Course: PARADIGMS AND COMPUTER PROGRAMMING FUNDAMENTALS (PCPF)

Course Instructor

Mrinmoyee Mukherjee B.E (Electronics), M.E (EXTC), PhD (Pursuing)

Assistant Professor

Department of Information Technology

St. Francis Institute of Technology

email: mrinmoyeemukherjee@sfit.ac.in

Academic Year: 2023-24 (Odd Semester)

Type in Programming

- Most high-level programming languages include a notion of type (data type) for expressions and/or objects. (Ch-7 from Scott)
- In general, a type is a collection of computational entities that share some common property.
- A type gives semantics to a piece of data.
- A type is a set of value. For the type integer, it will be a range of decimals. For the type string, it will be a range of possible strings.

1. Types **provide implicit context for many operations**, so the programmer does not have to specify that context explicitly.
   - Ex. In Pascal, for instance, the expression **z = a + b** will use integer addition if a and b are of integer type; or it will use floating-point addition if a and b are of real type.
2. Types **limit the set of operations** that may be performed in a semantically valid program.
   - Ex. They prevent the programmer from adding a character and a record, or
   - Ex. passing a file as a parameter to a subroutine that expects an integer.

Data Types	Sizes in byte	Sizes in bits	Range formula 2^n-1	Ranges
int	4 bytes	32bits	2^{32}-1	-2,147,483,648 to 2,147,483,647
unsigned int	4 bytes	32 bits	2^{32}-1	0 to 4294967295
float	4 bytes	32 bits	2^{32}-1(5 points)	3.4×10^{-38} to $3.4 \times 10^{+38}$
double	8 bytes	64 bits	2^{64}-1(15 points)	1.7×10^{-308} to $1.7 \times 10^{+308}$
long double	10 bytes	80 bits	2^{80}-1(19 points)	1.7×10^{-4932} to $1.7 \times 10^{+4932}$
char	1 byte	8 bits	2^{8}-1	0 to 255

Type in Programming

- There are three main uses of types in programming languages:
  - **naming** and **organizing** concepts,
  - making sure that bit sequences in computer memory are **interpreted consistently,**
  - **providing information to the compiler** about data manipulated by the program.
- Using types to organise a program **makes it easier** for someone to **read, understand, and maintain the program**.
- Types can serve an important purpose in **documenting the design and intent** of the program.
- Type information in programs can be **used for** many kinds of **optimisations**.

Example of Type Systems

- Ex of Type system

```
<?php
printf((3 + "Hello World!") . "\n");
// => PHP Warning: A non-numeric value encountered in /home/myusername/phpgoodies/test.php on line 3
// => 3
printf("The execution continue!");
// => The execution continue!
?>
```

- We are trying to add integer to string. When you violate the rules of a type system, the outcome can range between these two extremes:
  - The interpreter or compiler will silently try to fix the problem and continue.
  - The interpreter or compiler will throw an error and stop.

Type Checking

- If types push us to respect some rules, a programming language need an algorithm to check if we respect them. This is called type checking.
- In most statically typed languages, every definition of an object (constant, variable, subroutine, etc.) must specify the object's type
- A type error occurs when a computational entity (object) is used in a manner that is inconsistent with the concept it represents.
- Type checking is the process of ensuring that a program obeys the language's type compatibility rules.
- Type checking may occur at compile time (a static check) or at run-time.
- Type checking is used to prevent type errors, ensuring that the operations in a program are applied properly.
- A violation of the rules is known as a type clash.

Type Checking

- Strongly-typed Language:
  - If a language specification requires its typing rules strongly (i.e., more or less allowing only those automatic type conversions that do not lose information), one can refer to the process as strongly typed otherwise it is weakly typed.
  - It does not allow variables to be used in a way inconsistent with their type
  - If the compiler can guarantee that the accepted programs will execute without type errors.
  - Strong typing offers more safety, but cannot guarantee complete type safety.
- Statically-typed Language: A language is said to be statically typed if it is strongly typed and type checking can be performed at compile time.

Example: Type Safe

- Consider example

var x := 5;

var y := "37";

var z := x + y;

In this example z will be 42.

- Type safe language will convert y to integer and add to x
- But if var y := "Hello world"; program will crash
- Hence program of a language is both type-safe and memory-safe

Example: Type Safe

- Consider example in C

```
int x = 5;
char y[] = "37";
char* z = x + y;
printf("%c\n", *z);
```

- In this example z will point to a memory address five characters beyond y.
- This is memory that the program is not expected to access.
- In C terms this is simply undefined behaviour and the program may do anything; with a simple compiler it might actually print whatever byte is stored after the string "37".
- As this example shows, C is not memory-safe.
- As arbitrary data was assumed to be a character, it is also not a type-safe language.

Type Safety

- A programming language is type safe **if no program is allowed to violate its type distinctions.**
- A type system enables compiler to detect meaningless/ invalid code.
- E.g., we can identify an expression 3 / "Hello, World" as invalid.

Safety	Example language	Explanation
Not safe,	C, C++	Type casts pointer arithmetic
Almost safe	Pascal	Explicit deallocation;
Safe	LISP, SML, Smalltalk, Java	Type checking

Static and Dynamic Type Checking

- Run-time (Dynamic) type checking:
  - Dynamic type checking is the process of verifying the type safety of a program at runtime. Its slow process.
  - Implementations of dynamically type-checked languages generally associate each runtime object with a type tag (i.e., a reference to a type) containing its type information.
  - Examples: LISP, Smalltalk.
- Compile-time (Static) type checking:
  - The compiler checks the program text for potential type errors before run time. Its fast process.
  - Example: SML.
- Most programming languages use some combination of compile-time and run-time type checking because many useful features or properties are difficult or impossible to verify statically.

Type Compatibility

- A type checker for a statically-typed language must verify that the type of any expression is consistent with the type expected by the context in which that expression appears.
- For example, in an assignment statement of the form x := e, the inferred type of the expression e must be consistent with the declared type of the variable x.
- This notion of consistency, called compatibility, is specific to each programming language.

Type Equality

- The question of type equality arises during type checking.
- What does it mean for two types to be equal?
  - Structural equality.
  - Name equality

1. Structural equality.

- Two type expressions are structurally equal if and only if they are equivalent under the following three rules.
  - A type name is structurally equal to itself.
  - Two types are structurally equal if they are formed by applying the same type constructor to structurally equal types.
  - After a type declaration, say type n = T, the type name n is structurally equal to T.

Type Equality Example

```
typedef struct {
        int data[100];
        int count;
        } Stack;

  typedef struct {
        int data[100];
        int count;
        } Set;

  Stack x, y;
  Set r, s;
```

- x and y are name equivalent having same datatype, x=y is allowed
- r and s are name equivalent having same datatype, , r=s is allowed
- In structural equivalence two types are equal if, and only if, they have the same "structure", which can be interpreted in different ways.
- Here x and r have same structure
- C does not allow structural equivalence

THANK YOU