More on references

- References can be returned by functions.
- Avoids copying overhead
- Often used to return some reference argument
- DON’T return references to locals!
- Returning a reference to a member is a trapdoor:

```cpp
#include "point.h"
int& Point::x()
{
    return m_x;
}
p.x() = 14;
```

Returning “*this”

Sometimes we want a member function to return a reference to the object it was called on. Here, if `String::append` can return a reference to the object it was called on, we can chain append calls together:

```cpp
String s1, s2, s3;
...
// Functions called left-to-right
s1.append(s2).append(s3);
```

Solution: use the implicit local variable `this`

```cpp
String& String::append(String other)
{
    ...
    return *this;
}
```
References as members

• Classes can have references as members.
• Replaces pointer and adds constraints we may want
• Reference member must be initialized early
• In fact, before constructor body is entered!
• New syntax for constructors: the initializer list

class Point
{
private:
    Graph& m_g;
    Point( Graph& g, int x = 0, int y = 0 );
};

The initializer list

// point.cpp

Point::Point(Graph& g, int x, int y)
: m_g(g)
{
    m_x = x;
    m_y = y;
}

Point::Point(Graph& g, int x, int y)
: m_g(g), m_x(x), m_y(y)
{}
Initializer list usage

- Preferred (by some) for member initialization.
- **Must** be used for:
  1) References, and
  2) Members of classes without default c’tors, and
  3) Constant members
- Order of initialization set by order of declaration in class!! - NOT by initializer list order.
- Function calls OK, but be careful!
- A decent test of debugger quality

An initializer list bug

class Point
{
  ...
  int m_y;
  int m_x;
};

// point.cpp

Point::Point(Graph& g, int x, int y)
: m_g(g), m_x(x), m_y(m_x) // UH OH
{}
Adding “read-only” with \texttt{const}

- Some things shouldn’t change while program is running:
  \begin{verbatim}
  const double PI = 3.14159;
  \end{verbatim}
- Invites compiler optimization
- Prevents stupid bugs
- Preferred over \texttt{define} or literals
- \texttt{const} is a type modifier
- \texttt{const} variables must be initialized

\begin{verbatim}
#include "point.h"
void print_point( const Point& p )
{
  ...
}
\end{verbatim}

\textbf{const reference arguments}

- Avoids copying overhead, saving space and time
- Preserves caller’s peace of mind
- Should always be the first choice when passing objects. (consider never using non-const reference)
- Makes altering object (through reference) a compiler error.
const class members

- Must be initialized in init list of each c’tor
- May be different for different objects
- `static const` members are different
- Ok to initialize integral types in declaration
- External initialization required for other types

```cpp
class IntStack
{
    static const int STACK_SIZE = 128;
    int m_stack[STACK_SIZE];
}
```

Initializing const static members

```cpp
class Point
{
    ...
    static const double PI;
};

// point.cpp

// Note static keyword here is an error
const double Point::PI = 3.14159;
```
For outdated compilers...

• Older compilers may not support initialization of integral statics in declaration.
• External initialization works, but can’t be used for array sizes.
• Use the “enum hack” instead:

```cpp
class IntStack {
  enum { STACK_SIZE = 128 }; // nameless enum
  int m_stack[STACK_SIZE]; // works!
}
```

const member functions

```cpp
void myfunc( const String& s )
• What should myfunc be able to do with s?
• Modifying data members (even public ones) is clearly out, but what about calling functions?
• A: It’s up to the author of class String
• Class writers can declare member functions acceptable for const objects.
• All other member functions are off-limits for const objects.
```
const member functions

class String
{
public:
    int length() const;
    void append( char * s );
    void reserve( unsigned int amount ) ??Hmm?;
private:
    int m_length;
    int m_bufSize;
...}

void myfunc( const String& s ) {
    int len = s.length(); // OK
    s.append("foo"); // not OK
}

// String.cpp
int String::length() const
{
    return m_length;
}

• Also constrains class writer:
• Illegal to modify class data in const member fn
• Illegal to call non-const member function
• Sometimes this constraint is too tight
mutable data members

class String
{
public:
    ...
    void reserve( unsigned int amount ) const;
private:
    ...
    mutable int m_bufSize;
    mutable char* m_pBuffer;
};

int String::reserve(unsigned int amount) const
{
    if( m_bufSize < amount ) {
        // realloc and copy, then...
        m_bufSize = amount; // OK, because mutable
    }

const and pointers

double d = 3.5;
const double cd = 4.5;

double* dp1;       // Plain old pointer
dp1 = &d;         // OK
dp1 = &cd;        // Error - address of const can’t be
                 // assigned to non-const pointer

const double* dp2; // pointer to const double
dp2 = &d;         // OK
dp2 = &cd;        // OK
**const** and pointers

double d = 3.5;
cast double cd = 4.5;

// const ptr to double
double *const dp3 = &d; // OK
double *const dp3 = &cd; // Error - not cast dbl
double *const dp3; // Error - must init

// const ptr to cast double
const double* const dp4 = &d; // OK
const double* const dp4 = &cd; // OK
const double* const dp4; // Error - must init

How should functions take class arguments?

- How many of these options do we really need?
  1. void foo(std::string s)
  2. void foo(const std::string s)
  3. void foo(std::string& s)
  4. void foo(const std::string& s)
  5. void foo(std::string* s)
  6. void foo(const std::string* s)

- Two might be enough!
- Option 2 is just silly
- Use 4 whenever possible, don’t use 6 at all
- Use 5 rather than 3 when argument will be modified
- Use 4 plus an explicit copy rather than 1
Exceptions

• Problem: dealing with errors is a pain, even in a simple language like C.
• It’s even worse in C++, where some obvious approaches don’t work.
• Exceptions provide a new mechanism for dealing with errors.
• Make use of inheritance to make code more modular.
• Fundamentally a control construct
• “Unit of recovery” == “function”

Exceptions

• A function which detects an error might be able to deal with it, or might not.
• If not, it should notify someone who can. Obvious candidates are callers of the function.
• How do we notify callers of an error?
• If A calls B calls C calls D and only A can deal with the error detected by D, can we avoid involving B and C in the communication?
• What form should the communication take?
Exceptions

• A function which detects an error can *throw* an exception.
• A function which can deal with errors can *catch* an exception.
• When an exception is thrown, control “travels” up the call stack, “looking” for a function that can catch it.
• If none is found, some default behavior is invoked (typically termination)

What’s an exception?

• An object of any copy-able type can function as an exception.
• Avoid using native types
• Often new classes are created specifically for exceptions.
• The STL provides several exception types.
Throwing exceptions

struct AcctError
{
    AcctError( std::string p ) : problem(p) {}  
    std::string problem;
};

tool Account::withdraw( unsigned int amount )
{
    if ( amount > m_cashBalance )
    {
        throw AcctError();
    }
...
}
Catching exceptions

```cpp
main()
{
    Account acct1;
    acct1.deposit( 100000 );

    try
    {
        acct1.withdraw( 2000000 );
    }
    catch ( AcctError& e )
    {
        cout << "Error: " << e.problem << endl;
    }
    ...
}
```

throw syntax and semantics

- A throw statement is syntactically like a return statement.
- Execution leaves the function that throws.
- Local objects are all destroyed (like a return).
- Stack is “unwound,” destroying local objects in each frame until catch is found.
- Execution resumes with first statement of catch.
- When catch finished, execution continues with statements after catch.
try-catch syntax

• A try block must use {}s and is a scope.
• A try block must be followed by one or more catch blocks.
• A catch block must use {}s and is a scope.
• Each catch block must have a declaration list
• Declaration lists declare one type, with an optional identifier.
• The catch block can treat the identifier like a local variable.
• Catch block identifiers shadow other local vars.

The STL exceptions

#include<stdexcept>

exception
    bad_alloc
    bad_exception
    bad_cast
    bad_typeid
    ios_base::failure
    logic_error
        length_error
        domain_error
        out_of_range
        invalid_argument
    runtime_error
        range_error
        overflow_error
        underflow_error
OOP Philosophy

• Interface specifications make promises to clients.
• Sometimes, the right specification is not obvious.
• The compiler can help you keep these promises
• …and help keep clients honest, too.
• Minimize use of “workarounds” like mutable.
Object Oriented Programming in C++

Assignment 2 -- Due March 5

In this assignment we will modify the String class handed out in class. Copies of the code are available on the course website, as is the driver for this assignment.

You should not modify the driver, nor the existing implementation. Note that this implementation doesn't use any sort of termination in the buffer (unlike C strings). Your new functions shouldn't rely on such termination either.

It is acceptable (even desirable) to write helper functions in order to consolidate common code. If you do add helper functions, consider whether they should be public or private.

Part 1.

1. Write a function `size_t String::length() const` that returns the length of the string.

2. Add functions `String::append( const String& other)` and `String::append( const char* const other)` that append the argument to the string. Consider what the return value should be. You should be able to write code like:
   ```cpp
   String s1 = "foo";
   String s2 = "bar";
   s1.append( s2 );
   // now s1 is "foobar"
   String s3 = "moo";
   s3.append( "quack" ).append( "meow" );
   // now s3 is "mooquackmeow"
   ```

3. Add `String::substr( size_t start, size_t numChars) const` that returns a new string consisting of `numChars` chars starting with `start`. If there are fewer than `numChars` available, your function should return all there are. If start is out of range throw an std::out_of_range exception.

   ```cpp
   String s = "foobar";
   s.substr(0,3) returns foo
   s.substr(2,3) returns oba
   s.substr(2,5) returns obar
   ```

4. Add `char String::char_at( size_t pos) const` which returns the character at the 0-based position `pos`. Throw an std::out_of_range exception if pos is out of range.

5. Add `char& String::char_at (size_t pos)` which returns the character at the 0-based position `pos`. Throw an std::out_of_range exception if pos is out of range.

Part 2.

6. Add `char String::operator[] const` and `char& String::operator[]` which mimic the behavior of `char_at()`, but do not throw upon invalid input. They should not even check for
invalid input (therefore these are faster, less safe versions of char_at).

7. Add two `String::operator+=( )` which mimic the behavior of append()

8. Add `String& String::operator=( const String& other)` which correctly does assignment of Strings

9. Add `bool String::operator==( const String& other)` which correctly tests for case-sensitive equality of Strings.

10. Add `bool String::operator<( const String& other)` which correctly tests for case-sensitive ordering of Strings.

Use the provided driver to verify that your code works. Hand in string.h and string.cpp. You do not need to hand in any code from earlier steps. Also, you do not need to hand in the driver or any transcripts.

Grading Policy for Assignment 2

This assignment is worth 12 points, as follows:

- produces correct output = 3
- no memory leaks = 1
- no bugs undetected by driver = 1
- code reuse = 3
- code clarity/efficiency = 3 (code reads easily; implementations are short and to the point)
- grader discretion = 1