COEN-244 Tutorial #10

March 30th, 2023

Generic Programming

Generic Programming: a programming style about generalizing software components so that they can be easily reused in a wide variety of situations.

- In generalized programming, we usually tend to avoid any hard-coding, hence we also want to avoid having multiple software components for multiple datatypes.

Generics: the idea to allow a type (int, string, … etc and user-defined types) to be a parameter to functions, methods, classes, and interfaces.

- Generics can be implemented in **`C++`** using **`Templates`**.

Advantages of Generic Programming:

1. Code Reusability
2. Avoid Function Overloading
3. Once written it can be used for multiple times and cases.

Templates

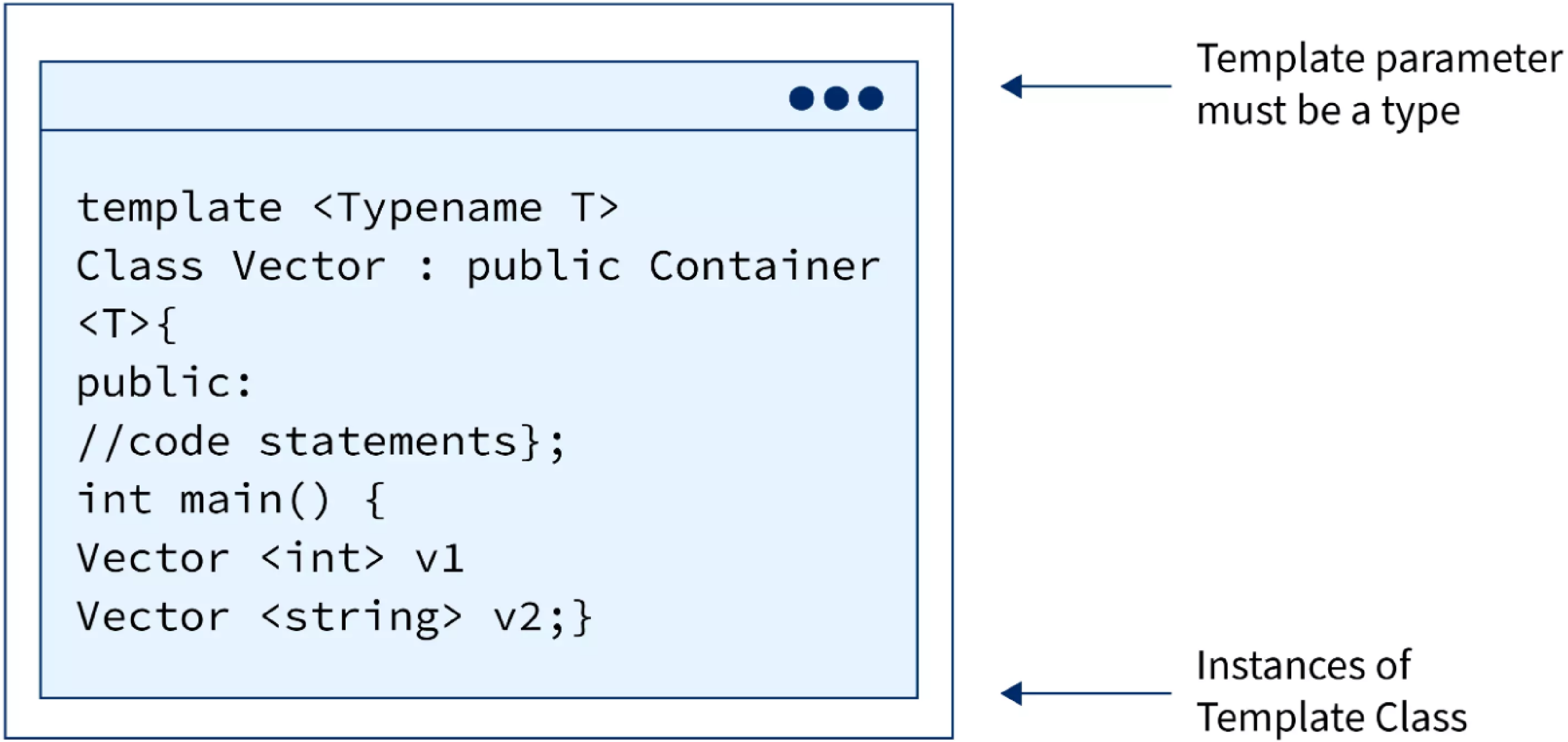
Templates: a powerful tool used to define generic functions and classes in C++.

- Templates work by passing the datatype as a **parameter**.
- Multiple type parameters can be passed.
- Keywords **class** and **typename** are used to specify that the passed parameter is of type **datatype** or **user-defined** type.
- **Note: 'typename'** can always be replaced by **'class'**.

Examples of templates in C++:

- `vector <int> vec;`
- `vector <char> vec;`
- `stack <string> s;`
- `queue <int> q; etc.`

Templates Blueprint



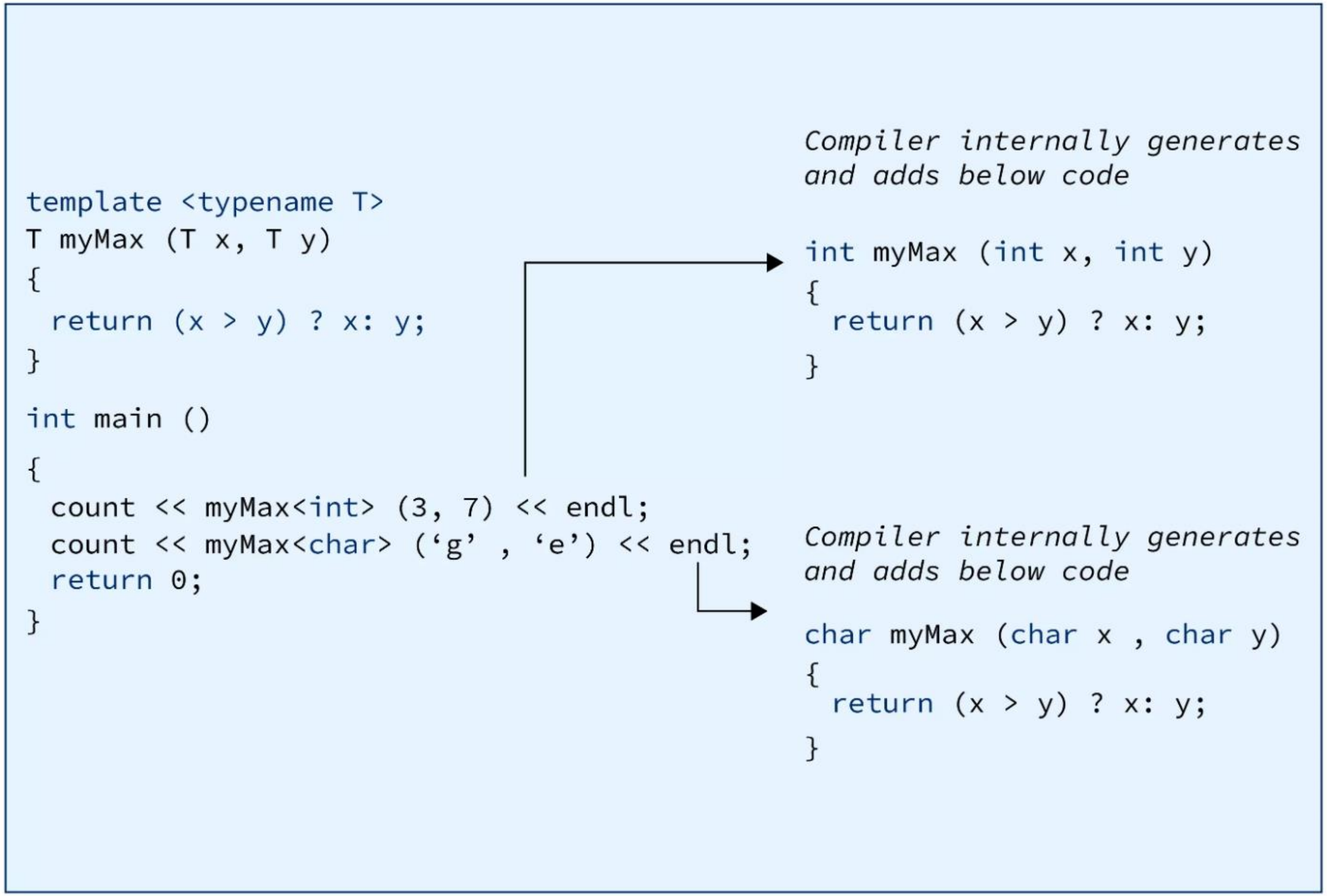
```
template <Typename T>
Class Vector : public Container
<T>{
public:
//code statements};
int main() {
Vector <int> v1
Vector <string> v2;}
```

Template parameter must be a type

Instances of Template Class

Source: https://www.scaler.com/topics/cpp/templates-in-cpp/

Templates Implementation



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

int main ()

{
  count << myMax<int> (3, 7) << endl;
  count << myMax<char> (‘g’ , ‘e’) << endl;
  return 0;
}
```

Compiler internally generates and adds below code

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

Compiler internally generates and adds below code

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

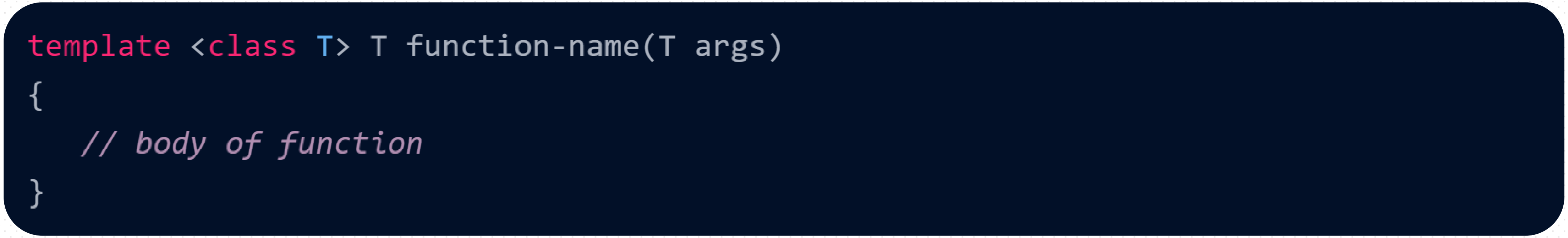
Source: https://www.scaler.com/topics/cpp/templates-in-cpp/

Generic Functions | Function Templates

Normal C++ functions can only work with one datatype, but **template functions** can work on multiple datatypes.

- **Alt. Solution (not recommended):** overloading standard functions
- **Template** functions can be overloaded too

Syntax:



```
template <class T> T function-name(T args)
{
   // body of function
}
```

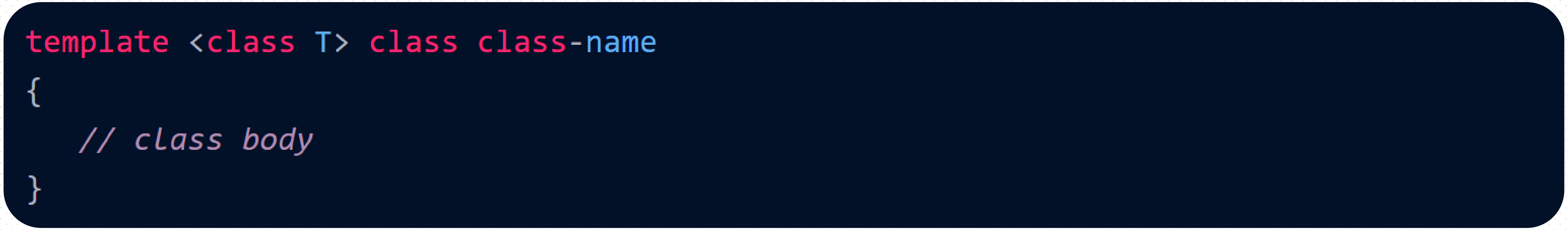
- **template** is the keyword to specify the beginning of generic software components
- **T** is the type of argument or placeholder that can accept various data types.
- **class** is a keyword used to specify a generic type in a template declaration. As we have seen earlier, we can always write **typename** in the place of class.

Generic Classes | Class Templates

Just like function templates, we can extend the concept to user-defined classes to define generic classes.

- This is known as **class templates.**
- **NOTE: Generic Classes** and **Abstract Classes** are not the same.
- A **static variable** in template classes remains shared among all objects of the **same type**.

Syntax:



```
template <class T> class class-name
{
   // class body
}
```

- **template** is the keyword to specify the beginning of a generic class
- **T** is the type of argument or placeholder that can accept various data types.
- **class** or **typename** is a keyword used to specify a generic type in a template declaration.

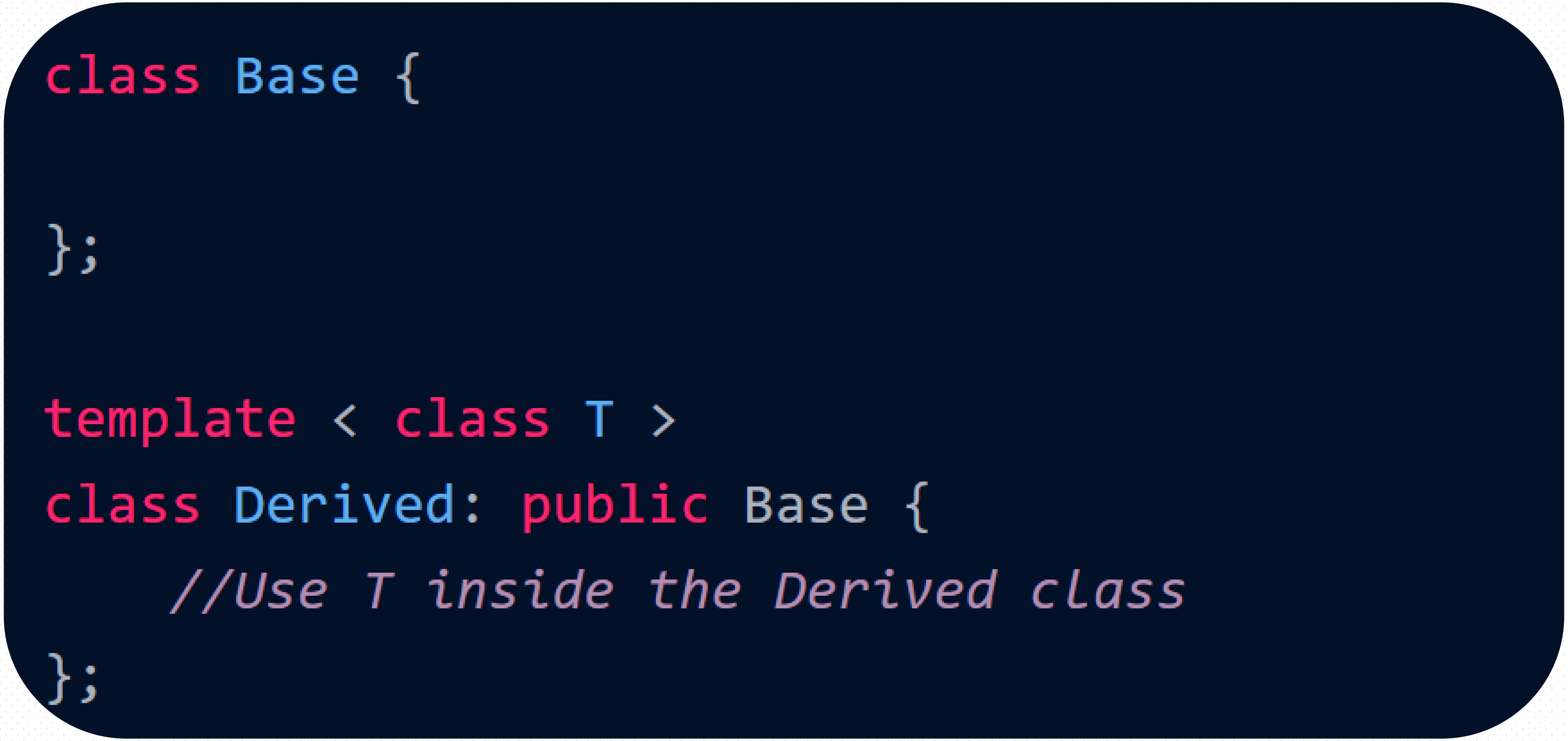
Class Templates and Inheritance

Concepts of inheritance work in a similar way for class templates depending on the scenario:

1. Base Class is not a Template class, but a Derived class is a Template class.
2. Base Class is a Template class, but Derived class is not a Template class.
3. Base Class is a Template class, and the Derived class is also a Template class.
4. Base Class is a Template Class, and derived class is a Template class with different Types.

These combinations allow for great utilization of templates in C++.

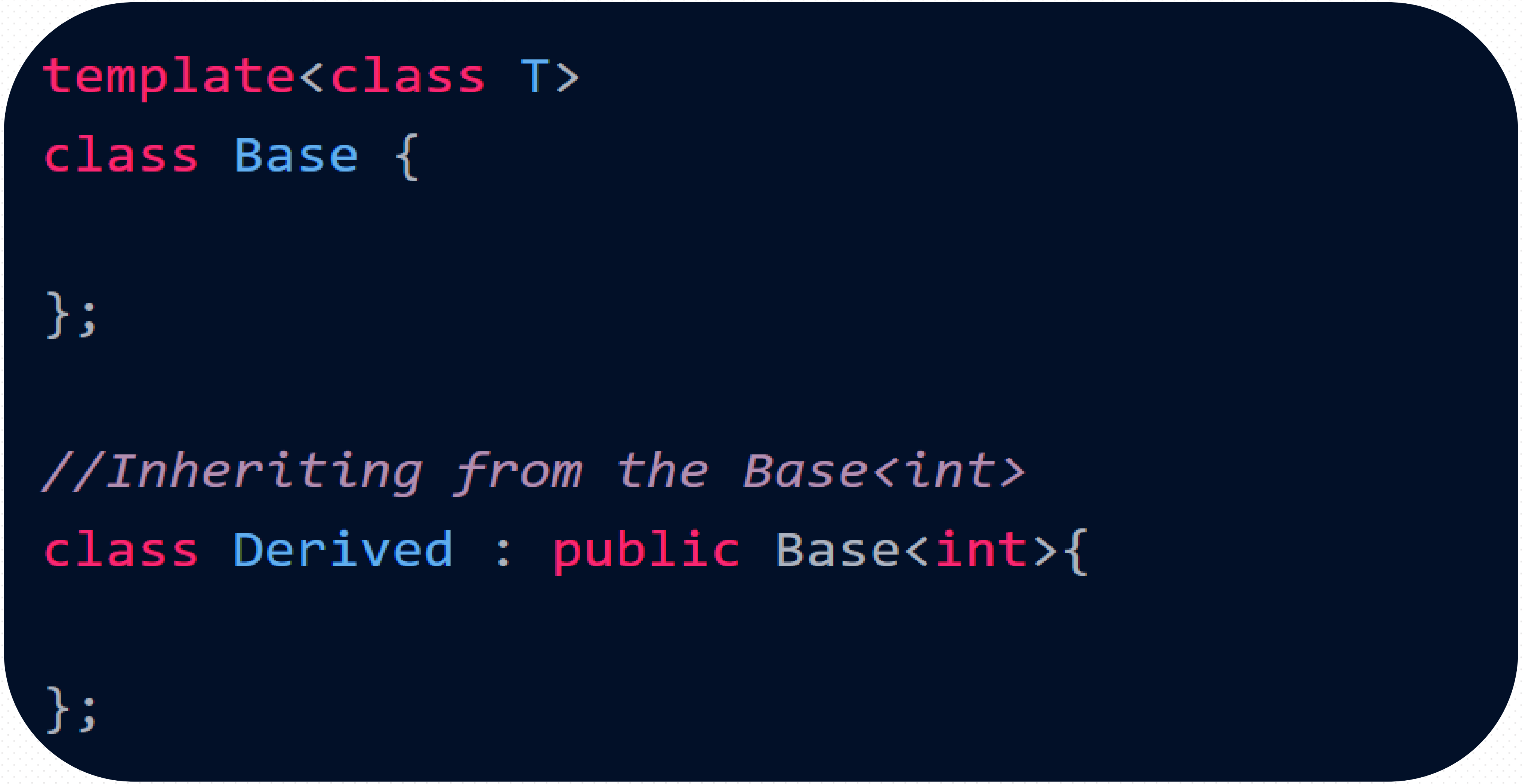
Case 1: Normal base class, template derived class



```
class Base {

};

template < class T >
class Derived: public Base {
    //Use T inside the Derived class
};
```

Source: https://www.scaler.com/topics/cpp/generic-programming-in-cpp/

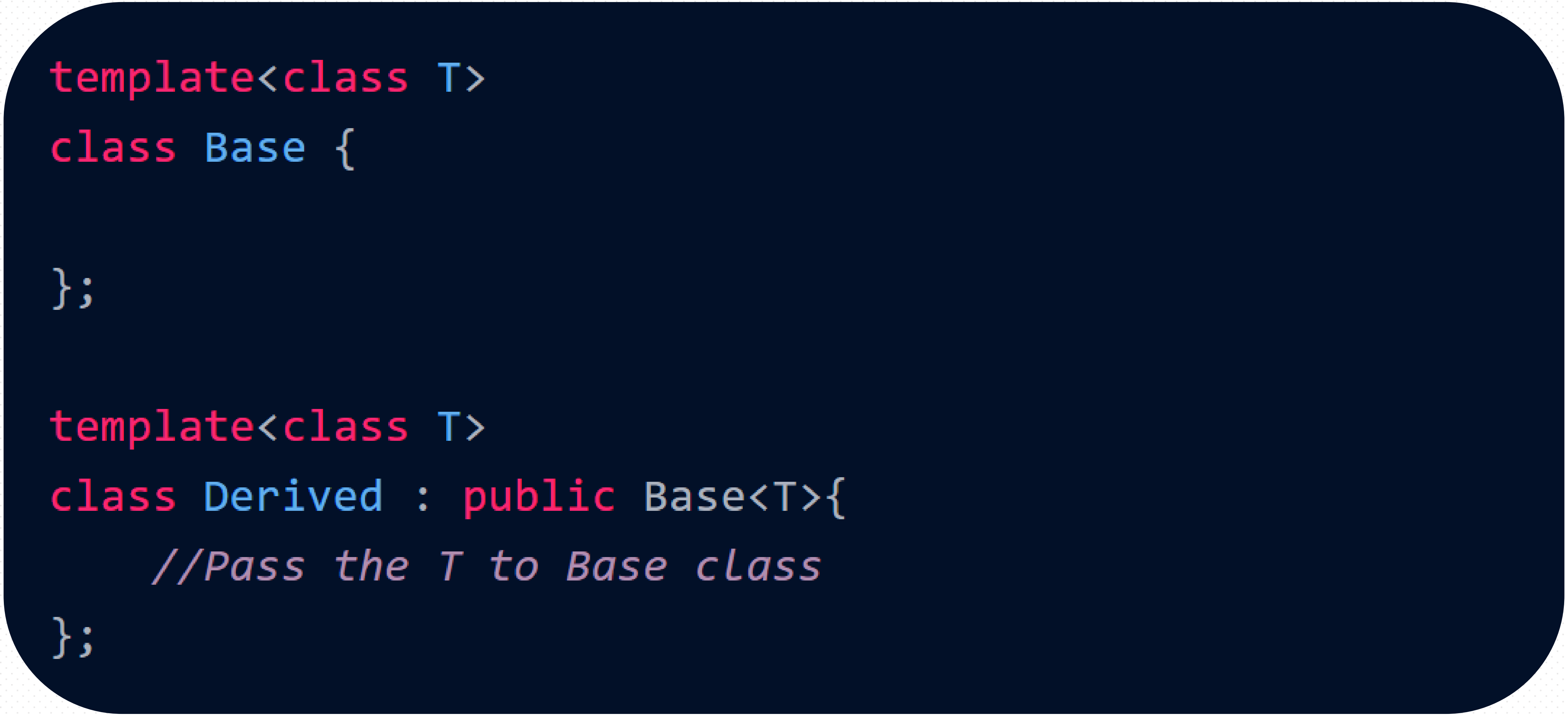
Case 2: Base template class, Normal derived class



```
template<class T>
class Base {


};


//Inheriting from the Base<int>
class Derived : public Base<int>{


};
```

Source: https://www.scaler.com/topics/cpp/generic-programming-in-cpp/

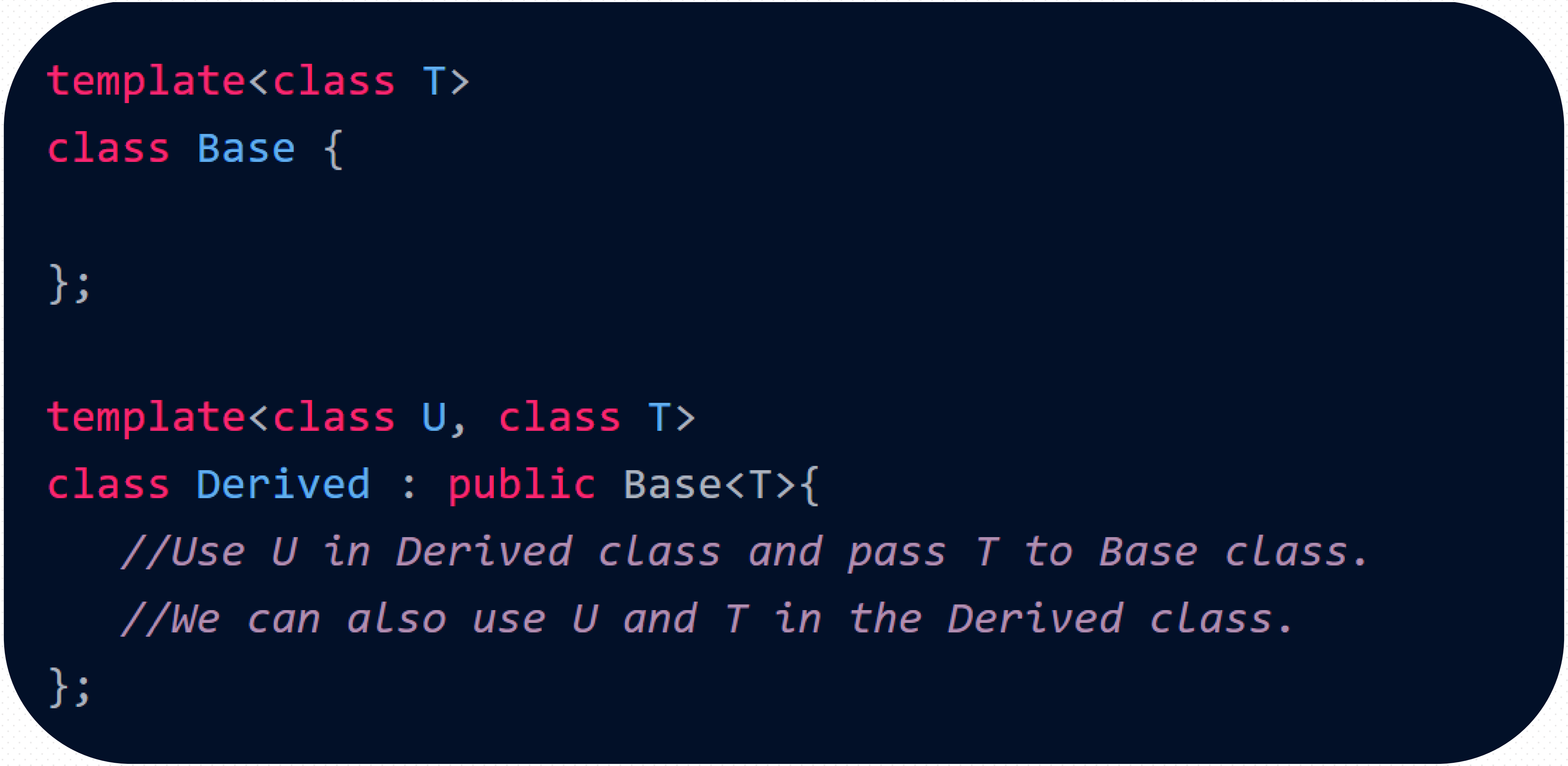
Case 3: Base & derived template classes; same type



```
template<class T>
class Base {


};


template<class T>
class Derived : public Base<T>{
    //Pass the T to Base class
};
```

Source: https://www.scaler.com/topics/cpp/generic-programming-in-cpp/

Case 4: Base & derived template classes; different types



```
template<class T>
class Base {

};

template<class U, class T>
class Derived : public Base<T>{
   //Use U in Derived class and pass T to Base class.
   //We can also use U and T in the Derived class.
};
```

Source: https://www.scaler.com/topics/cpp/generic-programming-in-cpp/

THANK YOU