**Simple String Template**

**Writer: Sayak Haldar**

**Theory:**

Consider a string of characters. A string is a class that holds characters and provides information such as subscripting, concatenation, and comparison that we usually associate with the notion of string. Now, suppose, we would like to provide behaviour for many kind of characters. For example: string of signed characters, unsigned characters, of greek characters etc. Then, we want to represent the notion of the string with minimal dependance of these characters. The definition of a string relies on the fact that a character can be copied, and little else. Thus, we can make a more general string type by taking the string of char and making the character type of a parameter.

template<class C> class String

{

struct Srep;

Srep \*rep;

public:

String();

String(const C\*);

String(const String &);

C read(int i)const;

};

The template <class C> prefix specifies that a template is being declared and that a type argument C will be used in the declaration. After it’s introduction, C is used like exactly other type names, the scope of C extends to the end of the declaration prefixed by template <class C>. Note that, template <class C> says that C is a type name, it need not be the name of the class.

Now,

If we want to define a String of signed char or normal char (since, normal one is signed)

**int main()**

**{**

**String<char> buf;**

**map<String<char>,int> m;**

**while(cin>buf) m[buf]++;**

**//write out result**

**}**

The version of our Japanese character type Jchar would be:

**int main()**

**{**

**String<Jchar> buf;**

**map<String<Jchar>,int> m;**

**while(cin>buf) m[buf]++;**

**//write out result**

**}**

**Defining A Template:**important point from this part is: a function template or a template function can be overloaded, but a class template cannot be overloaded in the same scope.