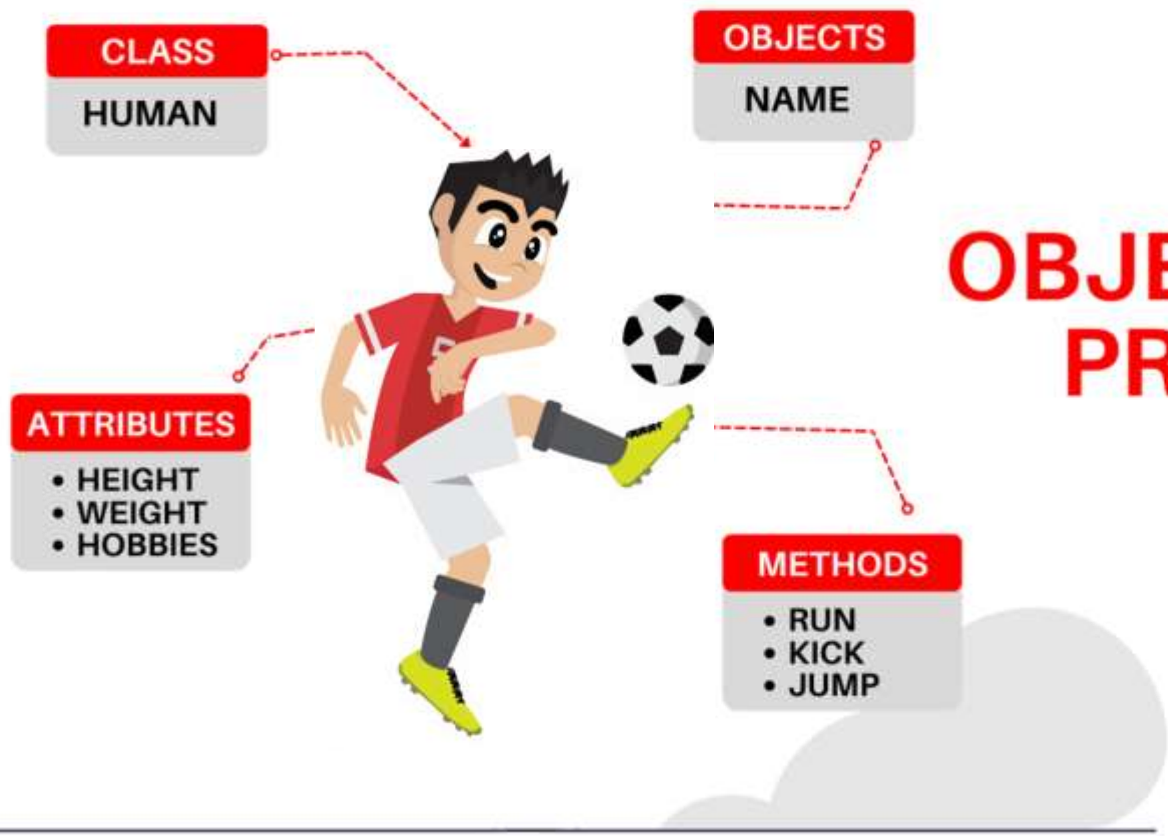
National University of Computer and Emerging Sciences



OBJECT-ORIENTED PRORAMMING

Summer 2023

Pir Sami Ullah Shah

Lecture # 11 Templates

Functions

Functions make pieces of code reusable by encapsulating them within a function.

e.g. Interchange the **values** of **two int variables x and y**.

Instead of inline code:

```
int temp = x;
x = y;
y = temp;
```

Write a function:

```
void Swap(int & first, int & second)
{
  int temp = first;
  first = second;
  second = temp;
}
```

General Solution

This function gives a general solution to the interchange problem for integers (to exchange the values of any two integer variables):

```
Swap(x, y);

   ...
Swap(w, z);
   ...
Swap(a, b);
```

But not for doubles…

To interchange the values of two double variables:

Cannot use the preceding function; it swaps ints not doubles.

However, ***overloading*** allows us to ***define multiple versions*** of the ***same function:***

```
/* Function to swap two double variables */
void Swap(double & first, double & second)
{
  double temp = first;
  first = second;
  second = temp;
}
```

The two different `Swap` functions are distinguished by the compiler according to each function's *signature* (name, number, type, and order of parameters).

$\underbrace{\hspace{20em}}_{\textbf{Signature}}$

And for strings…

To interchange the values of two string variables:

Again, overload function Swap():

```
/* Function to swap two string variables*/
void Swap(string & first, string & second)
{
  string temp = first;
  first = second;
  second = temp;
}
```

What about User Defined Types?

And so on ... for other types of variables.

We would have to overload Swap() for each user-defined type:

```
/* Function to swap two Time class objects
*/
void Swap(Time & first, Time & second)
{
   Time temp = first;
   first = second;
   second = temp;
}
```

Observations:

- The logic in each function is exactly the same.
- The only difference is in the *datatypes*
- If we could pass the datatype as an argument, we could write a general solution that could be used to exchange the values of any two variables.

Template Mechanism

Declare a datatype parameter (type placeholder) and use it in the function instead of a specific datatype. This requires a different kind of parameter list:

```
template <typename DataType >  // type parameter

void Swap(_________ & first, _________ & 
second)
{
  _________     temp  = first;
        first = second;
        second = temp;
}
```

Template Mechanism

Declare a datatype parameter (type placeholder) and use it in the function instead of a specific datatype. This requires a different kind of parameter list:

```
template <typename DataType >  // type parameter

void Swap( DataType & first, DataType &
second)
{
  DataType    temp  = first;
          first = second;
          second = temp;
}
```

Template Mechanism Comments

- The word *template* is a C++ keyword specifying that what follows is a pattern for a function not a function definition.

- Originally, the keyword *class* was used instead of *typename* in a type-parameter list.

- You can use typename or class

Template Instantiation

- The word *template* is a C++ keyword specifying that what follows is a pattern for a function not a function definition.

```cpp
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
    cout << myMax<int>(3, 7) << endl;
    cout << myMax<char>('g', 'e') << endl;
    return 0;
}
```

Compiler internally generates and adds below code

```cpp
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```cpp
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

Template Instantiation

- A function template is only a pattern that describes how individual functions can be built from given actual datatypes.
- This process of constructing a function is called instantiation.
- We instantiated Swap() four times — with datatypes int, double, string, and Time.
- In each instantiation, the template type parameter is said to be bound to the actual datatype passed.
- A template thus serves as a pattern for the definition of an unlimited number of instances.

How is a Template Used?

```
// One function works for all data types.  This would work
// even for user defined types if operator '>' is overloaded
template <typename T> T myMax(T x, T y)
{
    return (x > y) ? x : y;
}

int main()
{
    cout << myMax<int>(3, 7) << endl; // Call myMax for int
    cout << myMax<double>(3.0, 7.0)
         << endl; // call myMax for double
    cout << myMax<char>('g', 'e')
         << endl; // call myMax for char

    return 0;
}
```

General Form of Template

```
template <typename TypeParam>
FunctionDefinition
```

or

```
template <class TypeParam>
FunctionDefinition
```

where:

TypeParam **is a type-parameter (placeholder) naming the "generic" type of value(s) on which the function operates**

FunctionDefinition **is the definition of the function, using type** ***TypeParam*.**

Templates with more than one Parameter

- Like normal parameters, we can pass more than one data types as arguments to templates.

```
template <class T, class U>
class A {
    T x;
    U y;

public:
    A() { cout << "Constructor Called" << endl; }
};

int main()
{
    A<char, char> a;
    A<int, double> b;
    return 0;
}
```

Templates with default Parameter

- Like normal parameters, we can pass more than one data types as arguments to templates.

```
template <class T = int>
class A {
    T x;

public:
    A() { cout << "Constructor Called" << endl; }
};

int main()
{
    A a; // T is an int by default
    A<double> b; //T is a double

    return 0;
}
```

Class Templates

- The template concept can be extended to classes
- Write template before class definition
- If the member functions are out-of-line the expression **`template<class Type>`** must precede not only the class definition, but each out-of-line function as well

Class Templates

```
template <typename T>
class Array {
private:
    T* ptr;
    int size;

public:
    Array(T arr[], int s);
    void print();
};
```

```
int main()
{
    int arr[5] = { 1, 2, 3, 4, 5 };
    Array<int> a(arr, 5);
    a.print();
    return 0;
}
```

Instantiating class objects

```
template <typename T> Array<T>::Array(T arr[], int s)
{
    ptr = new T[s];
    size = s;
    for (int i = 0; i < size; i++)
        ptr[i] = arr[i];
}

template <typename T> void Array<T>::print()
{
    for (int i = 0; i < size; i++)
        cout << " " << *(ptr + i);
    cout << endl;
}
```

```cpp
// function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

```
// function template II
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  return (a>b?a:b);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax(i,j);
  n=GetMax(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```