Data Structures and Object Oriented Programming

Lecture 22

Dr. Naveed Anwar Bhatti

Webpage: naveedanwarbhatti.github.io

Generic Programming

Motivation

Suppose that…

- You want to write a function to compare two ints

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(int value1, int value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}
```

- You want to write a function to compare two floats

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(float value1, float value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}
```

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

- A function that accepts a type as a parameter
  - You define the function once in a type-agnostic way
  - When you invoke the function, you specify (one or more) types or values as arguments to it

Function Templates

- Example

```
template <typename T>

int compare(T value1, T value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}


int main() {

cout << compare(10, 20) << endl; // ok
cout << compare(12.55, 11.3) << endl; // ok
    return 0;
}
```

Output

```
-1
1
```

Explicit Type Parameterization

- Example

```
template <typename T>

int compare(T value1, T value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}


int main() {

    cout << compare(10, 20) << endl; // ok
    cout << compare(12.55, 11) << endl; // Error
    return 0;
}
```

float

int

Explicit Type Parameterization

- Example

```
template <typename T>

int compare(T value1, T value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}


int main() {

    cout << compare(10, 20) << endl; // ok
    cout << compare<float>(12.55, 11) << endl; // ok
    return 0;
}
```

float

float

Explicit Type Parameterization

- A function template may not have any parameter

```
template <typename T>

T getInput() {
    T x;
    cin >> x;
    return x;
}


int main() {
    int x;
    x = getInput();// Error!

    double y;
    y = getInput();// Error!
}
```

Explicit Type Parameterization

- A function template may not have any parameter

```
template <typename T>

T getInput() {
    T x;
    cin >> x;
    return x;
}

int main() {
    int x;
    x = getInput<int>();// Ok!

    double y;
    y = getInput<double>();// Ok!
}
```

User-defined Specializations

- A template may not handle all the types successfully

- Explicit specializations need to be provided for specific type(s)

Example – User Specializations

```
template< typename T >

bool isEqual(T x, T y)
{
    return (x == y);
}

int main()
{
    cout<<isEqual(5, 6); // ok
    cout<<isEqual(7.5, 7.5); // ok

    char a[] = "hello";
    char b[] = "hello";
    cout<<isEqual(a, b); // Logical Error
    return 0;
}
```

Example – User Specializations

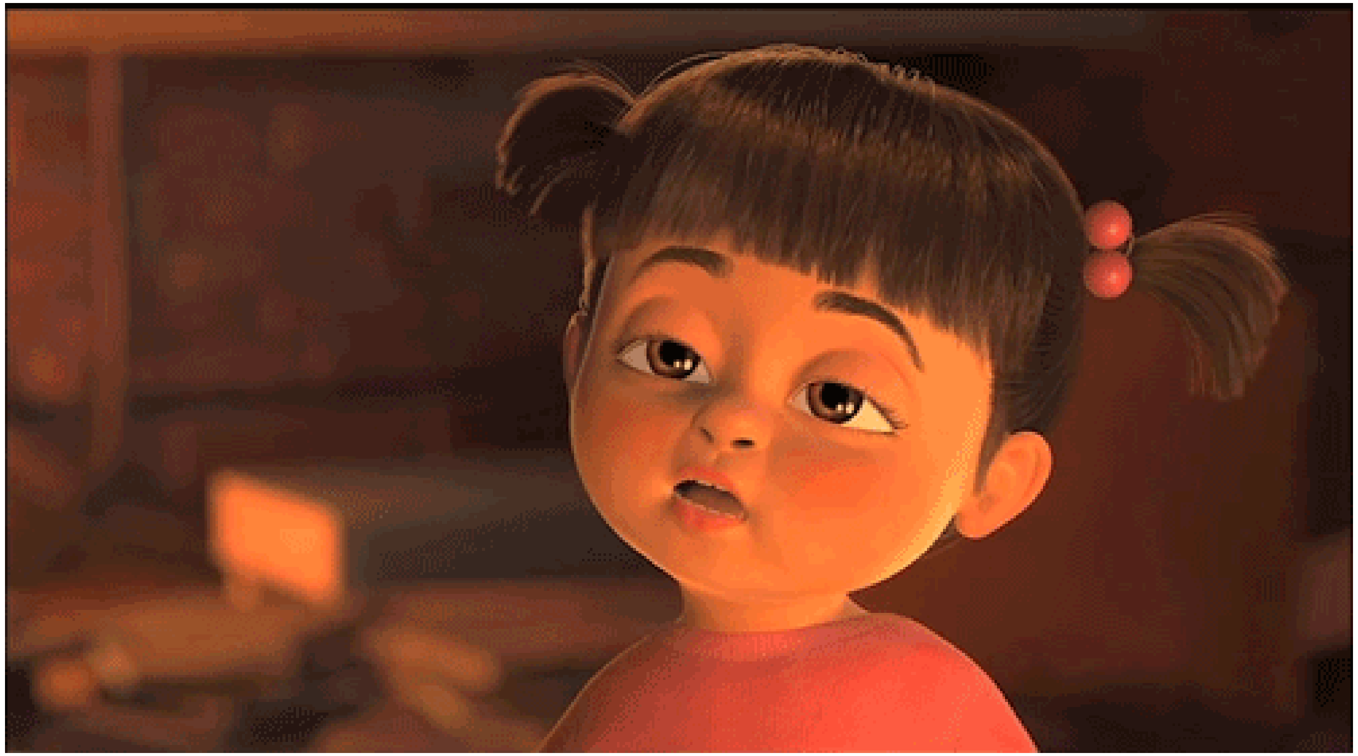
```
template< typename T >

bool isEqual(T x, T y)
{
    return (x == y);
}

template<>
bool isEqual<const char*>(const char* x, const char* y)
{
    return (strcmp(x, y) == 0);
}

int main()
{
    cout<<isEqual(5, 6); // ok
    cout<<isEqual(7.5, 7.5); // ok
    char a[] = "hello";
    char b[] = "hello";
    cout<<isEqual(a, b); // ok
    return 0;
}
```

Thanks a lot



If you are taking a Nap, **wake up**.......Lecture Over