Issue Date: **Monday – October 26, 2020**
Submission Deadline: **Saturday – November 7, 2020 (till 11:59 pm)**

Instructions!

1. This is a **graded** assignment so you are strictly **NOT** allowed to discuss your solutions with your fellow colleagues. You can **ONLY** discuss with your TAs or with me.
2. Strictly follow good coding conventions (commenting, meaningful variable and functions names, properly indented and modular code.
3. I hope you have learned the definition of cheating by now, so avoid any means of cheating.
4. Hard **DEADLINE** of this assignment is **Saturday, November 7, 2020**. No late submissions will be accepted after due date and time so manage emergencies beforehand.

Sets [40 Marks]

A set is a collection of elements or numbers or objects, represented within the curly brackets { }. For example: {1,2,3,4} is a set of numbers. You are required to implement a class to exhibit set of numbers.

```
class Set
{
   private:
       int size;
       int* arr;
       int noOfElements;
       void reduce_size();         // Reduce allocated memory by half
       void increase_size();       // Increase allocated memory by twice
       bool isEmpty();             // return true if the set is empty, else return false
       bool isFull();              // return true if the set is full. Else return false.

   public:
       Set();                      //default constructor
       Set(int size);              // allocates the memory on heap and check if the size is invalid
                                      then set null
       Set(const Set& ref);        // Copy constructor
       bool insertElement(int value); // This function should insert value at the most recent
                                       position. Add necessary checks and increase memory if
                                       required. Returns true if the value is inserted else returns
                                       false.
       bool removeElement(int value); // find the given value and remove the value from set. Decease
                                          the set size (if required). Return true if the value is
                                          removed else return false the value is not present in set.
       int searchElementPosition(int value); // if the element is present then return the index of
                                                 that element else return -1
       bool searchElement(int value);   // return true if the value is present in the set else
                                           returns false.
       void displaySet();              // display the contents of set. (in curly brakets and coma
                                          separated)
       Set intersection(Set a);       // return the intersection of the sets.
                                      /*
                                         Example 1:
                                                Arr = {1,2,3}
                                                a = {1,3}
                                                resultSet = {1,3}
                                                return resultSet
                                         Example 2:
                                                Arr = {1,2,3}
                                                a = {4,5}
                                                resultSet will be empty
                                       */
       Set union(Set a);              // return the union of the sets.
                                      /*
```

```
                                           Example 1:
                                                  Arr = {1,2,3}
                                                  a = {1,3}
                                                  resultSet = {1,2,3}
                                                  return resultSet
                                           Example 2:
                                                  Arr = {1,2,3}
                                                  a = {4,5}
                                                  resultSet = {1,2,3,4,5}
                                         */
        Set operator+ (Set& first);      // You may call union function in this function.
        Set difference(Set a);          // return the difference of the sets.
                                        /*
                                           Example:
                                                  Arr = {1,2,3,4}
                                                  a = {2,3,5}
                                                  resultSet = {1,4}
                                                  return resultSet
                                        */
        Set operator-(Set& first);      // You may call difference(Set a) function in this function.
        int isSubSet(Set a);            // return 1 for proper subset, return 2 for improper subset
                                           return 0 if not a subset.
        ~Set();                  //destructor
        int& operator[](int);   // Directly access element in the set. If element does not exist,
                                  return -1.
};
```

To insert an element in to the set, you should call function *insertElement* that will place value at the most recent position. If the array gets full then resize the array by calling *increase_size* function. This function should return true whenever the value is successfully inserted else return false. Here is an example demonstrating the insert senario:

Suppose, we have a set Arr = {1, 2 ,3}, to insert 4 into the set we call *insertElement(4)* that will return true and will result in the updated set {1, 2, 3, 4}.

Now, we want to add element *2* into the same set. Call to function *insertElement(2)* will return false because 2 is already in set. Recall that set is a collection of distinct values i.e. an element can only appear once.

Note: Any public function can be called after any public function, including the destructor. Your code needs to work without any memory leakages and illegal memory access. Allocate 0/nullptr in the pointer whenever necessary as a marker.

A **proper subset** of a set A is a subset of A that is not equal to A. In other words, if B is a proper subset of A, then all elements of B are in A but A contains at least one element that is not in B.
For example, if A={1,3,5} then B={1,5} is a proper subset of A.
An **improper subset** is a subset containing every element of the original set.
A proper subset contains some but not all of the elements of the original set. For example, consider a set {1,2,3,4,5,6}. Then {1,2,4} and {1} are the proper subset while {1,2,3,4,5} is an improper subset.

Task 02: Type Conversion [40 Marks]

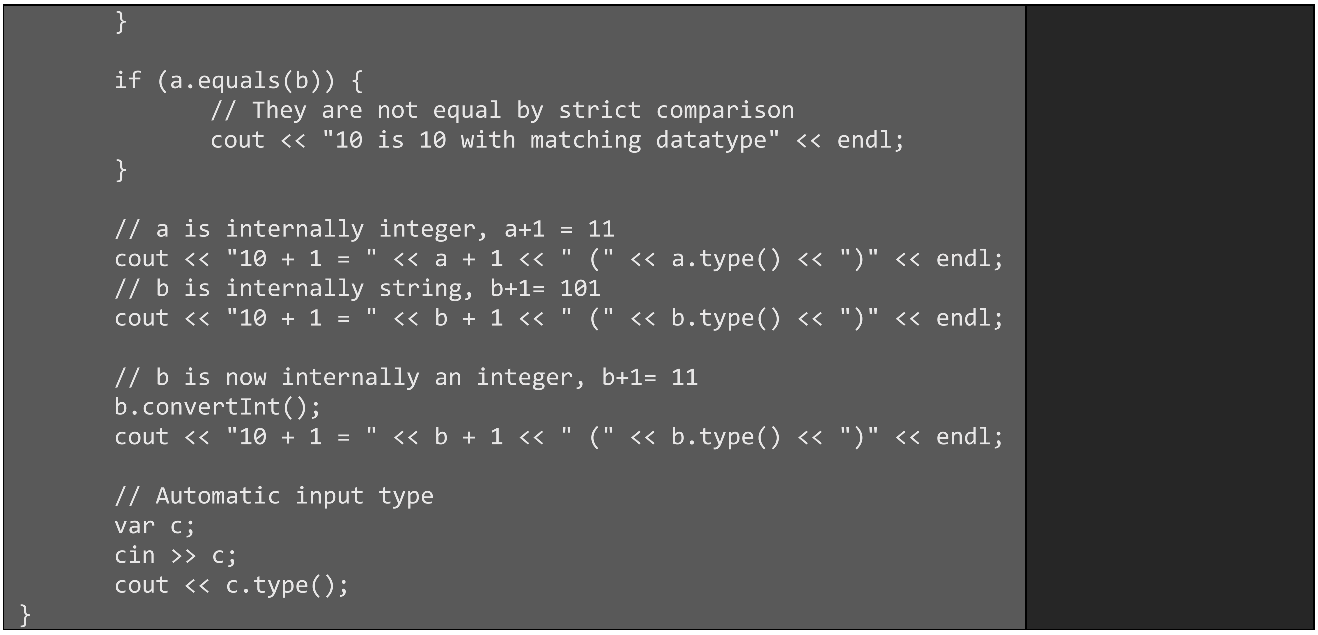
Languages like python, javascript and even the language you used for LARP in PF supported variables that did not have a specific datatype, or rather the datatype was dynamic. Interestingly, this same functionality can be implemented in C++ with the help of operator overloading.

We will create a class var (short for variable) that will change its datatype according to the data it has been given. We will deal with null, int, double and string values (arrays can be implemented too but are beyond the scope)
Here's sample main program and output:

```
int main() {
       var a = 10;
       var b = "10";

       if (a == b) {
              // They are equal by regular comparison
              cout << "10 == 10" << endl;
```

```
10 == 10
10 + 1 = 11 (i)
10 + 1 = 110 (s)
10 + 1 = 11 (i)
a
s
```



```
        }

        if (a.equals(b)) {
                // They are not equal by strict comparison
                cout << "10 is 10 with matching datatype" << endl;
        }

        // a is internally integer, a+1 = 11
        cout << "10 + 1 = " << a + 1 << " (" << a.type() << ")" << endl;
        // b is internally string, b+1= 101
        cout << "10 + 1 = " << b + 1 << " (" << b.type() << ")" << endl;

        // b is now internally an integer, b+1= 11
        b.convertInt();
        cout << "10 + 1 = " << b + 1 << " (" << b.type() << ")" << endl;

        // Automatic input type
        var c;
        cin >> c;
        cout << c.type();
}
```

Motivation for doing this: Imagine making a calculator and someone puts a string where you expect double. We can handle these I/O problems with a dynamic datatype.

We will take 3 data members (double d, int i, string s), They will store the value that we need to store. An additional data member (char dtype) is required, this will tell us WHERE and WHAT our value is (which can be ‘n’ (null), ‘i’ (integer), ‘d’ (double) and ‘s’ (string)).

```
class var {
        double d;
        int i;
        string s;
        char dtype; // Possible values: ‘n’, ‘i’, ‘d’ and ‘s’ (null, int, double, string)

        // conversion from given Double in parameter to String (use as it is)
        string convert(const double a) const
        {
                ostringstream oss;
                oss << a;
                return oss.str();
        }

        // conversion from given Integer in parameter to String (use as it is)
        string convert(const int a) const
                {
                ostringstream oss;
                oss << a;
                return oss.str();
        }

        // Detect the data-type of the given string in parameter (can be i/d/s) (use as it is)
        char whatType(const string a) const
        {
                bool d = 0;
                for (unsigned int i = 0; i < a.length(); i++)
                {
                        if (!(a[i] >= '0' && a[i] <= '9'))
                        {
                                if (a[i] == '.' && d == 0)
                                {
                                        d = 1;
                                }
                                else
```

```
                            {
                                   return 's';
                            }
                     }
              }
              if (d == 0)
                     return 'i';
              return 'd';
       }

       int toInt(const string a) const
       {
                     // Implement/Use any library function to convert strings to integers.
                     // If there are any non-integer characters, ignore them and convert rest
       }
       double toDbl(const string a) const
       {
                     // Implement/Use any library function to convert double to integers.
                     // If there are any non-double characters, ignore them but convert rest
       }
public:
              // Implement public functions as described below

       var()         // default, just do dtype = ‘n’ (NULL value)
       var(const int data)
       {
              i = data;
              dtype = 'i';
       }
       var(const double data) // Store double value on same pattern
       var(const string data) // Store string value
       var(const char* data) // Character array will be converted to string (don’t do any manual
                                 work, s = data is sufficient)
       var(const var& a)    // Copy constructor.

var& operator=(x)
       var& operator=(const int data)
       var& operator=(const double data)
       var& operator=(const char* data)
       var& operator=(const string data)
       var& operator=(const var& data)
       // All of these are single line functions, code is same as constructor
```

```
bool operator==(const var& data) const
```

Compare with another var object. (9 conditions and 19 lines of code if you work smartly)

- If dtype of both objects is integer, return integer comparison (i == data.i)
- If dtype of both objects is double, return double comparison (d == data.d)
- If dtype of both objects is string, return string comparison (s == data.s)
- If one is integer and other is string, use convert( ) on “i” and compare
  e.g. dtype= ’i’ and data.dtype=’s’ => return “data.s == convert(i)”
- Similarly, if data.dtype= ’i’ and dtype=’s’ => return “s == convert(data.i)”
- If one is double and other is string (or vice versa) => return “s == convert(d)” like above
- If both are null, return true else return false

```
bool var(const var& data) const
```

Compare with another var object along with datatype. (5 conditions and 11 lines of code)

- If dtype of both objects is different, return false simply
- If both dtypes are ‘i’ (you may only need to check one of them due to rule 1), return i == data.i
- For double/string(s), use the same above rule 2
- If both are null, return true,
- Anything else: return false

```
bool operator>(const var& data) const
```

Same pattern as ==

```
bool operator<(const var& data) const
```

Same pattern as ==

Other conditional operators
To save time and avoid redundancy, just copy paste this code as it is!

```
bool operator<=(const var& data) const
{
	return (*this) < data || (*this) == data;
}
bool operator>=(const var& data) const
{
	return (*this) > data || (*this) == data;
}
bool operator!=(const var& data) const
{
	return !((*this) == data);
}
```

```
var operator+(const var& data) const
```

Add value to another var object. (8 conditions and 17 lines of code if you work smartly)

- If dtype of both objects is integer, return integer addition (i + data.i)
- If dtype of both objects is double, return double addition (d + data.d)
- If dtype of both objects is string, return string addition (s + data.s)
- If one is integer and other is string (or vice versa) => add “s” to convert(i) and return
- If one is double and other is string (or vice versa) => add “s” to convert(d) and return
- else return var() (var() is an object that represents null)

```
var operator-(const var& data) const
```

Subtract value from another var object. (4 conditions and 9 lines of code if you work smartly)

- If dtype of both objects is integer, return integer subtraction (i - data.i)
- If dtype of both objects is double, return double subtraction (d - data.d)
- If dtype of one is integer and other is double return int/double subtraction (handle opposite as well)
- else return var() (var() calls the constructor to make an object that represents null/invalid)

```
var operator*(const var& data) const
```

Same pattern as -

```
var operator/(const var& data) const
```

Same pattern as –

Note: If `this` has dtype = ‘i’ and data has dtype = ‘d’ => you need to calculate and store the result in this.d and change dtype to ‘d’ because if we do 5/3.5 without changing datatype, we lose the decimal value.
If `this` has dtype = ‘i’ and data has dtype = ‘i’ => datatype needs to be updated **only if** remainder is non zero.

```
char& operator[](const int a)
```

- if dtype is string => simply return s[a]
- if dtype is integer/double => return convert(i/d)[a] (we can get value at the digit position like this but cannot modify)
- in else, simply use the following code to avoid exit(0) operation (we will consider it to be an invalid operation)

  ```
  else { i = 0; return convert(i)[0]; }
  ```

```
var& operator++()
```

Prefix increment: obvious for double and integers. For strings, increment the LAST character by 1. e.g. “ABC” will be “ABD”

```
var& operator++( int)
```

Postfix increment: make a copy of object, use prefix increment, return the copy

```
var& operator--()
```

See Prefix increment

```
var& operator--(int)
```

See Postfix increment

```
char type() const
```

Return dtype

```
string toString() const
```

- If dtype is string, return s
- If dtype is ‘d’ or ’i’, return the convert() of ‘d’ or ’i’
- else (dtype = ’n’ (null) is the only possibility), return “null”

```
double parseDouble() const
```

- If dtype is already integer or double, return as it is
- If dtype is string, use whatType(s) to know if “s” is numeric. Call toInt(s) or toDbl(s) based on result.
- else return 0

```
int parseInt() const
```

To simplify, use this code

```
int parseInt()
{
        return (int) parseDouble();
}
```

```
bool isInt() const
```

Return true if dtype is ‘i’ OR (if dtype is ‘s’ and whatType(s) gives ‘i’), double/null or anything else should give false

```
bool isNumeric() const
```

- dtype is ‘i’ or ‘d’ then return true
- dtype is ‘s’ and whatType(s) is ‘d’ or ‘i’ return true
- return false

```
void convertInt(), void convertDouble(), void convertString()
```

Force the values to be the datatype required. These functions will change the internal data type
Example for convertInt():

- dtype is int, simple return
- dtype is ‘d’ => change dtype to ‘i’, typecast “i = (int) d;”
- dtype is ‘s’ => change dtype to ‘i’, use toInt(s) (toInt() should ignore non-digit characters, if there all characters are not digits, toInt() should give 0)

Other two functions require a similar procedure.

```
void isNull() const
```

dtype == ‘n’;

```
~var()
```

dtype = ‘n’; *(don’t do anything else)*

```
void clear()
```

Call destructor

Making cin/cout work:
This overloading of stream operators (<< and >>) will be discussed later, for now, just paste this code in cpp file.

```
ostream& operator<<(ostream& s, const var& a)
{
        s << a.toString();
        return s;
}
```

```
istream& operator>>(istream& s, var& a)
{
       string temp; s >> temp;
       a = temp;
       // We stored a string, now let us fix the type…
       if (a.isInt())
              a.convertInt();
       else if (a.isNumeric())
              a.convertDouble();
       return s;
};
```

Any fool can write code that a computer can understand
Good programmers write code that humans can understand
-Martin Fowler