C++, allow the programmer to define conversions as part of the definition of the type, which can then be used implicitly by the compiler.

4. Abstraction mechanisms

4.1 Software complexity

Problem domain complexity

Much of the complexity in a software system results from the characteristics of the real world system that it models, and therefore is an essential property of the system. For example, the software that controls a manufacturing process must manage the properties and location of each component, the time dependencies among processes, quality control and error response, and so on. Similarly, a simulation of an astronomical system must represent the various types of stars, planets, nebulae and other celestial bodies, and their structure, properties and interactions.

Factors external to the system the software models make it difficult to specify the software system's requirements exactly and completely. For example, the software that controls a mobile robot must deal with objects and other robots in its environment. In addition, changes in the problem domain or in the requirements for the software's interaction with it frequently occur during the lifetime of the system. For

\textsuperscript{27} Both type casts and conversions are referred to as "casts" in the C literature.