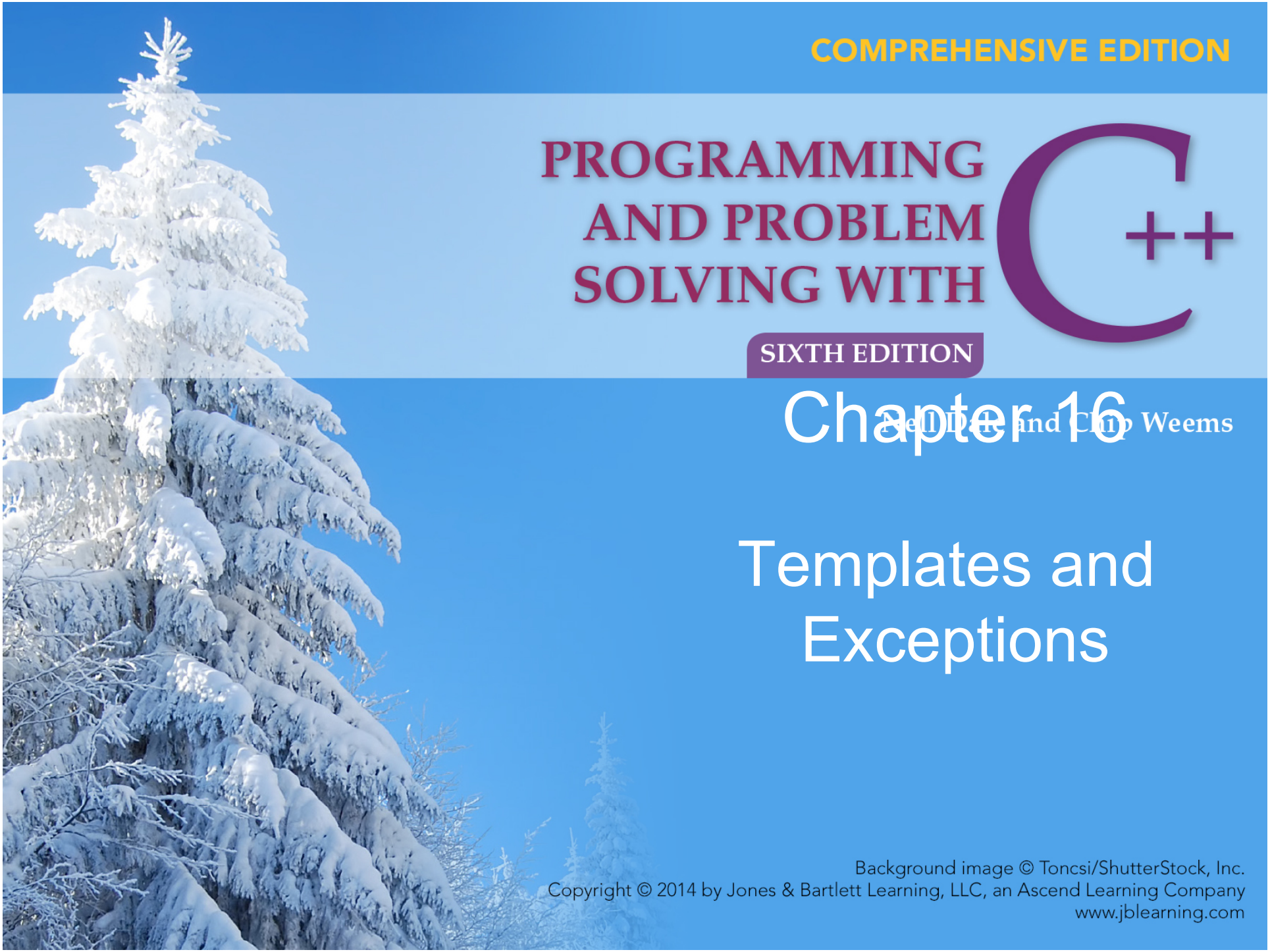


Chapter 16 Topics

- C++ Function Templates
- Instantiating a Function Templates
- User-defined Specializations
- C++ Class Templates

Chapter 16 Topics

- Instantiating Class Templates
- Function Definitions for Members of a Template Class
- Exception Classes, Throwing an Exception
- Exception Handlers

Generic Algorithms

- **Generic algorithms are algorithms in which the actions or steps are defined, but the data types of the items being manipulated are not**

Example of a Generic Algorithm

```
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:

```
sum = alpha + beta + gamma;
PrintInt(sum);

PrintChar(initial);

PrintFloat(angle);
```

Function Overloading

- **Function overloading** is the use of the same name for different functions, distinguished 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);
```

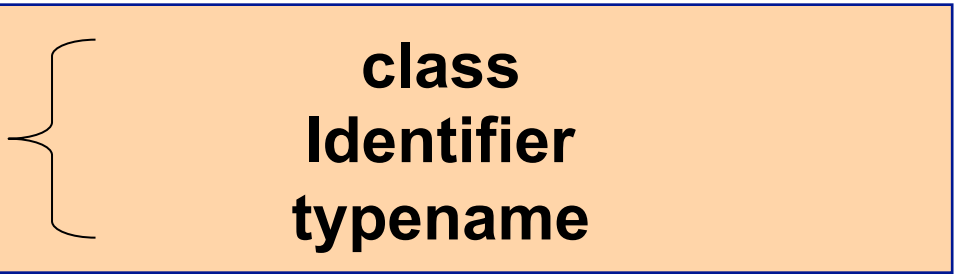
Function Template

- 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



Example of a Function Template

```
template<class SomeType>

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

Template parameter

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)
```

Generic Functions, Function Overloading, Template Functions

Generic Function
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

User-Defined Specializations

Example that demonstrates use of `template < >`

```
template<>
void Print(string     vName,    // Name of the variable
           StatusType val   )   // Value of the variable
{
    cout << "***Debug" << endl;
    cout << "Value of " << vName << " = ";
    switch (val)
```

Example, continued.

```
    {
        case OK            : cout << "OK";
                             break;
        case OUT_OF_STOCK  : cout << "OUT_OF_STOCK";
                             break;
        case BACK_ORDERED  : cout << "BACK_ORDERED";
                             break;
        default            : cout << "Invalid value";
    }
    cout << endl;
}
```

Organization of Program Code

Three possibilities:

1. **Template definitions at the beginning of program file, prior to `main` function**
2. **Function prototypes first, then the `main` function, then the template definitions**
3. **Template definition in the header file, use #include to insert that file into the program**

What is a Generic Data Type?

- **It is a type for which the operations are defined but the data types of the items being manipulated are not**

What is a Class Template?

- **It is a C++ language construct that allows the compiler to generate multiple versions of a class by allowing parameterized data types**

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 Reset() const;
    ItemType GetNextItem();
    GList();                          // Constructor
```

Template parameter

Example of a Class Template, cont. . .

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

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;
```

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

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];
};
```

int

int

int

int

Writing Function Templates

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

Writing Function Templates

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

Organization of Program Code

- A compiler must know the argument to the template in order to generate a function template, and this argument is located in the client code
- **Solutions**
  - Have specification file include implementation file
  - Combine specification file and implementation file into one file

Warning!

Are you using an IDE (integrated development environment) where the editor, compiler, and linker are bundled into one application?

Remember The compiler must know the template argument

How you organize the code in a project may differ depending on the IDE you are using

An Exception is…

An exception is:

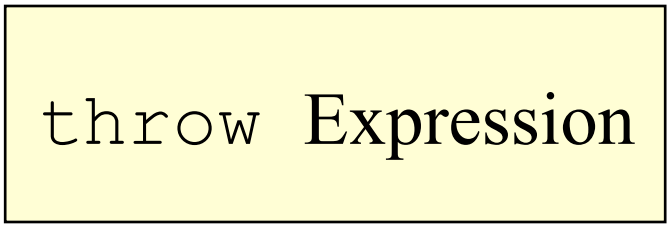
- **an unusual, often unpredictable event,**
- **detectable by software or hardware,**
- **requires special processing;**
- **also, in C++, a variable or class object that represents an exceptional event**

An exception handler is a section of program code that is executed when a particular exception occurs

The throw Statement

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

ThrowStatement



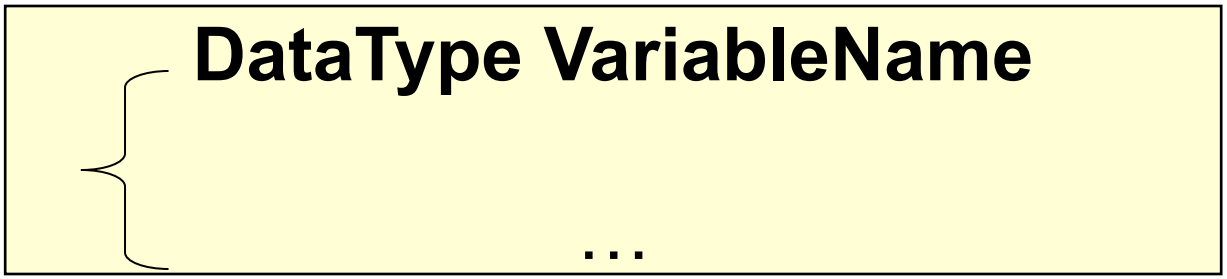
The `try-catch` Statement

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



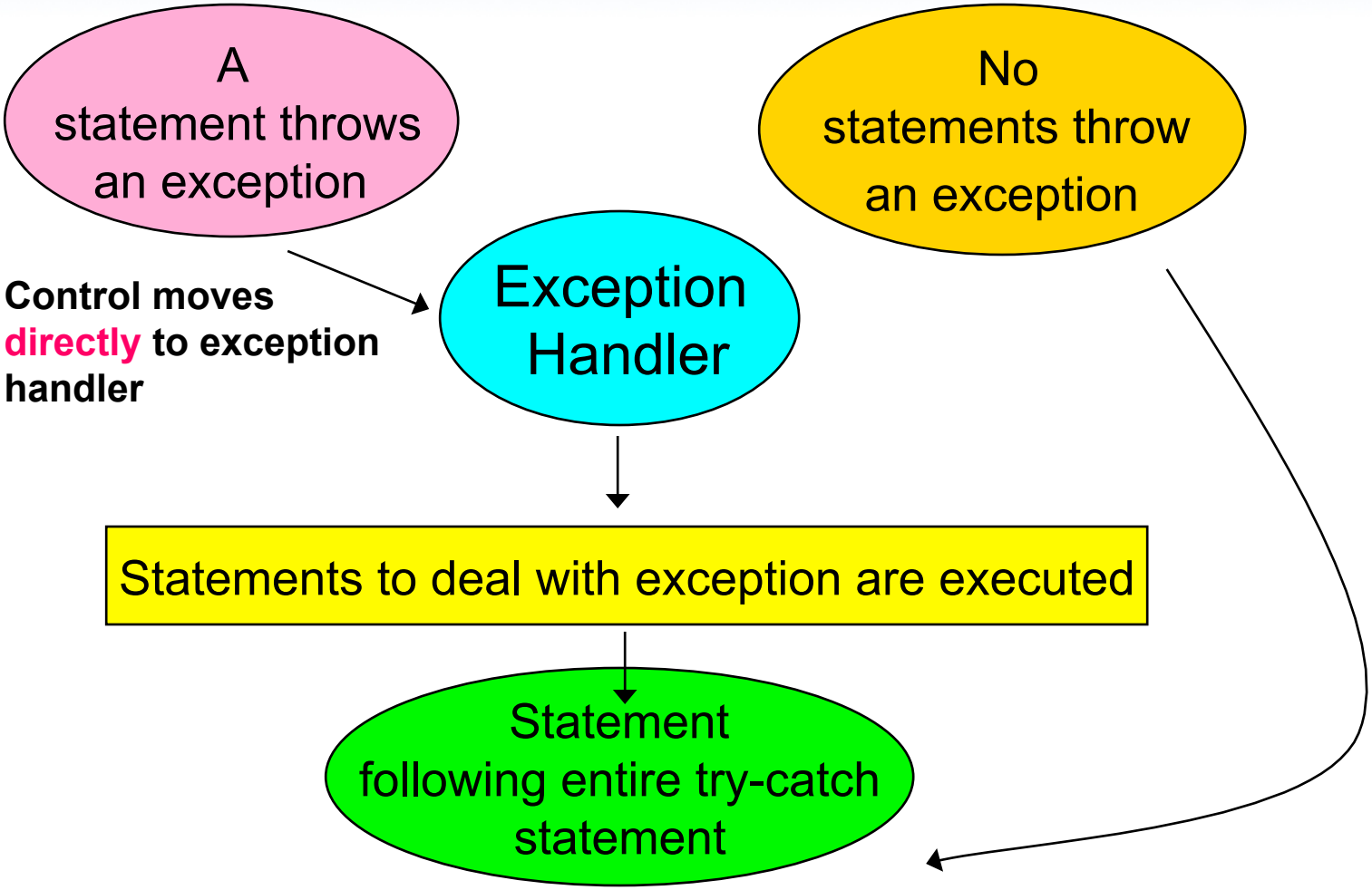
Example of a `try-catch` Statement

```
try
{
    // Statements that process personnel data and may throw
    // exceptions of type int, string, and SalaryError
}
catch (int)
{
    // Statements to handle an int exception
}
```

try-catch Continued

```
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`



Selecting an Exception Handler

The computer:

- **Examines data types of the formal parameters in exception handlers**
- **Searches in a "north-to-south" order**
- **Selects first formal parameter whose data type matches that of the thrown exception**
- **Ellipse parameters are a "wild card" and catch all (*Place the "catch all" handler last*)**

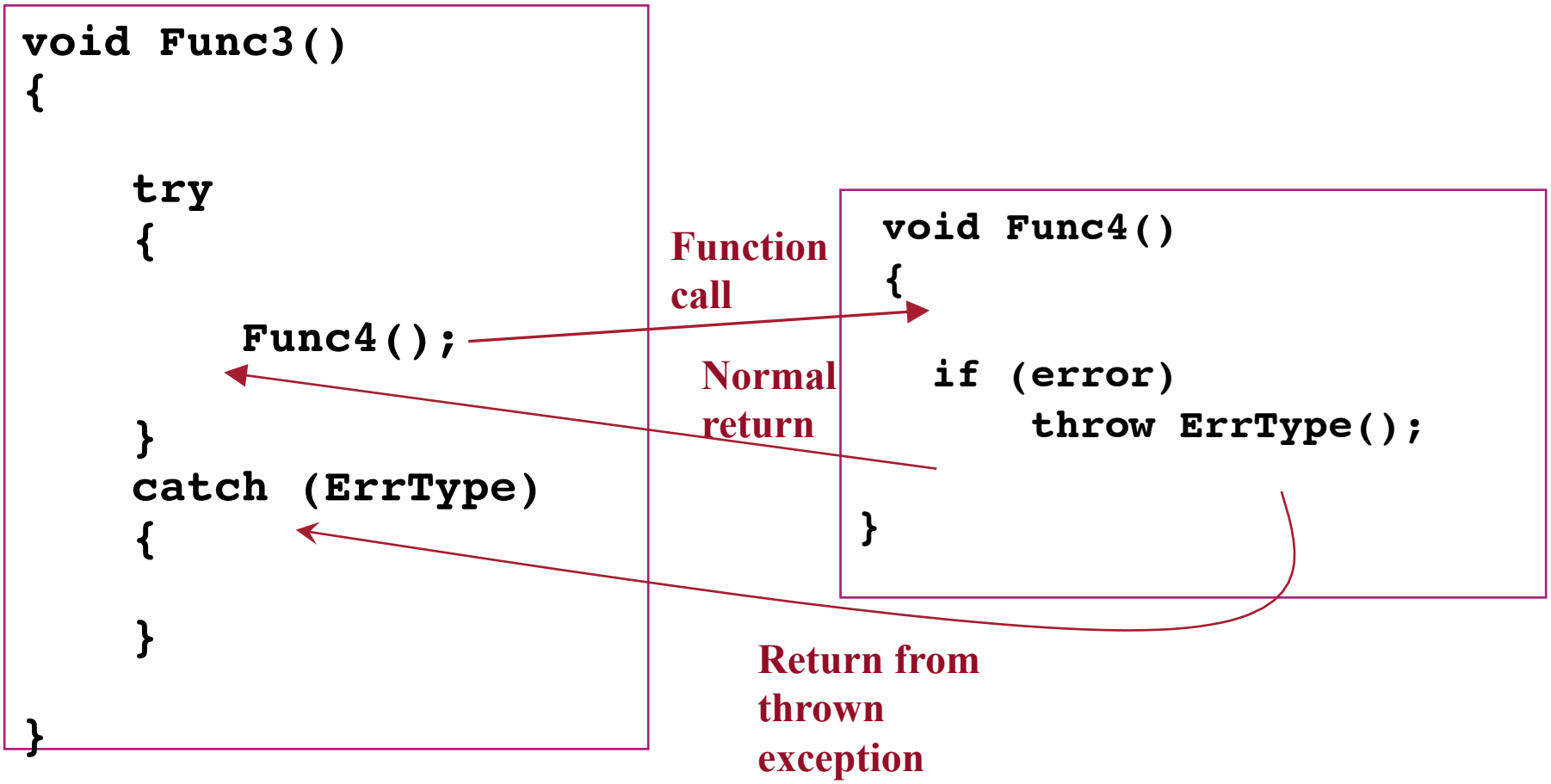
More on Selecting Exception Handlers

- The parameter's name is needed only if statements in the body of the exception handler use that variable
- It is a good idea to use only:
  - user-defined classes (and structs) as exception types
  - one type per exception
  - descriptive identifiers

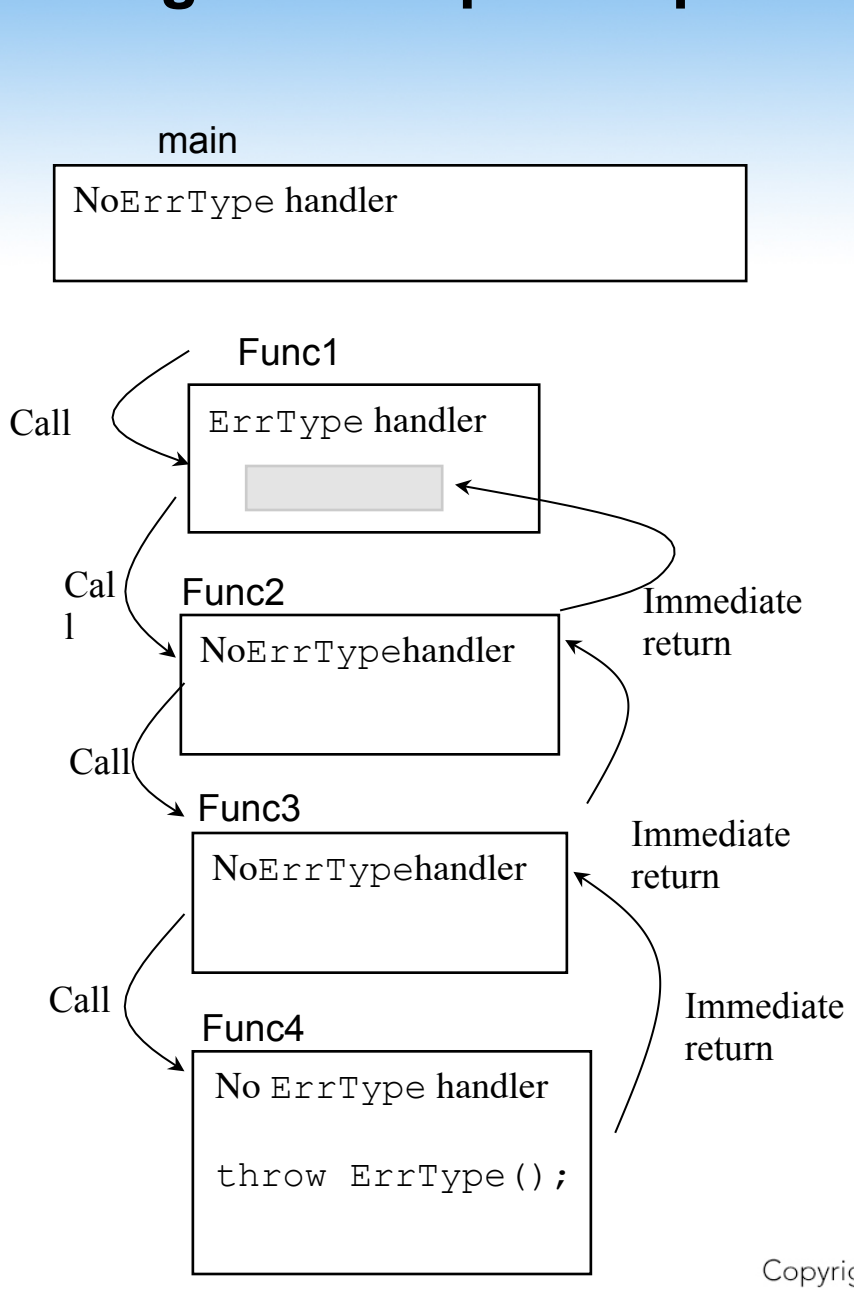
Nonlocal Exception Handlers

- **It is more common for the throw to occur inside a function that is *called* from within a try-clause than for the throw to be located *within* the try-catch statement**

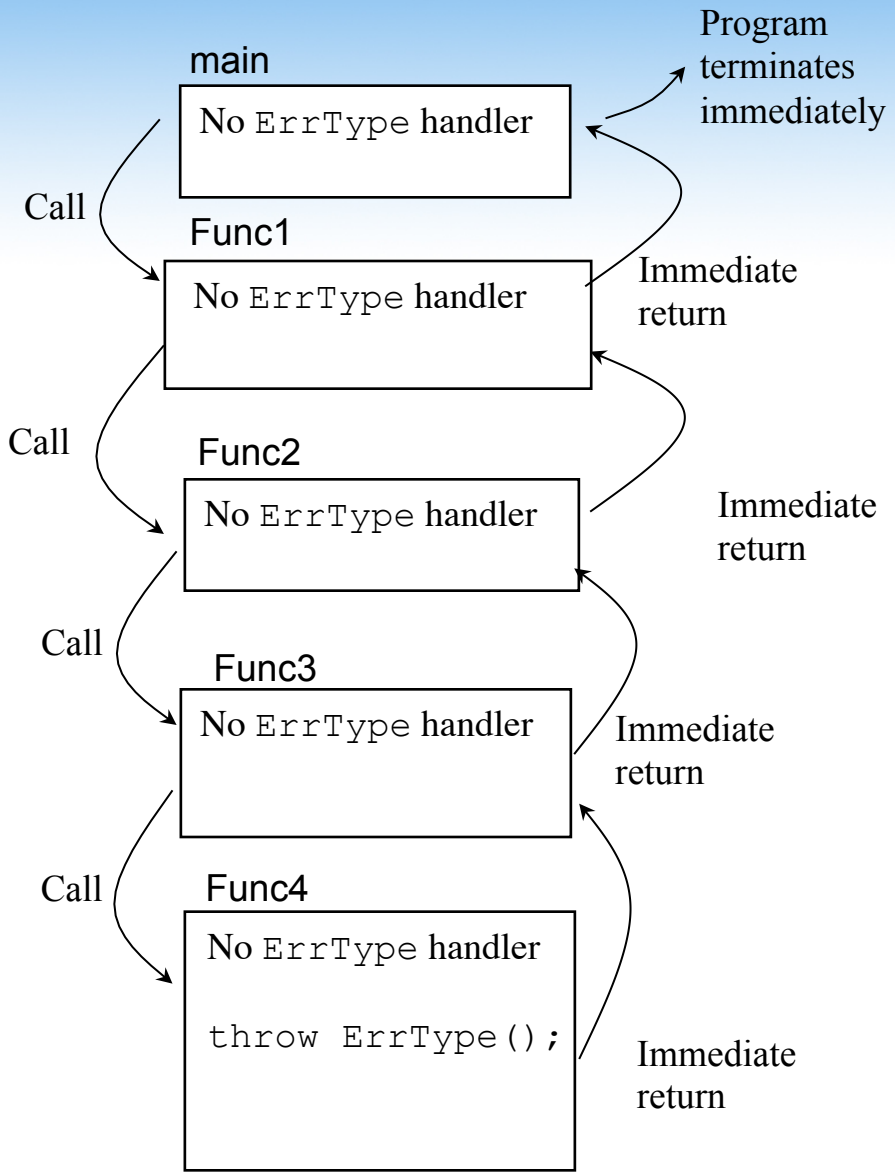
Throwing an Exception to be Caught by the Calling Code



Passing an Exception up the Chain of Function Calls



Function Func1 has a handler for ErrType



No function has a handler for ErrType

Re-Throwing an Exception

- The throw expression is optional

```
throw;
```

- Re-throwing an exception in C++ allows partial exception handling

Standard Exceptions

- **Exceptions Thrown by the Language**
  - **new, dynamic_cast, typeid, exception specification**
- **Exceptions Thrown by Standard Library Routines**
  - **Facilities inherited from the C language**
  - **Facilities designed specifically for C++**

Dividing by ZERO

Apply what you know:

```
int Quotient(/* in */ int numer,  // The numerator
             /* in */ int denom)  // The denominator
{
    if (denom != 0)
        return numer / denom;
    else
        // What to do??
}
```

A Solution

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

#include<iostream>
#include <string>

using namespace std;

int Quotient(int, int);

class DivByZero   // Exception class

{};

int main()

{

    int numer;    // Numerator

    int denom;    // Denominator

    cout << "Enter numerator and denominator: ";
```

```
cin >> numer >> denom;
while (cin)
{
    try
    {
        cout << "Their quotient: "
             << Quotient(numer, denom) << endl;
    }
    catch (DivByZero)
    {
        cout << "*** Denominator can't be 0"
            << endl;
    }
```

```
cout << "Enter numerator and denominator: ";
        cin >> numer >> denom;
    }
    return 0;
}
int Quotient(/* in */ int numer, // The numerator
          /* in */ int denom)   // The denominator
{
    if (denom == 0)
        throw DivByZero();
    return numer / denom;
}
```

Appointment Calendar

- Replace array-based list with linked list to demonstrate that changing implementation doesn't change client code
- Add exceptions to Appointment Calendar Program