# Advanced Object Oriented Programming

## *Template*

Seokhee Jeon
Department of Computer Engineering
Kyung Hee University
jeon@khu.ac.kr

믹스없이 엄마표로 즐기는! 3색 붕어빵 by꼬마츄츄
완소남매의 엄마표 바른먹거리
http://blog.naver.com/baby0817


http://blog.naver.com/lov2hsjk


# Functions

- Actions that are applied
- Data that are involved

If the data are different,
we need to write a different version of the function for each type of data

# Why we use template?

- **Example : Multiple max functions**
  - What's your approach for below problem – four separate functions?

```
int max (int x, int y)
{
  return (x > y) ? x : y;
}  // max
```

(a) Integer max

```
float max (float x, float y)
{
  return (x > y) ? x : y;
}  // max
```

(c) Float max

```
long max (long x, long y)
{
  return (x > y) ? x : y;
}  // max
```

(b) Long max

```
double max (double x, double y)
{
  return (x > y) ? x : y;
}  // max
```

(d) Double max

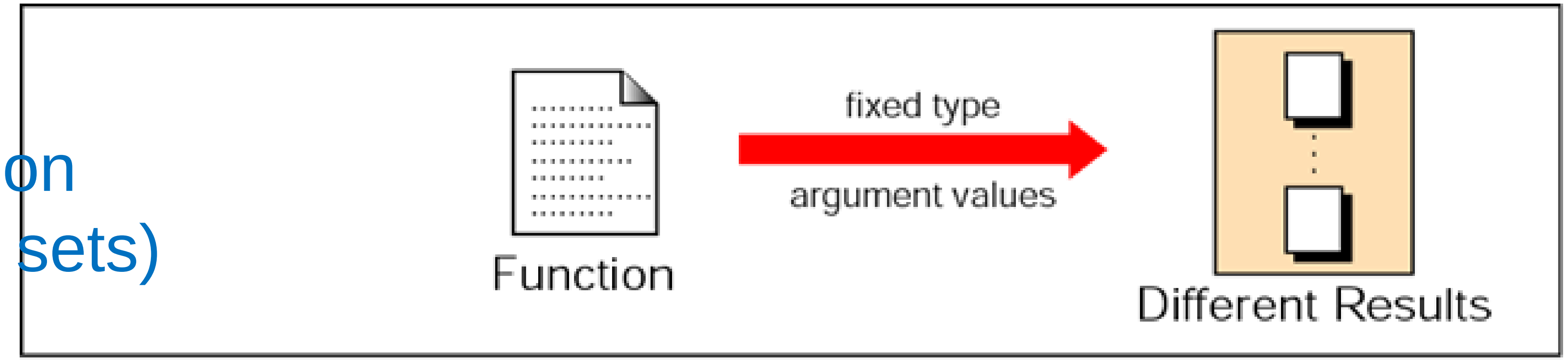
경희대학교 Kyung Hee University

# Template

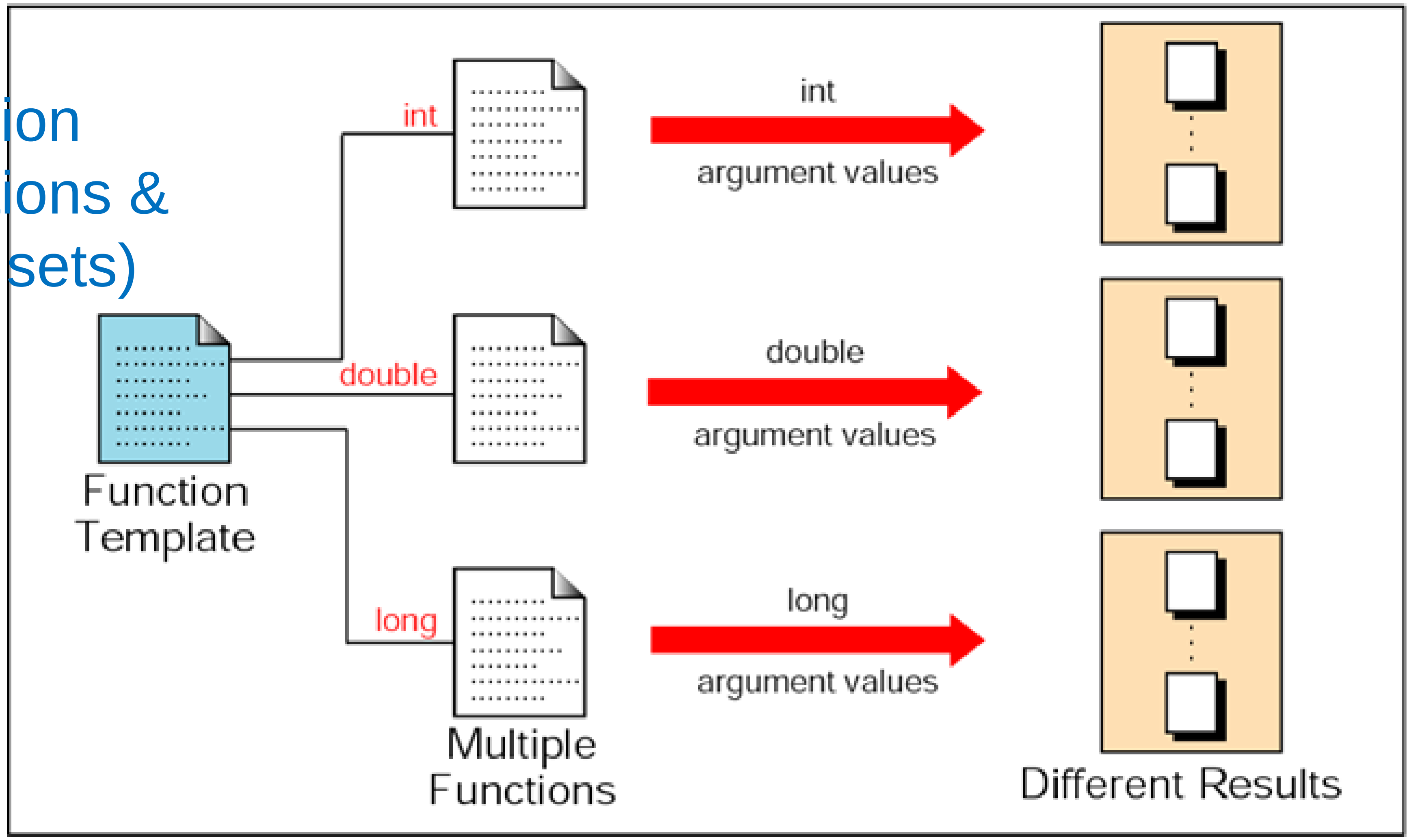
- Model of a function or a class that can be used to generate functions or classes
- During the compilation, C++ uses the template to generate functions and classes
- Two types of templates
  - Given a **function template**, one, two, or more concrete functions can be generated in the program
  - Given a **class template**, one, two, or more concrete classes can be generated in the program

# Function template operation

One level
of generalization
(different data sets)




Two levels
of generalization
(several functions &
different data sets)



# Function template implementation

```
template <class TYPE>
TYPE max (TYPE x, TYPE y)
{
   return (x > y) ? x : y;
}   // max
```

Compiler

```
int max (int x, int y)
{
   return (x > y) ? x : y;
}   // max
```

```
long max (long x, long y)
{
   return (x > y) ? x : y;
}   // max
```

```
float max (float x, float y)
{
   return (x > y) ? x : y;
}   // max
```

```
double max (double x, double y)
{
   return (x > y) ? x : y;
}   // max
```

# Function template implementation

*Function templates allow us to write a single function for a whole family of similar functions.*

# Function template invocation

```
template <class TYPE>
TYPE max (TYPE x, TYPE y)
{
   return (x > y) ? x : y;
}  // max
```

Generate

```
int max (int x, int y)
{
   return (x > y) ? x : y;
}  // max
```

Call

```
int num1;
int num2;
int result;
   .
   .
   .
result = max (num1, num2);
```

경희대학교 Kyung Hee University

# Sample code: Single template parameter

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

template <class TYPE>
TYPE max (TYPE x, TYPE y)
{
  return (x > y ) ? x : y;
}              //max template

int main ()
{
              int i1   = rand();
              int i2   = rand();
              cout << "Given "    << setw(5) << i1   << " and "    << setw(5) << i2   << ": "   << max(i1, i2)   << " is larger\n";
              float f1   = rand() / 3.3;
              float f2   = rand() * 6.7;
              cout << "\nGiven "   << setw(5) << f1   << " and "    << setw(5) << f2   << ": "   << max(f1, f2)   << " is larger\n";
              return 0;
}             //main

/* Results
Given 16838 and  5758: 16838 is larger

Given 3064.55 and 117350: 117350 is larger
*/
```

# Function template declaration summary

```
/* Demonstrate template declaration
      Written by:
      Date:
*/
#include <iostream>
using namespace std;
// Function Templates
template <class generic_type>
return_type function_name (arguments)
{
           Function Body
}  // function_name
```

Generic Type

# Overloading function templates

- When a function template will not work, write an overloaded function that handles the specific cases

```
template <class TYPE>
TYPE max (TYPE x, TYPE y)
{
    return (x > y) ? x : y;
}   // max

//   Overload max for Fractions
Fraction max (Fraction fr1, Fraction fr2)
{
    if (fr1.compare(fr2) > 0)
      return fr1;
    else
      return fr2;
}   // max Fraction
```

GREATER operator (>) is not defined for Fraction objects

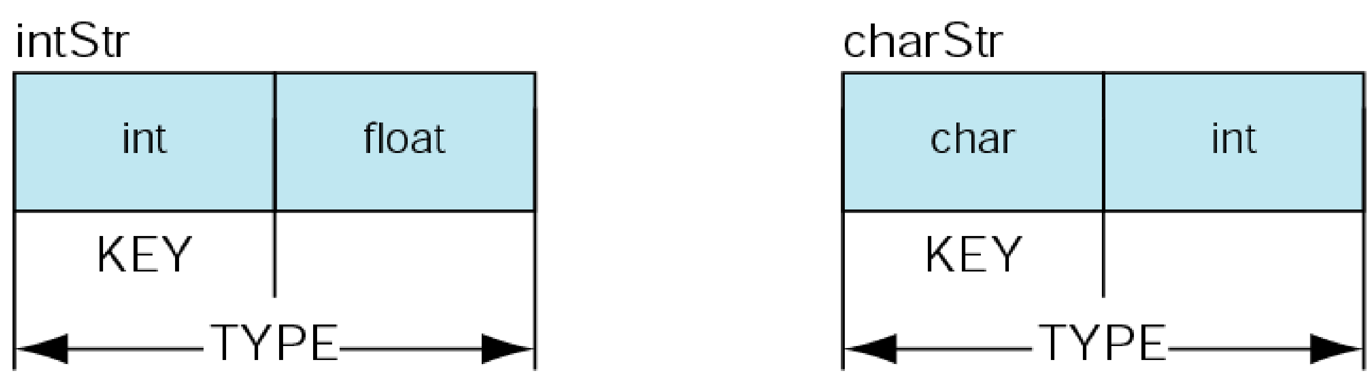
# Mixed argument types

- Within the parameter list of a function template, generic types and standard types can be intermixed in any order

```
template<class TYPE>
TYPE smallest  (TYPE arr[] , int size)
{
     TYPE smallestValue = arr[0];
     for (int index = 0;
           index < size;
           index++)
         {
          if (arr[index] < smallestValue)
              smallestValue = arr[index];
     } // for
     return smallestValue;
}    // smallest
```

# Multiple generic argument types

```
template <class TYPE, class KEY>
int search (TYPE arr[], KEY key, int size)
{
        int index = 0;
        while ( index < size )
          {
           if (key != arr[index].key)
              index++;
           else
              return index;
          } // while
        return -1;
}       // search Template
```




TYPE → intStr
KEY → int

TYPE → charStr
KEY → char

# Function templates vs. Overloading

- With overloaded functions, we must code the function for each usage
- With a function template, we code the function only once
  - The compiler automatically creates an instance of the function for each different calling type

# Function templates vs. Macros

- It is easier to make a mistake when coding a macro since the compiler may not catch it
  - Example: In Program 13-7 the 3rd example
- It is more difficult to debug macros because they are handled by the preprocessor
  - What we see in the listing is not what C++ is looking at

## Program 13-7 Function templates versus macros

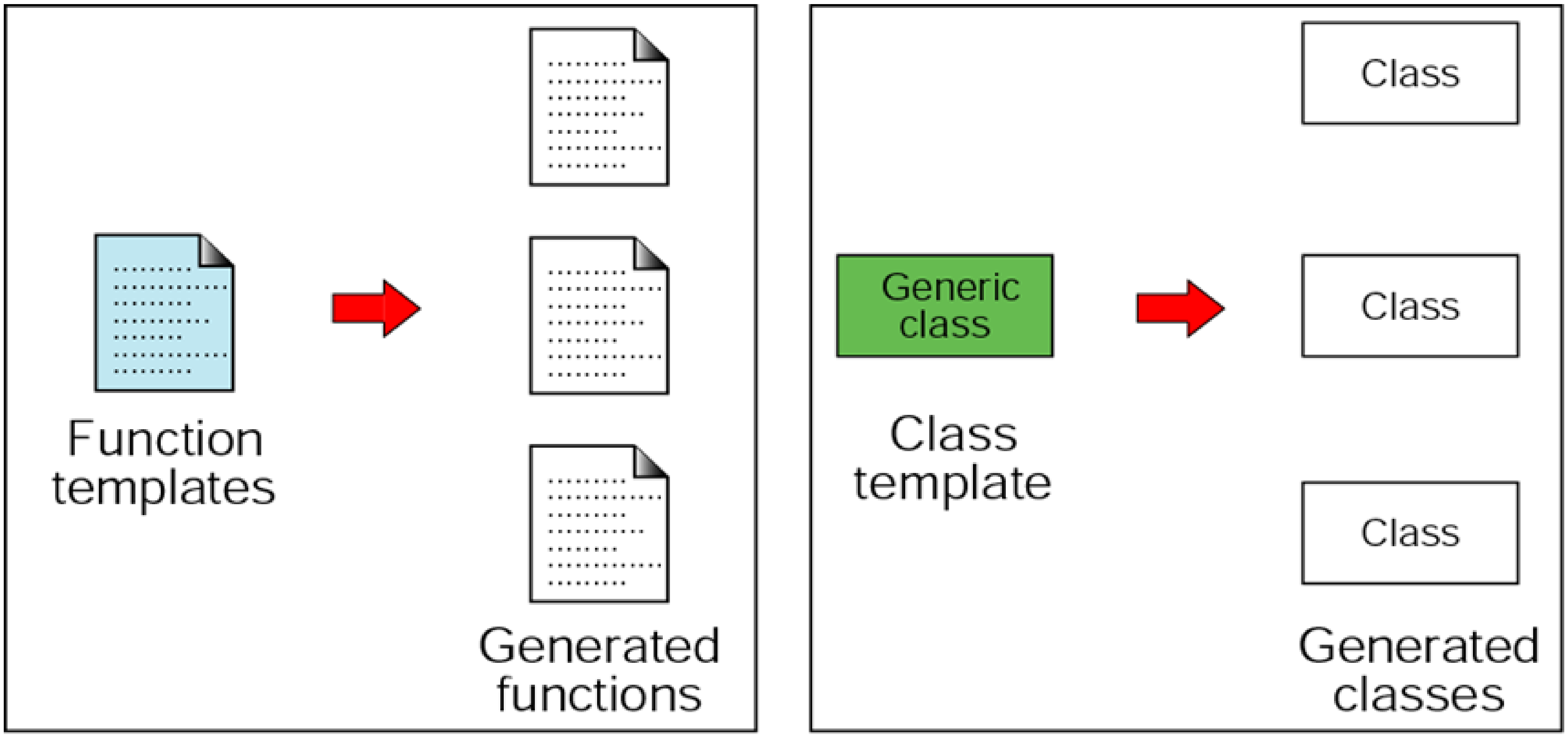
```
/*  This program uses a macro to write the max
    function.
*/
#define MAX(x, y) (((x) > (y)) ? (x) : (y))

#include <iostream>
using namespace std;

int main ()
{
    cout << "Begin macro tests\n";
    cout << "Test1: " << MAX (1, 4)     << endl;
    cout << "Test2: " << MAX ('A', 'B') << endl;
    cout << "Test3: " << MAX (4, 'A')   << endl;
    cout << "End of macro tests\n";
}   // main
```

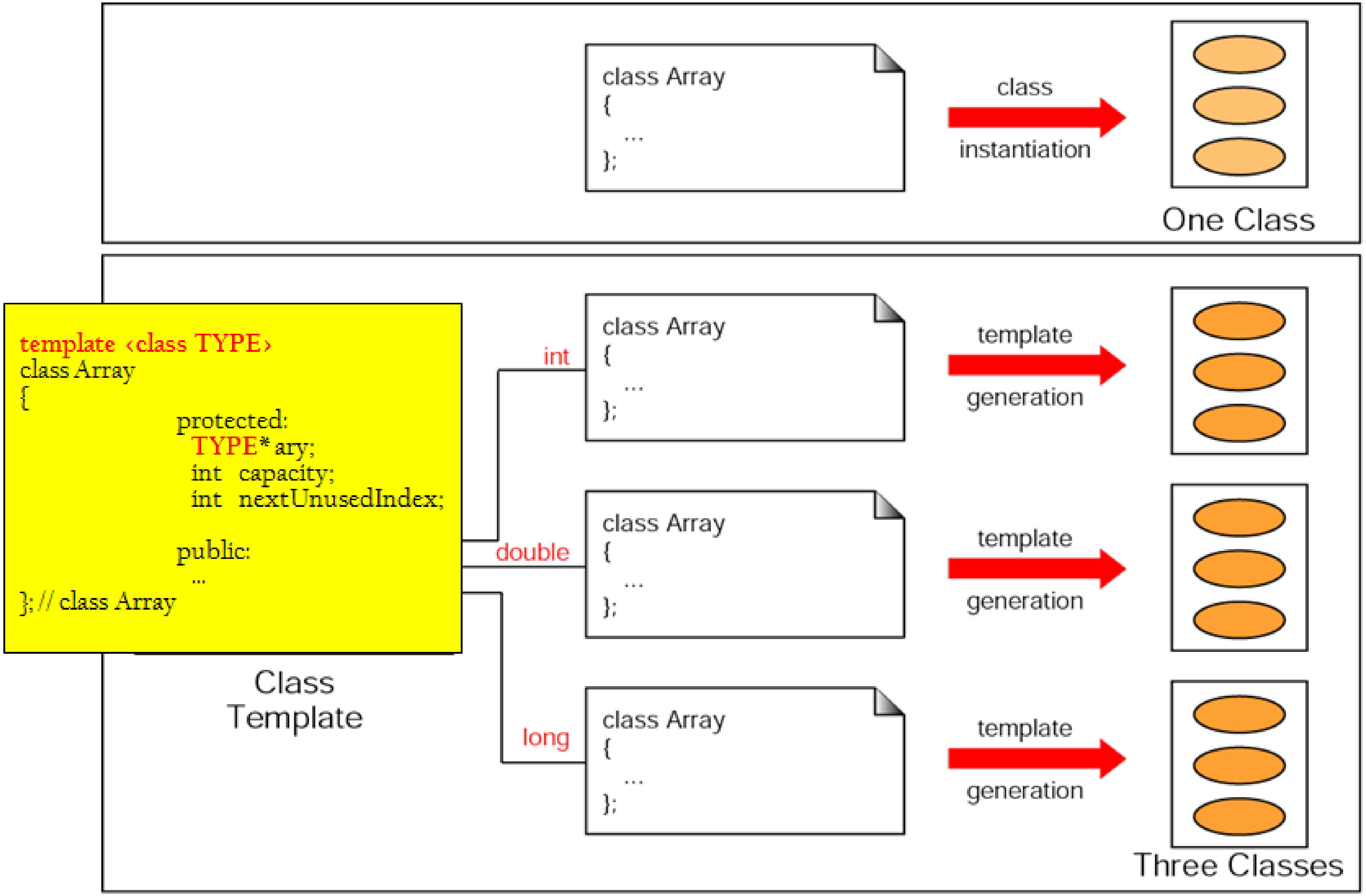
```
/*      Results
Begin macro tests
Test1: 4
Test2: B
Test3: 65
End of macro tests
*/
```

# Function template vs. Class template




(a) Function template (b) Class template

# Class template



# Class template definition & realization

```
#include <iostream>
using namespace std;

template <class TYPE>
class Array
{
          protected:
            TYPE* ary;                    // Parameterized
            int  capacity;
            int  nextUnusedIndex;

          public:
             Array  (int size);           // Constructor
          void append (TYPE data);       // Parameterized
}         ; // Array

//================== constructor =================
template <class TYPE>
Array <TYPE> :: Array (int size)
{
//        ...
}         // constructor

//================== append =================
template <class TYPE>
void Array <TYPE> :: append (<TYPE> data)
{
//        ...
}         // append
```

```
int main ()
{
//          Local Definitions
            Array <int>   intary  (...);
            Array <float> floatary(...);
            ...
}           // main
```

# Specialized member functions

- Write specialized member functions when some member functions in the template does not support a specific need

```
template <class T>
class X {
   public:
      void SomeFn();
}
```

```
template <class T>
X<T>::someFn()
{
         cout << "some type" << endl;
}
//specialized member function
X<int>::someFn()
{
        cout << "type int" << endl;
}
```

# Specialized classes

- Write specialized classes when the template does not support a specific need

```
template <class T>
class X {
   public:
      void SomeFn();
}
template <class T>
X<T>::someFn()
{
   cout << "some type" << endl;
}
```

```
//specialized class
template <>
class X<int> {
   public:
      void SomeFn();
}
template<>
X<int>::someFn()
{
        cout << "type int" << endl;
}
```

No parameters

# Class Template Inheritance

***When a class is inherited from a class template, the derived class is also a class template***

# Example: Class template inheritance

```
#include "p13-11.h"

template <class TYPE>
class MArray : public Array <TYPE>
{
                public:
                    MArray (int size);            // Constructor
                  void append (TYPE data);
                  void copy  (const MArray <TYPE> & toBeCopied);
};              // MArray

template <class TYPE>
Marray <TYPE> :: MArray (int size) : Array <TYPE> (size)
{

}               // constructor

template <class TYPE>
void   Marray <TYPE> :: append (TYPE data)
{
                if (capacity == nextUnusedIndex)
                  {
                   TYPE *temp = ary;
                   capacity += 10;
                   ary = new TYPE [capacity];
                   for (int index = 0; index < nextUnusedIndex;index++)
                       ary[index] = temp [index];
                   delete [] temp;
                  } // if
                ary[nextUnusedIndex] = data;
                nextUnusedIndex++;
}               // append
```

```
template <class TYPE>
void   Marray <TYPE> :: copy (const Marray <TYPE> &
                                  toBeCopied)
{
                if (capacity != toBeCopied.capacity)
                  {
                   delete [] ary;
                   ary  = new TYPE [toBeCopied.capacity];
                  } // if
                capacity = toBeCopied.capacity;
                nextUnusedIndex =
                  toBeCopied.nextUnusedIndex;
                for (int index = 0; index < nextUnusedIndex;
                    index++)
                    ary [index] = toBeCopied.ary[index];
}               // copy
```

# SOFTWARE ENGINEERING AND PROGRAMMING STYLE

# *Atomic Data Type*

*1. A set of values*

*2. A set of operations on values*

# Examples

- Int
  - Values: -∞, …, -2, -1, 0, 1, 2, …, ∞
  - Operations: *, +, -, %, /, ++, --, …
- Float
  - Values: -∞, …, 0.0, …, ∞
  - Operations: *, +, -, /, …
- Char
  - Values: □0, …, ‘A’, ‘B’, …, ‘a’, ‘b’, …, □127
  - Operations: +, -, …

# Data Structure

1. A combination of elements, each of which is either an atomic type or another data structure

2. A set of associations or relationships (structure) involving the combined elements

# Examples: Array and Class

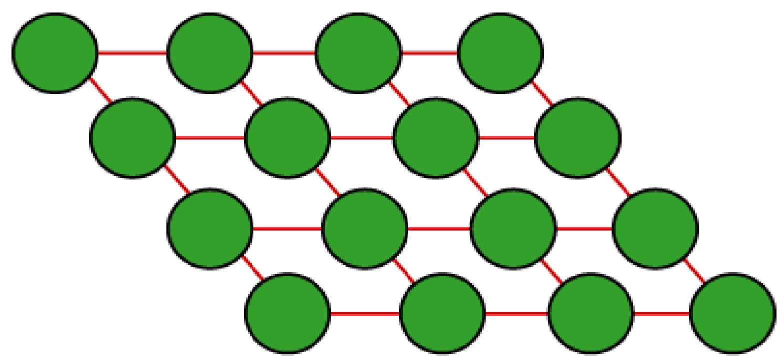
| array | class |
| --- | --- |
| 1. A homogeneous combination of data structures | 1. A heterogeneous combination of data structures |
| 2. Position association | 2. No association |
| 3. No use-defined operations | 3. Methods |

# Concept of Abstraction

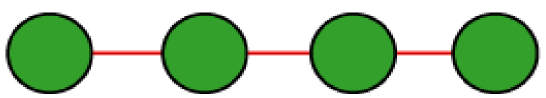
- ***We know what a data type can do.***
- ***How it is done is hidden.***

# Illustration of Abstraction: list

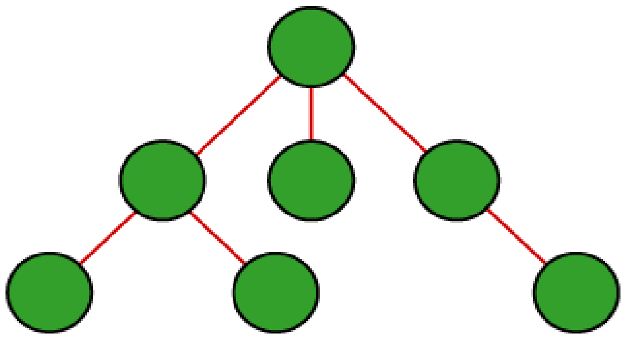
- What a data type can do
  - Hold a list of items (insert/remove/search)
- How it is done
  - Any structure that can hold a collection of items
- Examples



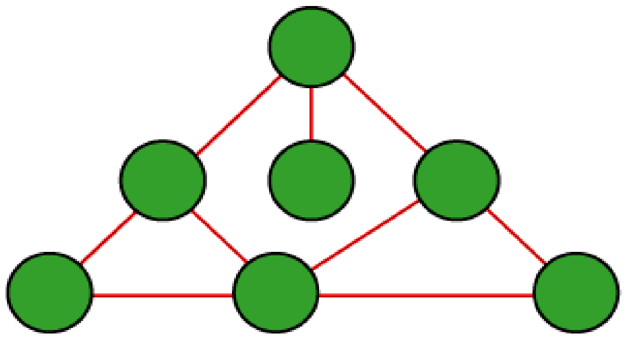
A matrix



A linked list



A tree



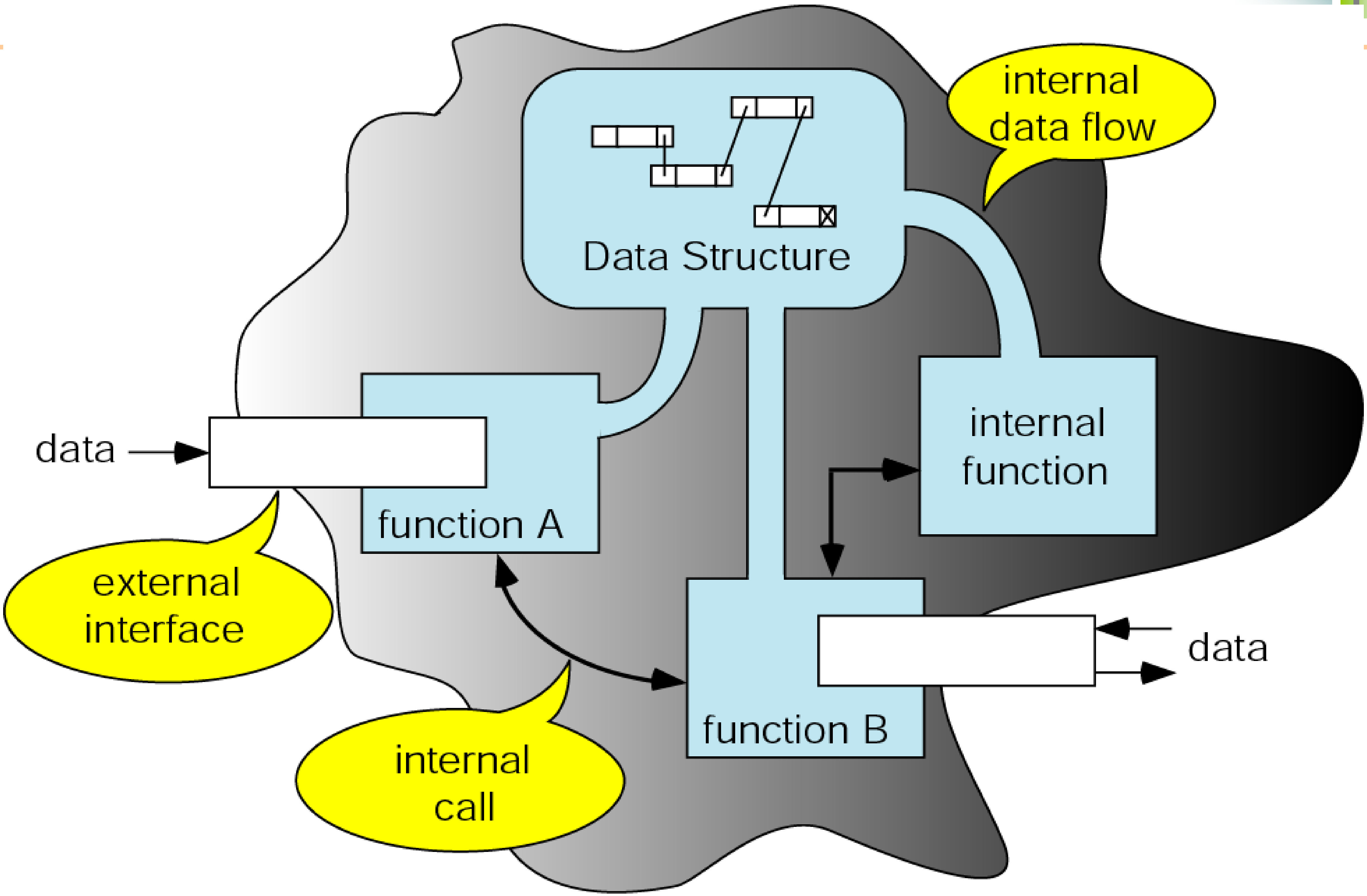
A network

# *Abstract Data Type*

***1. Declaration of data***

***2. Declaration of operations***

# Figure 13-11 Abstract data type model



# Questions?