Templates, Exceptions, and Files

Function Templates

Functions and classes help to make programs easier to write, safer, and more maintainable. However, while functions and classes do have all of those advantages, in certain cases they can also be somewhat limited by C++'s requirement that you specify types for all of your parameters.

For example, you might want to write a function that calculates the sum of two numbers, similar to this:

```
int sum(int a, int b) {
 return a+b;
}
```

You can use templates to define functions as well as classes. Let's see how they work.

Function Templates

We can now call the function for two integers in our main.

```
int sum(int a, int b) {
 return a+b;
}

int main () {
 int x=7, y=15;
 cout << sum(x, y) << endl;
}
// Outputs 22
```

Try It Yourself

The function works as expected, but is limited solely to **integers**.

Function Templates

It becomes necessary to write a new function for each new type, such as doubles.

```
double sum(double a, double b) {
 return a+b;
}
```

Wouldn't it be much more efficient to be able to write one version of sum() to work with parameters of **any** type?
Function templates give us the ability to do that!
With function templates, the basic idea is to avoid the necessity of specifying an exact type for each variable. Instead, C++ provides us with the capability of defining functions using placeholder types, called **template type parameters**.

To define a function template, use the keyword **template**, followed by the template type definition:

```
template <class T>
```

We named our template type **T**, which is a generic data type.
Tap **Continue** to learn more!

Function Templates

Now we can use our generic data type **T** in the function:

```
template <class T>
T sum(T a, T b) {
 return a+b;
}

int main () {
   int x=7, y=15;
   cout << sum(x, y) << endl;
}

// Outputs 22
```

Try It Yourself

The function returns a value of the generic type T, taking two parameters, also of type T.

Our new function worked exactly as the previous one for integer values did.

Function Templates

The same function can be used with other data types, for example doubles:

```
template <class T>
T sum(T a, T b) {
 return a+b;
}

int main () {
 double x=7.15, y=15.54;
 cout << sum(x, y) << endl;
}
// Outputs 22.69
```

Try It Yourself

The compiler automatically calls the function for the corresponding type.

When creating a template type parameter, the keyword **typename** may be used as an alternative to the keyword **class**: **template <typename T>**.
In this context, the keywords are identical, but throughout this course, we'll use the keyword **class**.

Function Templates

Template functions can save a lot of time, because they are written only once, and work with different types.
Template functions reduce code maintenance, because duplicate code is reduced significantly.

> Enhanced safety is another advantage in using template functions, since it's not necessary to manually copy functions and change types.

Function Templates

Function templates also make it possible to work with **multiple** generic data types. Define the data types using a comma-separated list.
Let's create a function that compares arguments of varying data types (an int and a **double**), and prints the smaller one.

```
template <class T, class U>
```

> As you can see, this template declares two different generic data types, **T** and **U**.

Function Templates

Now we can continue with our function declaration:

```
template <class T, class U>
T smaller(T a, U b) {
  return (a < b ? a : b);
}
```

> The ternary operator checks the a<b condition and returns the corresponding result. The expression (a < b ? a : b) is equivalent to the expression if **a** is smaller than **b**, return **a**, else, return **b**.

Function Templates

In our main, we can use the function for different data types:

```
template <class T, class U>
T smaller(T a, U b) {
  return (a < b ? a : b);
}

int main () {
  int x=72;
  double y=15.34;
  cout << smaller(x, y) << endl;
}

// Outputs 15
```

Try It Yourself

The output converts to an **integer**, because we specified the function template's return type to be of the same type as the first parameter (T), which is an integer.

Function Templates

T is short for Type, and is a widely used name for type parameters.
It's not necessary to use **T**, however; you can declare your type parameters using any identifiers that work for you. The only terms you need to avoid are C++ keywords.

Remember that when you declare a template parameter, you absolutely **must** use it in your function definition. Otherwise, the compiler will complain!

Class Templates

Just as we can define function templates, we can also define **class templates**, allowing classes to have members that use template parameters as types.
The same syntax is used to define the class template:

```
template <class T>
class MyClass {

};
```

Just as with function templates, you can define more than one generic data type by using a comma-separated list.

Class Templates

As an example, let's create a class **Pair**, that will be holding a pair of values of a generic type.

```
template <class T>
class Pair {
 private:
  T first, second;
 public:
  Pair (T a, T b):
   first(a), second(b) {
  }
};
```

The code above declares a class template **Pair**, with two private variables of a generic type, and one constructor to initialize the variables.

Class Templates

A specific syntax is required in case you define your member functions outside of your class - for example in a separate source file.

You need to specify the **generic type** in angle brackets after the class name.
For example, to have a member function **bigger()** defined outside of the class, the following syntax is used:

```
template <class T>
class Pair {
 private:
  T first, second;
 public:
  Pair (T a, T b):
   first(a), second(b){
  }
  T bigger();
};

template <class T>
T Pair<T>::bigger() {
  // some code
}
```

A specific syntax is required in case you define your member functions outside of your class.

Class Templates

The **bigger** function returns the greater value of the two member variables.

```
template <class T>
class Pair {
 private:
  T first, second;
 public:
  Pair (T a, T b):
   first(a), second(b){
  }
  T bigger();
};

template <class T>
T Pair<T>::bigger() {
  return (first>second ? first : second);
}
```

The ternary operator compares the two variables, returning the greater one.

Class Templates

To create objects of the template class for different types, specify the data type in angle brackets, as we did when defining the function outside of the class.
Here, we create a Pair object for integers.

```
Pair <int> obj(11, 22);
cout << obj.bigger();
// Outputs 22
```

Try It Yourself

We can use the same class to create an object that stores any other type.

```
Pair <double> obj(23.43, 5.68);
cout << obj.bigger();
// Outputs 23.43
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Template Specialization

In case of regular class templates, the way the class handles different data types is the same; the same code runs for all data types.
Template specialization allows for the definition of a different implementation of a template when a specific type is passed as a template argument.

For example, we might need to handle the character data type in a different manner than we do numeric data types.
To demonstrate how this works, we can first create a regular template.

```
template <class T>
class MyClass {
 public:
  MyClass (T x) {
   cout <<x<<" -  not a char"<<endl;
  }
};
```

As a regular class template, **MyClass** treats all of the various data types in the same way.

Template Specialization

To specify different behavior for the data type **char**, we would create a template specialization.

```
template <class T>
class MyClass {
 public:
  MyClass (T x) {
   cout <<x<<" -  not a char"<<endl;
  }
};

template < >
class MyClass<char> {
 public:
  MyClass (char x) {
   cout <<x<<" is a char!"<<endl;
  }
};
```

First of all, notice that we precede the class name with **template<>**, including an empty parameter list. This is because all types are known and no template arguments are required for this specialization, but still, it is the specialization of a class template, and thus it requires to be noted as such.

But more important than this prefix, is the <**char**> specialization parameter after the class template name. This specialization parameter itself identifies the type for which the template class is being specialized (**char**).

> In the example above, the first class is the generic template, while the second is the specialization.
> If necessary, your specialization can indicate a completely different behavior from the behavior of the generic template.

Template Specialization

The next step is to declare objects of different types and check the result:

```
int main () {
  MyClass<int> ob1(42);
  MyClass<double> ob2(5.47);
  MyClass<char> ob3('s');
}
/* Output:
42 - not a char
5.47 - not a char
s is a char!
*/
```

Try It Yourself

As you can see, the generic template worked for int and **double**. However, our template specialization was invoked for the char data type.

> Keep in mind that there is no member "inheritance" from the generic template to the specialization, so all members of the template class specializations must be defined on their own.

Exceptions

Problems that occur during program execution are called **exceptions**.
In C++ exceptions are responses to anomalies that arise while the program is running, such as an attempt to divide by zero.

> Tap **Continue** to learn about C++ mechanisms for handling exceptions.

Throwing Exceptions

C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**.
throw is used to throw an exception when a problem shows up.
For example:

```
int motherAge = 29;
int sonAge = 36;
if (sonAge > motherAge) {
  throw "Wrong age values";
}
```

The code looks at **sonAge** and **motherAge**, and throws an exception if **sonAge** is found to be the greater of the two.

> In the **throw** statement, the operand determines a type for the exception. This can be any expression. The type of the expression's result will determine the type of the exception thrown.

Catching Exceptions

A **try** block identifies a block of code that will activate specific exceptions. It's followed by one or more **catch** blocks. The **catch** keyword represents a block of code that executes when a particular exception is thrown.
Code that could generate an exception is surrounded with the **try/catch** block.
You can specify what type of exception you want to catch by the exception declaration that appears in parentheses following the keyword **catch**.
For example:

```
try {
  int motherAge = 29;
  int sonAge = 36;
  if (sonAge > motherAge) {
   throw 99;
  }
}
catch (int x) {
  cout<<"Wrong age values - Error "<<x;
}

//Outputs "Wrong age values - Error 99"
```

Try It Yourself

The **try** block throws the exception, and the **catch** block then handles it.
The error code 99, which is an integer, appears in the **throw** statement, so it results in an exception of type **int**.

> Multiple **catch** statements may be listed to handle various exceptions in case multiple exceptions are thrown by the try block.

Exception Handling

Exception handling is particularly useful when dealing with user input.
For example, for a program that requests user input of two numbers, and then outputs their division, be sure that you handle division by zero, in case your user enters 0 as the second number.

```
int main() {
  int num1;
  cout <<"Enter the first number:";
  cin >> num1;

  int num2;
  cout <<"Enter the second number:";
  cin >> num2;

  cout <<"Result:"<<num1 / num2;
}
```

Try It Yourself

This program works perfectly if the user enters any number besides 0.

In case of 0 the program crashes, so we need to handle that input.

Exception Handling

In the event that the second number is equal to 0, we need to **throw** an exception.

```
int main() {
 int num1;
 cout <<"Enter the first number:";
 cin >> num1;

 int num2;
 cout <<"Enter the second number:";
 cin >> num2;

 if(num2 == 0) {
  throw 0;
 }

 cout <<"Result:"<<num1 / num2;
}
```

Try It Yourself

This code throws an exception with the code 0 of type integer.

Next stop: Using the **try/catch** block to handle the exception!

Exception Handling

Now we need to handle the thrown exception using a **try/catch** block.

```
int main() {
 try {
  int num1;
  cout <<"Enter the first number:";
  cin >> num1;

  int num2;
  cout <<"Enter the second number:";
  cin >> num2;

  if(num2 == 0) {
   throw 0;
  }

  cout <<"Result:"<<num1 / num2;
 }
 catch(int x) {
  cout <<"Division by zero!";
 }
}
```

Try It Yourself

This results in the output of "Division by zero!" as an alternative to a program crash, when 0 is entered as the second number.

In our case, we catch exceptions of type **integer** only. It's possible to specify that your catch block handles any type of exception thrown in a try block. To accomplish this, add an **ellipsis (...)** between the parentheses of catch:

```
try {
  // code
} catch(...) {
  // code to handle exceptions
}
```

Tap **Try It Yourself** to play around with the code!

Working with Files

Another useful C++ feature is the ability to read and write to files. That requires the standard C++ library called **fstream**.
Three new data types are defined in fstream:
ofstream: Output file stream that creates and writes information to files.
ifstream: Input file stream that reads information from files.
fstream: General file stream, with both ofstream and ifstream capabilities that allow it to create, read, and write information to files.

To perform file processing in C++, header files <**iostream**> and <**fstream**> must be included in the C++ source file.

```
#include <iostream>
#include <fstream>
```

These classes are derived directly or indirectly from the classes istream and ostream. We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream.

Opening a File

A file must be opened before you can read from it or write to it.
Either the **ofstream** or **fstream** object may be used to open a file for writing.
Let's open a file called **"test.txt"** and write some content to it:

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

int main() {
  ofstream MyFile;
  MyFile.open("test.txt");

  MyFile << "Some text. \n";
}
```

The above code creates an **ofstream** object called MyFile, and uses the **open()** function to open the "test.txt" file on the file system. As you can see, the same stream output operator is used to write into the file.

> If the specified file does not exist, the **open** function will create it automatically.

Closing a File

When you've finished working with a file, close it using the member function **close()**.

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

int main() {
  ofstream MyFile;
  MyFile.open("test.txt");

  MyFile << "Some text! \n";
  MyFile.close();
}
```

Running this code will cause a "test.txt" file to be created in the directory of your project with "Some text!" written in it.

> You also have the option of specifying a path for your file in the **open** function, since it can be in a location other than that of your project.

Working with Files

You can also provide the path to your file using the **ofstream** objects constructor, instead of calling the **open** function.

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

int main() {
  ofstream MyFile("test.txt");

  MyFile << "This is awesome! \n";
  MyFile.close();
}
```

> As with the **open** function, you can provide a full path to your file located in a different directory.

Working with Files

Under certain circumstances, such as when you don't have file permissions, the **open** function can fail.
The **is_open()** member function checks whether the file is open and ready to be accessed.

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

```
int main() {
 ofstream MyFile("test.txt");

 if (MyFile.is_open()) {
  MyFile << "This is awesome! \n";
 }
 else {
  cout << "Something went wrong";
 }
 MyFile.close();
}
```

The is_open() member function checks whether the file is open and ready to be accessed.

File Opening Modes

An optional second parameter of the **open** function defines the **mode** in which the file is opened. This list shows the supported modes.

Mode Parameter	Meaning
ios::**app**	append to end of file
ios::**ate**	go to end of file on opening
ios::**binary**	file open in binary mode
ios::**in**	open file for reading only
ios::**out**	open file for writing only
ios::**trunc**	delete the contents of the file if it exists

All these flags can be combined using the bitwise operator OR (|).
For example, to open a file in write mode and truncate it, in case it already exists, use the following syntax:

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Reading from a File

You can read information from a file using an **ifstream** or **fstream** object.

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

int main () {
  string line;
  ifstream MyFile("test.txt");
  while ( getline (MyFile, line) ) {
   cout << line << '\n';
  }
  MyFile.close();
}
```

Try It Yourself

The **getline** function reads characters from an input stream and places them into a string.

> The example above reads a text file and prints the contents to the screen.
> Our **while** loop uses the **getline** function to read the file line by line.

End.