PPL II Mid Term -2018 Set-B Solution

Q.1 Type checking and type conversion

A **programming language** is called a **language** with static **type checking** or strongly typed **language**, if the **type** of each expression can be determined at compile-time, thereby guaranteeing that the **type**-related errors cannot occur in object program. Pascal is an example of a strongly typed **language**

Type checking means checking that each operation should receive proper number of arguments and of proper data type.

Like

A=B\*j+d;

\* and - are basically int and float data types based operations and if any variable in this A=B\*j+d;Is of other than int and float then compiler will generate type error.

Two ways of Type Checking:

1) Dynamic Type Checking:

• It is done at runtime.

• It uses concept of type tag which is stored in each data objects that indicates the data type of the object.

Example:

**An integer data object contains its'type' and 'values' attribute.**

**so Operation only be performed after type checking sequence in which type tag of each argument is checked. If the types are not correct then error will be generated.**

• Perl and Prolog follow basically dynamically type checking because data type of variables A+B in this case may be changed during program execution.

• so that type checking must be done at runtime.

## Advantages of Dynamic Type:

• It is much flexible in designing programs or we ca say that the flexibility in program design.

• In this no declarations are required.

• In this type may be changed during execution.

• In this programmerare free from most concern about data type.

## Disadvantage of Dynamic Type:

• 1) difficult to debug: We need to check program execution paths for testing and in dynamic type checking, program execution path for an operation is never checked.

• 2) extra storage: Dynamic type checking need extra storage to keep type information during execution.

• 3) Seldom hardware support : As hardware seldom support the dynamic type checking so we have to implement in software which reduces execution speed.

# Type checking: Static Type Checking:

Static Type Checking is done at complete time.

**Information needed at compile time is provided**- by declaration- by language structures.

The information required includes:

1) for each operation: The number, order, and data type, of its arguments.

2) For each variables: Name and data type of data object.

Example-

A+B

**in this type of A and B variables must not be changed.**

3) for each constant: Name and data type and value

const int x=28;

const float x=2.087;

**In this data type, the value and name is specified and in further if checked value assigned should match its data type.**

## Advantages of Static Type Checking:

1) compiler saves information:- if that type of data is according to the operation then compiler saves that information for checking later operations which further no need of compilation.

2) checked execution paths: As static type checking includes all operations that appear in any program statement, all possible execution paths are checked, and further testing for type error is not needed. So no type tag on data objects at run-time are not required, and no dynamic checking is needed.

## Disadvantages of Static Type Checking

: It affects many aspects of languages

1) declarations

2) data control structures

3) provision of compiling separately some subprograms.

# Strong Typing:

**If we change detect all types of errors statically in a program, we can say that language is' strongly typed'.**

It provides a level of security to our program.

Example

f:s-> R

**In this function f mail signature s generate output R and R is not outside the range of R data type.**

IF every operation is type safe then automatically language is strongly typed.

Example of strongly typed languages are:

C,Java, C++, RubyRail, smalltalk, python.

Type infer:- In this, like in ML, the language implementation will infer any missing type information from other declared type.

Example:

funarea(length:int, width:int):int= length \*width;

This is the standard declaration which tells length and width of int data type and its return type is int and function name area. But leaving any two of these declarations still leaves the function will only one interpretation. Knowing that \* can multiply together either two reals or two integers. ML interprets the following as equivalent to the previous example.

Funarea(length,width)int= length\*weight;

Funarea(length:int,width)= length\*weight;

Funarea(length,width:int)= length\*weight;

However:

Funarea(length,width)= length\*weight;

**Is invalid as it is now ambiguous as to that type of arguments. They could all be int or they could be real.**

Q.1.A **Type equivalence**

TYPE CHECKING RULES usually have the form

**if** two type expressions are equivalent

**then** return a given type

**else** return **type\_error**

KEY IDEAS. The central issue is then that we have to define when two given type expressions are equivalent.

* The main difficulty arises from the fact that most modern languages allow the naming of user-defined types.
* For instance, in C and C++ this is achieved by the typedef statement.
* When checking equivalence of named types, we have two possibilities.

**Name equivalence.**

Treat named types as basic types. Therefore two type expressions are *name equivalent* if and only if they are identical, that is if they can be represented by the same syntax tree, with the same labels.

**Structural equivalence.**

Replace the named types by their definitions and recursively check the substituted trees.

STRUCTURAL EQUIVALENCE. If type expressions are built from basic types and constructors (without type names, that is in our example, when using products instead of records), structural equivalence of types can be decided easily.

* For instance, to check whether the constructed types array(n1,T1) and array(n2,T2) are equivalent
  + we can check that the integer values n1 and n2 are equal and recursively check that T1 and T2 are equivalent,
  + or we can be less restrictive and check only that T1 and T2 are equivalent.
* Compilers use representations for type expressions (trees or dags) that allow type equivalence to be tested quickly.

RECURSIVE TYPES. In PASCAL a *linked list* is usually defined as follows.

type link = ^ cell;

cell = record

info: type;

next: link;

end;

The corresponding type graph has a cycle. So to decide structural equivalence of two types represented by graphs PASCAL compilers put a *mark* on each visited node (in order not to visit a node twice). In C, a *linked list* is usually defined as follows.

struct cell {

int info;

struct cell \*next;

};

To avoid cyclic graphs, C compilers

* require type names to be declared before they are used, except for pointers to records.
* use structural equivalence except for records for which they use name equivalence.

Q.1 structures are used to group together different types of variables under the same name. For example you could create a structure “telephone”: which is made up of a string (that is used to hold the name of the person) and an integer (that is used to hold the telephone number).  
Take a look at the example:

struct telephone

{

char \*name;

int number;

};

**Note:** the ; behind the last curly bracket.

With the declaration of the structure you have created a new type, called telephone. Before you can use the type telephone you have to create a variable of the type telephone. Take a look at the following example:

#include<stdio.h>

struct telephone

{

char \*name;

int number;

};

int main()

{

struct telephone index;

return 0;

}

**Note:** index is now a variable of the type telephone.

To access the members of the structure telephone, you must use a dot between the structure name and the variable name(variables:name or number.) Take a look at the next example:

#include<stdio.h>

struct telephone

{

char \*name;

int number;

};

int main()

{

struct telephone index;

index.name = "Jane Doe";

index.number = 12345;

printf("Name: %s\n", index.name);

printf("Telephone number: %d\n", index.number);

return 0;

}

## Type definitions and structures

Type definitions make it possible to create your own variable types. In the following example we will create a type definition called “intpointer” (a pointer to an integer):

#include<stdio.h>

typedef int \*int\_ptr;

int main()

{

int\_ptr myvar;

return 0;

}

It is also possible to use type definitions with structures. The name of the type definition of a structure is usually in uppercase letters. Take a look at the example:

#include<stdio.h>

typedef struct telephone

{

char \*name;

int number;

}TELEPHONE;

int main()

{

TELEPHONE index;

index.name = "Jane Doe";

index.number = 12345;

printf("Name: %s\n", index.name);

printf("Telephone number: %d\n", index.number);

return 0;

}

## Unions

A union is like a structure in which **all members** are stored at the **same** address. Members of a union can only be accessed one at a time. The union data type was invented to prevent memory fragmentation. The union data type prevents fragmentation by creating a standard size for certain data. Just like with structures, the members of unions can be accessed with the . and -> operators. Take a look at the example:

#include<stdio.h>

typedef union myunion

{

double PI;

int B;

}MYUNION;

int main()

{

MYUNION numbers;

numbers.PI = 3.14;

numbers.B = 50;

return 0;

}

Introduction



Subprograms are the fundamental building blocks of programs and are therefore among the most import concepts in programming language design.



The reuse results in several different kinds of savings, including memory space and coding time. Fundamentals of Subprograms

General Subprogram Characteristics

a. A subprogram has a single entry point.

b. The caller is suspended during execution of the called subprogram, which implies that there is only one subprogram in execution at any given time

.

c. Control always returns to the caller when the called subprogram’s execution terminates

Basic Definitions



A subprogram definition is a description of the actions of the subprogram abstraction. A subprogram call is an explicit request

that the called subprogram be executed. A subprogram is said to be

active if, after having been called, it has begun execution but has not yet completed that execution. The two fundamental types of the subprograms are:

oProcedures

oFunctions



A subprogram header is the first line of the definition, serves several definitions:

oIt specifies that the following syntactic unit is a subprogram definition of some particular kind.

oThe header provides a name for the subprogram.

oMay optionally specify a list of parameters.



Consider the following header examples:



Fortran

Subroutine Adder(parameters)



Ada

procedure Adder(parameters)



C

void Adder(parameters)



No special word appears in the header of a C subprogram to specify its kind.



The parameter profile (sometimes called the signature) of a subprogram is the

number, order, and types of its formal parameters.



The protocol of a subprogram is its parameter profile plus, if it is a function, its return type.



A subprogram declaration provides the protocol, but not the body, of the subprogram.



A formal parameter is a dummy variable listed in the subprogram header and used in the subprogram.



An actual parameter represents a value or address used in the subprogram call statement.



Function declarations are common in C and C ++ programs, where they are called prototypes

.



Java and C# do not need declarations of their methods, because there is no requirement that methods be defined before they are called in those languages. Parameters Subprograms typically describe computations. There are two ways that a non-local method program can gain access to the data that it is to process:

1. Through direct access to non-local variables.The only way the computation can proceed on different data is to assign new values to those non-local variables between calls to the subprograms.



Extensive access to non-locals can reduce reliability.

2. Through parameter passing “more flexible”. Data passed through parameters are accessed through names that are local to the subprogram. 

A subprogram with parameter access to the data it is to process is a

parameterized computation.



It can perform its computation on what ever data it receives through its parameters.



A formal parameter is a dummy variable listed in the subprogram header and used in the subprogram.



Subprograms call statements must include the name of the subprogram and a list of parameters to be bound to the formal

parameters of the subprogram.



An actual parameter represents a value or address used in the subprogram call statement.



Actual/Formal Parameter Correspondence:

1. Positional: The first actual parameter is bound to the first formal parameter and so forth. “Practical if list is short.”

2. Keyword: the name of the form al parameter is to be bound with the actual parameter. “Can appear in any order in the actual parameter list.”

SORT(LIST => A, LENGTH => N);



Advantage: order is irrelevant



Disadvantage: user must know the formal parameter’s names

.



Default Values:

procedure SORT(LIST : LIST\_TYPE;

LENGTH : INTEGER := 100);

...

SORT(LIST => A);



In C++, which has no keyword parameters, the rules for default parameters are necessarily different. 

The default parameters must appear last, for parameters are posit

ionally associated.



Once a default parameter is omitted in a call, all remaining formal parameters must have default values.

float compute\_pay(float income, float tax\_rate,

int exemptions =

1

)



An example call to the

C++

compute\_pay

function is:

pay = compute\_pay(20000.0, 0.15);

**Q.3 Exception handling** is the process of responding to the occurrence, during computation, of **exceptions** – anomalous or **exceptional** conditions requiring special processing – often changing the normal flow of program execution.

* **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.
* try {
* // protected code
* } catch( ExceptionName e1 ) {
* // catch block
* } catch( ExceptionName e2 ) {
* // catch block
* } catch( ExceptionName eN ) {
* // catch block
* }

## Throwing Exceptions

Exceptions can be thrown anywhere within a code block using **throw** statement. The operand of the throw statement 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);

}

## Catching Exceptions

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

try {

// protected code

} catch( ExceptionName e ) {

// code to handle ExceptionName exception

}

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows −

try {

// protected code

} catch(...) {

// code to handle any exception

}

The following is an example, which throws a division by zero exception and we catch it in catch block.

[Live Demo](http://tpcg.io/Nuo9hc)

#include <iostream>

using namespace std;

double division(int a, int b) {

if( b == 0 ) {

throw "Division by zero condition!";

}

return (a/b);

}

int main () {

int x = 50;

int y = 0;

double z = 0;

try {

z = division(x, y);

cout << z << endl;

} catch (const char\* msg) {

cerr << msg << endl;

}

return 0;

}

Because we are raising an exception of type **const char\***, so while catching this exception, we have to use const char\* in catch block. If we compile and run above code, this would produce the following result −

Division by zero condition!

## 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 −

[Live Demo](http://tpcg.io/FUdUJO)

#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

}

}

This would produce the following result −

MyException caught

C++ Exception

Q.4 **Scope** refers to the visibility of variables. In other words, which parts of your **program** can see or use it. Normally, every variable has a global **scope**. Once defined, every part of your **program** can access a variable.

Scope rules define the visibility rules for names in a programming language. What if you have references to a variable named k in different parts of the program? Do these refer to the same variable or to different ones?

Most languages, including Algol, Ada, C, Pascal, Scheme, and Haskell, are statically scoped. A block defines a new scope. Variables can be declared in that scope, and aren't visible from the outside. However, variables outside the scope -- in enclosing scopes -- are visible unless they are overridden. In Algol, Pascal, Haskell, and Scheme (but not C or Ada) these scope rules also apply to the names of functions and procedures.

Static scoping is also sometimes called lexical scoping.

## Simple Static Scoping Example

begin

integer m, n;

procedure hardy;

begin

print("in hardy -- n = ", n);

end;

procedure laurel(n: integer);

begin

print("in laurel -- m = ", m);

print("in laurel -- n = ", n);

hardy;

end;

m := 50;

n := 100;

print("in main program -- n = ", n);

laurel(1);

hardy;

end;

The output is:

in main program -- n = 100

in laurel -- m = 50

in laurel -- n = 1

in hardy -- n = 100 /\* note that here hardy is called from laurel \*/

in hardy -- n = 100 /\* here hardy is called from the main program \*/

Blocks can be nested an arbitrary number of levels deep.

## Dynamic Scoping

Dynamic scoping was used in early dialects of Lisp, and some older interpreted languages such as SNOBOL and APL. It is available as an option in Common Lisp. Using this scoping rule, we first look for a local definition of a variable. If it isn't found, we look up the calling stack for a definition. (See Lisp book.) If dynamic scoping were used, the output would be:

in main program -- n = 100

in laurel -- m = 50

in laurel -- n = 1

in hardy -- n = 1 ;; NOTE DIFFERENCE -- here hardy is called from laurel

in hardy -- n = 100 ;; here hardy is called from the main program

## Static Scoping with Nested Procedures

begin

integer m, n;

procedure laurel(n: integer);

begin

procedure hardy;

begin

print("in hardy -- n = ", n);

end;

print("in laurel -- m = ", m);

print("in laurel -- n = ", n);

hardy;

end;

m := 50;

n := 100;

print("in main program -- n = ", n);

laurel(1);

/\* can't call hardy from the main program any more \*/

end;

The output is:

in main program -- n = 100

in laurel -- m = 50

in laurel -- n = 1

in hardy -- n = 1

# Q.4 Parameter Passing Techniques

There are different ways in which parameter data can be passed into and out of methods and functions. Let us assume that a function B() is called from another function A(). In this case A is called the ***“caller function”*** and B is called the ***“called function or callee function”***. Also, the arguments which A sends to B are called actual arguments and the parameters of B are called formal arguments.

**Terminology**

* **Formal Parameter :** A variable and its type as they appear in the prototype of the function or method.
* **Actual Parameter :** The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.
* **Modes:**
  + **IN:** Passes info from caller to calle.
  + **OUT:** Callee writes values in caller.
  + **IN/OUT:** Caller tells callee value of variable, which may be updated by callee.

**Important methods of Parameter Passing**

1. **Pass By Value :** This method uses in-mode semantics. Changes made to formal parameter do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. This method is also called as **call by value**.

|  |
| --- |
| // C program to illustrate  // call by value  #include <stdio.h>    void func(int a, int b)  {      a += b;      printf("In func, a = %d b = %d\n", a, b);  }  int main(void)  {      int x = 5, y = 7;        // Passing parameters      func(x, y);      printf("In main, x = %d y = %d\n", x, y);      return 0;  } |

Output:

In func, a = 12 b = 7

In main, x = 5 y = 7

**Pass by reference(aliasing) :** This technique uses *in/out-mode* semantics. Changes made to formal parameter do get transmitted back to the caller through parameter passing. Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data. This method is also called as <**em>call by reference**. This method is efficient in both time and space.

|  |
| --- |
| // C program to illustrate  // call by reference  #include <stdio.h>    void swapnum(int\* i, int\* j)  {      int temp = \*i;      \*i = \*j;      \*j = temp;  }    int main(void)  {      int a = 10, b = 20;        // passing parameters      swapnum(&a, &b);        printf("a is %d and b is %d\n", a, b);      return 0;  } |

Output:

a is 20 and b is 10

**Q.4 Encapsulation** is a term that is found in Object-Oriented paradigm and refers to keeping the **data** in private fields and modify it only through methods. Thus **encapsulation** may be seen as a way of achieving **data hiding** in object-oriented systems. Creating a class includes the concept of **encapsulation**

| **Basis for comparison** | **Data Hiding** | **Encapsulation** |
| --- | --- | --- |
| Basic | Data hiding concern about data security along with hiding complexity. | Encapsulation concerns about wrapping data to hide the complexity of a system. |
| Focus | Data Hiding focuses on restricting or permitting the use of data inside the capsule. | Encapsulation focuses on enveloping or wrapping the complex data. |
| Access Specifier | The data under data hiding is always private and inaccessible. | The data under encapsulation may be private or public. |
| Process | Data hiding is a process as well as technique. | Encapsulation is a sub-process in data hiding. |

# Abstract Data Types

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.  
The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation independent view. The process of providing only the essentials and hiding the details is known as abstraction.  
The user of [data type](https://www.geeksforgeeks.org/data-types-in-c/) need not know that data type is implemented, for example, we have been using int, float, char data types only with the knowledge with values that can take and operations that can be performed on them without any idea of how these types are implemented. So a user only needs to know what a data type can do but not how it will do it. We can think of ADT as a black box which hides the inner structure and design of the data type. Now we’ll define three ADTs namely [List](https://www.geeksforgeeks.org/linked-list-set-1-introduction/) ADT, [Stack](https://www.geeksforgeeks.org/stack-data-structure-introduction-program/) ADT, [Queue](https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/) ADT.

**List ADT**  
A list contains elements of same type arranged in sequential order and following operations can be performed on the list.  
get() – Return an element from the list at any given position.  
insert() – Insert an element at any position of the list.  
remove() – Remove the first occurrence of any element from a non-empty list.  
removeAt() – Remove the element at a specified location from a non-empty list.  
replace() – Replace an element at any position by another element.  
size() – Return the number of elements in the list.  
isEmpty() – Return true if the list is empty, otherwise return false.  
isFull() – Return true if the list is full, otherwise return false.

**Stack ADT**  
A Stack contains elements of same type arranged in sequential order. All operations takes place at a single end that is top of the stack and following operations can be performed:  
push() – Insert an element at one end of the stack called top.  
pop() – Remove and return the element at the top of the stack, if it is not empty.  
peek() – Return the element at the top of the stack without removing it, if the stack is not empty.  
size() – Return the number of elements in the stack.  
isEmpty() – Return true if the stack is empty, otherwise return false.  
isFull() – Return true if the stack is full, otherwise return false.

**Queue ADT**  
A Queue contains elements of same type arranged in sequential order. Operations takes place at both ends, insertion is done at end and deletion is done at front. Following operations can be performed:  
enqueue() – Insert an element at the end of the queue.  
dequeue() – Remove and return the first element of queue, if the queue is not empty.  
peek() – Return the element of the queue without removing it, if the queue is not empty.  
size() – Return the number of elements in the queue.  
isEmpty() – Return true if the queue is empty, otherwise return false.  
isFull() – Return true if the queue is full, otherwise return false.

From these definitions, we can clearly see that the definitions do not specify how these ADTs will be represented and how the operations will be carried out. There can be different ways to implement an ADT, for example, the List ADT can be implemented using arrays, or singly linked list or doubly linked list. Similarly, stack ADT and Queue ADT can be implemented using arrays or linked lists.