

Comp 151

Exception Handling

Terminology

- **Exceptions** are run-time anomalies that a program may detect
 - division by 0,
 - access to an array outside of its bounds,
 - exhaustion of the heap memory.
- An exception is an unusual event, and may require special processing
- The special processing required after detection of an exception is called **exception handling**
- The exception handling code unit is called an **exception handler**

Introduction to Exception Handling

- Traditional approaches in case of exception:
 - Terminate the program: `exit()`, `abort()`
 - Return special values to indicate errors in a function
 - Set global error bits and return normally (leaving the system in an illegal state)
- Error detection and handling code is tedious to write; it clutters the program and is error-prone
- The C++ language provides built-in features to raise and handle exceptions:
 - Separate error-handling code from ordinary code
 - Exception propagation allows a high level of reuse of exception handling code
 - Release local resources automatically

An Example

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

int main () {
    try {
        throw 20; // throw an exception
    } catch (int e) {
        cout << "Exception No. " << e << endl;
    }
    return 0;
}
```

General Form

```
try {  
    -- code that is expected to raise an  
    exception  
}  
// Each catches one type of exception  
catch (type1 var1) {           // var is optional  
    -- handler code  
}  
...  
catch (type2 var2) {  
    -- handler code  
}
```

try

- Put statements and function calls that may generate exceptions in a **try** block
- Each try block is associated with a sequence of handlers that follow immediately
- **try** blocks can be *nested*

```
try {  
    try {  
        f(); // f() may throw an exception  
    } catch (int e) {  
        cout << "Exception No. " << e << endl;  
    }  
} catch(double) { cout << "Caught double." << endl; }
```

throw

- An exception is raised using a **throw expression**, composed of **throw** followed by an object whose type is that of the exception thrown
- Any object (built-in or user-defined) can be thrown

```
class to_be_thrown {};
```

```
...
```

```
throw to_be_thrown; // error, not an object
```

```
throw to_be_thrown(); // correct
```

```
throw 2.5; // correct, double
```

catch: The Handler

- **catch** is the name of all **handlers**
 - must immediately follow the try block
 - the formal parameter of each handler must be *unique*
 - no automatic type conversion
- The formal parameter does not have to be a variable
 - Can be simply a type name to distinguish its handler from others
 - A variable transfers information to the handler

catch: The Handler

- The formal parameter can be an ellipsis, in which case it handles all exceptions not yet handled

```
catch (...) { // catches everything
    -- handler code
}
```

- After a handler completes, control flows to the first statement after the last handler in the sequence
- When no exception occurs, all handlers are neglected (no performance loss)

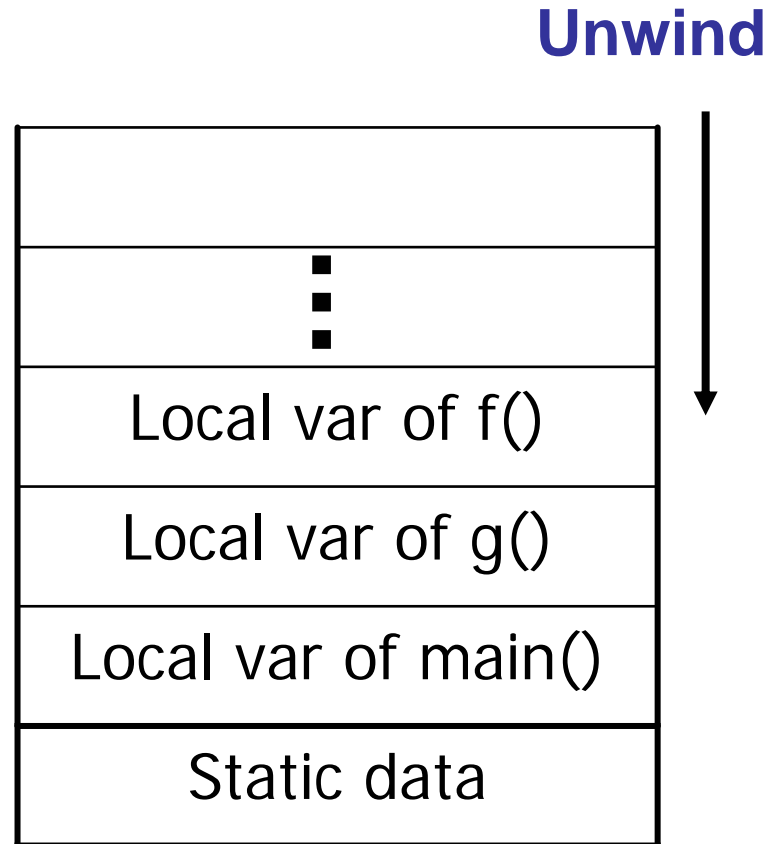
Propagation/Stack Unwinding

- Exceptions CANNOT be ignored
- If not caught by handlers right after **try** block, exception moves to next-higher level and may be caught there:
 - The next level of try block (if nested)
 - Try block surrounding the function call in which exception occurs
 - If no handler at any level catches the exception, **terminate()** will be called and program will terminate
- Passing an exception while searching for a handler can cause abnormal exit from a function while in middle of executing it (i.e., **without** any return value)
 - The stack frame corresponding to the exited function's scope is popped – this is called **stack unwinding**
 - So the lifetime of local objects in the exited functions ends
 - **C++ still guarantees correct destructors are called**

An Example

```
void f() {  
    Person p;  
    throw 20;  
}  
void g() {Person g; f();}  
void main(){  
    try {  
        g();  
    } catch(int) {  
        cout<<"error"<<endl;  
    }  
}
```

**call
function**



**Snapshot of
Memory Stack**

Release Your Own Resource

- Stack unwinding does not automatically delete pointers or close file handles. These should be handled locally.

```
void func() {
    resource res; res.lock();
    try {
        // use resource
        // some action throws an exception
    } catch (...) {
        res.release();
        throw; // re-throw the exception
    }
    res.release(); // skipped if exception thrown
}
```

Standard Exceptions

- All standard exception classes derive ultimately from the class `exception`, defined in the header `<exception>`.
- `logic_error` and `runtime_error` are derived from `exception` and are defined in `<stdexcept>`
- A handler for base class objects can also catch derived class objects
- Define your own error from standard exception classes

```
class DivideByZeroError : public runtime_error {  
public:  
    DivideByZeroError(const string& msg = "")  
        : runtime_error(msg) {}  
};
```

An Example

```
#include <stdexcept>
#include <iostream>
#include "myerror.hpp"
using namespace std;

int divide_int(int numer, int denom) {
    if (denom == 0) throw DivideByZeroError("divide_int");
    return numer/denom;
}

int main() {
    try {
        cout << divide_int(1, 0) << endl;
    } catch (runtime_error &e) { // pass by ref
        cout << "Error caught in " << e.what() << endl;
    }
    return 0;
}
```

Catch `bad_alloc`

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

int main() {
    int* p[9999];
    try {
        for (int i = 0; i < 9999; i++) {
            p[i] = new int[99999999];
        }
    } catch(bad_alloc) { // don't bother with the thrown object
        cout << "Problem in getting memory" << endl;
    }
    return 0;
}
```

Exception Specification

When declaring functions...

- `void some_function() throw ();`
 - Promises that the function will not throw any exception
- `void some_function() throw(DivideByZero,
OtherException);`
 - Promises that the function may only throw the exceptions **DivideByZero** and **OtherException**
- `void some_function();`
 - No promises – any type of exception might be thrown from this function