
RGPVNOTES.IN

Subject Name: **Principles of Programming Languages**

Subject Code: **IT-5002**

Semester: **5th**

## Unit 2: Record Types

### *Introduction*

A data type defines a collection of **data objects** and a set of **predefined operations** on those objects.

Computer programs produce results by manipulating data.

ALGOL 68 provided a few basic types and a few flexible structure-defining operators that allow a programmer to design a data structure for each need.

A **descriptor** is the collection of the attributes of a variable. In an implementation a descriptor is a collection of memory cells that store variable attributes. $f$ If the attributes are static, descriptor are required only at compile time. They are built by the compiler, usually as a part of the symbol table, and are used during

compilation. For dynamic attributes, part or all of the descriptor must be maintained during execution. Descriptors are used for type checking and by allocation and de-allocation operations.

***Primitive Data Types***

Those not defined in terms of other data types are called primitive data types.

The primitive data types of a language, along with one or more type constructors provide structured types.

**Numeric Types**

1. **Integer**

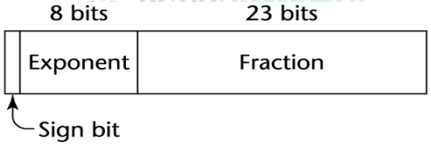
- Almost always an exact reflection of the hardware, so the mapping is trivial.
- There may be as many as eight different integer types in a language.
- Java has four: **byte**, **short**, **int**, and **long**.
- Integer types are supported by the hardware.

2. **Floating-point**

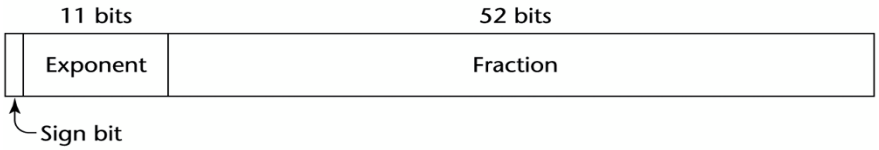
- Model real numbers, but only as approximations for most real values.
- On most computers, floating-point numbers are stored in binary, which exacerbates the problem.
- Another problem is the loss of accuracy through arithmetic operations.
- Languages for scientific use support at least two floating-point types; sometimes more (e.g. float, and double.)
- The collection of values that can be represented by a floating-point type is defined in terms of precision and range.

**Precision**: is the accuracy of the fractional part of a value, measured as the number of bits. Figure below shows single and double precision.

**Range**: is the range of fractions and exponents.




(a)




(b)

3. **Decimal**

- Most larger computers that are designed to support business applications have hardware support for decimal data types.
- Decimal types store a fixed number of decimal digits, with the decimal point at a fixed position in the value.
- These are the primary data types for business data processing and are therefore essential to COBOL.

**Advantage**: accuracy of decimal values.

**Disadvantages**: limited range since no exponents are allowed, and its representation wastes memory.

**Boolean Types**

- Introduced by ALGOL 60.
- They are used to represent switched and flags in programs.
- The use of Booleans enhances readability.

One popular exception is C89, in which **numeric** expressions are used as conditionals. In such expressions, all operands with **nonzero** values are considered **true**, and **zero** is considered **false**.

**Character Types**

- Char types are stored as numeric codings (ASCII / Unicode).
- Traditionally, the most commonly used coding was the **8-bit** code ASCII (American Standard Code for Information Interchange).
- A **16-bit** character set named Unicode has been developed as an alternative.

Java was the first widely used language to use the Unicode character set. Since then, it has found a way into JavaScript and C#.

***Character String Types***

A character string type is one in which values are sequences of characters.

**Important Design Issues**:

C and C++ use **char arrays** to store char strings and provide a collection of string operations through a standard library whose header is string.h

Ex: char *str = “apples”; // char ptr points at the str apples0

In this example, str is a char pointer set to point at the string of characters, apples0, where 0 is the null char.

**String Typical Operations**:

– Assignment
– Comparison (=, >, etc.)
– Catenation
– Substring reference
– Pattern matching
– Some of the most commonly used library functions for character strings in C and C++ are
  - strcpy: copy strings
  - strcat: catenates on given string onto another
  - strcmp: lexicographically compares (the order of their codes) two strings
  - strlen: returns the number of characters, not counting the null



In Java, strings are supported as a **primitive** type by **String** class

**String Length Options**

**Static Length String**: The length can be static and set when the string is created. This is the choice for the immutable objects of Java’s String class as well as similar classes in the C++ standard class library and the .NET class library available to C#.

**Limited Dynamic Length Strings**: allow strings to have varying length up to a declared and fixed maximum set by the variable’s definition, as exemplified by the strings in C.

**Dynamic Length Strings**: Allows strings various length with no maximum.

Requires the overhead of dynamic storage allocation and de-allocation but provides flexibility. Ex: Perl and JavaScript.

**Evaluation**

Aid to writability.

As a primitive type with static length, they are inexpensive to provide--why not have them?

– Dynamic length is nice, but is it worth the expense?

**Implementation of Character String Types**

**Static length:** Compile-time descriptor has three fields:

1. Name of the type
2. Type's length
3. Address of first char

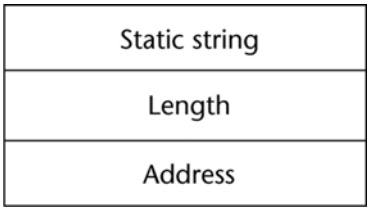



Fig: Compiler-time descriptor for static strings

**Limited dynamic length Strings** - may need a run-time descriptor for length to store both the fixed maximum length and the current length (but not in C and C++ because the end of a string is marked with the **null** character).

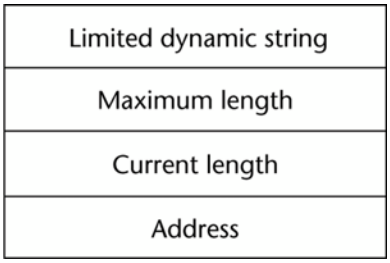



Fig: Run-time descriptor for limited dynamic strings



**Dynamic length Strings**–

- Need run-time descriptor because only current length needs to be stored.
- Allocation/deallocation is the biggest implementation problem. Storage to which it is bound must grow and shrink dynamically.
- There are two approaches to supporting allocation and deallocation:
  - Strings can be stored in a **linked list** "Complexity of string operations, pointer chasing"
  - Store strings in adjacent cells.

Find **a new area** of memory and the old part is moved to this area. Allocation and deallocation is **slower** but using adjacent cells results in faster string operations and requires less storage.

***User-Defined Ordinal Types***

An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers

Examples of **primitive** ordinal types in Java

- integer
- char
- boolean
- In some languages, users can define two kinds of ordinal types: **enumeration** and **subrange**.

**Enumeration Types**

All possible values, which are named constants, are provided in the definition

```
enum colors {red, blue, green, yellow, black};
colors myColor = blue, yourColor = red;
```

myColor++; // would assign green to myColor

The enumeration constants are typically implicitly assigned the integer values, 0, 1, …, but can be explicitly assigned any integer literal.
Java does not include an enumeration type, presumably because they can be represented as data classes. For example,

```
class colors {
public final int red = 0;
public final int blue = 1;
}
```

**Subrange Types**
An ordered contiguous subsequence of an ordinal type
Example: 12…18 is a subrange of integer type
**Ada's design**
type Days is (mon, tue, wed, thu, fri, sat, sun); subtype Weekdays is Days range mon...fri; subtype Index is Integer range 1..100;
Day1: Days; Day2: Weekday; Day2 := Day1;

**Array Types**
An array is an aggregate of **homogeneous** data elements in which an individual element is identified by its position in the aggregate, relative to the first element.
A reference to an array element in a program often includes one or more non- constant subscripts.
Such references require a run-time calculation to determine the memory location being referenced.



**Design Issues**

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does allocation take place?
- What is the maximum number of subscripts?
- Can array objects be initialized?

**Arrays and Indexes**

- Indexing is a mapping from indices to elements.
- The mapping can be shown as:

map(array_name, index_value_list) → an element
Ex: Ada
Sum := Sum + B(I);
Because **( )** are used for both subprogram parameters and array subscripts in Ada, this results in reduced readability.

C-based languages use [ ] to delimit array indices.
Two distinct types are involved in an array type:

- The element type, and
- The type of the subscripts.

The type of the subscript is often a sub-range of integers.
Ada allows other types as subscripts, such as Boolean, char, and enumeration.

Among contemporary languages, C, C++, Perl, and FORTRAN don't specify range checking of subscripts, but Java, and C# do.

**Subscript Bindings and Array Categories**

The binding of subscript type to an array variable is usually **static**, but the subscript value ranges are sometimes **dynamically bound**.

In C-based languages, the lower bound of all index ranges is fixed at **0**; **Fortran** 95, it defaults to **1**.

1. A **static array** is one in which the subscript ranges are statically bound and storage allocation is static (done before run time).

Advantages: efficiency "No allocation & deallocation."

Ex: Arrays declared in C & C++ function that includes the staticmodifier are static.

2. A **fixed stack-dynamic array** is one in which the subscript ranges are statically bound, but the allocation is done at elaboration time during execution.

Advantages: Space efficiency. A large array in one subprogram can use the same space as a large array in different subprograms.

Ex: Arrays declared in C & C++ function without the staticmodifier are fixed stack-dynamic arrays.

3. A **stack-dynamic array** is one in which the subscript ranges are dynamically bound, and the storage allocation is dynamic "during execution." Once bound they remain fixed during the lifetime of the variable

Advantages: Flexibility. The size of the array is not known until the array is about to be used.

Ex: Ada arrays can be stack dynamic:

```
Get (List_Len);
declare
List : array (1..List_Len) of Integer;
begin
. . . end;
```



The user inputs the number of desired elements for array **List**. The elements are then dynamically allocated when execution reaches the **declare** block. When execution reaches the end of the block, the array is deallocated.

4. A **fixed heap-dynamic array** is one in which the subscript ranges are dynamically bound, and the storage allocation is dynamic, but they are both fixed after storage is allocated.

The bindings are done when the user program requests them, rather than at elaboration time and the storage is allocated on the heap, rather than the stack.

Ex:

C & C++ also provide **fixed heap-dynamic arrays**. The function *malloc*and freeare used in C. The operations newand deleteare used in C++.

In Java all arrays are fixed heap dynamic arrays. Once created, they keep the same subscript ranges and storage.

5. A **heap-dynamic array** is one in which the subscript ranges are dynamically bound, and the storage allocation is dynamic, and can change any number of times during the array's lifetime.

Advantages: Flexibility. Arrays can grow and shrink during program execution as the need for space changes.

Ex: C# provides **heap-dynamic arrays** using an array class *ArrayList.*

```
ArrayList intList = new ArrayList( );
```

Elements are added to this object with the *Add* method, as in intArray.Add(nextOne);

Perl and JavaScript also support heap-dynamic arrays.

For example, in Perl we could create an array of five numbers with

@list = {1, 2, 4, 7, 10);
Later, the array could be lengthened with the push function, as in
push(@list, 13, 17);
Now the array value is (1, 2, 4, 7, 10, 13, 17 )

**Array Initialization**
Usually just a list of values that are put in the array in the order in which the array elements are stored in memory
FORTRAN uses the **DATA** statement, or put the values in **/ ... /** on the declaration.
Integer List (3)
Data List /0, 5, 5/ // List is initialized to the values
C and C++ - put the values in braces; let the compiler count them.
int stuff [] = {2, 4, 6, 8};
The compiler sets the length of the array.
Character Strings in C & C++ are implemented as arrays of **char**.
char name [ ] = "Freddie"; //how many elements in array name?
The array will have 8 elements because the null character is implicitly included by the compiler.
In Java, the syntax to define and initialize an array of references to String objects.
String [ ] names = ["Bob", "Jake", "Debbie"];
Ada positions for the values can be specified:
List : array (1..5) of Integer := (1, 3, 5, 7, 9);
Bunch : array (1..5) of Integer:= (1 => 3, 3 => 4, others => 0); Note: the array value is (3, 0, 4, 0, 0)

**Implementation of Arrays**

Access function maps subscript expressions to an address in the array
Access function for single-dimensioned arrays:
address(list[k]) = address (list[lower_bound]) + ((k-lower_bound) * element_size)

Accessing Multi-dimensioned Arrays
Two common ways:
- Row major order (by rows) – used in most languages
- column major order (by columns) – used in Fortran

For example, if the matrix had the values
3 4 7
6 2 5
1 3 8
It would be stored in row major order as:
3, 4, 7, 6, 2, 5, 1, 3, 8

If the example matrix above were stored in column major, it would have the following order in memory.
3, 6, 1, 4, 2, 3, 7, 5, 8

In all cases, sequential access to matrix elements will be faster if they are accessed in the order in which they are stored, because that will minimize the **paging**. (Paging is the movement of blocks of information between disk and main memory. The objective of paging is to keep the frequently needed parts of the program in memory and the rest on disk.)
Locating an Element in a Multi-dimensioned Array (row major)
Location (a[i,j]) = address of a [1,1 ] + ( (i - 1) * n + (j - 1) ) * element_size

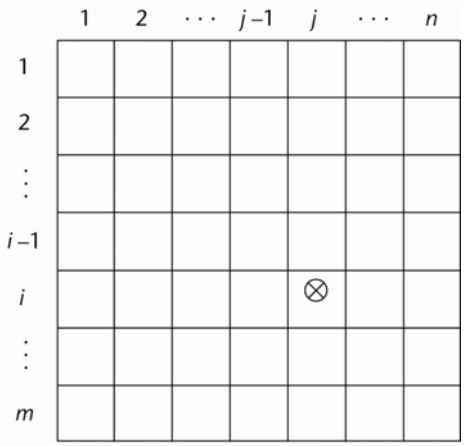



Fig: Multi Dimensional Array

**Associative Arrays**

An associative array is an unordered collection of data elements that are indexed by an equal number of values called **keys**.

So each element of an associative array is in fact a pair of entities, a key and a value. Associative arrays are supported by the standard class libraries of Java and C++
and Perl.

Example: In Perl, associative arrays are often called **hashes**. Names begin with %;
literals are delimited by parentheses

%temps = ("Mon" => 77, "Tue" => 79, “Wed” => 65);

Subscripting is done using braces and keys

$temps{"Wed"} = 83;

- Elements can be removed with delete delete $temps{"Tue"};
- Elements can be emptied by assigning the empty literal @temps = ( );

**Record Types**

A record is a possibly **heterogeneous** aggregate of data elements in which the individual elements are identified by names.

In C, C++, and C#, records are supported with the **struct** data type. In C++, structures are a minor variation on classes.

COBOL uses level numbers to show nested records; others use recursive definition

Definition of Records in COBOL

COBOL uses level numbers to show nested records; others use recursive definition

```
01 EMP-REC.
02 EMP-NAME.
05 FIRST PIC X(20).
05 MID   PIC X(10).
05 LAST  PIC X(20).
02 HOURLY-RATE PIC 99V99.
```

Definition of Records in Ada

Record structures are indicated in an orthogonal way

```
type Emp_Rec_Type is record
```

First: String (1..20); Mid: String (1..10); Last: String (1..20); Hourly_Rate: Float;
end record;
Emp_Rec: Emp_Rec_Type;

**References to Records**

Most language use dot notation
Emp_Rec.Name
Fully qualified references must include all record names
Elliptical references allow leaving out record names as long as the reference is unambiguous, for example in COBOL
FIRST, FIRST OF EMP-NAME, and FIRST of EMP-REC are elliptical references to the employee's first name

**Operations on Records**

- Assignment is very common if the types are identical
- Ada allows record comparison
- Ada records can be initialized with aggregate literals

COBOL provides MOVE CORRESPONDING
Copies a field of the source record to the corresponding field in the target record

**Unions Types**

A union is a type whose variables are allowed to store different type values at different times during execution.

**Discriminated vs. Free Unions**

**Fortran, C, and C++** provide union constructs in which there is no language support for type checking; the union in these languages is called *free union*
Type checking of unions require that each union include a type indicator called a *discriminanted union.*

**Ada Union Types**

type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
Filled: Boolean; Color: Colors; case Form is
when Circle => Diameter: Float;
when Triangle =>
Leftside, Rightside: Integer; Angle: Float;
when Rectangle => Side1, Side2: Integer;
end case;
end record;

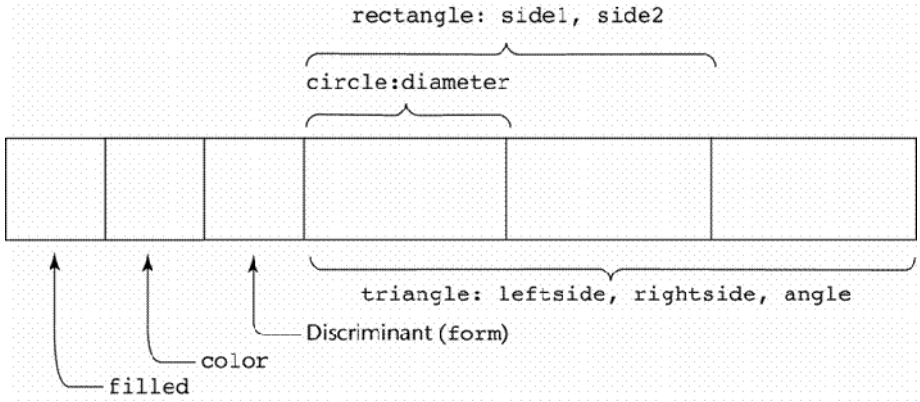



Fig: A discriminated union of three shape variables

**Evaluation of Unions**

Potentially unsafe construct

Java and C# do not support unions

Reflective of growing concerns for safety in programming language

***Pointers***

A pointer type in which the vars have a range of values that consists of **memory addresses** and a special value is nil. The value nil is not a valid address and is used to indicate that a pointer cannot currently be used to reference any memory cell.

**Pointer Operations**

A pointer type usually includes two fundamental pointer operations, assignment and dereferencing.

Assignment sets a pointer var's value to some useful address.

Dereferencing takes a reference through one level of indirection.

In C++, **dereferencing** is explicitly specified with the (*) as a prefix unary operation.

If *ptr* is a pointer var with the value 7080, and the cell whose address is 7080 has the value 206, then the assignment

j = *ptr //sets j to 206.

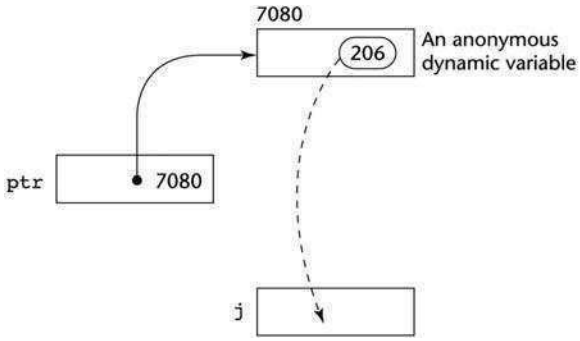



Fig: The assignment operation j = *ptr

**Pointer Problems**

1. **Dangling pointers** (**dangerous**)

A pointer points to a heap-dynamic variable that has been **deallocated**.

**Dangling pointers are dangerous for the following reasons**:

The location being pointed to may have been allocated to some new heap-dynamic var. If the new var is not the same type as the old one, type checks of uses of the dangling pointer are invalid.

Even if the new one is the same type, its new value will bear no relationship to the old pointer's dereference value.

If the dangling pointer is used to change the heap-dynamic variable, the value of the heap-dynamic variable will be destroyed.

It is possible that the location now is being temporarily used by the storage management system, possibly as a pointer in a chain of available blocks of storage, thereby allowing a change to the location to cause the storage manager to fail.

The following sequence of operations creates a dangling pointer in many languages:

- Pointer p1 is set to point at a new heap-dynamic variable.
- Set a second pointer p2 to the value of the first pointer p1.
- The heap-dynamic variable pointed to by p1 is explicitly deallocated (setting p1 to nil), but p2 is not changed by the operation. P2 is now a dangling pointer.

2. **Lost Heap-Dynamic Variables** (wasteful)

A heap-dynamic variable that is no longer referenced by any program pointer "no longer accessible by the user program."

Such variables are often called **garbage** because they are not useful for their original purpose, and also they can't be reallocated for some new use by the program.

Creating Lost Heap-Dynamic Variables:

- Pointer p1 is set to point to a newly created heap-dynamic variable.
- p1 is later set to point to another newly created heap-dynamic variable. c. The first heap-dynamic variable is now inaccessible, or lost.

The process of losing heap-dynamic variables is called **memory leakage**

**Pointers in Ada**

- Some dangling pointers are disallowed because dynamic objects can be **automatically** de-allocated at the end of pointer's type scope
- The lost heap-dynamic variable problem is not eliminated by Ada

**Pointers in Fortran 95**

- Pointers point to heap and non-heap variables
- Implicit dereferencing
- Pointers can only point to variables that have the TARGET attribute

The TARGET attribute is assigned in the declaration:

INTEGER, TARGET :: NODE



**Pointers in C and C++**

- Extremely flexible but must be used with care.
- Pointers can point at any variable regardless of when it was allocated
- Used for dynamic storage management and addressing
- Pointer arithmetic is possible in C and C++ makes their pointers more interesting than those of the other programming languages.
- Unlike the pointers of Ada, which can only point into the heap, C and C++ pointers can point at virtually any variable anywhere in memory.

Explicit dereferencing and address-of operators

In C and C++, the asterisk (*) denotes the dereferencing operation, and the ampersand (&) denotes the operator for producing the address of a variable. For example, in the code

```
int *ptr;
int count, init;
…
ptr = &init;
count = *ptr
```

The two assignment statement are equivalent to the single assignement

```
count = init;
```

Example: Pointer Arithmetic in C and C++

```
int list[10];
int *ptr;
ptr = list;
```

*(ptr+5) is equivalent to list[5] and ptr[5]
*(ptr+i) is equivalent to list[i] and ptr[i]

Domain type need not be fixed (**void ***)
void * can point to **any** type and can be type checked (cannot be de-referenced)

**Reference Types**
C++ includes a special kind of pointer type called a *reference type* that is used primarily for formal parameters in function definition.
A C++ reference type variable is a **constant** pointer that is always **implicitly** dereferenced.
Because a C++ reference type variable is a constant, it **must** be **initialized** with the address of some variable in its definition, and after initialization a reference type variable can **never** be set to reference any other variable.
Reference type variables are specified in definitions by preceding their names with ampersands (&). for example,

```
int result = 0;
int &ref_result = result;
…
ref_result = 100;
```

I this code segment, result and ref_result are **aliases**.
In Java, reference variables are extended from their C++ form to one that allow them to replace pointers entirely.
The fundamental difference between C++ pointers and Java references is that C++ pointers refer to memory addresses, whereas Java references refer to **class instances**.
Because Java class instances are implicitly deallocated (there is no explicit deallocation operator), there cannot be a dangling reference.
C# includes both the references of Java and the pointers of C++. However, the use of pointers is strongly discouraged. In fact, any method that uses pointers must include the unsafe modifier.
Pointers can point at any variable regardless of when it was allocated

**Variables**

- A program variable is an abstraction of a computer memory cell or collection of cells.
- A variable can be characterized as a sextuple of attributes: (name, address, value, type, lifetime, scope)

**Name**

- Variable names are the most common names in programs
- Names are often referred as identifiers.

**Concept of Binding**
In computing, a binding from a programming language to a library or operating system service is an application programming interface (API) providing glue code to use that library or service in a given programming language. Binding generally refers to a mapping of one thing to another. In the context of software libraries, bindings are wrapper libraries that bridge two programming languages, so that a library written for one language can be used in another language. Many software libraries are written in system programming languages such as C or C++. To use such libraries from another language, usually of higher-level, such as Java, Common Lisp, Python, or Lua, a binding to the library must be created in that language, possibly requiring recompiling the language's code, depending on the amount of modification needed

**Early Binding and Late binding**

The compiler performs a process called binding when an object is assigned to an object variable. The early binding (static binding) refers to compile time binding and late binding (dynamic binding) refers to runtime binding.

**Early Binding (Static binding)**

When perform Early Binding, an object is assigned to a variable declared to be of a specific object type. Early binding objects are basically a strong type objects or static type objects. While Early Binding, methods, functions and properties which are detected and checked during compile time and perform other optimizations before an application executes. The biggest advantage of using early binding is for performance and ease of development

**Late binding (Dynamic binding)**

By contrast, in Late binding functions, methods, variables and properties are detected and checked only at the run-time. It implies that the compiler does not know what kind of object or actual type of an object or which methods or properties an object contains until run time. The biggest advantages of Late binding is that the Objects of this type can hold references to any object, but lack many of the advantages of early-bound objects

**Strong Typing**

A strongly typed language is one in which each name in a program in the language has a single type associated with it, and that type is known as compile time. The essence of this definition is that all types are statically bound. The weakness of this definition is that it ignores the possibility that the storage location to which it is bound may store values of different types at different times. We define a programming language to be strongly typed if type errors are always detected.



**Type Compatibility**

*Compatibility* between types refers to the similarity of two types to each other. Type compatibility is important during type conversions and operations. All valid declarations in the same scope that refer to the same object or function must have compatible types. Two types are compatible if they fit any of the following categories:

- They are both real types.
- They are both integer types.
- One type is a subrange of the other.
- Both types are subranges of the same type.
- Both are set types with compatible base types.
- Both are packed-string types with the same number of characters.
- One is a string type and the other is a string, packed-string, or **Char** type.
- One type is Variant and the other is an integer, real, string, character, or Boolean type.
- Both are class, class-reference, or interface types, and one type is derived from the other.
- One type is **PAnsiChar** or **PWideChar** and the other is a zero-based character array of the form array[0..n] of **PAnsiChar** or **PWideChar**.
- One type is **Pointer** (an untyped pointer) and the other is any pointer type.
- Both types are (typed) pointers to the same type and the {$T+} compiler directive is in effect.
- Both are procedural types with the same result type, the same number of parameters, and type-identity between parameters in corresponding positions.
- Two types are compatible if they are the same.
- Two qualified types are compatible if they are identically qualified and the two types, unqualified, are compatible. The order of the qualifiers in the type declaration does not matter.

- The types short , signed short , short int , and signed short int are the same and are compatible.
- The types unsigned short and unsigned short int are the same and are compatible.
- The types int , signed , and signed int are the same and are compatible.
- The types unsigned and unsigned int are the same and are compatible.
- The types long , signed long , long int , signed long int are the same and are compatible.
- The types unsigned long and unsigned long int are the same and are compatible.
- Two array types are compatible if they are of the same size and contain elements of compatible types.
- If one array has an unknown size, it is compatible with all other array types having compatible element types.
- Two unions or structures are compatible if they are declared in different compilation units, share the same members in the same order, and whose members have the same widths (including bit fields).
- Two enumerations are compatible if all members have the same values. All enumerated types are compatible with other enumerated types. An enumerated type is also compatible with the signed int type.
- Two pointer types are compatible if they are identically qualified and point to objects of compatible types.
- A function type declared using the old-style declaration (such as int tree() ) is compatible with another function type if the return types are compatible.
- A function type declared using the new prototype- style declaration (such as int tree (int x) ) is compatible with another function type declared with a function prototype if:
- The return types are compatible.
- The parameters agree in number (including an ellipsis if one is used).
- The parameter types are compatible. For each parameter declared with a qualified type, its type for compatibility comparison is the unqualified version of the declared type.
- The function type of a prototype-style function declaration is compatible with the function type of an old-style function declaration if the return types are compatible, and if the old-style declaration is not a definition. Otherwise, the function type of a prototype- style function declaration is compatible with the function type of an old-style function definition if all of the following conditions are met:
- The return types of the two functions are compatible.
- The number of parameters agree
- The prototype-style function declaration does not contain an ellipsis as a parameter.
- The promoted types of the old-style parameters are compatible with the prototype-style parameter types. In the following example, the functions tree and tree2 are compatible. Tree and tree1 are not compatible, and tree1 and tree2 are not compatible.

```
int tree (int);
int tree1 (char);
int tree2 (x)
char x;   /* char promotes to int in old-style function parameters, and so is
compatible with tree          */
{  ...  };
```

- The following types, which may appear to be compatible, are not:
- unsigned int and int types are not compatible.
- char , signed char , and unsigned char types are not compatible.

**Composite Type**

A *composite type* is constructed from two compatible types and is compatible with both of the two types. Composite types satisfy the following conditions:

- If one type is an array of known size, the composite type is an array of that size.
- If only one type is a function type with a prototype, the composite type is a function type with the parameter type list.
- If both types are functions types with prototypes, the type of each parameter in the composite parameter type list is the composite type of the corresponding parameters.

**Named Constant:** In programming, a *constant* is like a variable, except that its value cannot change
There are two categories of constants:
**literal constants**, whose values are expressed literally;
int literal constants **EXAMPLES**: 5, 0, -127, +403298, -385092809
**named constants**, which have names.
In a named constant declaration, we indicate that it's a constant via the const *attribute*, and we MUST initialize it:
const float pi = 3.1415926;
**Advantage:** When you use named constants in the body of your program instead of literal constants, you isolate the constant values in the declaration section, making them trivial to find and to change
Programs become more self-documenting when special values are given meaningful names. This makes reading, debugging, and maintaining the program easier and faster.

**Variable Initialization** is the assignment of an initial value for a data object or variable. The manner in which initialization is performed depends on programming language, as well as type, storage class, etc., of an object to be initialized. Programming constructs which perform initialization are typically called initializers and initializer lists

**Default initialization**
Data initialization may occur without explicit syntax in a program to do so. For example, if static variables are declared without an initializer, then those of primitive data types are initialized with the value of zero of the corresponding type, while static objects of class type are initialized with their default constructors.

**Sequence Control With Expressions**
Sequence control refers to user actions and computer logic that initiate, interrupt, or terminate transactions. Sequence control governs the transition from one transaction to the next. General design objectives include consistency of control actions, minimized need for control actions, minimized memory load on the user, with flexibility of sequence control to adapt to different user needs. Methods of sequence control require explicit attention in interface design, and many published guidelines deal with this topic. The importance of good design for controlling user interaction with a computer system has been emphasized by Brown, Brown, Burkleo, Mangelsdorf, Olsen and Perkins. One of the critical determinants of user satisfaction and acceptance of a computer system is the extent to which the user feels in control of an interactive session. If users cannot control the direction and pace of the interaction sequence, they are likely to feel frustrated, intimidated, or threatened by the computer system. Their productivity may suffer, or they may avoid using the system at all. Complete user control of the interaction sequence and its pacing is not always possible, of course, particularly in applications where computer aids are used for monitoring and process control. The actions of an air traffic controller, for example, are necessarily paced in some degree by the job to be done. As a general principle, however, it is the user who should decide what needs doing and when to do it.

**Programming Control Structures**

You can write any program by using a combination of three control structures: (1) sequence (2) selection (3) repetition (a.k.a. iteration or looping) these three structures are the building blocks of all programs; they form the foundation of structured programming.

**The Sequence Control Structure**

The sequence control structure is the simplest of the three structures; it is a program segment where statements are executed in sequence, one after the other:

**1.Sequence control**

Sequence control : the control of the order of execution of the operations both primitive and user defined.

**Implicit:** determined by the order of the statements in the source program or by the built-in execution model

**Explicit:** the programmer uses statements to change the order of execution

(e.g. uses If statement)

**2. Levels of sequence control**

Expressions: computing expressions using precedence rules and parentheses.

Statements: sequential execution, conditional and iteration statements.

Declarative programming: an execution model that does not depend on the order of the statements in the source program.

Subprograms: transfer control from one program to another.

**3. Sequencing with expressions**

The issue: given a set of operations and an expression involving these operations,

- What is the sequence of performing the operations?
- How is the sequence defined, and how is it represented?
- An operation is defined in terms of an operator and operands.
- The number of operands determines the arity of the operator.

Basic sequence-control mechanism: functional composition

Given an operation with its operands, the operands may be:

- Constants
- Data objects
- Other operations

Example 1: 3 * (var1 + 5)

operation - multiplication, operator: *, arity - 2

operand 1: constant (3)

operand 2: operation addition

operand1: data object (var1)

operand 2: constant (5)

Functional compositions imposes a tree structure on the expression, where we have one main operation, decomposable into an operator and operands.

In a parenthesized expression the main operation is clearly indicated.

However we may have expressions without parentheses.

Example 2: 3* var1 +5

Question: is the example equivalent to the above one?

Example 3: 3 + var1 +5

Question: is this equivalent to (3 + var1) + 5, or to 3 + (var1 + 5) ?

In order to answer the questions we need to know:

- Operator's precedence
- Operator's associativity

Precedence concerns the order of applying operations, associativity deals with the order of operations of same precedence.

Precedence and associativity are defined when the language is defined - within the semantic rules for expressions.

**Arithmetic operations / expressions**

In arithmetic expressions the standard precedence and associativity of operations are applied to obtain the tree structure of the expression.

Linear representation of the expression tree:

- Prefix notation
- Postfix notation
- Infix notation

Prefix and postfix notations are parentheses-free.

There are algorithms to evaluate prefix and postfix expressions and algorithms to convert an infix expression into prefix/postfix notation, according to the operators' precedence and associativity.

**Other expressions**

Languages may have some specific operations, e.g. for processing arrays and vectors, built-in or user defined. Precedence and associativity still need to be defined - explicitly in the language definition or implicitly in the language implementation.

**Execution-time representation of expressions**

- Machine code sequence
- Tree structures - software simulation
- Prefix or postfix form - requires stack, executed by an interpreter.

**Evaluation of tree representation**

Eager evaluation - evaluate all operands before applying operators.

Lazy evaluation - first evaluate all operands and then apply operations

**Problems:**

- Side effects - some operations may change operands of other operations.
- Error conditions - may depend on the evaluation strategy (eager or lazy evaluation)
- Boolean expressions - results may differ depending on the evaluation strategy.

**Statement level sequence control**

**Forms of statement-level control**

- Composition – Statements are executed in the order they appear on the page.
- Alternation – Two sequences form alternatives so one sequence or the other sequence is executed but not both. (conditionals)
- Iteration – A sequence of statements that are executed repeatedly.
- Explicit Sequence Control

goto X

if Y goto X – transfer control to the statement labeled X if Y is true.

break

**Structured programming design**

- Hierarchical design of program structures
- Representation of hierarchical design directly in the program text using "structured" control statements.
- The textual sequence corresponds to the execution sequence
- Use of single-purpose groups of statements

**Structured control statements**

**Compound statements**

Typical syntax:

begin

statement1;

statement2;

... end;

Execute each statement in sequence.

Sometimes (e.g., C) { ... } used instead of begin ... end

**Conditional statements**

- if expression then statement1 else statement2
- if expression then statement1

If we need to make a choice among many alternatives

- nested if statements
- case statements

Example :

```
case Tag is
when 0 => begin
statement0
 end;
when 1 => begin
statement1
 end;
when 2 => begin
statement2
 end;
when others => begin
statement3
  end;
end case
```

Implementation: jump and branch machine instructions, jump table implementation for case statements k.

**Iteration statements**

Simple repetition (for loop) Specify a count of the number of times to execute a loop:

Examples:

perform statement K times;

for I=1 to 10 do statement;

for(I=0; I<10; I++) statement;

**Repetition while condition holds**

while expression do statement; - Evaluate expression and if true execute statement. then repeat process.

repeat statement until expression; - Execute statement and then evaluate expression. Quit if expression is true.

**Problems with structured sequence control:**

- Multiple exit loops
- Exceptional conditions

Solutions vary with languages, e.g. in C++ - break statement, assert for exceptions.

Conditional statements in C programming language are

- if statement
- if-else statement

- ternary statement or ternary operator
- nested if-else statement
- switch statement

**Usage**

Conditional statements cause variable flow of execution of the same program, each time the program is run, based on certain condition to be true or false! For example:

if Statement: Syntax

- if condition is true, then single statement or multiple statements inside the curly braces executes, otherwise skipped and control is transferred to statement following the if clause
- conditional expression can be exploited to be any expression, function call, literal constant, string functions, MACROs which results in some numerical value
- numerical values other than zero is considered true while zero is false

**Ternary Statement**

Ternary statement or Ternary operator is like if-else statement in its functioning. However, its syntax differs from if-else's syntax. For example:

```
z = a > b ? a : b;
```

- This is interpreted as: if a > b, then a is assigned to z else b is assigned to z
- Operator ">" greater than, is a Relational Operator.
- Other Relational Operators that can be used are <, >=, <=, ==, !=

**switch statement: Syntax**

expression in the switch must result in integer value.
Maximum one of the label values match the expression value in order for the corresponding statements to execute. Otherwise default statement executes if present in the switch statement.
break statement must be in the end in each case.
break causes the execution to jump out of the switch to the statement immediately following the switch.
break statement following the default clause is optional if default clause is placed as the LAST statement. However, break is must with default clause if the same is placed elsewhere within the switch!

**Loops**

**In do-while loop**, condition is always tested after the execution of the iteration and if condition fails, loop terminates. Therefore, do-while loop executes at least one iteration before termination if condition fails in first iteration

**In while loop,** condition is checked before execution of each iteration and as condition fails, loop terminates

**In while and for loops,** conditional expression is tested a priory the execution of each iteration while in the do-while loop, condition is tested after execution of each iteration. This means in do-while loop, first iteration always executes even if condition happens to be false in first iteration!

**Difference between for and while**

- In for loop, initialization, condition and adjustment statements are all put together in one line which make loop easier to understand and implement. While in the while loop, initialization is done prior to the beginning of the loop. Conditional statement is always put at the start of the loop. While adjustment can be either combined with condition or embedded into the body of the loop
- When using "continue;" statement in for loop, control transfers to adjustment statement while in while loop control transfers to the condition statement

**Exception Handling**

**An exception** is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,**and **throw**.

- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code,

**Throwing Exceptions**

Exceptions can be thrown anywhere within a code block using **throw** statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b) {
  if( b == 0 ) {
    throw "Division by zero condition!";
  }
  return (a/b);    }
```



**C++ Standard Exceptions**

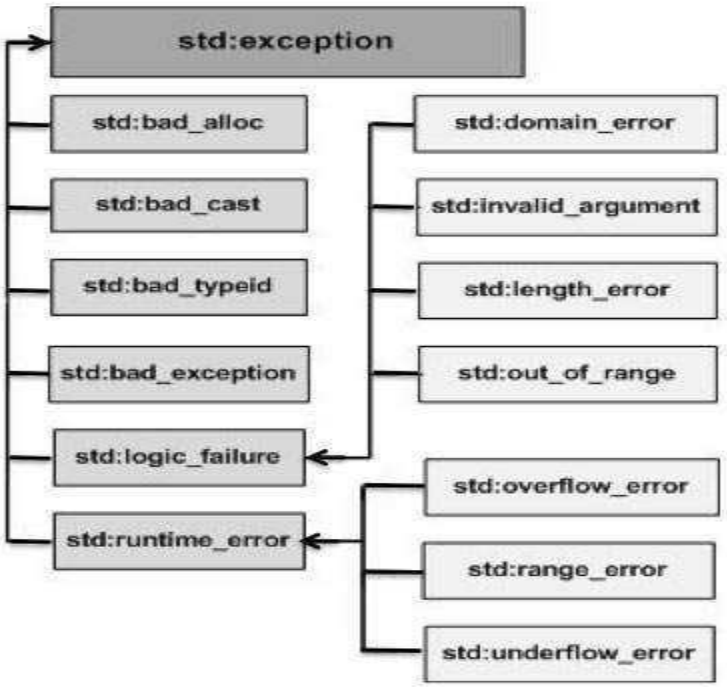



Fig: Standrad Exceptions in Java

| Exception | Description |
|---|---|

| | |
|---|---|
| **std::exception** | An exception and parent class of all the standard C++ exceptions. |
| std::bad_alloc | This can be thrown by **new**. |
| std::bad_cast | This can be thrown by **dynamic_cast**. |
| std::bad_exception | This is useful device to handle unexpected exceptions in a C++ program |
| std::bad_typeid | This can be thrown by **typeid**. |
| **std::logic_error** | An exception that theoretically can be detected by reading the code. |
| std::domain_error | This is an exception thrown when a mathematically invalid domain is used |
| std::invalid_argument | This is thrown due to invalid arguments. |
| std::length_error | This is thrown when a too big std::string is created |
| std::out_of_range | This can be thrown by the at method from for example a std::vector and std::bitset<>::operator[](). |
| **std::runtime_error** | An exception that theoretically can not be detected by reading the code. |
| std::overflow_error | This is thrown if a mathematical overflow occurs. |
| std::range_error | This is occured when you try to store a value which is out of range. |
| std::underflow_error | This is thrown if a mathematical underflow occurs. |

Table: Standrad Exceptions Discription

**Define New Exceptions**

You can define your own exceptions by inheriting and overriding **exception** class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way:

```
#include <iostream>   #include <exception>
using namespace std;
struct MyException : public exception {
  const char * what () const throw () {
    return "C++ Exception";
  } };
 int main() {
  try {       throw MyException();
  }catch(MyException& e) {
    std::cout << "MyException caught" << std::endl;
    std::cout << e.what() << std::endl;
  } catch(std::exception& e) {       //Other errors   } }
```


RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in