Hochiminh City University of Technology
Computer Science and Engineering
OOP in C++

Advanced Techniques

Lecturer:
Tran Huy
Vu Van Tien

Today's outline

- Exception Handling
- Debugging

Exception Handling

Introduction

- An exception is an occurrence of an undesirable situation that can be detected during program execution.
- There are situations when an exception occurs, but you don’t want the program to simply ignore the exception and terminate.

Case-study

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

int main()
{
    int dividend, divisor, quotient;
    cout << "Enter dividend: ";
    cin >> dividend;
    cout << "Enter divisor: ";
    cin >> divisor;
    quotient = dividend / divisor;
    cout << "Quotient = " << quotient << endl;
    return 0;
}
```

Basic approach

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

int main()
{
    int dividend, divisor, quotient;
    cin >> dividend >> divisor;
    if (divisor != 0) {
       quotient = dividend / divisor;
       cout << "Quotient = " << quotient << endl;
    }
    else cout << "Cannot divide by zero." << endl;
    return 0;
}
```

assert

- **assert** function is useful in stopping program execution when certain elusive errors occur.
- Use <cassert> / <assert.h> library.
- The syntax:
  - `assert(<expression>);`
  - <expression> == true: next statement executes.
  - <expression> == false: the program terminates.

Example

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

int main()
{
    int dividend, divisor, quotient;
    cin >> dividend >> divisor;
    assert(divisor != 0);
    quotient = dividend / divisor;
    cout << "Quotient = " << quotient << endl;
    return 0;
}
```

try/catch Block

- **Problems?** The above approach didn't **handle** the exceptions.
- C++ proposes a dedicate way to handle exceptions within program by introducing keywords: `try`, `catch` and `throw`.
  - The statements that may generate an exception are placed in a `try` block.
  - The `try` block is followed by one or more `catch` blocks.
  - A `catch` block specifies the type of exception it can catch and contains an exception handler.

try/catch Syntax

```
try {
   //statements
}
catch (dataType1 id) {
   //exception-handling code
}
.
.
.
catch (dataTypen id) {
   //exception-handling code
}
catch (...) {
   //exception-handling code
}
```

- If no exception is thrown in a `try` block, all catch blocks are ignored and program continue executing.
- If an exception is thrown, the remaining statements in that `try` block are ignored. The program searches the appropriate exception handler (`catch` block).
- The last `catch` block that has an **ellipses** (three dots) is designed to catch **any** type of exception.

throw an Exception

- In order for an exception to occur in a `try` block and be caught by a `catch` block, the exception must be **thrown** in the `try` block.
- The syntax:
  - `throw(<expression>);`
- E.g.:
  - `throw 4;`
  - `throw x;`
  - `throw string("Exception found!");`

Example

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

int main()
{
    int dividend, divisor, quotient;
    try {
        cin >> dividend >> divisor;
        if (divisor == 0) throw 0;
        quotient = dividend / divisor;
        cout << "Quotient = " << quotient << endl;
    }
    catch (int) {
        cout << "Division by 0." << endl;
    }
    return 0;
}
```

C++ Exception classes

- C++ provides support to handle exceptions via a hierarchy of classes.
  - `class exception:` the base exception handling class.
  - This class contains the function **`what`** which return a string containing a appropriate message.
  - Some common C++ provided exception classes:
    - `class logic_error: invalid_argument, out_of_range, bad_alloc`
    - `class runtime_error: overflow_error, underflow_error`

Example

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

int main()
{
     string sentence;
     try {
          sentence = "Testing string exceptions!";
          cout << sentence << endl;
          cout << sentence.substr(8, 20) << endl;
          cout << sentence.substr(28, 10) << endl;
     }
     catch (out_of_range re) {
          cout << "In the out_of_range catch block: " << re.what() << endl;
     }
     return 0;
}
```

More Example

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

int main()
{
     int *list[100];
     try {
         for (int i = 0; i < 100; i++) {
           list[i] = new int[5000000];
         }
         cout << "Created list[" << i
              << "] of 5000000 components." << endl;
     }
     catch (bad_alloc be) {
         cout << "In the bad_alloc catch block: " << be.what() << endl;
     }
     return 0;
}
```

User-defined Exception Class

- **Problems?** C++ cannott provide all exception classes to deal with all your situations
- **Solution:** You can write your own exception classes!
  - Syntax: Simply a class.
  - E.g.: `class dummyExceptionClass {};`
  - You can also inherit from the `class exception`.

User-defined Exception Class

- **Problems?** C++ cannott provide all exception classes to deal with all your situations
- **Solution:** You can write your own exception classes!
  - Syntax: Simply a class.
  - E.g.: `class dummyExceptionClass {};`
  - You can also inherit from the `class exception`.

Basic Example

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

class divisionByZero {};

int main()
{
    int dividend, divisor, quotient;
    try {
        cin >> dividend >> divisor;
        if (divisor == 0) throw divisionByZero();
        quotient = dividend / divisor;
        cout << "Quotient = " << quotient << endl;
    }
    catch (divisionByZero) {
        cout << "Division by 0." << endl;
    }
    return 0;
}
```

Complex Example

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

class divisionByZero {
public:
    divisionByZero() {
        message = "Division by zero ";
    }
    divisionByZero(string str) {
        message = str;
    }
    string what() {
        return message;
    }
private:
    string message;
};
```

```
int main()
{
    int dividend, divisor, quotient;
    try {
        cin >> dividend >> divisor;
        if (divisor == 0) throw divisionByZero();
        quotient = dividend / divisor;
        cout << "Quotient = " << quotient << endl;
    }
    catch (divisionByZero divByZeroObj) {
        cout << "In the divisionByZero catch block: "
             << divByZeroObj.what() << endl;
    }
    return 0;
}
```

Exception Handling Techniques

- When an exception occurs in a program, the programmer usually has **three** choices:
  - **Terminate the Program**
  - **Fix the Error and Continue**
  - **Log the Error and Continue:** If your program need to be continuously running, then it's a good behavior that the program report the exception and continue to run.

Fix the Error and Continue

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

int main()
{
      int number;
      bool done = false;
      do {
          try {
               cout << "Enter an integer: ";
               cin >> number;
               if (!cin) throw string("Invalid input");
               done = true;
               cout << "Number = " << number << endl;
          }
          catch (string messageStr) {
               cout << messageStr << endl;
               cout << "Restoring the input stream." << endl;
               cin.clear();
               cin.ignore(100, '\n');
          }
      } while(!done);
      return 0;
}
```

More

- When an exception occurs in a program, the programmer usually has **three** choices:
  - **Terminate the Program**
  - **Fix the Error and Continue**
  - **Log the Error and Continue:** If your program need to be continuously running, then it's a good behavior that the program report the exception and continue to run.

Debugging

Outline

- Debugger
- Debugger's Features
- Basic Debugger Operations
- Basic Debugger Commands

Debugger

- A program that is used to run other programs
- Provide commands in order that program may be examined

Debugger's Features

- Check the order of statement execution in a program.
- Evaluate a variable, parameter, or (possibly) an expression at a particular point during program execution.
- Determine where execution was occurring in a program when a program crashes, even if the crash was in a library.
- Stop a program's execution at a particular statement.

Basic Debugger Operations

- Setting a Breakpoint
  - To cause a program that is executing in the debugger to suspend itself, permitting debugging.
- Resume Execution
  - The program will execute until it terminates, or encounters a breakpoint or other suspend directive.

Basic Debugger Operations

- *Execute a statement*
  - Step over:
    - Execute the present statement.
    - If the statement is a function call, the function will be executed; the debugger will stop at the statement after the function call.

Basic Debugger Operations

- *Execute a statement*
  - Step into:
    - Execute the present statement.
    - If the statement is a function call, the function will be debugged; the debugger will stop at the first statement inside the function.
    - Note: if you perform a step into on a ***library*** function call, the debugger may attempt to step into the file containing the library function. Use of step into with function calls should be limited to functions you wrote.

Basic Debugger Operations

- *Execute a statement*
  - Step out:
    - Resumes execution until the function the program is executing terminates; the debugger will stop at the statement after the function call.
- *Watching Variables*
  - Help to examine the value of a variable (via a watch)

Enable Debugging with GDB

- Use the –g flag during compilation and linking of all files comprising your program
- The –g flag causes the compiler and linker to maintain variable names and untranslated C++ statements, in order that variables can be examined and C++ statements can be displayed and followed.

Start GDB

- Syntax: gdb <execution>
- Example: gdb ./main
- Debug in TUI (Text User Interface) Mode: gdb –tui <execution>

Basic GDB COMMANDS

- Essential Commands

break function_name **break line_number**	Set breakpoint at function Set breakpoint at line number
run [arglist]	Start program [with arglist]
bt (backtrace)	backtrace: display program stack
print expr	display the value of an expression
continue	continue running your program
next	next line, stepping over function calls
step	next line, stepping into function calls
finish	run until selected stack frame returns, stepping out

Basic GDB COMMANDS

- Source Files

list	show next ten lines of source
list -	show previous ten lines
list *lines*	display source surrounding lines
list *f, l*	show lines from line f to line l

Basic GDB COMMANDS

- Display

print ***expr***	show value of expr

- Automatic Display

display ***expr***	show value of expr each time program stops
display	display all enabled expressions on list
undisplay ***n***	remove number(s) n from list of automatically displayed expressions
info display	numbered list of display expressions
disable disp ***n***	disable display for expression(s) number n
enable disp ***n***	enable display for expression(s) number n

References

- A Brief Introduction to GDB, including its use within emacs, *Professor Daniel Spiegel*, Kutztown University, link (last access: 20/08/2022): https://faculty.kutztown.edu/spiegel/Debugging/DebugPrimer.html
- GDB QUICK REFERENCE, link (last access: 20/08/2022): gdb-reference-card.pdf (umd.edu)