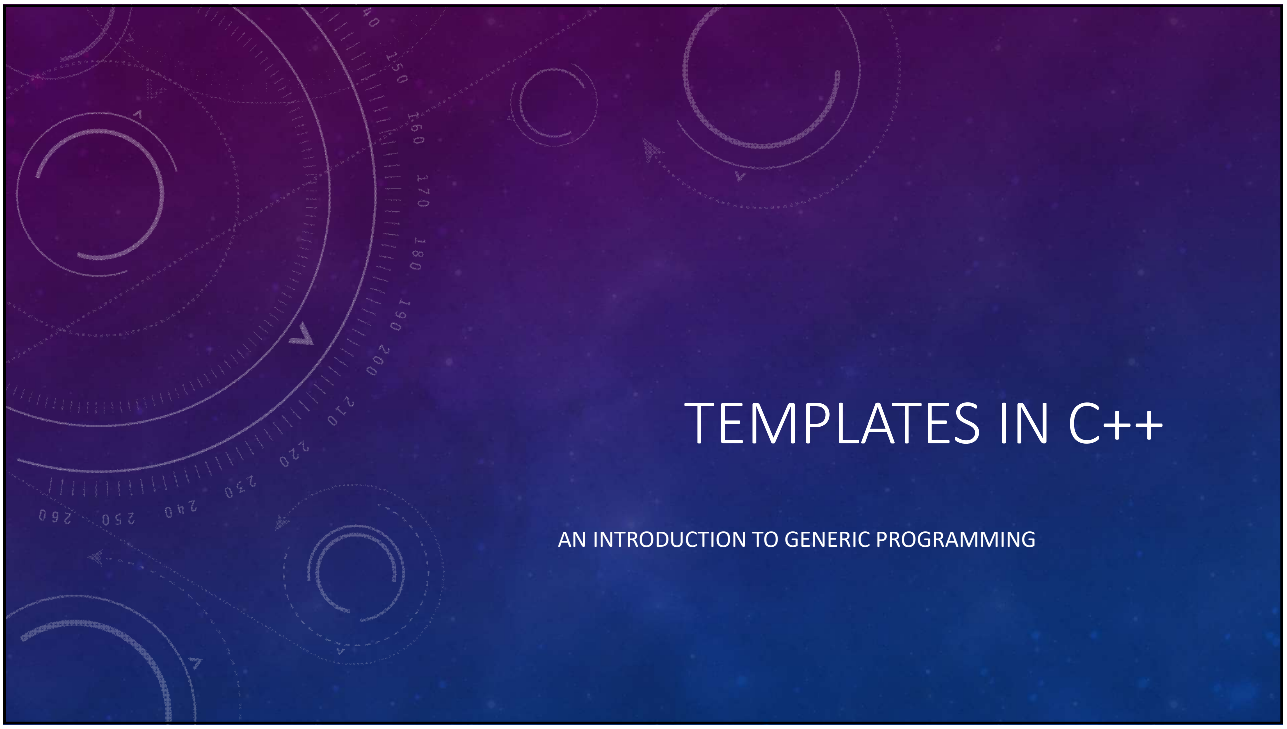
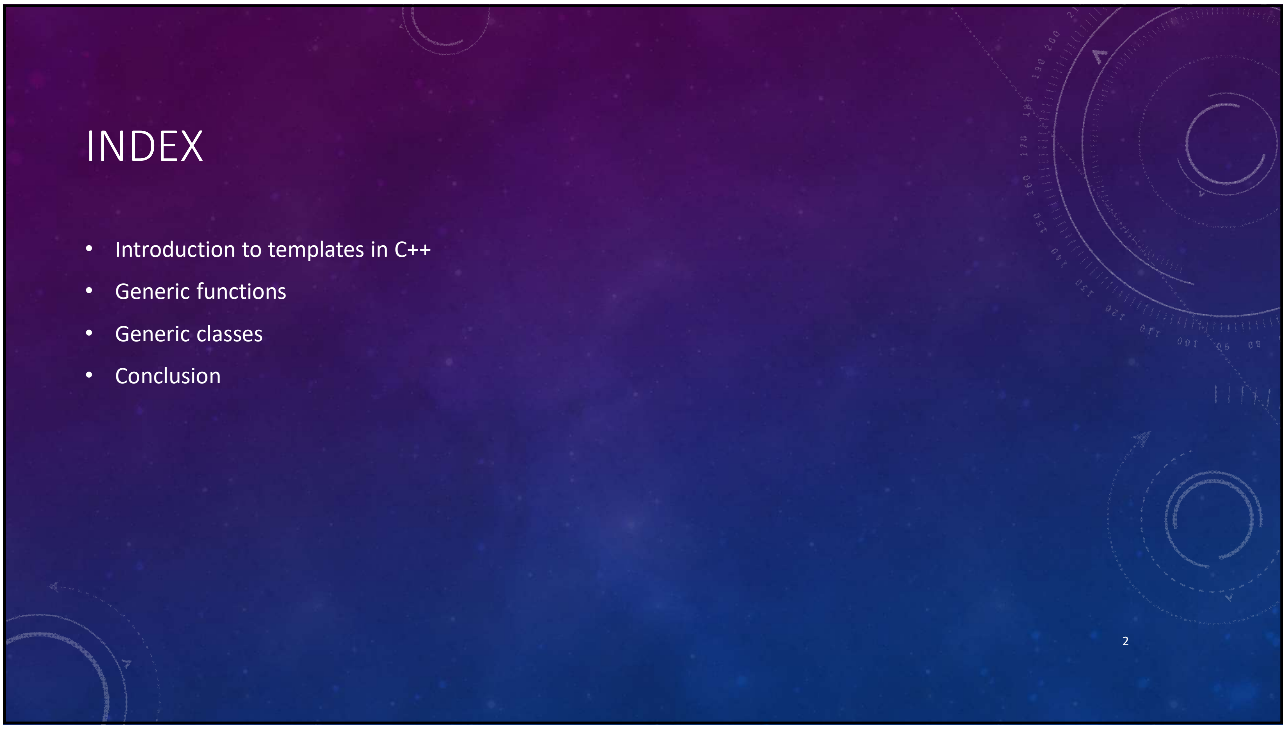


TEMPLATES IN C++

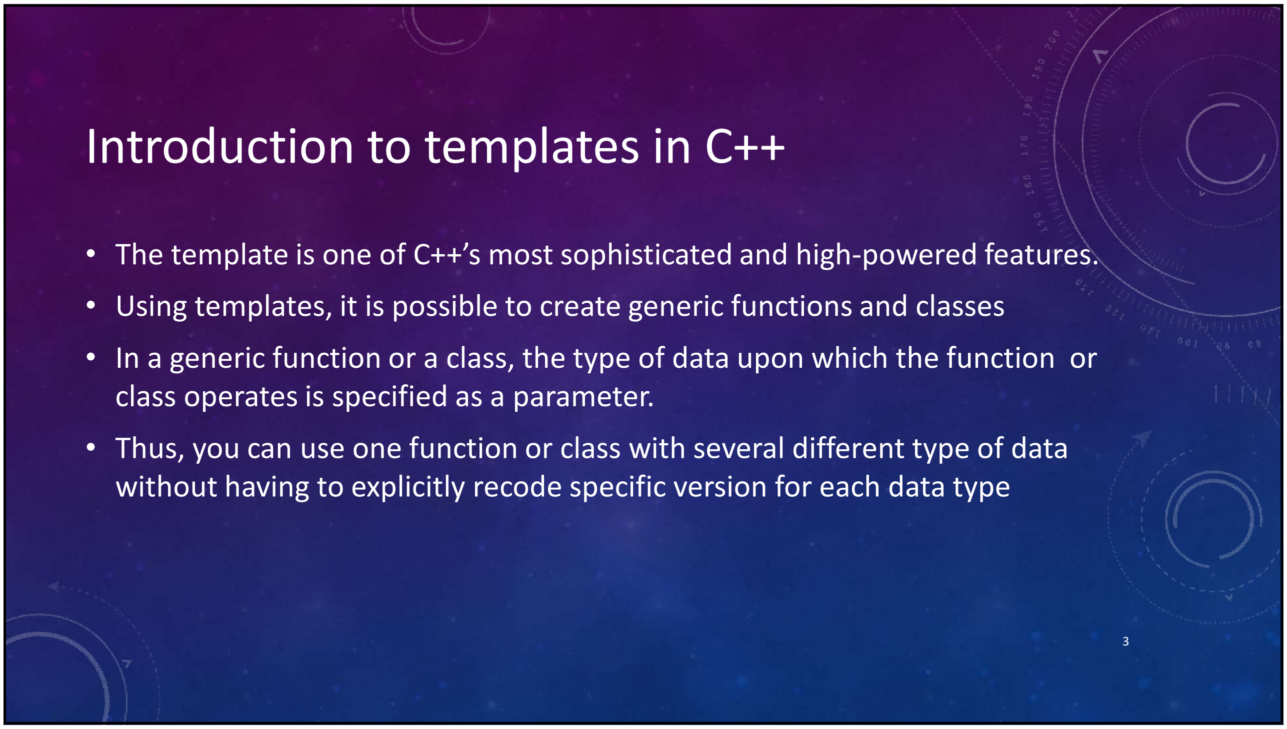
AN INTRODUCTION TO GENERIC PROGRAMMING



INDEX

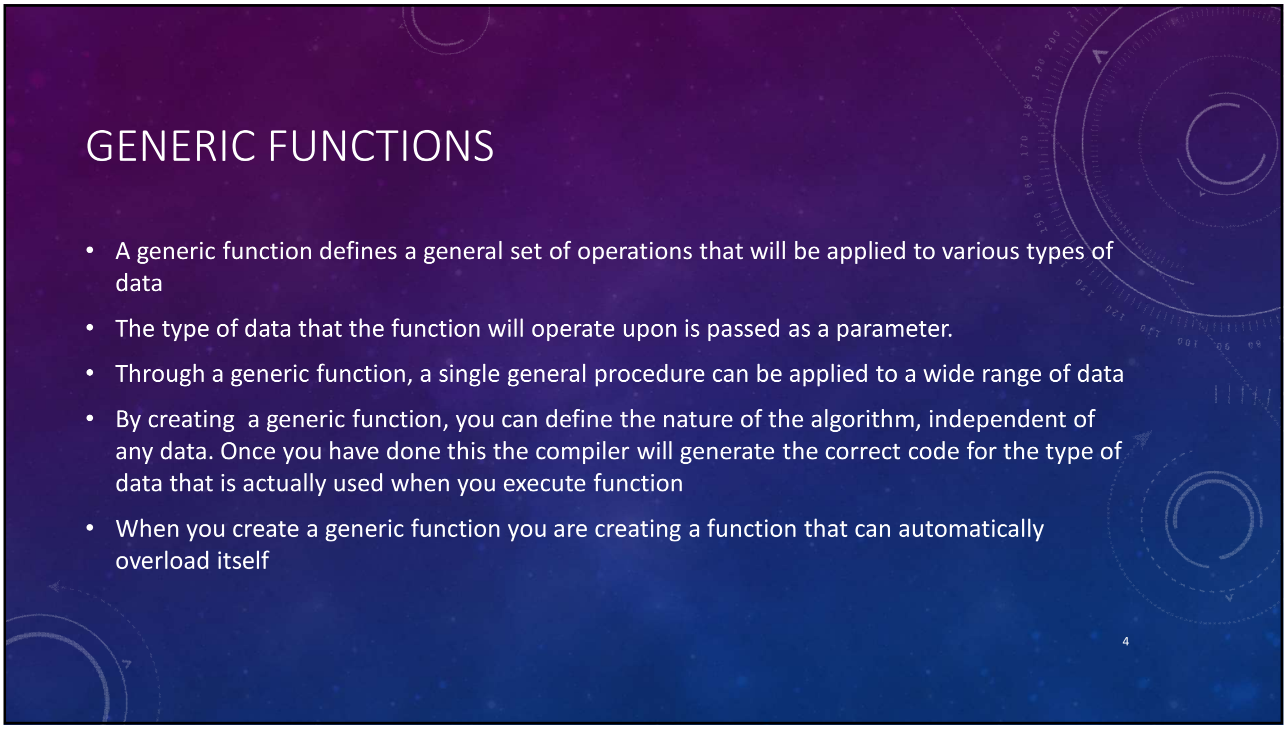
- Introduction to templates in C++
- Generic functions
- Generic classes
- Conclusion

2



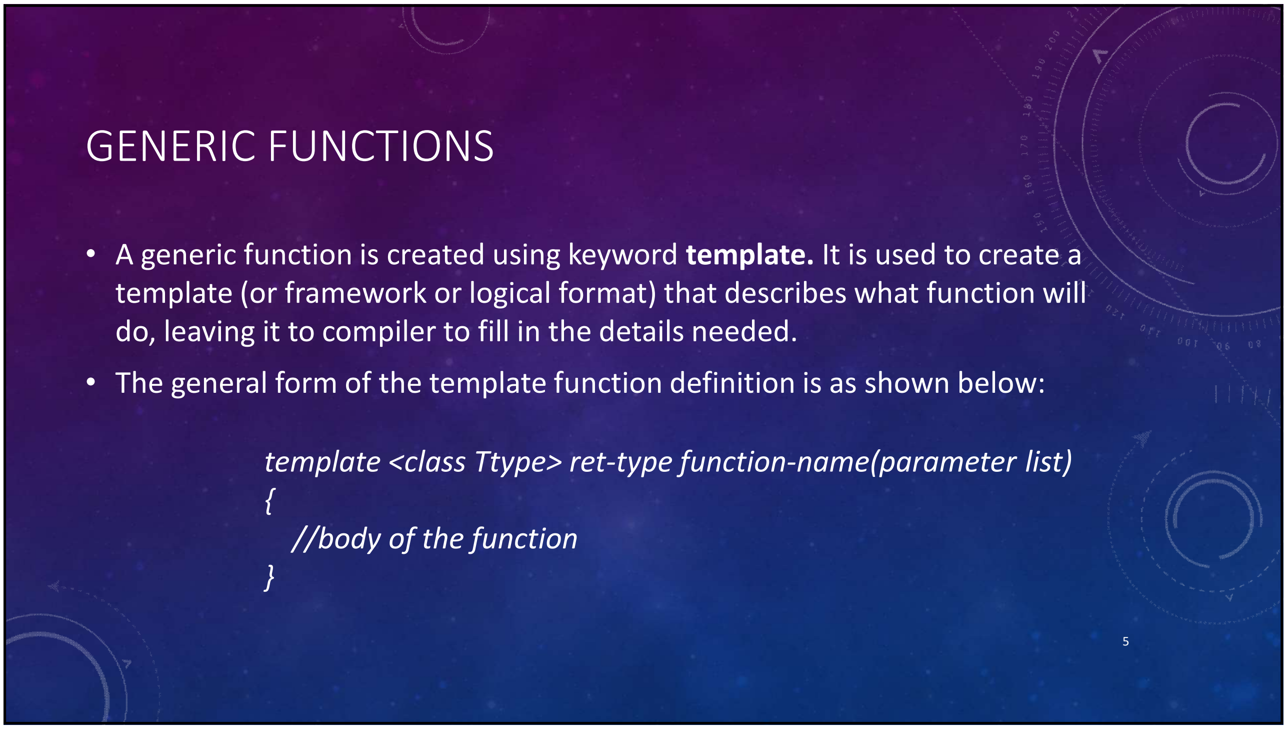
Introduction to templates in C++

- The template is one of C++’s most sophisticated and high-powered features.
- Using templates, it is possible to create generic functions and classes
- In a generic function or a class, the type of data upon which the function or class operates is specified as a parameter.
- Thus, you can use one function or class with several different type of data without having to explicitly recode specific version for each data type



GENERIC FUNCTIONS

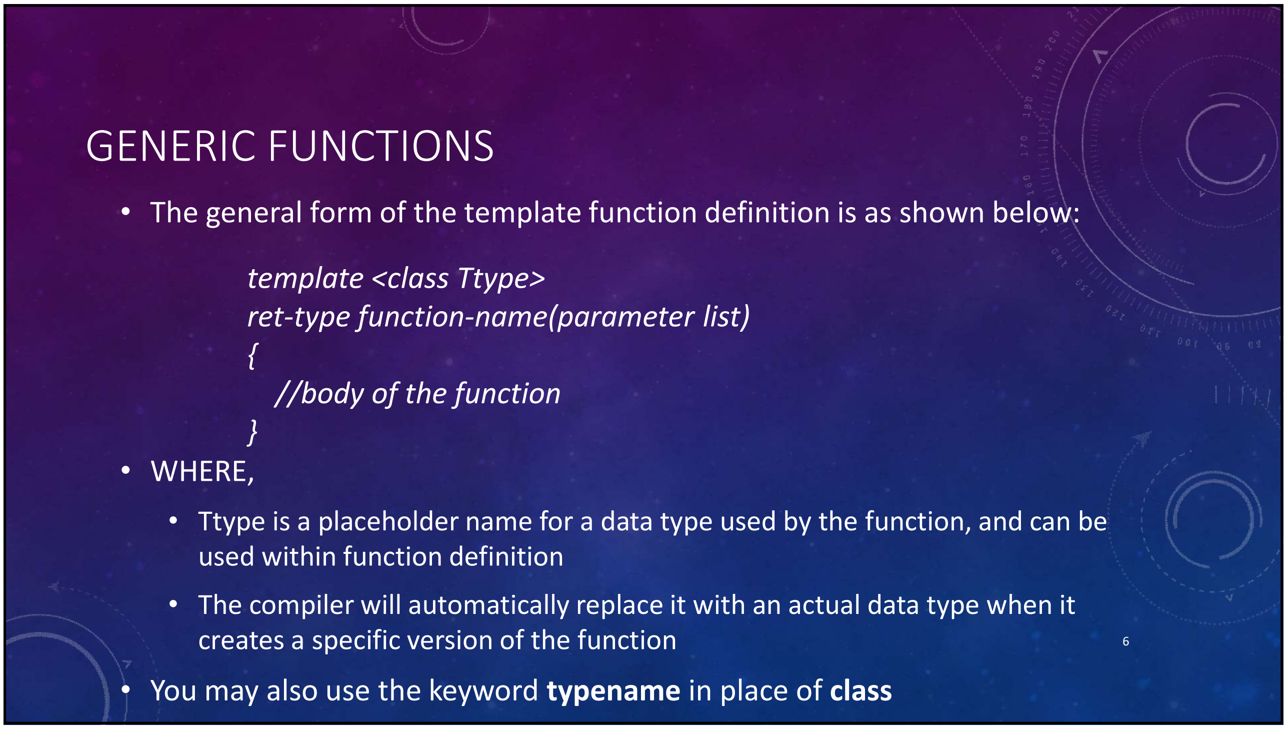
- A generic function defines a general set of operations that will be applied to various types of data
- The type of data that the function will operate upon is passed as a parameter.
- Through a generic function, a single general procedure can be applied to a wide range of data
- By creating a generic function, you can define the nature of the algorithm, independent of any data. Once you have done this the compiler will generate the correct code for the type of data that is actually used when you execute function
- When you create a generic function you are creating a function that can automatically overload itself



GENERIC FUNCTIONS

- A generic function is created using keyword **template.** It is used to create a template (or framework or logical format) that describes what function will do, leaving it to compiler to fill in the details needed.
- The general form of the template function definition is as shown below:

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

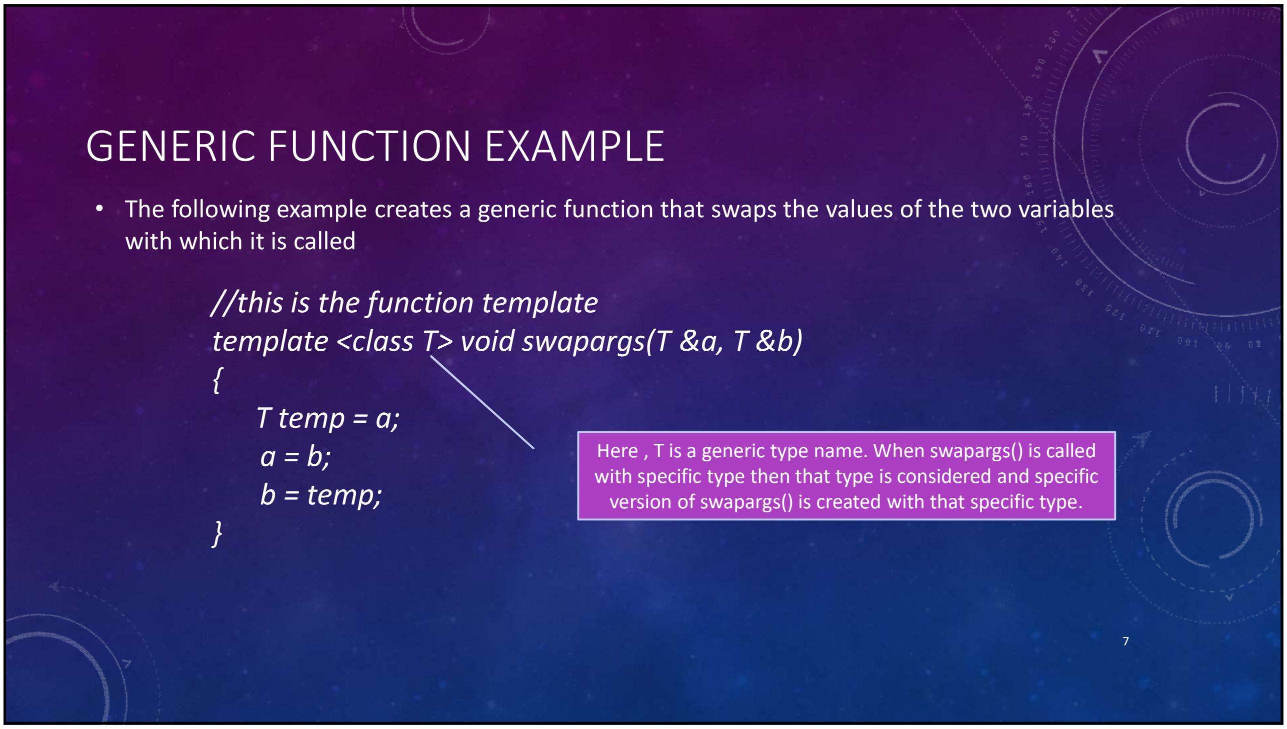


GENERIC FUNCTIONS

- The general form of the template function definition is as shown below:

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

- WHERE,
  - Ttype is a placeholder name for a data type used by the function, and can be used within function definition
  - The compiler will automatically replace it with an actual data type when it creates a specific version of the function
- You may also use the keyword **typename** in place of **class**

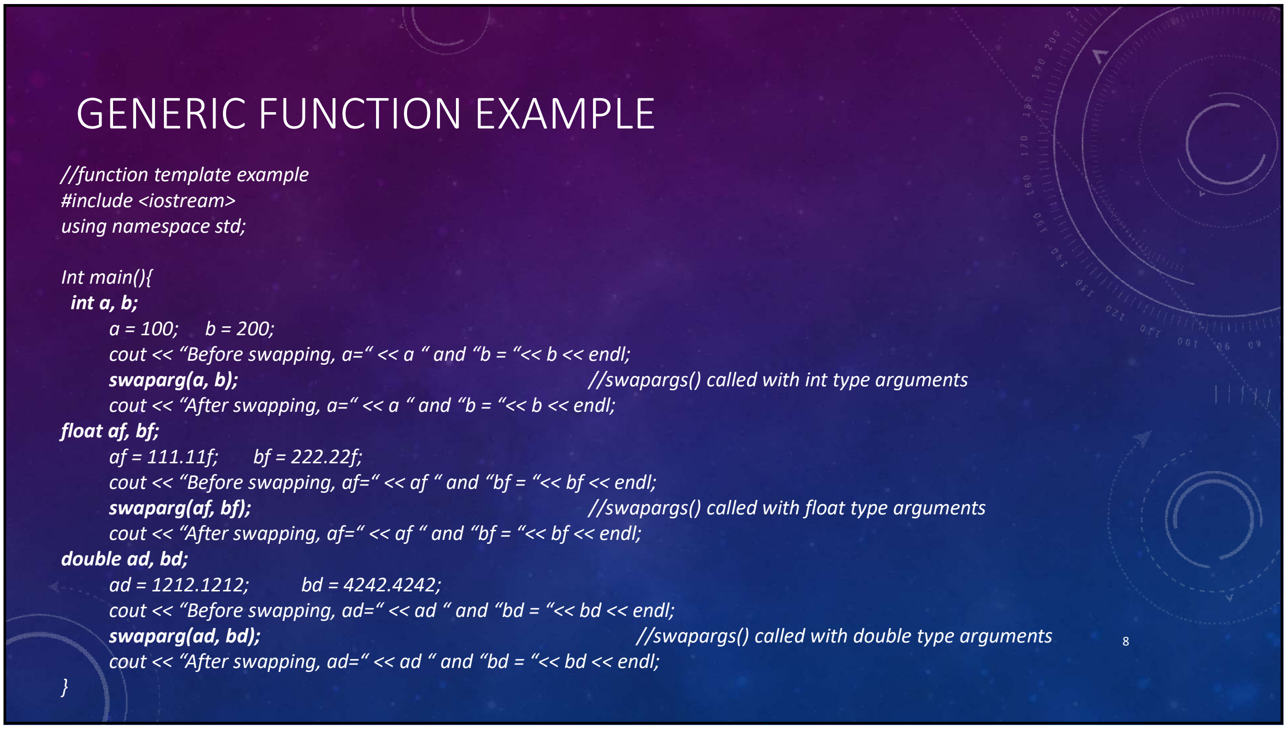


GENERIC FUNCTION EXAMPLE

- The following example creates a generic function that swaps the values of the two variables with which it is called

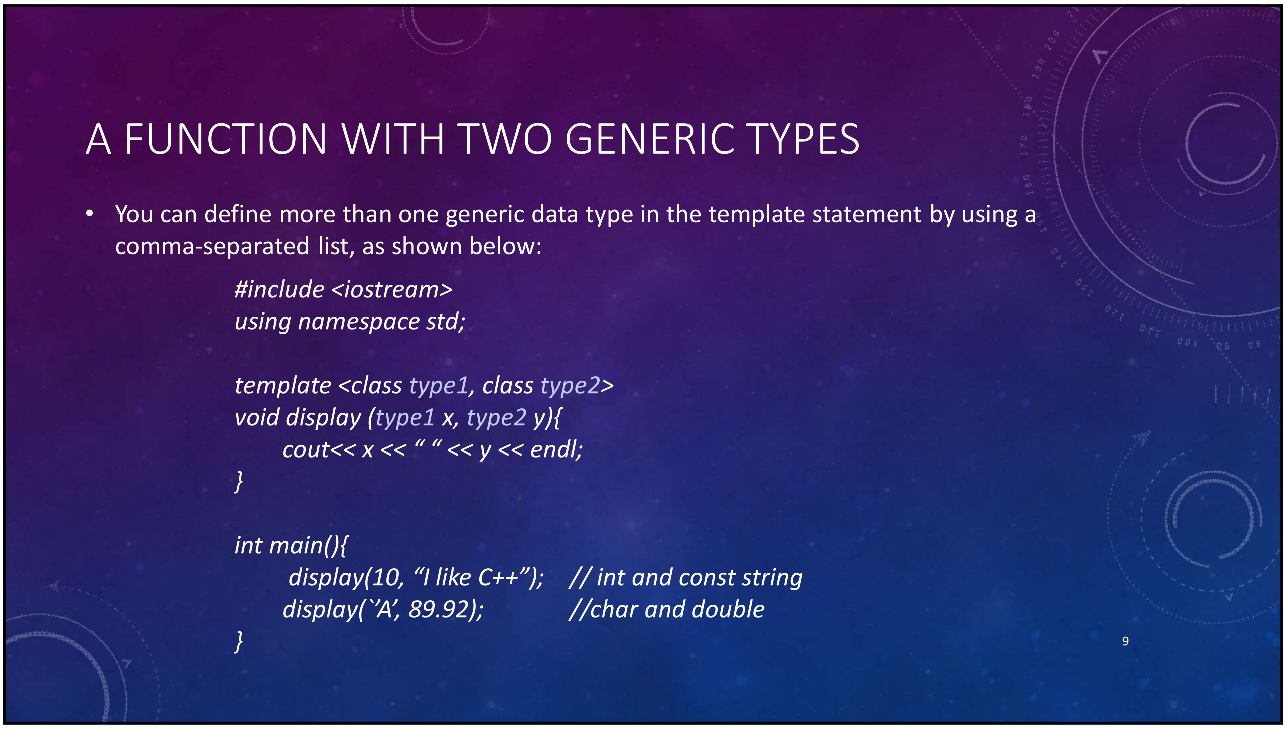
```
//this is the function template
template <class T> void swapargs(T &a, T &b)
{
      T temp = a;
       a = b;
       b = temp;
}
```

Here , T is a generic type name. When swapargs() is called with specific type then that type is considered and specific version of swapargs() is created with that specific type.



GENERIC FUNCTION EXAMPLE

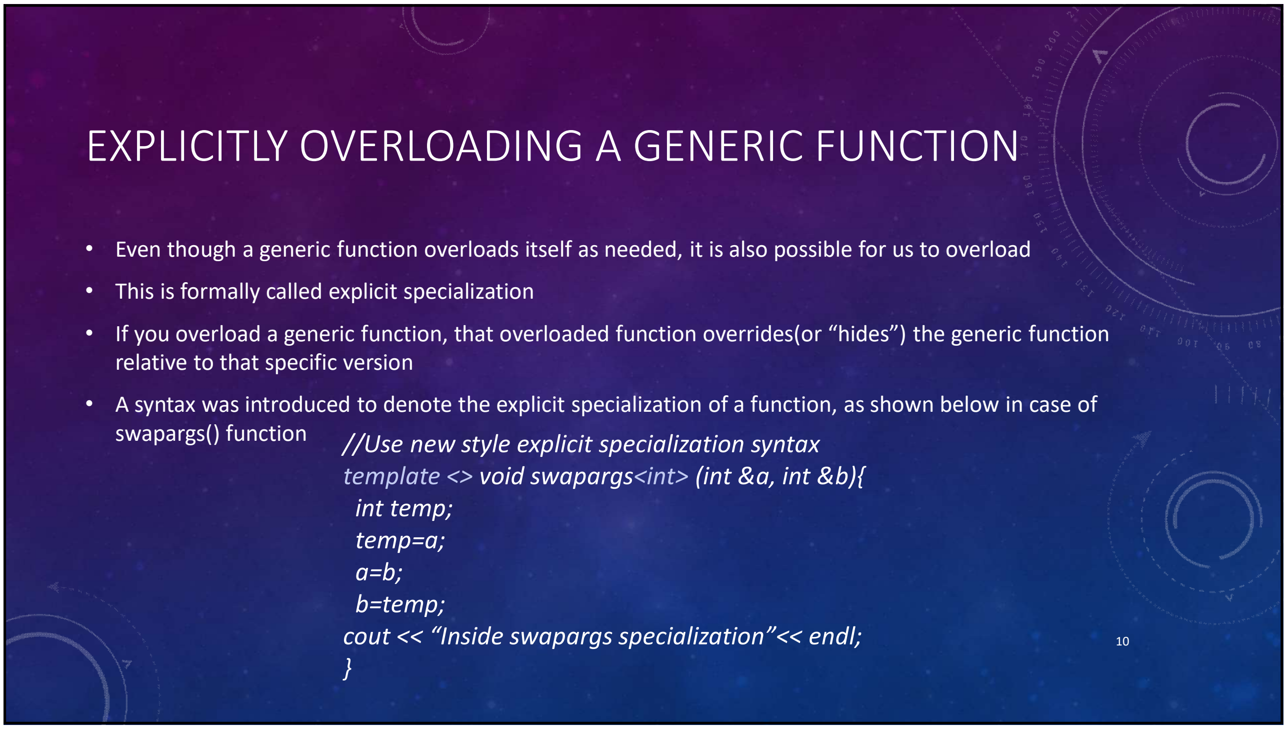
```
//function template example
#include <iostream>
using namespace std;

Int main(){
  int a, b;
        a = 100;     b = 200;
        cout << “Before swapping, a=“ << a “ and “b = “<< b << endl;
        swaparg(a, b);                                        //swapargs() called with int type arguments
        cout << “After swapping, a=“ << a “ and “b = “<< b << endl;
float af, bf;
        af = 111.11f;      bf = 222.22f;
        cout << “Before swapping, af=“ << af “ and “bf = “<< bf << endl;
        swaparg(af, bf);                                      //swapargs() called with float type arguments
        cout << “After swapping, af=“ << af “ and “bf = “<< bf << endl;
double ad, bd;
        ad = 1212.1212;         bd = 4242.4242;
        cout << “Before swapping, ad=“ << ad “ and “bd = “<< bd << endl;
        swaparg(ad, bd);                                             //swapargs() called with double type arguments
        cout << “After swapping, ad=“ << ad “ and “bd = “<< bd << endl;
}
```



A FUNCTION WITH TWO GENERIC TYPES

- You can define more than one generic data type in the template statement by using a comma-separated list, as shown below:

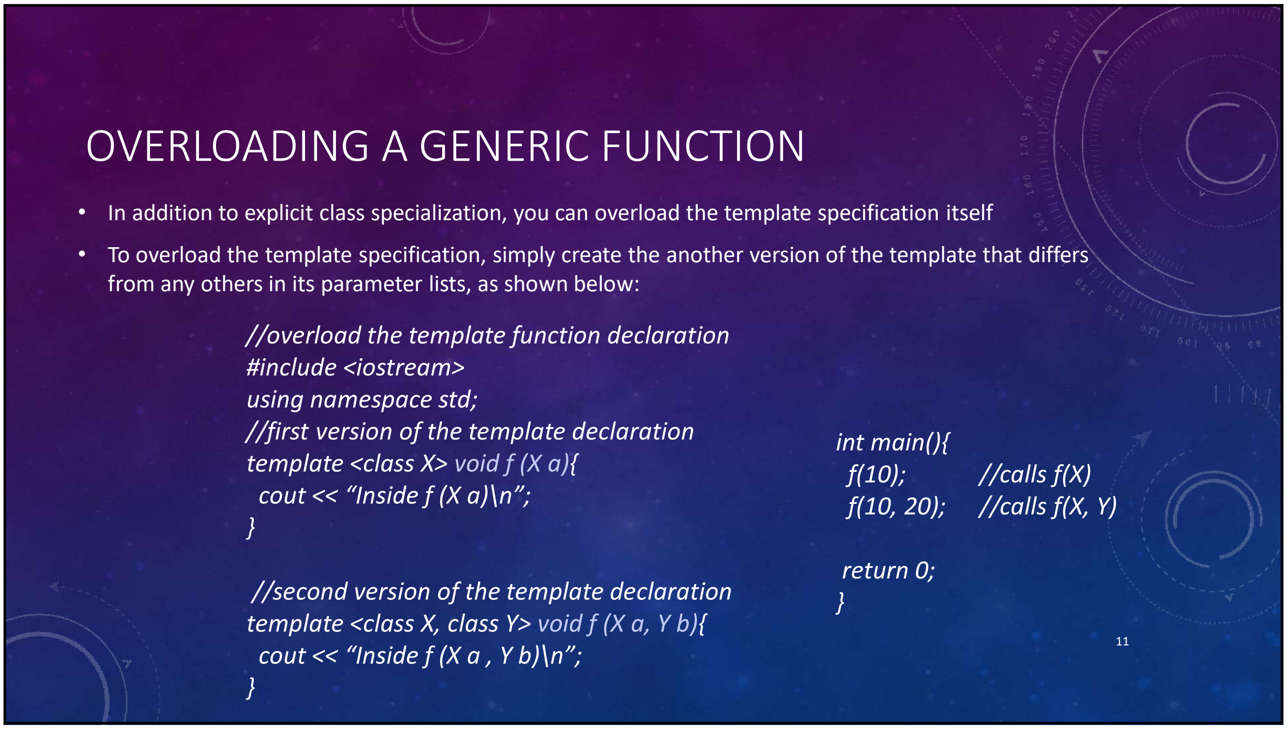
```
#include <iostream>
using namespace std;

template <class type1, class type2>
void display (type1 x, type2 y){
        cout<< x << “ “ << y << endl;
}

int main(){
         display(10, “I like C++”);    // int and const string
        display(`’A’, 89.92);           //char and double
}
```



EXPLICITLY OVERLOADING A GENERIC FUNCTION

- Even though a generic function overloads itself as needed, it is also possible for us to overload
- This is formally called explicit specialization
- If you overload a generic function, that overloaded function overrides(or “hides”) the generic function relative to that specific version
- A syntax was introduced to denote the explicit specialization of a function, as shown below in case of swapargs() function

```
//Use new style explicit specialization syntax
template <> void swapargs<int> (int &a, int &b){
  int temp;
  temp=a;
  a=b;
  b=temp;
cout << “Inside swapargs specialization”<< endl;
}
```

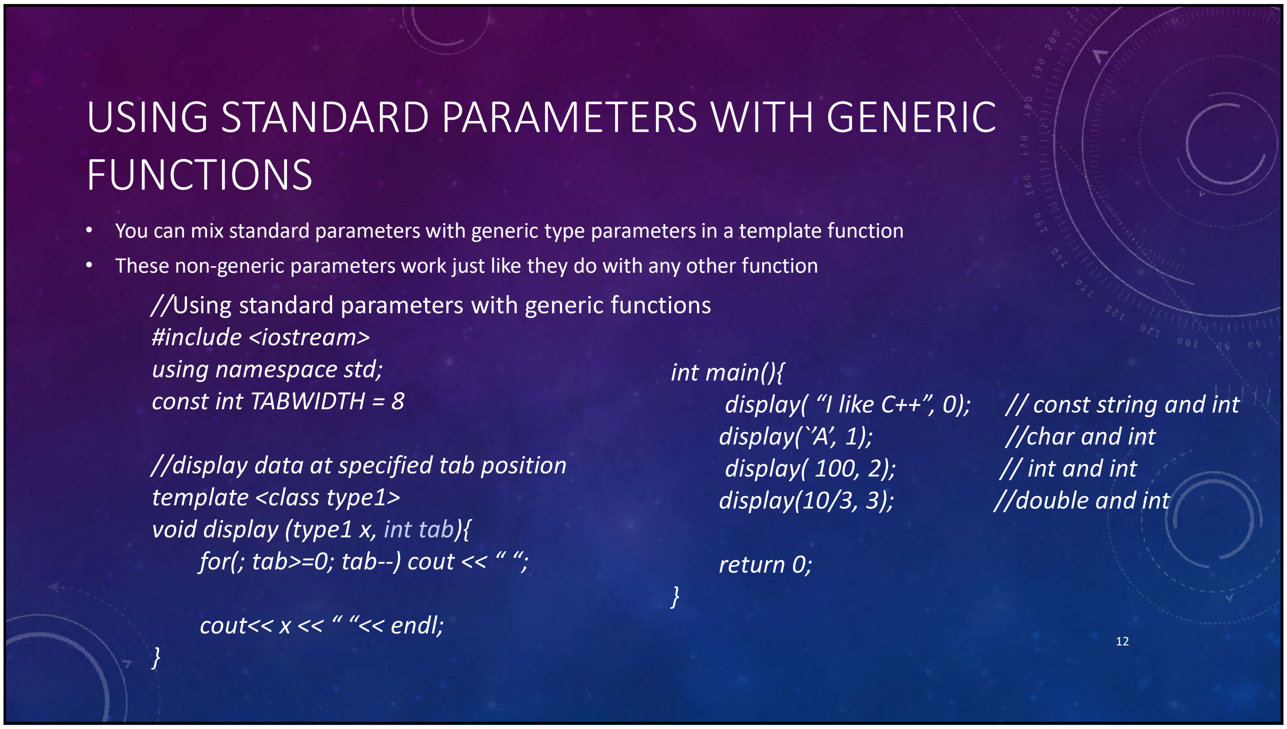


OVERLOADING A GENERIC FUNCTION

- In addition to explicit class specialization, you can overload the template specification itself
- To overload the template specification, simply create the another version of the template that differs from any others in its parameter lists, as shown below:

```
//overload the template function declaration
#include <iostream>
using namespace std;
//first version of the template declaration
template <class X> void f (X a){
  cout << “Inside f (X a)\n”;
}

 //second version of the template declaration
template <class X, class Y> void f (X a, Y b){
  cout << “Inside f (X a , Y b)\n”;
}
```

```
int main(){
 f(10);          //calls f(X)
 f(10, 20);      //calls f(X, Y)

 return 0;
}
```

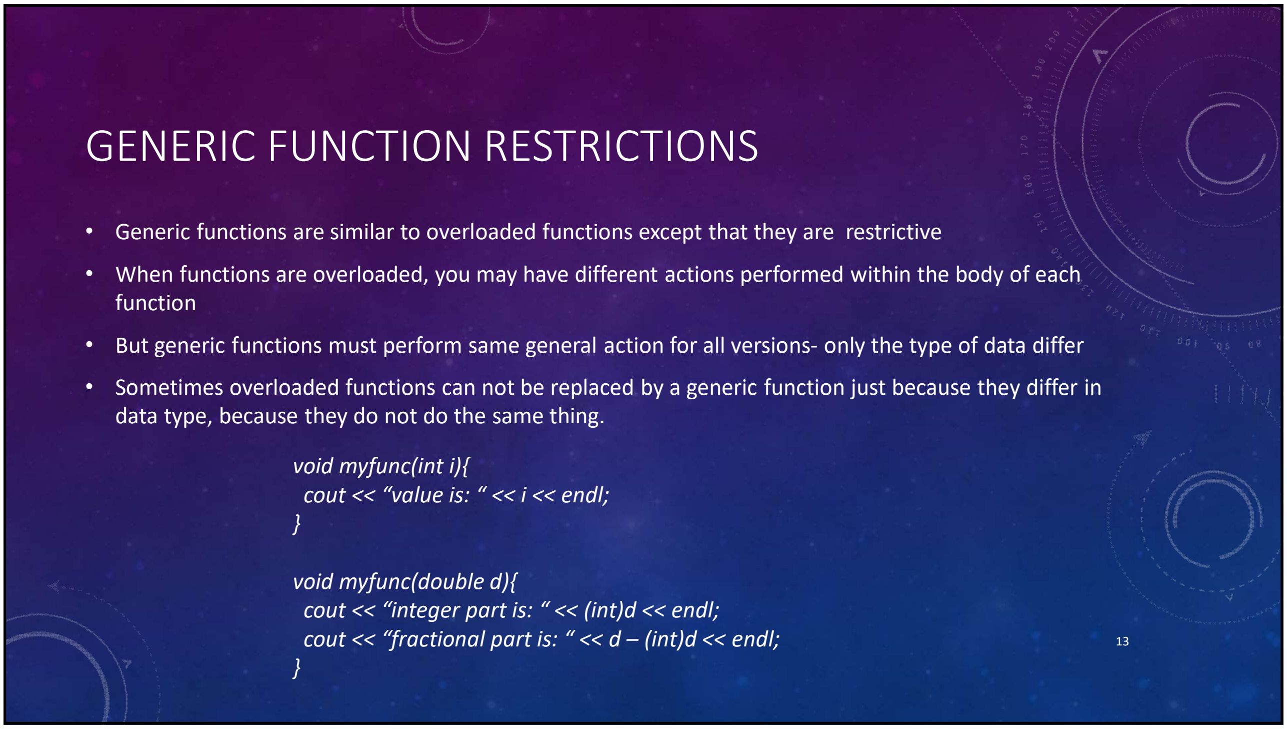


USING STANDARD PARAMETERS WITH GENERIC FUNCTIONS

- You can mix standard parameters with generic type parameters in a template function
- These non-generic parameters work just like they do with any other function

```
//Using standard parameters with generic functions
#include <iostream>
using namespace std;
const int TABWIDTH = 8

//display data at specified tab position
template <class type1>
void display (type1 x, int tab){
        for(; tab>=0; tab--) cout << “ “;

        cout<< x << “ “<< endl;
}
```

```
int main(){
         display( “I like C++”, 0);    // const string and int
        display(`A’, 1);               //char and int
         display( 100, 2);            // int and int
        display(10/3, 3);             //double and int

        return 0;
}
```

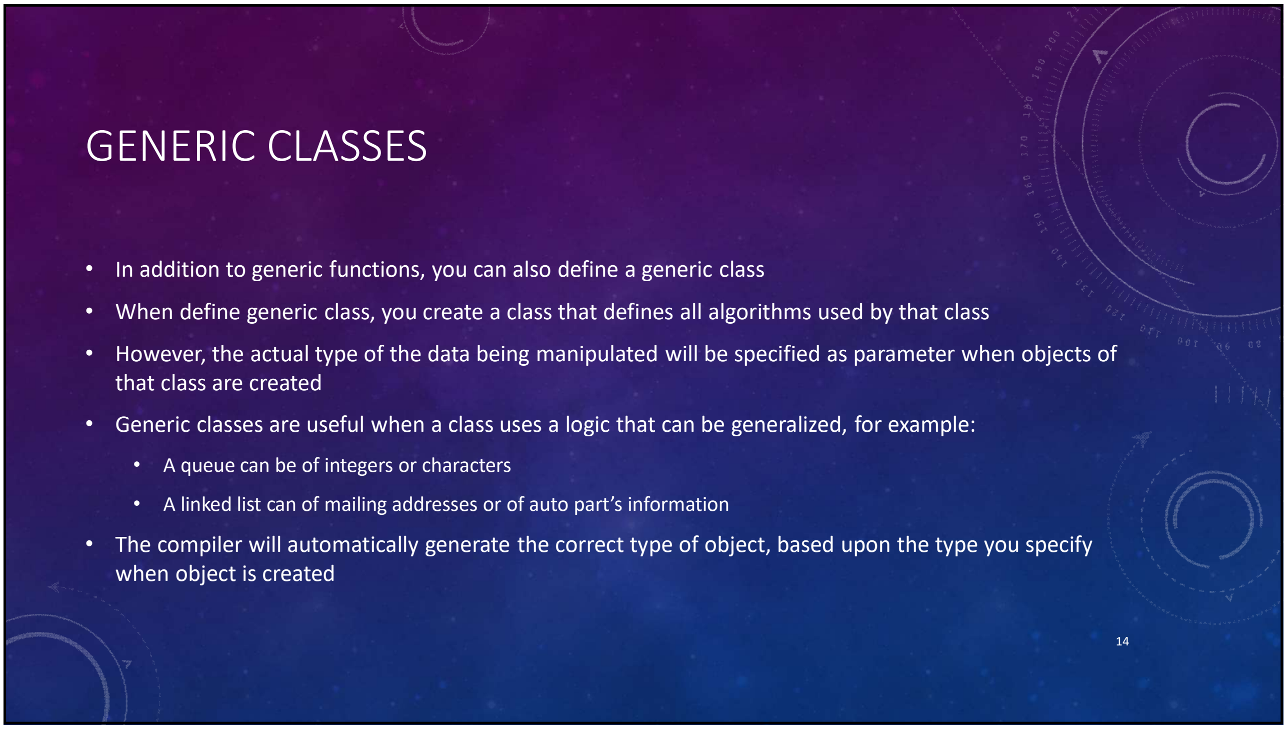


GENERIC FUNCTION RESTRICTIONS

- Generic functions are similar to overloaded functions except that they are restrictive
- When functions are overloaded, you may have different actions performed within the body of each function
- But generic functions must perform same general action for all versions- only the type of data differ
- Sometimes overloaded functions can not be replaced by a generic function just because they differ in data type, because they do not do the same thing.

```
void myfunc(int i){
  cout << “value is: “ << i << endl;
}

void myfunc(double d){
  cout << “integer part is: “ << (int)d << endl;
  cout << “fractional part is: “ << d – (int)d << endl;
}
```

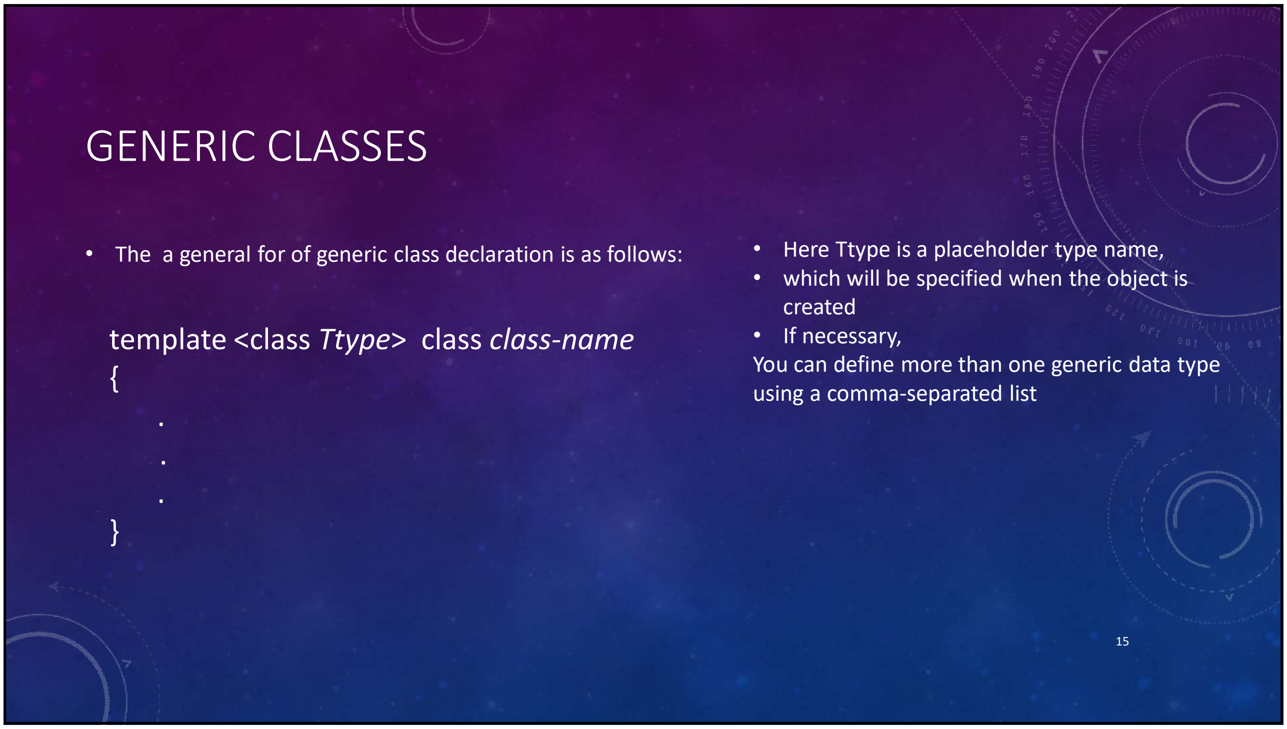
13



GENERIC CLASSES

- In addition to generic functions, you can also define a generic class
- When define generic class, you create a class that defines all algorithms used by that class
- However, the actual type of the data being manipulated will be specified as parameter when objects of that class are created
- Generic classes are useful when a class uses a logic that can be generalized, for example:
  - A queue can be of integers or characters
  - A linked list can of mailing addresses or of auto part’s information
- The compiler will automatically generate the correct type of object, based upon the type you specify when object is created

14



GENERIC CLASSES

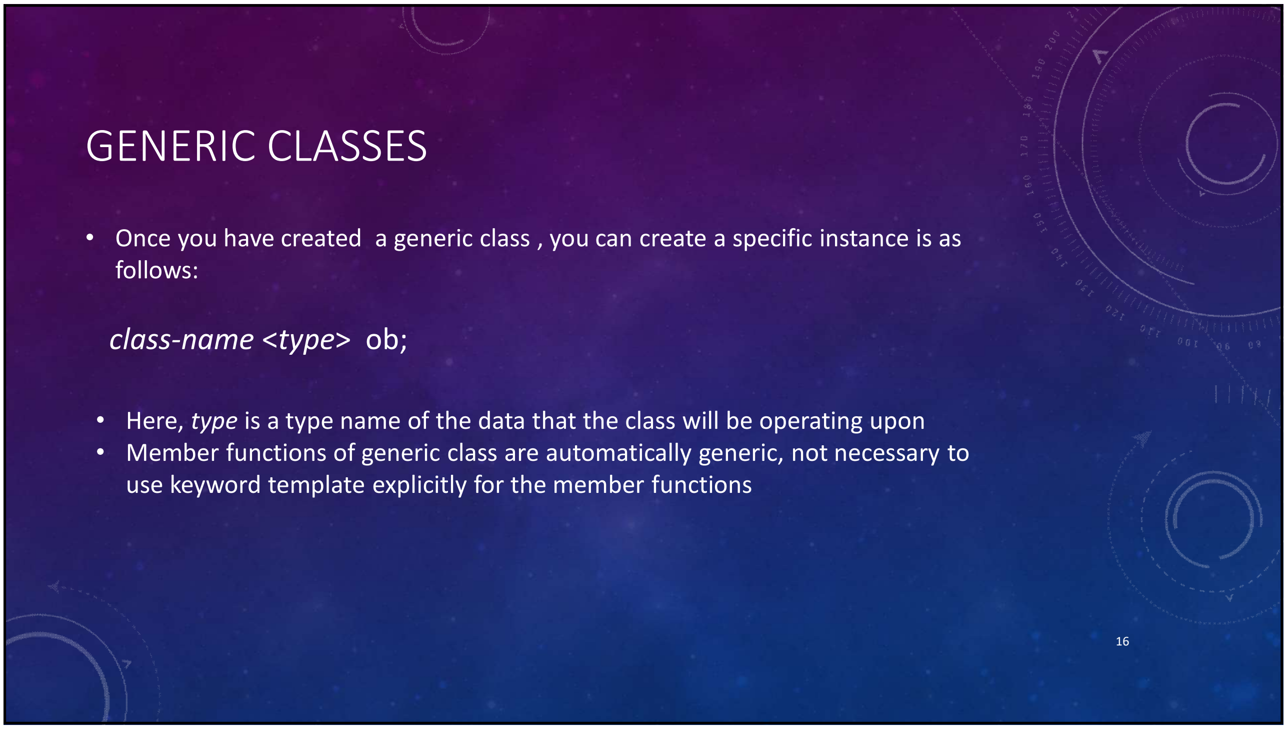
- The a general for of generic class declaration is as follows:

```
template <class Ttype>  class class-name
{
        .
        .
        .
}
```

- Here Ttype is a placeholder type name,
- which will be specified when the object is created
- If necessary,

You can define more than one generic data type using a comma-separated list

15



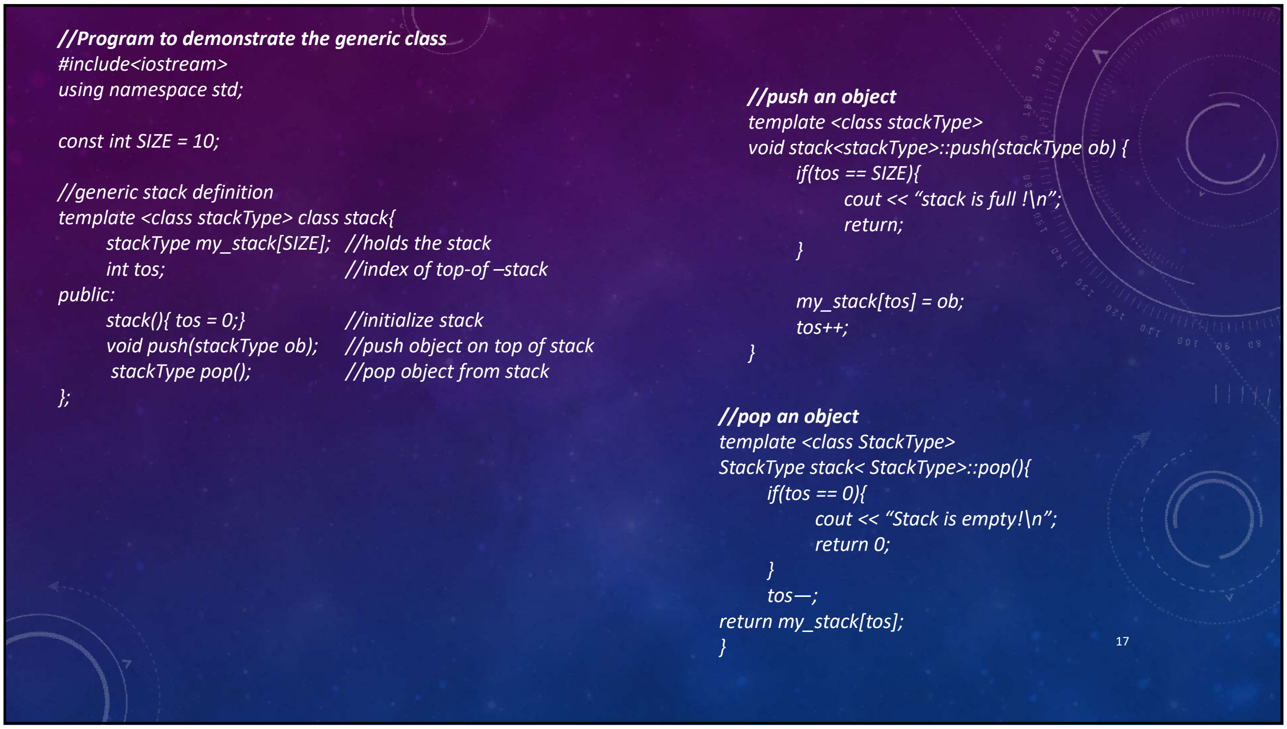
GENERIC CLASSES

- Once you have created a generic class , you can create a specific instance is as follows:

class-name <type> ob;

- Here, *type* is a type name of the data that the class will be operating upon
- Member functions of generic class are automatically generic, not necessary to use keyword template explicitly for the member functions

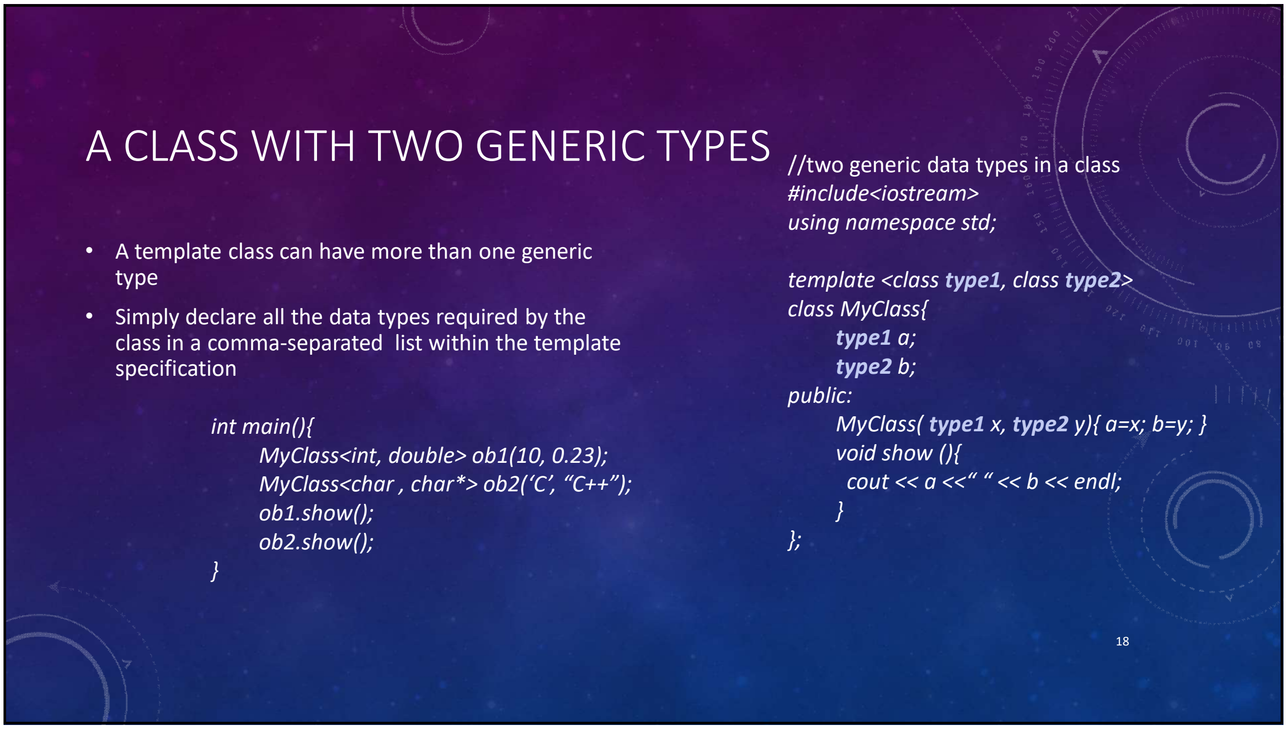
16



```
//Program to demonstrate the generic class
#include<iostream>
using namespace std;

const int SIZE = 10;

//generic stack definition
template <class stackType> class stack{
        stackType my_stack[SIZE];   //holds the stack
        int tos;                    //index of top-of –stack
public:
        stack(){ tos = 0;}          //initialize stack
        void push(stackType ob);    //push object on top of stack
         stackType pop();           //pop object from stack
};
```

```
//push an object
template <class stackType>
void stack<stackType>::push(stackType ob) {
        if(tos == SIZE){
                cout << “stack is full !\n”;
                return;
        }

        my_stack[tos] = ob;
        tos++;
}
```

```
//pop an object
template <class StackType>
StackType stack< StackType>::pop(){
        if(tos == 0){
                cout << “Stack is empty!\n”;
                return 0;
        }
        tos—;
return my_stack[tos];
}
```

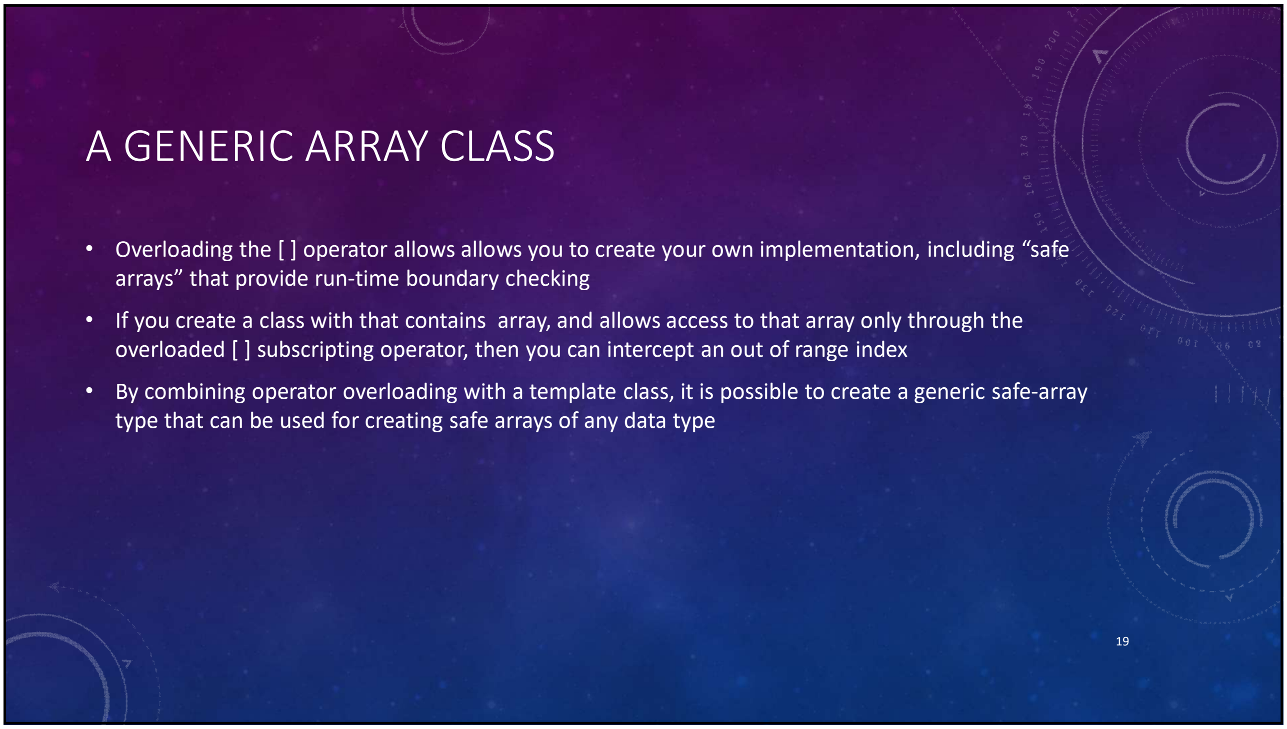


A CLASS WITH TWO GENERIC TYPES

- A template class can have more than one generic type
- Simply declare all the data types required by the class in a comma-separated list within the template specification

```
int main(){
        MyClass<int, double> ob1(10, 0.23);
        MyClass<char , char*> ob2(‘C’, “C++”);
        ob1.show();
        ob2.show();
}
```

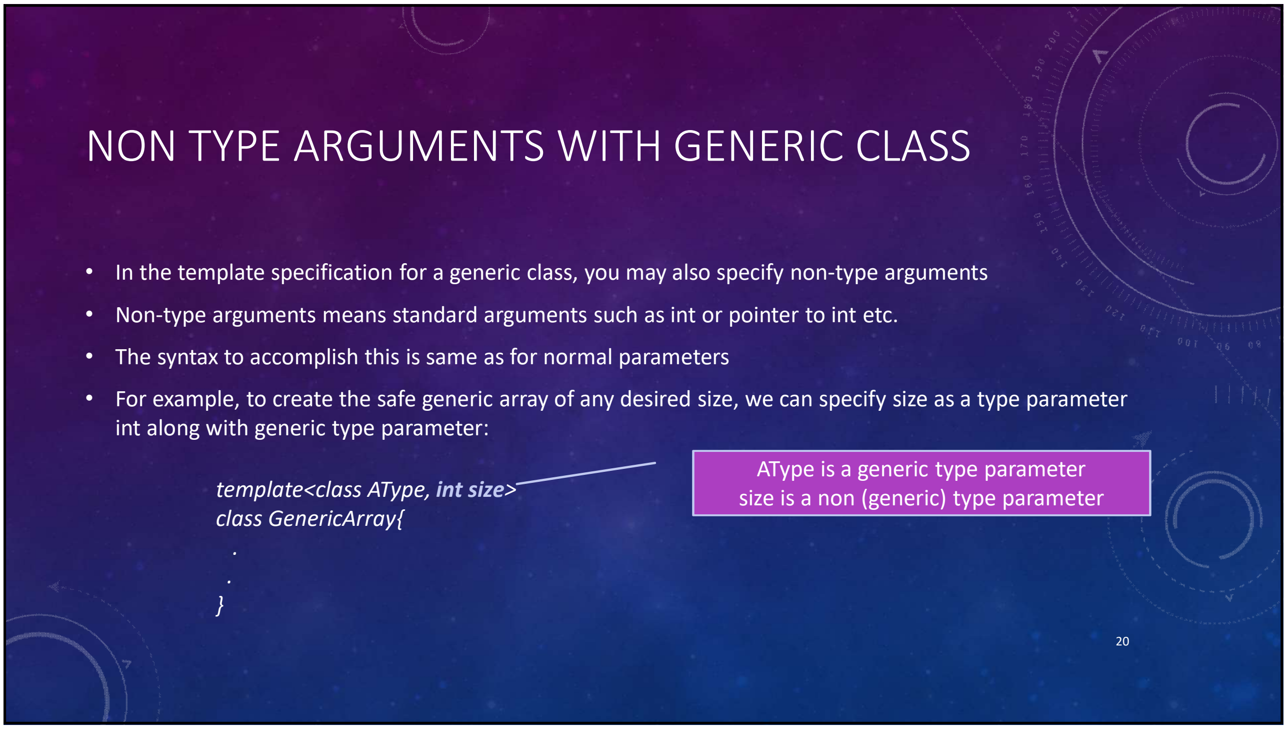
```
//two generic data types in a class
#include<iostream>
using namespace std;

template <class type1, class type2>
class MyClass{
        type1 a;
        type2 b;
public:
        MyClass( type1 x, type2 y){ a=x; b=y; }
        void show (){
          cout << a <<“ “ << b << endl;
        }
};
```



A GENERIC ARRAY CLASS

- Overloading the [ ] operator allows allows you to create your own implementation, including “safe arrays” that provide run-time boundary checking
- If you create a class with that contains array, and allows access to that array only through the overloaded [ ] subscripting operator, then you can intercept an out of range index
- By combining operator overloading with a template class, it is possible to create a generic safe-array type that can be used for creating safe arrays of any data type

19



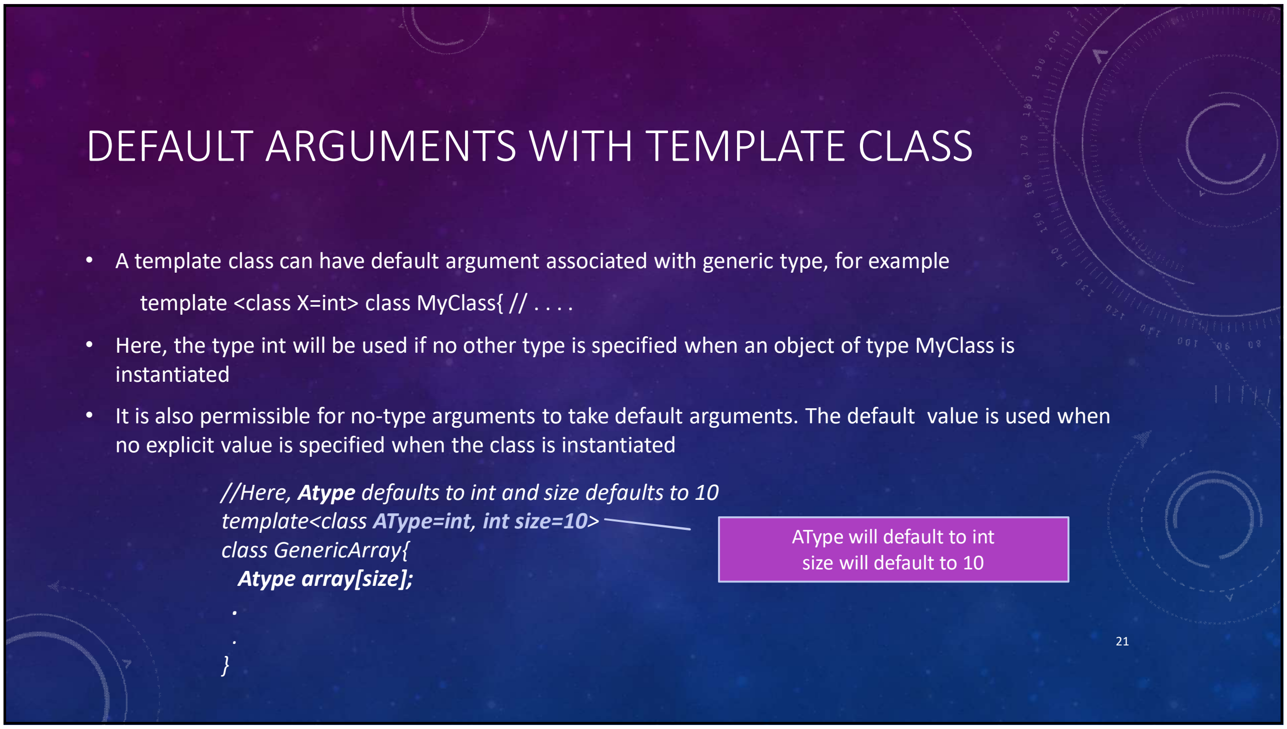
NON TYPE ARGUMENTS WITH GENERIC CLASS

- In the template specification for a generic class, you may also specify non-type arguments
- Non-type arguments means standard arguments such as int or pointer to int etc.
- The syntax to accomplish this is same as for normal parameters
- For example, to create the safe generic array of any desired size, we can specify size as a type parameter int along with generic type parameter:

```
template<class AType, int size>
class GenericArray{
   .
  .
}
```

AType is a generic type parameter
size is a non (generic) type parameter

20



DEFAULT ARGUMENTS WITH TEMPLATE CLASS

- A template class can have default argument associated with generic type, for example

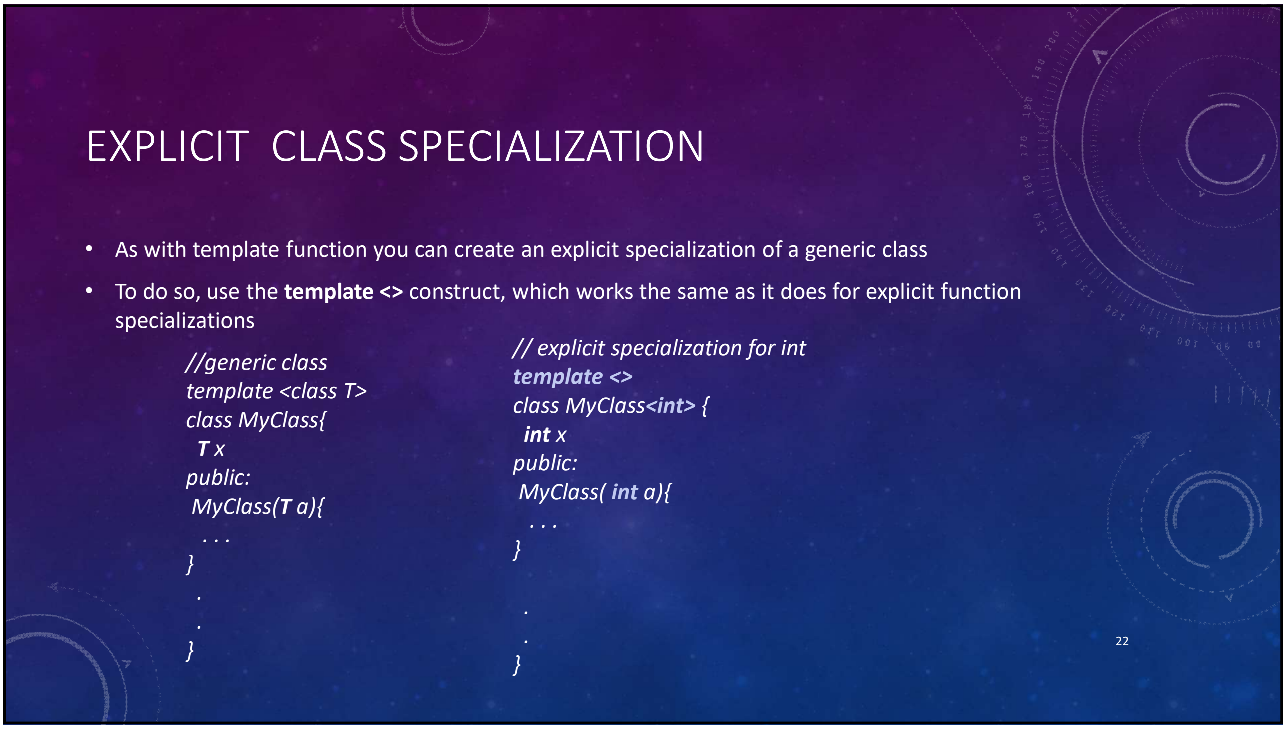
  template <class X=int> class MyClass{ // . . . .

- Here, the type int will be used if no other type is specified when an object of type MyClass is instantiated
- It is also permissible for no-type arguments to take default arguments. The default value is used when no explicit value is specified when the class is instantiated

```
//Here, Atype defaults to int and size defaults to 10
template<class AType=int, int size=10>
class GenericArray{
  Atype array[size];
  .
  .
}
```

AType will default to int
size will default to 10

21



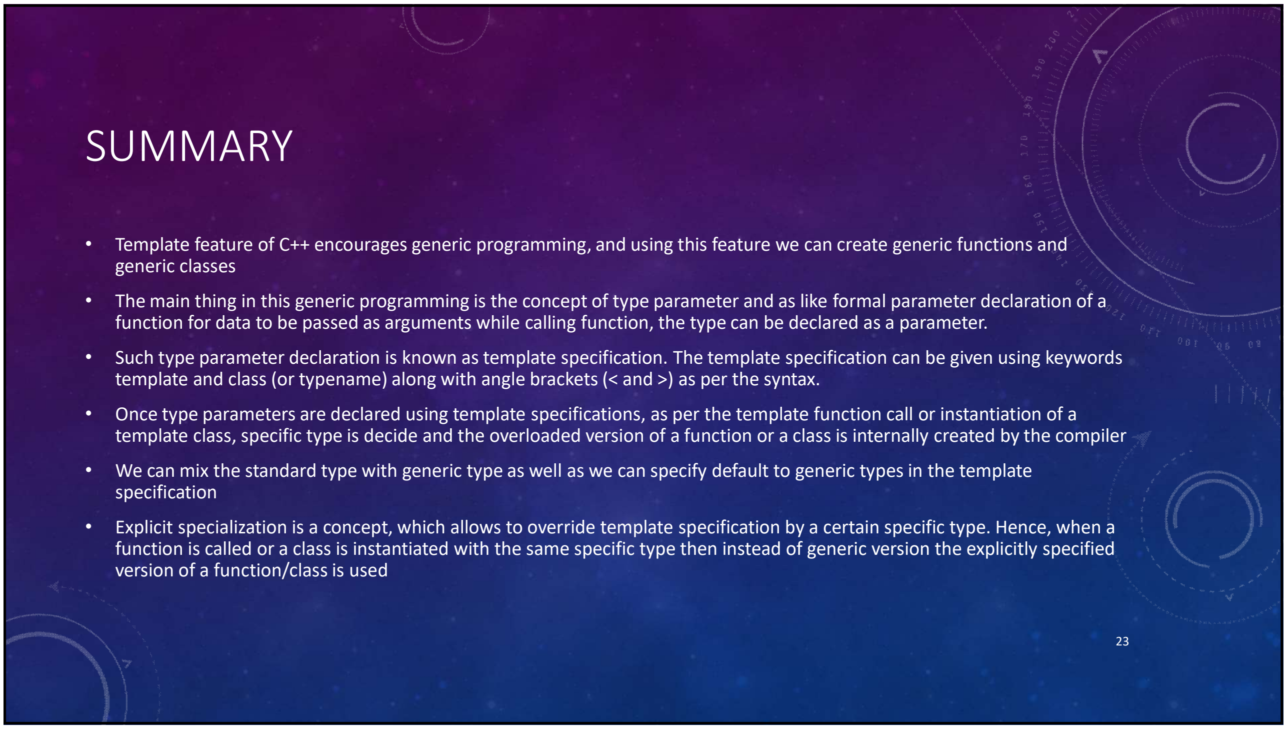
EXPLICIT CLASS SPECIALIZATION

- As with template function you can create an explicit specialization of a generic class
- To do so, use the **template <>** construct, which works the same as it does for explicit function specializations

```
//generic class
template <class T>
class MyClass{
  T x
public:
 MyClass(T a){
   . . .
}
  .
  .
}
```

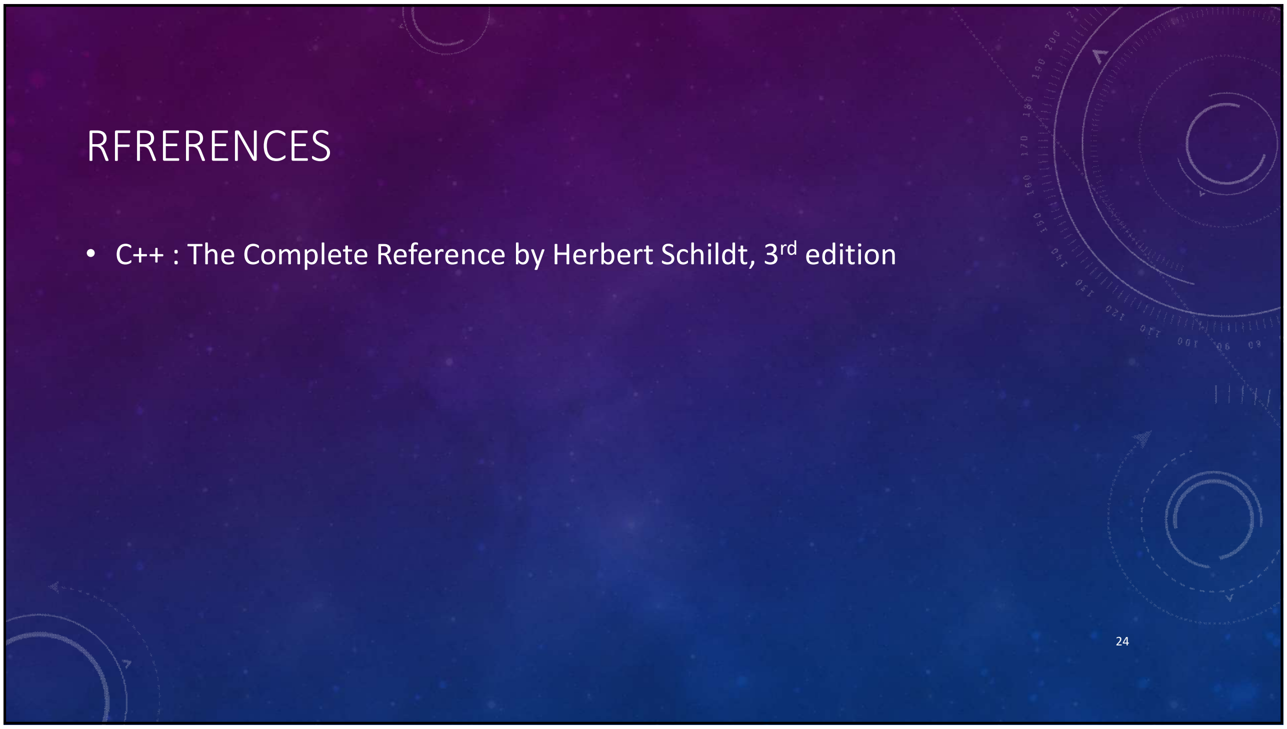
```
// explicit specialization for int
template <>
class MyClass<int> {
  int x
public:
 MyClass( int a){
   . . .
}

  .
  .
}
```

22



SUMMARY

- Template feature of C++ encourages generic programming, and using this feature we can create generic functions and generic classes
- The main thing in this generic programming is the concept of type parameter and as like formal parameter declaration of a function for data to be passed as arguments while calling function, the type can be declared as a parameter.
- Such type parameter declaration is known as template specification. The template specification can be given using keywords template and class (or typename) along with angle brackets (< and >) as per the syntax.
- Once type parameters are declared using template specifications, as per the template function call or instantiation of a template class, specific type is decide and the overloaded version of a function or a class is internally created by the compiler
- We can mix the standard type with generic type as well as we can specify default to generic types in the template specification
- Explicit specialization is a concept, which allows to override template specification by a certain specific type. Hence, when a function is called or a class is instantiated with the same specific type then instead of generic version the explicitly specified version of a function/class is used



RFRERENCES

- C++ : The Complete Reference by Herbert Schildt, 3rd edition