Objectives

- Generic Programming

Motivation

Following function prints an array of integer elements:

```
void printArray(int* array, int size)
{
 for ( int i = 0; i < size; i++ )
  cout << array[ i ] << ", ";
}
```

...Motivation

What if we want to print an array of characters?

```
void printArray(char* array,
        int size)
{
 for ( int i = 0; i < size; i++ )
  cout << array[ i ] << ", ";
}
```

...Motivation

What if we want to print an array of doubles?

```
void printArray(double* array,
          int size)
{
 for ( int i = 0; i < size; i++ )
  cout << array[ i ] << ", ";
}
```

...Motivation

Now if we want to change the way function prints the array. e.g. from

`1, 2, 3, 4, 5`

to

1 - 2 - 3 - 4 - 5

...Motivation

Now consider the **`Array`** class that wraps an array of integers

```
class Array {
  int* pArray;
  int size;
public:
  …
};
```


...Motivation

What if we want to use an **`Array`** class that wraps arrays of double?

```
class Array {
  double* pArray;
  int size;
public:
  …
};
```


...Motivation

What if we want to use an **`Array`** class that wraps arrays of boolean variables?

```
class Array {
   bool* pArray;
   int size;
 public:
   …
 };
```

...Motivation

Now if we want to add a function sum to **`Array`** class, we have to change all the three classes

Generic Programming

Generic programming refers to programs containing generic abstractions

A generic program abstraction (function, class) can be parameterized with a type

Such abstractions can work with many different types of data

Advantages

Reusability

Writability

Maintainability

Templates

In C++ generic programming is done using templates

Two kinds
- Function Templates
- Class Templates

Compiler generates different type-specific copies from a single template

Function Templates

- In C++, **function templates** are functions that serve as a pattern for creating other similar functions
  - Create a function without having to specify the exact type(s) of some or all of the variables

NORMAL FUNCTION

```
int max( int x, int  y ) {
        if( x > y){
                return x;
        }else{
                return y;
        }
}
```

TEMLATE FUNCTION

```
template< class T >
T maxT( T x, T y ) {
                if( x > y){
                return x;
        }else{
                return y;
        }
}
```

Function Templates

- Template Function Save lot of time
- Only need to write one function, and it will work with many different types
- Template fonctions reduce code maintenance by reducing duplicate code

How Function Templates Works

- C++ Doesn’t Compile template functions directly
- It replicates the template function and replaces the template type parameters with actual types when compiler encounter the call to template function at compile time
- The function with actual types is called a **function template instance**.

Function Templates Specialization

- The idea of template specialization is to override the default template implementation to handle a particular type in a different way.

```
template< class T >
void print( T* array, int size )
{
   cout<<"Printing Generic Array"<<endl;
   for ( int i = 0; i < size; i++ ){
     cout << array[ i ] << ", ";
   }
  cout<<endl;
}
```

```
template< >
void print( char* array, int size )
{
   cout<<"Printing Generic Array"<<endl;
   for ( int i = 0; i < size; i++ ){
     cout << array[ i ] << ", ";
   }
  cout<<endl;
}
```

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;
}
```

Class Template

Like function templates, a class template is supported by C++

Class Template Specialization

Like function templates, a class template may not handle all the types successfully

Explicit specializations are provided to handle such types