Template

Modern cpp Programming & Missing session

We have learned…

- Basic cpp features
- Pointer / reference
- Class
- Inheritance
- Polymorphism
- STL

We have learned…

- OOP features in cpp
- Why OOP?
  - simulates real world
  - easy *library* | *module* development
  - easy abstraction
  - supports GUI programming
  - enhance code reusability

We have learned…

- OOP features in cpp
- Why OOP?
  - simulates real world
  - easy *library* | *module* development
  - easy abstraction
  - supports GUI programming
  - **enhance code reusability**
    - how can we enhance *more???*

Advanced features for code reusability

- **Template**
- Operator Overloading
- Design pattern

Why Template??

- Consider a situation…
- You implemented *add* function

```
int add(int a, int b) {
    return a + b;
}
```

- How about adding `double`?
  - `string`?
  - `bool`?
  - user-defined class?

Why Template?

- Easy solution
  - define new functions! (overloading)
    - int

      ```
      int add(int a, int b) {
          return a + b;
      }
      ```

    - double

      ```
      double add(double a, double b) {
          return a + b;
      }
      ```

    - string

      ```
      string add(string a, string b) {
          return a + b;
      }
      ```

Why Template?

- Easy solution
  - define new functions!
  - Pros
    - Intuitive
    - Easy to implement
  - Cons
    - Code explosion
    - low reusability
    - *Imagine a case where you have to respond to 100 data types…*

Why Template?

- Better solution
  - use template!!

    ```
    template <typename T>
    T add(T a, T b) {
        return a + b;
    }
    ```

  - Simple, reusable code!!
  - responds to *any* type!
    - however it does not work properly/normally for every type
      - due to "+"
      - ***template specialization / operator overloading*** will help the problem

Why Template??

- Better solution
  - use template!!

```
template <typename T>
T add(T a, T b) {
    return a + b;
}

int a = 5;
int b = 3;
cout << add<int>(a, b) << endl;

float c = 5.3;
float d = 4.2;
cout << add<float>(c, d) << endl;

string e = "Hello, ";
string f = "World!";
cout << add<string>(e, f) << endl;
```

```
output:
8
9.5
Hello, World!
```

Template

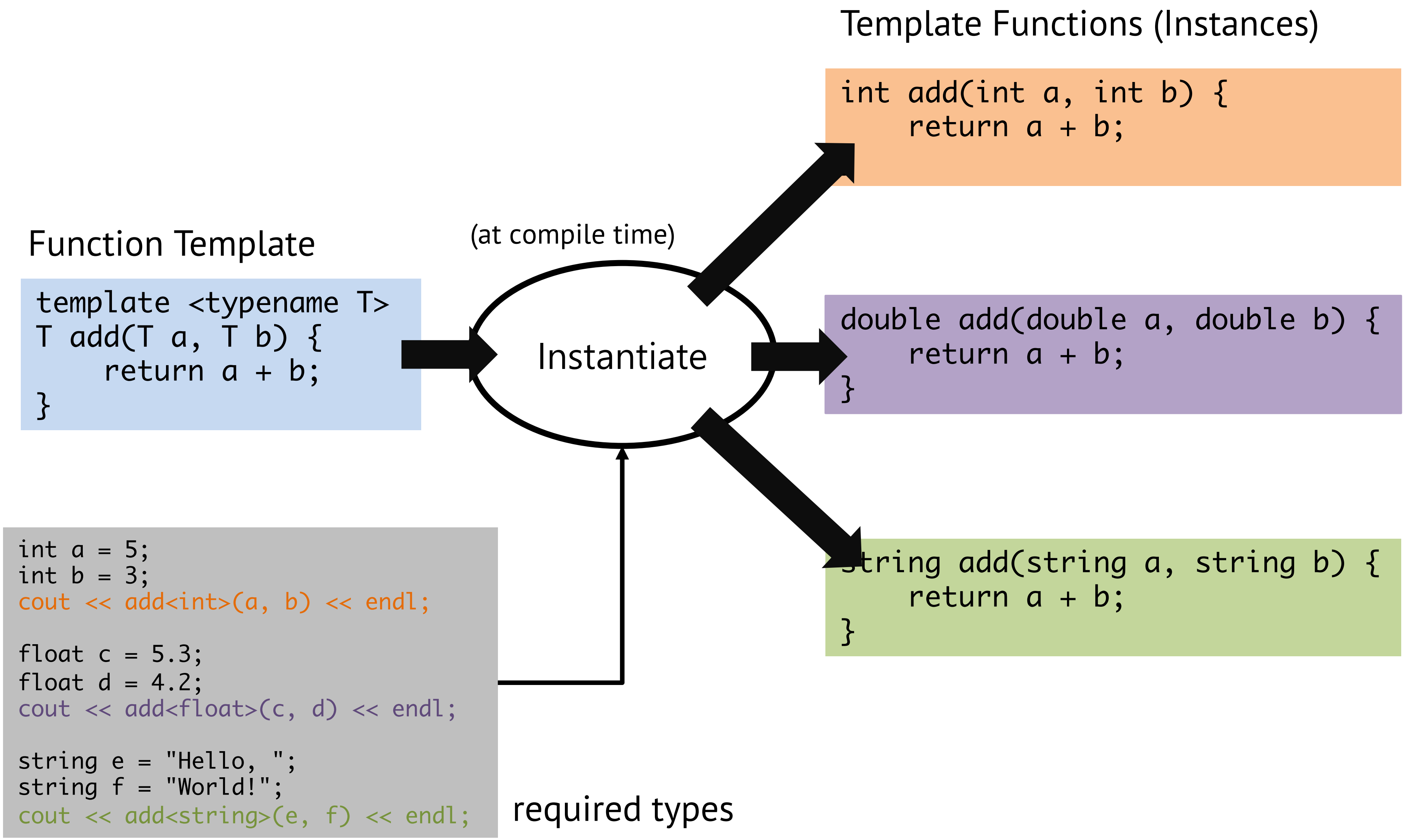
- Lexical Definition
  - *something that is used as a* ***pattern*** *for producing* ***similar things***
    - *(Cambridge Dictionary)*
- In cpp…
  - *something that provides user to make* ***pattern*** *to produce* ***similar functions/classes*** *that corresponds to various data types*

Template

- cpp template
  - supports both *Function* and *Class*
  - powerful tool for *generic programming*
  - Generic Programming?
    - A style of computer programming in which algorithms are written in terms of types *to-be-specified-later* that are then *instantiated* when needed for specific types *(Wikipedia)*
    - Many *static* modern PLs supports generic programming

Template

- How template works?



Template Functions (Instances)

```
int add(int a, int b) {
    return a + b;
```

```
double add(double a, double b) {
    return a + b;
}
```

```
string add(string a, string b) {
    return a + b;
}
```

Function Template

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

(at compile time)

Instantiate

```
int a = 5;
int b = 3;
cout << add<int>(a, b) << endl;

float c = 5.3;
float d = 4.2;
cout << add<float>(c, d) << endl;

string e = "Hello, ";
string f = "World!";
cout << add<string>(e, f) << endl;
```

required types

Template

- So far, we have considered function templates
- How about *Class Templates??*
  - Almost same to function template

Template

- Class Template
  - You want to build a calculator
  - calculator should provide addition, subtraction, multiplication
    - for various types
  - Class Template will help you!!

```
template <typename T>
class Calculator {
public:
    T add(T a, T b) { return a + b; }
    T sub(T a, T b) { return a - b; }
    T mul(T a, T b) { return a * b; }
};
```

Template

- Class Template

```
template <typename T>
class Calculator {
public:
    T add(T a, T b) { return a + b; }
    T sub(T a, T b) { return a - b; }
    T mul(T a, T b) { return a * b; }
};

Calculator<int> calInt;
Calculator<float> calFloat;
cout << calInt.mul(5, 3) << endl;                 \\ 15
cout << calFloat.add(5.3, 3.3) << endl;           \\ 8.6
```

Template

- Template can get multiple arguments

```
template <typename T1, typename T2>
void printArgs(T1 a, T2 b) {
    cout << a;
    cout << " ";
    cout << b << endl;
    return;
}

printArgs<int, string>(4, "abs");        // 4 abs
printArgs<float, int>(3.5, 10)          // 3.5 10
```

Template specialization

- Sometimes template works abnormally for certain data types

```
template <typename T>
class Calculator {
public:
    T add(T a, T b) { return a + b; }
    T sub(T a, T b) { return a - b; }
    T mul(T a, T b) { return a * b; }
};
```

```
Calculator<string> calStr;
cout << calStr.add("abc", "def") << endl   // abcdef
```

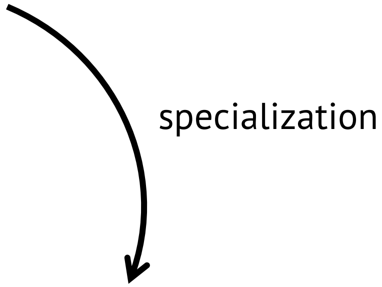
```
Calculator<string> calStr;
cout << calStr.add("abc", "def") << endl;
cout << calStr.sub("abc", "def") << endl;   // Compile error!!!
```

```
error: invalid operands to binary expression
('std::__1::basic_string<char>' and 'std::__1::basic_string<char>')
    T sub(T a, T b) { return a - b; }
```

Template Specialization

- specialization solves the problem!!

```
template <typename T>
class Calculator {
public:
    T add(T a, T b) { return a + b; }
    T sub(T a, T b) { return a - b; }
    T mul(T a, T b) { return a * b; }
};
```



```
template <>
class Calculator<string> {
public:
    string add(string a, string b) { return a + b; }
    string sub(string a, string b) { return "error: impossible operation!"; }
    string mul(string a, string b) { return "error: impossible operation!"; }
};
```

```
Calculator<string> calStr;
cout << calStr.add("abc", "def") << endl;    // abcedf
cout << calStr.sub("abc", "def") << endl;    // error: impossible operation!
```

Template Exercise

- Revisit STL
  - Standard ***Template*** Library
    - To understand & use STL properly, we should understand Template!!
    - Might noticed that…

```
#include <vector>

vector<int> intVec;            // []
intVec.push_back(1);           // [1]
intVec.push_back(2);           // [1, 2]

vector<string> strVec;         // []
strVec.push_back("abc");       // [“abc”]
strVec.push_back("def");       // [“abc”, “def”]
```

- <int>, <string> : template features!!

Template Exercise

- Implement tSTL (tiny STL)!!
  - tSTL consists of vector & stack
    - vector
      - void push_back(T element)
      - void pop_back()
      - int size()
      - resize(int newSize)
      - T front()
      - T end()
      - bool empty()
    - stack
      - void push(T element)
      - void pop()
      - bool empty()
      - T top()
      - int size()

Template Exercise

- tSTL features
  - Vector
    - void push_back(T element)
      - push new element to the end of the vector
    - void pop_back()
      - pop an element from the end of the vector
    - int size()
      - return the number of the elements in the vector
    - void resize(int newSize)
      - resizes the size of the vector
      - if newSize > current size: empty blocks appear
      - if newSize < current size: delete exceeded elements

Template Exercise

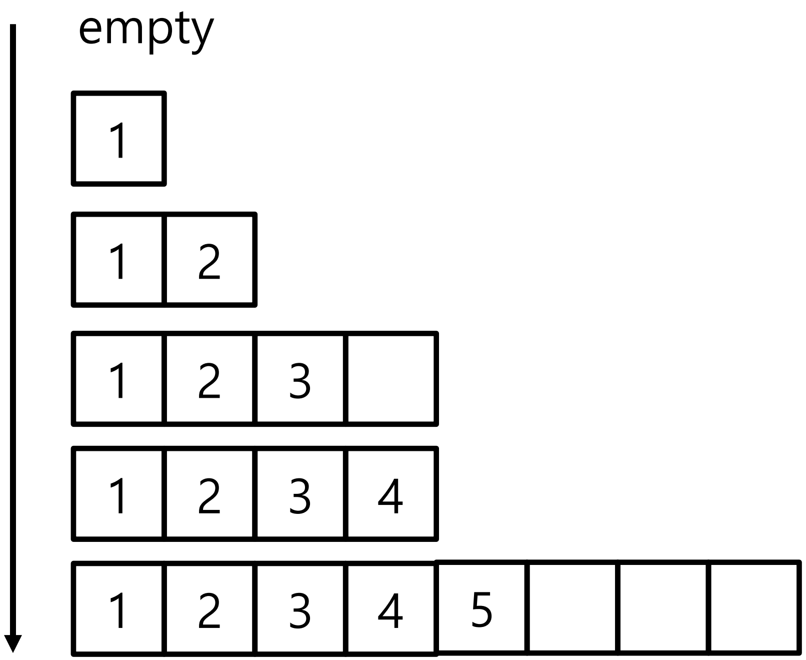
- tSTL features
  - Vector
    - T front()
      - returns the first element of the vector
      - assumes nonempty
      - assumes size = element num (no resize)
    - T back()
      - returns the last element of the vector
      - assumes nonempty
      - assumes size = element num (no resize)
    - bool empty()
      - returns true if current size > 0; otherwise false

Template Exercise

- tSTL features
  - Stack
    - void push(T element)
      - push new element to the top of the stack
    - void pop()
      - pop the top element from the stack
    - T top()
      - returns the top element
      - assumes nonempty
    - int size()
      - returns the size of the stack
    - bool empty()
      - returns true if current size > 0; otherwise false

Template Exercise

- tSTL features
  - How tSTL containers work
    - simple rule!! doubles whenever it needs more



empty

1

1 2

1 2 3

1 2 3 4

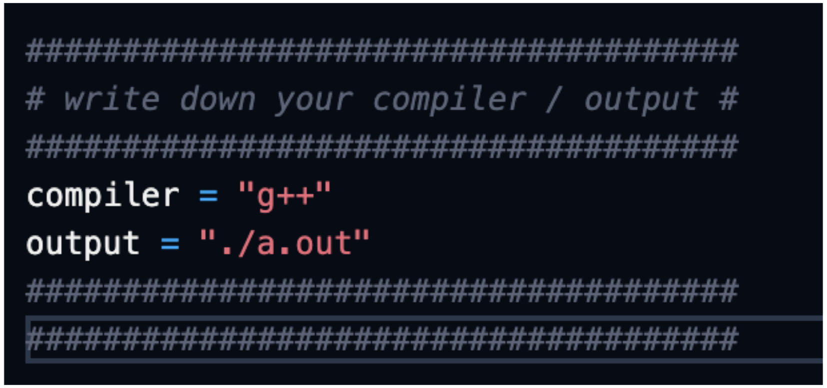
1 2 3 4 5

Template Exercise

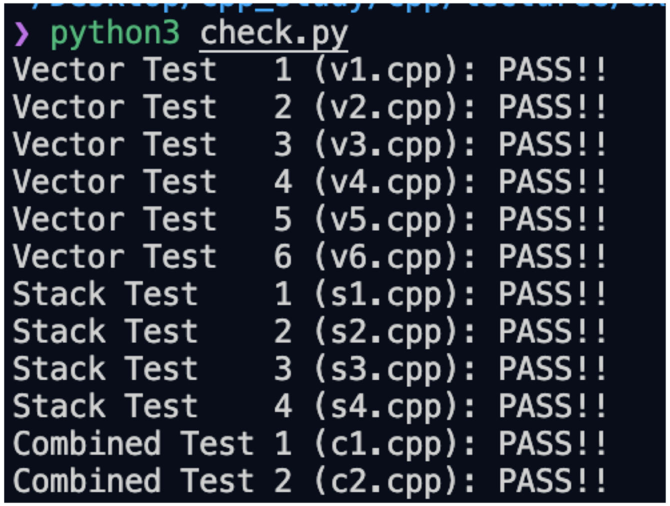
- Implement tSTL!!
  - you must implement src/tVector.h and src/tStack.h
  - Sample code

Template Exercise

- Implement tSTL!!
  - you must implement src/tVector.h and src/tStack.h
  - after the implementation, run check.py
    - you might need to install python3 (DIY)
    - The code will automatically check your answer
    - before running check.py, please write down your compiler / default output format that you use



```
#####################################
# write down your compiler / output #
#####################################
compiler = "g++"
output = "./a.out"
#####################################
#####################################
```



```
❯ python3 check.py
Vector Test   1 (v1.cpp): PASS!!
Vector Test   2 (v2.cpp): PASS!!
Vector Test   3 (v3.cpp): PASS!!
Vector Test   4 (v4.cpp): PASS!!
Vector Test   5 (v5.cpp): PASS!!
Vector Test   6 (v6.cpp): PASS!!
Stack Test    1 (s1.cpp): PASS!!
Stack Test    2 (s2.cpp): PASS!!
Stack Test    3 (s3.cpp): PASS!!
Stack Test    4 (s4.cpp): PASS!!
Combined Test 1 (c1.cpp): PASS!!
Combined Test 2 (c2.cpp): PASS!!
```

Thank you!!

contact: jeonhyun97@postech.ac.kr