Introduction to C++ Templates and Exceptions

C++ Function Templates

C++ Class Templates

Exception and Exception Handler

C++ Function Templates

Approaches for functions that implement identical tasks for different data types

- **Naïve Approach**
- **Function Overloading**
- **Function Template**

Instantiating a Function Templates

Approach 1: Naïve Approach

create unique functions with unique names for each combination of data types

difficult to keeping track of multiple function names

lead to programming errors

Example

```
void PrintInt( int n )
{
    cout << "***Debug" << endl;
    cout << "Value is " << n << endl;
}
void PrintChar( char ch )
{
    cout << "***Debug" << endl;
    cout << "Value is " << ch << endl;
}
void PrintFloat( float x )
{
  …
}
void PrintDouble( double d )
{
  …
}
```

To output the traced values, we insert:

```
PrintInt(sum);

PrintChar(initial);

PrintFloat(angle);
```

Approach 2:Function Overloading (Review)

- **The use of the same name for different C++ functions, distinguished from each other by their parameter lists**
  - **Eliminates need to come up with many different names for identical tasks.**
  - **Reduces the chance of unexpected results caused by using the wrong function name.**

Example of Function Overloading

```
void Print( int n )
{
    cout << "***Debug" << endl;
    cout << "Value is " << n << endl;
}
void Print( char ch )
{
    cout << "***Debug" << endl;
    cout << "Value is " << ch << endl;
}
void Print( float x )
{

}
```

To output the traced values, we insert:

```
Print(someInt);
Print(someChar);
Print(someFloat);
```

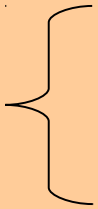
Approach 3: Function Template

- **A C++ language construct that allows the compiler to generate multiple versions of a function by allowing parameterized data types.**

FunctionTemplate

```
Template < TemplateParamList >
FunctionDefinition
```

TemplateParamDeclaration: placeholder



```
class  typeIdentifier
typename variableIdentifier
```

Example of a Function Template

```
template<class SomeType>

void Print( SomeType val )
{
     cout << "***Debug" << endl;
     cout << "Value is " << val << endl;
}
```

Template parameter

(class, user defined type, built-in types)

Template argument

To output the traced values, we insert:

```
Print<int>(sum);
Print<char>(initial);
Print<float>(angle);
```

Instantiating a Function Template

- **When the compiler instantiates a template, it substitutes the template argument for the template parameter throughout the function template.**

TemplateFunction Call

Function < TemplateArgList > (FunctionArgList)

Summary of Three Approaches

Naïve Approach
Different Function Definitions
Different Function Names

Function Overloading
Different Function Definitions
Same Function Name

Template Functions
One Function Definition (a function template)
Compiler Generates Individual Functions

Class Template

• A C++ language construct that allows the compiler to generate multiple versions of a class by allowing parameterized data types.

Class Template

```
Template < TemplateParamList >
ClassDefinition
```

TemplateParamDeclaration: placeholder

{ class typeIdentifier
typename variableIdentifier

Example of a Class Template

```
template<class ItemType>
class GList
{
 public:
    bool IsEmpty() const;
    bool IsFull() const;
    int  Length() const;
    void Insert( /* in */ ItemType item );
    void Delete( /* in */ ItemType item );
    bool IsPresent( /* in */ ItemType item ) const;
    void SelSort();
    void Print() const;
    GList();                          // Constructor
 private:
    int      length;
    ItemType data[MAX_LENGTH];
};
```

Template parameter

Instantiating a Class Template

- **Class template arguments *must* be explicit.**
- **The compiler generates distinct class types called template classes or generated classes.**
- **When instantiating a template, a compiler substitutes the template argument for the template parameter throughout the class template.**

Instantiating a Class Template

To create lists of different data types

```
// Client code

GList<int> list1;
GList<float> list2;
GList<string> list3;

list1.Insert(356);
list2.Insert(84.375);
list3.Insert("Muffler bolt");
```

template argument

Compiler generates 3 distinct class types

```
GList_int list1;
GList_float list2;
GList_string list3;
```

Substitution Example

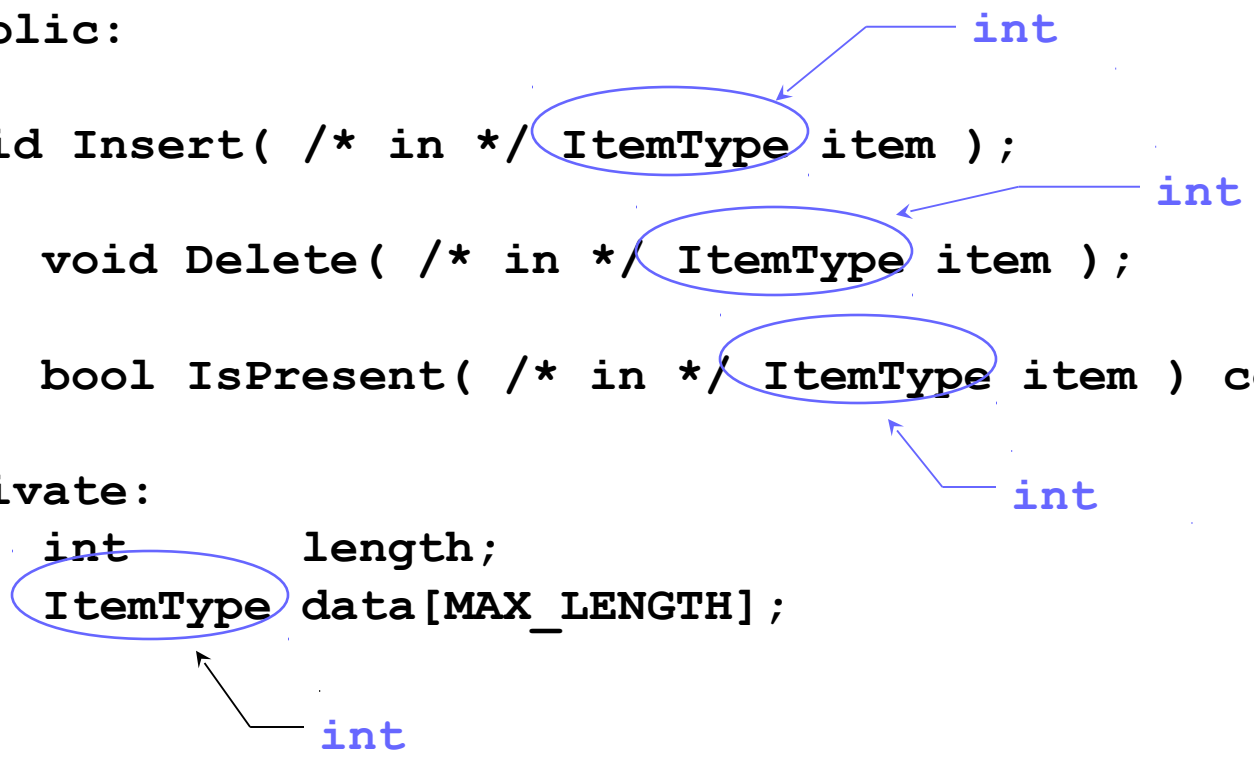
```
class GList_int
{
public:

void Insert( /* in */ ItemType item );

    void Delete( /* in */ ItemType item );

    bool IsPresent( /* in */ ItemType item ) const;

private:
    int      length;
    ItemType data[MAX_LENGTH];
};
```



Function Definitions for Members of a Template Class

```
template<class ItemType>
void GList<ItemType>::Insert( /* in */ ItemType item )
{
    data[length] = item;
    length++;
}
```

```
//after substitution of float
void GList<float>::Insert( /* in */ float item )
{
    data[length] = item;
    length++;
}
```

Another Template Example: passing two parameters

```
template <class T, int size>
  class Stack {...
  };

Stack<int,128> mystack;
```

non-type parameter

Exception

- An exception is a unusual, often unpredictable event, detectable by software or hardware, that requires special processing occurring at runtime
- In C++, a variable or class object that represents an exceptional event.

Handling Exception

- If without handling,
  - Program crashes
  - Falls into unknown state
- An exception handler is a section of program code that is designed to execute when a particular exception occurs
  - Resolve the exception
  - Lead to known state, such as exiting the program

Standard Exceptions

Exceptions Thrown by the Language

- `new`

Exceptions Thrown by Standard Library Routines

Exceptions Thrown by user code, using *throw* statement

The *throw* Statement

Throw: to signal the fact that an exception has occurred; also called *raise*

ThrowStatement

```
throw Expression
```

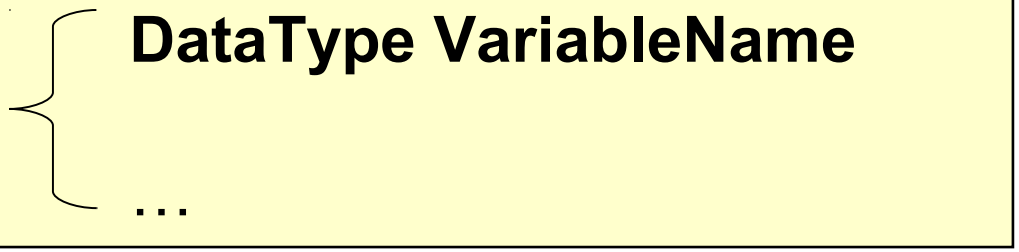
The `try-catch` Statement

How one part of the program catches and processes the exception that another part of the program throws.

TryCatchStatement

```
try
  Block
catch (FormalParameter)
  Block
catch (FormalParameter)
```

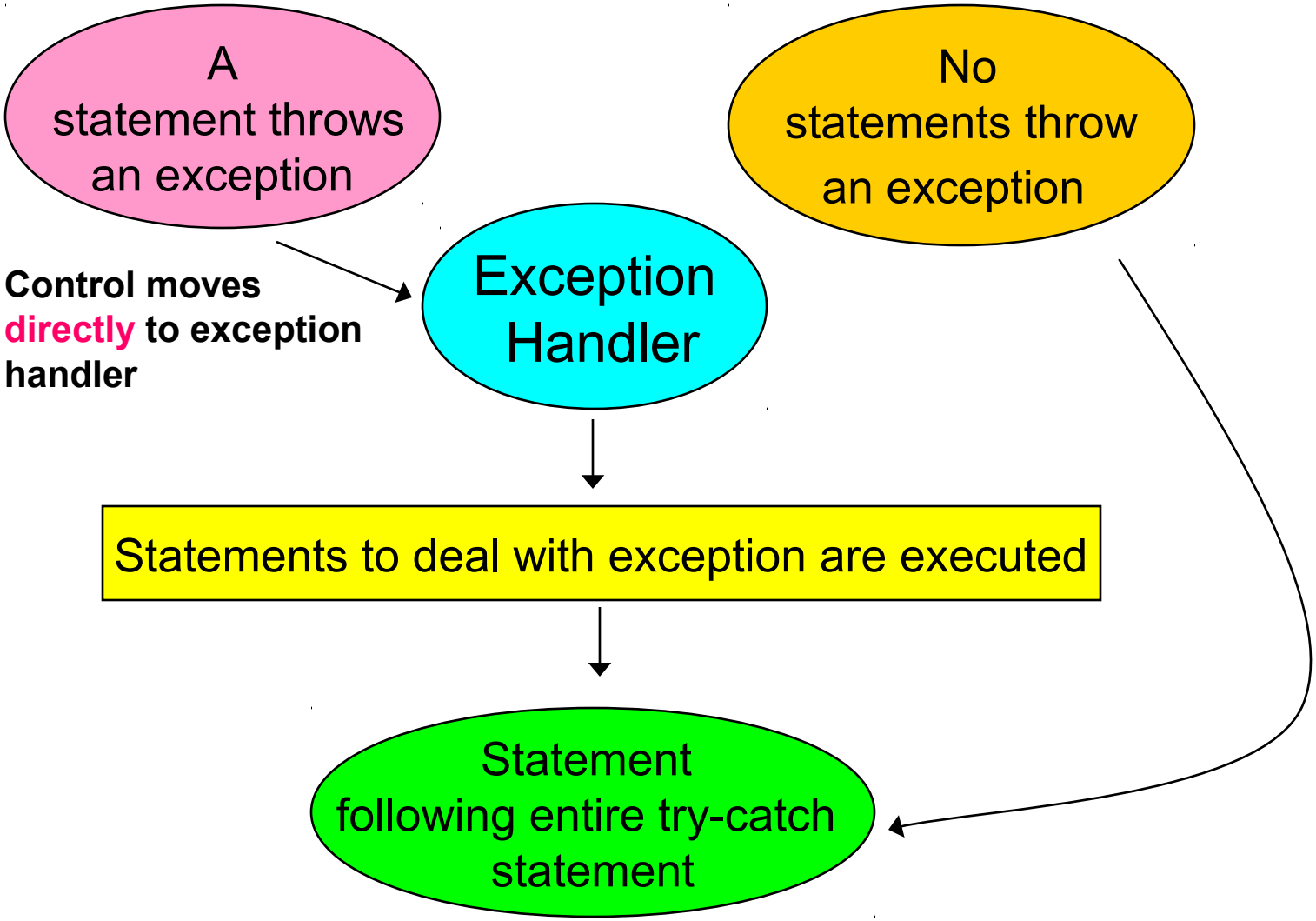
FormalParameter



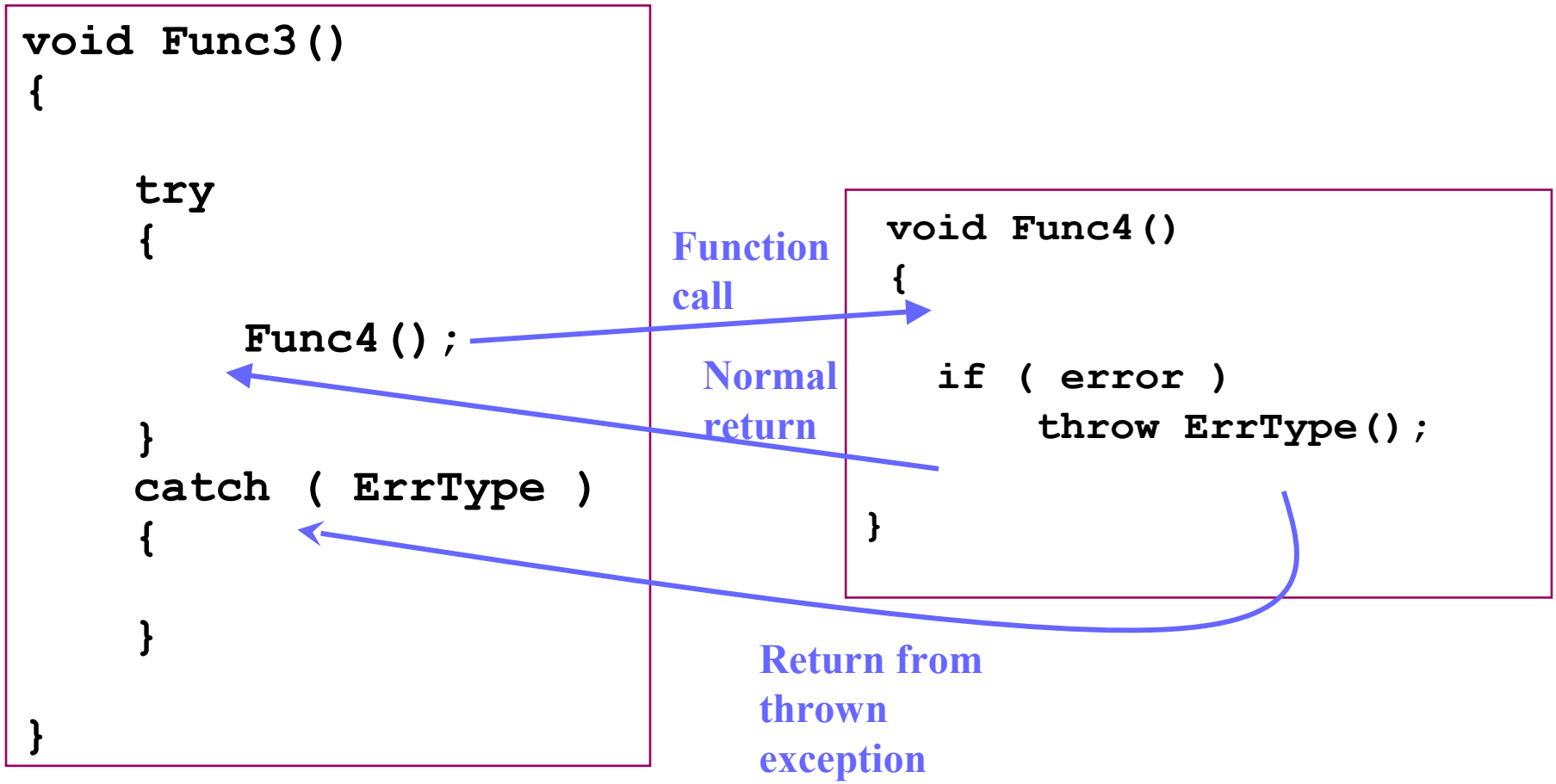
Example of a `try-catch` Statement

```
try
{
        // Statements that process personnel data and may throw
        // exceptions of type int, string, and SalaryError
}
catch ( int )
{
        // Statements to handle an int exception
}
catch ( string s )
{
       cout << s << endl;  // Prints "Invalid customer age"
        // More statements to handle an age error
}
catch ( SalaryError )
{
       // Statements to handle a salary error
}
```

Execution of `try-catch`



Throwing an Exception to be Caught by the Calling Code



Practice: Dividing by ZERO

Apply what you know:

```
int Quotient(int numer,     // The numerator
             int denom )    // The denominator
{
    if (denom != 0)
        return numer / denom;
    else
        //What to do?? do sth. to avoid program
        //crash
}
```

A Solution

```
int Quotient(int numer,     // The numerator
             int denom )    // The denominator
{
  if (denom == 0)
     throw DivByZero();
       //throw exception of class DivByZero
  return numer / denom;
}
```

A Solution

```
// quotient.cpp -- Quotient program

#include<iostream.h>
#include <string.h>
int Quotient( int, int );
class DivByZero {}; // Exception class
int main()
{
   int numer;   // Numerator
   int denom;   // Denominator

   //read in numerator
     and denominator
```

```
while(cin)
{
                                        try
                                          {
      cout << "Their quotient: "
           << Quotient(numer,denom) <<endl;
                                          }
    catch ( DivByZero )//exception handler
    {
     cout<<“Denominator can't be 0"<< endl;
    }
    // read in numerator and denominator
}
return 0;
}
```