Chapter 13 - Overloading and Templates

Spring 2022

Objectives

- In this chapter, you will:
  - Learn about overloading
  - Become familiar with the restrictions on operator overloading
  - Examine the pointer this
  - Learn about friend functions
  - Learn how to overload operators as members and nonmembers of a class
  - Discover how to overload various operators
  - Become familiar with the requirements for classes with pointer member variables
  - Learn about templates
  - Explore how to construct function templates and class templates
  - Become aware of C++14 random number generators

Introduction

- **Templates** enable you to write generic code for related functions and classes
- **Function templates** simplify function overloading

Why Operator Overloading Is Needed (1 of 2)

- Consider the following statements:

```
clockType myClock(8, 23, 34);
clockType yourClock(4, 5, 30);
```

- Which version of C++ statements would you prefer?

```
myClock.printTime();
myClock.incrementSeconds();
if (myClock.equalTime(yourClock)) { ...

  -- OR --

cout << myClock;
myClock++;
if (myClock == yourClock) { ...
```

Why Operator Overloading Is Needed (2 of 2)

- Assignment and member selection are the only built-in operations on classes
  - Other operators cannot be applied directly to class objects

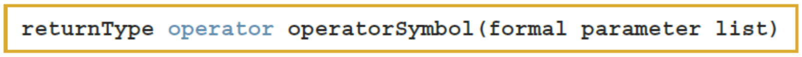
- **Operator overloading** extends the definition of an operator to work with a userdefined data type
  - C++ allows you to extend the definitions of most of the operators to work with classes

Operator Overloading

- Most existing C++ operators can be overloaded to manipulate class objects
- New operators cannot be created
- An **operator function** is a function that overloads an operator
  - Use reserved word `operator` followed by the operator as the function name

Syntax for Operator Functions

- Syntax of an operator function heading:



```
returnType operator operatorSymbol(formal parameter list)
```

  - It is a value-returning function
  - `operator` is a reserved word
- To overload an operator for a class:
  - Include the operator function declaration in the class definition
  - Write the definition of the operator function

Overloading an Operator: Some Restrictions

- Cannot change the precedence of an operator
- Associativity cannot be changed
- Default parameters cannot be used
- Cannot change number of parameters
- Cannot create new operators
- Cannot overload: `. .* :: ?: sizeof`
- How the operator works with built-in types remains the same
- Can overload for user-defined objects or for a combination of user-defined and built-in objects

Pointer `this`

- Every object of a class maintains a (hidden) pointer to itself called `this`
- When an object invokes a member function
  - `this` is referenced by the member function

Friend Functions of Classes

- A **friend function** (of a class) is a nonmember function of the class that has access to all the members of the class

- Use the reserved word `friend` in the function prototype in the class definition

- Friendship is always given by the class

```
class classIllusFriend {
    friend void two(/*parameters*/);
    .
    .
    .
};
```

Definition of a friend Function

- friend does not appear in the heading of the function’s definition
- When writing the friend function’s definition
  - The name of the class and the scope resolution operator are not used

```
void two(/*parameters*/) {
    .
    .
    .
}
```

Operator Functions as Member and Nonmember Functions

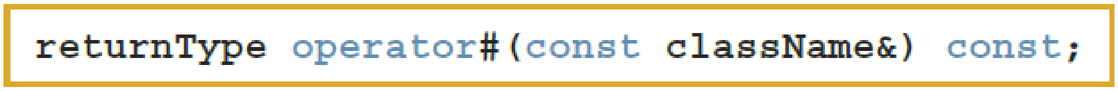
- To overload (), [], ->, or = for a class, the function must be a member of the class
- Suppose op is overloaded for opOverClass:
  - If the leftmost operand of op is an object of a different type, the overloading function must be a nonmember (friend) of the class
  - If the overloading function for op is a member of opOverClass, then when applying op on objects of type opOverClass, the leftmost operand must be of type opOverClass

Overloading Binary Operators

- If # represents a binary operator (e.g., + or ==) that is to be overloaded for rectangleType
  - It can be overloaded as either a member function of the class or as a friend function

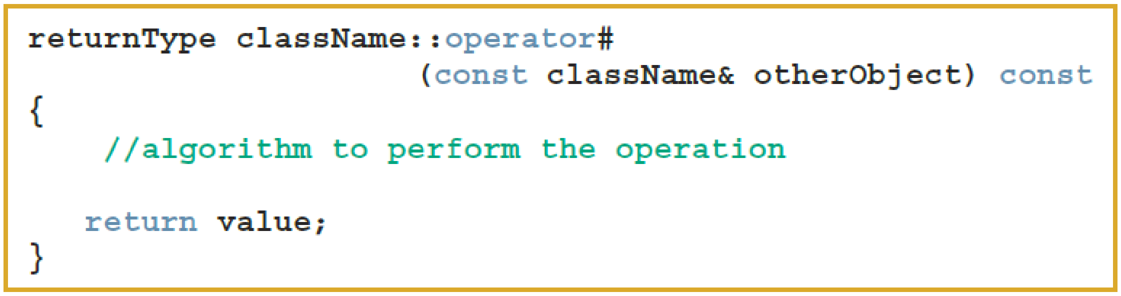
Overloading the Binary Operators as Member Functions

- Function prototype (included in the class definition):



```
returnType operator#(const className&) const;
```

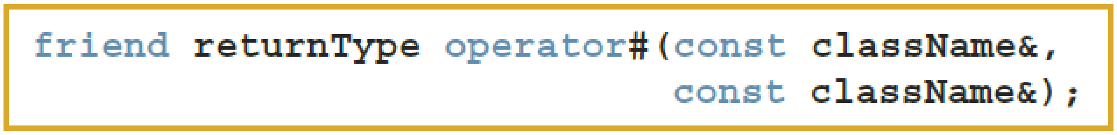
- Function definition:



```
returnType className::operator#
                     (const className& otherObject) const
{
    //algorithm to perform the operation

   return value;
}
```

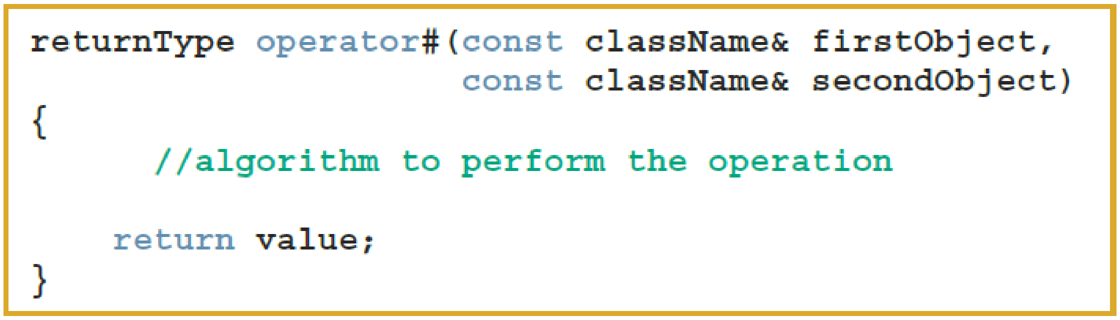
Overloading the Binary Operators (Arithmetic or Relational) as Nonmember Functions

- Function prototype (included in class definition):



```
friend returnType operator#(const className&,
                            const className&);
```

- Function definition:



```
returnType operator#(const className& firstObject,
                     const className& secondObject)
{
      //algorithm to perform the operation

    return value;
}
```

Overloading the Stream Insertion (<<) and Extraction (>>) Operators

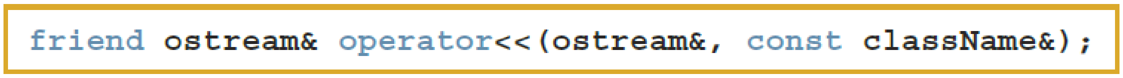
- Consider the expression:

  `cout << myRectangle;`

  - Leftmost operand is an `ostream` object, not a `rectangleType` object
- Thus, the operator function that overloads << for `rectangleType` must be a nonmember function of the class
  - The same applies to the function that overloads >>

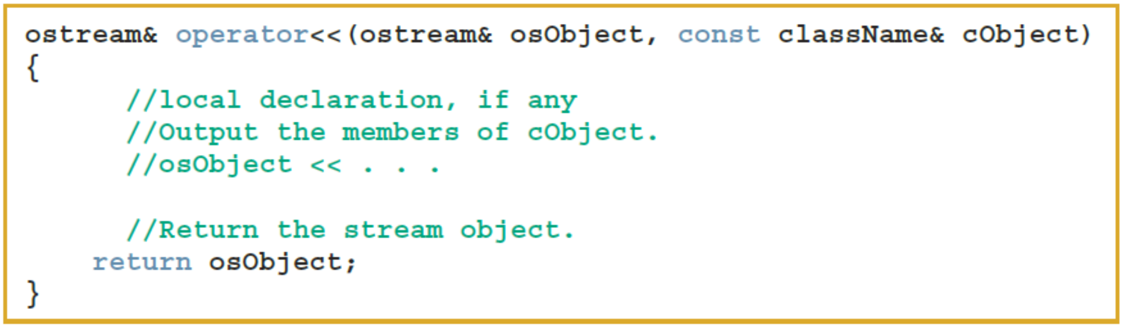
Overloading the Stream Insertion Operator (<<)

- Function prototype:



```
friend ostream& operator<<(ostream&, const className&);
```

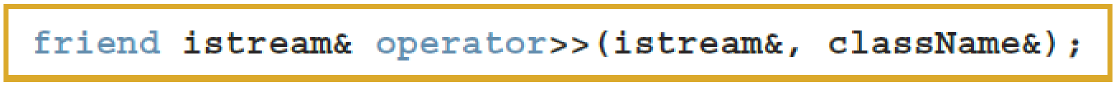
- Function definition:



```
ostream& operator<<(ostream& osObject, const className& cObject)
{
      //local declaration, if any
      //Output the members of cObject.
      //osObject << . . .

      //Return the stream object.
    return osObject;
}
```

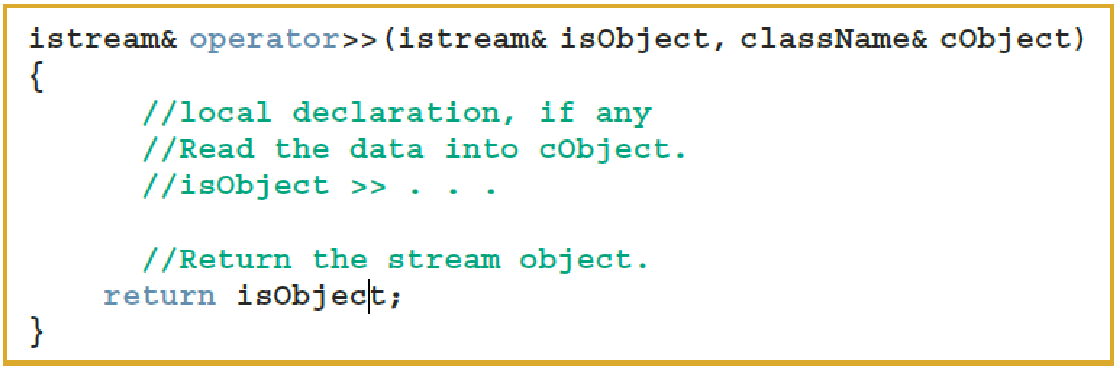
Overloading the Stream Extraction Operator (>>)

- Function prototype:



```
friend istream& operator>>(istream&, className&);
```

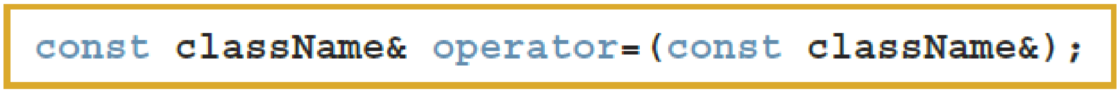
- Function definition:



```
istream& operator>>(istream& isObject, className& cObject)
{
      //local declaration, if any
      //Read the data into cObject.
      //isObject >> . . .

      //Return the stream object.
    return isObject;
}
```

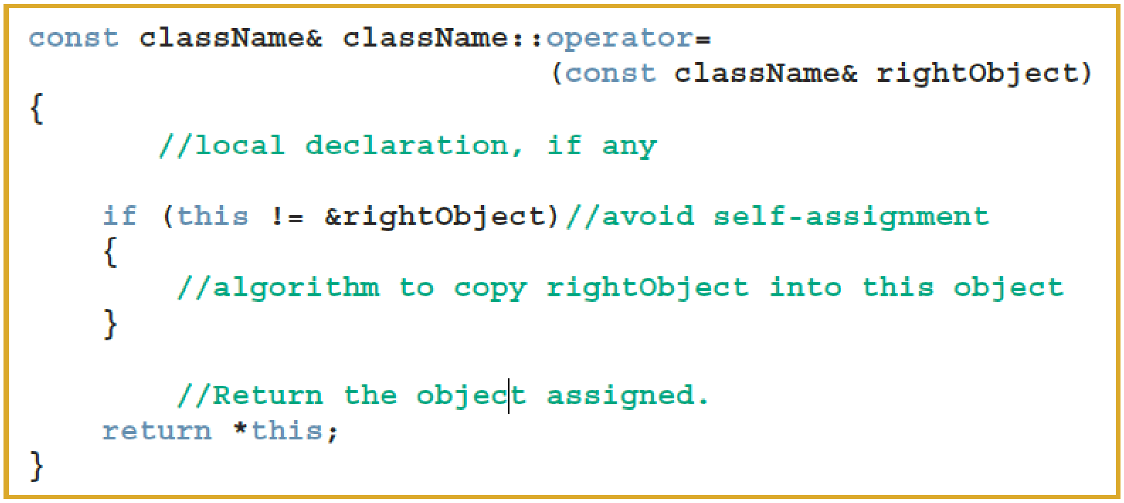
Overloading the Assignment Operator (=)

- Function prototype:



```
const className& operator=(const className&);
```

- Function definition:



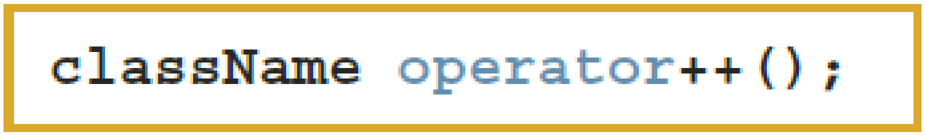
```
const className& className::operator=
                            (const className& rightObject)
{
       //local declaration, if any

    if (this != &rightObject)//avoid self-assignment
    {
        //algorithm to copy rightObject into this object
    }

        //Return the object assigned.
    return *this;
}
```

Overloading Unary Operators

- To overload a unary operator for a class:
  - If the operator function is a member of the class, it has no parameters
  - If the operator function is a nonmember (i.e., a `friend` function), it has one parameter

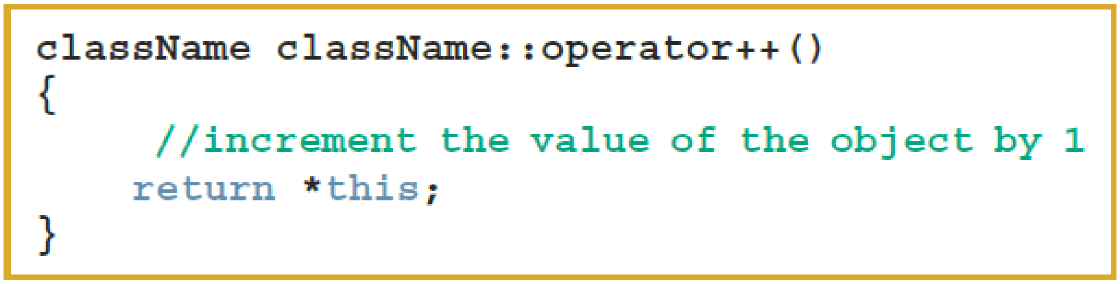
Overloading the Increment (++) and Decrement (--) Operators (1 of 4)

- General syntax to overload the pre-increment operator ++ as a **member** function
  - Function prototype:



```
className operator++();
```

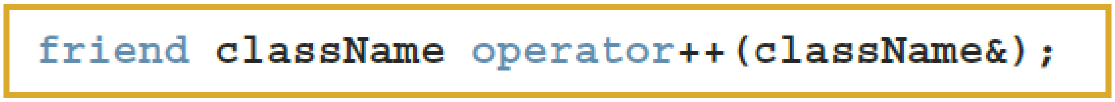
  - Function definition:



```
className& className::operator++()
{
     //increment the value of the object by 1
    return *this;
}
```

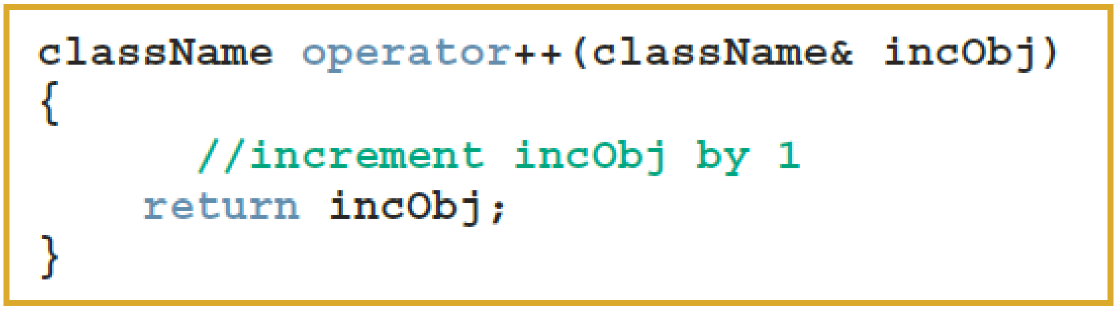
Overloading the Increment (++) and Decrement (--) Operators (2 of 4)

- General syntax to overload the pre-increment operator ++ as a **nonmember** function
  - Function prototype:



```
friend className operator++(className&);
```

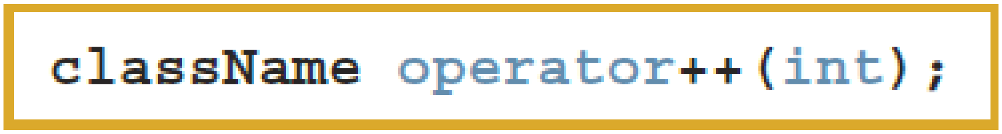
- Function definition:



```
className operator++(className& incObj)
{
        //increment incObj by 1
     return incObj;
}
```

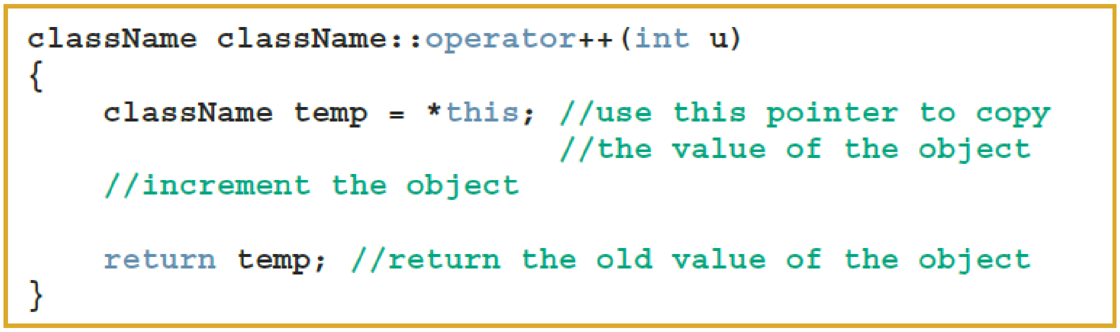
Overloading the Increment (++) and Decrement (--) Operators (3 of 4)

- General syntax to overload the post-increment operator ++ as a **member** function:
  - Function prototype:



```
className operator++(int);
```

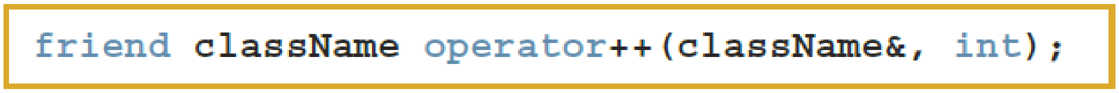
  - Function definition:



```
className className::operator++(int u)
{
    className temp = *this; //use this pointer to copy
                            //the value of the object
    //increment the object

    return temp; //return the old value of the object
}
```

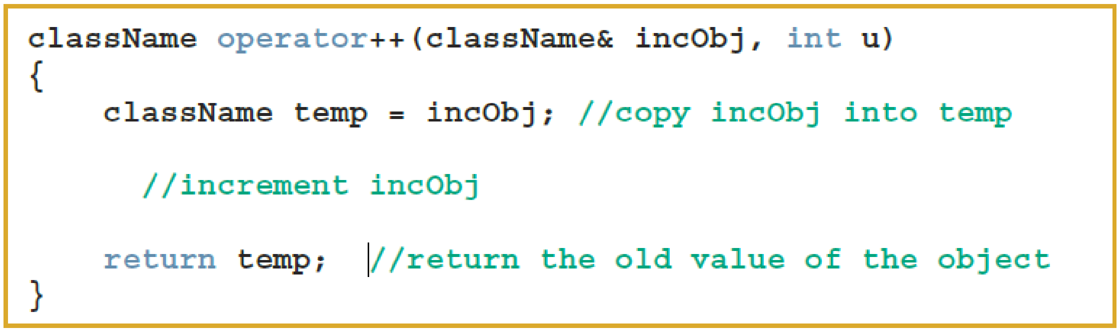
Overloading the Increment (++) and Decrement (--) Operators (4 of 4)

- General syntax to overload the post-increment operator ++ as a **nonmember** function:
  - Function prototype:



```
friend className operator++(className&, int);
```

  - Function definition:



```
className operator++(className& incObj, int u)
{
    className temp = incObj; //copy incObj into temp

      //increment incObj

    return temp;  //return the old value of the object
}
```

Operator Overloading: Member versus Nonmember (1 of 2)

- Some operators must be overloaded as member functions and some must be overloaded as nonmember (`friend`) functions
- Binary arithmetic operator + can be overloaded either way
  - As a member function, operator + has direct access to data members of one of the objects
  - Need to pass only one object as a parameter

Operator Overloading: Member versus Nonmember (2 of 2)

- Overload + as a nonmember function
  - Must pass both objects as parameters
  - Code may be somewhat clearer this way

Classes and Pointer Member Variables (Revisited)

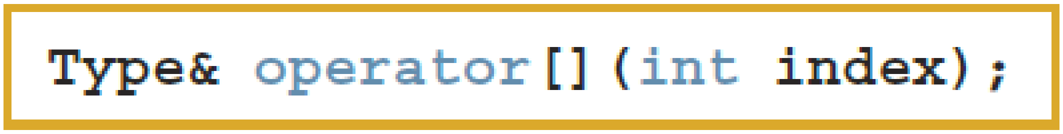
- Recall that the assignment operator copies member variables from one object to another of the same type
  - Does not work well with pointer member variables
- Classes with pointer member variables must:
  - Explicitly overload the assignment operator
  - Include the copy constructor
  - Include the destructor

Operator Overloading: One Final Word

- If an operator op is overloaded for a class, e.g., `rectangleType`
  - When you use op on objects of type `rectangleType`, the body of the function that overloads the operator op for the class rectangleType executes
  - Therefore, whatever code you put in the body of the function executes

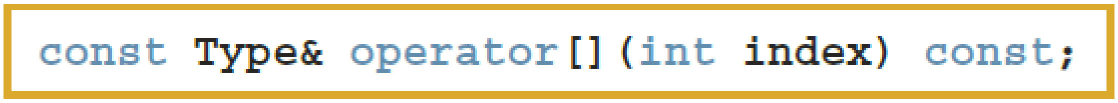
Overloading the Array Index (Subscript) Operator ([])

- Syntax to declare `operator[]` as a member of a class for non-constant arrays:



```
Type& operator[](int index);
```

- Syntax to declare `operator[]` as a member of a class for constant arrays:



```
const Type& operator[](int index) const;
```

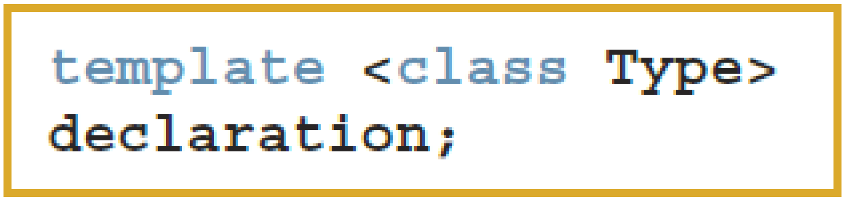
Function Overloading

- Overloading a function refers to having several functions with the same name, but different parameters
  - The parameter list determines which function will execute
  - Must provide the definition of each function

Templates (1 of 2)

- Template: a single code body for a set of related functions (**function template**) and related classes (**class template**)

- Syntax:



```
template <class Type>
declaration;
```

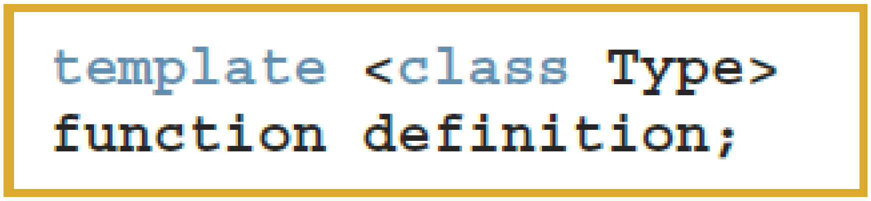
  - **Type** is the data type
  - **declaration** is either a function declaration or a class declaration

Templates (2 of 2)

- `class` in the heading refers to any user-defined type or built-in type
- **Type** is a formal parameter to the template
- Just as variables are parameters to functions, data types are parameters to templates

Function Templates

- Syntax of the function template:



```
template <class Type>
function definition;
```

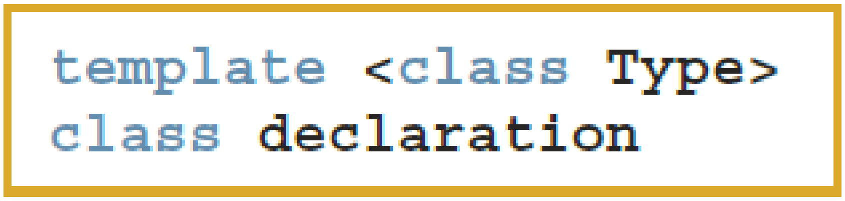
- **Type** is a formal parameter of the template used to:

  - Specify type of parameters to the function
  - Specify return type of the function
  - Declare variables within the function

Class Templates

- Class template: a single code segment for a set of related classes • Called parameterized types

- Syntax:



```
template <class Type>
class declaration
```

- A template instantiation can be created with either a built-in or user-defined type

- The function members of a class template are considered to be function templates

Header File and Implementation File of a Class Template (1 of 2)

- Passing a parameter to a function takes effect at run time
- Passing a parameter to a class template takes effect at compile time
- Cannot compile the implementation file independently of the client code
  - Can put class definition and definitions of the function templates directly in the client code
  - Can put class definition and the definitions of the function templates in the same header file

Header File and Implementation File of a Class Template (2 of 2)

- Another alternative is to put class definition and function definitions in separate files
  - Include directive to the implementation file at the end of the header file
- In either case, function definitions and client code are compiled together

C++14 Random Number Generator

- To use C++14 random number generator functions we use an engine and a distributor
  - An engine returns unpredictable (random) bits
  - A distribution returns random numbers whose likelihoods correspond to a specific shape such as a uniform or normal distribution
- The C++14 standard library provides 25 distribution types in five categories
  - `uniform_int_distribution` and `uniform_real_distribution` fall in the category of uniform distributions

Quick Review

- An operator that has different meanings with different data types is said to be overloaded
- Operator function: a function that overloads an operator

    - `operator` is a reserved word
    - Operator functions are value-returning
- Operator overloading provides the same concise notation for user-defined data types as for built-in data types
- Only existing operators can be overloaded
- The pointer `this` refers to the object
- A `friend` function is a nonmember of a class
- If an operator function is a member of a class
    - The leftmost operand of the operator must be a class object (or a reference to a class object) of that operator's class
- Classes with pointer variables must overload the assignment operator, and include both the copy constructor and the destructor
- In C++, `template` is a reserved word
    - Function template: a single code segment for a set of related functions
    - Class template: a single code segment for a set of related classes - Are called parameterized types
- C++14 provides many functions to implement random number generator.

Questions?