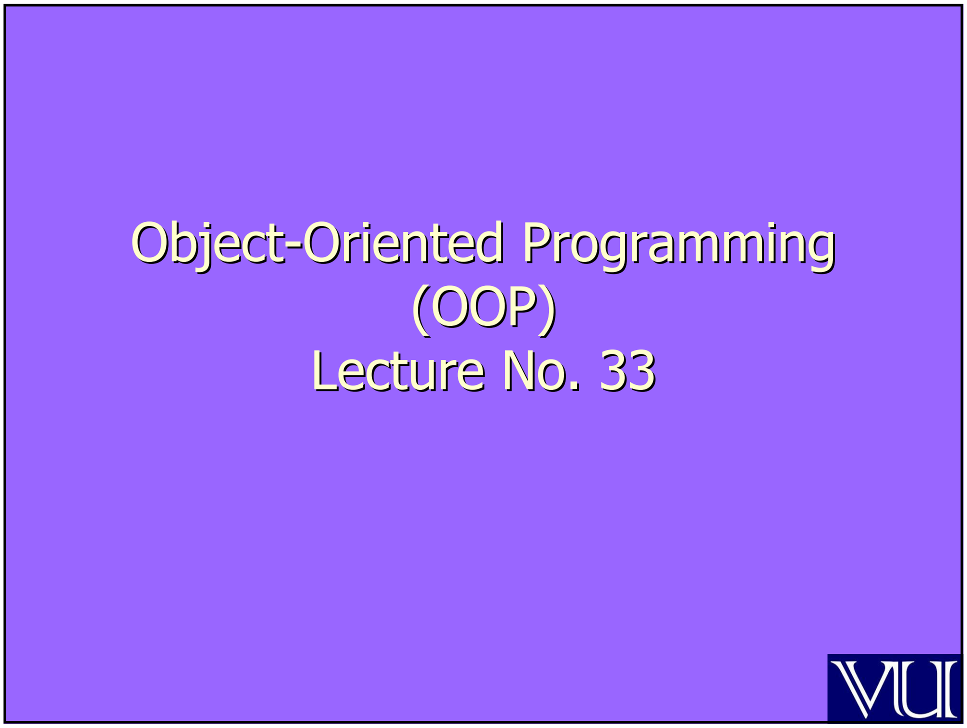
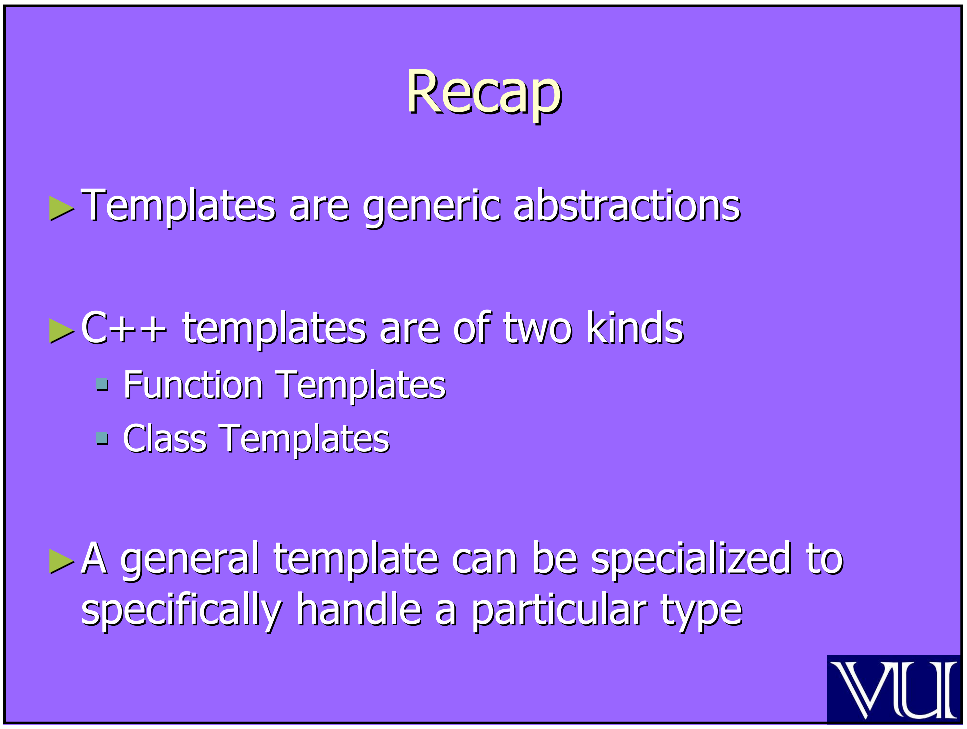


Object-Oriented Programming (OOP)
Lecture No. 33



Recap

- Templates are generic abstractions
- C++ templates are of two kinds
  - Function Templates
  - Class Templates
- A general template can be specialized to specifically handle a particular type

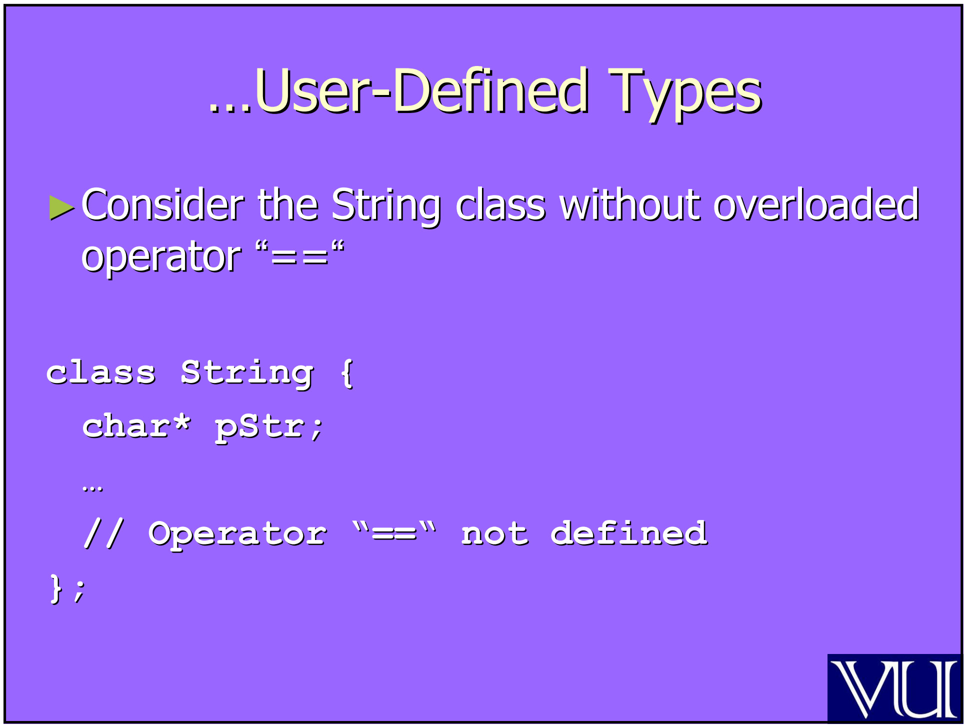
Multiple Type Arguments

```
template< typename T, typename U >
T my_cast( U u ) {
  return (T)u;
}

int main() {
  double d = 10.5674;
  int j = my_cast( d );    //Error
  int i = my_cast< int >( d );
  return 0;
}
```

User-Defined Types

- Besides primitive types, user-defined types can also be passed as type arguments to templates

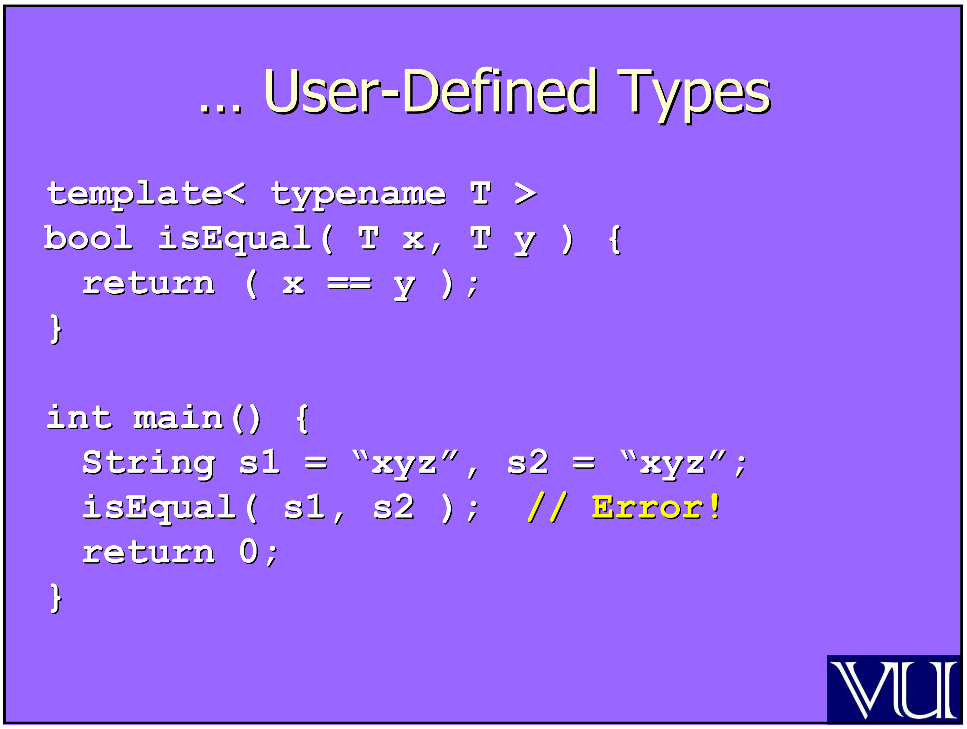
- Compiler performs static type checking to diagnose type errors



…User-Defined Types

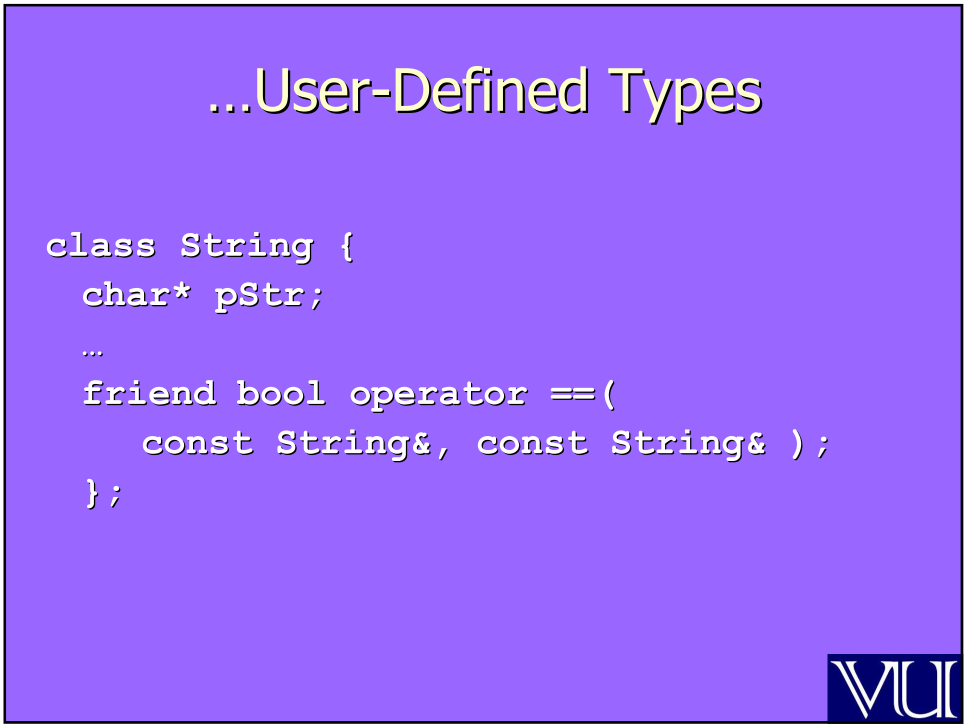
- Consider the String class without overloaded operator “==“

```
class String {
  char* pStr;
  …
  // Operator “==“ not defined
};
```



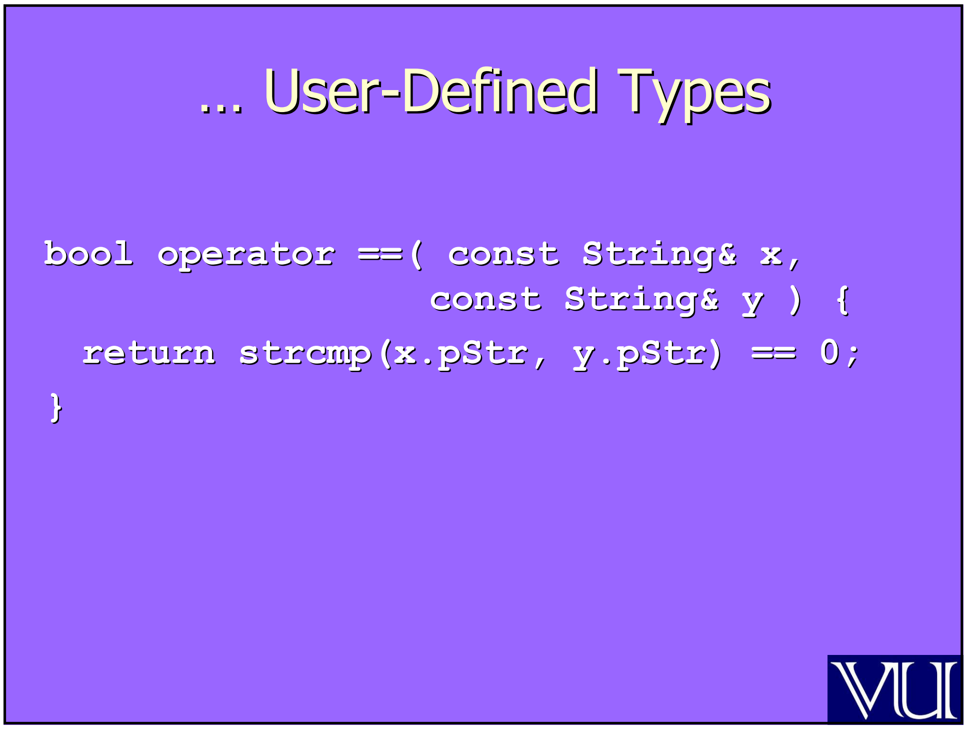
… User-Defined Types

```
template< typename T >
bool isEqual( T x, T y ) {
  return ( x == y );
}

int main() {
  String s1 = “xyz”, s2 = “xyz”;
  isEqual( s1, s2 );  // Error!
  return 0;
}
```



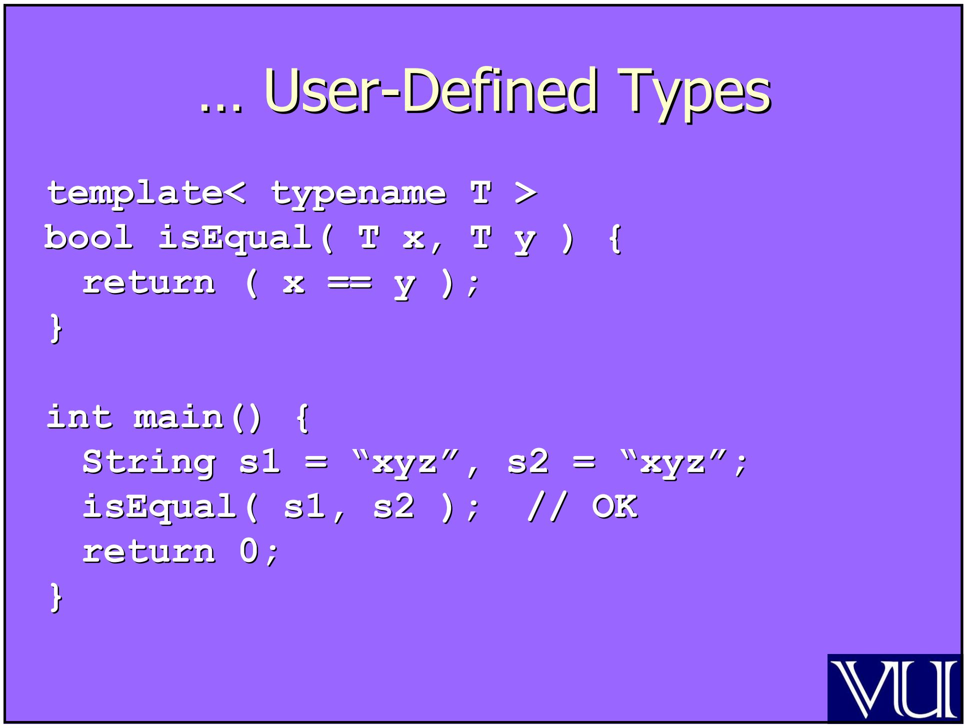
…User-Defined Types

```
class String {
  char* pStr;
  …
  friend bool operator ==(
    const String&, const String& );
  };
```



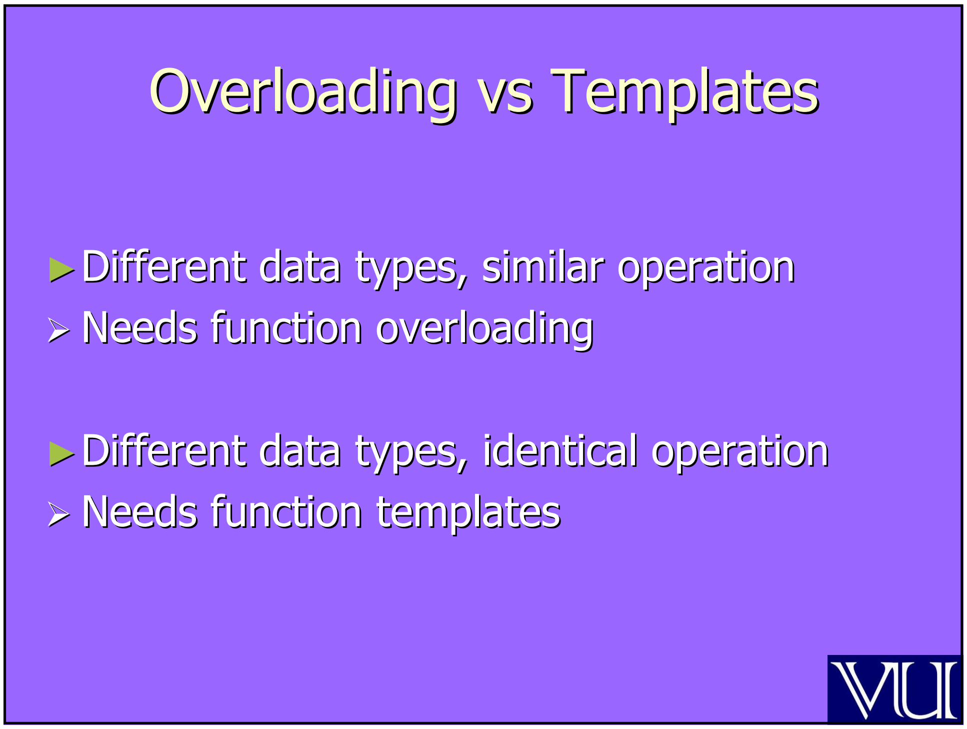
… User-Defined Types

```
bool operator ==( const String& x,
                 const String& y ) {
  return strcmp(x.pStr, y.pStr) == 0;
}
```



… User-Defined Types

```
template< typename T >
bool isEqual( T x, T y ) {
  return ( x == y );
}

int main() {
  String s1 = “xyz”, s2 = “xyz”;
  isEqual( s1, s2 );  // OK
  return 0;
}
```



Overloading vs Templates

▶ Different data types, similar operation

➢ Needs function overloading

▶ Different data types, identical operation

➢ Needs function templates

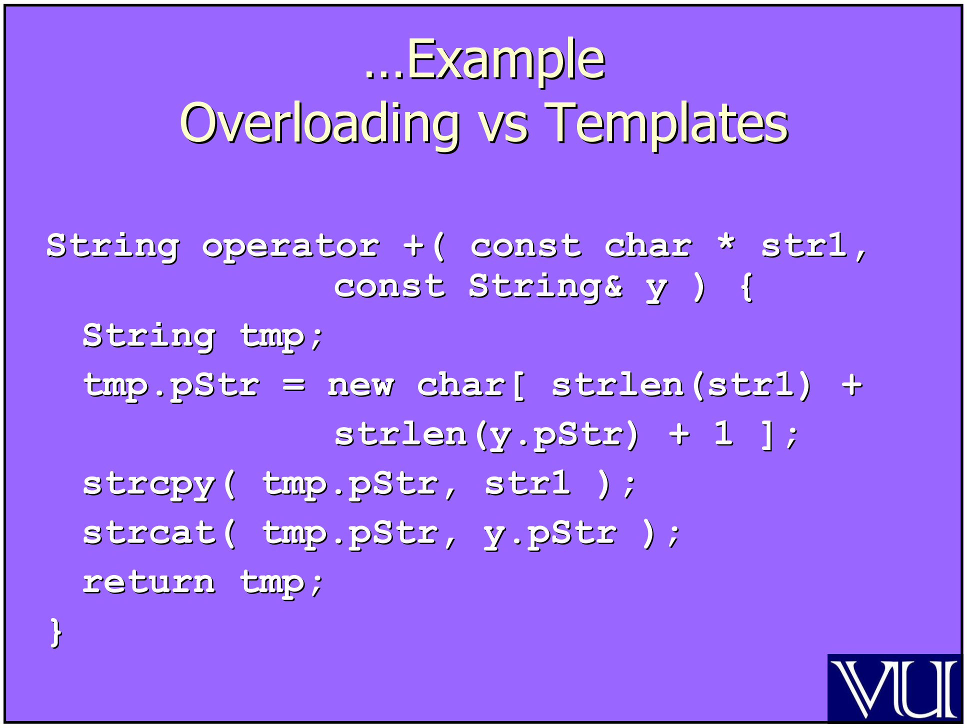
Example
Overloading vs Templates

- '+' operation is overloaded for different operand types

- A single function template can calculate sum of array of many types

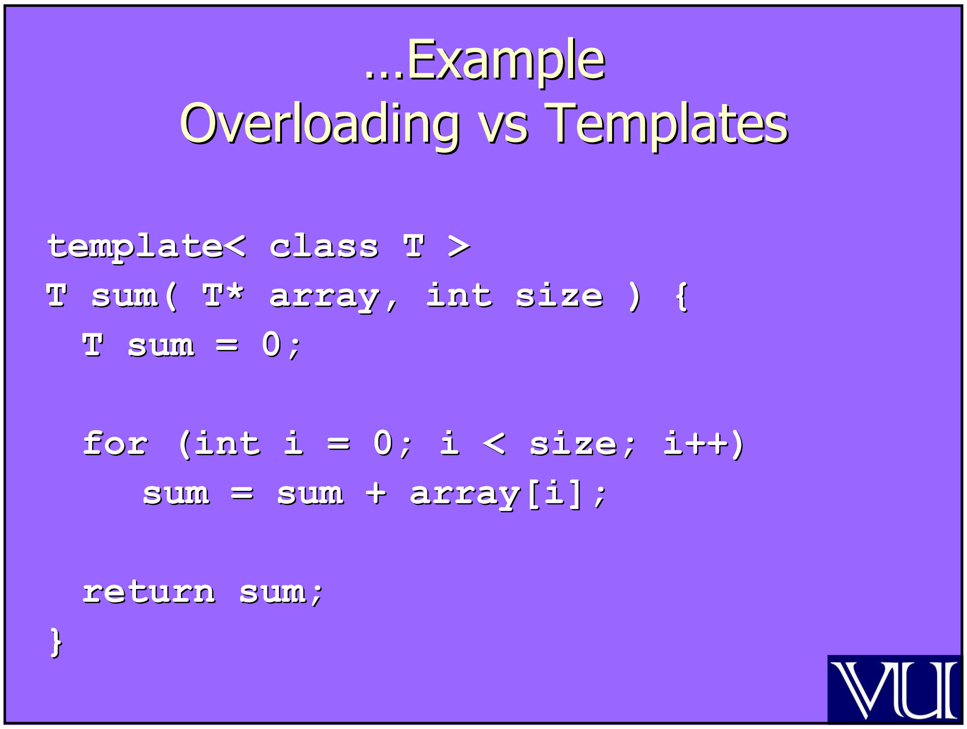
...Example
Overloading vs Templates

```
String operator +( const String& x,
                   const String& y ) {
  String tmp;
  tmp.pStr = new char[strlen(x.pStr) +
                   strlen(y.pStr) + 1 ];
  strcpy( tmp.pStr, x.pStr );
  strcat( tmp.pStr, y.pStr );
  return tmp;
}
```



...Example
Overloading vs Templates

```
String operator +( const char * str1,
                   const String& y ) {
  String tmp;
  tmp.pStr = new char[ strlen(str1) +
                   strlen(y.pStr) + 1 ];
  strcpy( tmp.pStr, str1 );
  strcat( tmp.pStr, y.pStr );
  return tmp;
}
```



...Example
Overloading vs Templates

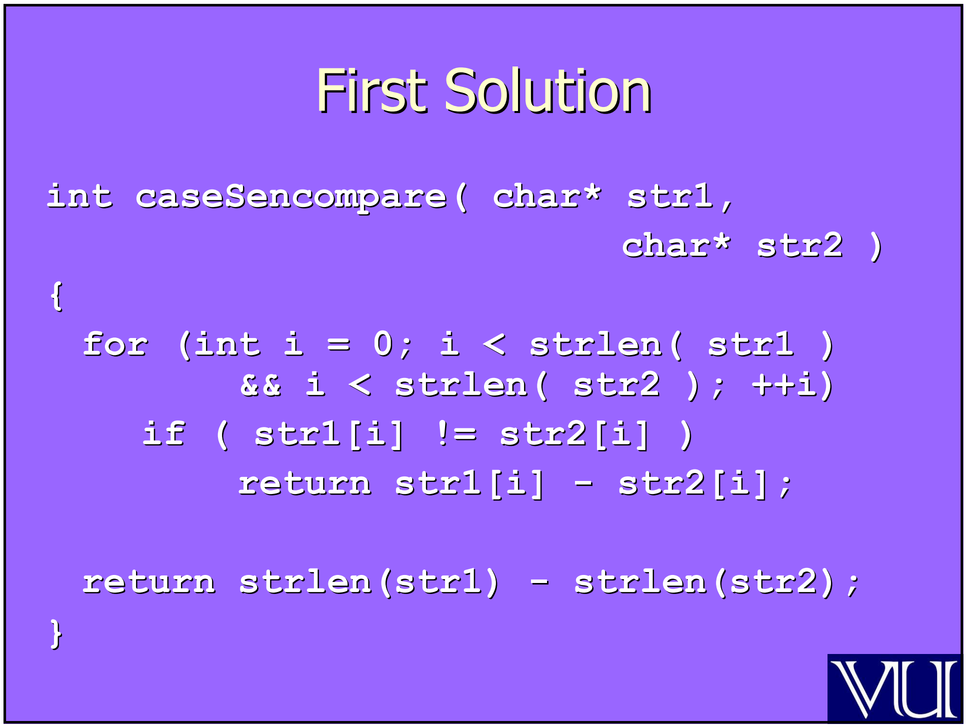
```
template< class T >
T sum( T* array, int size ) {
  T sum = 0;

  for (int i = 0; i < size; i++)
     sum = sum + array[i];

  return sum;
}
```

Template Arguments as Policy

- Policy specializes a template for an operation (behavior)

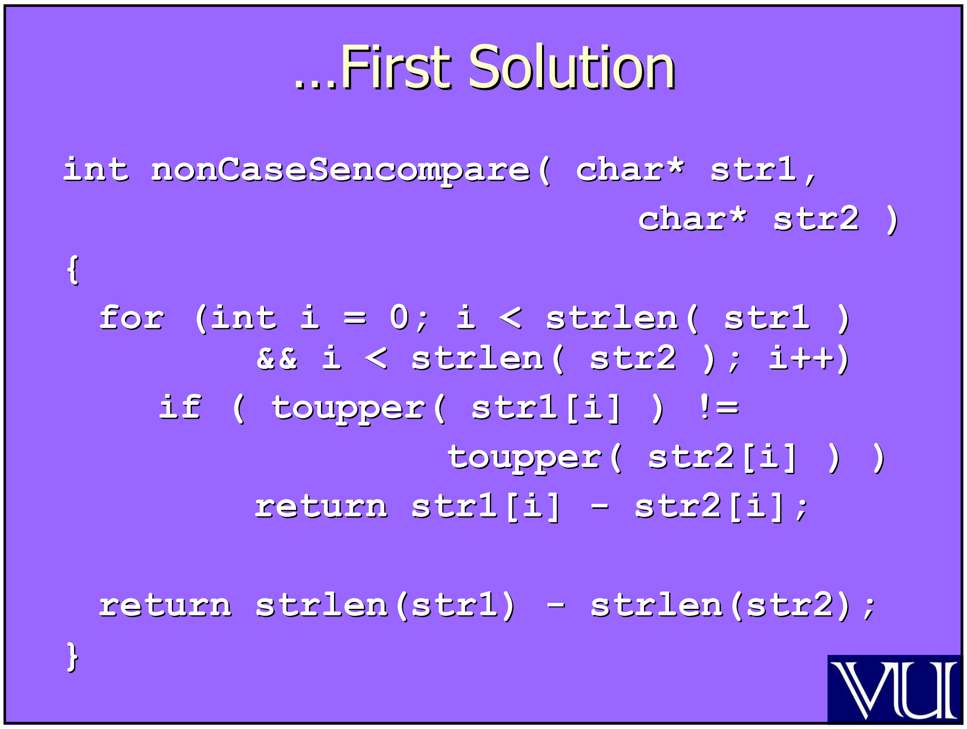
Example – Policy

- Write a function that compares two given character strings
- Function can perform either case-sensitive or non-case sensitive comparison



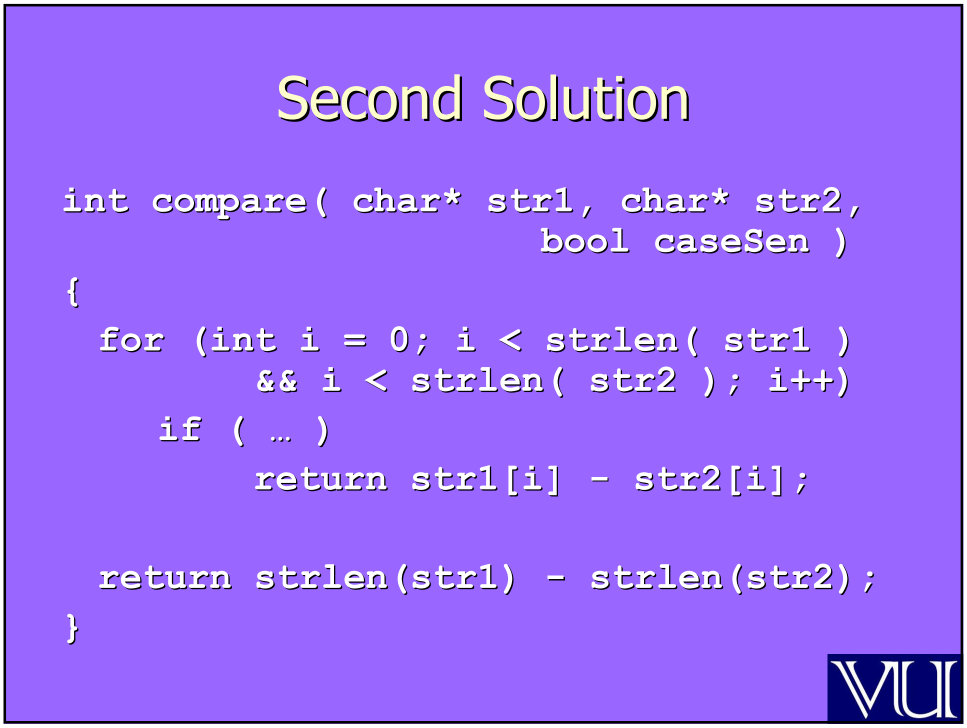
First Solution

```
int caseSencompare( char* str1,
                               char* str2 )
{
  for (int i = 0; i < strlen( str1 )
          && i < strlen( str2 ); ++i)
     if ( str1[i] != str2[i] )
          return str1[i] - str2[i];

  return strlen(str1) - strlen(str2);
}
```



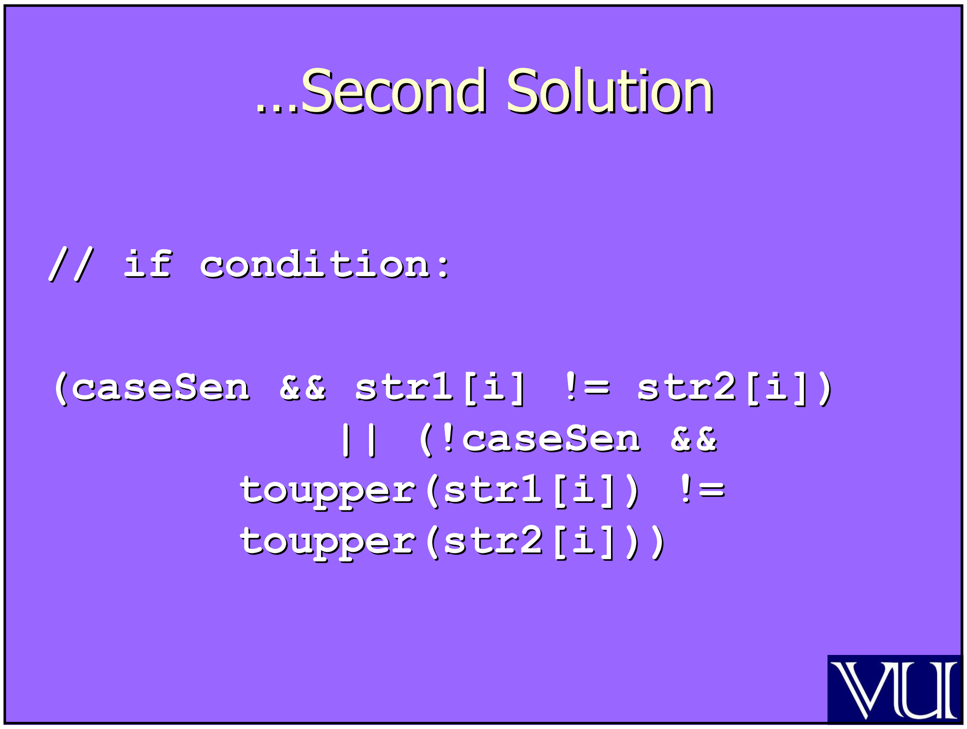
…First Solution

```
int nonCaseSencompare( char* str1,
                              char* str2 )
{
  for (int i = 0; i < strlen( str1 )
          && i < strlen( str2 ); i++)
     if ( toupper( str1[i] ) !=
                    toupper( str2[i] ) )
          return str1[i] - str2[i];

  return strlen(str1) - strlen(str2);
}
```



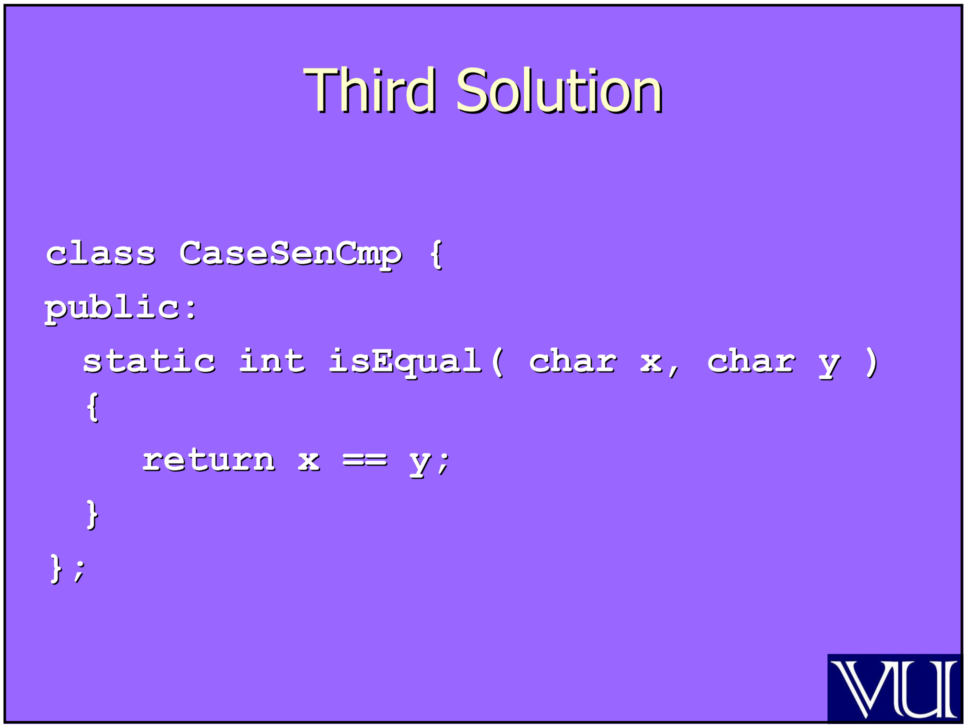
Second Solution

```
int compare( char* str1, char* str2,
                          bool caseSen )
{
  for (int i = 0; i < strlen( str1 )
          && i < strlen( str2 ); i++)
     if ( … )
          return str1[i] - str2[i];

  return strlen(str1) - strlen(str2);
}
```



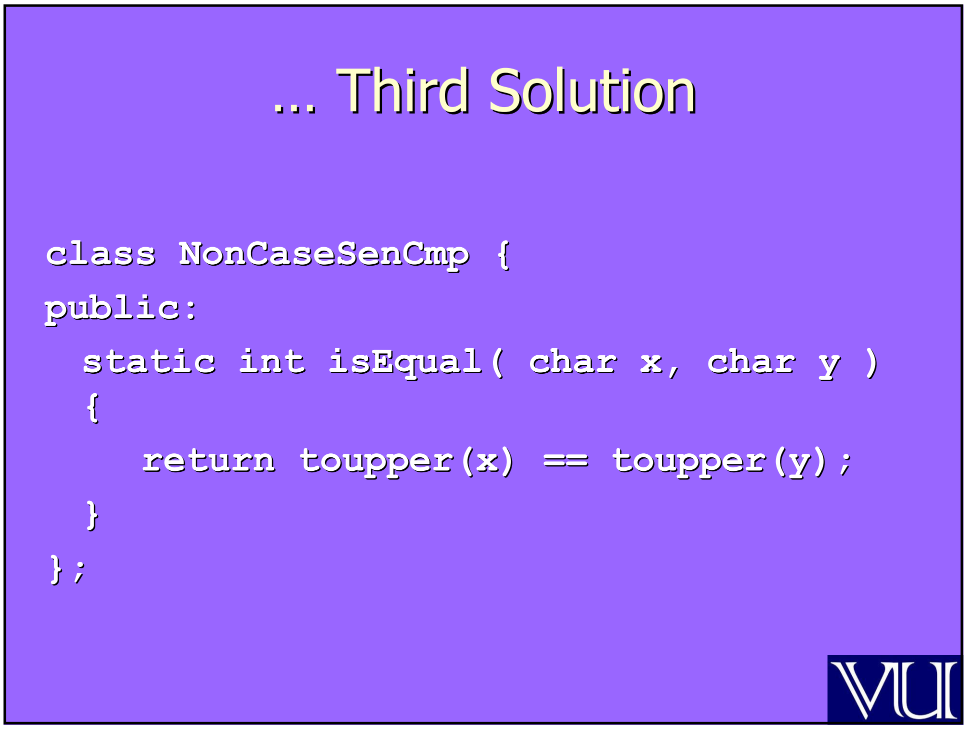
…Second Solution

```
// if condition:

(caseSen && str1[i] != str2[i])
              || (!caseSen &&
         toupper(str1[i]) !=
         toupper(str2[i]))
```



Third Solution

```
class CaseSenCmp {
public:
  static int isEqual( char x, char y )
  {
     return x == y;
  }
};
```



… Third Solution

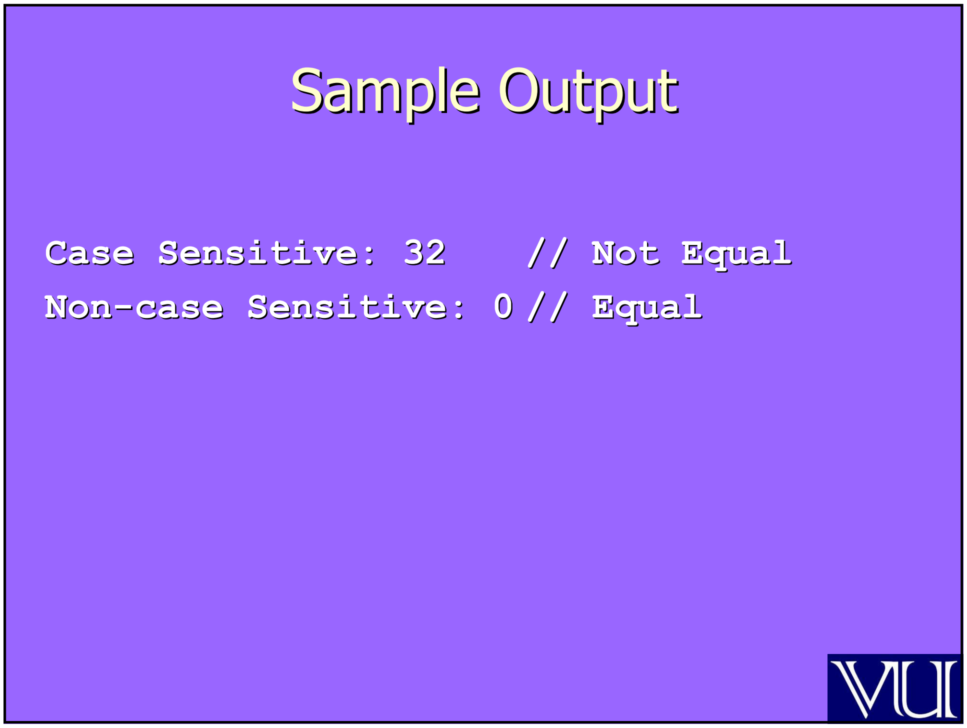
```
class NonCaseSenCmp {
public:
  static int isEqual( char x, char y )
  {
     return toupper(x) == toupper(y);
  }
};
```

...Third Solution

```
template< typename C >
int compare( char* str1, char* str2 )
{
  for (int i = 0; i < strlen( str1 )
           && i < strlen( str2 ); i++)
     if ( !C::isEqual
                (str1[i], str2[i]) )
           return str1[i] - str2[i];

  return strlen(str1) - strlen(str2);
};
```

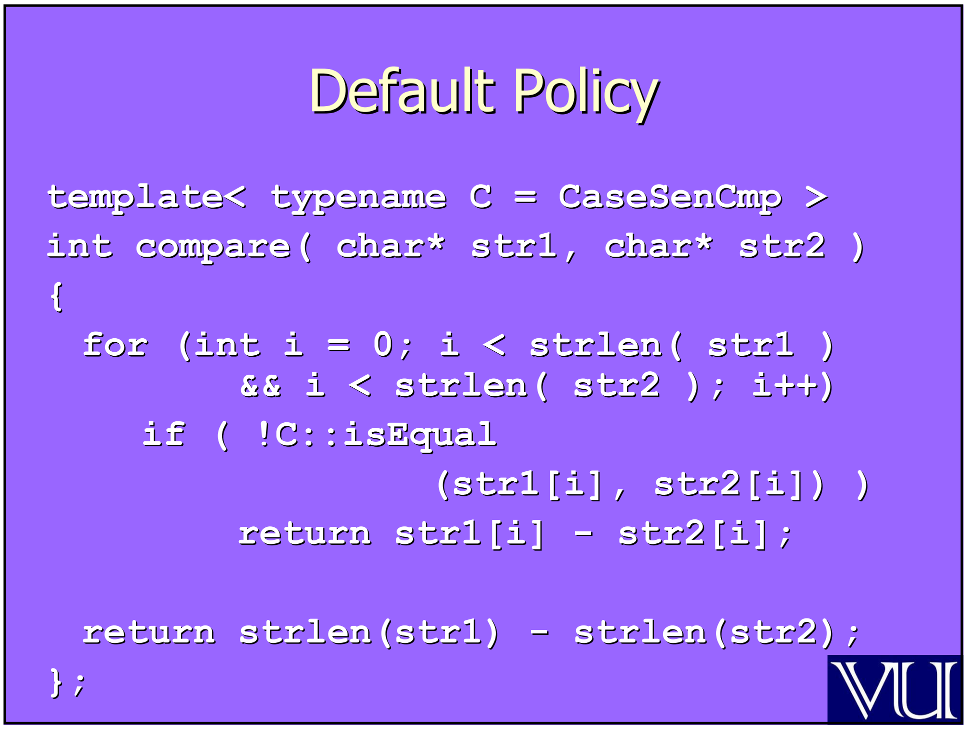
...Third Solution

```
int main() {
  int i, j;
  char *x = "hello", *y = "HELLO";
  i = compare< CaseSenCmp >(x, y);
  j = compare< NonCaseSenCmp >(x, y);
  cout << "Case Sensitive: " << i;
  cout << "\nNon-Case Sensitive: "
           << j << endl;
  return 0;
}
```



Sample Output

```
Case Sensitive: 32    // Not Equal
Non-case Sensitive: 0 // Equal
```



Default Policy

```
template< typename C = CaseSenCmp >
int compare( char* str1, char* str2 )
{
  for (int i = 0; i < strlen( str1 )
           && i < strlen( str2 ); i++)
     if ( !C::isEqual
                      (str1[i], str2[i]) )
           return str1[i] - str2[i];

  return strlen(str1) - strlen(str2);
};
```

...Third Solution

```
int main() {
  int i, j;
  char *x = "hello", *y = "HELLO";
  i = compare(x, y);
  j = compare< NonCaseSenCmp >(x, y);
  cout << "Case Sensitive: " << i;
  cout << "\nNon-Case Sensitive: “
          << j << endl;
  return 0;
}
```