

# C++ Exceptions and Assertions

Some content adapted from 'Absolute C++' by Walter Savitch

# Outline

- Exceptions Try, Catch and Throw
- How to throw
- How to catch
- Built ins and making your own
- Exception Specs
- Multiple catch blocks and (...)
- Special cases (Con/De structors)
- Assert

# Exceptions

- Similar to Java syntactically

```
• int x, y;  
cin >> x >> y;  
• try  
{  
    if (y == 0) throw x;  
    cout << "x/y is " << x/y << endl;  
}  
catch(int &num) {  
    cout << "Div by 0 error when " << "dividing" << num << " by 0\n";  
}
```

- try – use when you are not sure something will work
- catch – what to do if it doesn't
- throw – how to indicate something went wrong

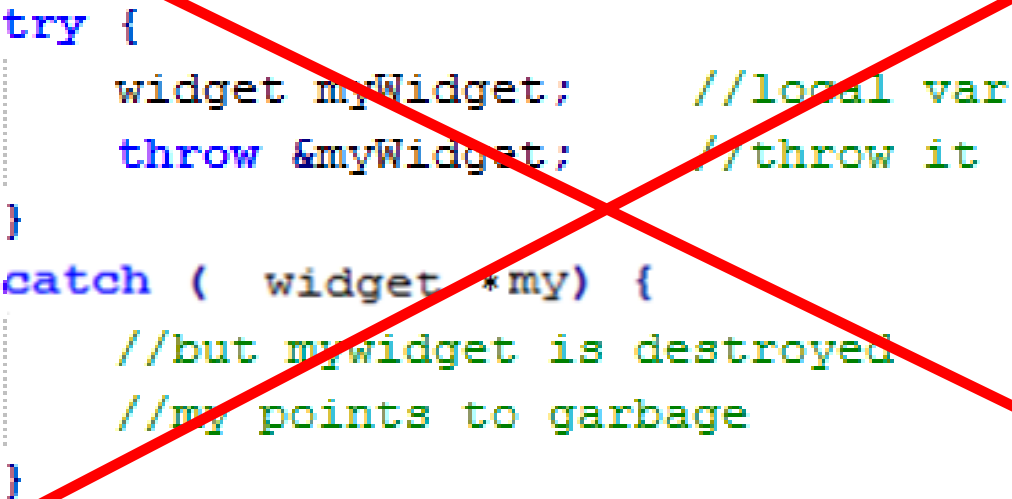
# Exceptions – throw and rethrowing

```
try {  
    widget myWidget;    //local var  
    throw myWidget;    //throw it  
}  
catch ( widget &my) {  
    throw;    //rethrow my — Prefer  
}  
catch ( widget &my) {  
    throw my;    //rethrow a COPY of my  
}
```

- Doesn't myWidget go out of scope?
- No. C++ specifies that an object thrown as an exception is **always copied**. (uses copy constructor)
- Prefer the first rethrow method, it does not incur the cost of a second copy

# Exceptions – catch by pointer, value or reference?

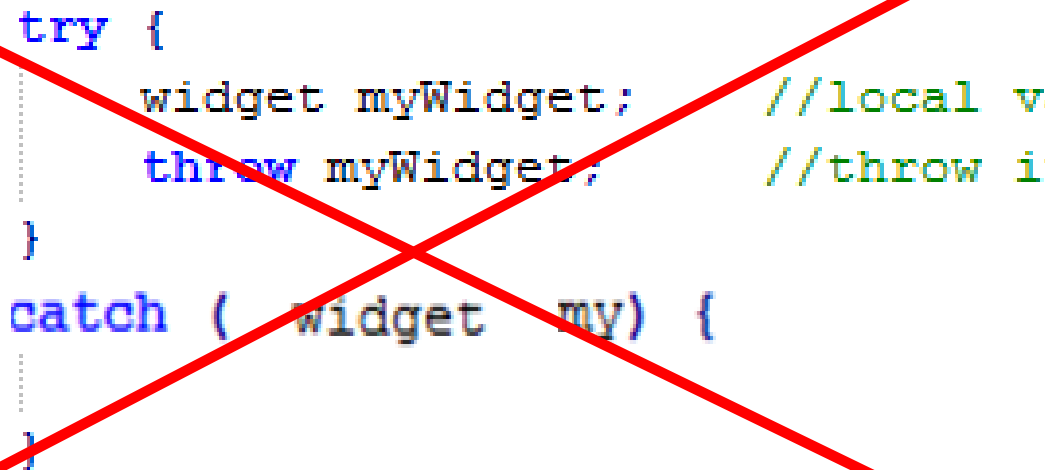
- Can do all 3, only 1 works well though
- Pointer: copy of pointer passed, what if object pointed to goes out of scope?



```
try {  
    widget myWidget;    //local var  
    throw &myWidget;    //throw it  
}  
catch ( widget *my) {  
    //but myWidget is destroyed  
    //my points to garbage  
}
```

# Exceptions – catch by value?

- value: works mostly, but need to make 2 copies of exception: once when thrown, once when caught.

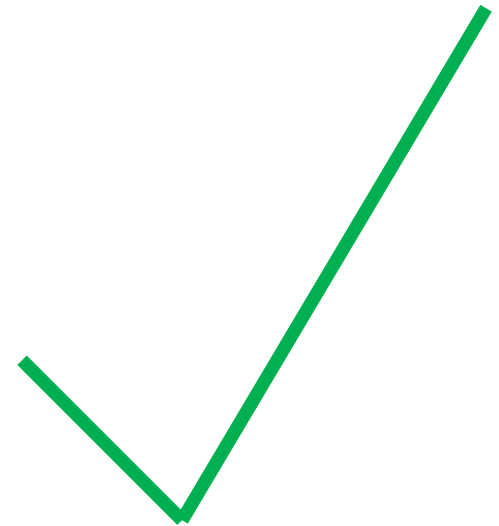


```
try {  
    widget myWidget;           //local var  
    throw myWidget;           //throw it  
}  
catch ( widget my) {
```

# Exceptions – catch by reference?

- Reference: works, single copy made when thrown

```
try {  
    widget myWidget;    //local var  
    throw myWidget;    //throw it  
}  
catch ( widget &my) {  
  
}
```



# Built in Exceptions

- Standard library exceptions:
- Can be thrown by many library constructs

Exception	description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when fails with a referenced type
bad_exception	thrown when an exception type doesn't match any catch
bad_typeid	thrown when dynamic cast a null pointer
ios_base::failure	thrown by functions in the iostream library



# Multiple Catch Blocks

- Arrange catch blocks from specific to general
- First match is the first caught
- `catch(...)` catches all exceptions, needs to be last caught, or will be the only one caught

```
try {  
    widget myWidget;    //local var  
    throw myWidget;    //throw it  
}  
catch (widget &my) {  
    //catch my widget  
}  
catch(std::bad_alloc &myAlloc){  
    //catch bad allocation  
}  
catch(...){  
    //catch everything else  
}
```

# Custom Exceptions

- Two useful generic exception types for custom exceptions

<code>logic_error</code>	error related to the internal logic of the program
<code>runtime_error</code>	error detected during runtime

•

# Custom Exceptions

```
class myexception : public exception
{
    virtual const char* what() const throw()
    {
        return "exception happened";
    }
};

int main() {
    try
    {
        myexception myex;
        throw myex;
    }
    catch (exception& e)
    {
        cout << e.what() << '\n';
    }
    return 0;
}
```

# Dangerous Exceptions – Constructors, Destructors

- Don't let them leave constructor
  - C++ destroys only fully constructed objects, you throw an exception in your constructor your destructor is never called
- Don't let them leave destructor
  1. Destructor called when object goes out of scope or is deleted
  2. Also during stack unwinding part of exception propagation
- Cant tell which of the 2 is the case
- If control leaves a destructor due to an exception, while another exception is active, C++ calls the terminate function.

# ASSERTS

- Used to debug debug builds.
- Compile to null operations in release.
  - This is from <assert.h>. If NDEBUG is defined then in release mode.

```
#ifndef NDEBUG
/*
 * If not debugging, assert does nothing.
 */
#define assert(x)    ((void)0)
#else /* debugging enabled */
```

# ASSERTS- Debug

- Dont define NDEBUG
- To use;  
    `assert (myInt!=NULL);`
- If the expression in () evaluates to 0, causes an assertion failure that terminates the program
- Message to `std::err` with at least :  
the *expression* whose assertion failed, the name of the source file, and the line number where it happened.

# Summary -

- Use try catch when an error can be thrown by an API
- Catch by reference
- Use built in exceptions
- Don't let an exception leave constructor or destructor
- Use asserts to debug code, useless at runtime