UNIVERSITY
OF WOLLONGONG
AUSTRALIA

CSCI251

Advanced Programming

C++ Function and Class Template

Outline

Object Oriented Programming in C++

Function Template

1. Problem
2. Concept - Syntax
3. Example 1
4. Practice 1

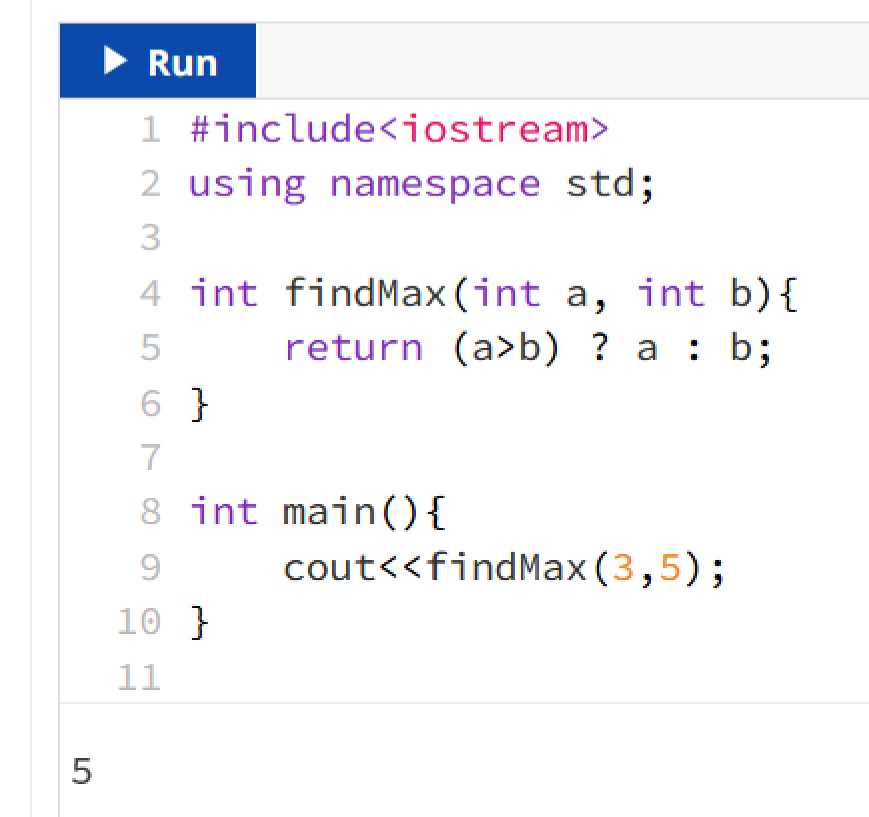
Function Template and more

5. Overloading
6. Multiple parameters
7. Non-type arguments
8. Explicit specialization
9. Practice 2

Class Template

10. Concept
11. Example and Practice 3

Problem: write a function to find the maximum of two integer numbers (Using conditional operators)



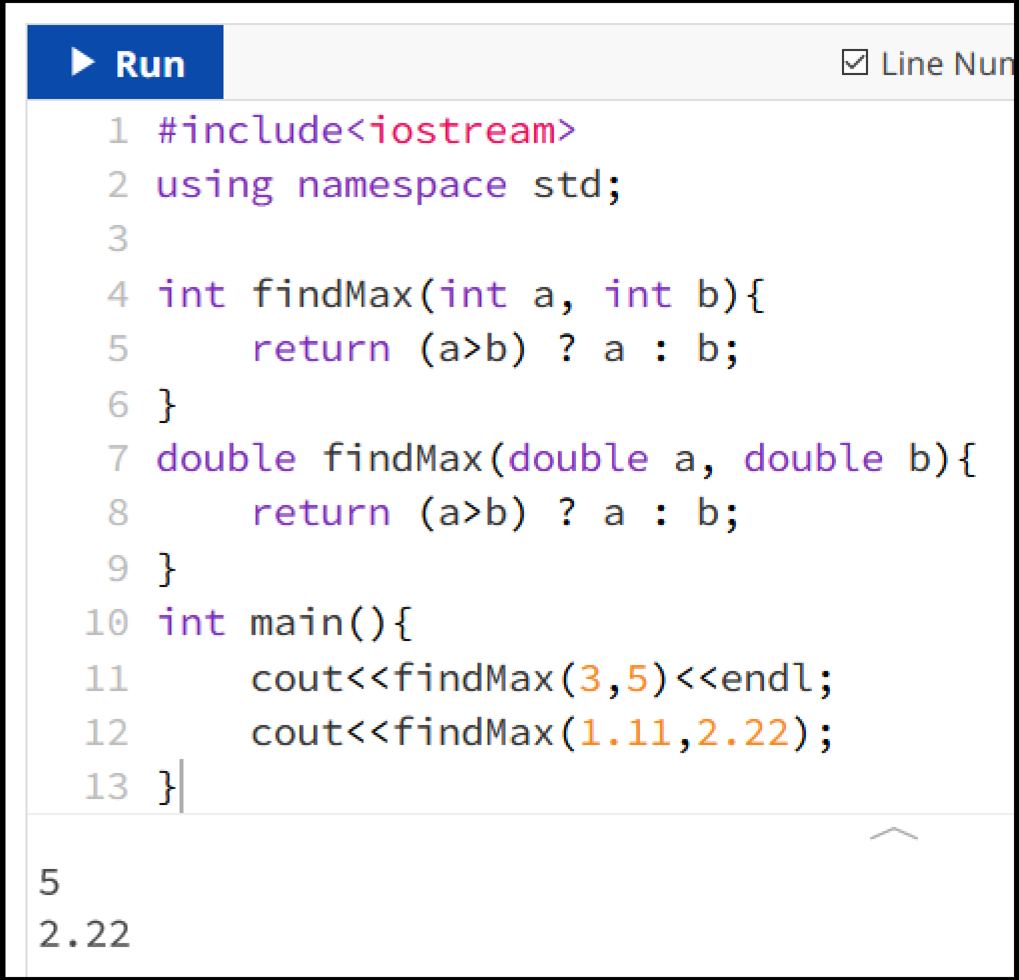
```cpp
#include<iostream>
using namespace std;

int findMax(int a, int b){
    return (a>b) ? a : b;
}

int main(){
    cout<<findMax(3,5);
}

```

```
5
```

Another problem: Can we write another function to find the maximum of double numbers?

YES: Function overloading

findMax with int and double data



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

int findMax(int a, int b){
    return (a>b) ? a : b;
}
double findMax(double a, double b){
    return (a>b) ? a : b;
}
int main(){
    cout<<findMax(3,5)<<endl;
    cout<<findMax(1.11,2.22);
}
```

```
5
2.22
```

What is the point here?

Answer: We have to write another overloaded function

Is there any way that we do not need to write another overloaded function?

Answer: YES

Function Template

What is it?

Function Template, is a type of generic function programming,

It provides a function blueprint that uses generic data types

It defines a group of functions which may be generated by the compiler with different types of function parameters

What is it?

You write a function template and the compiler generates one or more template functions, assuming there is at least one call to the function template

Why we use function template

- Save code
- Re-use
- Easy to create and edit

How to use it - Syntax

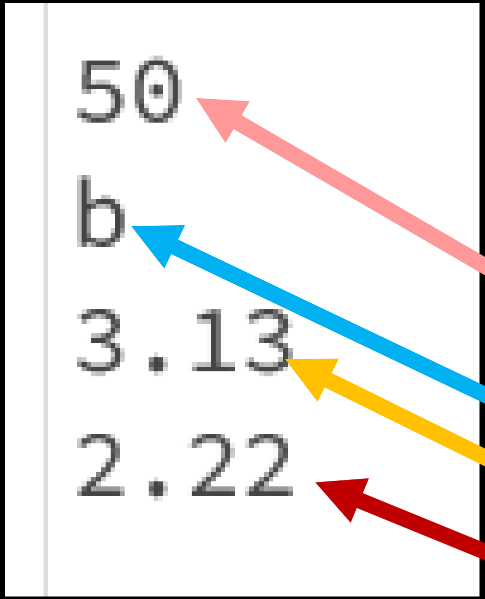
```
template <typename T>
return_type function_name (function_parameter_list) {
    function_body;
}
```

OR

```
template <class T>
return_type function_name (function_parameter_list) {
    function_body;
}
```

Example 1: findMax

```
template <typename T>
return_type function_name (function_parameter_list
) {
    function_body;
}
```

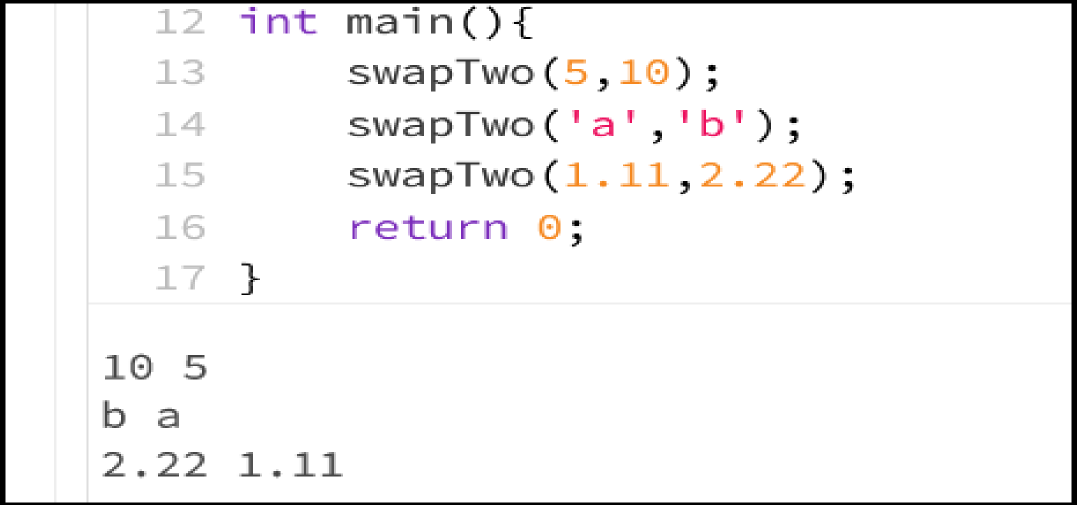


```
50
b
3.13
2.22
```

```
// finMax.cpp
#include<iostream>
using namespace std;

template<typename T>
T findMax(T a, T b){
    return (a>b) ? a : b;
}
int main(){
    cout << findMax(20, 50)<<endl;
    //cout << findMax<int>(20, 50)<<endl;
    cout << findMax('a', 'b')<<endl;
    cout << findMax(3.12f, 3.13f)<<endl;
    cout << findMax(1.11, 2.22)<<endl;
    return 0;
}
```

Practice 1: swapTwo

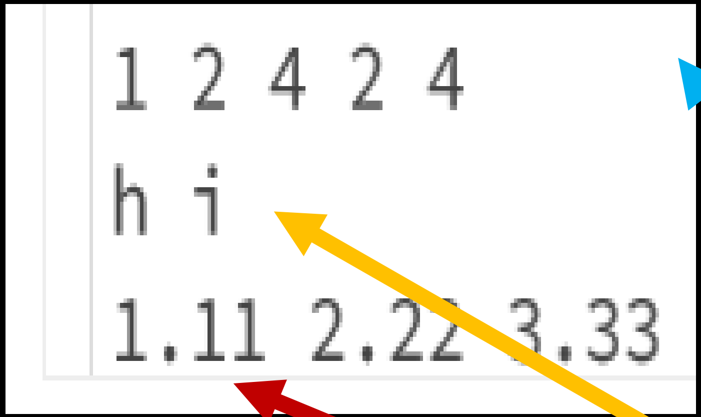
- Do on Code:Block - write a template function, swapTwo, to swap values of two variables
- You can use the code from the lecture.
- Create the swapTwo template function
  - ➢ Using template<typename T> or template<class T>
  - ➢ T swapTwo(T a, T b){… cout<<a<<“ ”<<b;}
- In main() function:
  - ➢ Call the template functions to swap 3 pairs data types: integer, char, double like the picture



```
int main(){
    swapTwo(5,10);
    swapTwo('a','b');
    swapTwo(1.11,2.22);
    return 0;
}
```

```
10 5
b a
2.22 1.11
```

Example 2: showArray



```
1 2 4 2 4
h i
1.11 2.22 3.33
```

```cpp
//showArr.cpp
#include<iostream>
using namespace std;

template<typename T>
void showArr(T* arr, size_t size){
    for (size_t i=0; i < size; i++){
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main(){
    int a[]={1, 2, 4, 2, 4};
    char b[]={'h', 'i'};
    double c[]={1.11, 2.22, 3.33};
    showArr<int>(a, sizeof(a)/sizeof(int));
    showArr<char>(b, sizeof(b)/sizeof(char));
    showArr<double>(c, sizeof(c)/sizeof(double));
    return 0;
}
```

Function template overloading

Swap

integers

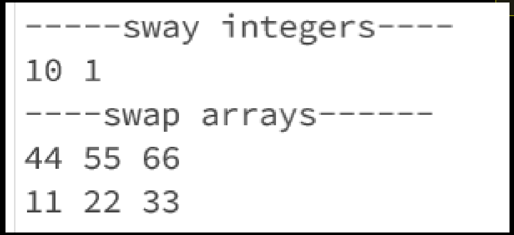
vs

Swap

Arrays

```
//swapArr.cpp
#include <iostream>
using namespace std;

template <typename T>
void mySwap(T &a, T &b);

template <typename T>
void mySwap(T a[], T b[], int size);

template <typename T>
void showArr(T * arr, const int size);

int main() {
   cout<<"-----sway integers----"<<endl;
   int a = 1, b = 10;
   mySwap(a, b);   // Compiler generates mySwap(int &, i
nt &)
   cout<<"----swap arrays------"<<endl;
   const int SIZE = 3;
   int ar1[] = {11, 22, 33}, ar2[] = {44, 55, 66};
   mySwap(ar1, ar2, SIZE);
   showArr(ar1, SIZE);
   showArr(ar2, SIZE);
}
```

```
template <typename T>
void mySwap(T &a, T &b) {
   T temp;
   temp = a;
   a = b;
   b = temp;
   cout << a << " " << b <<endl;
}

template <typename T>
void mySwap(T a[], T b[], int size) {
   T temp;
   for (int i = 0; i < size; ++i) {
      temp = a[i];
      a[i] = b[i];
      b[i] = temp;
   }
}

template<typename T>
void showArr(T* arr, const int size){
    for (size_t i=0; i < size; i++){
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

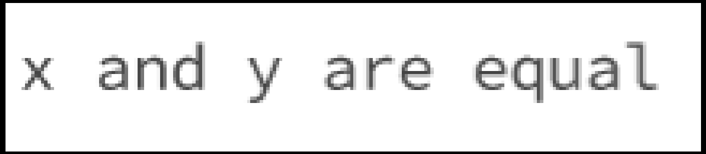


```
-----sway integers----
10 1
----swap arrays------
44 55 66
11 22 33
```

Multiple template parameters

Compare two different

types of data numbers



```
x and y are equal
```

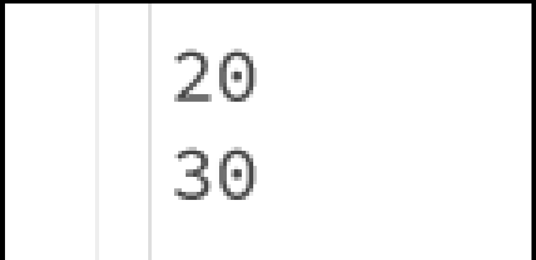
```
// multiPara.cpp
#include <iostream>
using namespace std;

template <class T, class U>
bool are_equal (T a, U b)
{
  return (a==b);
}

int main ()
{
  if (are_equal(10,10.0))
    cout << "x and y are equal\n";
  else
    cout << "x and y are not equal\
n";
  return 0;
}
```

https://www.cplusplus.com/doc/tutorial/functions2/

Non-type template arguments

The template parameters can not only include types introduced by class or typename, but can also include expressions of a particular type:

```
//nonTemplate.cpp
#include <iostream>
using namespace std;

template <class T, int N>
T fixed_multiply (T val)
{
  return val * N;
}

int main() {
  std::cout << fixed_multiply<int,2>(10) << '\n';
  std::cout << fixed_multiply<int,3>(10) << '\n';
}
```

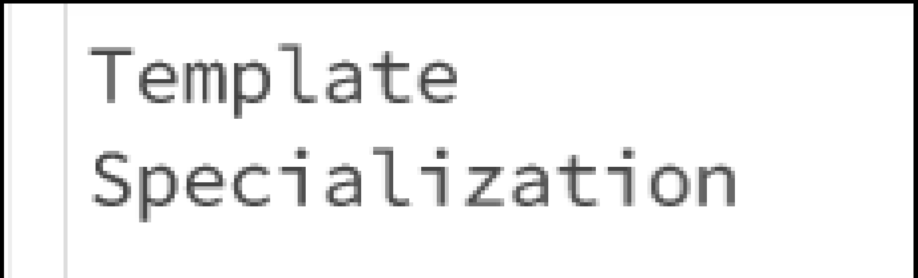


```
20
30
```

https://www.cplusplus.com/doc/tutorial/functions2/

Explicit Specialization

If there is any non-template definition that matches the function call. The non-template version will take precedence over explicit specialization, then template.

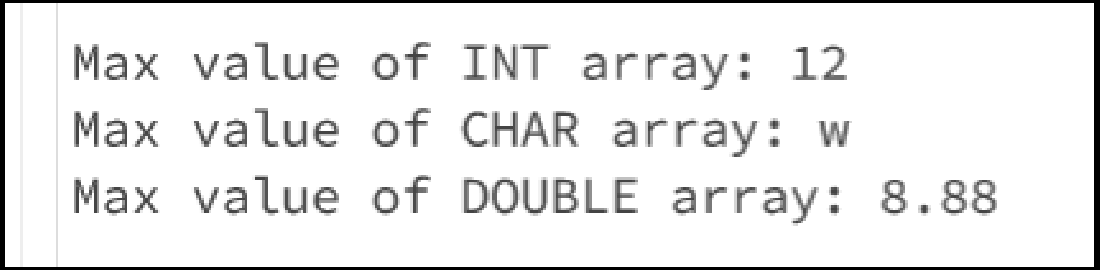


```
Template
Specialization
```

```cpp
//explicit.cpp
#include <iostream>
using namespace std;

// Template
template <typename T>
void mySwap(T &a, T &b) {
   cout << "Template" << endl;
   T temp;
   temp = a;
   a = b;
   b = temp;
}
// Explicit Specialization for type int
template <>
void mySwap<int>(int &a, int &b) {
   cout << "Specialization" << endl;
   int temp;
   temp = a;
   a = b;
   b = temp;
}
int main() {
   double a = 1, b = 2;
   mySwap(a, b);   // use template
   int c = 1, d = 2;
   mySwap(c, d);   // use specialization
}
```

Practice 2: Max of Array

- Do on Code:Block IDE
- Create the findMaxArr template function
  - ➢ Using template<typename T> or template<class T>
  - ➢ Use for loop to find the Max Value
- In main() function:
  - ➢ Call the template functions to create 3 Array with data types: integer, char, double
  - ➢ Cout to the screen the max value of each array like the picture below



```
Max value of INT array: 12
Max value of CHAR array: w
Max value of DOUBLE array: 8.88
```

Class Template

What is Class Template?

If we need to create several similar classes, it may be useful to consider developing a class template in which at least one type is generic or parameterized.

The syntax for class template is similar to that for function templates.

So we write a class template that the compiler turns into template classes.

What is Class Template?

Classes are blueprints for objects, so since templates provide another layer of blueprint, class templates can be thought of as being blueprints for blueprints

Class templates are sometimes referred to as parameterized types, effectively incomplete types where there is a to-be-specified type.

Class Template - Syntax

To build Class template

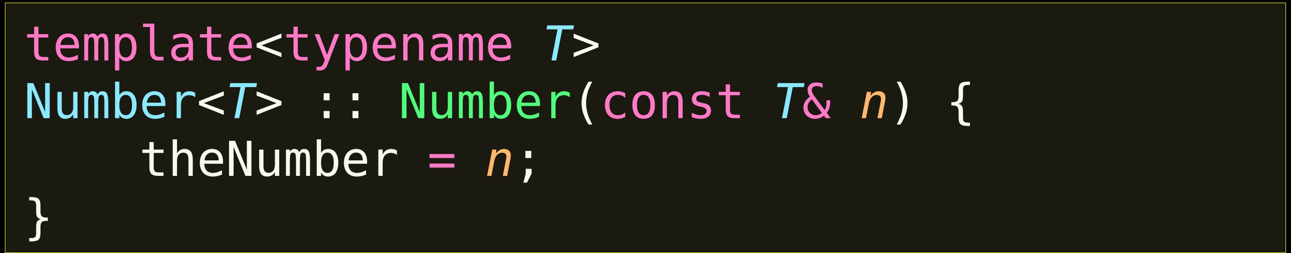
```
template <class T>// OR template <typename T>
class ClassName {
    ......
};
```

To create an object from Class template

```
ClassName<actual-type> obj;
```

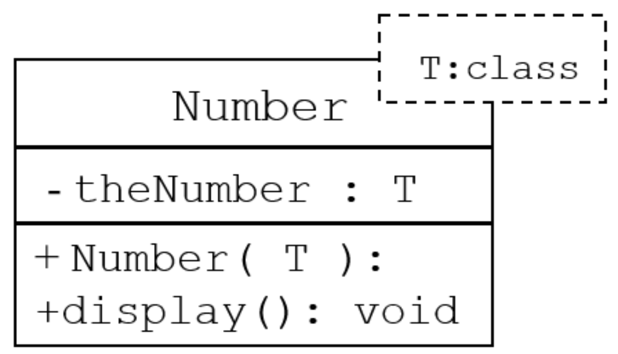
Separating Template Function Declaration and Definition

- When separating the function implementation, we use keyword template <typename T> on each of the function implementation. For example,



```
template<typename T>
Number<T> :: Number(const T& n) {
    theNumber = n;
}
```

Example with UML



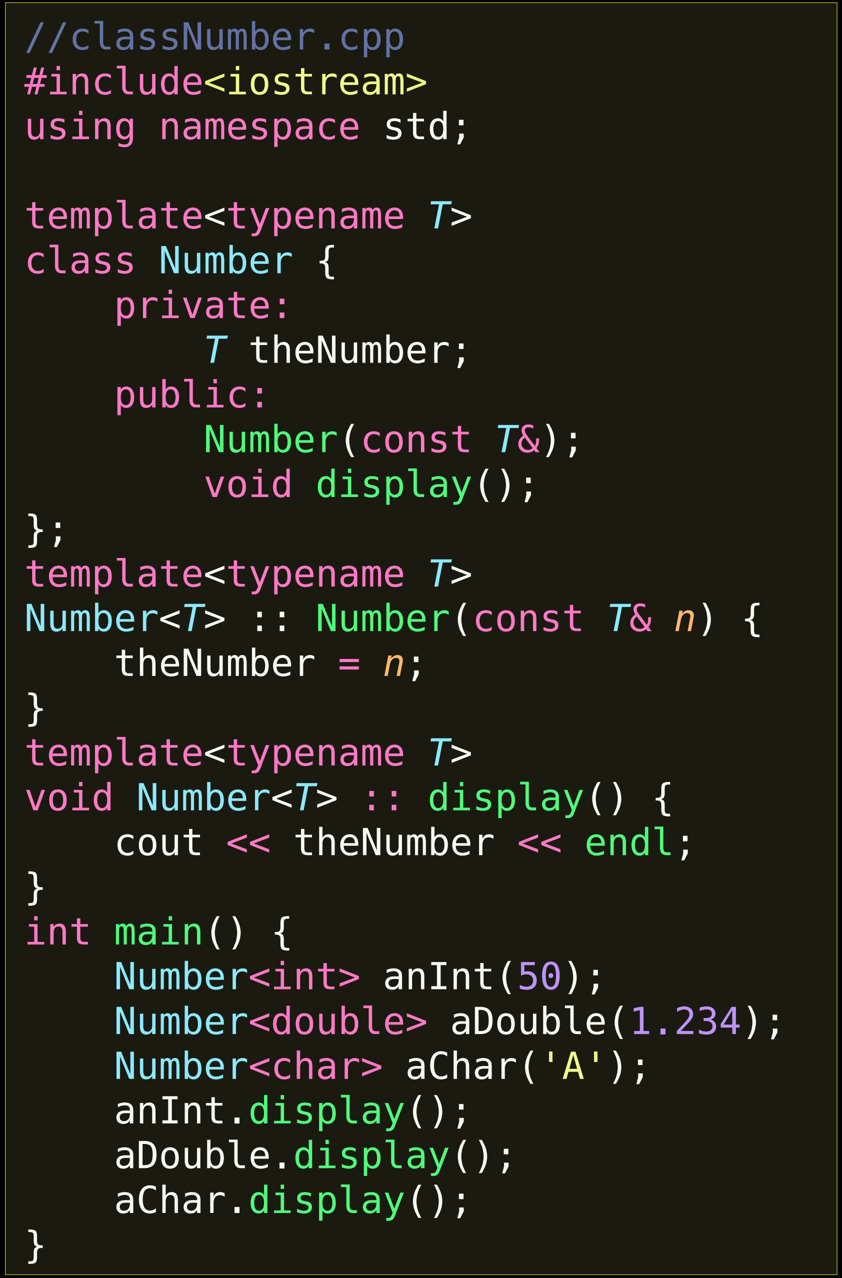
We define typename T

Whenever we use the same data type

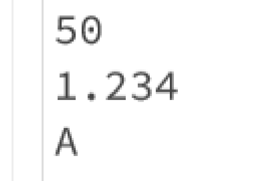
inside the class, we set type T.

In the main we can call with any type of

data int, double, char.



```cpp
//classNumber.cpp
#include<iostream>
using namespace std;

template<typename T>
class Number {
    private:
        T theNumber;
    public:
        Number(const T&);
        void display();
};
template<typename T>
Number<T> :: Number(const T& n) {
    theNumber = n;
}
template<typename T>
void Number<T> :: display() {
    cout << theNumber << endl;
}
int main() {
    Number<int> anInt(50);
    Number<double> aDouble(1.234);
    Number<char> aChar('A');
    anInt.display();
    aDouble.display();
    aChar.display();
}
```



```
50
1.234
A
```

Another Example getMax

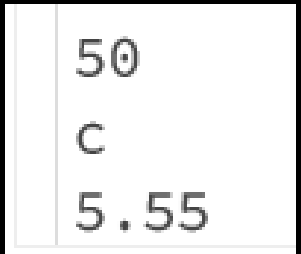
Create a template class mypair to store two elements with any data type.

Inside class, define a non-default constructor (pass by value).

Declare function getMax return data type

Define this function outside the class to get the max of the two numbers.

```cpp
//classMax.cpp
#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 ()
{
  return (a>b) ? a : b;
}

int main () {
  mypair <int> intPair (10, 50);
  mypair <char> charPair ('a', 'c');
  mypair <double> doublePair (1.11, 5.55
);
  cout << intPair.getMax()<<endl;
  cout << charPair.getMax()<<endl;
  cout << doublePair.getMax();
  return 0;
}
```



```
50
c
5.55
```

Practice 3: getmin

- Do on Code:Block:
- Declare getmin inside class
- Then define getmin outside class
  - ➢ Using `template <class T> T mypair<T>::getmin()`
- In main() function:
  - ➢ Create 3 objects with data types: integer, char, double
  - ➢ Cout to the screen the min value of each pair

Where to place code for class templates

- Templates are not class nor member function definition. They are instructions to the C++ compiler how to generate the class definition. Hence, placing member functions in a separate file will not work
- The template codes shall be placed in the header file - to be included in all files using the template. Template cannot be compiled separately.
- A particular realization of template is called an instantiation or specialization. The C++ compiler generate a class for each of the parameterized type used in the program

Multiple Type Parameters

- Like function template, for class template we also have multiple type parameters.
- The syntax is:

```
template <typename T1, typename T2, ....>
class ClassName { ...... }
```

Default Type

- You can also provide default type in template. For example we have the type in integer.

```
template <typename T = int>
class ClassName { ...... }
```

- To create object for the default type we can use syntax with <>. We do not need to declare the actual-type.

```
ClassName<> obj;
```

Specialization

- Like function template, specialization will be applied we call the objects match with type data. We can define a different implementation for a template.

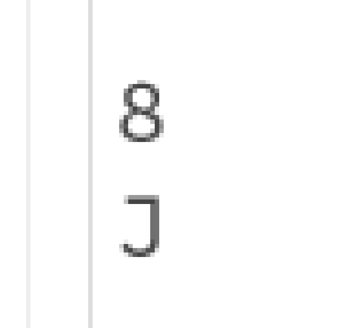
```
// General Template
template <typename T>
class Complex { ...... }

// Specialization for type double
template <>
class Complex<double> { ...... }

// Specialization for type int
template <>
class Complex<int> { ....... }
```

Specialization - Demo

- We have a template mycontainer that stores one element of any type and have one member function. For int we have increase function. But for char type we will have uppercase as a member function.
- So, we will have a class template specialization for that types as follows:

Specialization - Demo

```
int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```



```
8
J
```

```
// classSpecial.cpp
//template specialization
#include <iostream>
using namespace std;
// class template:
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};

// class template specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg) {element=arg;}
    char uppercase ()
    {
      if ((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};
```

Practice 4: specialization

- Do on Code:Block
- For the template, int type is decrease the number, for char type is to add lowercase function.
- In main() function:
  - ➢ Create 2 objects with data types: integer and char
  - ➢ Calling method decrease() and lowercase() from appropriate objects.

Demo – Complex number

- We will play with complex number and use multiple files (.h and .cpp)
- Template class will be stored in a header file (complex.h)
- We will demo to use template class in cpp file (testcomplex.cpp)
- Define complex number

```
complex_num = a*i +b;
square(i) = -1;
```

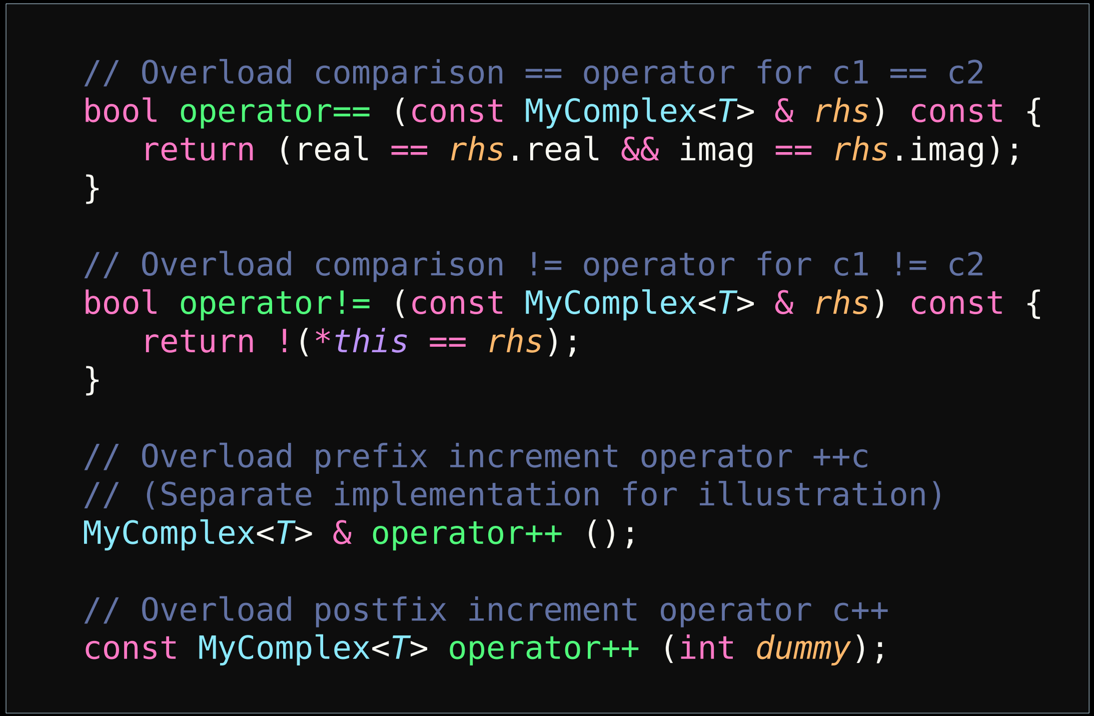
Demo - complex.h

```
//complex.h
#ifndef COMPLEX_H
#define COMPLEX_H

#include <iostream>

// Forward declaration
template <typename T> class MyComplex;

template <typename T>
std::ostream & operator<< (std::ostream & out, const M
yComplex<T> & c);
template <typename T>
std::istream & operator>> (std::istream & in, MyComple
x<T> & c);
```

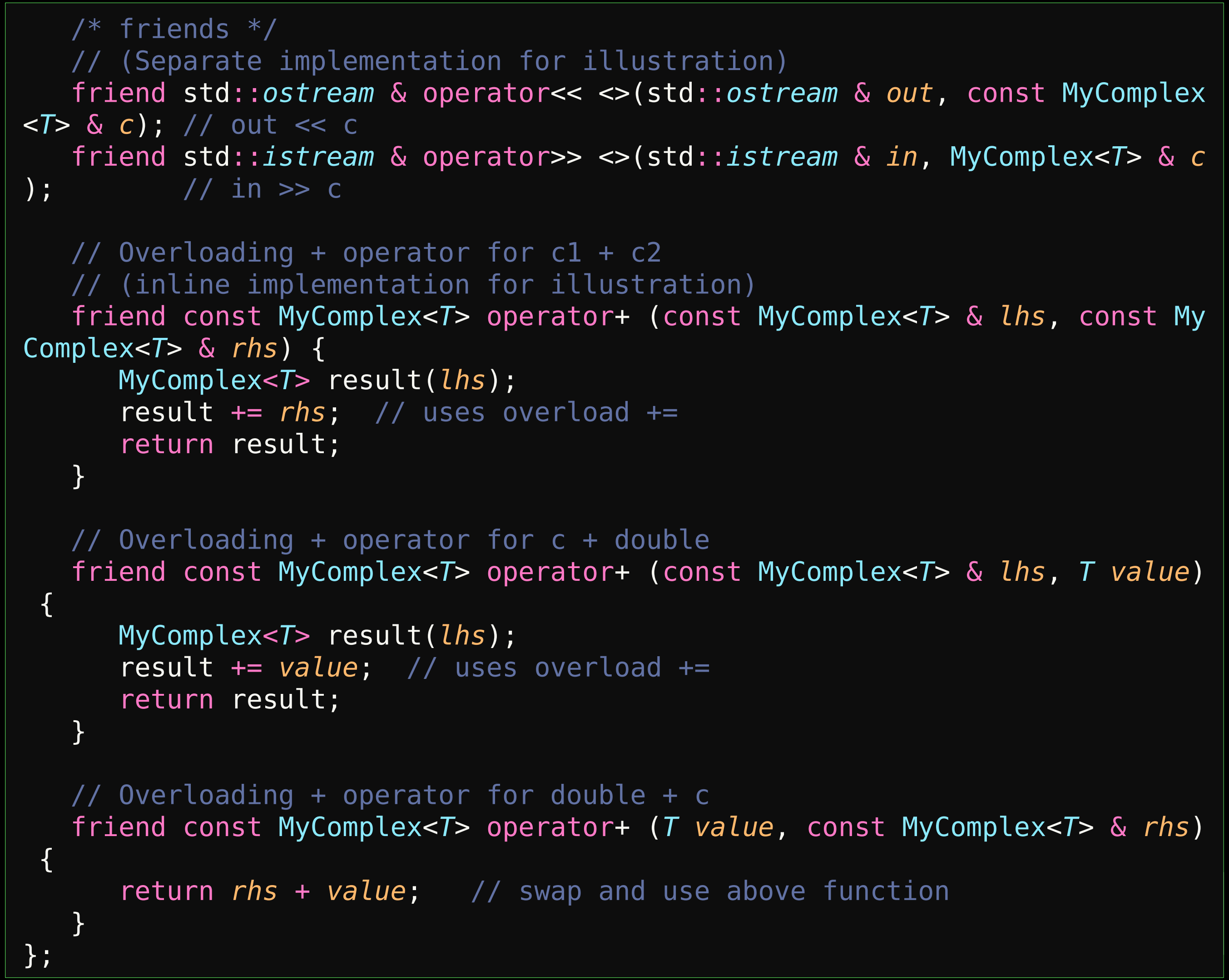
```
// MyComplex template class declaration
template <typename T>
class MyComplex {
private:
   T real, imag;

public:
   // Constructor
   explicit MyComplex<T> (T real = 0, T imag = 0)
         : real(real), imag(imag) { }

   // Overload += operator for c1 += c2
   MyComplex<T> & operator+= (const MyComplex<T> & rhs) {
      real += rhs.real;
      imag += rhs.imag;
      return *this;
   }

   // Overload += operator for c1 += value
   MyComplex<T> & operator+= (T value) {
      real += value;
      return *this;
   }
```

https://www3.ntu.edu.sg/home/ehchua/
programming/cpp/

Demo - complex.h



```
// Overload comparison == operator for c1 == c2
bool operator== (const MyComplex<T> & rhs) const {
   return (real == rhs.real && imag == rhs.imag);
}

// Overload comparison != operator for c1 != c2
bool operator!= (const MyComplex<T> & rhs) const {
   return !(*this == rhs);
}

// Overload prefix increment operator ++c
// (Separate implementation for illustration)
MyComplex<T> & operator++ ();

// Overload postfix increment operator c++
const MyComplex<T> operator++ (int dummy);
```



```
   /* friends */
   // (Separate implementation for illustration)
   friend std::ostream & operator<< <>(std::ostream & out, const MyComplex
<T> & c); // out << c
   friend std::istream & operator>> <>(std::istream & in, MyComplex<T> & c
);        // in >> c

   // Overloading + operator for c1 + c2
   // (inline implementation for illustration)
   friend const MyComplex<T> operator+ (const MyComplex<T> & lhs, const My
Complex<T> & rhs) {
      MyComplex<T> result(lhs);
      result += rhs;  // uses overload +=
      return result;
   }

   // Overloading + operator for c + double
   friend const MyComplex<T> operator+ (const MyComplex<T> & lhs, T value)
 {
      MyComplex<T> result(lhs);
      result += value;  // uses overload +=
      return result;
   }

   // Overloading + operator for double + c
   friend const MyComplex<T> operator+ (T value, const MyComplex<T> & rhs)
 {
      return rhs + value;   // swap and use above function
   }
};
```

Demo - complex.h

```
// Overload prefix increment operator ++c
template <typename T>
MyComplex<T> & MyComplex<T>::operator++ () {
  ++real;   // increment real part only
  return *this;
}

// Overload postfix increment operator c++
template <typename T>
const MyComplex<T> MyComplex<T>::operator+
+ (int dummy) {
   MyComplex<T> saved(*this);
   ++real;  // increment real part only
   return saved;
}
```

```
/* Definition of friend functions */
// Overload stream insertion operator out << c (friend)
template <typename T>
std::ostream & operator<< (std::ostream & out, const MyComplex<T> &
c) {
   out << '(' << c.real << ',' << c.imag << ')';
   return out;
}
// Overload stream extraction operator in >> c (friend)
template <typename T>
std::istream & operator>> (std::istream & in, MyComplex<T> & c) {
   T inReal, inImag;
   char inChar;
   bool validInput = false;
   // Input shall be in the format "(real,imag)"
   in >> inChar;
   if (inChar == '(') {
      in >> inReal >> inChar;
      if (inChar == ',') {
         in >> inImag >> inChar;
         if (inChar == ')') {
            c = MyComplex<T>(inReal, inImag);
            validInput = true;
         }
      }
   }
   if (!validInput) in.setstate(std::ios_base::failbit);
   return in;
}
#endif
```

Demo - testComplex.cpp

```cpp
//testComplex.cpp
#include <iostream>
#include <iomanip>
#include "complex.h"

int main() {
   std::cout << std::fixed << std::setprecision(2);

   MyComplex<double> c1(3.1, 4.2);
   std::cout << c1 << std::endl;  // (3.10,4.20)
   MyComplex<double> c2(3.1);
   std::cout << c2 << std::endl;  // (3.10,0.00)

   MyComplex<double> c3 = c1 + c2;
   std::cout << c3 << std::endl;  // (6.20,4.20)
   c3 = c1 + 2.1;
   std::cout << c3 << std::endl;  // (5.20,4.20)
   c3 = 2.2 + c1;
   std::cout << c3 << std::endl;  // (5.30,4.20)
```

```cpp
 c3 += c1;
   std::cout << c3 << std::endl;  // (8.40,8.40)
   c3 += 2.3;
   std::cout << c3 << std::endl;  // (10.70,8.40)

   std::cout << ++c3 << std::endl; // (11.70,8.40)
   std::cout << c3++ << std::endl; // (11.70,8.40)
   std::cout << c3   << std::endl; // (12.70,8.40)

// c1+c2 = c3;  // error: c1+c2 returns a const
// c1++++;      // error: c1++ returns a const

// MyComplex<int> c4 = 5;  // error: implicit conversion disabled
   MyComplex<int> c4 = (MyComplex<int>)5;  // explicit type casting
allowed
   std::cout << c4 << std::endl; // (5,0)

   MyComplex<int> c5;
   std::cout << "Enter a complex number in (real,imag): ";
   std::cin >> c5;
   if (std::cin.good()) {
      std::cout << c5 << std::endl;
   } else {
      std::cerr << "Invalid input" << std::endl;
   }
   return 0;
}
```

References

Conditional operators:

- http://www.cplusplus.com/doc/tutorial/operators/

Function template

- http://www.cplusplus.com/doc/tutorial/operators/

Class template

- https://www.cplusplus.com/doc/tutorial/templates/