#include <sstream>

CPP Advance:

Template

Template

- Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type. This support developer avoid write same codes for each data type.
- Another form of polymorphism
- Further simplifies the function overloading
- C++ adds two new keywords to support templates: *'template'* and *'typename'*. The second keyword can always be replaced by keyword 'class'.
- There are 2 types of template:
  - ✓ Function template
  - ✓ Class template

nền tảng
lập trình tổng quát
bao gồm
cụ thể
tránh
Templates là nền tảng của lập trình tổng quát (generic programming), bao gồm việc viết mã theo cách không phụ thuộc vào bất kỳ kiểu dữ liệu cụ thể nào. Điều này giúp lập trình viên tránh phải viết cùng một mã cho mỗi kiểu dữ liệu.

Templates là một dạng khác của tính đa hình (polymorphism).

Templates còn giúp đơn giản hóa việc nạp chồng hàm (function overloading).

thay thế
C++ bổ sung hai từ khóa mới để hỗ trợ templates: template và typename. Từ khóa thứ hai có thể luôn được thay thế bằng từ khóa class.

CONFIDENTIAL

Function Template: Purpose

xem xét

- Consider the following function:

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

- It only works for the int type of variable ! Hàm này chỉ hoạt động với kiểu biến int!
- How can we make it work for other types of variables ?

Làm thế nào để chúng ta làm cho nó hoạt động với các kiểu biến khác?

CONFIDENTIAL

Function Template: Purpose

- Use function overloading !

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

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

char max (char x, char y)
{
      return (x > y? x : y);
}

 ……
```

- Do we have a smarter way to do that ?
- This way:

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

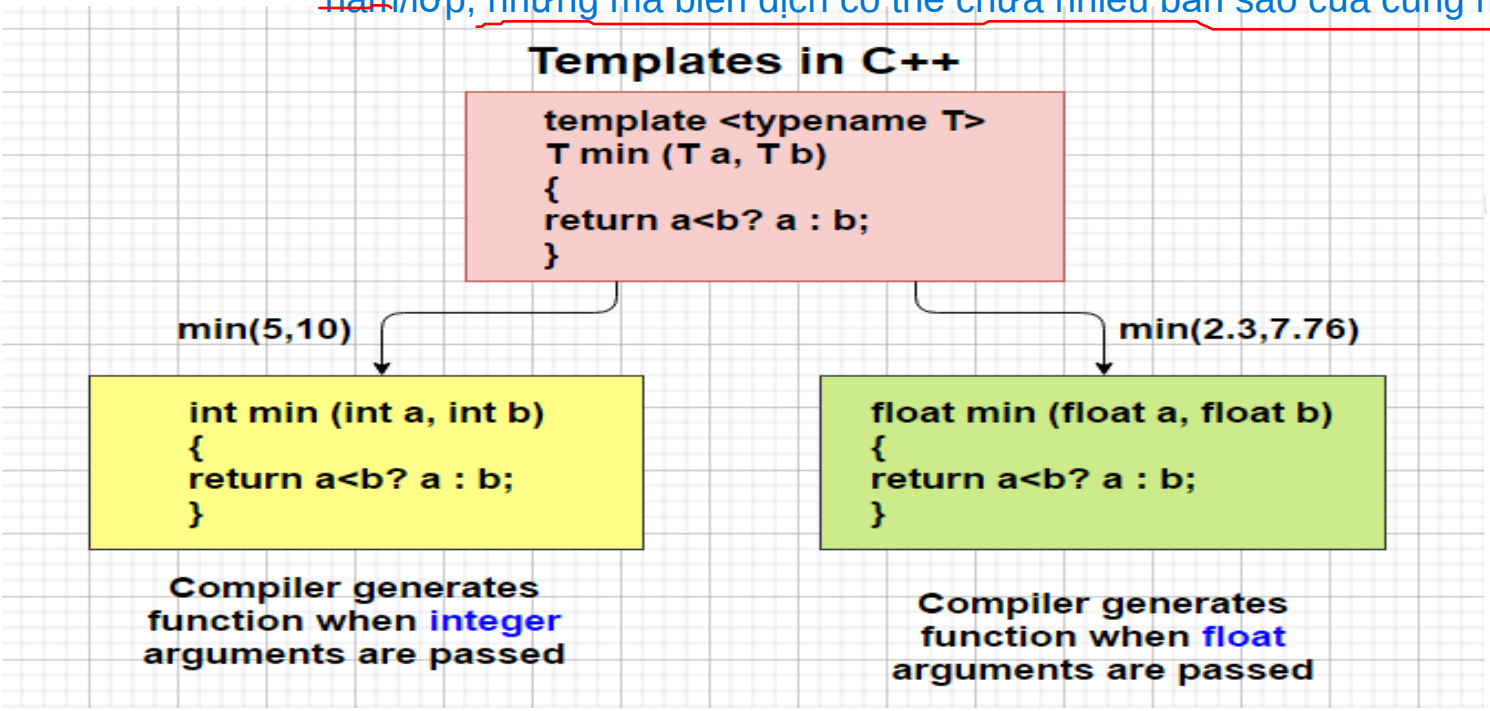
Type parameter

Function Template: Principle

nguyên tắc

- **How templates work?**
  - ✓ Templates are expanded at compiler time. This is like macros. The difference is, compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.

Templates được mở rộng tại thời điểm biên dịch. Điều này giống như macros. Sự khác biệt là, trình biên dịch thực hiện kiểm tra kiểu trước khi mở rộng template. Ý tưởng rất đơn giản, mã nguồn chỉ chứa một hàm/lớp, nhưng mã biên dịch có thể chứa nhiều bản sao của cùng một hàm/lớp.



Function Template: Principle

- **Why follow example have problem when compile?**

```
// declare template function
template<class T>
void showStuff(int stuff1, T stuff2,
               T stuff3);
……

// define the template function
template<class T>
void showStuff(int stuff1, T stuff2,
               T stuff3)
{
        cout << stuff1 << endl
             << stuff2 << endl
             << stuff3 << endl;
}

class ABC{
public:
    ABC(float inx=0,
        float iny=0)
    {x=inx; y=iny;}
private:
   float x, y;
};
```

```
void main() {
     int n1 = -5, n2 = 0.5;
     double d1 = -5.5, d2 = -5.9;
     ABC obj1(3, 4.4),obj2(3.1, 0.2);
     showStuff(1, n1, n2);
     showStuff(2, d1, d2);
     showStuff(3, obj1, obj2);
}
```

chú ý khi truyền kiểu dữ liệu vào trong template phải kiểm tra xem chúng có tương thích với hàm hoặc class template không

Sai đơn giản vì class ABC không có operator << nên không chạy được

cú pháp template<class T> : class ở đây không phải là 1 lớp

"cout<<" cannot handle ABC type of objects!

Solution?

Function Template : Format

The format :

```
template <class type>
ret-type func-name(parameter list)
{
        // body of function
}
```

bắt buộc luôn phải có khai báo template trên cùng

Example:

```
// 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: Multiple Type Parameters

- **Can there be more than one arguments to function templates?** Có thể có nhiều tham số cho function template không?
  - ✓ Yes, like normal parameters, we can pass more than one data types as arguments to templates.

Có, giống như các tham số thông thường, chúng ta có thể truyền nhiều kiểu dữ liệu làm tham số cho các mẫu hàm.

- Cannot have “unused” template parameters:
  - ✓ Each must be used in definition
  - ✓ Error otherwise !

Không thể có các tham số mẫu "không sử dụng":
Mỗi tham số phải được sử dụng trong định nghĩa.
Nếu không, sẽ xảy ra lỗi!

```
template <class T, class X>
void someFunction(T arg1, X arg2)
{
   ...
}
```

```
// function template
 #include <iostream>
using namespace std;
template <class T, class U>
T multiple(T a, U b)
{
     U result = a * b;
      return result;
}
int main ()
{
                int i=5;
                double j = 1.5;
                double result = multiple<int, double>(i, j);
                return 0;
}
```

Function Template: Format

- What is the problem of the following template functions ?

```
template<class T1, class T2>
T1 max (T1 num1, T1 num1)
{
    return num1 > num2 ? num1 : num2;
}

T2 min (T2 num1, T2 num2)
{
    return num1 > num2 ? num2 : num1;
}
```

→ chưa được dùng

Sai, không viết chung như thế này được

ENTIAL

Function Template: Format

- What is the problem of the following template functions ?

```
template<class T1, class T2>
T1 max (T1 num1, T1 num1)
{
    return num1 > num2 ? num1 : num2;
}

T2 min (T2 num1, T2 num2)
{
    return num1 > num2 ? num2 : num1;
}
```

T2 is not used!

Missing a template prefix!

TIAL

Class Template: Format

- We can also define template class, which can be considered as the “generalized” classes.
  - Chúng ta cũng có thể định nghĩa mẫu lớp, có thể được coi là các lớp "tổng quát hóa".

tổng quát hóa

```
template<class T>
class Pair
{
public:
        Pair();
        Pair(T firstVal, T secondVal);
        void setFirst(T newVal);
        void setSecond(T newVal);
        T getFirst() const;
        T getSecond() const;
private:
        T first;
        T second;
};
```

template prefix

declaration of the template class

```
template<class T>
Pair<T>::Pair(T firstVal, T secondVal)
{
        first = firstVal;
        second = secondVal;
}
```

template prefix

definition of the member functions of the template class

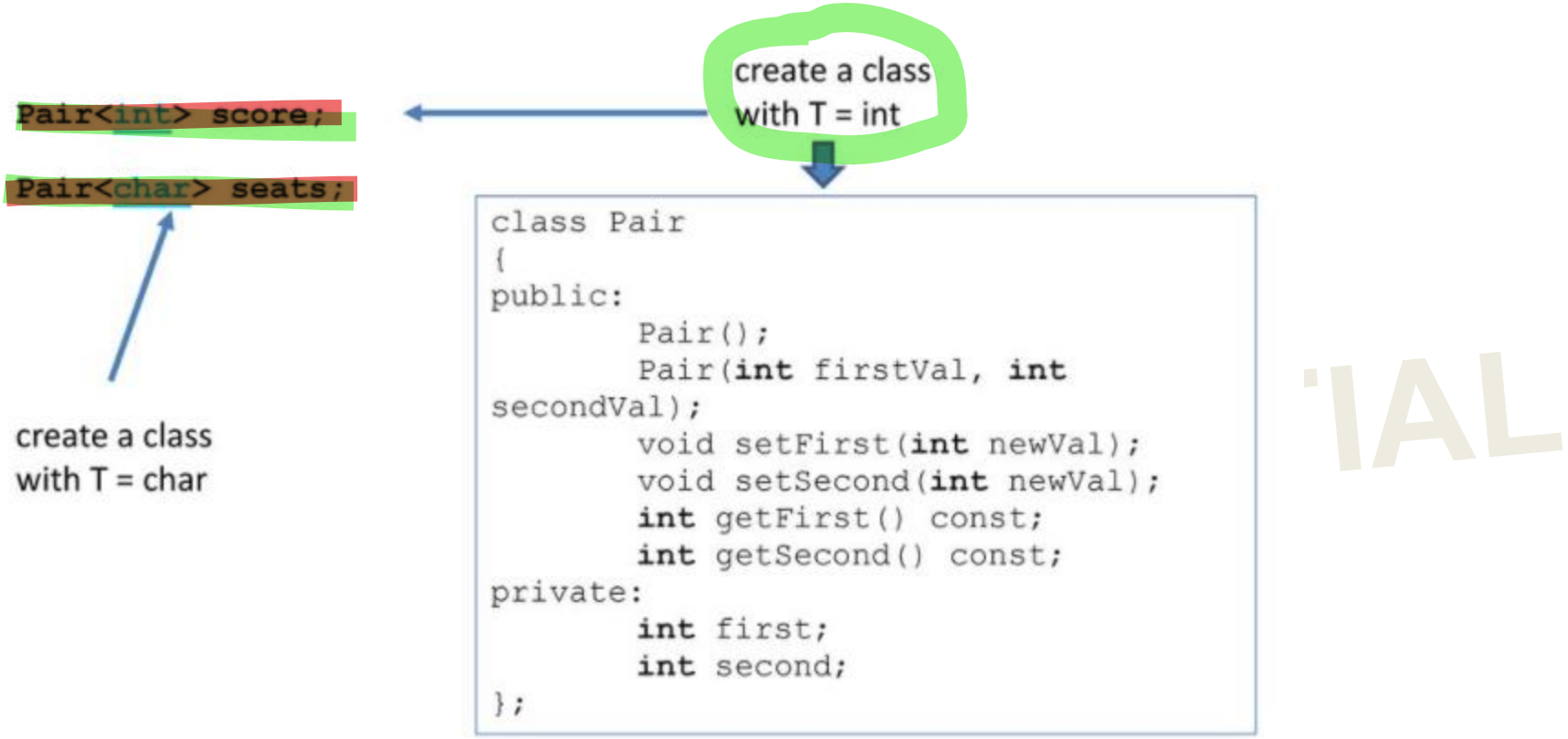
```
template<class T>
void Pair<T>::setFirst(T newVal)
{
        first = newVal;
}
```

Need to repeat this prefix for each member function

Class name before:: is “**Pair<T>**”!

Class Template: How to Use

- How to use the template class ?



Class Template: How to Use

- How to use the template class as the parameter of a function ?

```
int addUP(const Pair<int>& the_Pair);
```

khi muốn sử dụng các lớp tổng quát hóa (class template), cần phải truyền cả kiểu dữ liệu mà bạn muốn sử dụng.

create a class with T = int

```
class Pair
{
public:
        Pair();
        Pair(int firstVal, int
secondVal);
        void setFirst(int newVal);
        void setSecond(int newVal);
        int getFirst() const;
        int getSecond() const;
private:
        int first;
        int second;
};
```

Overall, template types can be used anywhere standard types can

- We can further define the following function as a template to avoid defining multiple overload Chúng ta có thể tiếp tục định nghĩa hàm sau đây dưới dạng một template để tránh việc phải định nghĩa nhiều overload.

```
int addUP(const Pair<int>& the_Pair);
char addUP(const Pair<char>& the_Pair);
float addUP(const Pair<float>& the_Pair);
……
```

tên class đi cùng với kiểu dữ liệu
tên function thì không cần

```
template<class T>
T addUp(const Pair<T> & the_pair)
{
    return the_pair.first + the_pair.second;
}
```

CONFIDENTIAL

Class Template

- **Can there be more than one arguments to class templates?**
  - ✓ Yes, like normal parameters, we can pass more than one data types as arguments to templates.

```
template <class T, class U>
Class class-name {
        T x;
        U y;
….
};
```

```
// function template
 #include <iostream>
using namespace std;
template <class T, class U>
class A  {
        T  x;
        U  y;
public:
        void func();
};
int main ()
{
        A< char, char> a;
        A< int, double> b;
        return 0;
}
```

Class Template

- **Can we declare a function of class as a template function ? Yes**

  Chúng ta có thể khai báo một hàm của lớp dưới dạng hàm template không? Có.

Format:

```
class class-name
{
            . . .
template<typename T>
returnType func(T arg)
} ;
template<typename T>
returnType class-name::func(T arg)
{
....
}
```

Example:

```
// class templates
#include <iostream>
using namespace std;
class MyClass {
...
public:
    template<typename T>
    T add(T first, T second);
};
template<typename T>
T MyClass::add(T first, T second)
{
    return first*second;
}

int main ()
{
      MyClass obj;
      obj.add(10,40);
      obj.add(10.4, 10.5);
      return 0;
}
```

Template Non-Type Parameter

- A non-type template argument provided within a template argument list is an expression whose value can be determined at compile time. Such arguments must be constant expressions.

Một tham số template không phải kiểu được cung cấp trong danh sách tham số template là một biểu thức mà giá trị của nó có thể được xác định tại thời gian biên dịch. Những tham số như vậy phải là biểu thức hằng.

```
class class-name
{
            . . .
template<typename T>
returnType func(T arg)
} ;
template<typename T>
returnType class-name<T>::func(T arg)
{
....
}
```

```
// class templates
#include <iostream>
using namespace std;
template <class T, int size>
class MyArray {
    T data[size];
public:
     T& operator[](int index)
     {
        return data[index];
     }
};
int main ()
{
       MyArray< int, 20> obj;
       int value = obj[0];
       return 0;
}
```

phải là hằng hoặc biểu thức hằng

non-type là cách truyền một giá trị hoặc biểu thức có thể xác định được tại thời điểm biên dịch vào template. Những đối số này phải là biểu thức hằng số.

Template: Partial Specialization

khác với sự đặc tả một phần

- In contrast of a full template specialization partial template specialization allows to introduce template with some of the arguments of existing template fixed.

Khác với sự đặc tả đầy đủ, đặc tả một phần của template cho phép định nghĩa một template mới với một số tham số đã cố định của template hiện có.

- Partial template specialization is only available for template class.

Đặc tả một phần của template chỉ có sẵn cho các lớp template.

- When a partially specialized template is instantiated, the most suitable specialization is selected.

khởi tạo

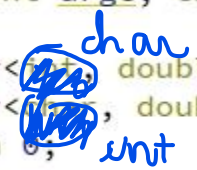
Khi một template được đặc tả một phần được khởi tạo, phiên bản đặc tả phù hợp nhất sẽ được lựa chọn.

CONFIDENTIAL

Template: Partial Specialization

```
template <class T, class U>
class MyPair {
public:
    MyPair(T firstValue, U secondValue)
    {
        first = firstValue;
        second = secondValue;
        std::cout<<"General Template";
    }
private:
    T first;
    U second;
};
template <class V>
class MyPair<int, V> {
public:
    MyPair(int firstValue, V secondValue)
    {
        first = firstValue;
        second = secondValue;
        std::cout<<"Special Template";
    }
private:
    int first;
    V second;
};
int main(int argc, char *argv[])
{
    MyPair<char, double> m1('s', 4.5);
    MyPair<int, double> m2(1, 5.0);
    return 0;
}
```

chuyên hóa 1 phần, khi truyền vào kiểu int thì sẽ đặc biệt hơn



Template: Partial Specialization

- Function templates may only be fully specialized

function template chỉ có thể đặc tả toàn bộ

```
template<typename T, typename U>
void foo(T t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
}

// fully specialization -> OK.
template<>
void foo<int, int>(int a1, int a2) {
    std::cout << "Two ints: " << a1 << " " << a2 << std::endl;
}
// Compilation error: partial function specialization is not allowed.
template<typename U>
void foo<std::string, U>(std::string t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
}
```

chuyên hóa toàn bộ, viết riêng cho <int, int>

với temple function thì bắt buộc phải chuyên hóa toàn bộ

Thank you