Programming in Modern C++

Module M36: Exceptions (Error handling in C): Part 1

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Weekly Recap

- Leveraging an innovative solution to the Salary Processing Application in C using function pointers, we compare C and C++ solutions to the problem
- The new C solution with function pointers is used to explain the mechanism for dynamic binding (polymorphic dispatch) based on `virtual` function tables
- Understood casting in C and C++
- Explained cast operators in C++ and discussed the evils of C-style casting
- Studied `const_cast`, `static_cast`, `reinterpret_cast`, and `dynamic_cast` with examples
- Understood casting at run-time with RTTI and `typeid` operator
- Introduced the Semantics of Multiple Inheritance in C++
- Discussed the Diamond Problem and solution approaches
- Illustrated the design choice between inheritance and composition

Module Objectives

- Understand the Error handling in C

Module Outline

Exception Fundamentals

Exception Fundamentals

What are Exceptions?

- Conditions that arise
  - Infrequently and Unexpectedly
  - Generally betray a Program Error
  - Require a considered Programmatic Response
  - Run-time Anomalies – yes, but not necessarily
- Leading to
  - Crippling the Program
  - May pull the entire System down
  - Defensive Technique
    - Crashing Exceptions verses Tangled Design and Code

Module M36

Partha Pratim Das

Exception Causes

- Unexpected Systems State
  - Exhaustion of Resources
    - ▷ Low Free Store Memory
    - ▷ Low Disk Space
  - Pushing to a Full Stack
- External Events
  - Ĉ
  - Socket Event
- Logical Errors
  - Pop from an Empty Stack
  - Resource Errors – like Memory Read/Write
- Run time Errors
  - Arithmetic Overflow / Underflow
  - Out of Range
- Undefined Operation
  - Division by Zero

Exception Handling?

- Exception Handling is a mechanism that separates the detection and handling of circumstantial Exceptional Flow from Normal Flow
- Current state saved in a pre-defined location
- Execution switched to a pre-defined handler

> Exceptions are C++'s means of separating error reporting from error handling
>
> – Bjarne Stroustrup

Types of Exceptions

- **Asynchronous Exceptions**:
  - Exceptions that come Unexpectedly
  - Example – an Interrupt in a Program
  - Takes control away from the Executing Thread context to a context that is different from that which caused the exception
- **Synchronous Exceptions**:
  - Planned Exceptions
  - Handled in an organized manner
  - The most common type of Synchronous Exception is implemented as a `throw`

Exception Stages

[1] **Error Incidence**

- Synchronous (S/W) Logical Error
- Asynchronous (H/W) Interrupt (S/W Interrupt)

[2] **Create Object & Raise Exception**

- An Exception Object can be of any Complete Type - an `int` to a full blown C++ class object

[3] **Detect Exception**

- Polling – Software Tests
- Notification – Control (Stack) Adjustments

[4] **Handle Exception**

- Ignore: hope someone else handles it, that is, Do Not Catch
- Act: but allow others to handle it afterwards, that is, Catch, Handle and Re-Throw
- Own: take complete ownership, that is, Catch and Handle

[5] **Recover from Exception**

- Continue Execution: If handled inside the program
- Abort Execution: If handled outside the program

Exception Stages


```
int f() {
    int error;
    /* ... */
    if (error) /* Stage 1: error occurred */
        return -1; /* Stage 2: generate exception object */
    /* ... */
}

int main(void) {
    if (f() != 0) /* Stage 3: detect exception */
    {
        /* Stage 4: handle exception */
    }
    /* Stage 5: recover */
}
```

Error Handling in C

Support for Error Handling in C

- Support for Error Handling in C
  - C language does not provide any specific feature for error handling. Consequenly, developers are forced to use normal programming features in a disciplined way to handle errors. This has led to industry practices that the developers should abide by
  - C Standard Library provides a collection of headers that can be used for handling errors in different contexts. None of them is complete in itself, but together they kind of cover most situations. This again has led to industry practices that the developers should follow
- Language Features
  - Return Value & Parameters
  - Local `goto`
- Standard Library Support
  - Global Variables (`<errno.h>`)
  - Abnormal Termination (`<stdlib.h>`)
  - Conditional Termination (`<assert.h>`)
  - Non-Local `goto` (`<setjmp.h>`)
  - Signals (`<signal.h>`)

Return Value & Parameters

- Function Return Value Mechanism
  - Created by the Callee as Temporary Objects
  - Passed onto the Caller
  - Caller checks for Error Conditions
  - Return Values can be ignored and lost
  - Return Values are temporary
- Function (output) Parameter Mechanism
  - Outbound Parameters
  - Bound to arguments
  - Offer multiple logical Return Values

Example: Return Value & Parameters

```
int Push(int i) {
    if (top_ == size-1) // Incidence
        return 0; // Raise
    else
        stack_[++top_] = i;

    return 1;
}

int main() {
    int x;
    // ...
    if (!Push(x)) { // Detect
        // Handling
    }
    // Recovery
}
```

Local goto

- Local goto Mechanism
  - (At Source) *Escapes*: Gets Control out of a Deep Nested Loop
  - (At Destination) *Refactors*: Actions from Multiple Points of Error Inception
- A group of C Features
  - `goto` Label;
  - `break` `continue`;
  - `default` `switch` `case`

Example: Local goto

```
_PHNDLR _cdecl signal(int signum, _PHNDLR sigact)
{ // Lifted from VC98\CRT\SRC\WINSIG.C
...     /* Check for sigact support */
        if ( (sigact == ...) ) goto sigreterror;

        /* Not exceptions in the host OS. */
        if ( (signum == ... ) { ... goto sigreterror; }
    else { ... goto sigretok; }

        /* Exceptions in the host OS. */
        if ( (signum ...) ) goto sigreterror;
...
sigretok:
        return(oldsigact);

sigreterror:
        errno = EINVAL;
        return(SIG_ERR);
}
```

Example: Local goto

```
_PHNDLR __cdecl signal(int signum, _PHNDLR sigact)
{ // Lifted from VC98\CRT\SRC\WINSIG.C
...      /* Check for sigact support */
         if ( (sigact == ...) ) goto sigreterror;

         /* Not exceptions in the host OS. */
         if ( (signum == ... ) { ... goto sigreterror; }
        else { ... goto sigretok; }

         /* Exceptions in the host OS. */
         if ( (signum ...) ) goto sigreterror;
...
sigretok:
         return(oldsigact);

sigreterror:
         errno = EINVAL;
         return(SIG_ERR);
}
```

Example: Local goto

```
_PHNDLR __cdecl signal(int signum, _PHNDLR sigact)
{ // Lifted from VC98\CRT\SRC\WINSIG.C
...     /* Check for sigact support */
        if ( (sigact == ...) ) goto sigreterror;

        /* Not exceptions in the host OS. */
        if ( (signum == ... ) { ... goto sigreterror; }
       else { ... goto sigretok; }

        /* Exceptions in the host OS. */
        if ( (signum ...) ) goto sigreterror;
...
sigretok:
        return(oldsigact);

sigreterror:
        errno = EINVAL;
        return(SIG_ERR);
}
```

Example: Local goto

```
_PHNDLR __cdecl signal(int signum, _PHNDLR sigact)
{ // Lifted from VC98\CRT\SRC\WINSIG.C
...      /* Check for sigact support */
         if ( (sigact == ...) ) goto sigreterror;

         /* Not exceptions in the host OS. */
         if ( (signum == ... ) ) { ... goto sigreterror; }
        else { ... goto sigretok; }

         /* Exceptions in the host OS. */
         if ( (signum ...) ) goto sigreterror;
...
sigretok:
         return(oldsigact);

sigreterror:
         errno = EINVAL;
         return(SIG_ERR);
}
```

Global Variables

Module M36

Partha Pratim Das

Weekly Recap

Objective & Outline

Exception Fundamentals

Types of Exceptions

Exception Stages

Error Handling in C

C Language Features

RV & Params

Local goto

C Standard Library Support

Global Variables

Abnormal Termination

Conditional Termination

Non-Local goto

Signals

Shortcomings

Module Summary

- GV Mechanism
  - Use a designated Global Error Variable
  - Set it on Error
  - Poll / Check it for Detection
- Standard Library GV Mechanism
  - `<errno.h>/<cerrno>`

Example: Global Variables

```
#include <errno.h>
#include <math.h>
#include <stdio.h>

int main() {
    double x, y, result;
    /*... somehow set 'x' and 'y' ...*/
    errno = 0;

    result = pow(x, y);

    if (errno == EDOM)
        printf("Domain error on x/y pair \n");
    else
        if (errno == ERANGE)
            printf("range error in result \n");
        else
            printf("x to the y = %d \n", (int) result);
}
```

Abnormal Termination

- Program Halting Functions provided by
  - `<stdlib.h>/<cstdlib>`
- `abort()`
  - Catastrophic Program Failure
- `exit()`
  - Code Clean up via `atexit()` Registrations
- `atexit()`
  - Handlers called in reverse order of their Registrations

Example: Abnormal Termination


```
#include <stdio.h>
#include <stdlib.h>

static void atexit_handler_1(void) {
    printf("within 'atexit_handler_1' \n");
}

static void atexit_handler_2(void) {
    printf("within 'atexit_handler_2' \n");
}

int main() {
    atexit(atexit_handler_1);
    atexit(atexit_handler_2);
    exit(EXIT_SUCCESS);

    printf("This line should never appear \n");

    return 0;
}
```

```
within 'atexit_handler_2'
within 'atexit_handler_1'
```

Conditional Termination

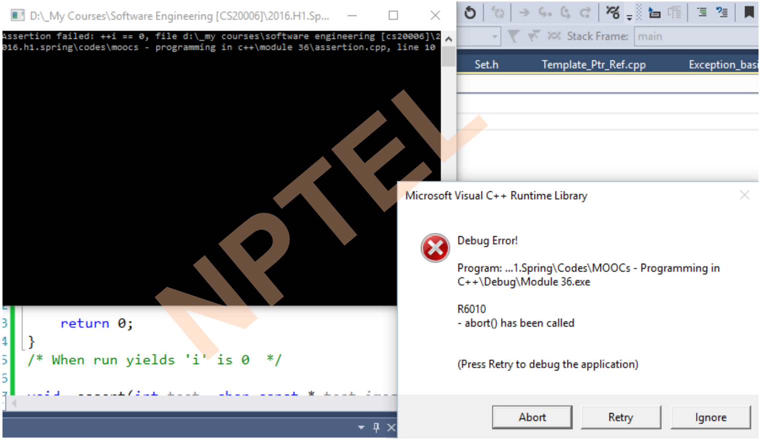
- Diagnostic ASSERT macro defined in
  - `<assert.h>/<cassert>`
- Assertions valid when `NDEBUG` macro is not defined (debug build is done)
- Assert calls internal function, reports the source file details and then Terminates

Example: Conditional Termination

```
/* Debug version */
//#define NDEBUG
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>

/* When run - Asserts  */
int main() {  int i = 0;
    assert(++i == 0); // Assert 0 here

    printf(" i is %d \n", i);

    return 0;
}
void _assert(int test, char const * test_image, char const * file, int line) {
    if (!test) { printf("assertion failed: %s , file %s , line %d\n", test_image, file, line);
        abort();
    }
}
```

```
Assertion failed: ++i == 0, // On MSVC++
file d:\ppd\my courses...\codes\msvc\programming in modern c++\exception in c\assertion.c,
line 8

a.out: main.c:17: main: Assertion '++i == 0' failed. // On onlinegdb
```

Example: Conditional Termination (On MSVC++)



Example: Conditional Termination

```
/* Release version */
#define NDEBUG
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>

/* When run yields 'i' is 0  */
int main() {
    int i = 0;
    assert(++i == 0);  // Assert 0 here

    printf(" i is %d \n", i);

    return 0;
}
void _assert(int test, char const * test_image, char const * file, int line) {

    if (!test) {
        printf("assertion failed: %s , file %s , line %d\n", test_image, file, line);
        abort();
    }
}
```

```
i is 0
```

Non-Local goto

- `setjmp()` and `longjmp()` functions provided in `<setjmp.h>` Header along with collateral type `jmp_buf`
- `setjmp(jmp_buf)`
  - Sets the Jump point filling up the `jmp_buf` object with the current program context
- `longjmp(jmp_buf, int)`
  - Effects a Jump to the context of the `jmp_buf` object
  - Control return to `setjmp` call last called on `jmp_buf`

Example: Non-Local goto: The Dynamics

Caller	Callee

`#include <stdio.h>`
`#include <stdbool.h>`
`#include <setjmp.h>`
 	
`int main() {`
`    if (setjmp(jbuf) == 0) {`
`        printf("g() called\n");`
`        g();`
`        printf("g() returned\n");`
`    }`
`    else`
`        printf("g() failed\n");`
`    return 0;`
`}`
	`jmp_buf jbuf;`
	
	`void g() {`
	`    bool error = false;`
	`    printf("g() started\n");`
	`    if (error)`
	`        longjmp(jbuf, 1);`
	`    printf("g() ended\n");`
	`    return;`
	`}`

Example: Non-Local goto: The Dynamics

Caller	Callee

<pre>int main() {
    if (setjmp(jbuf) == 0) {
        printf("g() called\n");
        g();
        printf("g() returned\n");
    }
    else
        printf("g() failed\n");
    return 0;
}</pre>
	<pre>jmp_buf jbuf;
	
	void g() {
	    bool error = false;
	    printf("g() started\n");
	    if (error)
	        longjmp(jbuf, 1);
	    printf("g() ended\n");
	    return;
	}</pre>

(1) g() called	(2) g() successfully returned

```
g() called
g() started
g() ended
g() returned
```

Example: Non-Local goto: The Dynamics

Caller	Callee

<pre>int main() {
    if (setjmp(jbuf) == 0) {
        printf("g() called\n");
        g();
        printf("g() returned\n");
    }
    else
        printf("g() failed\n");
    return 0;
}</pre>
	<pre>jmp_buf jbuf;
	
	void g() {
	    bool error = true;
	    printf("g() started\n");
	    if (error)
	        longjmp(jbuf, 1);
	    printf("g() ended\n");
	    return;
	}</pre>

(1) g() called (3) setjmp takes to handler	(2) longjmp executed

```
g() called
g() started
g() failed
```

Example: Non-Local goto


```
#include <setjmp.h>
#include <stdio.h>

jmp_buf j;

void raise_exception() {
    printf("Exception raised. \n");
    longjmp(j, 1); /* Jump to exception handler */
    printf("This line should never appear \n");
}
int main() {
    if (setjmp(j) == 0) {
        printf("'setjmp' is initializing  j. \n");
        raise_exception();
        printf("This line should never appear \n");
    }
    else
        printf("'setjmp' was just jumped into. \n");
        /* The exception handler code here */
    return 0 ;
}
```

```
'setjmp' is initializing j.
Exception raised.
'setjmp' was just jumped into.
```

Signals

- Header `<signal.h>`
- `raise()`
  - Sends a signal to the executing program
- `signal()`
  - Registers interrupt signal handler
  - Returns the previous handler associated with the given signal
- Converts h/w interrupts to s/w interrupts

Example: Signals

```
// Use signal to attach a signal
// handler to the abort routine
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void SignalHandler(int signal) {
    printf("Application aborting...\n");
}

int main() {
    typedef void (*SignalHandlerPointer)(int);

    SignalHandlerPointer previousHandler;

    previousHandler = signal(SIGABRT, SignalHandler);

    abort();

    return 0;
}
```

```
Application aborting...
```

Shortcomings

- **Destructor-ignorant**:
  - Cannot release Local Objects i.e. Resources Leak
- **Obtrusive**:
  - Interrogating RV or GV results in Code Clutter
- **Inflexible**:
  - Spoils Normal Function Semantics
- **Non-native**:
  - Require Library Support outside Core Language

Module Summary

- Introduced the concept of exceptions
- Discussed error handling in C
- Illustrated various language features and library support in C for handling errors
- Demonstrated with examples

Programming in Modern C++

Module M37: Exceptions (Error handling in C++): Part 2

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Introduced the concept of exceptions
- Discussed error handling in C
- Illustrated various language features and library support in C for handling errors
- Demonstrated with examples

Module Objectives

- Understand the Error handling in C++

Module Outline

Exceptions in C++

Exceptions in C++

Expectations

- Separate *Error-Handling code* from *Normal code*
- *Language Mechanism* rather than of the Library
- Compiler for *Tracking Automatic Variables*
- Schemes for *Destruction of Dynamic Memory*
- *Less Overhead* for the Designer
- *Exception Propagation* from the deepest of levels
- *Various Exceptions* handled by a single Handler

Error Handling Dynamics: C and C++

Header | **Caller** | **Callee**

C Scenario

Header:
```
#include <stdio.h>
#include <stdbool.h>
#include <setjmp.h>
```

Caller:
```
int main() {
    if (setjmp(jbuf) == 0) {
        printf("g() called\n");
        g();
        printf("g() returned\n");
    }
    else printf("g() failed\n"); // On longjmp
    return 0;
}
```

Callee:
```
jmp_buf jbuf;
void g() {
    bool error = false;
    printf("g() started\n");
    if (error)
        longjmp(jbuf, 1);
    printf("g() ended\n");
    return;
}
```

C++ Scenario

Header:
```
#include <iostream>
#include <exception>
using namespace std;
```

Caller:
```
int main() {
    try {
        cout << "g() called\n";
        g();
        cout << "g() returned\n";
    }
    catch (Excp&) { cout << "g() failed\n"; }
    return 0;
}
```

Callee:
```
class Excp: public exception {};
void g() {
    bool error = false;
    cout << "g() started\n";
    if (error)
        throw Excp();
    cout << "g() ended\n";
    return;
}
```

try-throw-catch

Caller	Callee

<pre>int main() {
    try {
        cout << "g() called\n";
        g();
        cout << "g() returned\n";
    }
    catch (Excp&) { cout << "g() failed\n"; }
    return 0;
}</pre>
	<pre>class Excp: public exception {};
	void g() {
	    bool error = false;
	    cout << "g() started\n";
	    if (error)
	        throw Excp();
	    cout << "g() ended\n";
	    return;
	}</pre>

(1) g() called	(2) g() successfully returned

```
g() called
g() started
g() ended
g() returned
```

try-throw-catch

Caller	Callee

<pre>int main() {
    try {
        cout << "g() called\n";
        g();
        cout << "g() returned\n";
    }
    catch (Excp&) { cout << "g() failed\n"; }
    return 0;
}</pre>
	<pre>class Excp: public exception {};
	class A {};
	void g() { A a;
	    bool error = true;
	    cout << "g() started\n";
	    if (error)
	        throw Excp();
	    cout << "g() ended\n";
	    return;
	}</pre>

(1) g() called (5) Exception caught by catch clause (6) Normal flow continues	(2) Exception raised (3) Stack frame of g() unwinds and destructor of a called (4) Remaining execution of g() and cout skipped

```
g() called
g() started
g() failed
```

Exception Flow

```
#include <iostream>
#include <exception>
using namespace std;
class MyException: public exception { };
class MyClass { public: ~MyClass() { } };
void h() { MyClass h_a;
    //throw 1;               // Line 1
    //throw 2.5;             // Line 2
    //throw MyException();   // Line 3
    //throw exception();     // Line 4
    //throw MyClass();       // Line 5
}   // Stack unwind, h_a.~MyClass() called
    // Passes on all exceptions
void g() { MyClass g_a;
    try { h();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 1
    catch (int) { cout << "int\n"; }
    // Catches exception from Line 2
    catch (double) { cout << "double\n"; }
    // Catches exception from Line 3-5 & passes on
    catch (...) { throw; }
}   // Stack unwind, g_a.~MyClass() called
```

```
void f() { MyClass f_a;
    try { g();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 3
    catch (MyException) { cout << "MyException\n"; }
    // Catches exception from Line 4
    catch (exception) { cout << "exception\n"; }
    // Catches exception from Line 5 & passes on
    catch (...) { throw; }
}  // Stack unwind, f_a.~MyClass() called

int main() {
    try { f();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 5
    catch (...) { cout << "Unknown\n"; }

    cout << "End of main()\n";
}
```

try Block: Exception Scope

- try block
  - Consolidate areas that might throw exceptions
- function try block
  - Area for detection is the entire function body
- Nested try block
  - Semantically equivalent to nested function calls

Function try

```
void f()
    try {
        throw E();
    }
    catch (E& e) {
    }
```

Note: The usual curly braces for the function scope are not to be put here

Nested try

```
try {
    try { throw E(); }
    catch (E& e) { }
}
catch (E& e1) {
}
```

catch Block: Exception Arguments

- catch block
  - Name for the Exception Handler
  - Catching an Exception is like invoking a function
  - Immediately follows the try block
  - Unique Formal Parameter for each Handler
  - Can simply be a Type Name to distinguish its Handler from others

try-catch: Exception Matching

- **Exact Match**
  - The catch argument type matches the type of the thrown object
    - *No implicit conversion is allowed*
- **Generalization / Specialization**
  - The catch argument is a public base class of the thrown class object
- **Pointer**
  - Pointer types – convertible by standard conversion

try-catch: Exception Matching

- In the *order of appearance* with matching
- If Base Class `catch` block precedes Derived Class `catch` block
  - Compiler issues a warning and continues
  - Unreachable code (derived class handler) ignored
- `catch(...)` block must be the last `catch` block because it catches all exceptions
- If no matching Handler is found in the current scope, the search continues to find a matching handler in a dynamically surrounding `try` block
  - *Stack Unwinds*
- If eventually no handler is found, `terminate()` is called

throw *Expression*: Exception Raise

- *Expression* is treated the same way as
  - A *function argument* in a call or the *operand of a return* statement
- Exception Context
  - `class Exception { };`
- The *Expression*
  - Generate an Exception object to throw
    - ▷ `throw Exception();`
  - Or, Copies an existing Exception object to throw
    - ▷ `Exception ex;`
    - ▷ `...`
    - ▷ `throw ex; // Exception(ex);`
- *Exception object is created on the Free Store*

`throw` *Expression*: Restrictions

- For a UDT Expression
  - Copy Constructor and Destructor should be supported
- The type of Expression cannot be an incomplete type or a pointer to an incomplete type
  - No incomplete type like `void`, array of unknown size or of elements of incomplete type, Declared but not Defined `struct` / `union` / `enum` / `class` Objects or Pointers to such Objects
  - No pointer to an incomplete type, except `void*`, `const void*`, `volatile void*`, `const volatile void*`

(re)-throw: Throwing Again?

- Re-throw
  - catch may pass on the exception after handling
  - Re-throw is not same as throwing again!

Throws again

```
try { ... }
catch (Exception& ex) {
    // Handle and
    ...
    // Raise again
    throw ex;
// ex copied
// ex destructed
}
```

Re-throw

```
try { ... }
catch (Exception& ex) {
    // Handle and
    ...
    // Pass-on
    throw;
    // No copy
// No Destruction
}
```

Advantages

- **Destructor-savvy**:
  - Stack unwinds; Orderly destruction of Local-objects
- **Unobtrusive**:
  - Exception Handling is implicit and automatic
  - No clutter of error checks
- **Precise**:
  - Exception Object Type designed using semantics
- **Native and Standard**:
  - EH is part of the C++ language
  - EH is available in all standard C++ compilers

Advantages

- **Scalable**:
  - Each function can have multiple try blocks
  - Each try block can have a single Handler or a group of Handlers
  - Each Handler can catch a single type, a group of types, or all types
- **Fault-tolerant**:
  - Functions can specify the exception types to throw; Handlers can specify the exception types to catch
  - Violation behavior of these specifications is predictable and user-configurable

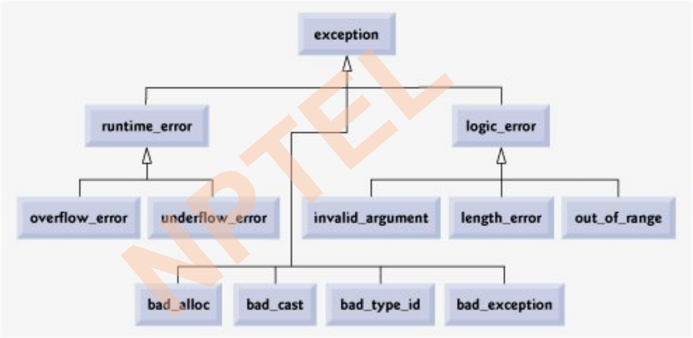
Exceptions in Standard Library: std::exception

All objects thrown by components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type by reference.

```
class exception {
public:
    exception() throw();
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
}
```

Sources: std::exception and std::exception in C++11, C++14, C++17 & C++20

Exceptions in Standard Library: std::exception



Sources: Standard Library Exception Hierarchy

Exceptions in Standard Library: `std::exception`

- `logic_error`: Faulty logic like violating logical preconditions or class invariants (may be preventable)
  - `invalid_argument`: An argument value has not been accepted
  - `domain_error`: Situations where the inputs are outside of the domain for an operation
  - `length_error`: Exceeding implementation defined length limits for some object
  - `out_of_range`: Attempt to access elements out of defined range
- `runtime_error`: Due to events beyond the scope of the program and can not be easily predicted
  - `range_error`: Result cannot be represented by the destination type
  - `overflow_error`: Arithmetic overflow errors (Result is too large for the destination type)
  - `underflow_error`: Arithmetic underflow errors (Result is a subnormal floating-point value)
- `bad_typeid`: Exception thrown on typeid of null pointer
- `bad_cast`: Exception thrown on failure to dynamic cast
- `bad_alloc`: Exception thrown on failure allocating memory
- `bad_exception`: Exception thrown by unexpected handler

Sources: `std::exception` and `std::exception in C++11, C++14, C++17 & C++20`

Exceptions in Standard Library: std::exception: C++98, C++11, C++14, C++17 & C++20

- logic_error
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error (C++11)
- bad_optional_access (C++17)
- runtime_error
  - range_error
  - overflow_error
  - underflow_error
  - regex_error (C++11)
  - system_error (C++11)
    - ios_base::failure (C++11)
    - filesystem::filesystem_error (C++17)
  - txtion (TM TS)
  - nonexistent_local_time (C++20)
  - ambiguous_local_time (C++20)
  - format_error (C++20)
- bad_typeid
- bad_cast
  - bad_any_cast (C++17)
- bad_weak_ptr (C++11)
- bad_function_call (C++11)
- bad_alloc
  - bad_array_new_length (C++11)
- bad_exception
- ios_base::failure (until C++11)
- bad_variant_access (C++17)

Module Summary

- Discussed exception (error) handling in C++
- Illustrated `try-throw-catch` feature in C++ for handling errors
- Demonstrated with examples

Programming in Modern C++

Module M38: Template (Function Template): Part 1

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Discussed exception (error) handling in C++
- Illustrated `try-throw-catch` feature in C++ for handling errors
- Demonstrated with examples

Module Objectives

- Understand Templates in C++
- Understand Function Templates

Module Outline

What is a Template?

What is a Template?

- Templates are specifications of a *collection of functions or classes which are parameterized by types*
- Examples:
  - Function search, min etc.
    - The basic algorithms in these functions are the same independent of types
    - Yet, we need to write different versions of these functions for strong type checking in C++
  - Classes list, queue etc.
    - The data members and the methods are almost the same for list of numbers, list of objects
    - Yet, we need to define different classes

Function Template

Function Template

Function Template: Code reuse in Algorithms

- We need to compute the maximum of two values that can be of:
  - `int`
  - `double`
  - `char *` (C-String)
  - `Complex` (user-defined class for complex numbers)
  - ...
- We can do this with overloaded `Max` functions:

```
int Max(int x, int y);
double Max(double x, double y);
char *Max(char *x, char *y);
Complex Max(Complex x, Complex y);
```

  With every new type, we need to add an overloaded function in the library!
- Issues in `Max` function
  - *Same algorithm* (compare two values using the appropriate operator of the type and return the larger value)
  - *Different code versions* of these functions for strong type checking in C++

Max as Overload


```
#include <iostream>
#include <cstring>
#include <cmath>
using namespace std;
// Overloads of Max
int Max(int x, int y) { return x > y ? x : y; }
double Max(double x, double y) { return x > y ? x : y; }
char *Max(char *x, char *y) { return strcmp(x, y) > 0 ? x : y; }

int main() { int a = 3, b = 5, iMax; double c = 2.1, d = 3.7, dMax;
    cout << "Max(" << a << ", " << b << ") = " << Max(a, b) << endl;
    cout << "Max(" << c << ", " << d << ") = " << Max(c, d) << endl;

    char *s1 = new char[6], *s2 = new char[6];
    strcpy(s1, "black"); strcpy(s2, "white");
    cout << "Max(" << s1 << ", " << s2 << ") = " << Max(s1, s2) << endl;
    strcpy(s1, "white"); strcpy(s2, "black");
    cout << "Max(" << s1 << ", " << s2 << ") = " << Max(s1, s2) << endl;
}
```

- Overloaded solutions work
- In some cases (C-string), similar algorithms have exceptions
- With every new type, a new overloaded Max is needed
- Can we make Max generic and make a library to work with future types?
- How about macros?

Max as a Macro

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

// Max as a macro
#define Max(x, y) (((x) > (y))? x: y)

int main() {
    int a = 3, b = 5;
    double c = 2.1, d = 3.7;

    cout << "Max(" << a << ", " << b << ") = " << Max(a, b) << endl; // Output: Max(3, 5) = 5

    cout << "Max(" << c << ", " << d << ") = " << Max(c, d) << endl; // Output: Max(2.1, 3.7) = 3.7

    return 0;
}
```

- Max, being a macro, is type oblivious – can be used for int as well as double, etc.
- Note the parentheses around parameters to protect precedence
- Note the parentheses around the whole expression to protect precedence
- Looks like a function – but does not behave as such

Max as a Macro: Pitfalls


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

#define Max(x, y) (((x) > (y))? x: y)

int main() { int a = 3, b = 5; double c = 2.1, d = 3.7;
    // Side Effects
    cout << "Max(" << a << ", " << b << ") = ";    // Output: Max(3, 5) = 6
    cout << Max(a++, b++) << endl;
    cout << "a = " << a << ", b = " << b << endl; // Output: a = 4, b = 7

    // C-String Comparison
    char *s1 = new char[6], *s2 = new char[6];
    strcpy(s1, "black"); strcpy(s2, "white");
    cout << "Max(" << s1 << ", " << s2 << ") = " << Max(s1, s2) << endl; // Max(black, white) = white

    strcpy(s1, "white"); strcpy(s2, "black");
    cout << "Max(" << s1 << ", " << s2 << ") = " << Max(s1, s2) << endl; // Max(white, black) = black
}
```

- In "Side Effects" – the result is wrong, the larger values gets incremented twice
- In "C-String Comparison" – swapping parameters changes the result – actually compares pointers

Function Template

- A function template
  - describes how a function should be built
  - supplies the definition of the function using some arbitrary types, (as place holders)
    - a parameterized definition
  - can be considered the definition for a set of overloaded versions of a function
  - is identified by the keyword `template`
    - followed by comma-separated list of parameter identifiers (each preceded by keyword `class` or keyword `typename`)
    - enclosed between $<$ and $>$ delimiters
    - followed by the signature the function
  - Note that every template parameter is a built-in type or class – type parameters

Max as a Function Template

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

template<class T>
T Max(T x, T y) {
    return x > y ? x : y;
}

int main() {
    int a = 3, b = 5, iMax;
    double c = 2.1, d = 3.7, dMax;

    iMax = Max<int>(a, b);
    cout << "Max(" << a << ", " << b << ") = " << iMax << endl; // Output: Max(3, 5) = 5

    dMax = Max<double>(c, d);
    cout << "Max(" << c << ", " << d << ") = " << dMax << endl; // Output: Max(2.1, 3.7) = 3.7
}
```

- `Max`, now, knows the type
- Template type parameter `T` explicitly specified in instantiation of `Max<int>`, `Max<double>`

Max as a Function Template: Pitfall "Side Effects" – Solved


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

template<class T>
T Max(T x, T y) {
    return x > y ? x : y;
}

int main() {
    int a = 3, b = 5, iMax;

    // Side Effects
    cout << "Max(" << a << ", " << b << ") = ";
    iMax = Max<int>(a++, b++);
    cout << iMax << endl; // Output: Max(3, 5) = 5

    cout << "a = " << a << ", b = " << b << endl; // Output: a = 4, b = 6
}
```

- Max is now a proper function call – no side effect

Max as a Function Template: Pitfall "C-String Comparison" – Solved


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

template<class T> T Max(T x, T y) { return x > y ? x : y; }

template<> // Template specialization for 'char *' type
char *Max<char *>(char *x, char *y) { return strcmp(x, y) > 0 ? x : y; }

int main() { char *s1 = new char[6], *s2 = new char[6];
    strcpy(s1, "black"); strcpy(s2, "white");
    cout << "Max(" << s1 << ", " << s2 << ") = " << Max<char*>(s1, s2) << endl;
        // Output: Max(black, white) = white

    strcpy(s1, "white"); strcpy(s2, "black");
    cout << "Max(" << s1 << ", " << s2 << ") = " << Max<char*>(s1, s2) << endl;
        // Output: Max(black, white) = white
}
```

- Generic template code does not work for C-Strings as it compares pointers, not the strings pointed by them
- **We provide a specialization to compare pointers using comparison of strings**
- Need to specify type explicitly is bothersome

Max as a Function Template: Implicit Instantiation

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

template<class T> T Max(T x, T y) { return x > y ? x : y; }

int main() { int a = 3, b = 5, iMax; double c = 2.1, d = 3.7, dMax;
    iMax = Max(a, b); // Type 'int' inferred from 'a' and 'b' parameters types
    cout << "Max(" << a << ", " << b << ") = " << iMax << endl;
            // Output: Max(3, 5) = 5

    dMax = Max(c, d); // Type 'double' inferred from 'c' and 'd' parameters types
    cout << "Max(" << c << ", " << d << ") = " << dMax << endl;
            // Output: Max(2.1, 3.7) = 3.7
}
```

- Often template type parameter T may be inferred from the type of parameters in the instance
- If the compiler cannot infer or infers wrongly, we use explicit instantiation

Template Argument Deduction: Implicit Instantiation

- Each item in the template parameter list is a template argument
- When a template function is invoked, the values of the template arguments are determined by seeing the types of the function arguments

```
template<class T> T Max(T x, T y);
template<> char *Max<char *>(char *x, char *y);
template <class T, int size> T Max(T x[size]);

int a, b; Max(a, b);         // Binds to Max<int>(int, int);
double c, d; Max(c, d);      // Binds to Max<double>(double, double);
char *s1, *s2; Max(s1, s2); // Binds to Max<char*>(char*, char*);

int pval[9]; Max(pval);      // Error!
```

- Three kinds of conversions are allowed
  - L-value transformation (for example, Array-to-pointer conversion)
  - Qualification conversion
  - Conversion to a base class instantiation from a class template
- If the same template parameter are found for more than one function argument, template argument deduction from each function argument must be the same

Max as a Function Template: With User-Defined Class

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

class Complex { double re_; double im_; public:
    Complex(double re = 0.0, double im = 0.0) : re_(re), im_(im) { };
    double norm() const { return sqrt(re_*re_+im_*im_); }
    friend bool operator>(const Complex& c1, const Complex& c2) { return c1.norm() > c2.norm(); }
    friend ostream& operator<<(ostream& os, const Complex& c) {
        os << "(" << c.re_ << ", " << c.im_ << ")"; return os;
    }
};
template<class T> T Max(T x, T y) { return x > y ? x : y; }
template<> char *Max<char *>(char *x, char *y) { return strcmp(x, y) > 0 ? x : y; }

int main() { Complex c1(2.1, 3.2), c2(6.2, 7.2);
    cout << "Max(" << c1 << ", " << c2 << ") = " << Max(c1, c2) << endl;
            // Output: Max((2.1, 3.2), (6.2, 7.2)) = (6.2, 7.2)
}
```

- When `Max` is instantiated with `class Complex`, we need comparison operator for `Complex`
- The code, therefore, will not compile without `bool operator>(const Complex&, const Complex&)`
- **Traits of type variable `T` include `bool operator>(T, T)` which the instantiating type must fulfill**

Max as a Function Template: Overloads


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

template<class T> T Max(T x, T y) { return x > y ? x : y; }

template<> char *Max<char *>(char *x, char *y) // Template specialization
    { return strcmp(x, y) > 0 ? x : y; }

template<class T, int size> T Max(T x[size]) { // Overloaded template function
    T t = x[0];
    for (int i = 0; i < size; ++i) { if (x[i] > t) t = x[i]; }

    return t;
}

int main() {
    int arr[] = { 2, 5, 6, 3, 7, 9, 4 };
    cout << "Max(arr) = " << Max<int, 7>(arr) << endl; // Output: Max(arr) = 9
}
```

- Template function can be overloaded
- A template parameter can be non-type (`int`) constant

Swap as a Function Template


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

template<class T> void Swap(T& one, T& other) { T temp;
    temp = one; one = other; other = temp;
}
int main() { int i = 10, j = 20;
    cout << "i = " << i << ", j = " << j << endl;
    Swap(i, j);
    cout << "i = " << i << ", j = " << j << endl;

    string s1("abc"), s2("def");

    cout << "s1 = " << s1 << ", s2 = " << s2 << endl;
    Swap(s1, s2);
    cout << "s1 = " << s1 << ", s2 = " << s2 << endl;
}
```

- **The traits of type variable `T` include**
  default constructor (`T::T()`) and
  copy assignment operator (`T operator=(const T&)`)
- **Our template function cannot be called `swap`, as `std` namespace has such a function**

typename

typename Keyword

- Consider:

```
template <class T> f (T x) {
    T::name * p;
}
```

- What does it mean?
  - `T::name` is a *type* and `p` is a *pointer* to that type
  - `T::name` and `p` are *variables* and this is a *multiplication*
- To resolve, we use keyword `typename`:

```
template <class T> f (T x) { T::name * p; } // Multiplication

template <class T> f (T x) { typename T::name * p; } // Type
```

- The keywords `class` and `typename` have almost the same meaning in a template parameter
- `typename` is also used to tell the compiler that an expression is a type expression

Module Summary

- Introduced the templates in C++
- Discussed function templates as generic algorithmic solution for code reuse
- Explained templates argument deduction for implicit instantiation
- Illustrated with examples

Programming in Modern C++

Module M39: Template (Class Template): Part 2

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Introduced the templates in C++
- Discussed function templates as generic algorithmic solution for code reuse
- Explained templates argument deduction for implicit instantiation
- Illustrated with examples

Module Objectives

- Understand Templates in C++
- Understand Class Templates

Module Outline

What is a Template?

What is a Template?

What is a Template?: RECAP (Module 38)

- Templates are specifications of a collection of functions or classes which are parameterized by types
- Examples:
  - Function search, min etc.
    - ▷ The basic algorithms in these functions are the same independent of types
    - ▷ Yet, we need to write different versions of these functions for strong type checking in C++
  - Classes list, queue etc.
    - ▷ The data members and the methods are almost the same for list of numbers, list of objects
    - ▷ Yet, we need to define different classes

Function Template

Function Template: Code reuse in Algorithms: RECAP (Module 38

- We need to compute the maximum of two values that can be of:
  - `int`
  - `double`
  - `char *` (C-String)
  - `Complex` (user-defined class for complex numbers)
  - ...
- We can do this with overloaded `Max` functions:

```
int Max(int x, int y);
double Max(double x, double y);
char *Max(char *x, char *y);
Complex Max(Complex x, Complex y);
```

  With every new type, we need to add an overloaded function in the library!

- Issues in `Max` function
  - *Same algorithm* (compare two values using the appropriate operator of the type and return the larger value)
  - *Different code versions* of these functions for strong type checking in C++

Class Template

Class Template: Code Reuse in Data Structure

- Solution of several problems needs stack (LIFO)
  - Reverse string (`char`)
  - Convert infix expression to postfix (`char`)
  - Evaluate postfix expression (`int` / `double` / `Complex` ...)
  - Depth-first traversal (`Node *`)
  - ...
- Solution of several problems needs queue (FIFO)
  - Task Scheduling (`Task *`)
  - Process Scheduling (`Process *`)
  - ...
- Solution of several problems needs list (ordered)
  - Implementing stack, queue (`int` / `char` / ...)
  - Implementing object collections (UDT)
  - ...
- Solution of several problems needs ...
- Issues in Data Structure
  - Data Structures are **generic - same interface, same algorithms**
  - C++ **implementations are different** due to element type

Stack of char and int

```
class Stack {
    char data_[100];            // Has type char
    int top_;
public:
    Stack() :top_(-1) { }
    ~Stack() { }

    void push(const char& item) // Has type char
    { data_[++top_] = item; }

    void pop()
    { --top_; }

    const char& top() const     // Has type char
    { return data_[top_]; }

    bool empty() const
    { return top_ == -1; }
};
```

```
class Stack {
    int data_[100];             // Has type int
    int top_;
public:
    Stack() :top_(-1) { }
    ~Stack() { }

    void push(const int& item)  // Has type int
    { data_[++top_] = item; }

    void pop()
    { --top_; }

    const int& top() const      // Has type int
    { return data_[top_]; }

    bool empty() const
    { return top_ == -1; }
};
```

- Stack of char
- Stack of int
- Can we combine these Stack codes using a type variable T?

Class Template

- A class template
  - describes how a class should be built
  - supplies the class description and the definition of the member functions using some arbitrary type name, (as a place holder)
  - is a:
    - parameterized type with
    - parameterized member functions
  - can be considered the definition for a unbounded set of class types
  - is identified by the keyword `template`
    - followed by comma-separated list of parameter identifiers (each preceded by keyword `class` or keyword `typename`)
    - enclosed between `<` and `>` delimiters
    - followed by the definition of the class
  - is often used for container classes
  - Note that every template parameter is a built-in type or class – type parameters

Stack as a Class Template: Stack.h


```
template<class T>
class Stack {
    T data_[100];
    int top_;
public:
    Stack() :top_(-1) { }
    ~Stack() { }

    void push(const T& item) { data_[++top_] = item; }

    void pop() { --top_; }

    const T& top() const { return data_[top_]; }

    bool empty() const { return top_ == -1; }
};
```

- Stack of type variable `T`
- **The traits of type variable `T` include**
  copy assignment operator (`T operator=(const T&)`)
- **We do not call our template class as `stack` because `std` namespace has a `class stack`**

Reverse String: Using Stack template

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

#include "Stack.h"

int main() {
    char str[10] = "ABCDE";

    Stack<char> s;          // Instantiated for char

    for (unsigned int i = 0; i < strlen(str); ++i)
        s.push(str[i]);

    cout << "Reversed String: ";
    while (!s.empty()) {
        cout << s.top();
        s.pop();
    }

    return 0;
}
```

- **Stack of type char**

Postfix Expression Evaluation: Using Stack template


```
#include <iostream>
#include "Stack.h"
using namespace std;

int main() { // Postfix expression: 1 2 3 * + 9 -
    unsigned int postfix[] = { '1', '2', '3', '*', '+', '9', '-' }, ch;

    Stack<int> s;         // Instantiated for int

    for (unsigned int i = 0; i < sizeof(postfix) / sizeof(unsigned int); ++i) {
        ch = postfix[i];
        if (isdigit(ch)) { s.push(ch - '0'); }
        else {
            int op1 = s.top(); s.pop();
            int op2 = s.top(); s.pop();
            switch (ch) {
                case '*': s.push(op2 * op1); break;
                case '/': s.push(op2 / op1); break;
                case '+': s.push(op2 + op1); break;
                case '-': s.push(op2 - op1); break;
            }
        }
    }
    cout << "\n Evaluation " << s.top();
}
```

Template Parameter Traits

- Parameter Types
  - may be of any type (including user defined types)
  - may be parameterized types, (that is, templates)
  - MUST support the methods used by the template functions:
    - What are the required constructors?
    - The required operator functions?
    - What are the necessary defining operations?

Function Template Instantiation: RECAP (Module 38)

- Each item in the template parameter list is a template argument
- When a template function is invoked, the values of the template arguments are determined by seeing the types of the function arguments

```
template<class T> T Max(T x, T y);
template<> char *Max<char *>(char *x, char *y);
template <class T, int size> T Max(T x[size]);

int a, b; Max(a, b);           // Binds to Max<int>(int, int);
double c, d; Max(c, d);        // Binds to Max<double>(double, double);
char *s1, *s2; Max(s1, s2);    // Binds to Max<char*>(char*, char*);

int pval[9]; Max<int, 7>(arr); // Binds to Max<int, n>(int[]);
```

- Three kinds of conversions are allowed
  - L-value transformation (for example, Array-to-pointer conversion)
  - Qualification conversion
  - Conversion to a base class instantiation from a class template
- If the same template parameter are found for more than one function argument, template argument deduction from each function argument must be the same

Class Template Instantiation

- Class Template is instantiated *only when it is required*:
  - `template<class T> class Stack;` // Is a forward declaration
  - `Stack<char> s;` // Is an error
  - `Stack<char> *ps;` // Is okay
  - `void ReverseString(Stack<char>& s, char *str);` Is okay
- Class template is instantiated before
  - An object is defined with class template instantiation
  - If a pointer or a reference is dereferenced (for example, a method is invoked)
- A template definition can refer to a class template or its instances but a non-template can only refer to template instances

Class Template Instantiation Example

```
#include <iostream>
#include <cstring>
using namespace std;
template<class T> class Stack;                       // Forward declaration
void ReverseString(Stack<char>& s, char *str);   // Stack template definition is not needed

template<class T>                                    // Definition
class Stack { T data_[100]; int top_;
public: Stack() :top_(-1) { } ~Stack() { }
    void push(const T& item) { data_[++top_] = item; }
    void pop() { --top_; }
    const T& top() const { return data_[top_]; }
    bool empty() const { return top_ == -1; }
};
int main() { char str[10] = "ABCDE";
    Stack<char> s;                                   // Stack template definition is needed
    ReverseString(s, str);
}
void ReverseString(Stack<char>& s, char *str) { // Stack template definition is needed
    for (unsigned int i = 0; i < strlen(str); ++i)
        s.push(str[i]);
    cout << "Reversed String: ";
    while (!s.empty())
        { cout << s.top(); s.pop(); }
}
```

Partial Template Instantiation and Default Template Parameters


```
#include <iostream>
#include <string>
#include <cstring>
template<class T1 = int, class T2 = string> // Version 1 with default parameters
class Student { T1 roll_; T2 name_;
public: Student(T1 r, T2 n) : roll_(r), name_(n) { }
    void Print() const { std::cout << "Version 1: (" << name_ << ", " << roll_ << ")" << std::endl; }
};
template<class T1> // Version 2: Partial Template Specialization
class Student<T1, char *> { T1 roll_; char *name_;
public: Student(T1 r, char *n) : roll_(r), name_(std::strcpy(new char[std::strlen(n) + 1], n)) { }
    void Print() const { std::cout << "Version 2: (" << name_ << ", " << roll_ << ")" << std::endl; }
};
int main() {
    Student<int, string> s1(2, "Ramesh"); s1.Print();    // Version 1: T1 = int, T2 = string
    Student<int>         s2(11, "Shampa"); s2.Print();   // Version 1: T1 = int, defa T2 = string
    Student<>            s3(7, "Gagan"); s3.Print();     // Version 1: defa T1 = int, defa T2 = string
    Student<string>      s4("X9", "Lalita"); s4.Print(); // Version 1: T1 = string, defa T2 = string
    Student<int, char*>  s5(3, "Gouri"); s5.Print();     // Version 2: T1 = int, T2 = char*
}
```

```
Version 1: (Ramesh, 2)
Version 1: (Shampa, 11)
Version 1: (Gagan, 7)
Version 1: (Lalita, X9)
Version 2: (Gouri, 3)
```

Templates and Inheritance: Example (List.h)

```
#ifndef __LIST_H
#define __LIST_H

#include <vector>
using namespace std;

template<class T>
class List {
public:
    void put(const T &val) { items.push_back(val); }
    int length() { return items.size(); }          // vector<T>::size()
    bool find(const T &val) {
        for (unsigned int i = 0; i < items.size(); ++i)
            if (items[i] == val) return true;      // T must support operator==(). Its trait
        return false;
    }
private:
    vector<T> items;                               // T must support T(), ~T()), T(const t&) or move
};                                                 // Its traits

#endif // __LIST_H
```

- List is basic container class

Templates and Inheritance: Example (Set.h)


```
#ifndef __SET_H
#define __SET_H
#include "List.h"

template<class T>
class Set { public:
    Set()   { };
    virtual ~Set()   { };
    virtual void add(const T &val);
    int length();                    // List<T>::length()
    bool find(const T &val);         // List<T>::find()
private:
    List<T> items;                   // Container List<T>
};
template<class T>
void Set<T>::add(const T &val) {
    if (items.find(val)) return;     // Don't allow duplicate
    items.put(val);
}
template<class T> int Set<T>::length() { return items.length(); }
template<class T> bool Set<T>::find(const T &val) { return items.find(val); }
#endif // __SET_H
```

- Set is a base class for a set
- Set uses List for container

Templates and Inheritance: Example (BoundSet.h)

```
#ifndef __BOUND_SET_H
#define __BOUND_SET_H

#include "Set.h"

template<class T>
class BoundSet: public Set<T> {
    public:
        BoundSet(const T &lower, const T &upper);
        void add(const T &val); // add() overridden to check bounds
    private:
        T min;
        T max;
};

template<class T> BoundSet<T>::BoundSet(const T &lower, const T &upper): min(lower), max(upper) { }
template<class T> void BoundSet<T>::add(const T &val) {
    if (find(val)) return;              // Set<T>::find()
    if ((val <= max) && (val >= min))  // T must support operator<=() and operator>=(). Its trait
        Set<T>::add(val);               // Uses add() from parent class
}
#endif // __BOUND_SET_H
```

- BoundSet is a specialization of Set
- BoundSet is a set of bounded items

Templates and Inheritance: Example (Bounded Set Application)


```
#include <iostream>
using namespace std;
#include "BoundSet.h"

int main() {
    BoundSet<int> bsi(3, 21);                       // Allow values between 3 and 21
    Set<int> *setptr = &bsi;

    for (int i = 0; i < 25; i++)
        setptr->add(i);                              // Set<T>::add(const T&) is virtual

    if (bsi.find(4))                                 // Within bound
        cout << "We found an expected value\n";
    if (!bsi.find(0))                                // Outside lower bound
        cout << "We found NO unexpected value\n";
    if (!bsi.find(25))                               // Outside upper bound
        cout << "We found NO unexpected value\n";
}
```

```
We found an expected value
We found NO unexpected value
We found NO unexpected value
```

- Uses BoundSet to maintain and search elements

Module Summary

- Introduced the templates in C++
- Discussed class templates as generic solution for data structure reuse
- Explained partial template instantiation and default template parameters
- Demonstrated templates on inheritance hierarchy
- Illustrated with examples

Programming in Modern C++

Module M40: Functors: Function Objects

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Discussed class templates as generic solution for data structure reuse
- Explained partial template instantiation and default template parameters
- Demonstrated templates on inheritance hierarchy
- Illustrated with examples

Module Objectives

- Understand the Function Objects or Functor
- Study the utility of functor in design, especially in STL

Module Outline

Callable Entities

Callable Entities in C / C++

- A Callable Entity is an object that
  - Can be called using the function call syntax
  - Supports *Function Call Operator*: `operator()`
- Such objects are often called
  - A *Function Object* or
  - A *Functor*

Functors

Some authors distinguish between *Callable Entities*, *Function Objects* and *Functors*, but we will treat these terminology equivalently depending on the context

Several Callable Entities C++

- *Function-like Macros*
- *C Functions* (Global or in Namespace)
- *Member Functions*
  - Static
  - Non-Static
- *Pointers to Functions*
  - C Functions
  - Member Functions (static / Non-Static)
- *References to functions*: Acts like const pointers to functions
- *Functors*: Objects that define `operator()`

Function Pointers

Function Pointers

- *Points to the address of a function*
  - Ordinary C functions
  - Static C++ member functions
  - Non-static C++ member functions
- *Points to a function with a specific signature*
  - List of Calling Parameter Types
  - Return-Type
  - Calling Convention

Function Pointers in C

- *Define a Function Pointer*

```
int (*pt2Function) (int, char, char);
```

- *Calling Convention*

```
int DoIt (int a, char b, char c); // __cdecl, __stdcall used in MSVC
int DoIt (int a, char b, char c) {
    printf ("DoIt\n");
    return a+b+c;
}
```

- *Assign Address to a Function Pointer*

```
pt2Function = &DoIt; // OR
pt2Function = DoIt;
```

- *Compare Function Pointers*

```
if (pt2Function == &DoIt) {
    printf ("pointer points to DoIt\n");
}
```

- *Call the Function pointed by the Function Pointer*

```
int result = (*pt2Function) (12, 'a', 'b');
```

Function Pointers in C

Direct Function Pointer	Using typedef

Direct Function Pointer

```
#include <stdio.h>

int (*pt2Function) (int, char, char);
int DoIt (int a, char b, char c);

int main() {
    pt2Function = DoIt; // &DoIt

    int result = (*pt2Function)(12, 'a', 'b');

    printf("%d", result);

    return 0;
}
int DoIt (int a, char b, char c) {
    printf ("DoIt\n");

    return a + b + c;
}
```

```
DoIt
207
```

Using typedef

```
#include <stdio.h>

typedef int (*pt2Function) (int, char, char);
int DoIt (int a, char b, char c);

int main() {
    pt2Function f = &DoIt; // DoIt

    int result = f(12, 'a', 'b');

    printf("%d", result);

    return 0;
}
int DoIt (int a, char b, char c) {
    printf ("DoIt\n");

    return a + b + c;
}
```

```
DoIt
207
```

Function Reference In C++

- *Define a Function Pointer*

```
int (A::*pt2Member)(float, char, char);
```

- *Calling Convention*

```
class A {
int DoIt (float a, char b, char c) {
    cout << "A::DoIt" << endl; return a+b+c; }
};
```

- *Assign Address to a Function Pointer*

```
pt2Member = &A::DoIt;
```

- *Compare Function Pointers*

```
if (pt2Member == &A::DoIt) {
    cout <<"pointer points to A::DoIt" << endl;
}
```

- *Call the Function pointed by the Function Pointer*

```
int result = (*this.*pt2Member)(12, 'a', 'b');
```

Function Pointer: Operations and Programming Techniques

- **Operations**
  - *Assign* an Address to a Function Pointer
  - *Compare* two Function Pointers
  - *Call* a Function using a Function Pointer
  - *Pass* a Function Pointer as an Argument
  - *Return* a Function Pointer
  - *Arrays* of Function Pointers
- **Programming Techniques**
  - *Replacing switch/if-statements*
  - *Realizing user-defined late-binding*, or
    - Functions in Dynamically Loaded Libraries
    - Virtual Functions
  - *Implementing callbacks*

Function Pointers: Replace Switch/ IF Statements

Solution Using switch

```
#include <iostream>
using namespace std ;
// The four arithmetic operations
float Plus(float a, float b) { return a+b ; }
float Minus(float a, float b) { return a-b ; }
float Multiply(float a, float b) { return a*b; }
float Divide(float a, float b) { return a/b ; }
void Switch(float a, float b, char opCode) {
    float result;
    switch (opCode) { // execute operation
      case '+': result = Plus(a, b); break;
      case '-': result = Minus(a, b); break;
      case '*': result = Multiply(a, b); break;
      case '/': result = Divide(a, b); break;
    }
    cout << "Result of = "<< result << endl;
}
int main() { float a = 10.5, b = 2.5 ;
    Switch(a, b, '+');
    Switch(a, b, '-');
    Switch(a, b, '*');
    Switch(a, b, '/');
    return 0;
}
```

Solution Using Function Pointer

```
#include <iostream>
using namespace std ;
// The four arithmetic operations
float Plus(float a, float b)
    { return a+b; }
float Minus(float a, float b)
    { return a-b; }
float Multiply(float a, float b)
    { return a*b; }
float Divide(float a, float b)
    { return a/b; }
// Solution with Function pointer
void Switch (float a, float b,
    float (*pt2Func)(float, float)) {
    float result = pt2Func(a, b);
    cout << "Result := " << result << endl;
}
int main() { float a = 10.5, b = 2.5 ;
    Switch(a, b, &Plus);
    Switch(a, b, &Minus);
    Switch(a, b, &Multiply);
    Switch(a, b, &Divide);
    return 0;
}
```

Function Pointers: Late Binding / Dynamically Loaded Library

- A C Feature in Shared Dynamically Loaded Libraries

Program Part-1

```
#include <dlfcn.h>
int main() {
    void* handle = dlopen("hello.so", RTLD_LAZY);
    typedef void (*hello_t)();

    hello_t myHello = 0;
    myHello = (hello_t)dlsym(handle, "hello");
    myHello();

    dlclose(handle);
}
```

Program Part-2

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

extern "C" void hello() {
    cout << "hello" << endl;
}
```

Function Pointers: Late Binding / Virtual Function

- A C++ Feature for Polymorphic Member Functions

Code Snippet Part-1

```
class A {
    public:
        void f();
        virtual void g();
};

class B: public A {
    public:
        void f();
        virtual void g();
};
```

Code Snippet Part-2

```
int main() {
    A a;
    B b;
    A *p = &b;

    a.f();  // A::f()
    a.g();  // A::g()
    p->f(); // A::f()
    p->g(); // B::g()
}
```

Example: Callback, Function Pointers

- It is a Common C Feature

```
// Application
extern void (*func)();
void f() { }
int main() {
    func = &f;
    g();
}

// Library
void (*func)();
void g() {
    (*func)();
}
```

Function Pointers: Callback Illustration (Step 1)


```
// Application
extern void (*func)();
void f()
{


}

void main()
{
    func = &f;

    g();
}
```

```
// Library
void (*func)();

void g()
{

    (*func)();

}
```

Function Pointers: Callback Illustration (Step 2)


```
// Application
extern void (*func)();
void f()
{

}

void main()
{
    func = &f;

    g();
}
```

```
// Library
void (*func)();

void g()
{

    (*func)();

}
```

Function Pointers: Callback Illustration (Step 3)

```
// Application
extern void (*func)();
void f()
{


}

void main()
{
    func = &f;

    g();
}
```

```
// Library
void (*func)();

void g()
{

    (*func)();

}
```

Function Pointers: Callback Illustration (Step 4)


```
// Application
extern void (*func)();
void f()
{

}

void main()
{
    func = &f;

    g();
}
```

Callback

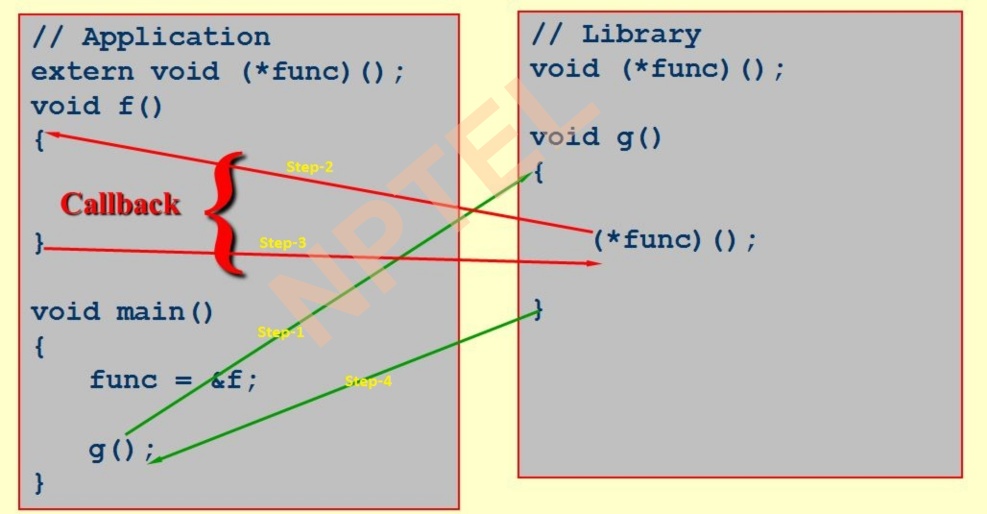
```
// Library
void (*func)();

void g()
{

    (*func)();

}
```

Function Pointers: Callback Illustration (Step-Final)

```
// Application
extern void (*func)();
void f()
{


}

void main()
{
    func = &f;

    g();
}
```

```
// Library
void (*func)();

void g()
{

    (*func)();

}
```

Function Pointers: Callback Illustration: Whole Process



Function Pointers: Callback: qsort to Quick Sort

```
void qsort(void *base,    // Pointer to the first element of the array to be sorted
           size_t nitems, // Number of elements in the array pointed by base
           size_t size,   // Size in bytes of each element in the array
           int (*compar)(const void *, const void*)); // Function that compares two elements

int CmpFunc(const void* a, const void* b) { // Compare function for int
    int ret = (*(const int*)a > *(const int*) b)? 1:
                  (*(const int*)a == *(const int*) b)? 0: -1;
    return ret;
}

int main() {
    int field[10];

    for(int c = 10; c>0; c--)
        field[10-c] = c;

    qsort((void*) field, 10, sizeof(field[0]), CmpFunc);
}
```

Function Pointers: Issues

- No value semantics
- Weak type checking
- Two function pointers having identical signature are necessarily indistinguishable
- No encapsulation for parameters

Functors in C++

Functors or Function Objects

- Smart Functions
  - Functors are *functions with a state*
  - Functors *encapsulate C / C++ function pointers*
    - Uses templates and
    - Engages polymorphism
- Has its own *Type*
  - A class with zero or more private members to store the state and an overloaded `operator()` to execute the function
- Usually *faster* than ordinary Functions
- Can be used to implement *callbacks*
- Provides the basis for *Command Design Pattern*

Basic Functor

- Any class that overloads the function call operator:
  - `void operator()();`
  - `int operator()(int, int);`
  - `double operator()(int, double);`
  - `...`

Functors: Simple Example

- Consider the code below

```
int AdderFunction(int a, int b) { // A function
    return a + b;
}

class AdderFunctor {
public:
    int operator()(int a, int b) { // A functor
        return a + b;
    }
};

int main() {
    int x = 5;
    int y = 7;
    int z = AdderFunction(x, y);  // Function invocation

    AdderFunctor aF;
    int w = aF(x, y);             // aF.operator()(x, y); -- Functor invocation
}
```

Functors: Examples from STL: Function Pointer for Functor

- **Fill a vector with random numbers**
  - **generate** algorithm

```
#include <algorithm>
template <class ForwardIterator, class Generator>
    void generate(ForwardIterator first, ForwardIterator last, Generator gen) {
        while (first != last) {
            *first = gen();
            ++first;
        }
    }
```

  - ▷ **first, last**: Iterators are defined for a range in the sequence. "[" or "]" means include the element and "(" or ")" means exclude the element. `ForwardIterator` has a range `[first,last)` spanning from `first` element to the element before the `last`
  - ▷ **gen**: Generator function that is called with no arguments and returns some value of a type convertible to those pointed by the iterators
  - ▷ This can either be a function pointer or a function object
  - Function Pointer **rand** as Function Object

```
#include <cstdlib>

// int rand (void);

vector<int> V(100);
generate(V.begin(), V.end(), rand);
```

Functors: Examples from STL: Functor without a state

- **Sort a vector of double by magnitude**
  - sort algorithm

```
#include <algorithm>
template <class RandomAccessIterator, class Compare>
    void sort (RandomAccessIterator first, // Simple interface
               RandomAccessIterator last,  // Difficult to use incorrectly
               Compare comp); // Compare Functor
```

  - **first, last**: RandomAccessIterator has a range [first,last)
  - RandomAccessIterator shall point to a type for which swap is properly defined and which is both move-constructible and move-assignable (C++11)
  - **comp**: Binary function that accepts two elements in the range as arguments, and returns a value convertible to bool. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific strict weak ordering it defines.
  - The function shall not modify any of its arguments
  - This can either be a function pointer or a function object

Functors: Examples from STL: Functor without a state

- **Sort a vector of double by magnitude**

Using qsort in C with User-defined **Function** less_mag

```
#include <stdlib.h>
// Compare Function pointer
void qsort(void *base,
     size_t nitems,
     size_t size,
     int (*compar)(const void *, const void*))
// Complicated interface. Difficult to use correctly

// Type-unsafe comparison function
// Intricate and error-prone with void*
int less_mag(const void* a, const void* b) {
    return (fabs(*(const double*)a) <
            fabs(*(const double*)b) ? 1: 0;
}

double V[100]; // Capacity = 100
// 10 elements are filled - needs to be tracked

// Difficult to call
qsort((void*) V, 10, sizeof(V[0]), less_mag);
```

Using sort in C++ with User-defined **Functor** less_mag

```
#include <algorithm>
// Compare Functor
template <class RandomAccessIterator, class Compare>
    void sort (RandomAccessIterator first,
               RandomAccessIterator last,
               Compare comp);
// Simple interface. Difficult to use incorrectly

// Type-safe comparison functor
struct less_mag: public
    binary_function<double, double, bool> {
        bool operator()(double x, double y)
        { return fabs(x) < fabs(y); }
};

vector<double> V(100);
// 10 elements are filled tracked automatically

// Easy to call
sort(V.begin(), V.end(), less_mag());
```

Functors: Examples from STL: Functor with a state

- **Compute the sum of elements in a vector**
  - **for_each** algorithm

```
#include <algorithm>
template<class InputIterator, class Function>
    Function for_each(InputIterator first, InputIterator last, Function fn) {
        while (first!=last) {
            fn (*first);
            ++first;
        }
        return fn;      // or, since C++11: return move(fn);
    }
```

    - **first, last**: `InputIterator` has a range `[first,last)`
    - **fn**: Unary function that accepts an element in the range as argument
    - This can either be a function pointer or a move constructible function object (C++11)
    - Its return value, if any, is ignored.
  - User-defined Functor **adder** with local state

```
    struct adder: public unary_function<double, void> { adder() : sum(0) { }
        double sum; // Local state
        void operator()(double x) { sum += x; }
    };
    vector<double> V;
    ...
    adder result = for_each(V.begin(), V.end(), adder());
    cout << "The sum is " << result.sum << endl;
```

Module Summary

- Introduced Function Objects or Functors
- Illustrated functors with several simple examples and examples from STL