Object Oriented Programming (2019 pat)

Unit 5: Exception Handling and Templates

What is exception

Conditions that arise

- Infrequently and Unexpectedly
- Generally betray a Program Error
- Require a considered Programmatic Response
- Run-time Abnormalities

Leading to

- Crippling the Program
- May pull the entire System down

Causes of Exception

- Unexpected Systems State
- Logical Errors
- Run time Errors
- Undefined Operation

Exception Handling

- Runtime errors are termed as **exception**.
- **Exception handling** is the process to manage the runtime errors by converting the abnormal termination of a program to normal termination of a program.

- Exception Handling is a mechanism that separates the detection and handling of circumstantial **Exception Flow** from **Normal Flow.**

Types of Exception

Asynchronous Exception:

- Exceptions that come Unexpectedly, which are beyond the program’s control. Eg. an Interrupt in a Program, Disc failure etc
- Takes control away from the Executing Thread context to a context that is different from that which caused the exception.

Synchronous Exception:

- Planned Exceptions
- Handled in an organized manner.
- The most common type of Synchronous Exception is implemented as a throw.

Exception Stages

- Error Incidences
- Create Object and Raise Exception
- Detect Exception
- Handle Exception
- Recover from Exception

Introduction to Exception

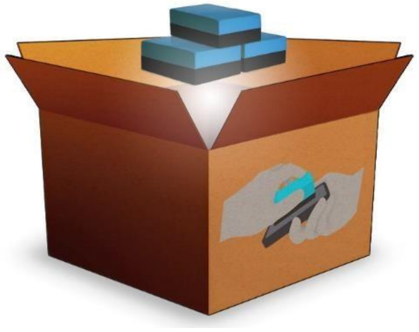
```
int main()
{
    int a,b,c;
    cout<<"Enter  value  a=";
    cin>>a;
    cout<<"Enter  value  b=";
    cin>>b;
    c=a/b;
    cout<<"answer="<<c;
}
```

```
Output:
Enter value a=5
Enter value b=2
answer=2
```

```
Output:
Enter value a=5
Enter value b=0
Abnormal Termination occur
```

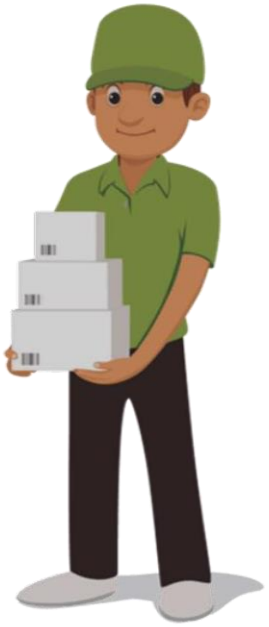
try, throw and catch

try



throw

catch



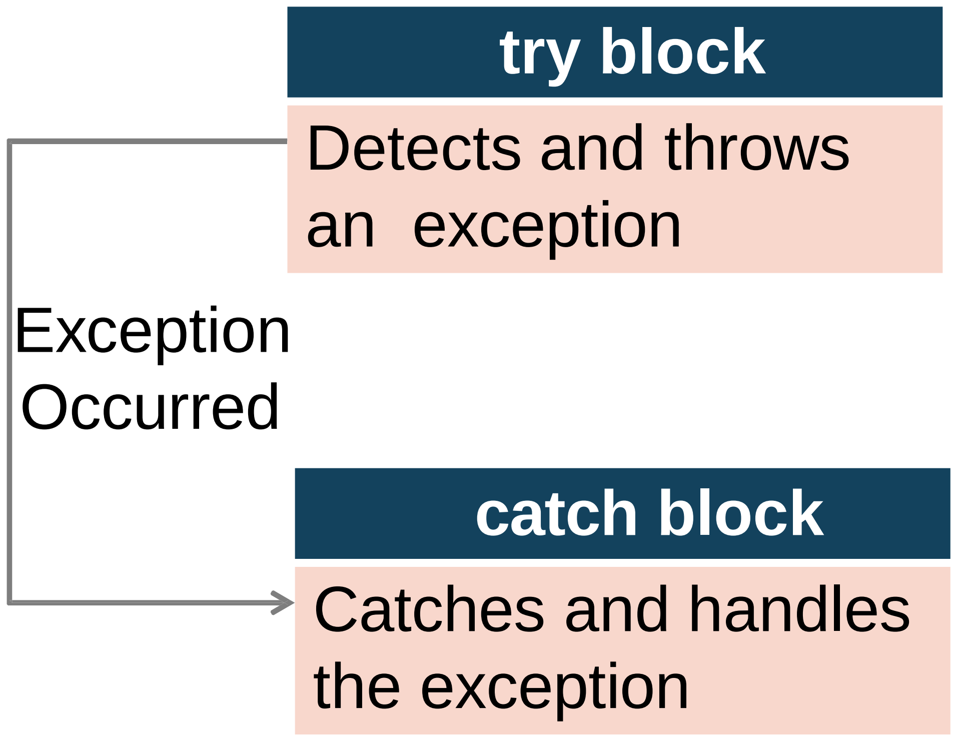
keywords: try, catch, and throw.

- **try** − A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.
- **throw** − A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.

- Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords.
- A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code,

try, throw and catch

- C++ exception handling mechanism is built upon three keywords **try**, **throw** and **catch**.


try block
Detects and throws an exception
Exception Occurred
catch block
Catches and handles the exception

```
try
{
   .....
   throw exception; //this block
   detects and throws an exception
}
```

```
catch(type arg)
{
   .....
   ...  //exception handling block
}
.....
```

try, throw and catch example 1

```
int main()
{
    int a,b,c;
    cout<<"Enter two values=";
    cin>>a>>b;
    try
    {
        if(b!=0)
                c=a/b;
                cout<<"answer="<<c;
        else
                throw(b);
    }
    catch(int x)
    {
        cout<<"Exception caught: Divide by zero\n";
    }
}
```

```
Output:
Enter value a=5
Enter value b=0
Exception caught: Divide by zero
```

try, throw and catch example 2

```
try
{
  int age = 15;
  if (age > 18) {
    cout << "Access granted - you are old enough.";
  }
  else {
    throw (age);
  }
}
catch (int num)
{
  cout << "Access denied - You must be at least 18
years old.\n";
  cout << "Age is: " << num;
}
```

Multiple catch example

```
#include<iostream>
using namespace std;
void test(int x){
    try
    {
        if(x==1)
            throw x;
        else if(x==0)
            throw 'x';
        else if(x==-1)
            throw 5.14;
    }
    catch(int i){
        cout<<"\nCaught an integer";
    }
    catch(char ch){
        cout<<"\nCaught a character";
    }
    catch(double i){
        cout<<"\nCaught a double";
    }
}
```

```
int main()
{
    test(1);
    test(0);
    test(-1);
}
```

Output:

```
Caught an integer
Caught a character
Caught a double
```

Catch all Exceptions

catch all exception

- In some situations, we may not predict all possible types of exceptions and therefore may not be able to design independent catch handlers to catch them.
- Handle Any Type of Exceptions

Syntax:

```
catch(...)
{
        //statements for processing all exceptions
}
```

Catch all exception example

```
#include<iostream>
usingnamespacestd;
void test(int x)
{
    try
    {
        if(x==0) throw x;
        if(x==-1) throw 'a';
        if(x==1) throw 5.15;
    }
    catch(...)
    {
    cout<<"Caught an exception\n";
    }
}
```

```
int main()
{
    test(-1);
    test(0);
    test(1);
}
```

```
Output:
Caught  an  exception
Caught  an  exception
Caught an exception
```

Re-Throwing exception

- An exception is **thrown from the catch block** is known as the **re- throwing** exception.
- It can be simply invoked by **throw** without arguments.
- Rethrown exception will be caught by **newly defined catch statement**.

Re-Throwing exception

```
#include<iostream>
using  namespace  std;
void  divide(double  x,  double  y)
{  try
   {
    if(y==0)
      throw  y;
    else
      cout<<"Division="<<x/y;
   }
   catch(double)
   {
    cout<<"Exception inside
function\n";
    throw;  //rethrowing
   }
}
```

```
int  main()
{
   try
   {
    divide(10.5,2.0);
    divide(20.0,0.0);
   }
   catch(double)
   {
    cout<<"Exception inside
main function";
   }
}
```

```
Output:
Division=5.25
Exception inside function
Exception inside main function
```

Exceptions thrown from functions

Exceptions thrown from functions

```
#include <iostream>
using namespace std;
void test(int x)
{
 cout<<"Inside function:"<<x<<endl;
 if(x) throw x;
}
int main()
{
 cout<<" Start" <<endl ;
  try
  {
   test(0);
   test(1);
   test(2);
  }
  catch(int x)
  {
   cout<<"Caught an int exception:"<< x<<endl;
  }
}
```

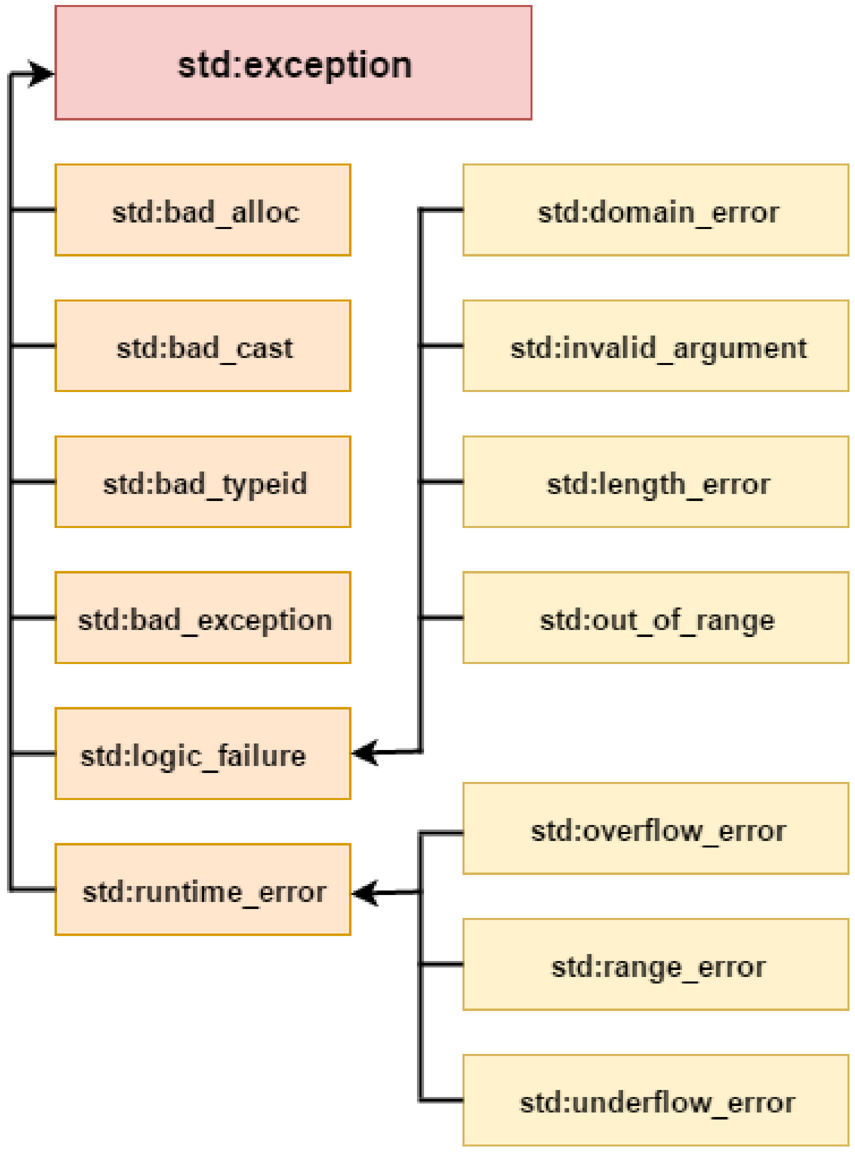
Output:

```
Start
Inside function: 0
Inside function:1
Caught an int exception:1
```

User defined Exception

- There maybe situations where you want to generate some **user specific exceptions** which are not pre-defined in C++.
- In such casesC++provided the mechanism to create our own exceptions by inheriting the exception class in C++.

Exception handling in C++



In C++, standard exceptions are defined in <exception> class that we can use inside our programs.

The arrangement of parent-child class hierarchy is shown in fig:

Exception	**Description**
std::exception	It is an exception and parent class of all standard C++ exceptions.
std::logic_failure	It is an exception that can be detected by reading a code.
std::runtime_error	It is an exception that cannot be detected by reading a code.
std::bad_exception	It is used to handle the unexpected exceptions in a c++ program.
std::bad_cast	This exception is generally be thrown by **dynamic_cast.**
std::bad_typeid	This exception is generally be thrown by **typeid.**
std::bad_alloc	This exception is generally be thrown by **new.**

std::domain_error This is an exception thrown when a mathematically invalid domain is used.
std::invalid_argument This is thrown due to invalid arguments.
std::length_error This is thrown when a too big std::string is created.
std::out_of_range This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[]().
std::runtime_error An exception that theoretically cannot be detected by reading the code.
std::overflow_error This is thrown if a mathematical overflow occurs.
std::range_error This is occurred when you try to store a value which is out of range.
std::underflow_error This is thrown if a mathematical underflow occurs.

User defined Exception

You can define your own exceptions by inheriting and overriding **exception** class functionality.

Let's see the simple example of user-defined exception in which **std::exception** class is used to define the exception.

Here, **what()** is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

User defined Exception

```
#include <iostream>
#include <exception>
using namespace std;
class myexception: public exception
{
    public:
    const char* what() const throw()
  {
     return "My exception happened";
  }
}  myex; //myexception object declared
int main (){
  try
  {
     throw myex;
  }
  catch (exception& e)
  {
     cout << e.what() << '\n';
  }
}
```

Output
My exception happened

Program to implement exception handling with single class -User defined Exception

We can use Exception handling with class too. Even we can throw an exception of user defined class types.

Program to implement exception handling with single class -User defined Exception

```
#include <iostream>
using namespace std;
class demo {
...
};
int main()
{
        try {
                throw demo();
        }
        catch (demo d) {
        cout << "Caught exception of demo class \n";
        }
}
```

Output:

```
Caught exception of demo class
```

Program to implement exception handling with two class

```
#include <iostream>
using namespace std;
class demo1 {
…
};

class demo2 {
…
};
int main()
{
```

```
for (int i = 1; i <= 2; i++) {
    try {
                    if (i == 1)
                            throw demo1();
                    else if (i == 2)
                            throw demo2();
        }
    catch (demo1 d1) {
        cout << "Caught exception of demo1 class \n";
        }
     catch (demo2 d2) {
        cout << "Caught exception of demo2 class \n";
        }
 }
```

```
Output:
Caught exception of demo1 class
Caught exception of  demo2 class
```

Exception handling with inheritance

```
#include <iostream>
using namespace std;
class demo1 {
…
};
class demo2 : public demo1 {
…
};
```

```
int main()
{
 for (int i = 1; i <= 2; i++) {
  try {
            if (i == 1)
                        throw demo1();
            else if (i == 2)
                        throw demo2();
      }
  catch (demo1 d1) {
    cout << "Caught exception of demo1 class \n";
            }
  catch (demo2 d2) {
    cout << "Caught exception of demo2 class \n";
            }
 }//for ends
}//main ends
```

Output:

```
Caught exception of  demo1 class
Caught exception of  demo1 class
```

Exception handling with constructor

```
#include <iostream>
using namespace std;
class demo {
            int num;
public:
            demo(int x)
            {
            try {
            if (x == 0) // catch block would be called
            throw "Zero not allowed ";
            num = x;
            show();
            }
            catch (const char* exp) {
            cout << "Exception caught \n ";
            cout << exp << endl;
            }
            }
            void show()
            {
            cout << "Num = " << num << endl;
            }
};
```

```
int main()
{
// constructor will be called
demo(0);
cout << "Again creating object \n";
demo(1);
}
```

```
Output:

Exception caught
Zero not allowed
Again creating object
Num=1
```

Exception handling and destructor

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

class Test {
public:
 Test()  { cout << "Constructing an object of Test " << endl; }
 ~Test()  { cout << "Destructing an object of Test " << endl; }
};

int main() {
 try {
        Test t1;
        throw 10;
 }
 catch(int i) {
        cout << "Caught " << i << endl;
 }
}
```

Output:

```
 Constructing an object of Test
 Destructing an object of Test
 Caught 10
```

Note: When an exception is thrown, destructors of the objects (whose scope ends with the try block) is automatically called before the catch block gets exectuted.

Note about exceptions :

Use throw only to signal an error.

Use catch only to specify error handling actions when you know you can handle an error.

Do not use throw to indicate a coding error in usage of a function.

Do not use throw if you discover unexpected violation of an invariant of your component, use assert or other mechanism to terminate the program.

In particular, **do not** use exceptions for control flow.

Low-Cost Deterministic C++ Exceptions for Embedded Systems

James Renwick
University of Edinburgh
Edinburgh, UK
s1227500@sms.ed.ac.uk

Tom Spink
University of Edinburgh
Edinburgh, UK
tspink@inf.ed.ac.uk

Björn Franke
University of Edinburgh
Edinburgh, UK
bfranke@inf.ed.ac.uk

ABSTRACT

The C++ programming language offers a strong exception mechanism for error handling at the language level, improving code readability, safety, and maintainability. However, current C++ implementations are targeted at general-purpose systems, often sacrificing code size, memory usage, and resource determinism for the sake of performance. This makes C++ exceptions a particularly undesirable choice for embedded applications where code size and resource determinism are often paramount. Consequently, embedded coding guidelines either forbid the use of C++ exceptions, or embedded C++ tool chains omit exception handling altogether. In this paper, we develop a novel implementation of C++ exceptions that eliminates these issues, and enables their use for embedded systems. We combine existing stack unwinding techniques with a new approach to memory management and run-time type information (RTTI). In doing so we create a compliant C++ exception handling implementation, providing bounded runtime and memory usage, while reducing code size requirements by up to 82%, and incurring only a minimal runtime overhead for the common case of no exceptions.

CCS CONCEPTS

• **Software and its engineering → Error handling and recovery**; *Software performance*; *Language features*;

which their use is often associated. However, C++'s wide usage has not been without issues. For many of the different domains within industry the use of exceptions in C++, a core part of the language, has been disallowed. For example, the Google C++ Style Guide [7], the Joint Strike Fighter Air Vehicle C++ Coding Standards and the Mars Rover flight software [14] do not allow the use of exceptions at all, while MISRA C++ specifies detailed rules for their use [1].

While there are many advantages to the use of exceptions in C++, perhaps their greatest disadvantage lies in their implementation. The current exception implementations were designed to prioritise performance in the case where exceptions do not occur, motivated primarily by the idea that exceptions are exceptional, and thus should rarely happen. This prioritisation may suit applications for which errors are rare and execution time is paramount, but it comes at the cost of other factors – notably increased binary sizes, up to 15% in some cases [4], memory usage and a loss of determinism, in both execution time and memory, when exceptions do occur. These drawbacks make exceptions unsuitable for use in embedded systems, where binary size and determinism are often as, if not more, important than overall execution time [12, 18]. However, the C++ language, its standard library and many prominent 3rd party libraries, such as ZMQ [10] and Boost [2], rely on exceptions. As a result, embedded software developed in C++ generally disables exceptions [8], at the cost of reduced disciplined error handling

Contrasting exception handling code across languages: An experience report involving 50 open source projects

Benjamin Jakobus
OPUS Research Group, Informatics Department
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
bjakobus@inf.puc-rio.br

Alessandro Garcia
OPUS Research Group, Informatics Department
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
afgarcia@inf.puc-rio.br

Eiji Adachi Barbosa
OPUS Research Group, Informatics Department
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
ebarbosa@inf.puc-rio.br

Carlos José Pereira de Lucena
Informatics Department
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
lucena@inf.puc-rio.br

***Abstract*—Exception handling mechanisms have been introduced into programming languages in an effort to help deal with runtime irregularities. These mechanisms aim to improve code reliability by providing constructs for sectioning code into exception scopes (e.g. Java `try` blocks) and exception handlers (e.g. Java `catch` blocks). Whilst exception handling mechanisms have been the focus of much research over the past years, empirical studies have only focused on characterising *exception handling* code of Java and C# programs. There exists little empirical evidence on how exception handling mechanisms are**

deal with runtime irregularities in a reliable manner. Specifically, the objective of EHM constructs is to facilitate the development of reliable systems by allowing developers to deal with runtime irregularities that would otherwise result in erroneous program behaviour if left ignored. The EHM constructs themselves allow developers to reason with these irregularities in a structured, well-defined manner.

EHM constructs are typically used to define two parts of the exceptional behaviour of a program: *exception scopes* and

A Study on the Effects of Exception Usage in Open-Source C++ Systems

Kirsten Bradley and Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON Canada
{kcpbradl,migod}@uwaterloo.ca

***Abstract*—Exception handling (EH) is a feature common to many modern programming languages, including C++, Java, and Python, that allows error handling in client code to be performed in a way that is both systematic and largely detached from the implementation of the main functionality. However, C++ developers sometimes choose not to use EH, as they feel that its use increases complexity of the resulting code; new control flow paths are added to the code, "stack unwinding" adds extra responsibilities for the developer to worry about, and EH arguably detracts from the modular design of the system. In this paper, we perform an exploratory empirical study on how exceptions are handled in 2721 open source C++ systems taken from GitHub. We observed that the number of edges in a call graph grows, on average, by 22% when edges for exception flow are added to a graph. Additionally, about 8 out of 9 functions that may throw an exception do not initiate that exception. These results suggest that, in practice, the use of C++ EH can add subtle complexity to the design of the system that developers must strive to be aware of.**

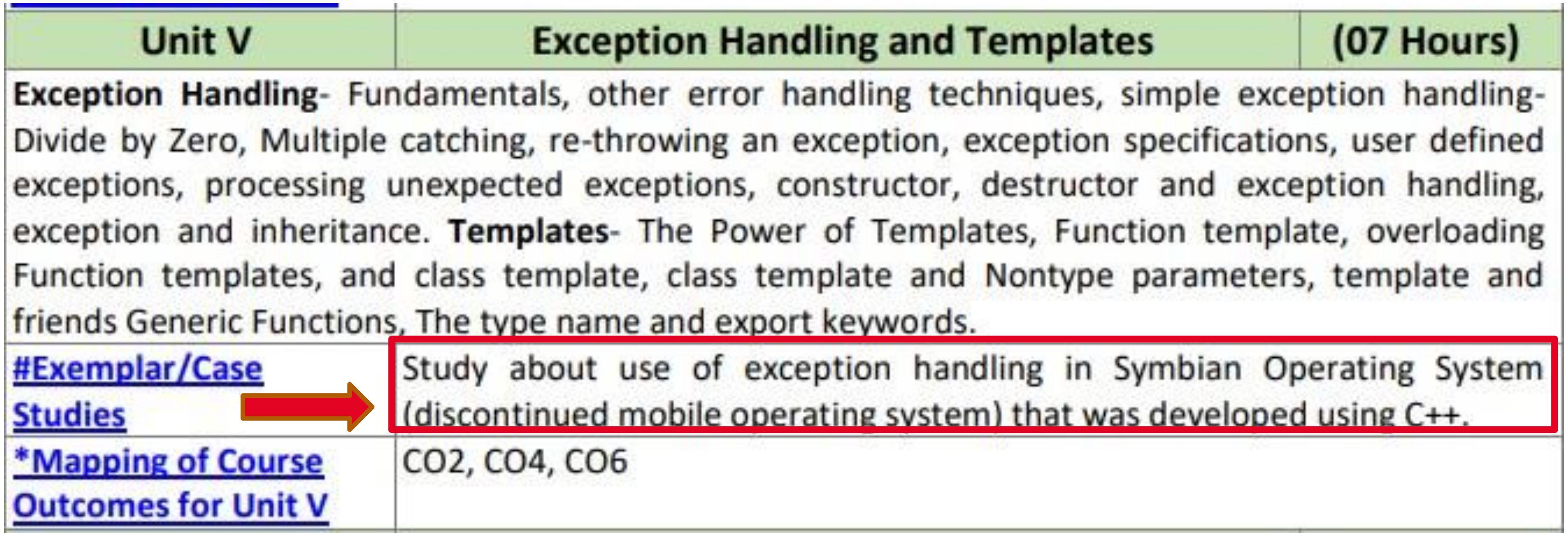
***Index Terms*—exceptions, static analysis, C++, exception flow**

I. Introduction

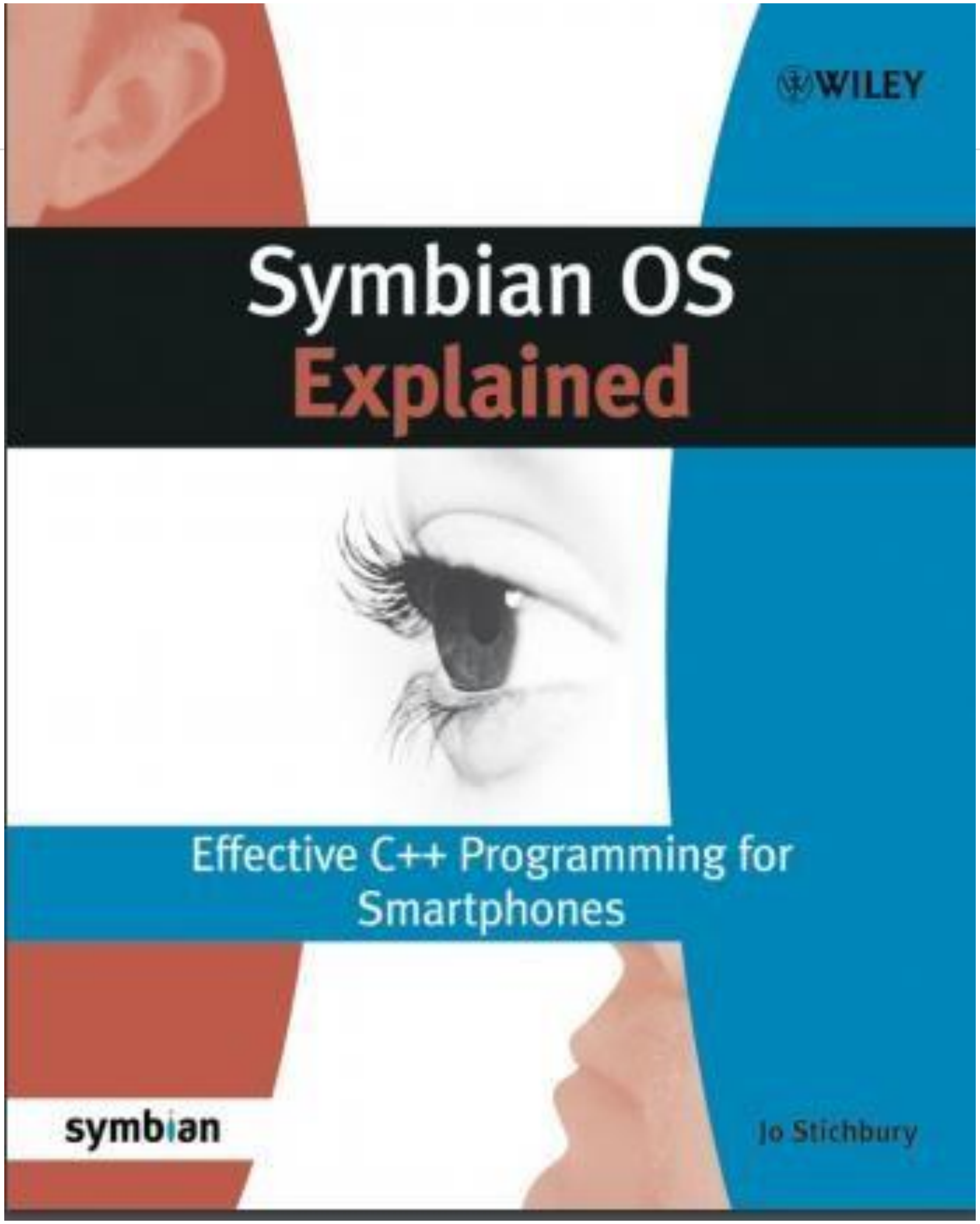
tion use such as improving error handling across modules [1]. While there may be some inherent difficulty with exception code, this difficulty could come from the tasks for which exceptions are used [20].

Most previous exception studies have looked mainly at Java code, so there is a dearth of empirical results about C++ code. While EH is similar Java and C++, there are also important differences, such as whether an exception specification is used, the need to consider stack unwinding in C++, and the ability of C++ code to throw objects other than explicit exceptions. For these reasons, we chose to only study C++ code. This meant that we had to develop a tool to extract EH information from source code; a secondary contribution of our work is the development of such a tool, which we will release as open source.

This paper has two main contributions. First, we present an exploratory empirical study on understanding EH usage in real-world C++ code taken from GitHub, and we evaluate the potential impact of EH usage that may not be immediately



Unit V	Exception Handling and Templates	(07 Hours)
Exception Handling- Fundamentals, other error handling techniques, simple exception handling- Divide by Zero, Multiple catching, re-throwing an exception, exception specifications, user defined exceptions, processing unexpected exceptions, constructor, destructor and exception handling, exception and inheritance. **Templates**- The Power of Templates, Function template, overloading Function templates, and class template, class template and Nontype parameters, template and friends Generic Functions, The type name and export keywords.		
#Exemplar/Case Studies	Study about use of exception handling in Symbian Operating System (discontinued mobile operating system) that was developed using C++.	
*Mapping of Course Outcomes for Unit V	CO2, CO4, CO6	


WILEY

Symbian OS Explained

Effective C++ Programming for Smartphones

symbian

Jo Stichbury

Exception Handling in Symbian OS

Limited memory and resources.

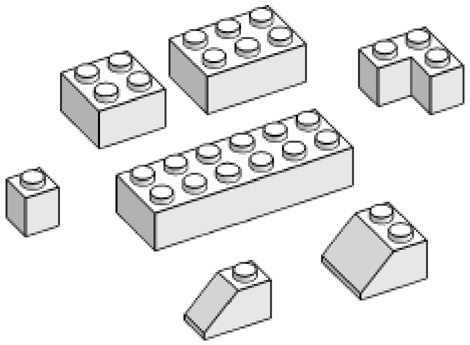
Leave/Trap framework and has cleanup

stack: Leave() is analogous to c++ throw

Trap & trapD are analogous to a combination of try and catch in c++

Clean up stack

Template



Templates

- C++ **templates** are a powerful mechanism for **code reuse**, as they enable the programmer to write code that behaves the same for any data type.
- By **template** we can define **generic classes and functions**.
- In simple terms, you can create a single function or a class to work with different data types using **templates**.
- It can be considered as a kind of macro. When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type.

Need of Templates

```
int add(int x, int y)
{
  return x+y;
}
```

```
float add(float x, float y)
{
   return x+y;
}
```

```
char add(char x,char y)
{
       return x+y;
}
```

```
double add(double x, double y)
{
   return x+y;
}
```

We need a single function that will work for int, float, double etc…

Template for add()

```
template <typename  T1 >
T1 ADD (T1 x, T1 y)
{
        return x+y;
}
```

e.g. function Calls:

```
Ans  =  add <int> (5, 6);
Ans  =  add <float> (11.5, 33.6);
```

Boild Water

Add Milk

Add Sugar

Add Bru Coffee

BruCoffee

Boil Water

Add Milk

Add Sugar

Add Nescafe Coffee

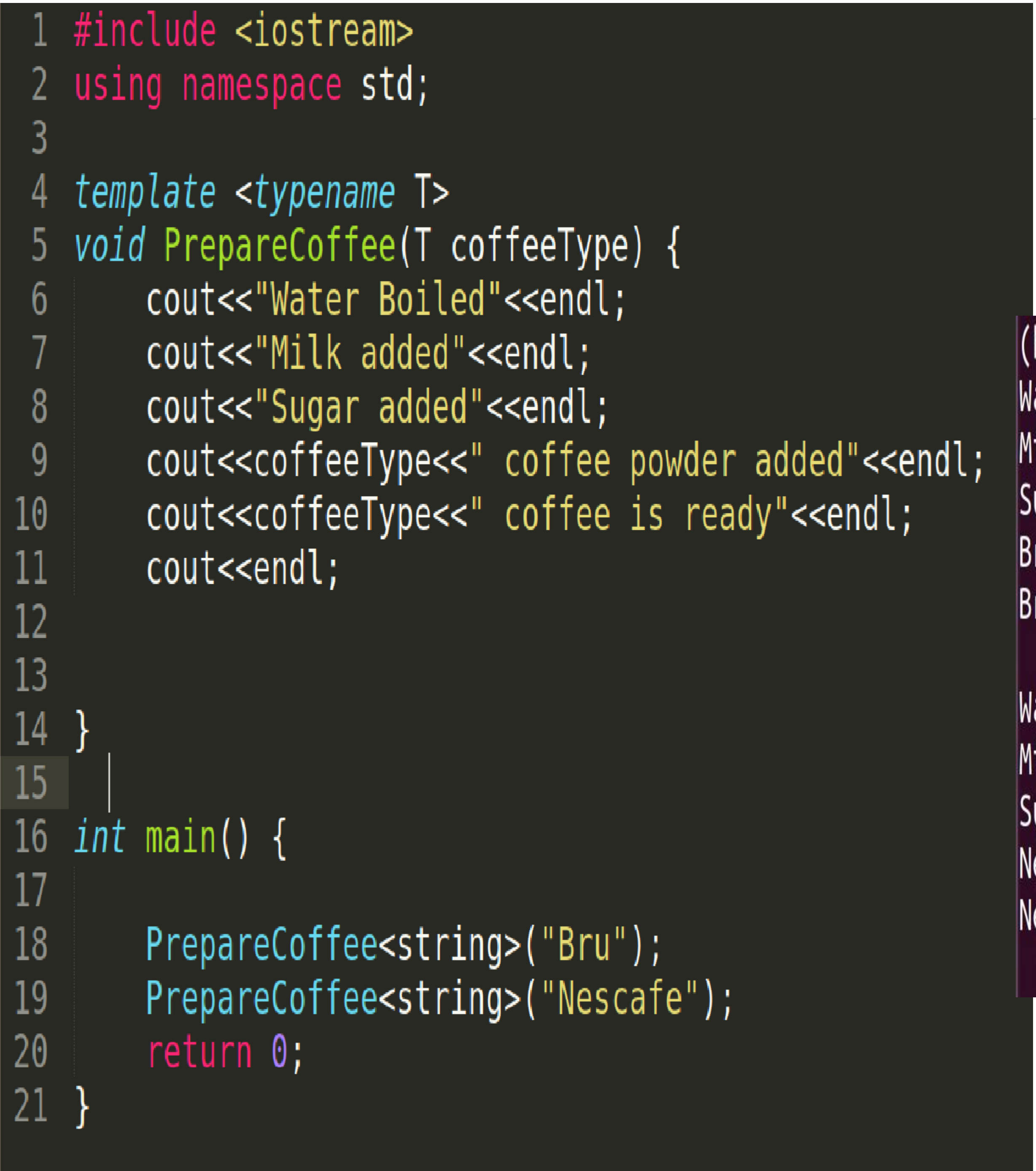
Nescafe Coffee

Template Method PrepareCoffee

1. Boild Water
2. Add Milk
3. Add Sugar
4. Add Coffee Powder

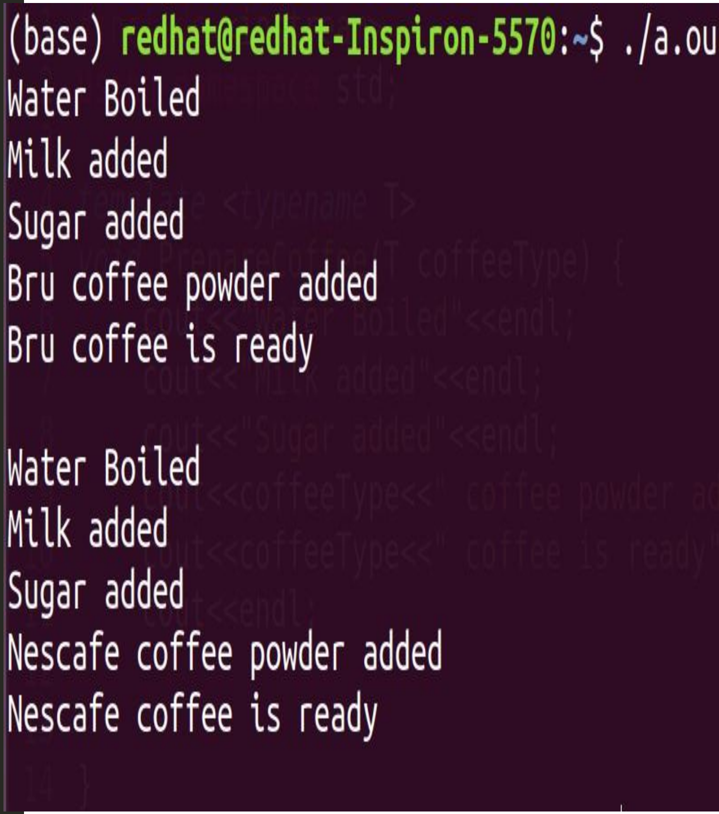
Bru Coffee

Nescafe Coffee



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

template <typename T>
void PrepareCoffee(T coffeeType) {
    cout<<"Water Boiled"<<endl;
    cout<<"Milk added"<<endl;
    cout<<"Sugar added"<<endl;
    cout<<coffeeType<<" coffee powder added"<<endl;
    cout<<coffeeType<<" coffee is ready"<<endl;
    cout<<endl;


}

int main() {

    PrepareCoffee<string>("Bru");
    PrepareCoffee<string>("Nescafe");
    return 0;
}
```

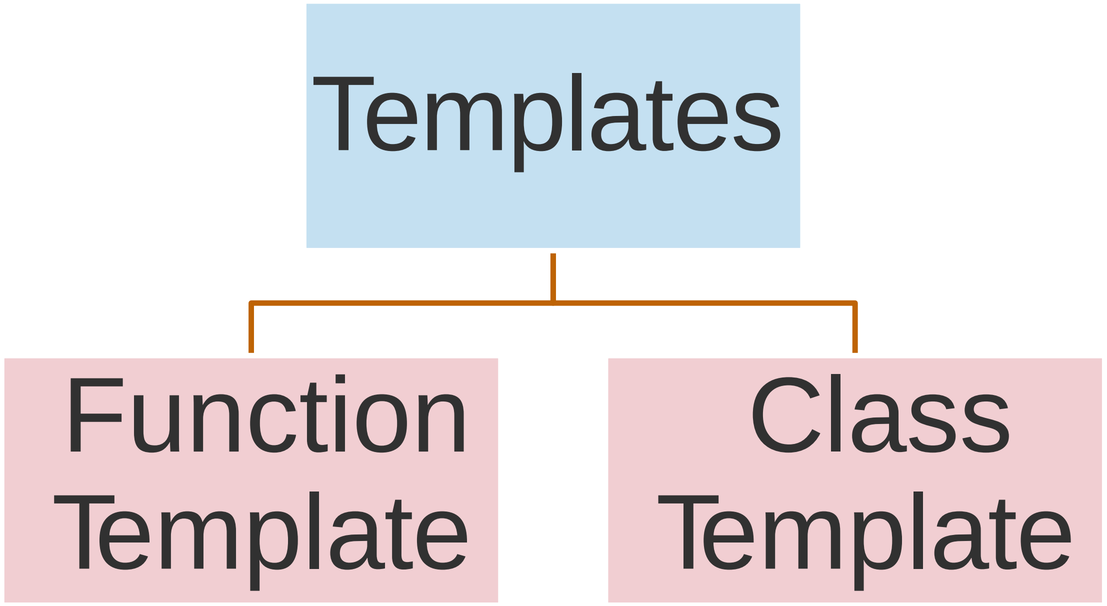
Output



```
(base) redhat@redhat-Inspiron-5570:~$ ./a.ou
Water Boiled
Milk added
Sugar added
Bru coffee powder added
Bru coffee is ready

Water Boiled
Milk added
Sugar added
Nescafe coffee powder added
Nescafe coffee is ready
```

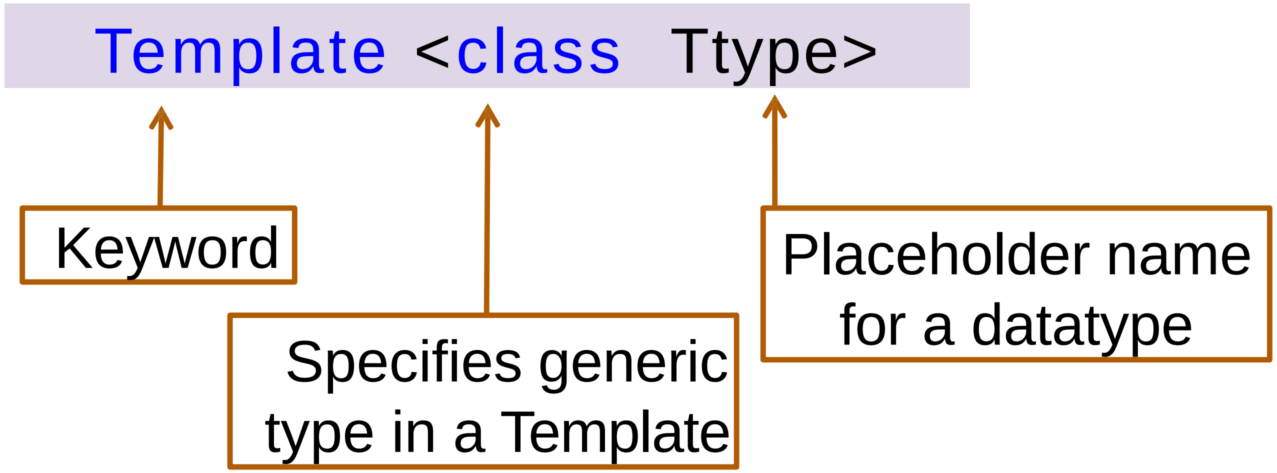
Templates

- Templates concept enables us to define **generic classes** and **functions**.
- This allows a function or class to work on many different data types without being rewritten for each one.



Function Template

Syntax:



```
Template <class  Ttype>
```

Keyword
Specifies generic type in a Template
Placeholder name for a datatype

```
Template <typename Ttype>
```

Function Template

- Suppose you write a function printData to print int value:

```
void printData(int value) {
        cout<<"The value is "<<value;
}
```

- Now, if you want to print double values or string values, then you have to overload the function:

```
void printData(float value) {
        cout<<"The value is "<<value;
}

void printData(char * value) {
        cout<<"The value is "<<*value;
}
```

- To perform same operation with different data type, we have to write same code multiple time.

Function Template (Cont…)

- C++ provides templates to reduce this type of duplication of code.

```
template <typename    T>
void printData(T value)
{
        cout<<"The value is "<<value;
}
```

- We can now use printData for any data type. Here **T** is a template parameter that identifies a type.
- Then, anywherein the function where **T** appears, it is replaced with whatever type the function is instantiated.

```
int i=3;
float d=4.75;
char *s="hello";
printData(i); //    T  is int
printData(d); //    T  is float
printData(s); //    T  is string
```

```
#include <iostream>
using namespace std;
template <typename T>

T Large(T n1, T n2)
{
 return (n1 > n2) ? n1 : n2;
}

int main() {
   int i1, i2; float f1, f2; char c1, c2;
   cout << "Enter two integers:\n";
   cin >> i1 >> i2;
   cout <<Large(i1, i2)<<" is larger." << endl;
   cout <<"\nEnter two floating-point numbers:\n";
   cin >> f1 >> f2;
   cout << Large(f1, f2)<<" is larger." << endl;
   cout << "\nEnter two characters:\n";
   cin >> c1 >> c2;
   cout << Large(c1, c2) << " has larger ASCII value.";
}
```

- T is a **template** argument that accepts different data types.
- **typename** is a keyword.
- You can also use keyword **class** instead of **typename**

Class Template

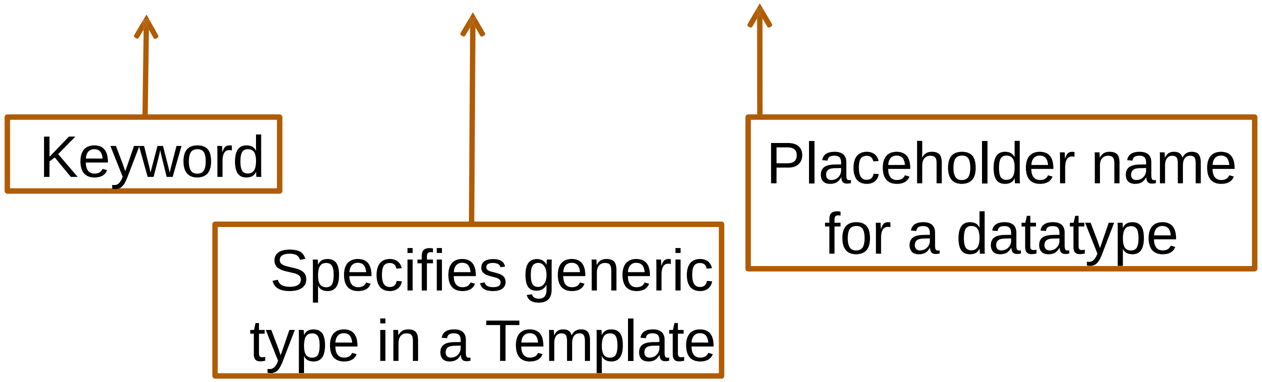
- Sometimes, you need a class implementation logic that is same for multiple classes, only the data types used are different.

- Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.

Class Template

Syntax:

```
template<class  Ttype>
```


Keyword
Specifies generic type in a Template
Placeholder name for a datatype

Object of template class

The object of template class are created as follows

classname <datatype> objectname;

```
template<class Ttype>
class sample
{
   Ttype a,b;
   public:
      void getdata()
      {
        cin>>a>>b;
      }
      void sum();
};
```

```
int main()
{
  sample <int>s1;
  sample <float>s2;
  s1.getdata();
  s1.sum();
  s2.getdata();
  s2.sum();
}
```

Class Template Example

```
template<class T1, class T2>
class Sample
{
    T1 num1;  T2 num2;
    public:
    Sample(T1  x,T2  y){
        num1 = x;
        num2 = y;
    }
    void disp(){
        cout<<"\na="<<a<<"\tb="<<b;
    }
};
int main(){
 Sample <int,float> S1(12,23.3);
 Sample <char, int> S2('N',12);
 S1.disp();
 S2.disp();
}
```

- To create a class template object, define the data type inside **< >** at the time of object creation.
- className<int> classObj;
  className<float> classObj;

Further readings

A Semantic Analysis of C++ Templates *

Jeremy Siek and Walid Taha
Jeremy.G.Siek@rice.edu, taha@rice.edu

Rice University,
Houston, TX 77005, USA

Abstract. Templates are a powerful but poorly understood feature of the C++ language. Their *syntax* resembles the parameterized classes of other languages (e.g., of Java). But because C++ supports template specialization, their *semantics* is quite different from that of parameterized classes. Template specialization provides a Turing-complete sub-language within C++ that executes at compile-time. Programmers put this power to many uses. For example, templates are a popular tool for writing program generators.

The C++ Standard defines the semantics of templates using natural language, so it is prone to misinterpretation. The meta-theoretic properties of C++ templates have not been studied, so the semantics of templates has not been systematically checked for errors. In this paper we present the first formal account of C++ templates including some of the more complex aspects, such as template partial specialization. We validate our semantics by proving type safety and verify the proof with the Isabelle proof assistant. Our formalization reveals two interesting issues in the C++ Standard: the first is a problem with member instantiation and the second concerns the generation of unnecessary template specializations.

1 Introduction

We start with a review of C++ templates, demonstrating their use with some

How C++ Templates Are Used for Generic Programming: An Empirical Study on 50 Open Source Systems

LIN CHEN, Nanjing University, China
DI WU, Momenta, China
WANWANGYING MA, YUMING ZHOU, and BAOWEN XU, Nanjing University, China
HARETON LEUNG, Hong Kong Polytechnic University, China

Generic programming is a key paradigm for developing reusable software components. The inherent support for generic constructs is therefore important in programming languages. As for C++, the generic construct, templates, has been supported since the language was first released. However, little is currently known about how C++ templates are actually used in developing real software. In this study, we conduct an experiment to investigate the use of templates in practice. We analyze 1,267 historical revisions of 50 open source systems, consisting of 566 million lines of C++ code, to collect the data of the practical use of templates. We perform statistical analyses on the collected data and produce many interesting results. We uncover the following important findings: (1) templates are practically used to prevent code duplication, but this benefit is largely confined to a few highly used templates; (2) function templates do not effectively replace C-style generics, and developers with a C background do not show significant preference between the two language constructs; (3) developers seldom convert dynamic polymorphism to static polymorphism by using CRTP (Curiously Recursive Template Pattern); (4) the use of templates follows a power-law distribution in most cases, and C++ developers who prefer using templates are those without other language background; (5) C developer background seems to override C++ project guidelines. These findings are helpful not only for researchers to understand the tendency of template use but also for tool builders to implement better tools to support generic programming.

CCS Concepts: • **Software and its engineering → Object oriented languages; Multiparadigm languages;**