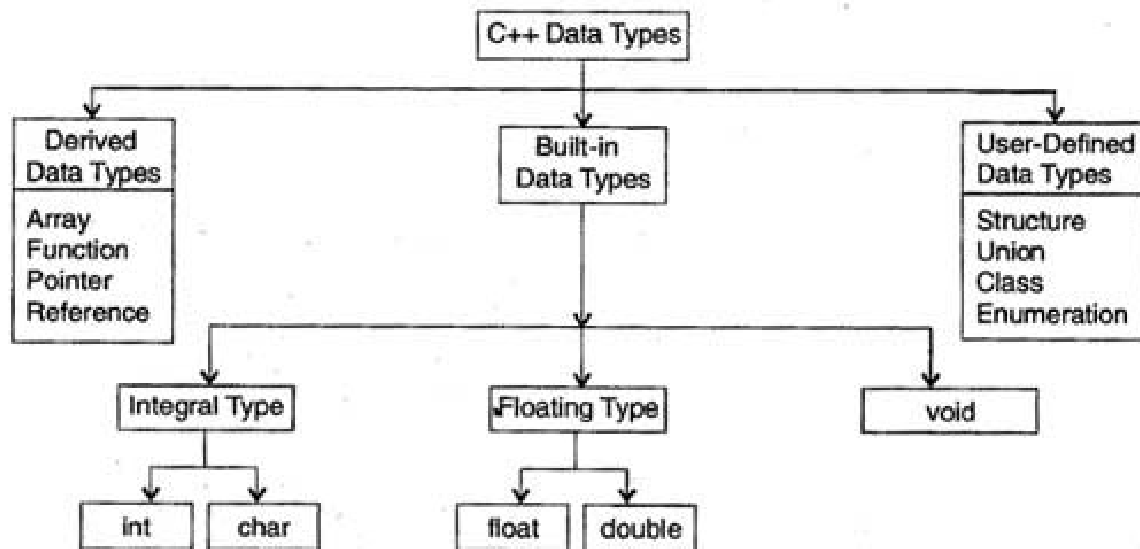
A] STRUCTURING OF DATA:

WHAT IS A TYPE?

1. Programming languages organize data through the concept of type. Types are used as a way to classify data according to different categories.
2. They are more, however, than pure sets of data. Data belonging to a type also share certain semantic behaviors.
3. A type is thus more properly defined as a set of values and a set of operations that can be used to manipulate them.
4. For example, the type BOOLEAN of languages like Ada and Pascal consists of the values TRUE and FALSE; Boolean algebra defines operators NOT, AND, and OR for BOOLEANs. BOOLEAN values may be created, for example, as a result of the application of relational operators (<, ð, >, Š, +, ¦) among INTEGER expressions.

B] Built-in types and primitive types:



Various Data Types in C++

Variable Type	Keyword	Bytes Required	Range
Character	char	1	-128 to 127
Unsigned character	unsigned char	1	0 to 255
Integer	int	2	-32768 to 32767
Short Integer	short int	2	-32768 to 32767
Long Integer	long int	4	-2,147,483,648 to 2,147,438,647
Unsigned Integer	unsigned int	2	0 to 65535
Unsigned Short integer	unsigned short int	2	0 to 65535
Unsigned Long Integer	unsigned long int	4	0 to 4,294,967,295
Float	float	4	1.2E-38 to
Double	double	8	2.2E-308 to
Long Double	long double	10	3.4E-4932 to 1.1E+4932

B-1]Advantages of built-in types:

1. **Hiding of the underlying representation.** This is an advantage provided by the abstractions of higher-level languages over lower-level (machine-level) languages. The programmer does not have access to the underlying bit string that represents a value of a certain type. The programmer may change such bit string by applying operations, but the change is visible as a new value of the built-in type, not as a new bit string. Invisibility of the underlying representation has the following benefits:

   Programming style. The abstraction provided by the language increases program readability by protecting the representation of objects from undisciplined manipulation. This contrasts with the underlying conventional hardware, which does not enforce protection, but usually allows any view to be applied on any bit string. For example, a location containing an integer may be added to one containing a character, or even to a location containing an instruction.

   Modifiability. The implementation of abstractions may be changed without affecting the programs that make use of the abstractions. Consequently, portability of programs is also improved, that is, programs can be moved to machines that use different internal data representations. One must be careful, however, regarding the precision of data representation, that might change for different implementations. For example, the range of representable integer values is different for 16- and 32-bit machines.

2. **Correct use of variables** can be checked at translation time. If the type of each variable is known to the compiler, illegal operations on a variable may be caught while the

program is translated. Although type checking does not prevent all possible errors to be caught, it improves our reliance on programs. For example, in Pascal or Ada, it cannot ensure that J will never be zero in some expression I/J, but it can ensure that it will never be a character.

3. **Resolution of overloaded** operators can be done at translation time. For readability purposes, operators are often overloaded. For example, + is used for both integer and real addition, * is used for both integer and real multiplication. In each program context, however, it should be clear which specific hardware operation is to be invoked, since integer and real arithmetic differ. In a statically typed language, where all variables are bound to their type at translation time, the binding between an overloaded operator and its corresponding machine operation can be established at translation time, since the types of the operands are known. This makes the implementation more efficient than in dynamically typed languages, for which it is necessary to keep track of types in run-time descriptor.
4. **Accuracy control.** In some cases, the programmer can explicitly associate a specification of the accuracy of the representation with a type. For example, FORTRAN allows the user to choose between single and double-precision floating-point numbers. In C, integers can be short int, int, or long int. Each C compiler is free to choose appropriate size for its underlying hardware, under the restriction that short int and int are at least 16 bits long, long int is at least 32 bits long, and the number of bits of short int is no more than the number of bits of int, which is no more than the number of bits of long int. In addition, it is possible to specify whether an integer is signed or unsigned. Similarly, C provides both float (for single-precision floating point numbers) and double (for double precision floating point numbers). Accuracy specification allows the programmer to direct the compiler to allocate the exact amount of storage that is needed to represent the data with the desired precision.

C] Data aggregates and type constructors:

1. Programming languages allow the programmer to specify aggregations of elementary data objects and, recursively, aggregations of aggregates. They do so by providing a number of constructors. The resulting objects are called compound objects.

2. A well-known example is the array constructor, which constructs aggregates of homogeneous-type elements. An aggregate object has a unique name.
3. In some cases, manipulation can be done on a single elementary component at a time, each component being accessible by a suitable selection operation. In many languages, it is also possible to manipulate (e.g.,assign and compare) entire aggregates. Older programming languages, such as FORTRAN and COBOL, provided only a limited number of constructors.
4. For example, FORTRAN only provided the array constructor; COBOL only provided the record constructor. In addition, through constructors, they simply provided a way to define a new single aggregate object, not a type.
5. Later languages, such as Pascal, allowed new compound types to be defined by specifying them as aggregates of simpler types. In such a way, any number of instances of the newly defined aggregate can be defined. According to such languages, constructors can be used to define both aggregate objects and new aggregate types.

✓ **Cartesian product:**

1. The Cartesian product of n sets $A_1, A_2, \ldots, A_n$, denoted $A_1 \times A_2 \times \ldots \times A_n$, is a set whose elements are ordered n -tuples $(a_1, a_2, \ldots, a_n)$, where each a_k belongs to A_k.
2. Programming languages view elements of a Cartesian product as composed of a number of symbolically named fields. In the example, a polygon could be declared as composed of an integer field (no_of_edges) holding the number of edges and a real field (edge_size) holding the length of each edge.
3. Examples of Cartesian product constructors in programming languages are structures in C, C++, Algol 68 and PL/I, records in COBOL, Pascal, and Ada. COBOL was the first language to introduce Cartesian products, which proved to be very useful in data processing applications.
4. As an example of a Cartesian product constructor, consider the following C declaration, which defines a new type reg_polygon and two objects a_pol and b_pol ;

```
struct reg_polygon {
int no_of_edges;
```

float edge_size;

};

struct reg_polygon pol_a, pol_b = {3, 3.45};

5. The two regular polygons pol_ a and pol_b are initialized as two equilateral triangles whose edge is 3.45. The notation {3, 3.45} is used to implicitly define a constant value (also called a compound value) of type reg_polygon (the polygon with 3 edges of length 3.45).
6. The fields of an element of a Cartesian product are selected by specifying their name in an appropriate syntactic notation. In the C example, one may write:

pol_a.no_of_edges = 4;

7. to make pol_a quadrilateral. This syntactic notation for selection, which is common in programming languages, is called the dot notation.

✓ **Powerset:**

1. It is often useful to define variables whose value can be any subset of a set of elements of a given type T . The type of such variables is powerset (T) , the set of all subsets of elements of type T . Type T is called the base type. For example, suppose that a language processor accepts the following set O of options

LIST_S, to produce a listing of the source program;

LIST_O, to produce a listing of the object program;

OPTIMIZE, to optimize the object code;

SAVE_S, to save the source program in a file;

SAVE_O, to save the object program in a file;

EXEC, to execute the object code.

A command to the processor can be any subset of O , such as

{LIST_S, LIST_O}

{LIST_S, EXEC}

That is, the type

{OPTIMIZE, SAVE_O, EXEC}

of a command is powerset (O) .

2. Variables of type powerset (T) represent sets. The operations permitted on such variables are set operations, such as union and intersection. Although sets (and powersets) are common and basic mathematical concepts, only a few languages–notably, Pascal and Modula-2–provide them through built-in constructors and operations.
3. Also, the set-based language SETL makes sets the very basic data structuring mechanism. For most other languages, set data structures are provided through libraries. For example, the C++ standard library provides many data structures, including sets.

- **Sequencing:**

1. A sequence consists of any number of occurrences of elements of a certain component type CT. The important property of the sequencing constructor is that the number of occurrences of the component is unspecified; it therefore allows objects of arbitrary size to be represented.
2. It is rather uncommon for programming languages to provide a constructor for sequencing. In most cases, this is achieved by invoking operating system primitives which access the file system.
3. It is therefore difficult to imagine a common abstract characterization of such a constructor. Perhaps the best example is the file constructor of Pascal, which models the conventional data processing concept of a sequential file. Elements of the file can be accessed sequentially, one after the other.
4. Modifications can be accomplished by appending a new values at the end of an existing file. Files are provided in Ada through standard libraries, which support both sequential and direct files. Arrays and recursive list definitions (defined next) may be used to represent sequences, if they can be stored in main memory.
5. If the size of the sequence does not change dynamically, arrays provide the best solution. If the size needs to change while the program is executing, flexible arrays or lists must be used. The C++ standard library provides a number of sequence implementations, including vector and list.

✓ Finite mapping

1. Finite mapping is a function from a finite set of values of a domain type DT onto values of a range type RT . Such function may be defined in programming languages through the use of the mechanisms provided to define routines.
2. This would encapsulate in the routine definition the law associating values of type RT to values of type DT . This definition is called intensional. In addition, programming languages, provide the array constructor to define finite mappings as data aggregates. This definition is called extensional, since all the values of the function are explicitly enumerated. For example, the C declaration

   char digits [10];

3. defines a mapping from integers in the subrange 0 to 9 to the set of characters, although it does not state which character corresponds to each element of the subrange. The following statements

   for (i = 0; i < 10; ++i)

   digits [i] = ' ';

4. define one such correspondence, by initializing the array to all blank characters. This example also shows that an object in the range of the function is selected by indexing, that is, by providing the appropriate value in the domain as an index of the array.
5. Thus the C notation digits [i] can be viewed as the application of the mapping to the argument i . Indexing with a value which is not in the domain yields an error. Some languages specify that such an error is to be trapped.
6. Such a trap, however, may in general only occur at run time. C arrays provide only simple types of mappings, by restricting the domain type to be an integer subrange whose lower bound is zero.
7. Other programming languages, such as Pascal, require the domain type to be an ordered discrete type. For example, in Pascal, it is possible to declare

   var x: array [2. .5] of integer;

8. which defines x to be an array whose domain type is the subrange 2. .5.

Union and discriminated union

9. Cartesian products defined in Section C allow an aggregate to be constructed through the conjunction of its fields. For example, we saw the example of a polygon, which was represented as an integer (the number of edges) and a real (the edge size). In this section we explore a constructor which allows an element (or a type) to be specified by a disjunction of fields.
10. For example, suppose we wish to define the type of a memory address for a machine providing both absolute and relative addressing. If an address is relative, it must be added to the value of some INDEX register in order to access the corresponding memory cell. Using C, we can declare union address

11. {

12. short int offset;

13. long unsigned int absolute;

14. };

15. The declaration is very similar to the case of a Cartesian product. The difference is that here fields are mutually exclusive. Values of type address must be treated differently if they denote offsets or absolute addresses.
16. Given a variable of type address, however, there is no automatic way of knowing what kind of value is currently associated with the variable (i.e., whether it is an absolute or a relative address).
17. The burden of remembering which of the fields of the union is current rests on the programmer. A possible solution is to consider an address to be an element of the following type:

18. struct safe_address {

19. address location;

20. descriptor kind;

21. };

22. where descriptor is defined as an enumeration enum descriptor {abs, rel};

23. A safe address is defined as composed of two fields: one holds an address, the other holds a descriptor. The descriptor field is used to keep track of the current address kind. Such a field must be updated for each assignment to the corresponding location field.

✓ Compound values

1. Besides supporting the ability to define structured variables, some languages allow constant values of compound (or composite) objects to be denoted. For example, in C++ one can write:
2. char hello[ ] = {’h’, ’e’, ’l’, ’l, ’o’, ’\0’};

```
struct complex {
float x, y;
};
complex a = {0.0, 1.1};
```

3. This fragment initializes array hello to the array value {’h’, ’e’, ’l’, ’l,’ ’o’, ’\0’} , i.e., the string "hello" (’\0’ is the null character denoting the end of the string). Structure a is initialized to the structure value {0.0, 1.1} . Ada provides a rich and elaborate set of facilities to define values of compound objects.

D] User-defined types and abstract data types:

✓ User-defined types

1. Modern programming languages provide many ways of defining new types, starting from built-in types. The simplest way, mentioned in Section B, consists of defining new elementary types by enumerating their values.
2. The constructors reviewed in the previous sections go one step further, since they allow complex data structures to be composed out of the built-in types of the language. Modern languages also allow aggregates built through composition of built-in types to be named as new types.
3. Having given a type name to an aggregate data structure, one can declare as many variables of that type as necessary by simple declarations.
4. For example, after the C declaration which introduces a new type name complex.

```
struct complex {
float real_part, imaginary_part;
```

}

any number of instance variables may be defined to hold complex values: complex a, b, c, . . .;

5. By providing appropriate type names, program readability can be improved. In addition, by factoring the definition of similar data structures in a type declaration, modifiability is also improved.
6. A change that needs to be applied to the data structures is applied to the type, not to all variable declarations. Factorization also reduces the chance of clerical errors and improves consistency. The ability to define a type name for a user defined data structure is only a first step in the direction of supporting data abstractions.

✓ Abstract data types in C++

1. Abstract data types can be defined in C++ through the class construct. A class encloses the definition of a new type and explicitly provides the operations that can be invoked for correct use of instances of the type.
2. As an example, Figure shows a class defining the type of the geometrical concept of point.

```
class point {
int x, y;
public:
point (int a, int b) { x = a; y = b; }
void x_move (int a) { x += a; }
void y_move (int b ){ y += b; }
void reset ( ) { x = 0; y = 0; }
};
```

3. A class can be viewed as an extension of structures (or records), where fields can be both data and routines. The difference is that only some fields (declared public) are accessible from outside the class.
4. Non-public fields are hidden to the users of the class. In the example, the class construct encapsulates both the definition of the data structure defined to represent points (the two integer numbers x and y) and of the operations provided to manipulate points.

5. The data structure which defines a geometrical point (two integer coordinates) is not directly accessible by users of the class. Rather, points can only be manipulated by the operations defined as public routines, as shown by the following fragment:

```
point p1 (1, 3);
point p2 (55, 0);
point* p3 = new point (0, 0);
p1.x_move (3);
p2.y_move (99);
p1.reset ( );
```

6. The fragment shows how operations are invoked on points by means of the dot notation; that is, by writing "object_name.public_routine_name". The only exceptions are the invocations of constructors and destructor. We discuss constructors below; destructor will be discussed in a later example.
7. A constructor is an operation that has the same name of the new type being defined (in the example, point). A constructor is automatically invoked when an object of the class is allocated. In the case of points p1 and p2 , this is done automatically when the scope in which they are declared is entered. In the case of the dynamically allocated point referenced by p3 , this is done when the new instruction is executed.
8. Invocation of the constructor allocates the data structure defined by the class and initializes its value according to the constructor's code. A special type of constructor is a copy constructor. The constructor we have seen for point builds a point out of two int values.
9. A copy constructor is able to build a point out of an existing point. The signature of the copy constructor would be:

```
point (point&)
```

10. It is also possible to define generic abstract data types, i.e., data types that are parametric with respect to the type of components. The construct provided to support this feature is the template.

E] Type systems:

1. Types are a fundamental semantic concept of programming languages. More-over, programming languages differ in the way types are defined and behave, and typing issues are often quite subtle.
2. Having discussed type concepts informally in different languages so far, we now review the foundations for a theory of types.
3. The goal is to help the reader understand the type system adopted by a language, defined as the set of rules used by the language to structure and organize its collection of types. Understanding the type system adopted by a language is perhaps the major step in understanding the language's semantics.

✓ Static versus dynamic program checking

1. Before focusing our discussion on type errors, a brief digression is necessary to discuss more generally the kinds of errors that may occur in a program, the different times at which such errors can be checked, and the effect of checking times on the quality of the resulting programs.
2. Errors can be classified in two categories: language errors and application errors. Language errors are syntactic and semantic errors in the use of the programming language. Application errors are deviations of the program behavior with respect to specifications (assuming specifications capture the required behavior correctly).
3. The programming language should facilitate both kinds of errors to be identified and removed. Ideally, it should help prevent them from being introduced in the program. In general, programs that are readable and well structured are less error prone and easier to check.
4. Hereafter we concentrate on language errors. A discussion of application errors is out of the scope of this book: software design methods address application errors. Therefore, here the term "error" implicitly refers to "language error".
5. Error checking can be accomplished in different ways, that can be classified in two broad categories: static and dynamic. Dynamic checking requires the program to be executed on sample input data.

6. Static checking does not. In general, if a check can be performed statically, it is preferable to do so instead of delaying the check to run-time for two main reasons. First, potential errors are detected at run time only if one can provide input data that cause the error to be revealed. For example, a type error might exist in a portion of the program that is not executed by the given input data.
7. Second, dynamic checking slows down program execution. Static checking is often called compile-time (or translation-time) checking. Actually, the term “compile-time checking” may not be an accurate synonym of “static checking”, since programs may be subject to separate compilation and some static checks might occur at link time.
8. For example, the possible mismatch between a routine called by one module and defined in another might be checked at link time. Conventional linkers, unfortunately, seldom perform such checks.
9. For simplicity, we will continue to use the terms static checking and compile-time (or translation-time) checking interchangeably. Static checking, though preferable to dynamic checking, does not uncover all language errors.
10. Some errors only manifest themselves at run time. For example, if div is the operator for integer division, the compiler might check that both operands are integer. However, the program would be erroneous if the value of the divisor is zero. This possibility, in general, cannot be checked by the compiler.

✓ Strong typing and type checking

1. The type system of a language was defined as the set of rules to be followed to define and manipulate program data. Such rules constrain the set of legal programs that can be written in a language.
2. The goal of a type system is to prevent the writing of type unsafe programs as much as possible. A type system is said to be strong if it guarantees type safety; i.e., programs written by following the restrictions of the type system are guaranteed not to generate type errors.
3. A language with a strong type system is said to be a strongly typed language. If a language is strongly typed, the absence of type errors from programs can be guaranteed

by the compiler. A type system is said to be weak if it is not strong. Similarly, a weakly typed language is a language that is not strongly typed.

✔ Type compatibility

1. A strict type system might require operations that expect an operand of a type T to be invoked legally only with a parameter of type T . Languages, however, often allow more flexibility, by defining when an operand of another type say Q –is also acceptable without violating type safety.
2. In such a case, we say that the language defines whether, in the context of a given operation, type Q is compatible with type T . Type compatibility is also sometimes called conformance or equivalence. When compatibility is defined precisely by the type system, a type checking procedure can verify that all operations are always invoked correctly, i.e., the types of the operands are compatible with the types expected by the operation.
3. Thus a language defining a notion of type compatibility can still have a strong type system. Figure 38 shows a sample program fragment written in a hypothetical programming language.

```
struct s1{
int y;
int w;
};
struct s2{
int y;
int w;
};
struct s3 {
int y;
};
s3 func (s1 z)
{
. . .
};
```

```
. . .
s1 a, x;
s2 b;
s3 c;
int d;
. . .
a = b; --(1)
x = a; --(2)
c = func (b); --(3)
d = func (a); --(4)
```

4. The strict conformance rule where a type name is only compatible with itself is called name compatibility. Under name compatibility, in the above example, instruction (2) is type correct, since a and x have the same type name. Instruction (1) contains a type error, because a and b have different types. Similarly, instructions (3) and (4) contain type errors.
5. In (3) the function is called with an argument of incompatible type; in (4) the value returned by the function is assigned to a variable of an incompatible type. Structural compatibility is another possible conformance rule that languages may adopt.
6. Type T1 is structurally compatible with type T2 if they have the same structure. This can be defined recursively as follows:
   • T1 is name compatible with T2; or
   • T1 and T2 are defined by applying the same type constructor to structurally compatible corresponding type components.
7. According to structural equivalence, instructions (1), (2), and (3) are type correct. Instruction (4) contains a type error, since type s3 is not compatible with int.
8. Note that the definition we gave does not clearly state what happens with the field names of Cartesian products (i.e., whether they are ignored in the check or they are required to coincide and whether structurally compatible fields are required to occur in the same order or not).

9. For simplicity, we assume that they are required to coincide and to occur in the same order. In such a case, if we rename the fields of s2 as y1 and w1 , or permute their occurrence, s2 would no longer be compatible with s1 .

✔ Type conversions

1. Suppose that an object of type T 1 is expected by some operation at some point of a program. Also, suppose that an object of type T 2 is available and we wish to apply the operation to such object.
2. If T 1 and T 2 are compatible according to the type system, the application of the operation would be type correct. If they are not, one might wish to apply a type conversion from T 2 to T 1 in order to make the operation possible.
3. More precisely, let an operation be defined by a function fun expecting a parameter of type T 1 and evaluating a result of type R 1 : fun: T 1 -> R 1
4. Let x 2 be a variable of type T 2 and y 2 of type R 2 . Suppose that T 1 and T 2 (R 1 and R 2) are not compatible. How can fun be applied to x 2 and the result of the routine be assigned to y 2 ? This would require two conversion functions to be available, t 21 and r 12 , transforming objects of type T 2 into objects of type T 1 and objects of type R 1 into objects of type R 2 , respectively:
   t 21 : T 2 -> T 1
   r 12 : R 1 -> R 2
5. Thus, the intended action can be performed by first applying t 21 to x 2 , evaluating fun with such argument, applying r 12 to the result of the function, and finally assigning the result to y 2 . That is:
   (i) y 2 = r 12 (fun (t 21 (x 2)))
6. For some languages any required conversions are applied automatically by the compiler. Following the Algol 68 terminology, we will call such automatic conversions coercions. In the example, if coercions are available, the programmer might simply write
   (ii) y 2 = fun (x 2)
7. and the compiler would automatically convert (ii) into (i) . In general, the kind of coercion that may occur at a given point (if any) depends on the context. For example, in C if we write

x = x + z;

8. where z is float and x is int , x is coerced to float to evaluate the arithmetic operator + (which stands for real addition), and the result is coerced to int for the assignment. That is, the arithmetic coercion is from int to float , but the assignment coercion is from float to int .
9. C provides a simple coercion system. In addition, explicit conversions can be applied in C using the cast construct. For example, a cast can be used to override an undesirable coercion that would otherwise be applied in a given context. For example, in the above assignment, one can force a conversion of z to int by writing

   x = x + (int) z;
10. Such an explicit conversion in C is semantically defined by assuming that the expression to be converted is implicitly assigned to an unnamed variable of the type specified in the cast, using the coercion rules of the language.
11. The conversion function INTEGER provided by Ada computes an integer from a floating point value by rounding to the nearest integer. The existence of coercion rules in a language has both advantages and disadvantages.
12. The advantage is that many desirable conversions are automatically provided by the implementation.
13. The disadvantage is that since implicit transformations happen behind the scenes, the language becomes complicated and programs may be obscure. In addition, coercions weaken the usefulness of type checking, since they override the declared type of objects with default, context sensitive transformations.

✔ Types and subtypes

1. If a type is defined as a set of values with an associated set of operations, a subtype can be defined to be a subset of those values (and, for simplicity, the same operations). In this section we explore this notion in the context of conventional languages, ignoring the ability to specify user-defined operations for subtypes.
2. If ST is a subtype of T , T is also called ST ’s supertype (or parent type). We assume that the operations defined for T are automatically inherited by ST . A language supporting subtypes must define:

1. a way to define subsets of a given type;

2. compatibility rules between a subtype and its supertype.

3. Pascal was the first programming language to introduce the concept of a subtype, as a subrange of any discrete ordinal type (i.e., integers, boolean, character, enumerations, or a subrange thereof). For example, in Pascal one may define natural numbers and digits as follows:

   type natural = 0. .maxint;

   digit = 0. .9;

   small = -9. .9;

   where maxint is the maximum

   integer value representable by an implementation.

4. A Pascal program can only define a subset of contiguous values of a discrete type. For example, it cannot define a subtype EVEN of all even integers or multiples of ten in the range -1000. .1000. Different subtypes of a given type are considered to be compatible among themselves and with the supertype.

5. However, type safe operations are not guaranteed to evaluate with no error. No error arises if an object of a subtype is provided in an expression where an object of its supertype is expected. For example, if an expression requires an integer, one may provide a natural; if it expects a natural, one might provide a digit.

6. If, however, a small is provided where a digit is expected, an error arises if the value provided is not in the range expected. That is, if an argument of type T is provided to an operation expecting an operand of type R , the expression is type safe if either R or T is a subtype of the other, or both are subtypes of another type Q .

7. No value error will occur at run time if T is a subtype of R . In all other cases, the operation must be checked at run time and an error may arise if the value transmitted does not belong to the expected type. Ada provides a richer notion of subtype than Pascal. A subtype of an array type can constrain its index; a subtype of a variant record type can freeze the variant; a subtype of a discrete ordinal type is a finite subset of contiguous values.

8. Examples of Ada types and subtypes are shown in Figure.

   type Int_Vector is array (Integer range < >) of Integer;

```
type Var_Rec (Tag: Boolean) is
record X: Float;
case Tag of
when True => Y: Integer;
Z: Real;
when False=> U: Char;
end case;
end record;
subtype Vec_100 is Int_Vector (0. .99);
--this subtype constrains the bounds of the array to 0. .99
subtype X_true is X (True);
--this subtype freezes the variant where Tag = True; objects of the subtype thus
--have fields X, Y, and Z;
subtype SMALL is Integer range -9. .9;
--this subtype defines a small set of integers
```

9. Ada subtypes do not define new types. All values of all subtypes of a certain type T are of type T. The subtype construct can be viewed as a way to signal that certain run-time checks must be inserted by the compiler to ensure that objects of a given subtype always receive the specified restricted set of values.

✔ Generic types

1. As we mentioned, modern languages allow parameterized (generic) abstract data types to be defined. A typical example is a stack of elements of a parameter type T, whose operations have the following signatures:

```
push: stack (T) x T -> stack (T)
pop: stack (T) -> stack (T) x T
length: stack (T) -> int
--pushes an element on top of the stack
--extracts the topmost element from the stack
--compute the length of the stack
```

2. In the example, the abstract data type being defined is parametric with respect to a type, and the operations of the generic type are therefore defined as generic routines.
3. The operations defined for the type stack(T) are supposed to work uniformly for any possible type T . However, since the type is not known, how can such routines be type checked to guarantee type safety?
4. A possibility is provided by languages like Ada, C++, and Eiffel, where generic types must be explicitly instantiated at compile time by binding parameter types to “real” types, that are known at compile time.
5. This achieves static typing for each instance of each generic type, and therefore each instance is statically checked to ensure type safety.

✔ Monomorphic versus polymorphic type systems:

1. A simple strong type system can be provided by a statically typed language where every program entity (constant, variable, routine) has a specific type, defined by a declaration, and every operation requires that an operand of exactly the type that appears in the operation definition can be provided.
2. For such a language, it is possible to verify at compile time that any occurrence of that constant, variable, or routine is type correct. Such a type system is called monomorphic (from ancient Greek, “single shape”): every object belongs to one and only one type.
3. By contrast, in a polymorphic (“multiple shape”) programming languages every constant and every variable can belong to more than one type. Routines (e.g., functions) can accept as a formal parameter actual parameters of more than one type.
4. By examining closely traditional programming languages like C, Pascal, or Ada, however, we have seen in the previous secions that all deviate from strict monomorphism in one way or another.
5. Polymorphism can be classified as shown in Figure. For the sake of simplicity and abstraction, let us discuss Figure in the case of polymorphic functions, i.e., mathematical functions whose arguments (domain and range) can belong to more than one type.

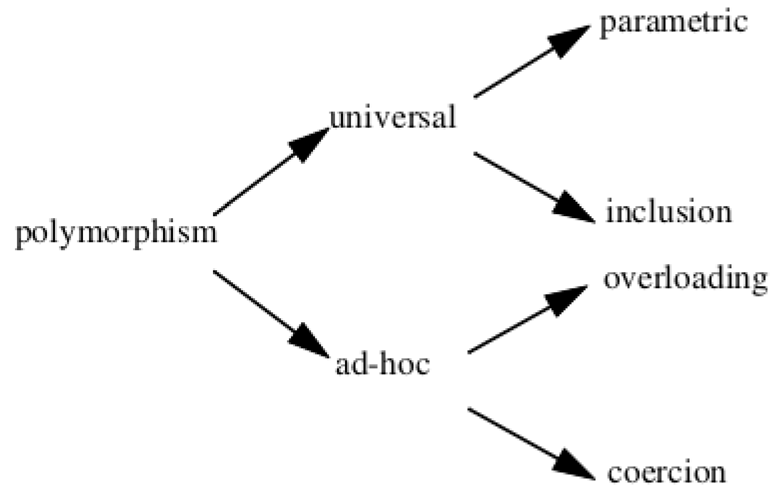


Fig: Ploymorphism

6. A first distinction is between universal polymorphism and ad-hoc polymorphism. Ad-hoc polymorphism does not really add to the semantics of a monomorphic language. Ad-hoc polymorphic functions work on a finite and often small set of types and may behave differently for each type.
7. Universal polymorphism characterizes functions that work uniformly for an infinite set of types, all of which have some common structure. Whereas an ad-hoc polymorphic function can be viewed as a syntactic abbreviation for a small set of different monomorphic functions, a universal polymorphic function executes the same code for arguments of all admissible types. The two major kinds of ad-hoc polymorphism are overloading and coercion.In overloading, the same function name can be used in different contexts to denote different functions, and in each context the function actually denoted by a given name is uniquely determined.
8. A coercion is an operation that converts the argument of a function to the type expected by the function. In such a case, polymorphism is only apparent: the function actually works for its prescribed type, although the argument of a different type may be pased to it, but it is automatically transformed to the required type prior to function evaluation.
9. Coercions can be provided statically by code inserted by the compiler in the case of statically typed languages, or they are determined dynamically by run-time tests on type descriptors, in the case of dynamically typed languages.

10. Overloading and coercion can be illustrated by the C example of the arithmetic expression a + b . In C, + is an ad-hoc polymorphic function, whose behavior is different if it is applied to float values or int numbers. In the two cases, the two different machine instructions float+ (for real addition) or int+ would be needed.
11. If the two operands are of the same type–say, float –the + operator is bound to float+ ; if both are bound to int , + is bound to int+ . The fact that + is an overloaded operator is a purely syntactic phenomenon. Since the types of the operands are known statically, one might eliminate overloading statically by substituting the overloaded + operator with float+ or int+ , respectively.
12. If the types of the two operands are different (i.e., integer plus real or real plus integer), however, the float+ operator is invoked after converting the integer operand to real.

F] Structuring of Computations:

✔ Expressions and statements

1. Expressions define how a value can be obtained by combining other values through operators. The values from which expresions are evaluated are either denoted by a literal, as in the case of the real value 57.73, or they are the r_value of a variable.
2. Operators appearing in an expression denote mathematical functions. They are characterized by their aritiy (i.e., number of operands) and are invoked using the function’s signature. A unary operator is applied to only one operand. A binary operator is applied to two operands. In general, a n-ary operator is applied to n operands.
3. For example, ’-’ can be used as a unary operator to transform–say–the value of a positive expression into a negative value. In general, however, it is used as a binary operator to subtract the value of one expression from the value of another expression. Functional routine invocations can be viewed as n-ary operators, where n is the number of parameters.
4. Regarding the operator’s notation, one can distinguish between infix, prefix, and postfix. Infix notation is the most common notation for binary operators: the operator is written between its two operands, as in x + y .

5. Postfix and prefix notations are common especially for non-binary operators. In prefix notation, the operator appears first, and then the operands follow. This is the conventional form of function invocation, where the function name denotes the operator.
6. In postfix notation the operands are followed by the corresponding operator. Assuming that the arity of each operator is fixed and known, expressions in prefix and postfix forms may be written without resorting to parentheses to specify subexpressions that are to be evaluated first. For example, the infix expression

   a * (b + c)

   can be written in prefix form as

   * a + b c

   and in postfix form as

   a b c + *
7. In C, the increment and decrement unary operators ++ and -- can be written both in prefix and in postfix notation. The semantics of the two forms, however, is different; that is, they denote two distinct operators.
8. Both expressions ++k and k++ have the side effect that the stored value of k is incremented by one. In the former case, the value of the expression is the value of k incremented by one (i.e., first, the stored value of k is incremented, and then the value of k is provided as the value of the expression).
9. In the latter case, the value of the expression is the value of k before being incremented. Infix notation is the most natural one to use for binary operators, since it allows programs to be written as conventional mathematical expressions.
10. Although the programmer may use parentheses to explicitly group subexpressions that must be evaluated first, programming languages complicate matters by introducing their own conventions for operator associativity and precedence.
11. Indeed, this is done to facilitate the programmer's task of writing expressions by reducing redundancy, but often this can generate confusion and make expressions less understandable, especially when switching languages. For example, the convention adopted by most languages is such that

    a + b * c

    is interpreted implicitly as

a + (b * c)

i.e., multiplication has precedence over binary addition (as in standard mathematics). However, consider the Pascal expression

a = b < c

and the C expression

a == b < c

12. In Pascal, operators < and = have the same precedence, and the language specifies that application of operators with the same precedence proceeds left to right. The meaning of the above expression is that the result of the equality test (a=b), which is a boolean value, is compared with the value of c (which must be a boolean variable).
13. In Pascal, FALSE is assumed to be less than TRUE , so the expression yields TRUE only if a is not equal to b, and c is TRUE ; it yelds FALSE in all other cases. For example, if a, b and c are all FALSE , the expression yields FALSE .
14. In C, operator "less than" (<) has higher precedence than "equal" (==). Thus, first b < c is evaluated. Such partial result is then compared for equality with the value of a . For example, assuming a = b = c = false (represented in C as zero), the evaluation of the expression yields 1, which in C stands for true.
15. Some languages, like C++ and Ada, allow operators to be programmer defined. For example, having defined a new type Set , one can define the operators + for set union and - for set difference.
16. The ability of providing programmer-defined operators, as any other feature that is based on overloading, can in some cases make programs easier to read, and in other cases harder. Readability is improved since the programmer is allowed to use familiar standard operators and the infix notation also for newly defined types.
17. The effect of this feature, however, is such that several actions happen behind the scenes when the program is processed. This is good whenever what happens behind the scenes matches the programmer's intuition; it is bad whenever the effects are obscure or counterintuitive to the programmer.
18. Some programming languages support the ability of writing conditional expressions, i.e., expressions that are composed of subexpressions, of which only one is to be evaluated, depending on the value of a condition. For example, in C one can write

(a > b) ? a : b

which would be written in a perhaps more conventionally understandable form in ML as

if a > b then a else b

to yield the maximum of the values of a and b .

19. ML allows for more general conditional expressions to be written using the "case" constructor, as shown by the following simple example.

    case x of

    1 => f1 (y)

    | 2 => f2 (y)

    | _ => g (y)

20. In the example, the value yielded by the expression is f1 (y) if x = 1 , f2 (y) if x =2 , g (y) otherwise.
21. Functional programming languages are based heavily on expressions. In such languages, a program is itself an expression, defined by a function applied to operands, which may themselves be defined by functions applied to operands.
22. Conventional languages, instead, make the values of expressions visible as a modification of the computation's state, through assignment of expressions to variables. An assignment statement, like x = y + z in C, changes the state by associating a new r_value with x , computed as y + z . To evaluate the expression, the r_values of variables y and z are used.
23. The result of the expression (an r_value) is then assigned to a memory location by using the l_value of x . Since the assignment changes the state of the computation, the statement that executes next operates in the new state.
24. Often, the next statement to be executed is the one that textually follows the one that just completed its execution. This is the case of a sequence of statements, which is represented in C as

    statement_1;

    statement_2;

    . . .

    statement_n;

25. The sequence can be made into a compound statement by enclosing it between the pair of brackets { and }. In other languages, like Pascal and Ada, the keywords begin and end are used instead of brackets.
26. In many conventional programming languages, like Pascal, the distinction between assignment statements and expressions is sharp. In others, like C, an assignment statement is actually an expression with a side-effect.
27. The value returned by an assignment statement is the one that is stored in the left operand of the assignment operator "=". A typical example is given by the following loop which reads successive input characters until the end of file is encountered:

    ```
    while ((c = getchar ( )) != EOF)
    /* assigns the character read to c and yields the read value, which is compared to the end of file symbol */
    . . .
    ```
28. Furthermore, in C the assignment operator associates from right to left. That is, the statement

    ```
    a = b = c = 0;
    ```

    is interpreted as

    ```
    a = (b = (c = 0))
    ```
29. Many programing languages, like Pascal, require the left-hand side of an assignment operator to be a simple denotation for an l_value.
30. For example, it can be a variable name, or an array element, or the cell pointed by some variable. More generally, other languages, like C, allow any expression yielding a modifiable l_value to appear on the left-hand side. Thus, it is possible to write the following kind of statement

    ```
    ( p > q) ? p* : q* = 0;
    ```

    which sets to zero the element pointed by the maximum of p and q .

✔ Conditional execution and iteration:

1. Conditional execution of different statements can be specified in most languages by the if statement. Languages differ in several important syntactic details concerning the way such a construct is offered.

2. Semantically, however, they are all alike.Let us start with the example of the if statement as originally provided by Algol 60. Two forms are possible, as shown by the following examples:

   ```
   if i = 0
   then i := j;
   ```

   ```
   if i = 0
   then i := j
   else begin
   i := i + 1;
   j := j - 1
   end
   ```

3. In the first case, no alternative is specified for the case i ¦ 0 , and thus nothing happens if i ¦ 0 . In the latter, two alternatives are present. Since the case where i ¦ 0 is described by a sequence, it must be made into a compound statement by bracketing it between begin and end .
4. The selection statement of Algol 60 raises a well-known ambiguity problem, illustrated by the following example

   if x > 0 then if x < 10 then x := 0 else x := 1000
5. It is unclear if the else alternative is part of the innermost conditional (if x < 10. .) or the outermost conditional (if x > 0 . . .).
6. The execution of the above statement with x = 15 would assign 1000 to x under one interpretation, but leave it unchanged under the other. To eliminate ambiguity, Algol 60 requires an unconditional statement in the then branch of an if statement. Thus the above statement must be replaced by either

   if x > 0 then begin if x < 10 then x := 0 else x := 1000 end

   or

   if x > 0 then begin if x < 10 then x := 0 end else x := 1000
7. The same problem is solved in C and Pascal by automatically matching an else branch to the closest conditional without an else.

8. Even though this rule removes ambiguity, however, nested if statements are difficult to read, especially if the program is written without careful indentation (as shown above). A syntactic variation that avoids this problem is adopted by Algol 68, Ada, and Modula-2, which use a special keyword as an enclosing final bracket of the if statement (fi in the case of Algol 68, end if in the case of Ada, end in the case of Modula-2). Thus, the above examples would be coded in Modula-2 as

   ```
   if i = 0
   then i := j
   else i := i + 1;
   j := j - 1
   end
   ```

   and

   ```
   if x > 0 then if x < 10 then x := 0 else x := 1000 end end
   ```

   or

   ```
   if x > 0 then if x < 10 then x := 0 end else x := 1000 end
   ```

   depending on the desired interpretation.
9. Choosing among more than two alternatives using only if-then-else statements may lead to awkward constructions, such as

   ```
   if a
   then S1
   else
   if b
   then S2
   else
   if c
   then S3
   else S4
   end
   end
   end
   ```

10. To solve this syntactic inconvenience, Modula-2 has an else-if construct that also serves as an end bracket for the previous if. Thus the above fragment may be written as

```
if a
then S1
else if b
then S2
else if c
then S3
else S4
end
```

11. C, Algol 68, and Ada provide similar abbreviations. Most languages also provide an ad-hoc construct to express multiple-choice selection. For example, C++ provides the switch construct, illustrated by the following fragment:

```
switch (operator) {
case ’+’:
result = operand1 + operand2;
break;
case ’*’:
result = operand1 * operand2;
break;
case ’-’:
result = operand1 - operand2;
break;
case ’/’:
result = operand1 / operand2;
break;
default:
break; --do nothing
};
```

12. Each branch is labelled by one (or more) constant values. Based on the value of the switch expression, the branch labelled by the same valuenate each branch; otherwise

execution would fall into the next branch. The same example may be written in Ada as

case OPERATOR is

when ’+’ => result = operand1 + operand2;

when ’*’ => result = operand1 * operand2;

when ’-’ => result = operand1 - operand2;

when ’/’ => result = operand1 / operand2;

when others => null;

end case

13. In Ada, after the selected branch is executed, the entire case statement terminates. Iteration allows a number of actions to be executed repeatedly.
14. Most programming languages provide different kinds of loop constructs to define iteration of actions (called the loop body). Often, they distinguish between loops where the number of repetitions is known at the start of the loop, and loops where the body is executed repeatedly as long as a condition is met.
15. The former kind of loop is usually called a for loop; the latter is often called the while loop. For-loops are named after a common statement provided by languages of the Algol family. For statements define a control variable which assumes all values of a given predefined sequence, one after the other. For each value, the loop body is executed.
16. Pascal allows iterations where control variables can be of any ordinal type: integer, boolean, character, enumeration, or subranges of them. A loop has the following general appearance:

    for loop_ctr_var := lower_bound to upper_bound do statement A control variable assumes all of its values from the lower to the upper bound. The language prescribes that the control variable and its lower and upper bounds must not be altered in the loop.
17. The value of the control variable is also assumed to be undefined outside the loop. is selected. If the value of the switch expression does not match any of the labels, the (optional) default branch is executed. If the default branch is not present, no action takes place.
18. The order in which the branches appear in the text isnate each branch; otherwise execution would fall into the next branch. The same example may be written in Ada as

    case OPERATOR is

```
when '+' => result = operand1 + operand2;
when '*' => result = operand1 * operand2;
when '-' => result = operand1 - operand2;
when '/' => result = operand1 / operand2;
when others => null;
end case
```

19. In Ada, after the selected branch is executed, the entire case statement terminates. Iteration allows a number of actions to be executed repeatedly. Most programming languages provide different kinds of loop constructs to define iteration of actions (called the loop body).
20. Often, they distinguish between loops where the number of repetitions is known at the start of the loop, and loops where the body is executed repeatedly as long as a condition is met. The former kind of loop is usually called a for loop; the latter is often called the while loop.
21. For-loops are named after a common statement provided by languages of the Algol family. For statements define a control variable which assumes all values of a given predefined sequence, one after the other. For each value, the loop body is executed.
22. Pascal allows iterations where control variables can be of any ordinal type: integer, boolean, character, enumeration, or subranges of them. A loop has the following general appearance:

    for loop_ctr_var := lower_bound to upper_bound do statement
23. A control variable assumes all of its values from the lower to the upper bound. The language prescribes that the control variable and its lower and upper bounds must not be altered in the loop. The value of the control variable is also assumed to be undefined outside the loop. Immaterial.

✔ Routines

1. Routines are a program decomposition mechanism which allows programs to be broken into several units. Routine calls are control structures that govern the flow of control among program units.

2. The relationships among routines defined by calls are asymmetric: the caller transfers control to the callee by naming it explicitly. The callee transfers control back to the caller without naming it.
3. The unit to which control is transfered when a routine R terminates is always the one that was executing immediately before R. Routines are used to define abstract operations. Most modern languages allow such abstract operations to be defined recursively.
4. Moreover, many such languages allow generic operations to be defined. Most languages distinguish between two kinds of routines: procedures and functions. A procedure does not return a value: it is an abstract command which is called to cause some desired state change.
5. The state may change because the value of some parameters transmitted to the procedure gets modified, or because some nonlocal variables are updated by the procedure, or because some actions are performed on the external environment (e.g., reading or writing).
6. A function corresponds to its mathematical counterpart: its activation is supposed to return a value, which depends on the value of the transmitted parameters. Pascal provides both procedures and functions.
7. It allows formal parameters to be either by value or by reference. It also allows procedures and functions to be parameters, as shown by the following example of a procedure header:

   procedure example (var x: T; y: Q; function f (z: R): integer);
8. In the example, x is a by-reference parameter of type T ; y is a by-value parameter of type Q ; f is a function parameter which takes one by-value parameter z of type R and returns an integer. Ada provides both procedures and functions.
9. Parameter passing mode is specified in the header of an Ada routine as either in , out , or in out . If the mode is not specified, in is assumed by default. A formal in parameter is a constant which only permits reading of the value of the corresponding actual parameter.
10. A formal in out parameter is a variable and permits both reading and updating of the value of the associated actual parameter. A formal out parameter is a variable and permits updating of the value of the associated actual parameter.

11. In the implementation, parameters are passed either by copy or by reference. Except for cases that are explicitly stated in the language standard, it is left to the implementation to choose whether a parameter should be passed by reference or by copy.
12. In C all routines are functional, i.e., they return a value, unless the return type is void , which states explicitly that no value is returned. Parameters can only be passed by value. It is possible, however, to achive the effect of call by reference through the use of pointers. For example, the following routine

    ```
    void proc (int* x, int y);
    {
    *x = *x + y;
    }
    ```

    increments the object referenced by x by the value of y . If we call proc as follows

    ```
    proc (&a, b); /* &a means the address of a */
    ```

    x is initialized to point to a , and the routine increments a by the value of b .
13. C++ introduced a way of directly specifying call by reference. This frees the programmer from the lower level use of pointers to simulate call by reference. The previous example would be written in C++ as follows.

    ```
    void proc (int& x, int y);
    {
    x = x + y;
    }
    ```

14. proc (a, b); -- no address operator is needed in the call While Pascal only allows routines to be passed as parameters, C++ and Ada get closer to treating routines as first-class objects. For example, they provide pointers to routines, and allow pointers to be bound dynamically to different routines at run time.

✔ Style issues: side effects and aliasing:

1. Side effects are used principally to provide a method of communication among program units. Communication can be established through nonlocal variables. However, if the set of nonlocal variables used for this purpose is large and each unit has unrestricted access

to the set of nonlocal variables, the program becomes difficult to read, understand, and modify.

2. Each unit can potentially reference and update every variable in the nonlocal environment, perhaps in ways not intended for the variable. The problem is that once a global variable is used for communication, it is difficult to distinguish between desired and undesired side effects.
3. For example, if unit u1 calls u2 and u2 inadvertently modifies a nonlocal variable x used for communication between units u3 and u4 , the invocation of u2 produces an undesired side effect. Such errors are difficult to find and remove, because the symptoms are not easily traced to the cause of the error. (Note that a simple typing error could lead to this problem.) Another difficulty is that examination of the call instruction alone does not reveal the variables that can be affected by the call.
4. This reduces the readability of programs because, in general, the entire program must be scanned to understand the effect of a call. Communication via unrestricted access to nonlocal variables is particularly dangerous when the program is large and composed of several units that have been developed independently by several programmers.
5. One way to reduce these difficulties is to use parameters as the only means of communication among units. The overhead caused by parameter passing is almost always tolerable, except for critical applications whose response times must be within severe bounds.
6. Alternatively, it must be possible to restrict the set of nonlocal variables held in common by two units to exactly those needed for the communication between the units. Also, it can be useful to specify that a unit can only read, but not modify some variable.
7. Side effects also are used in passing parameters by reference. In such a case, a side effect is used to modify the actual parameter. The programmer must be careful not to produce undesired side effects on actual parameters.
8. The same problem arises with call by name. A more substantial source of obscurity in call by name is that each assignment to the same formal parameter can affect different locations in the environment of the calling unit.
9. Such problems do not arise in call by copy. Languages that distinguish between functions and procedures suggest a programming style in which the use of side effects is restricted.

Side effects are an acceptable programming practice for procedures. Indeed, this should be the way a procedure sends results back to the caller.

10. Side effects, however, are unadvisable for function subprograms. In fact, function subprograms are invoked by writing the subprogram name within an expression, as in
v : = x+ f (x, y) + z
11. In the presence of side effects –in Pascal, for example–the call to f might produce a change to x or y (if they are passed by reference), or even z (if z is global to the function) as a side effect. This reduces the readability of the program, since a reader expects a function to behave like a mathematical function.
12. Also, one cannot rely on the commutativity of addition in general. In the example, if f modifies x as a side effect, the value produced for w is different if x is evaluated before or after calling f . Besides affecting readability, side effects can prevent the compiler from generating optimized code for the evaluation of certain expressions. In the example
u: = x+ z+ f (x, y) + f (x, y) + x+ z
13. the compiler cannot evaluate function f and subexpression x+ z just once. The recognition that side effects on parameters are undesirable for functions affected the design of Ada, which allows only in formal parameters for functions.
14. The disadvantages of aliasing affect programmers, readers, and language implementers. Subprograms can become hard to understand because, occasionally, different names denote the same data object.
15. This problem cannot be discovered by inspecting the subprogram: rather, discovery requires examining all the units that may invoke the subprogram. As a consequence of aliasing, a subprogram call may produce unexpected and incorrect results.

✔ Exceptions:

1. Programmers often write programs under the optimistic assumption that nothing will go wrong when the program executes. Unfortunately, however, there are many reasons which may invalidate this assumption.
2. For example, it may happen that under certain conditions an array is indexed with a value which exceeds the declared bounds. An arithmetic expression may cause a division by zero, or the square root operation may be executed with a negative argument.

3. A request for new memory allocation issued by the run-time system might exceed the amount of storage available for the program execution. Or, finally, an embedded application might receive a message from the field which overrides a previously received message, before this message has been handled by the program.
4. Often programs fail unexpectedly, maybe simply displaying some obscure message, as an erroneous program state is entered. This behavior, however, is unacceptable in many cases. To improve reliability, it is necessary that such erroneous conditions can be recognized by the program, and certain actions are executed in response to the error.
5. To do so, however, the conventional control structures we have discussed so far are simply inadequate. For example, to check that an index never exceeds the array bounds, one would need to explicitly test the value of the index before any indexing takes place, and insert appropriate response code in case the bounds are violated.
6. Alternatively, one would like the run-time machine to be able to trap such anomalous condition, and let the response to it be programmable in the language. This would be more efficient under the assumption that bound violations are the exceptional case. To cope with this problem, programming languages provide features for exception handling. According to the standard terminology, an exception denotes an undesirable, anomalous behavior which supposedly occurs rarely.
7. The language can provide facilities to define exceptions, recognize them, and specify the response code that must be executed when the exception is raised (exception handler). Exceptions have a wider meaning than merely computation errors. They refer to any kind of anomalous behavior that, intuitively and informally, corresponds to a deviation from the expected course of actions, as envisioned by the programmer.
8. The concept of "deviation" cannot be stated in absolute and rigorous terms. It represents a design decision taken by the programmer, who decides that certain states are "normal", and "expected", while others are "anomalous".
9. Thus, an exception does not necessarily mean that we are in the presence of a catastrophic error. It simply means that the unit being executed is unable to proceed in a manner that leads to its normal termination as specified by the programmer. For example, consider a control system which processes input messages defined by a given protocol.

✔ Exception handling in C++:

1. Exceptions may be generated by the run-time environment (e.g., due to a division by zero) or may be explicitly raised by the program. An exception is raised by a throw instruction, which transfers an object to the corresponding handler.
2. A handler may be attached to any piece of code (a block) which needs to be fault tolerant. To do so, the block must be prefixed by the keyword try . As an example, consider the following simple case:

```
class Help {. . . }; // objects of this class have a public attribute "kind" of type enumeration
                    // which describes the kind of help requested, and other public fields which
                    // carry specific information about the point in the program where help
                    // is requested
class Zerodivide { }; // assume that objects of this class are generated by the run-time system
. . .
try {
      // fault tolerant block of instructions which may raise help or zerodivide exceptions
. . .
}
catch (Help msg) {
            // handles a Help request brought by object msg
switch (msg.kind) {
case MSG1:
. . .;
case MSG2:
. . .;
. . .
}
. . .
```

```
}
catch (Zerodivide) {
// handles a zerodivide situation
. . .
}
```

3. Suppose that the above try block contains the statement throw Help (MSG1); A throw expression causes the execution of the block to be abandoned, and control to be transferred to the appropriate handler.
4. It also initializes a temporary object of the type of the operand of throw and uses the temporary to initialize the variable named in the handler. In the example, Help (MSG1) actually invokes the constructor of class Help passing a parameter which is used by the constructor to initialize field kind .
5. The temporary object so created is used to initialize the formal parameter msg of the matching catch , and control is then transferred to the first branch (case MSG1) of the switch in the first handler attached to the block. The above block might call routines which, in turn may raise exceptions.
6. If one such routine raises a–say–help request and does not provide a handler for it, the routine’s execution is abandoned and the exception is propagated to the point of call within the block. Execution of the block, in turn, is abandoned, and control is transferred to the handler as in the previous case.
7. In other terms, C++, like Ada, propagates unhandled exceptions. Like Ada, a caught exception can be propagated explicitly, by simply saying throw . Also, as in Ada, after a handler is executed, execution continues from the statement that follows the one to which the matched handler is attached.
8. Unlike Ada, any amount of information can flow along with an exception. To raise an exception, in fact, one can throw an object, which contains data that can be used by the handler.
9. For example, in the previous example, a help request was signalled by providing an object which contained specific information on the kind of help requested. If the data in the thrown object are not used by the handler, the catch statement can simply specify a type, without naming an object. This happens in our example for the division by zero.