Module 36: Programming in C++

Exceptions (Error handling in C)

Intructors: Abir Das and Sourangshu Bhattacharya

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

{*abir, sourangshu*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in Modern C++

by **Prof. Partha Pratim Das**

Module Objectives

- Understand the Error handling in C

Module Outline

1. Exception Fundamentals
   - Types of Exceptions
   - Exception Stages
2. Error Handling in C
   - C Language Features
     - Return Value & Parameters
     - Local `goto`
   - C Standard Library Support
     - Global Variables
     - Abnormal Termination
     - Conditional Termination
     - Non-Local `goto`
     - Signals
   - Shortcomings
3. Module Summary

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

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 */
}
```

Support for Error Handling in C

- Support for Error Handling in C
  - C language does not provide any specific feature for error handling. Consequently, 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);
}
```

Global Variables

- 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

```
/* 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

```
#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;
}
```

Callee

```
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

```
int main() {
    if (setjmp(jbuf) == 0) {
        printf("g() called\n");
        g();
        printf("g() returned\n");
    }
    else
        printf("g() failed\n");
    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;
}
```

(1) g() called

(2) g() successfully returned

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

Example: Non-Local goto: The Dynamics

Caller

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

Callee

```
jmp_buf jbuf;

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

(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
- **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