

TCC CSCI-2843 C++ Programming Language

Lecture 9 Notes

String Streams

The standard provides stream types that work on internal memory instead of external devices. Because these buffers hold characters, these stream types are known as “string streams” and provide input and output facilities. To access these classes, use the following include

```
#include <sstream>
```

The std::ostringstream Class

The std::ostringstream class provides a method to perform output to an internal memory buffer that is immediately convertible to a std::string.

```
...  
ostringstream s;  
int result = 10;  
string label = "The result is ";  
s << label << result;  
string text = s.str();  
...
```

Once constructed, we can use an ostringstream as any other ostream (from which it inherits). When we are done, we can simply call the str member function which returns a string that has the contents of whatever we output to the stream. All standard formatting available with any other ostream applies to the ostringstream, making this a nice way to convert numeric data into their textual equivalent.

The std::istringstream Class

To perform input using an already-constructed string, use the std::istringstream class:

```
string data = "123 456 text";  
int i;  
int j  
string text;  
istringstream input(data);  
input >> i >> j >> text;
```

Once constructed, an istringstream provides all the input facilities of any other istream (from which it inherits). It also inherits all the limitations of any other istream (i.e. there is no high-level input formatting), but can be useful for turning textual representations of numeric data into their numeric representations or picking through a string as if it were coming from a file.

Exception Handling

The Old Way

Consider two popular methods that programmers historically have had to communicate errors from one level of a program to another:

- 1) A return value from a function.
- 2) A global variable that keeps a “status” that can be inspected by any and everyone.

An example of (1) would be checking the return value from an invocation of a stream operator:

```
if (!getline(stream, str, 256, '\n')) { ... error ... }
```

Another example would be the old fopen call to open a file which uses a null pointer to indicate “failure”.

```
FILE* fp = fopen("some_data.dat", "rw");  
if (fp == NULL) { ... file wasn't open ... }
```

An example of (2) above is the POSIX errno:

```
// include to gain access to the extern int "errno"  
#include <sys/errno.h>
```

```

errno = 0;
read(socket, buffer, sizeof(buffer));

if (errno != 0) { ... error ... }

```

Each one of the above methods has one common flaw: they **all** require explicit checks after the function call to be of any use. This means that there is **always** the opportunity for the caller of a function to **completely ignore** any status that is returned:

```

getline(stream, str, 256, '\n'); // return value not checked – operation COULD have failed
...
FILE* fp = fopen("some_data.dat", "rw");

fread(fp, buffer, sizeof(buffer)); // could cause the program to crash if the file was not open (fp is NULL)
...
read(socket, buffer, sizeof(buffer)); // socket could be closed or read could have failed

```

These types of methods all rely on “programmer discipline” to work properly, and we all know that THAT is an oxymoron.

Consider some other error detection methods themselves. When error checking is enforced, about a third to one-half the code can end up being error-handling related:

```

int f() {
    ifstream f("whatever.dat");
    if (!f) {
        ... handle error...
        return 1;
    }
    string str;
    for (;;) {
        if (!getline(f, str, 1024, '\n')) {
            if (!f.eof()) {
                break;
            }
            if (f.fail()) {
                ... handle failure ...
                return 2;
            }
            if (f.bad()) {
                ... handle “bad” stream
                return 3;
            }
        }
        ... “normal” processing
    }
}

```

Many programmers would rather forgo the error checking:

```

int f() {
    ifstream f("whatever.dat");
    string s;
    while (getline(f, str, 1024, '\n')) {
        ... “normal” processing
    }
    return 0;
}

```

While this is far fewer lines of code, it also is less robust and gives less-than-desirable diagnostics when something does go wrong, indeed it just bails from reading the stream at the first error **or** end of stream with **no** error detection whatsoever.

Note that even though in the first (robust) example, we can go through all the trouble of detecting the error and returning a “code” or “status”, but that is **no** guarantee that the caller of our function will ever pay any attention to it.

Speaking of convoluting code, one of my “favorite” methods is the “ok” flag method:

```
bool f() {
    bool ok = true;
    ... insert convoluted and twisted code here, setting ok to false if there is an error
    return ok;
}
```

The more convoluted the intervening code, the more the chance that the “ok” flag might be set inadvertently, or more likely, not set when it is supposed to be, giving false positives and negatives.

We would like a facility that would do three things:

- 1) Standardize on the way errors are detected and checked
- 2) **Require** that once an error is detected that it **must** be handled (i.e. **cannot** be ignored)
- 3) Reduce the syntax for detecting and handling errors, thus “unconvoluting” the code.

The C++ exception facility gives us all of these.

Exceptions

Like many modern languages, exceptions are based on a **try/throw/catch** model where we “**try**” an operation, and if there are errors, we “**throw**” an exception, which we “**catch**” with code that handles the error. The general syntax is

```
...
try {
    ...
    if (an error occurs) { throw some exception object; }
    ...
} catch (some exception object) {
    ... handle the exception
}
...
```

What can be thrown as an “exception object”? **Any** object, including primitive types (e.g. **int**, **double**) or objects constructed from classes. For instance, we can do the following:

```
...
try {
    ...
    if (an error occurs here) { throw 1; }
    ...
    if (an error occurs here) { throw 2; }
    ...
} catch (int code) {
    cout << “an error occurred: “ << code << endl;
}
...
```

We can even use strings:

```
...
try {
    ...
    if (there is an error) { throw “something’s wrong”; }
    ...
} catch (const string& s) {
    cout << s << endl;
}
...
```

We can have multiple catch blocks following a try block to catch multiple types of exceptions:

```

...
try {
    ...
    if (an error occurs here) { throw 1; }
    ...
    if (an error occurs here) { throw "something's wrong"; }
    ...
} catch (const string& s) {
    cout << s << endl;
} catch (int code) {
    cout << "an error occurred: " << code << endl;
}
...

```

This is similar to an **if/else if/else ...** such that depending on the type of the object that is **thrown**, the corresponding **catch** will be invoked to handle it. The important thing to note is that it will always pick the **first catch** that matches the type of the object **not the "best" type**.

```

...
try {
    ...
    if (an error occurs) { throw 1; }
    ...
} catch (double r) {
    ...
} catch (int i) {
    ...
}
...

```

This will always catch the **int**, 1, in the **double catch** clause because **double** is type-compatible with **int** and is the **first** that matches, even though there is a catch of **int** that matches **better**.

To catch **any** exception object, we can use the ... (literally, three dots) in the **catch**:

```

try {
    if (error) { throw 1; }
    if (another error) { throw "problem"; }
} catch (...) {
    cout << "something went wrong" << endl;
}

```

Since there is no name given to the object that is caught, you cannot query anything to help diagnose the problem. That is why this is usually used as a "last resort" or "catch-all" mechanism when you don't need to know or care about the specific exception. Obviously, this "catch-all" needs to go as the last catch clause when present, since it will match **any** object thrown.

Some things to note about exceptions:

- 1) **Any** object may be thrown as an exception object.
- 2) If the object is a non-trivial type, we catch it by a (not necessarily **const**) reference to prevent a copy.
- 3) Every object within the **try's** scope that has been fully constructed up to the point that an exception is **thrown** is destructed in the **opposite order** than they were constructed before control is transferred into the **catch** body.
- 4) Multiple **catch** clauses may be placed after a **try** block (there **must** be at least one), allowing us to catch multiple types of exceptions from one scope.
- 5) An exception object is always caught by the **first** catch clause that matches its type, **not the best** match of its type.
- 6) Use the **catch (...)** to catch **any** exception.

Unwinding

As mentioned before, objects within a **try** block that are constructed before an exception is thrown are destructed before a **catch** clause is entered. This is true even if an exception is thrown from one function and caught in another:

```
void f() {
    string u = "in f";
    ...
    if (some error) { throw 1; }
    ...
}

void g() {
    string s = "in g";
    try {
        string t = " in the try";
        ...
        f()
        ...
    } catch (int code) {
        cout << "code: " << code << endl;
    }
}
```

If `f` throws the `int` exception, the exception mechanism notes that there is no immediate **try/catch** blocks surrounding it, so therefore it will destruct **all** automatic objects (local variables) within `f` that have been constructed so far (e.g. the string, `u`) and **unwinds** the stack as if it was executing **return**, but instead of returning to the code immediately following the function call, keeps searching for a matching **catch** clause, destructing objects as needed.

In this case, it finds the catch of an `int`, but must destruct all of the automatic objects within **that try** block, which includes the string, `t`. If there were still no matching **catch** clause, **all** of `g`'s automatic objects would be destructed and `g` would be unwound and the search would continue. This stack **unwinding** process ensures that everything is cleaned up by the time a matching **catch** clause is entered.

What if there is never any matching **catch** clause found by time `main` is unwound? The program is terminated with an "uncaught exception" status. Since this is tantamount to a crash, it behooves us to **at least** provide a "last-chance" catch in `main`:

```
int main() {
    try {
        ... do all of your other processing
    } catch (...) {
        cerr << "Exception caught in main" << endl;
        return 1;
    }
}
```

Nontrivial Exception Objects

Remember that since we can throw **any** type of object, we can throw an object of our own type. We've already seen that we can throw a `std::string`, so why can't we build our own class that encompasses information about an exception? Why not indeed.

```
class my_exception {
private:
    string description;
    int code;
public:
    my_exception(const string& init_description, int init_code) :
        description(init_description),
        code(init_code)
```

```

    {}
    string get_description() const { return description; }
    int code() const { return code; }
};

```

This can be used like

```

...
try {
    ...
    if (some error) { throw my_exception("something's wrong", 1);
    ...
    if (some other error) { throw my_exception("something else is wrong", 2); }
    ...
} catch (my_exception& e) {
    cerr << e.get_description() << ": " << e.code() << endl;
}
...

```

Note the anonymous construction of our exception objects. This keeps copying to a minimum and makes our error detection clean and succinct.

The `std::exception` Class

The standard library provides a base class for many useful types of constructors and several types of standard exception derivations. To get these exception classes we use

```
#include <stdexcept>
```

The `std::exception` class is used by the standard itself for a hierarchy of exceptions:

```

exception
  logic_error
    domain_error
    invalid_argument
    length_error
    out_of_range
  runtime_error
    range_error
    overflow_error
    underflow_error

```

The `std::exception` class looks like this:

```

namespace std {
class exception {
public:
    virtual const char* what() const throw();
};

```

This doesn't seem like much, and what there is a little confusing, so let me explain.

Since this is supposed to be a base class, it provides a virtual function, `what`. This function's sole purpose is to return a C-style (nul-terminated) string that indicates what exception was thrown. Why a C-style string? Because one of the types of exceptions that can be thrown is when memory runs out. If it returned `std::string`, the operation of throwing a `std::exception` would require dynamic memory allocation (via `std::string`), which is what caused the exception to begin with. This would not be good, so it returns a **const char*** to a nul-terminated string as not to **require** dynamic memory. That doesn't mean that we **can't** use dynamic memory in **our** exception classes, though – just not if memory was the original problem.

Further into the `what` function, we see that it is a `const` member function, and we know what that means, but what is the `throw()` on the end? This is called a **no throw** clause. Its presence tells the compiler (and the run-time) that no exceptions will be thrown from within the body of this function. This is critical for

some operations, especially operations that are to be called when handling exceptions themselves: throwing an exception while handling an exception complicates matters **greatly**.

Remember that when we inherit from the base class we get two things (along with other things)

- 1) We get to overload any virtual functions
- 2) We get to use objects of a derived type anywhere an object of the base type is used by reference or pointer.

We can take advantage of all of this – the standard already does.

The `std::runtime_error` Exception

The `std::runtime_error` class is one of the more useful exceptions in that it allows us to give it a `std::string` on its constructor and that string will be returned (through a pointer, of course) as the result of the `what` function.

```
#include <stdexcept>
...
try {
    ...
    if (some error) { throw std::runtime_error("uh oh"); }
    ...
} catch (std::exception& e) {
    cerr << e.what() << endl;
}
...
```

Note that because `std::exception` is a base for all standard exceptions, and is therefore a more general type than the derived types, we must be careful of the order that they appear in catch clauses:

```
...
try {
    ...
} catch (std::exception& e) {
    ... this catches all std::exception derived types
} catch (std::runtime_error& e) {
    ... this will never be executed
}
...
```

A smart compiler might be able to warn you about this, but don't hold your breath.

Another Example

As hinted at earlier, we can inherit from `std::exception` or any other standard exception class for that matter and provide our own functionality, but keep with the standard interface.

```
#include <stdexcept>
#include <sstream>
#include <string>

using std::string;

class my_exception : public std::exception {
private:
    string file;
    int line;
    string description;
    string what_string;
public:
    my_exception(
        const string& init_file,
        int init_line,
        const string& init_description
```

```

    ): file(init_file),
       line(init_line),
       description(init_description)
    {
        std::ostringstream s;
        s << file << "@" << line << ' ' << description;
        what_string = s.str();
    }
    string get_file() const { return file; }
    string get_line() const { return line; }
    string get_description() const { return description; }
    const char* what() const throw() { return what_string.c_str(); }
};

```

Note the use of the `c_str` member to turn the `std::string` into a C-style string. We could use this like

```

...
try {
    ...
    throw my_exception(__FILE__, __LINE__, "first exception");
    ...
} catch (std::exception& e) {
    cerr << e.what() << endl;
}
...

```

Rethrowing Exceptions

From within a **catch** clause, if you decide that you cannot handle the exception that was caught, you can rethrow the exception that you just caught by simply using

```
throw;
```

This allows a **catch** at an outer level to handle an exception if the current **catch** cannot.

Exceptions in Constructors

Exceptions are particularly useful in constructors, especially since a constructor has no good way (such as a global indicator or dedicated “status” member) of returning a status to indicate that the object was constructed properly. By throwing an exception within the body of a constructor, we prohibit the object from ever being constructed in the first place.

```

class X {
private:
    string s;
public:
    X(const string& init_s) : s(init_s) {
        if (s.empty()) { throw std::runtime_error("empty string passed into X"); }
    }
};

int main() {
    try {
        X y("");
        ...
    } catch (std::exception& e) {
        cerr << e.what() << endl;
        return 1;
    } catch (...) {
        cerr << "oops" << endl;
    }
    return 0;
}

```

Note that if the exception is thrown, the already-constructed member, `s` is destructed.

Exception Retrying Schemes

Many situations might call for re-attempting an operation once an exception is thrown. Consider this scheme for handling this situation:

```
const int max_attempts = 3;
...
for (int attempt = 1;; ++attempt) {
    try {
        ... attempt operation
        break; // operation successful – get out of loop
    } catch (std::exception& e) {
        if (attempt >= max_attempts) {
            throw; // merely rethrow the last exception (could throw something else)
        }
        ... possibly perform corrective measures or wait on user input to proceed
    }
}

// if we get here, the operation was successful within max_attempts attempts
... carry on
```

Just food for thought.