System Verilog Basics

Daniyal Tahsildar

Disclaimer:

Welcome to this educational document! Here, you'll find a comprehensive overview of various SystemVerilog concepts, accompanied by example code. Each topic is conveniently linked to its corresponding code repository on GitHub. Look for a symbol!

Please note that this document is designed solely for educational purposes, aiming to provide foundational knowledge of SystemVerilog concepts.

All code snippets included have been thoroughly tested using tools such as Mentor Questa 2021.3 and Synopsys VCS 2021.03, available on EDA Playground.

If you seek a deeper understanding of the topics covered in this document, please refer the SystemVerilog reference manual. This document primarily serves to provide a foundational understanding of these concepts.

Your feedback, input, and critique are highly valued! Feel free to connect with me on LinkedIn or via Email to share your thoughts.

Contents

Data types

A data type serves as a classification defining the kind of data a variable can hold and the operations that can be performed on that data. For instance, in SystemVerilog, data types such as logic, real, and string specify the nature of the information stored within variables.

1. Integral Data Types

Integral data types are used to represent integer numbers, and do not represent fractional parts of a number. Integral data types like logic and reg are synthesizable.

An unsigned data type exclusively stores positive values, even if assigned a negative value, it retains it as positive. Conversely, for a signed data type, the Most Significant Bit (MSB) indicates the sign; if it's a negative value, the MSB bit is set to 1.

The data types byte (8 bits), shortint (16 bits), int (32 bits), integer (32 bits), and longint (64 bits) default to signed. Data types time, bit, reg, and logic default to unsigned, as do arrays of these types. The default values of these data types can be changed using the signed and unsigned keywords.

For example: bit signed [2:0] a; int unsigned b;

Types that can have X and Z values are termed 4-state types, such as logic, reg, integer (distinct from int), and time. The default value for these data types is X, whereas for 2-state data types like int, shortint, longint, byte, and bit, it is 0.

2. Real Data Types

Real data types include real, shortreal and realtime. They are designed to handle floating-point numbers. The real data type resembles a double in C, occupying 64 bits, while shortreal is analogous to a float in C, occupying 32 bits. realtime is primarily utilized for representing time values in simulations.

3. String Data Type

Strings represent dynamically allocated collections of characters, with each character being byte-sized. When stored in memory, every character of the string is converted to a byte value (ASCII values), utilizing one byte for each ASCII character.

The size of a string can be adjusted during runtime, including increases, decreases, or deletions. SV offers several string manipulation methods.

len():

Returns the count of characters in a string.

```
string s1;
int count;
count = s1.len();
```

putc():

Substitutes a character in the string at the specified index passed as the putc() argument. Does not change the size of the string only replaces characters.

```
// string_name.putc(index-position, character)
    s1.putc(8, "a");
```

getc():

Retrieves a character (ASCII value) from the string at the specified index provided as the getc() argument.

```
// string_name.getc(index-position)
    int return_v;
    return_v = s1.getc(5);
```

toupper():

Converts all characters in a string to uppercase and returns it.

```
s2 = s1.toupper();
```

tolower():

Converts all characters in a string to lowercase and returns it.

```
s2 = s1.tolower();
```

compare():

Compares two strings. Returns 0 since both the strings are identical or the difference in ASCII values if strings are different.

```
// string_1.compare(string_2)
    int compare_v;
    compare_v = s2.compare(s1);
```

icompare():

Performs a case-insensitive comparison of two strings, disregarding upper and lower cases.

```
compare_v = s2.icompare(s3);
```

Additionally, the following functions are used to convert ASCII characters to numbers (integers), which proves useful when conducting file handling operations with characters as numbers in the file.

atoreal():

Converts ASCII to real and returns values.

```
real return_v_real;
s1 = "1234.5";
return_v_real = s1.atoreal();
```

atoi():

Converts ASCII to integer and returns values.

```
return_v = s1.atoi();
```

itoa():

Converts integer to ASCII values and stores in the string.

```
s1.itoa(2345.6);
```

4. Enum Datatype

SV introduces a concept called enum enabling users to represent numbers in words, enhancing code readability without altering functionality. Instead of numerical values, words can be used.

Syntax: enum {enum_word_1 = value, enum_word_2 = value, …….} enum_name

The *value* parameter is optional and if not specified, it starts from 0 for *enum_word_1*, 1 for *enum_word_2* and so forth.

For example:

```
enum bit [1:0] { SMALL, MEDIUM, LARGE}size_t1;
```

In this example, default values for SMALL is 0, MEDIUM is 1, and LARGE is 2.

Specific values can be assigned as follows:

```
enum bit [1:0] {
    SMALL = 2'b11,
    MEDIUM = 2'b01,
    LARGE = 2'b10
} size_t2;
```

The values of the enum type are derived from the value of the first element. If the first *enum_word* value is 4, subsequent *enum_word* values will increment accordingly if not specified.

For example:

```
// enum without a bounded size
    enum  {
        SMALL = 4,
        MEDIUM,  // value is 5
        LARGE    // value is 6
    } size_t3;
```

5. User Defined Data Types

There are situations where SV provided data types may not meet specific requirements. In such cases, user-defined data types are required.

The typedef keyword is used to create new data types using existing ones.

The syntax is: typedef data_type data_name.

Example:

```
// creating a new data type of 6 bits.
typedef bit [5:0] bit_type;
bit_type bit_v;
// creating enum type
typedef enum logic [1:0] {
    SMALL, MEDIUM, LARGE
} enum_type;
enum_type size_v;
```

typedef can also be used to create an array of class data type:

```
class sample_pkt;
    ..............
    ..............
endclass
// typedef class_name class_array[size];
typedef sample_pkt pktFA_t[5];
pktFA_t pktFA;
```

Here, *class_array* stores data of the *class_name* type, enabling the storage of multiple instances of a class within an array. This approach proves beneficial in scoreboarding logic, where multiple data transactions need to be compared. Instead of storing class data in separate arrays, a single array of class type can be utilized, where different elements can be accessed through array indexing.

6. Structure (struct) Data Type

There can be requirements where a variable is needed to represent various data types grouped into a single entity. To address this requirement, SV provides a data type known as struct.

A struct is a collection of different data types into a single entity. encapsulated within a single entity. It can be referenced as a whole, or its individual data elements can be referenced by name. struct supports all common array methods, including *locator, ordering*, and *reduction*. By default, structs are unpacked types and can contain any data type.

The syntax for defining a struct is: struct {data_elements} struct_name;

It can also be used with typedef.

For example:

```
typedef struct {
    string name;
    int count;
    bit [7:0] data;
} data_set;
```

This defines a struct named *data_set* consisting of three elements. Now, this struct data type can be utilized to create an array: `data_set data_setQ[$];`

Here, *data_setQ* collectively refers to a queue of *data_set* instances, with each *data_set* containing three elements.

While classes serve a similar purpose, they are dynamic in nature, requiring memory allocation. In contrast, structs are static and lack *tasks, functions*, or *constraints*.

Accessing struct fields is done using the '.' operator, such as *struct_name.field_name*.

For example:

```
data_set.count; data_set.name;
```

7. Void Data Type

Void is used to refer to non-existent data. It is particularly useful for functions that do not return a value.

In SV, a function prototype follows this structure:

```
function return_data_type function_name(arguments);
```

When a function is intended to perform a task without returning any data, the *return_data_type* is set as void. This ensures that the function does not produce any output data.

```
// takes an int value as input
function void print_all(int count);
    $display("Printing sample_pkt fields %0d", count);
    $display("size = %0d", size);
    $display("data = %0d\n", data);
endfunction
..................
..................
sample_pkt pkt;
..................
..................
pkt.print_all(3); // function has no return value
```

Operators

The collection of operators in SV includes all Verilog operators and extends to additional ones. SV operators streamline implementation, requiring less effort, time, and code, while enhancing code readability. This documentation primarily highlights SV-specific operators and excludes discussions on many conventional Verilog operators such as assignment (=, +=, <<=, etc.), conditional (?), increment/decrement (++, --), and others.

1. Concatenation Operator

Concatenation refers to the process of combining bits from one or more expressions into a single entity. In SV, concatenation supports the merging of *bit streams, integers, strings*, and *arrays* (with only one element of the array usable, not the entire array).

It is required to use variables with bounded sizes in concatenations. This ensures that the tool can calculate the complete size of the concatenation accurately. If unbounded size variables are used in concatenations, the tool defaults to the size of the left-hand side (LHS) variable, potentially overwriting existing elements within the concatenation.

For example:

```
logic [27:0] data_1;
logic [3:0] data_2 = ;
data_1 = {8'd11, 16'h13, 5}; // Incorrect
data_1 = {8'd11, 16'd13, 4'd5}; // Correct
data_1 = {8'd11, 16'd13, data_2}; // Possible
```

Concatenation can also be utilized on the left-hand side of an assignment or within an expression.

For example:

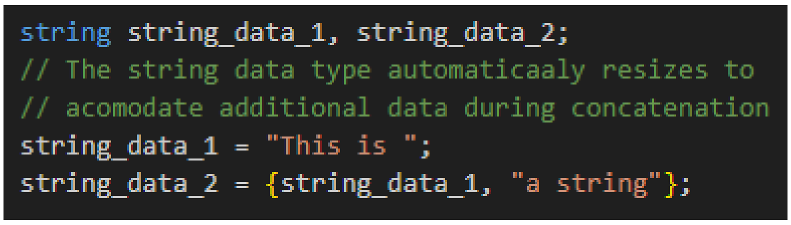
```
logic a1, a2, a3, a4;
logic [1:0] a5;
{a1, a2, a3, a4} = data_2;
// partial assignmnet is also possible,
// but the MSB bits are lost
{a5,a4} = data_2;
```

Part selection within a concatenation operation is also possible but restricted to the right-hand side (RHS) concatenation.

```
bit [7:0] a, b;
bit [3:0] msb_int;
a = 150 ; b = 50;
// Only the 4 most significant bits of
// the sum of a and b are selected
msb_int = {a + b} [7: 4];
```

Do keep in mind that it is essential to avoid using unbounded values during concatenation.

A concatenation of data objects of type string is allowed. If any operand is a string, the concatenation is treated as a string, with all other arguments being converted to string data type. The destination variable (*of type string*) is resized to accommodate the resulting string (*dynamic nature*).



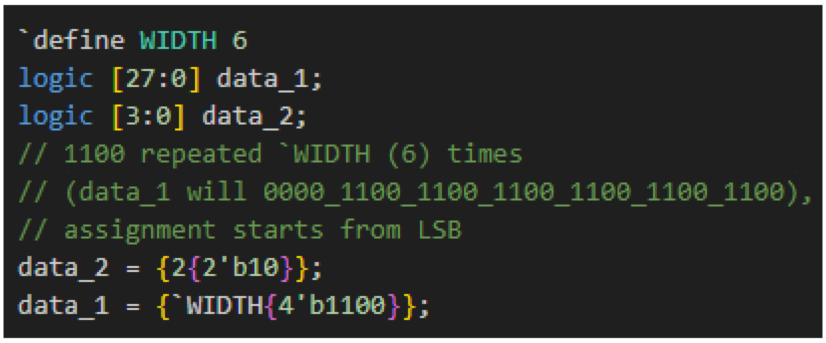
```
string string_data_1, string_data_2;
// The string data type automaticaaly resizes to
// acomodate additional data during concatenation
string_data_1 = "This is ";
string_data_2 = {string_data_1, "a string"};
```

2. Replication Operator

Replication involves duplicating a specified number of copies of a concatenation. The replication operator assigns vectors with the necessary number of 1s and 0s based on the vector size.

The syntax for replication is: {replication_constant {value}}

For example:



```
`define WIDTH 6
logic [27:0] data_1;
logic [3:0] data_2;
// 1100 repeated `WIDTH (6) times
// (data_1 will 0000_1100_1100_1100_1100_1100_1100),
// assignment starts from LSB
data_2 = {2{2'b10}};
data_1 = {`WIDTH{4'b1100}};
```

3. Wildcard Equality Operator

Wildcard equality operator, is used when specific positions in a vector need not be checked, regardless of whether they are set to 1 or 0. This operator, denoted by ?, acts as a wildcard equality or inequality operator.

Any X or Z bit in the variable on the RHS of the comparison is treated as a wildcard. X and Z values in a given bit position of the right operand serve as wildcards. X and Z values in the LHS are not considered as wildcards. If the relation is false, the output is 0. If the relation is unknown, it the output is X.

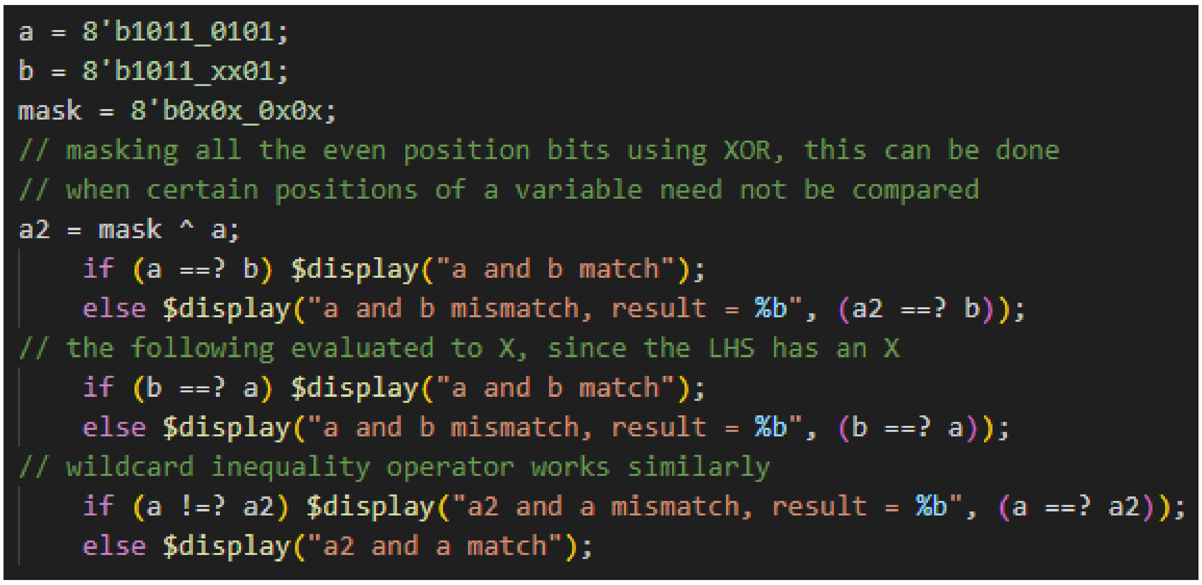
If user wants specific positions of a vector to not be compared, these positions can be made X or Z using XOR operations before applying the wildcard operation.

Wildcard Equality Operator (==?):

a ==? b: Indicates that *a* equals *b*, with X and Z values in *b* acting as wildcards. Positions with X or Z bits in *b* (*RHS expression*) will not be compared.

Wildcard Inequality Operator (!=?):

a !=? b: Indicates that *a* does not equal *b*, with X and Z values in *b* acting as wildcards. If *a* (*LHS expression*) has X or Z, the output of the comparison will be X.



```
a = 8'b1011_0101;
b = 8'b1011_xx01;
mask = 8'b0x0x_0x0x;
// masking all the even position bits using XOR, this can be done
// when certain positions of a variable need not be compared
a2 = mask ^ a;
    if (a ==? b) $display("a and b match");
    else $display("a and b mismatch, result = %b", (a2 ==? b));
// the following evaluated to X, since the LHS has an X
    if (b ==? a) $display("a and b match");
    else $display("a and b mismatch, result = %b", (b ==? a));
// wildcard inequality operator works similarly
    if (a !=? a2) $display("a2 and a mismatch, result = %b", (a ==? a2));
    else $display("a2 and a match");
```

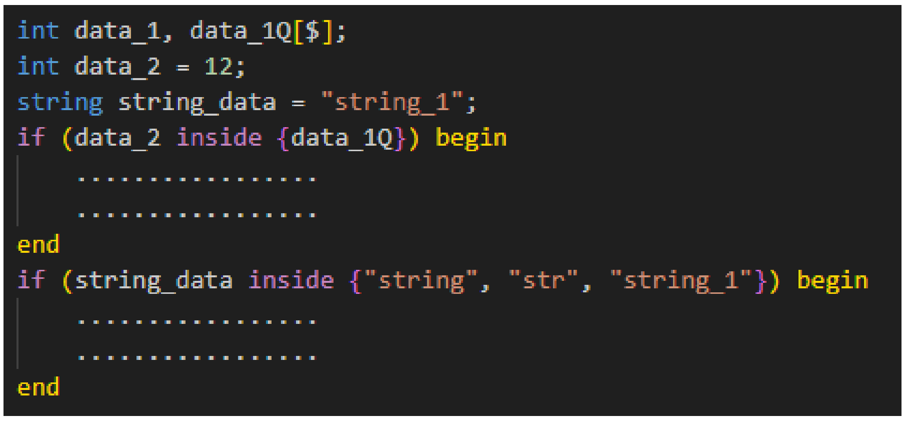
4. *Set Membership Operator*

The set membership operator selects a value from a range of values provided and assigns it to the LHS variable. It is mainly used in conditional statements (if/assert) or constraint blocks in classes.

Syntax: LHS_variable = expression inside {open_range_list}

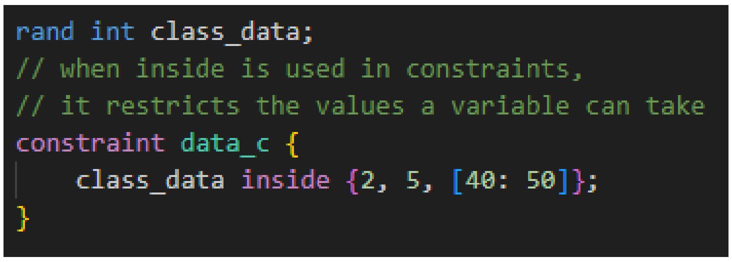
The open_range_list can include *queues, individual elements, arrays, dynamic arrays, strings*, etc., separated by commas. A range can be defined with a low and high bound enclosed by square brackets [ ] and separated by a colon (:) such as *[low_bound:high_bound]*.

Example:



```
int data_1, data_1Q[$];
int data_2 = 12;
string string_data = "string_1";
if (data_2 inside {data_1Q}) begin
    .................
    .................
end
if (string_data inside {"string", "str", "string_1"}) begin
    .................
    .................
end
```

When used in a constraint block in class:



```
rand int class_data;
// when inside is used in constraints,
// it restricts the values a variable can take
constraint data_c {
    class_data inside {2, 5, [40: 50]};
}
```

5. *Streaming Operator*

Streaming operators perform the packing and unpacking of bit-stream types into a sequence of bits in a user-specified order. When applied to the left-hand side, streaming operators perform the reverse operation by unpacking a stream of bits into one or more variables.

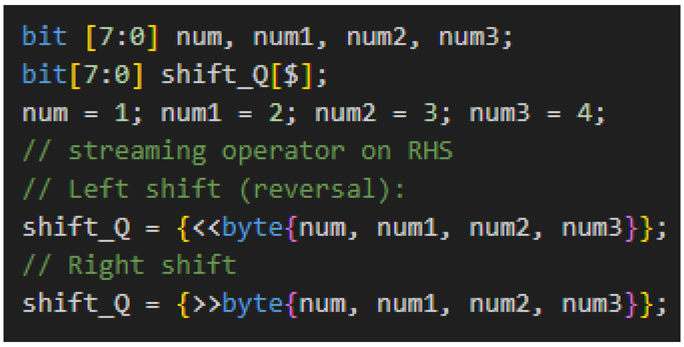
The output of streaming operators is always a queue when it is on the RHS. These operators are crucial in packing large bit streams into *queues of bytes, bits* or *user specified size*.

Syntax: Q_name = {>> / << data_type_size {stream_data}}

The >> or << stream operators determine the order in which blocks of data are streamed. >> causes blocks of data to be streamed in a left-to-right order, while << causes blocks of data to be streamed in a right-to-left order. In the case of right-to-left streaming using <<, the order of blocks in the stream is reversed.

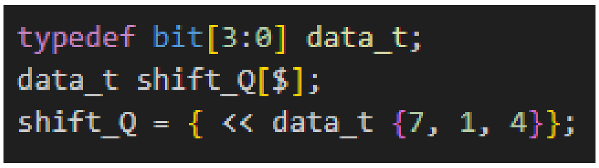
The data_type_size determines the size of each block, measured in bits. It can be any integral type (*bit, byte, int, etc.*) and can also be user-defined using typedef.

Example:



```
bit [7:0] num, num1, num2, num3;
bit[7:0] shift_Q[$];
num = 1; num1 = 2; num2 = 3; num3 = 4;
// streaming operator on RHS
// Left shift (reversal):
shift_Q = {<<byte{num, num1, num2, num3}};
// Right shift
shift_Q = {>>byte{num, num1, num2, num3}};
```

Using typedef:

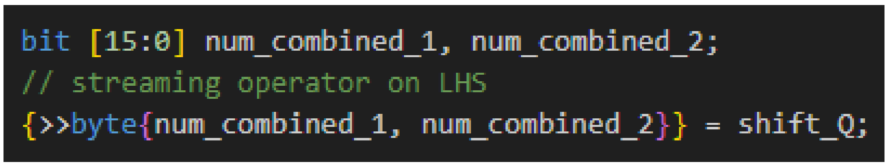


```
typedef bit[3:0] data_t;
data_t shift_Q[$];
shift_Q = { << data_t {7, 1, 4}};
```

Each piece of stream_data undergoes conversion into a bit-stream, which is then appended to a packed array, forming a continuous stream of bits. In the case of an unpacked array, this process is iteratively applied to every element within the array, ensuring that each element contributes to the final bit-stream.

When a streaming operator is located on the left-hand side, it performs the unpacking operation, which involves assigning a stream of bits to one or more variables. In this scenario, the queue serves as the input to the streaming operator.

For example:



```
bit [15:0] num_combined_1, num_combined_2;
// streaming operator on LHS
{>>byte{num_combined_1, num_combined_2}} = shift_Q;
```

In this example, the streaming operator extracts data from the queue shift_Q and assigns it to variables num_combined_1 and num_combined_2.

Arrays

Arrays can be categorized based on how memory is allocated, leading to two primary classifications: Packed arrays and Unpacked arrays.

1. Packed Arrays

In the case of Packed arrays, similar to Verilog vectors, memory is allocated in a single address location for the entire array. The array dimensions are declared before the array name, and the data type is restricted to *bit, logic,* or *reg*. Packed arrays, representing singular data variables stored in a single address location, support all SV operations.

Example:
```
reg [3:0] a; logic [1:0] b;
```

2. Unpacked Arrays

Unpacked arrays resemble traditional verilog arrays as memory is distributed across multiple address locations. The number of locations corresponds to the size of the array, with each location having a size equal to the array's data type (*32bit for int type, 4bit for logic [3:0] etc*). Array dimensions are declared after the array name, and there are no restrictions on data type; any SV data type can be employed. Unpacked arrays may be fixed size arrays, dynamic arrays, associative arrays, or queues.

Example:
```
logic a [3:0]; int b [2:0];
```

However, it's essential to note that unpacked arrays do not support all SV operations. Unlike Packed arrays, which represent singular data variables, Unpacked arrays represent multiple address locations and, as a result, only permit whole array operations. While whole array assignment or comparison is possible, any other operation requires the use of specific indices.

For example:
```
b + 3    // is not possible, but
b[1] + 3 // is possible.
int b [2:0]; int c [2:0];
// Following are also possible:
assert(b == c); // Assertion
b = c;          // Assignment
b = '{3{1}};    // Replication
```

3. Multidimensional arrays (array of arrays)

Multidimensional arrays offer a structured approach to organize data, with accessing starting from the top and moving downwards. The addressing concept plays a pivotal role in determining which dimensions of the array should be packed and which should remain unpacked.

Let's illustrate this concept using the analogy of a building with 8 floors, each floor containing 10 apartments, and each apartment comprising 5 rooms, each with 2 windows.

Unique Addressing for Each Floor:

Addressing each floor individually can be represented as follows:

```
building [9:0] [4:0] [1:0] floor [7:0];
```

(yielding 8 address locations).

Unique Addressing for Each Room:

Addressing each room within each floor can be denoted as:

```
building [1:0] room [7:0] [9:0] [4:0];
```

(resulting in 8*10*5 address locations).

Similarly, for a 128 Byte Size Memory:

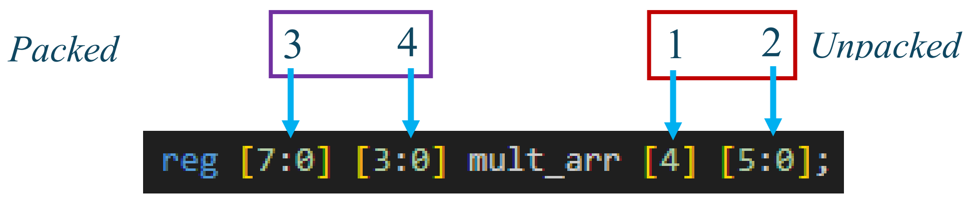
At the byte level, a memory array can be declared as:

```
reg [7:0] memory [127:0];
```

offering a total of 128 locations.

3.1. Multidimensional array allocation

Let's consider the declaration:



```
reg [7:0] [3:0] mult_arr [4] [5:0];
```

This is a multidimensional array with 4 dimensions. This array encapsulates a total of 4*6*8*4 bits, stored across 4*6 addresses, with each location accommodating 8*4 bits.

Unpacked dimensions determine the number of address locations, while packed dimensions specify the number of bits stored in each location.

Assigning values for multidimensional unpacked arrays is facilitated using {} notation.

For example:

```
int int_v [0:1] [0:3] = {'{1,3,7,5},'{0,7,9,4}};
```

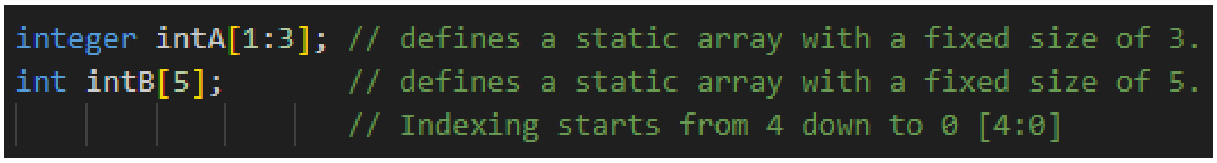
Each number here has its own unique memory.

Another classification based on memory allocation distinguishes between Static arrays and Dynamic arrays.

4. Static Arrays

Static arrays have a fixed size allocated at compile time, making them immutable throughout simulation. They must be declared with a predetermined integer number for the array size, and they can be both packed and unpacked, including multidimensional arrays.

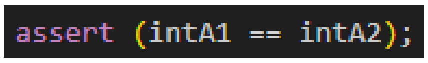
For example:



```
integer intA[1:3]; // defines a static array with a fixed size of 3.
int intB[5];       // defines a static array with a fixed size of 5.
                   // Indexing starts from 4 down to 0 [4:0]
```

foreach() loops are exclusive to static arrays (*or arrays with known size*) and are utilized for traversing through elements efficiently. For instance, looping through elements of *intA* (with a size of 3) can be achieved with *foreach* (intA[i]).

Comparing arrays involves the use of *assert* or if statements. *For example*:



```
assert (intA1 == intA2);
```

compares entire arrays, a functionality not available in Verilog.

5. Dynamic Arrays

In contrast, dynamic arrays allocate memory at runtime, allowing for size adjustments during execution. They necessitate the use of the *new[]* keyword for memory allocation, although dynamic arrays can operate without new through array assignment. Unlike static arrays, the size is not specified during array declaration.

They also offer flexibility in memory management, enabling size increments, decrements, or deletions during runtime. They are exclusively unpacked arrays; packed arrays cannot be dynamic.

Associative arrays and queues (discussed later), are also dynamic as they allocate memory upon usage but do not require *new[]* for memory allocation. Dynamic arrays can be assigned to fixed size arrays (*static arrays*) and vice versa, provided both arrays share a compatible data type and have the same size.

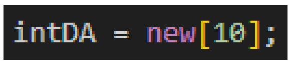
5.1. Dynamic Array Allocation

Dynamic arrays can be declared using the format:

data_type name[], where the *data_type* can be any SV datatype.

The syntax for creating a dynamic array involves using new[], where the array size is specified within square brackets.

For example:



```
intDA = new[10];
```

signifies the creation of a dynamic array named *intDA* with a size of 10.

Dynamic arrays offer flexibility in resizing or allocating memory, with two special cases where using new[] is not required:

During Initialization: When initializing a dynamic array, memory allocation happens automatically without using *new[]*.

For example: `int intDA[] = {1, 4, 6, 13};`

Here, *intDA* is initialized with the specified elements, and memory allocation is handled automatically.

Assigning Static Arrays or Queues: Dynamic arrays can be assigned values from static arrays or queues without invoking new[].

For example:

```
intDA = intQ;
intDA = intSA;
```

Here, *intDA* dynamically adjusts its size to accommodate the elements from *intQ* (*queue*) or *intSA* (*static array*) without requiring explicit memory allocation.

5.2. Dynamic Elements in Dynamic Arrays (e.g., Class Instances)

For dynamic arrays containing dynamic elements, such as a class, the allocation process involves two steps: first, allocating memory for the dynamic array itself, and second, allocating memory for each dynamic element within the array.

Consider the example of a *packet class* declared as *packet* pkt_array[]:

pkt_array is allocated memory by doing: `pkt_array = new[10];`

However, this only allocates memory for the array, not the individual packet elements. To allocate memory for each packet, a loop is used:

```
foreach (pkt_array[i]) pkt_array[i] = new();
// new[] : For arrays
// new() : For class objects
```

Here, memory for each packet is explicitly allocated using new().

5.3. Dynamic Array Functions

new[] :

Allocates memory to dynamic array elements. *new[]* supports preserving and copying elements from one dynamic array to another.

For example:

```
pkt = new [100];         // Allocates memory for a dynamic array
                         // named pkt with 100 elements.
pkt_2 = new [50] (pkt); // Preserves and copies the first 50
                         // elements of pkt to a new array pkt_2.
```

delete():

Deletes all contents of a dynamic array. However, there is no built-in method to delete only one element of a dynamic array.

For example:

```
pkt = new[100];
pkt.delete();
```

size():

Retrieves the size of the dynamic array, returning it as a value. It returns the size as an integer.

For example:

```
int count;
.........
.........
count = pkt.size();
```

If the number of elements in pkt is 5, then *count* = *5* here.

6. Queues

A queue represents a variable sized, ordered collection of uniform (homogeneous) elements. Variable size allows for both expansion and reduction as needed. In addition, the elements within a queue are ordered, with each element assigned an index starting from 0, and can be accessed based on index values.

Queues are dynamic data types, and memory allocation does not require the use of the *new[]* keyword. They are declared using a syntax similar to unpacked arrays, with the size specified using *$*. The maximum size of a queue can be defined by indicating its optional last index.

The syntax for declaring a queue is: *data_type* Q_name[$]

Queue declaration examples:

```
int intQ[$];      // Queue of int type
int intQ [$:31]; // Queue capable of accommodating
                  // 32 elements, initialized as empty
```

Queues can be compared as a whole, assuming both queues have the same number of elements. Comparison operations can be performed directly between queues, between a queue and a dynamic array, or between a queue and a fixed array without the need for a foreach loop.

Queues are particularly valuable in scenarios where the number of elements frequently grow and shrink, and where users require efficient search and sort functionalities. One such example is in the implementation of *scoreboards*.

6.1. Queue Methods

size():

Retrieves the size of the queue, returning it as an integer value.

```
Q_size = intQ.size();
```

insert():

Inserts elements at a specific index of the queue. If a queue has hit the maximum storage limit, data cannot be inserted, the *insert()* method returns the queue as it is with a *Queue operation would exceed max* warning.

The *index-position* must be available in the queue.

```
Q_name.insert(index-position, data);
stringQ.insert(1, "inserted_data");
```

delete():

Queue elements can be deleted at specific index location or the entire queue can be deleted

For specific index location:

```
Q_name.delete(index-position)
stringQ.delete(1);
```

For entire queue, there are two ways:

```
Q_name.delete()
stringQ.delete();
// or
Q_name = {} // Rewrites the queue as empty
intQ_2 = {};
```

pop_front():

Removes and returns the first element from the queue.

```
data = intQ_2.pop_front();
```

pop_back():

Removes and returns the last element from the queue.

```
data = intQ_1.pop_back();
```

push_front():

Adds data into the queue at the first index position.

```
intQ_2.push_front(data);
```

push_back():

Adds data into the queue at the end of the queue.

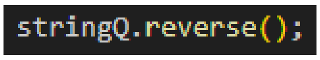
```
intQ_1.push_back(data);
```

shuffle():

Shuffles the queue elements in random order.

```
stringQ.shuffle();
```

reverse():

Reverses the order of elements in the queue. 

```
stringQ.reverse();
```

**Note: The data variable used in pop_front(), pop_back(), push_front() and push_back() needs to be of the same type as the elements in the queue.*

7. Associative array

Associative arrays are a type of array where elements are stored and accessed using corresponding index values. These index values are not required to be continuous and can start from any value.

Declaration: *data_type* arr_name[*index_type*]

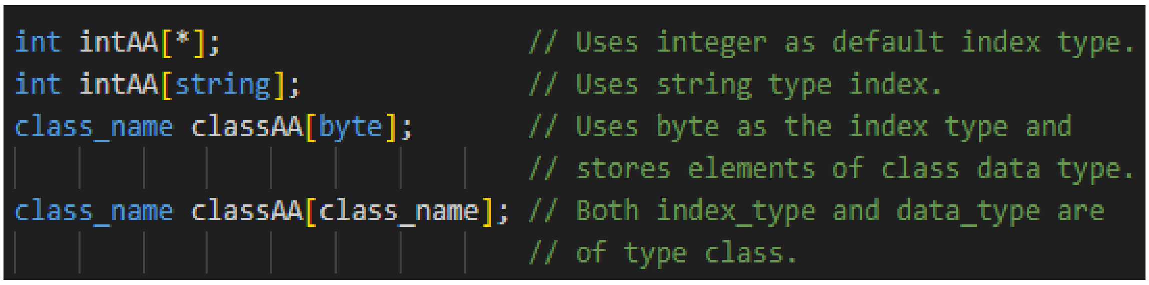
Where, *data_type* represents the type of data stored, and *index_type* is the lookup key used to access array elements. Both *data_type* and *index_type* can be of any SV data type.

Associative arrays do not require elements to exist at every index, allowing for non-continuous indexing. In the case of a fixed size array, the index range implies that elements must exist at all those indices. However, associative arrays have the flexibility to have elements only at specific indices that are accessed. They behave similar to linked lists and keep track of next and previous indices.

They are particularly useful for modelling large memories when the user doesn't want to allocate memory to each location of the memory range. Memory is only allocated for elements that are accessed, making them more memory efficient.

Associative arrays support various index types such as *integer, bit, string,* or even *class* types. Most commonly, index types are integers, bits, or strings. For string indices, ASCII values are treated as address values, and for class indices, handle address values are used for indexing.

Examples of different associative array declarations:



```
int intAA[*];                      // Uses integer as default index type.
int intAA[string];                 // Uses string type index.
class_name classAA[byte];          // Uses byte as the index type and
                                   // stores elements of class data type.
class_name classAA[class_name];    // Both index_type and data_type are
                                   // of type class.
```

7.1. Why Use Associative Arrays

Consider a scenario where testing a very large memory with 1 million addresses is necessary. To verify its functionality, it's sufficient to write to a few random locations and read those locations in a random order. In this case, using only 10 addresses for verification suffices.

Challenges with Static Arrays:

Using static arrays poses a problem even when dealing with only 10 addresses. Despite needing a small subset of addresses, declaring a memory of size 1 million is compulsory. Static arrays require storing data in a continuous index basis, meaning if location 100 exists, locations 0 to 99 must also exist, leading to unnecessary memory allocation.

Challenges with Dynamic Arrays:

Similarly, dynamic arrays allow runtime size allocation, but they encounter the same issue as static arrays. They still require the declaration of a large memory size upfront, regardless of the actual usage.

The Solution:

A storage method is needed where only assigned locations are allocated, leaving the remaining memory addresses non-existent. This approach optimizes memory usage, especially when dealing with vast memory spaces.

Associative arrays offer a solution by creating a linked list of the indices that are used, allowing for dynamic memory allocation based solely on the utilized addresses. In contrast to static or dynamic arrays, where elements are stored in continuous address locations, associative arrays store elements at discrete locations. Therefore, indexing becomes a crucial aspect of accessing elements within associative arrays.

Associative arrays can only be assigned to another associative array of a compatible type and with the same index type. Other types of arrays cannot be assigned to an associative array, nor can associative arrays be assigned to other types of arrays, whether fixed-size or dynamic.

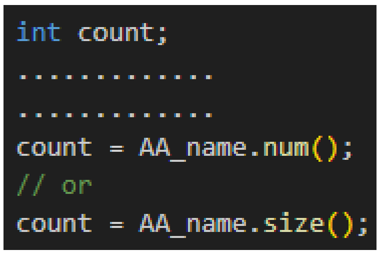
7.2. Associative Array Methods

Associative arrays offer several useful methods for managing and accessing elements efficiently. They do not have *new[]* method. Here's a breakdown of some key methods:

num() or size():

Returns the number of elements in the associative array.

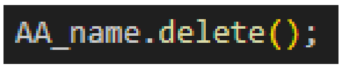
Count is updated with the total number of elements in the array.



```
int count;
.............
.............
count = AA_name.num();
// or
count = AA_name.size();
```

delete():

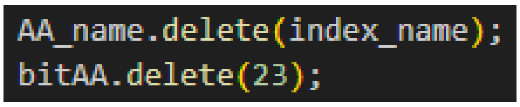
Allows deletion of either all array elements or a specific element.

Deleting all elements: 

```
AA_name.delete();
```

Deleting a specific element:

Returns 0 if the index 23 is not present in the array, otherwise, deletes the element at index 23 and returns 1.

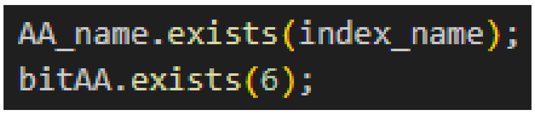


```
AA_name.delete(index_name);
bitAA.delete(23);
```

exists():

Checks if an element exists at a given index.

Returns 1 if index 6 exists; otherwise, returns 0.

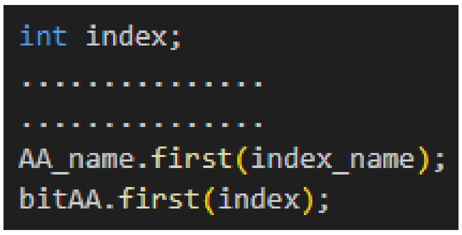


```
AA_name.exists(index_name);
bitAA.exists(6);
```

first():

Returns the first index value in the array and assigns it to *index_name*.

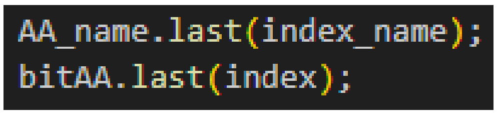
Index can be declared as *int, string*, etc.



```
int index;
...............
...............
AA_name.first(index_name);
bitAA.first(index);
```

last():

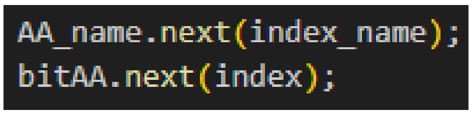
Returns the last index value in the array and assigns it to *index_name*



```
AA_name.last(index_name);
bitAA.last(index);
```

next():

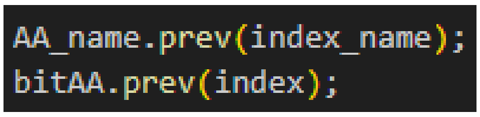
index_name needs to be initialized to an index value present in the associative array. Returns the next index value in the array for a given *index_name*.



```
AA_name.next(index_name);
bitAA.next(index);
```

prev():

Returns the previous index value in the array for a given *index_name*. *index_name* needs to be initialized to an index value present in the associative array.



```
AA_name.prev(index_name);
bitAA.prev(index);
```

8. Common array manipulation methods

SV offers a range of built-in methods tailored for array manipulation, including searching, ordering, and reduction. These methods can be applied across different array types: static arrays, dynamic arrays, queues, and associative arrays. When combined with the *with()* method, they become particularly powerful.

The *with()* method accepts an optional expression enclosed in parentheses. As the array manipulation methods iterate over the array elements, the expression specified in the *with()* clause is evaluated using these elements.

8.1. Array locator methods

These methods return a queue containing the elements that fulfil the specified criteria. The return type of the index method varies depending on the array iterator items: for most arrays, it's an integer, while for associative arrays, it matches the associative index type. It's important to note that associative arrays with a wildcard index type ([*]) are not permitted.

Here are the key array locator methods:

All the following methods require a with() clause, not using with() will cause an error

find():

Returns all elements satisfying the given expression.

```
// Find all items greater than 70,
value_Q = bitAA.find(xyz) with (xyz > 70);
// Find all items with values equal to their index
value_Q = bitAA.find(item) with (item == item.index);
```

find_first():

Returns the first element satisfying the given expression.

```
// Find the first element divisible by 3
value_Q = bitAA.find_first(item) with (item % 3 == 0);
```

find_last():

Returns the last element satisfying the given expression.

```
// Find the last element divisible by 3
value_Q = bitAA.find_last(item) with (item % 3 == 0);
```

find_index():

Returns the indices of all elements satisfying the given expression.

```
// Find all elements greater than 70 and less than 80,
index_Q = bitAA.find_index with (item > 70 && item < 80);
```

find_first_index():

Returns the index of the first element satisfying the given expression.

```
// Find first index with specified value
index_Q = bitAA.find_first_index(s) with (s == 86);
```

find_last_index():

Returns the index of the last element satisfying the given expression.

```
// Find last index with specified value,
index_Q = bitAA.find_last_index(s) with (s == 86);
```

For the following methods a with() clause is optional

max(), min():

Returns the element with the maximum/minimum value.

```
// convert string_SA array string elements from ASCI to integer (atoi)
// and based on that find the max and min
string_Q = string_SA.max() with(item.atoi);
string_Q = string_SA.min() with(item.atoi);
index_Q = bitAA.max();
```

unique():

Returns all elements with unique values. The returned queue contains only one entry for each unique value found in the array.

```
value_Q = bitAA.unique();
```

unique_index():

Returns the indices of all elements with unique values.

```
index_Q = bitAA.unique_index();
```

8.2. Array ordering methods

Array ordering methods reorder the elements of unpacked arrays, whether they are fixed or dynamically sized, except for associative arrays. Here are the supported ordering methods:

sort():

Sorts the unpacked array in ascending order.

```
array_name.sort();
```

rsort():

Sorts the unpacked array in descending order.

```
array_name.rsort();
```

reverse():

Reverses all the elements of an array. Note that *with()* cannot be used with the reverse method.

```
array_name.reverse();
```

shuffle():

Randomizes the order of the elements in the array.

```
array_name.shuffle();
```

8.3. *Array reduction methods*

Array reduction methods are used to condense arrays with integral values into a single value. *with()* can be used with all of them.

sum():

Returns the sum of all the array elements.

```
// we can use with() to change the width of the result to 16 bits
bit_reduction = value_Q.sum() with (16'(item));
int_reduction = int_Q.sum();
```

product():

Returns the product of all the array elements.

```
bit_reduction = value_Q.product() with (16'(item));
int_reduction = intDA.product();
```

and():

Returns the bitwise AND of all the array elements.

```
bit_reduction = value_Q.and();
```

or():

Returns the bitwise OR of all the array elements.

```
bit_reduction = bitAA.or();
```

xor():

Returns the bitwise XOR of all the array elements.

```
int_reduction = int_Q.xor();
```

9. *Printing arrays*

Printing arrays in SV can be done efficiently using the packed print method (%p), which prints the entire array in a single line, starting from the most significant bit (MSB) side. Additionally, printing individual elements is also feasible.

9.1. *Printing Entire Array*

To print the entire array *intA1* using the packed print method, you can use the *%p* specifier within the *$display* statement:

```
$display("intA1 = %p", intA1);
```

This line prints the entire array *intA1* in a single line, starting from the MSB side.

9.2. *Printing Single Elements*

If you wish to print specific elements of the array, you can specify the index within square brackets []:

```
$display("intA1[0] = %d", intA1[0]);
```

Here, *intA1[0]* denotes the element at index 0 of the array *intA1*. Using *%d*, the value of this specific element is printed.

10. *String formatting in arrays*

String formatting within arrays is done using the *$sformatf* string format function. This function generates formatted strings based on specific patterns or templates.

For example: Assume an array of *string* type named *names* declared as *string* names[10:1], you can populate its elements inside a loop using *$sformatf* as follows:

```
for (int i = 1; i <= 10; i++) begin
    names[i] = $sformatf("str%0d", i);
end
```

In this loop, *i* iterates from 1 to 10, and for each iteration, the *$sformatf* function formats the string according to the pattern *"str%0d"*, where *%0d* represents a placeholder for the integer *i*. This results in strings like *"str1", "str2"*, and so forth being assigned to elements of the *names* array.

Additionally, you can use *$sformat* which is string format task:

```
$sformat(name, "str%0d", i);
```

Object Oriented Programming

Unlike Verilog, SV has constructs implemented for object Oriented Programming (OOP) that revolves around the concept of objects (*instances of a class, building blocks of a testbench*), which can contain data in the form of fields (*often called attributes or properties*) and code in the form of methods (*known as functions, or tasks*). Key principles of OOP include inheritance, polymorphism, encapsulation, and abstraction.

1. *Inheritance*

Inheritance is a fundamental concept in programming where a derived class can inherit properties, methods, and constraints from a base class. This means that all the attributes and functions defined in the base class are automatically accessible in the derived class when it's created. In SV, this inheritance relationship is established using the extends keyword.

For example:

```
class derived_class_name extends base_class_name;
```

2. *Polymorphism*

Polymorphism in programming is where the behaviour of objects can vary depending on the specific instance at runtime. It allows a derived class to modify or redefine a method declared in its base class to suit its particular needs. For polymorphism to work, the method prototype must remain consistent across both the base class and the derived class.

In SV, the virtual keyword is required to enable polymorphism. When applied to a method in the base class, it indicates that this method can be overridden by a method with the same prototype in a derived class.

It's considered a good practice to declare all base class methods as virtual if polymorphism is desired. Methods that lack the virtual keyword in the base class cannot be overwritten, thus limiting the flexibility and adaptability of the derived classes. Note that polymorphism cannot happen without inheritance.

For example:

```
// Definition can be altered or overridden in derived classes.
virtual function return_type base_class_function_name();
// Unlike virtual methods, the following function definition cannot
// be changed or overridden in derived classes.
function return_type base_class_function_name();
```

3. *Encapsulation*

Encapsulation involves bundling together properties (*data*) and methods (*functions or tasks*) into a cohesive unit known as a class. This means that the data and the operations that manipulate that data are encapsulated within the same entity.

In essence, encapsulation allows for the organization of data and functionality related to a specific concept or entity into a single container (*class*). This container is self-contained and responsible for managing its own properties and methods.

4. *Abstraction*

Abstraction involves hiding the complex inner workings of a class and presenting only the essential features to the outside world. It allows programmers to focus on what an object does, rather than how it achieves its functionality.

In the context of OOP, abstraction is achieved through data hiding, where the internal implementation details of a class are concealed from external users. This is done by restricting access to certain variables or methods of a class, ensuring that they can only be accessed or modified in specific ways defined by the class itself.

Keywords such as local or protected are often used to control the visibility and accessibility of class elements, limiting their exposure to the outside world.

Note:

This section offers a brief overview of fundamental OOP concepts. For a deeper understanding, each concept is explored in detail in the subsequent section of classes and methods.

Introduction to Classes and Methods I

Classes and methods serve as the foundation of an SV-based testbench, empowering SystemVerilog with its robustness for design verification. These components are dynamic and require memory allocation when instantiated. The memory allocated to a class object comprises the combined sizes of all its properties (*or signals*). To interact with properties, methods, or constraints, memory allocation for the object (*class instances*) is essential. This section primarily focuses on class properties and methods, with constraints slated for discussion in a separate section.

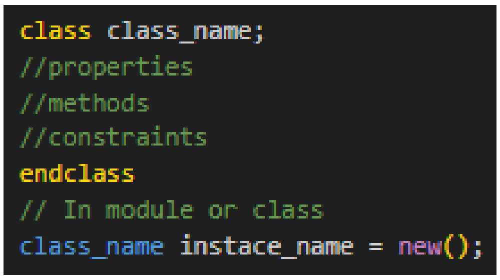
1. Defining a Class

Classes are dynamic constructs that facilitate the creation and deletion of objects (*class instances*) dynamically. These objects can be passed around using object handles (*pointers to the class instances*). The process of defining a class data type typically includes:

1. Identifying the properties needed for the class.

2. Listing down the required methods, determining whether they should be tasks or functions.

3. Considering if any constraints are necessary for the properties.



```
class class_name;
//properties
//methods
//constraints
endclass
// In module or class
class_name instace_name = new();
```

Classes consist of three main elements: properties (*data variables/signals*), methods (*tasks or functions*), and constraints. Objects, or instances of a class, are fundamental to all SV testbenches.

Methods within a class can be categorized into two types:

Built-in methods: A class inherently possesses four built-in methods: new, randomize, pre-randomize, and post-randomize.

User-defined methods: These methods are defined by the user to suit specific requirements.

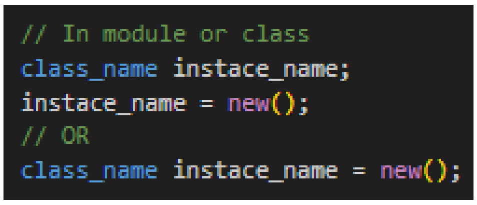
2. Creating Class Instances (Objects)

To create class instances, also known as objects, you first declare a variable of the class type. Then, you instantiate an object of that class type using the new() keyword.

Here's the syntax: class_name instance_name;

This creates an object handle for the class, but it doesn't allocate memory to it. The new() keyword is used specifically for memory allocation.

For example:



```
// In module or class
class_name instace_name;
instace_name = new();
// OR
class_name instace_name = new();
```

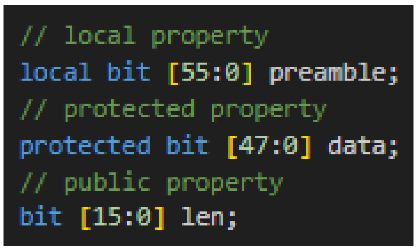
Once an object is created, its methods can be accessed using the dot operator (`.`).

3. Properties

Properties, also known as fields or signals, are essential components of a testbench, representing data required by both the Design Under Test (*DUT*) and the test environment. Typically, properties are of integral, real, or string types, although they can encompass any SV data type.

Properties can be categorized as follows:

Local: These properties are accessible only within the methods and constraints of the same class.



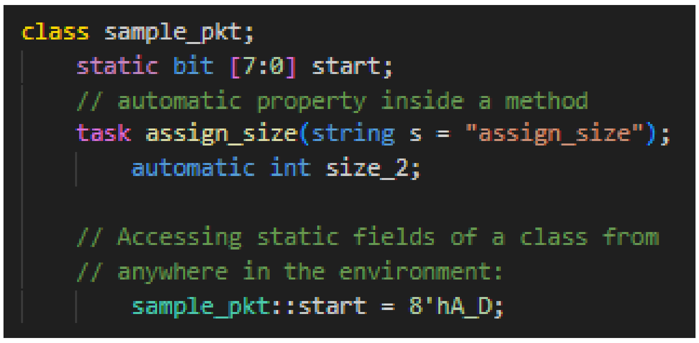
```
// local property
local bit [55:0] preamble;
// protected property
protected bit [47:0] data;
// public property
bit [15:0] len;
```

Protected: Accessible within the methods and constraints of the same class and any inherited classes. Properties marked as protected cannot be accessed by other classes or modules.

Public: These properties, which are neither local nor protected, are accessible throughout the test environment or testbench.

Additionally, properties can have the following characteristics:

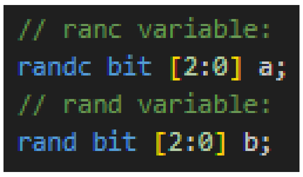
Static: Memory allocation for these properties occurs at a single location shared by all instances of the class. This allows for the creation of a commonly shared database for the entire environment. Static variables can be accessed using the scope resolution operator (::).



```
class sample_pkt;
    static bit [7:0] start;
    // automatic property inside a method
    task assign_size(string s = "assign_size");
        automatic int size_2;

    // Accessing static fields of a class from
    // anywhere in the environment:
        sample_pkt::start = 8'hA_D;
```

Automatic: Each class instance has dedicated memory allocation for automatic properties. Properties can be declared as automatic only inside procedural blocks. Once the block is executed, the automatic property memory is deallocated.

For randomization purposes, properties can be declared as rand or randc. The randomize method is built-in and can randomize any property declared as rand or randc. When using randc, cyclic randomization occurs in a specific order, ensuring that all possible values are generated before repetition starts. It's important to note that randc should not be used in conjunction



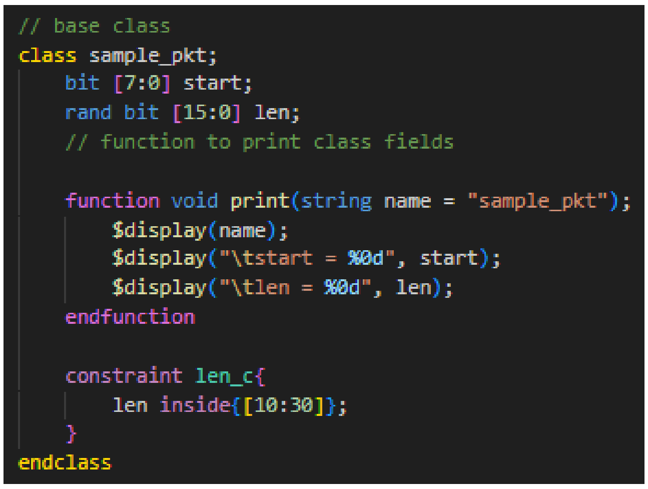
```
// ranc variable:
randc bit [2:0] a;
// rand variable:
rand bit [2:0] b;
```

with new() inside loops in the initial block. This is because new() resets the memory used for previous values, disrupting the cyclic order.

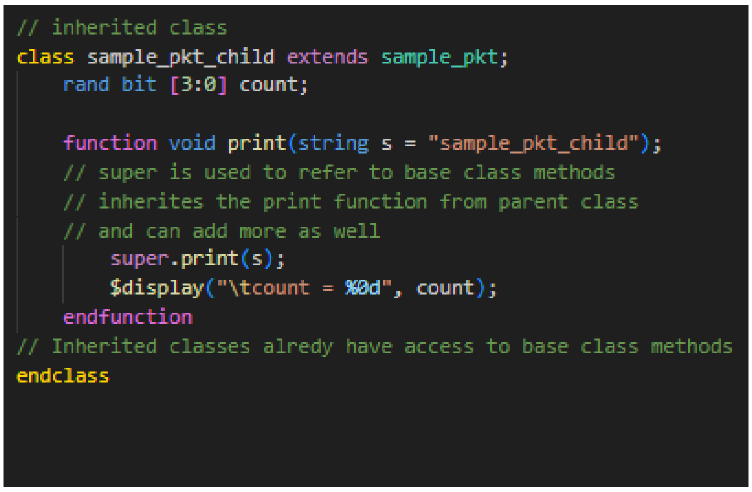
As a best practice, it's advisable to declare every property as protected by default unless there's a specific need for it to be public. If external access to a protected property is necessary, it's recommended to declare separate functions to access and modify the property.

4. *Class Inheritance*

Class inheritance allows for the extension of a base class into a new subclass using the extends keyword. This process grants the subclass access to all the properties and methods of the base class. Additionally, the derived class can implement its own unique properties and methods while retaining access to those inherited from the base class.



```
// base class
class sample_pkt;
    bit [7:0] start;
    rand bit [15:0] len;
    // function to print class fields

    function void print(string name = "sample_pkt");
        $display(name);
        $display("\tstart = %0d", start);
        $display("\tlen = %0d", len);
    endfunction

    constraint len_c{
        len inside{[10:30]};
    }
endclass
```



```
// inherited class
class sample_pkt_child extends sample_pkt;
    rand bit [3:0] count;

    function void print(string s = "sample_pkt_child");
    // super is used to refer to base class methods
    // inherites the print function from parent class
    // and can add more as well
        super.print(s);
        $display("\tcount = %0d", count);
    endfunction
// Inherited classes alredy have access to base class methods
endclass
```

5. *Methods*

A method refers to a procedural block of code that is associated with a class. Methods encapsulate behaviour and can manipulate the data (properties) of the class. They are similar to functions or procedures in other programming languages.

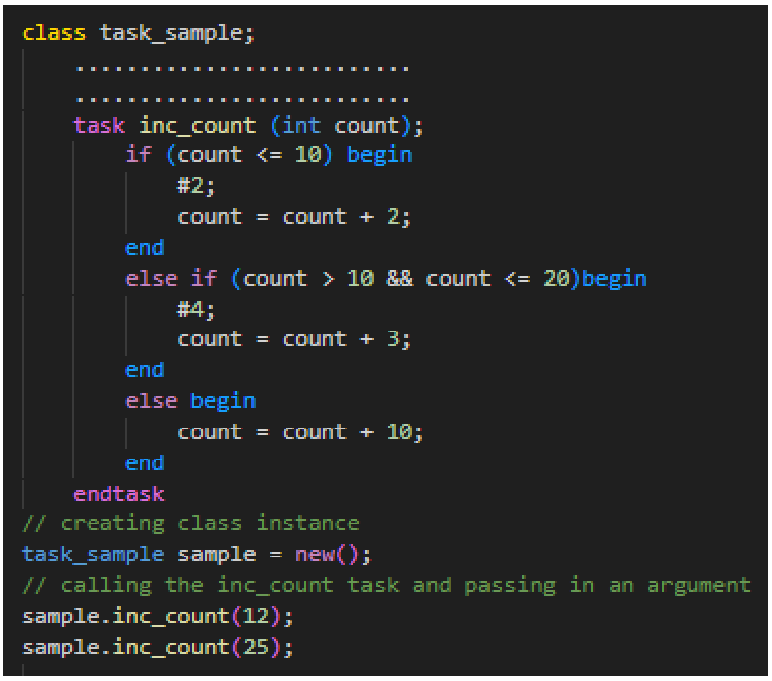
SV defines two main types of methods:

5.1. *Task*

A task represents a procedural block of code that can have zero or more input/output arguments. It does not return a value. Tasks can execute in zero or more simulation time units. If an operation needs to take time to execute, using a task is recommended.

The prototype for a task is: task task_name(arguments);

For example: The following code snippet demonstrates how a task is capable of executing in 0 or more simulation time.



```
class task_sample;
    .........................
    .........................
    task inc_count (int count);
        if (count <= 10) begin
            #2;
            count = count + 2;
        end
        else if (count > 10 && count <= 20)begin
            #4;
            count = count + 3;
        end
        else begin
            count = count + 10;
        end
    endtask
// creating class instance
task_sample sample = new();
// calling the inc_count task and passing in an argument
sample.inc_count(12);
sample.inc_count(25);
```

5.2. *Function*

A function is also a procedural block of code that can have zero or more input/output arguments. Unlike a task, a function returns a value. Functions execute instantaneously, without consuming simulation time.

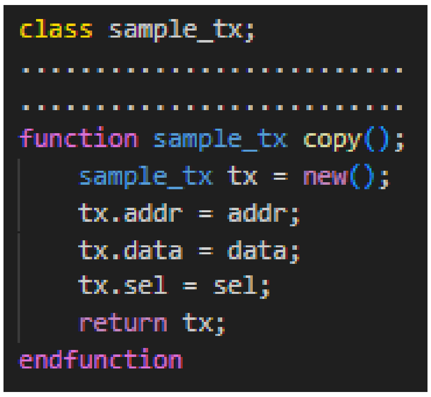
The prototype for a function is:

function return_type function_name(arguments);

In SV, all function arguments are considered as input by default. To specify an argument as an output, it must be explicitly declared.

Example: function void add (a, b, output c);

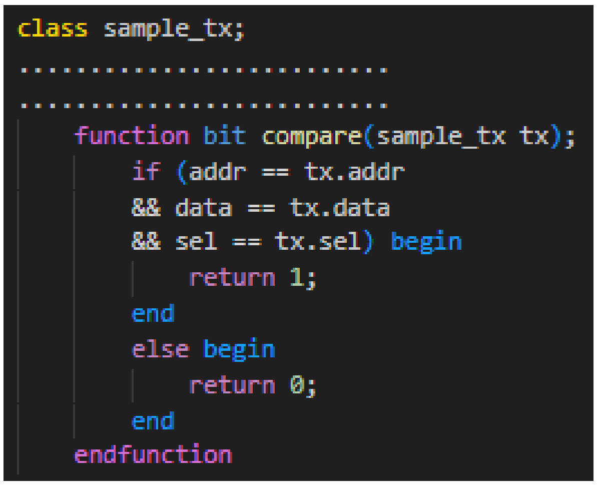
Following is an example of a function that returns a class instance (*object*).



```
class sample_tx;
.........................
.........................
function sample_tx copy();
    sample_tx tx = new();
    tx.addr = addr;
    tx.data = data;
    tx.sel = sel;
    return tx;
endfunction
```

A function can also accept the class itself as an input/output argument or return type.

For example, let's consider implementing a compare function within a *sample_pkt* class:



```
class sample_tx;
..........................
..........................
    function bit compare(sample_tx tx);
        if (addr == tx.addr
        && data == tx.data
        && sel == tx.sel) begin
            return 1;
        end
        else begin
            return 0;
        end
    endfunction
```

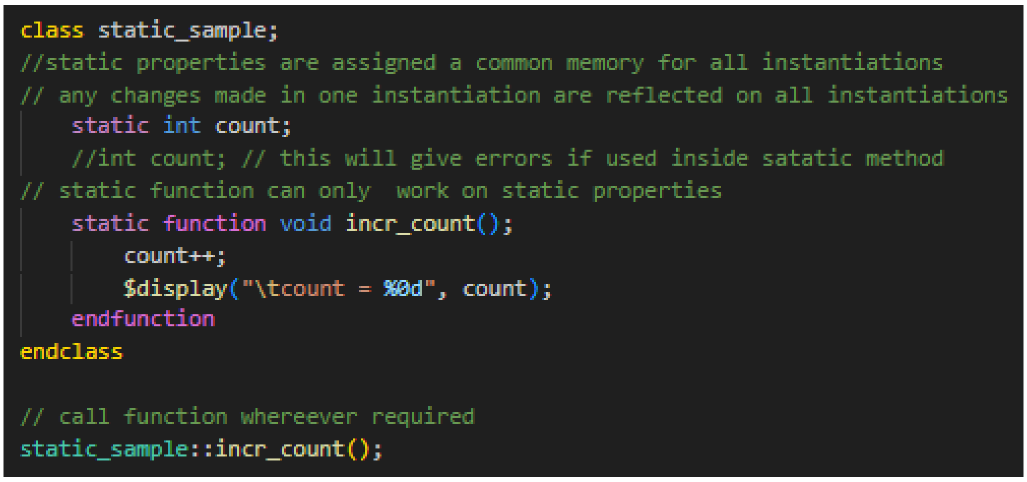
In this example, the compare function takes another *sample_pkt* object *pkt* as an argument. It compares the properties of the current object (*this*) with the properties of the provided *pkt* object.

5.3. *Static Methods and Properties*

Static methods exclusively access static properties and cannot access automatic properties within their method definition. They are integrated into the class definition, eliminating the necessity for object instantiation to access them. They can be invoked directly via the class name using the scope resolution operator(::).

Attempting to access automatic properties within a static method can lead to conflicts, as static methods try to access non-existent properties since automatic properties are instantiated only with new().

Memory allocation for static methods or properties occurs as soon as the class is defined. They do not require the use of new().



```
class static_sample;
//static properties are assigned a common memory for all instantiations
// any changes made in one instantiation are reflected on all instantiations
    static int count;
    //int count; // this will give errors if used inside satatic method
// static function can only  work on static properties
    static function void incr_count();
        count++;
        $display("\tcount = %0d", count);
    endfunction
endclass

// call function whereever required
static_sample::incr_count();
```

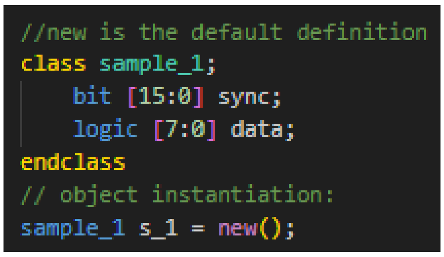
6. *New Constructor*

The new constructor is inherent to every class and is responsible for object initialization, allocating memory for object fields, and assigning default values to them. Users can either utilize the default definition provided by the language or define their own implementation of the new constructor. Fields can be accessed using the dot ('.') operator.

There are various coding styles for implementing the new constructor:

6.1. Default Definition

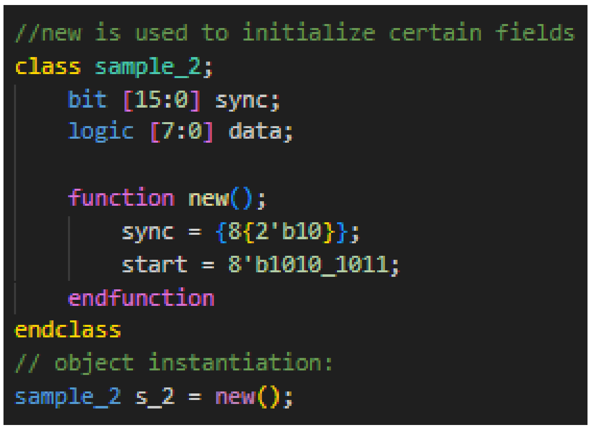
No need to explicitly define the new constructor; the language provides a built-in version.



```
//new is the default definition
class sample_1;
    bit [15:0] sync;
    logic [7:0] data;
endclass
// object instantiation:
sample_1 s_1 = new();
```

6.2. Field Initialization

Implement the new constructor and assign specific values to the fields (*using the this keyword is optional here*). This method is suitable when initializing objects with specific field values.

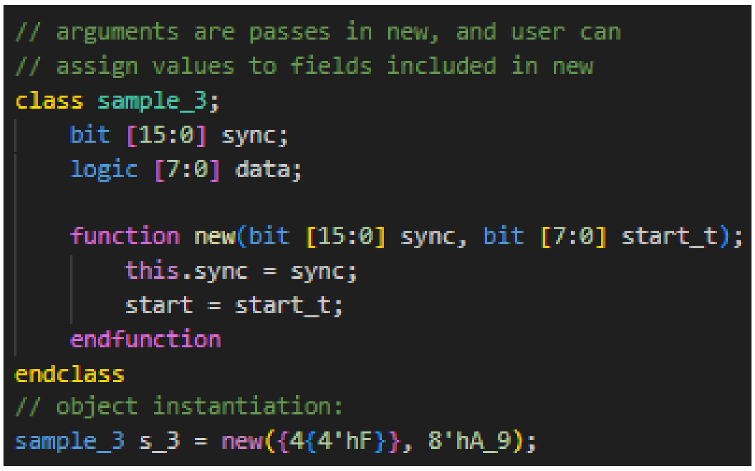


```
//new is used to initialize certain fields
class sample_2;
    bit [15:0] sync;
    logic [7:0] data;

    function new();
        sync = {8{2'b10}};
        start = 8'b1010_1011;
    endfunction
endclass
// object instantiation:
sample_2 s_2 = new();
```

6.3. Initialization Using new Arguments

Define new() with input arguments to initialize the fields with user-specified values. This approach is helpful when users want to pass specific values during object creation.

Example:



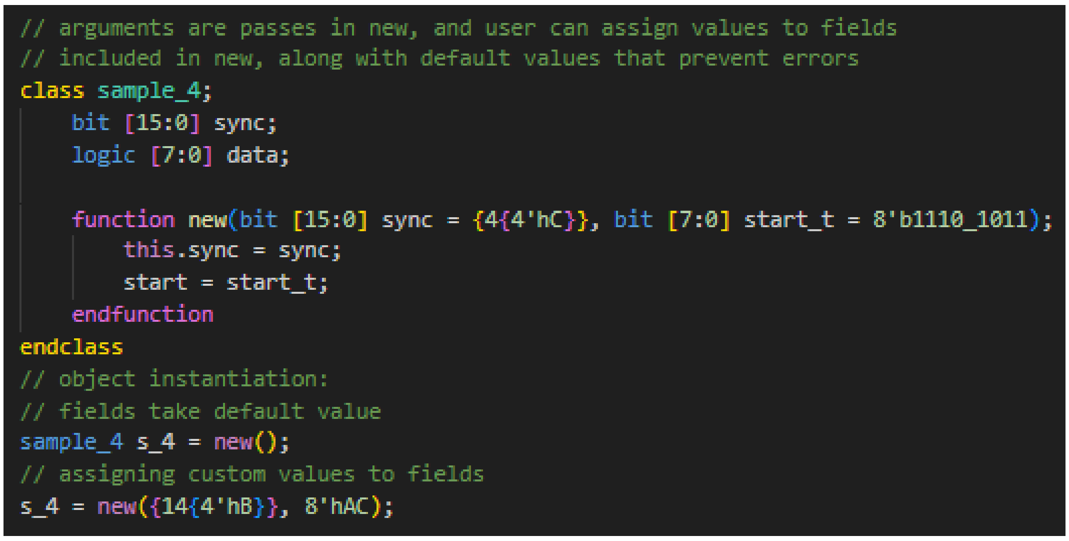
```
// arguments are passes in new, and user can
// assign values to fields included in new
class sample_3;
    bit [15:0] sync;
    logic [7:0] data;

    function new(bit [15:0] sync, bit [7:0] start_t);
        this.sync = sync;
        start = start_t;
    endfunction
endclass
// object instantiation:
sample_3 s_3 = new({4{4'hF}}, 8'hA_9);
```

The example illustrates two distinct approaches to initializing the new() with input arguments. *this.sync* refers to the size field defined within the class, while sync represents the input argument passed to the constructor.

6.4. Initialization with Default Values

Using default values in new() helps prevent errors and simplifies object creation. This concept applies to all tasks and functions.

Example:



```
// arguments are passes in new, and user can assign values to fields
// included in new, along with default values that prevent errors
class sample_4;
    bit [15:0] sync;
    logic [7:0] data;

    function new(bit [15:0] sync = {4{4'hC}}, bit [7:0] start_t = 8'b1110_1011);
        this.sync = sync;
        start = start_t;
    endfunction
endclass
// object instantiation:
// fields take default value
sample_4 s_4 = new();
// assigning custom values to fields
s_4 = new({14{4'hB}}, 8'hAC);
```

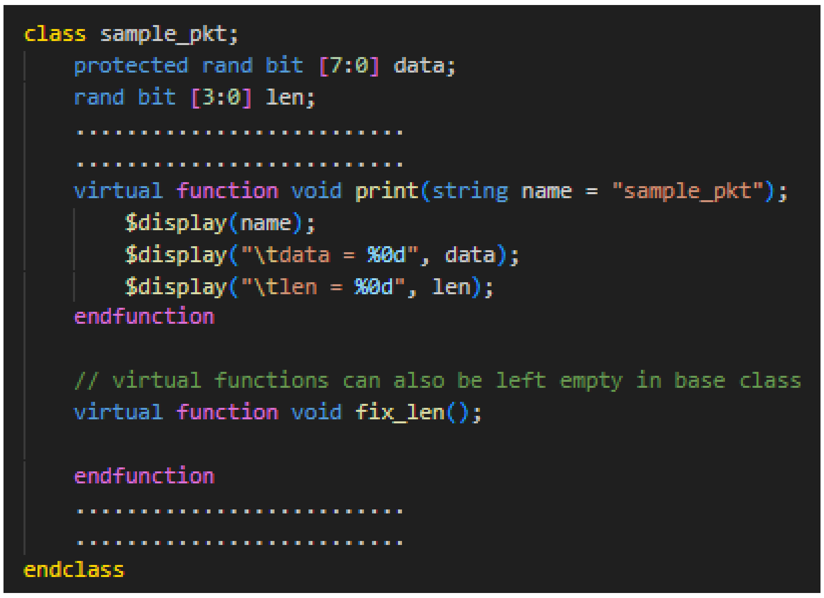
Any data type, including bit, byte, int, arrays, mailbox, events, interfaces, etc., can be passed as arguments to the new constructor.

7. *Polymorphism of Classes*

Polymorphism in classes refers to the ability of a derived class to modify the functionality of a method that has been declared in the base class. However, this feature is only applicable to methods declared as virtual in the base class.

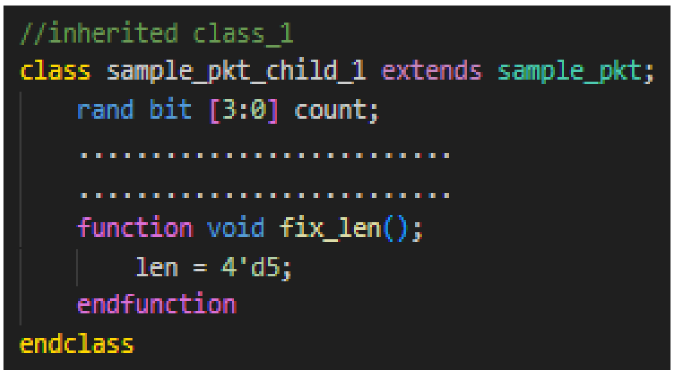
7.1. *Virtual Methods*

virtual methods allow derived classes to override the implementation of a method defined in the base class. For instance, if a base class declares a function as virtual:

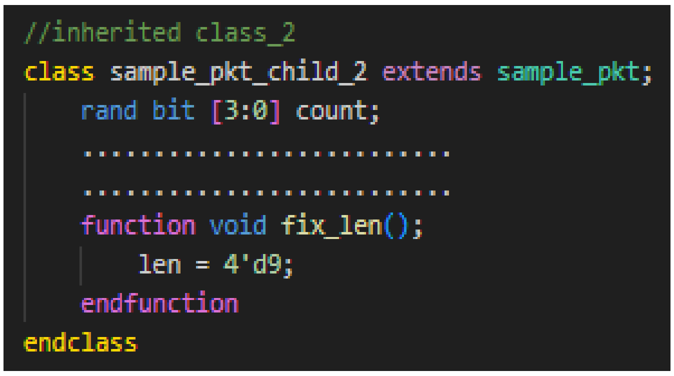


```
class sample_pkt;
    protected rand bit [7:0] data;
    rand bit [3:0] len;
    .........................
    .........................
    virtual function void print(string name = "sample_pkt");
        $display(name);
        $display("\tdata = %0d", data);
        $display("\tlen = %0d", len);
    endfunction

    // virtual functions can also be left empty in base class
    virtual function void fix_len();

    endfunction
    .........................
    .........................
endclass
```

Multiple derived classes from this base class can provide different implementations for the *fix_len* function. This flexibility allows each derived class to tailor the behaviour of the method to its specific requirements. It's considered good practice to declare methods in the base class as virtual, leaving them empty, to enable customization by derived classes.

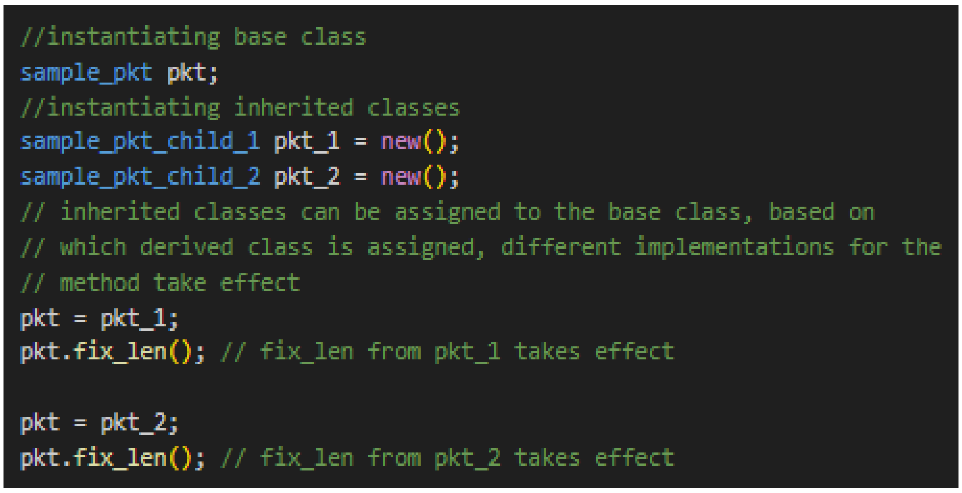


```
//inherited class_1
class sample_pkt_child_1 extends sample_pkt;
    rand bit [3:0] count;
    ..........................
    ..........................
    function void fix_len();
        len = 4'd5;
    endfunction
endclass
```



```
//inherited class_2
class sample_pkt_child_2 extends sample_pkt;
    rand bit [3:0] count;
    ..........................
    ..........................
    function void fix_len();
        len = 4'd9;
    endfunction
endclass
```

In SV, polymorphism is dynamic in nature. It enables the use of a variable of the base class (*superclass*) type to hold objects of derived classes (*subclasses*) and to directly access the methods of these subclasses using the superclass variable.



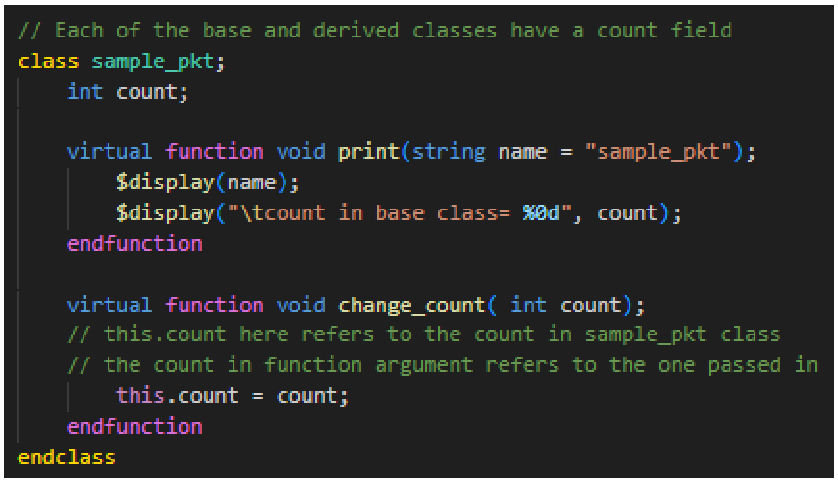
```
//instantiating base class
sample_pkt pkt;
//instantiating inherited classes
sample_pkt_child_1 pkt_1 = new();
sample_pkt_child_2 pkt_2 = new();
// inherited classes can be assigned to the base class, based on
// which derived class is assigned, different implementations for the
// method take effect
pkt = pkt_1;
pkt.fix_len(); // fix_len from pkt_1 takes effect

pkt = pkt_2;
pkt.fix_len(); // fix_len from pkt_2 takes effect
```

When a method of a subclass is accessed using a superclass variable, the compiler, during compile time, is unaware of which specific class will be loaded into the variable. However, at runtime, the appropriate method is accessed based on the actual class instance stored in the variable.

8. this, super

In scenarios where both a base class and a derived class contain a property with the same name, accessing that property directly in the derived class might lead to ambiguity. For instance, if both classes have a property named *count*, accessing *count* in the derived class would refer to the one defined within the derived class itself. However, if we specifically want to access the *count* property from the base class, we can use the super keyword.

super allows users to refer to properties or methods with the same name in both the base and derived classes as needed. this keyword is utilized to reference class properties or methods of the current instance, specifically those inherited from the base class. It becomes necessary to use super when accessing members of a base class that have been overridden in the derived class. In case of multiple levels of inheritance, super refers to the immediate parent (*one level above class*).



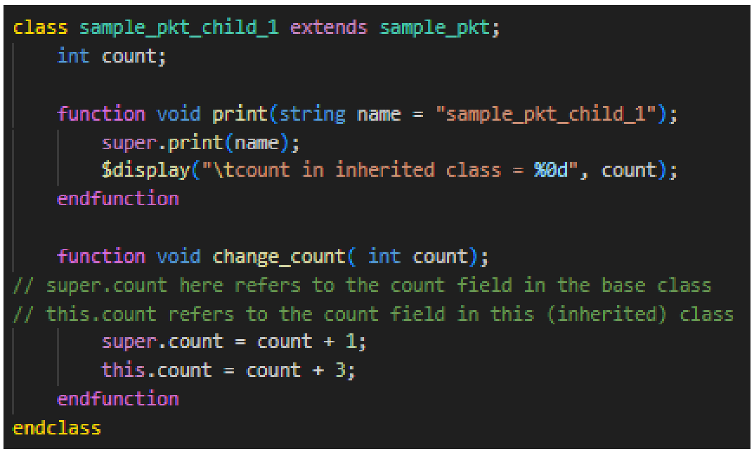
```
// Each of the base and derived classes have a count field
class sample_pkt;
    int count;

    virtual function void print(string name = "sample_pkt");
        $display(name);
        $display("\tcount in base class= %0d", count);
    endfunction

    virtual function void change_count( int count);
    // this.count here refers to the count in sample_pkt class
    // the count in function argument refers to the one passed in
        this.count = count;
    endfunction
endclass
```

From the perspective of the derived class, the syntax would be as follows:

this.count: Refers to the count property in the derived class (*optional to use this here*)

super.count: Refers to the count property in the base class

The choice between super and this depends on the specific context and requirements.



```
class sample_pkt_child_1 extends sample_pkt;
    int count;

    function void print(string name = "sample_pkt_child_1");
        super.print(name);
        $display("\tcount in inherited class = %0d", count);
    endfunction

    function void change_count( int count);
// super.count here refers to the count field in the base class
// this.count refers to the count field in this (inherited) class
        super.count = count + 1;
        this.count = count + 3;
    endfunction
endclass
```

Using *super.method_name()* in a derived class ensures that the functionality available in the base class is inherited while allowing for polymorphism, enabling users to modify or extend the functionality as needed. This syntax can be employed in any derived class method that requires the functionality provided by *method_name()*. It is required that *super.method_name()* is the first line after the method prototype declaration in derived class.

As a best practice, it's recommended to place all common properties and methods in the base class to promote code reusability and maintainability.

9. Class Randomization

Class randomization differs between objects and non-objects.

Non-objects:

For non-objects, such as integers, bytes, and logic, the randomize() function cannot be utilized. Instead, $random, $urandom_range, and $urandom functions are used.

Objects:

On the other hand, for objects, which are instances of a class, the randomize() function is used. The randomize() function returns 0 if randomization fails and 1 if it succeeds.

This function comes with two callback methods: pre_randomize() and post_randomize(). These callback methods are automatically invoked when the main randomize() method is called. Further details on callbacks will be covered in a separate section.

```
// in module
rand_check check = new();
    initial begin
        assert(check.randomize());
    end
```

While *inline constraints* can accomplish tasks that are performed using pre_randomize(), users cannot override the randomize() function. However, SV offers users the option to modify the randomization behaviour by utilizing pre_randomize() and post_randomize().

```
// using inline constraint
assert(check.randomize() with {some_signal inside {[1:5]};});
```

9.1. *Rand Mode*

Rand mode is a feature used to enable or disable randomization for a property. For instance, *addr.rand_mode(0)* disables randomization, while *addr.rand_mode(1)* enables it for a property named *addr*.

```
class rand_check;
    randc bit [2:0] addr;
endclass

module top;
    rand_check check = new();
    initial begin
        assert(check.randomize());
        // disabling randomization
        check.addr.rand_mode(0);
        // enabling randomization
        check.addr.rand_mode(1);
    end
endmodule
```

9.2. *Assert for Randomization*

Assert statements provide a compact way of checking conditions, similar to using if conditions. The syntax for assert is simple: assert(check_condition). If the check condition evaluates to false, the assert statement triggers an error message. In contrast, if an if statement is used for checking conditions, the user needs to manually include display statements to indicate success or failure.

Assert statements can be used anywhere in the code where if statements are valid. When combined with the randomize() function, assert ensures that an object's fields declared as rand or randc undergo proper randomization.

Here's an example comparing assert with if statements for randomization

We'll discuss assertions in more detail in a later section.

```
assert(class_instance.randomize());
// Without assert:
if (class_instance.randomize())
    $display("Randomization passed");
else
    $display("Randomization failed");
```

10. Array of Class

For arrays containing dynamic elements, such as a class, the allocation process involves two steps: first, allocating memory for the array itself if it's a dynamic array, and second, allocating memory for each dynamic element within the array.

Consider the example of a *packet class* declared as *packet* pkt_array[]:

pkt_array is allocated memory by doing: `pkt_array = new[10];`

However, this only allocates memory for the array, not the individual packet elements. To allocate memory for each packet, a loop is used:

```
foreach (pkt_array[i]) pkt_array[i] = new();
// new[] : For arrays
// new() : For class objects
```

Here, memory for each packet is explicitly allocated using new().

11. Null Check on Object Handle

Null check on object handles is crucial for ensuring that memory has been properly allocated before accessing it. Without this check, attempting to access uninitialized memory can result in errors such as "*Bad handle or reference*."

During debugging processes, null checks play a significant role in identifying and resolving issues. When a simulation fails due to a *bad handle or reference error*, the simulator provides the line number and file name where the error occurred, simplifying the debugging process.

Here's an example of how to perform a null check:

```
some_pkt pkt;
initial begin
    if (pkt == null) begin
        $error ("Class object has not been allocated memory");
        pkt = new(); //assigning memory if not assigned
    end
end
```

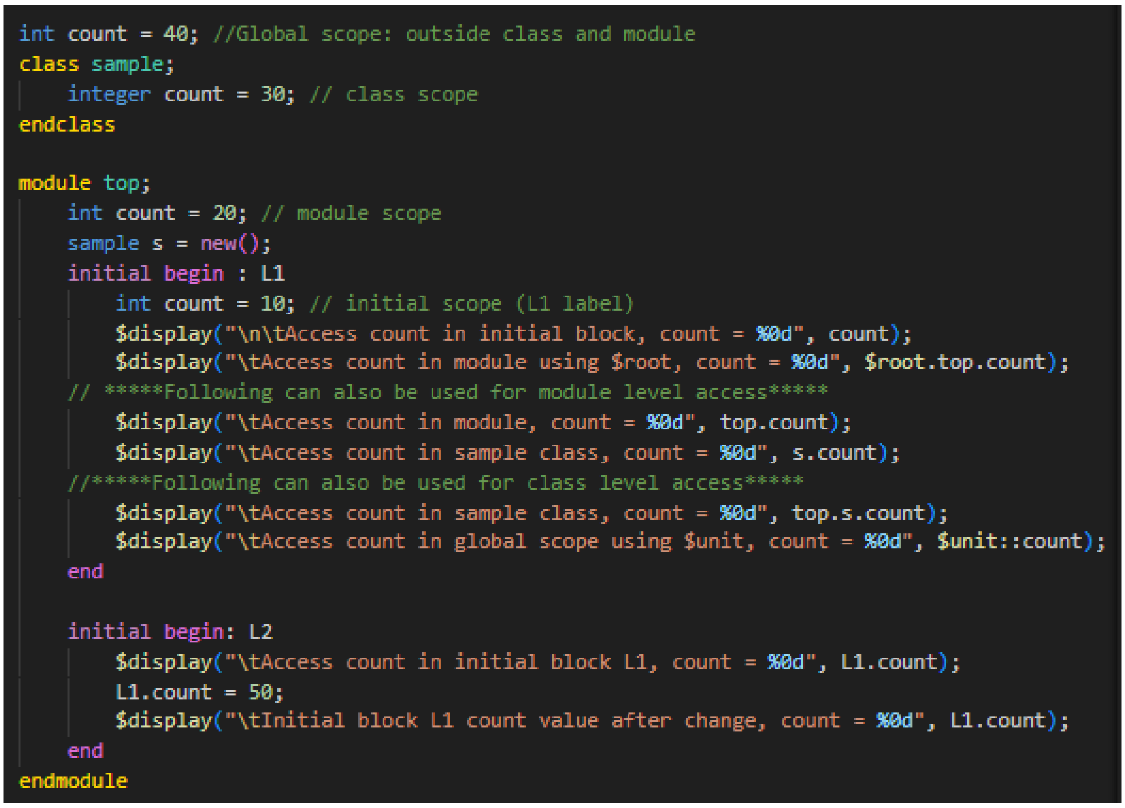
12. Variable Scope

Variable scope encompasses three distinct types: global scope, module/class scope, and method/process scope.

In the global scope, variables, tasks, and functions can be declared freely. To reference variables within the global scope, utilize the syntax $unit::variable_name. Here, $unit refers to declarations at the outermost level of the compilation unit. Tasks and functions can also be declared in global scope, accessible using $unit.

For accessing variables in hierarchies beyond the global scope, use the syntax $root.module_or_class_name.variable. Here, $root signifies the top-level instance of the design or test environment.

In referencing a variable or method, precedence is given to the closest declared variable or method.



```
int count = 40; //Global scope: outside class and module
class sample;
    integer count = 30; // class scope
endclass

module top;
    int count = 20; // module scope
    sample s = new();
    initial begin : L1
        int count = 10; // initial scope (L1 label)
        $display("\n\tAccess count in initial block, count = %0d", count);
        $display("\tAccess count in module using $root, count = %0d", $root.top.count);
    // *****Following can also be used for module level access*****
        $display("\tAccess count in module, count = %0d", top.count);
        $display("\tAccess count in sample class, count = %0d", s.count);
    //*****Following can also be used for class level access*****
        $display("\tAccess count in sample class, count = %0d", top.s.count);
        $display("\tAccess count in global scope using $unit, count = %0d", $unit::count);
    end

    initial begin: L2
        $display("\tAccess count in initial block L1, count = %0d", L1.count);
        L1.count = 50;
        $display("\tInitial block L1 count value after change, count = %0d", L1.count);
    end
endmodule
```

The previous section provided an introduction to classes, delving into the implementation of object-oriented concepts, randomization, and method usage, covering the fundamental aspects of Object-Oriented Programming (*OOP*) in SystemVerilog. Building upon this foundation, the current section shifts focus to exploring advanced topics such as various class types, their applications and object copying techniques.

1. Forward Declaration of Classes

In certain scenarios, it becomes necessary to declare a class variable before the actual class definition is available. For instance, in the file where a top-level module is present, all required files for the testbench need to be included. If these files are not included in the correct order, errors may ensue. Forward declaration of classes serves as a preventive measure against such errors. It finds use in scenarios requiring a class declaration that has not yet been fully defined or implemented.

The syntax for forward declaration is as follows:

```
typedef class class_name;
```

When employing forward declaration prior to including files, it mitigates errors arising from incorrect file order. Essentially, it declares the class, and the class obtains its definition as the compilation progresses to the line that includes the respective file. This approach prevents potential errors stemming from incorrect file sequencing.

2. extern Method

An extern method refers to a function (*or task*) prototype specified within a class, with the actual implementation residing outside the class. They are also called as out-of-block declarations. The declaration of such a method outside the class involves indicating the class to which the method belongs using a scope resolution operator. Out of block declarations using extern enhances code readability.

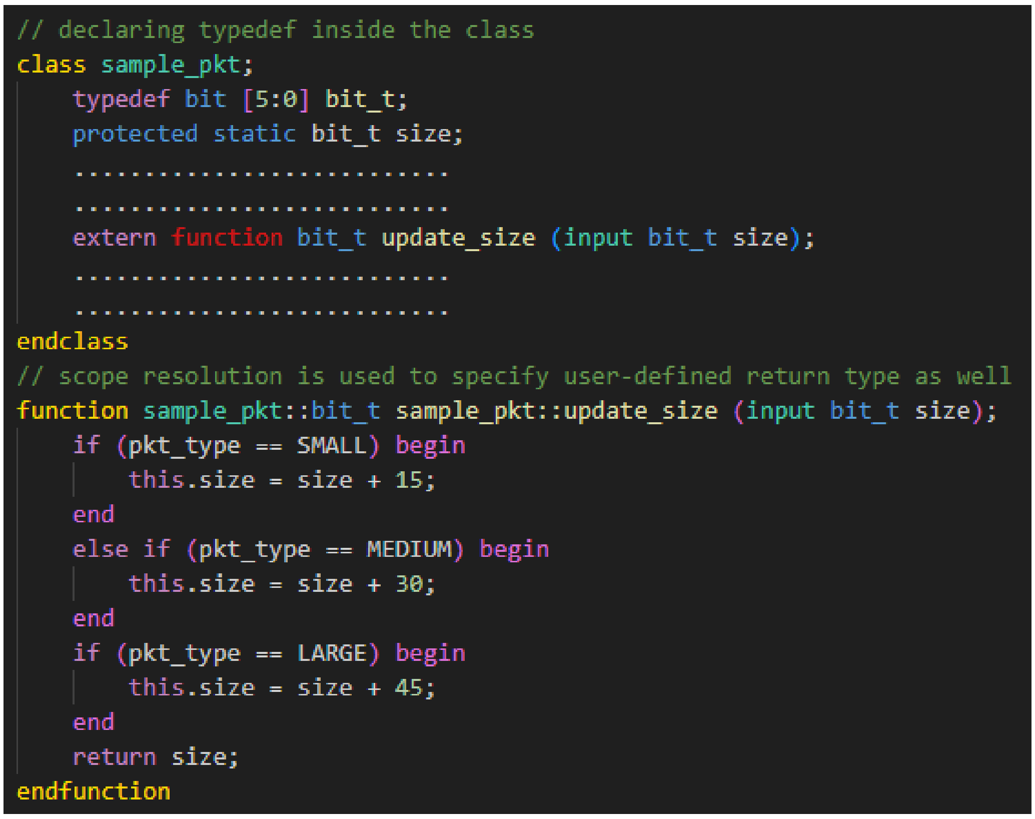
For example: Consider a *sample_pkt* class as shown here.

```
// defining method prototype inside the class
class sample_pkt;
    ............................
    ............................
    extern function void copy(output sample_pkt pkt);
    ............................
    ............................
endclass
// outside the class, using scope resolution
// operator to denote the class the function belongs to.
function void sample_pkt::copy(output sample_pkt pkt);
    pkt = new();
    pkt.addr = addr;
    pkt.len = len;
    pkt.count = count;
endfunction
```

It is important to note that when the return type of a method is user-defined and declared within the class, defining the method externally necessitates specifying the return type with the class and scope resolution operator. However, since external methods have access to class properties and declarations, method arguments do not require the use of class scope resolution.

Moreover, it's crucial to declare typedef definitions before the function prototype to prevent unidentified type errors.

For example:



```
// declaring typedef inside the class
class sample_pkt;
    typedef bit [5:0] bit_t;
    protected static bit_t size;
    ...........................
    ...........................
    extern function bit_t update_size (input bit_t size);
    ...........................
    ...........................
endclass
// scope resolution is used to specify user-defined return type as well
function sample_pkt::bit_t sample_pkt::update_size (input bit_t size);
    if (pkt_type == SMALL) begin
        this.size = size + 15;
    end
    else if (pkt_type == MEDIUM) begin
        this.size = size + 30;
    end
    if (pkt_type == LARGE) begin
        this.size = size + 45;
    end
    return size;
endfunction
```

3. Constant Class Property

A constant class property is a variable declared within a class whose value remains fixed throughout the simulation. Such properties can be defined as read-only by declaring them as const. Given the dynamic nature of class objects, there exist two types of read-only variables: *global constants* and *instance constants*.

Global constant class properties are initialized with a fixed value as part of their declaration. Similar to other const variables, they cannot be assigned a value anywhere other than in their declaration.

On the other hand, instance constants do not include an initial value in their declaration, only the const qualifier. However, they can be assigned a value at runtime, but this assignment can only occur once within the corresponding class constructor (*new()*).

Example:

For each class instance, the *size* is specified during object creation and remains fixed until the end of simulation or object deletion.

```
class sample;
// Global constant:
    const int count = 2;
// Instance constant:
    const int size;
    function new(int size);
        this.size = size;
    endfunction
endclass
// In module
    sample s1 = new(4);
// sets the size field to 4
```

4. *pure virtual Methods*

Pure virtual methods consist solely of a prototype without an implementation. Their purpose is to serve as placeholders within a base class, compelling derived classes to implement them. These methods are denoted with the keyword pure virtual and can only be used in abstract classes.

```
// inside abstract class
pure virtual function void print(string name);
pure virtual function void fix_len();

//inside inherited class 1
function void print(string name = "sample_pkt_child_1");
    $display(name);
    $display("\tdata = %0d", data);
    $display("\tlen = %0d", len);
endfunction

function void fix_len();
    len = 4'd5;
endfunction

//inside inherited class 2
function void print(string name = "sample_pkt_child_2");
    $display(name);
    $display("\tcount = %0d", count);
endfunction

function void fix_len();
    len = $urandom_range(0, 4);
endfunction
```

5. *Abstract Class*

In scenarios where a group of classes share properties and methods, a common base class can be created to encapsulate this shared functionality and structure. However, this base class cannot be instantiated directly; rather, it serves as a blueprint to be inherited by other classes. Such classes are called abstract classes. Any class declared as virtual is inherently an abstract class.

The syntax is:
```
virtual class class_name;
```

For example:
```
virtual class sample_pkt;
    protected rand bit [7:0] data;
    protected rand bit [7:0] sa;
    rand bit [3:0] len;
//methods
// if any derved class does not implement the
// following methods, it leads to errors
    pure virtual function void print(string name);
    pure virtual function void fix_len();
//constraints
...........................
...........................
endclass
```

It's important to note that declaring a class as virtual doesn't directly relate to polymorphism. Instead, an abstract class signifies a class to which memory allocation should not be directly assigned.

Virtual classes cannot be utilized directly; they must be inherited by another class. The subclasses derived from these virtual classes can then be instantiated if they are not marked as virtual.

It's crucial to distinguish between virtual classes and virtual methods. While virtual classes denote abstract classes, virtual methods refer to a different concept. As a best practice, all tasks and functions within virtual classes should typically be declared as virtual.

6. Parameterized Classes

It's often advantageous to define a generic class that can be instantiated with varying array sizes or data types. This concept, known as parameterization, facilitates code reuse across different requirements, fostering flexibility and efficiency.

The syntax for parameterizing classes involves using the # symbol, similar to Verilog. Classes can be parameterized with any data type available in SV, enabling type parameterization.

Syntax:
```
class class_name #(parameters);
```

While Verilog only supports value parameterization, SV classes offer the flexibility of parameterization for both data type and value.

For example:
```
// value parameterization
class lifo_fifo #(byte DEPTH = 8);
// type parameterization
class lifo_fifo #(type DT = int);
```

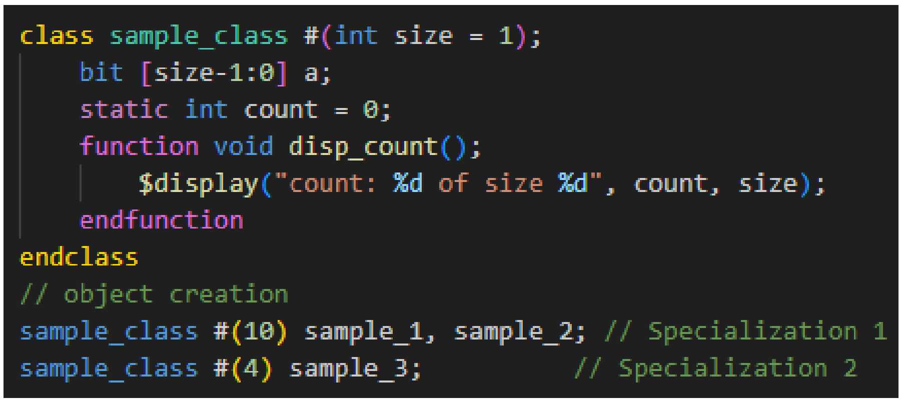
Parameterization significantly enhances code reusability. Depending on specific requirements, classes can be parameterized with either value or type parameters. For value parameterization, any data type such as bit, byte, int, or string can be used. For type parameterization, the keyword type is required. Parameterized classes can also be inherited and utilized as needed.

6.1. Parameterized Classes Specialization

Specializing a class by passing unique parameter values results in a tailored version of the class. While the fundamental structure of the class remains unchanged, each specialization possesses overridden parameters, allowing for customization.

Notably, each specialization of a class maintains its own distinct set of static member variables. However, to share static member variables across multiple class specializations, they must be placed within a non-parameterized base class.

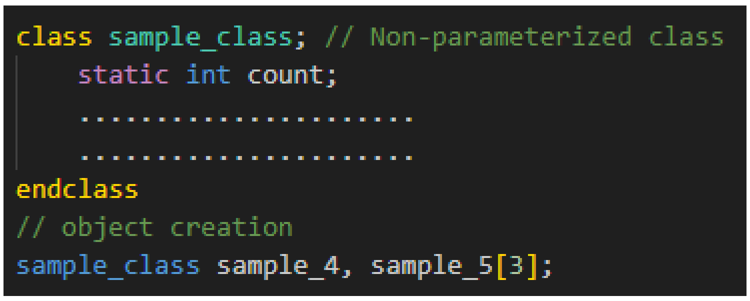
Consider the following example:



```
class sample_class #(int size = 1);
    bit [size-1:0] a;
    static int count = 0;
    function void disp_count();
        $display("count: %d of size %d", count, size);
    endfunction
endclass
// object creation
sample_class #(10) sample_1, sample_2; // Specialization 1
sample_class #(4) sample_3;          // Specialization 2
```

In this example, each specialization of the class *sample_class* possesses its own unique copy of the *count* variable. Despite sharing the same base class structure, classes with different specializations are effectively treated as distinct entities. For instance, *sample_1* and *sample_2* share the same *count* variable since they have the same specialization while *sample_3* has its own *count* variable.

To illustrate shared static member variables among multiple class specializations, consider the following non-parameterized base class:



```
class sample_class; // Non-parameterized class
    static int count;
    ......................
    ......................
endclass
// object creation
sample_class sample_4, sample_5[3];
```

Here, *sample_4* and *sample_5[3]* share a common memory for the *count* variable, as it resides within a non-parameterized class.

6.2. *Parameterized Classes with Inheritance*

There exist four distinct combinations for parameterization with inheritance, each serving a specific purpose:

6.2.1. Non-parameterized class extended from a non-parameterized class

- Both the derived and base classes lack parameterization.
- Typically utilized when polymorphism isn't required.

```
class derived_class extends base_class;
```

6.2.2. Parameterized class extended from a non-parameterized class

- Useful for sharing a static variable among all class specializations.
- Facilitates the creation of a derived class with required parameters from a base class with no parameters.

```
class derived_class #(int size = 15) extends base_class;
```

6.2.3. Non-parameterized class extended from a parameterized class

- Employed when further parameterization is unnecessary.

```
class derived_class extends base_class #(type DT = int);
```

6.2.4. Parameterized class extended from a parameterized class

- Utilized to introduce additional parameters to a parameterized base class without altering its prototype or functionalities.
- Ensures compatibility by ensuring that the derived class includes both existing and newly declared parameters from the base class.

Declaration Syntax:

```
class derived_class #(new_parameters) extends base_class #(existing_parameters);
```

For existing parameters, both read by name and read by position methods are applicable.

```
// base class
class base_class #(byte DEPTH = 8, type DT = int);
// derived class with additional parameters
class derived_class #(byte DEPTH = 16, type DT = byte,
string name = "default name") extends base_class #(DEPTH, .DT(DT));
                                        // Read by:  position    name
```

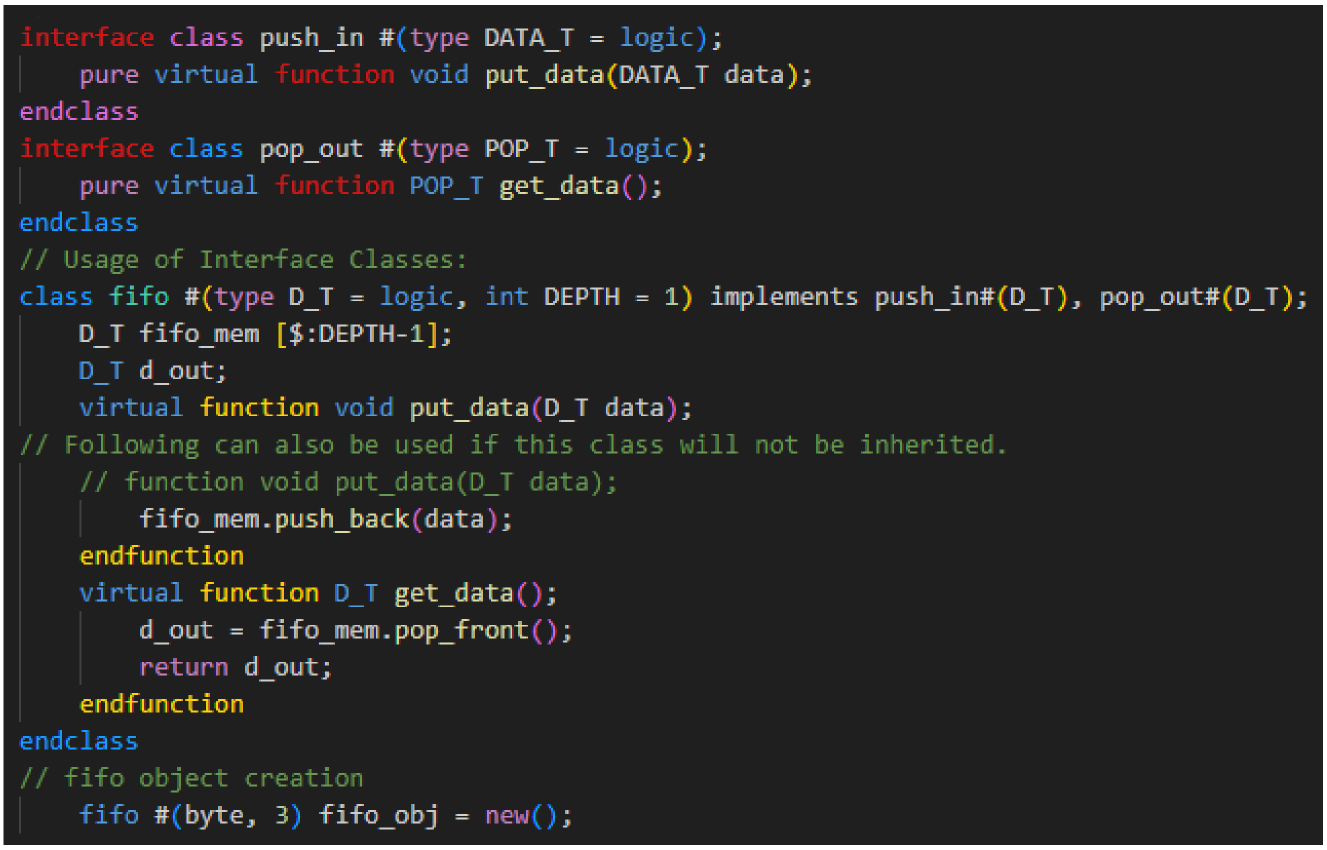
7. *Interface Class*

An interface class is a specialized class that exclusively contains pure virtual tasks/functions and parameter/type declarations. It does not include any other properties or methods within its definition, thereby enforcing a contract-like structure for derived classes.

Notably, constraints or properties cannot be implemented within an interface class, distinguishing it from other class types.

Classes can implement one or more interface classes using the implements keyword (use comma (,) to separate classes that are inherited from). This approach ensures that the derived class must compulsorily define the pure virtual methods specified in the interface class.

For example:



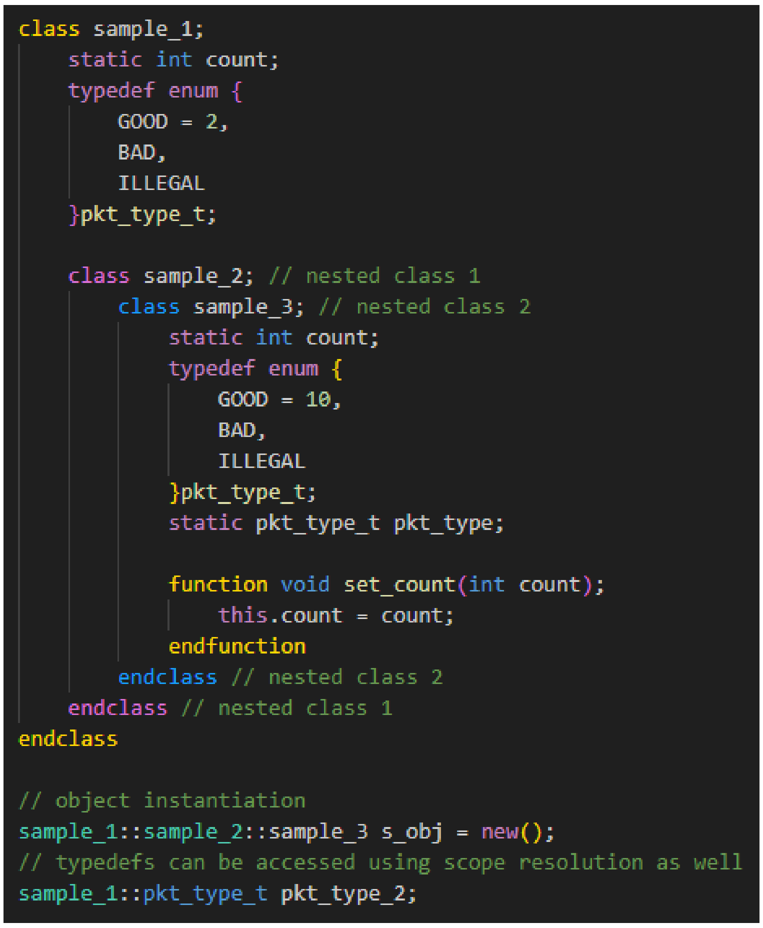
```
interface class push_in #(type DATA_T = logic);
    pure virtual function void put_data(DATA_T data);
endclass
interface class pop_out #(type POP_T = logic);
    pure virtual function POP_T get_data();
endclass
// Usage of Interface Classes:
class fifo #(type D_T = logic, int DEPTH = 1) implements push_in#(D_T), pop_out#(D_T);
    D_T fifo_mem [$:DEPTH-1];
    D_T d_out;
    virtual function void put_data(D_T data);
// Following can also be used if this class will not be inherited.
    // function void put_data(D_T data);
        fifo_mem.push_back(data);
    endfunction
    virtual function D_T get_data();
        d_out = fifo_mem.pop_front();
        return d_out;
    endfunction
endclass
// fifo object creation
    fifo #(byte, 3) fifo_obj = new();
```

In the provided example, the *fifo* class implements both the *push_in* and *pop_out* interface classes, obligating it to define the *put* and *get* methods as per the interface class specifications. This promotes adherence to a predefined contract, enhancing code modularity.

8. *Nested Class*

A nested class refers to a class defined within another class. This nested structure allows for the organization of classes into hierarchical levels, enabling the creation of multiple layers of nesting. Much like modules, classes serve as scopes and can be nested within one another.

This nesting feature provides several advantages, including the ability to conceal local names and allocate resources locally within the nested scope.



```
class sample_1;
    static int count;
    typedef enum {
        GOOD = 2,
        BAD,
        ILLEGAL
    }pkt_type_t;

    class sample_2; // nested class 1
        class sample_3; // nested class 2
            static int count;
            typedef enum {
                GOOD = 10,
                BAD,
                ILLEGAL
            }pkt_type_t;
            static pkt_type_t pkt_type;

            function void set_count(int count);
                this.count = count;
            endfunction
        endclass // nested class 2
    endclass // nested class 1
endclass

// object instantiation
sample_1::sample_2::sample_3 s_obj = new();
// typedefs can be accessed using scope resolution as well
sample_1::pkt_type_t pkt_type_2;
```

9. Object Copying

Object copying encompasses four primary techniques:

1. Copy by Handle

2. Shallow Copy

3. Deep Copy

4. $cast operations

9.1. Copy by Handle

This technique involves copying objects by reference, where the destination object merely points to the same memory location as the source object. As a result, no new memory allocation occurs for the destination object, and both objects share the same underlying data. Consequently, any modifications made to one object will be reflected in the other. This approach is ideal when the intention is for both objects to reference the same data.

However, it's important to note the drawback: since both objects share the same memory location, changes made to one object affect the other.

Example:

```
// object instantiation
sample_class s1, s2 = new();
s1 = s2; // Copy by handle
```

In this example, *s1* and *s2* refer to the same memory location, allowing both objects to access and modify the shared data.

9.2. Shallow Copy

Shallow copy involves the creation of a new memory allocation for the destination object, wherein all fields of the source object are duplicated into the destination object. However, only non-object properties are copied to the new memory location.

During a shallow copy, if another object is present inside the source object, only the object handle is copied, not the internal values, resulting in both objects referencing the same memory location, similar to copy by handle.

It is done as follows:

```
destination_handle = new source_handle
```

For non-object fields, different memory locations are utilized, ensuring independence between objects. However, for object fields, the same memory is shared between the source and destination objects after the copy.

It's important to note that shallow copy has limitations. It's only effective when there are no objects below the class hierarchy. If there are objects nested within the class, shallow copy fails as it copies object handles only, leading to shared memory between instances even after the shallow copy operation.

```
class sample;
    int count;
endclass
class sample_2;
    sample s = new();
    .......................
    .......................
endclass
// object instantiation
sample_2 s1 = new();
sample_2 s2 = new();
s2 = new s1; //shallow copy
//changing count value for s1
s1.s.count = 50;
// count will reflect in both s1
// and s2 since its an object field
```

Drawbacks:

Limited applicability when objects are nested within the class hierarchy.

Objects below the class level are copied by handle, resulting in shared memory between instances even after shallow copying.

9.3. Deep Copy

Deep copy involves a user-defined implementation of copying, wherein each field from the source object is replicated into the destination object. Notably, new memory allocations are created for objects nested within the class hierarchy.

The copy function can be implemented in various ways, offering flexibility in usage:

1. Using Method Call

source_handle.copy(destination_handle);

```
class sample_class;
    sample_2 s_2 = new();
    function void copy (output sample_class s_inst);
        s_inst.s_2 = new(); // memory allocation for objects
        s_inst.s_2.data = s_2.data;
        s_inst.size = size;
        s_inst.addr = addr;
    endfunction
endclass
// object instantiation
sample_class s1 = new();
sample_class s2 = new();
s1.copy(s2); //deep copy (s1 gets copied to s2)
```

2. Using Assignment

destination_handle = source_handle.copy();

```
class sample_class;
    sample_2 s_2 = new();
    function sample_class copy ();
        sample_class s_inst;
        s_inst.s_2 = new(); // memory allocation for objects
        s_inst.s_2.data = s_2.data;
        s_inst.size = size;
        s_inst.addr = addr;
        return s_inst;
    endfunction
endclass
// object instantiation
sample_2 s1 = new();
sample_2 s2 = new();
s2 = s1.copy();
```

Deep copy can also leverage shallow copy techniques. In cases where objects exist below the class hierarchy, shallow copy is performed for those objects as well to ensure independent memory allocation.

Example:

```
class sample_class;
    sample_2 s_2 = new();
    function void copy (output sample_class s_inst);
        //deep copy is multiple levels of shallow copy
        s_inst = new this; // shallow copy for non-objects
        s_inst.s_2 = new this.s_2; //shallow copy for objects
    endfunction
endclass
// object instantiation
sample_class s1 = new();
sample_class s2 = new();
s1.copy(s2); //deep copy (s1 gets copied to s2)
```

In this implementation, the copy function replicates each field (*non-object*) from the source object into the destination object. Similarly, for class objects (*e.g., sample_2*), a shallow copy is employed allocating memory to object instances.

9.4. Casting

Casting involves converting one variable from one data type into another format. This process can only be applied to singular variables; arrays or other complex data structures are not singular.

For example:

```
int a; /*singular,*/ int b[4:0]; //not singular
```

Variables subject to casting can be either objects or non-objects. Non-objects, where the data type is fully known at compile time, are subject to static casting. On the other hand, objects, where the data type is determined at runtime, are subject to dynamic casting. Objects can be of any class type (including base or derived classes). Hence, this process is referred to as dynamic casting.

9.4.1. Static Casting

Static casting involves the syntax datatype '(), allowing for the conversion of data types at compile time. This type of casting is particularly crucial when dealing with typedef enum type conversions.

Example:

```
typedef enum bit [1:0]{
    GOOD, BAD, ILL
} pkt_type_t;
int a;
// incorrect static casting
pkt_type = a;
// correct way of static casting,
// converting 'a' from int to enum type
pkt_type = pkt_type_t'(a);
```

```
int prod;
byte b, c;
// converting byte type to int type
prod = int '( b * c);
```

9.4.2. Dynamic Casting

Dynamic casting involves the transformation of one class into another class using the $cast operator, a built-in SV method. This process enables the conversion of *class_2* into *class_1*.

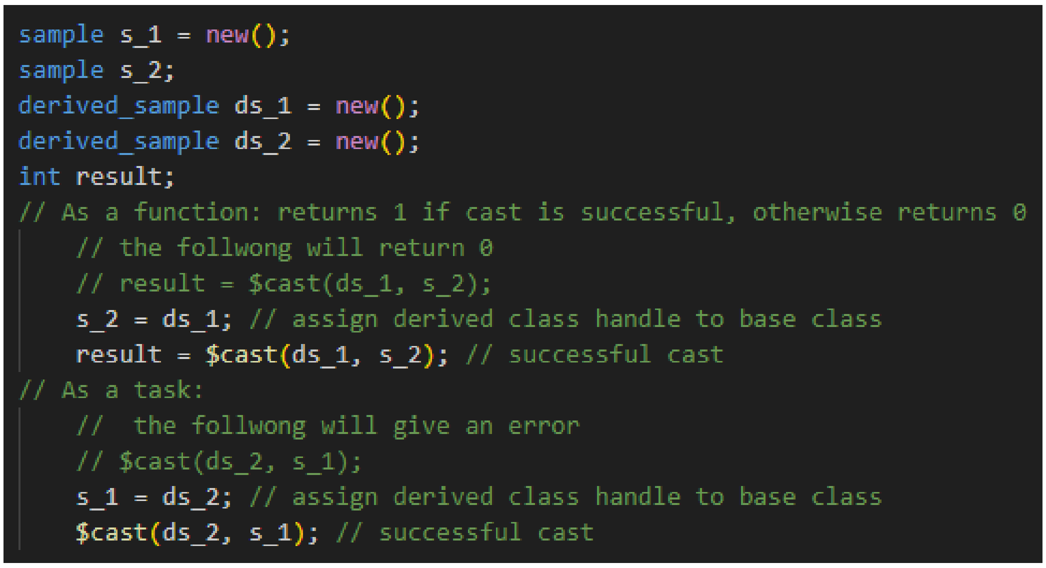
Syntax:

```
$cast(class_1, class_2); // class_2 is casted to class_1
```

$cast functions similarly to copy by handle, allowing for conversion between class types. Both the destination and source objects need not be of the same class handles; they can be related through inheritance.

Furthermore, $cast can behave as both a function and a task depending on its usage. While a base class object cannot be casted from a derived class objects due to potential type mismatches, the reverse operation is possible; derived classes can be casted from base classes since they are related through inheritance. Dynamic casting is successful only if the parent is previously pointing to the same derived object type.

Example:



```
sample s_1 = new();
sample s_2;
derived_sample ds_1 = new();
derived_sample ds_2 = new();
int result;
// As a function: returns 1 if cast is successful, otherwise returns 0
    // the follwong will return 0
    // result = $cast(ds_1, s_2);
    s_2 = ds_1; // assign derived class handle to base class
    result = $cast(ds_1, s_2); // successful cast
// As a task:
    //  the follwong will give an error
    // $cast(ds_2, s_1);
    s_1 = ds_2; // assign derived class handle to base class
    $cast(ds_2, s_1); // successful cast
```