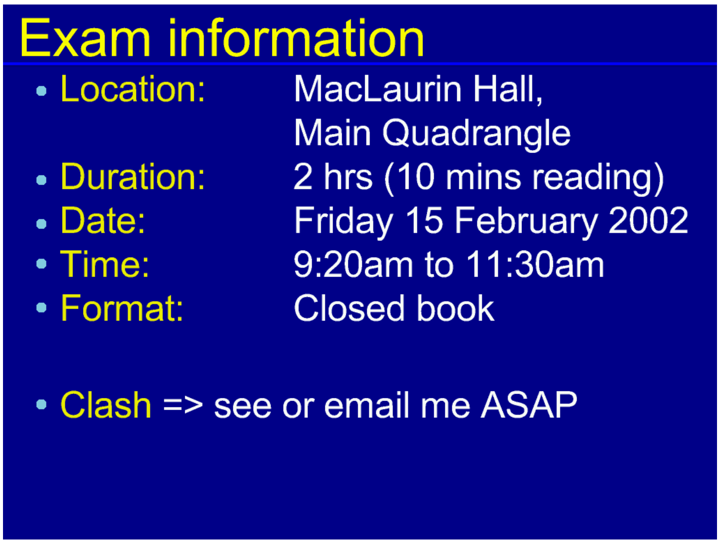
COMP2004 Programming Practice 2002 Summer School

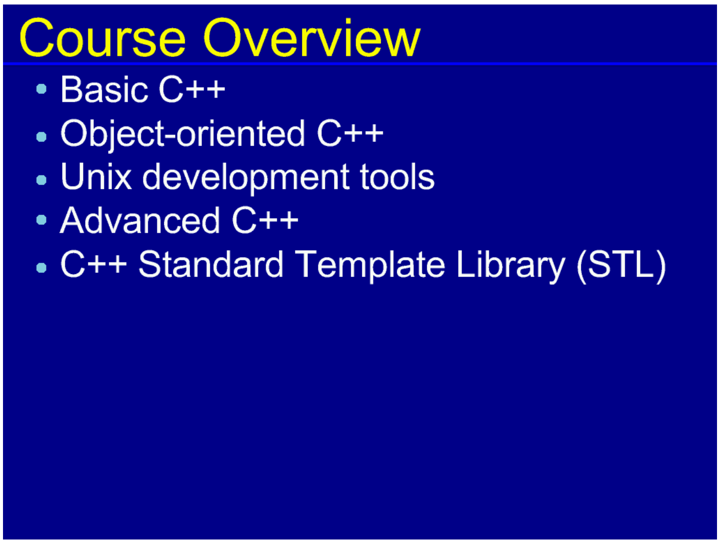
Kevin Pulo
School of Information Technologies
University of Sydney



Exam information

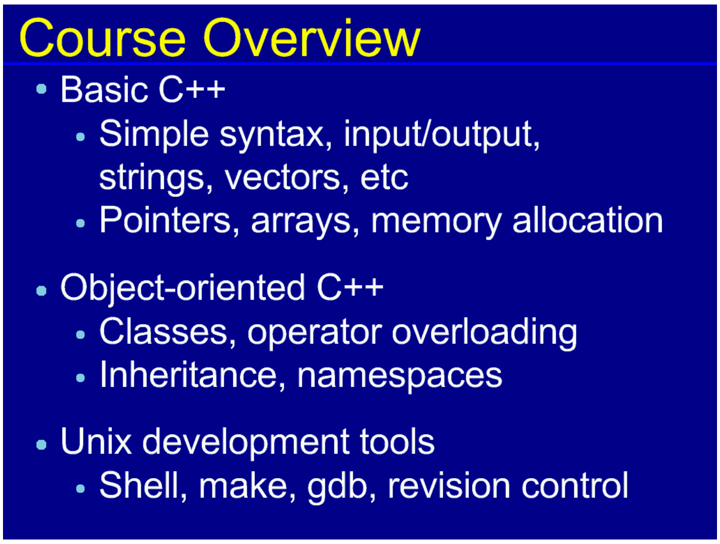
- Location: MacLaurin Hall, Main Quadrangle
- Duration: 2 hrs (10 mins reading)
- Date: Friday 15 February 2002
- Time: 9:20am to 11:30am
- Format: Closed book

- Clash => see or email me ASAP



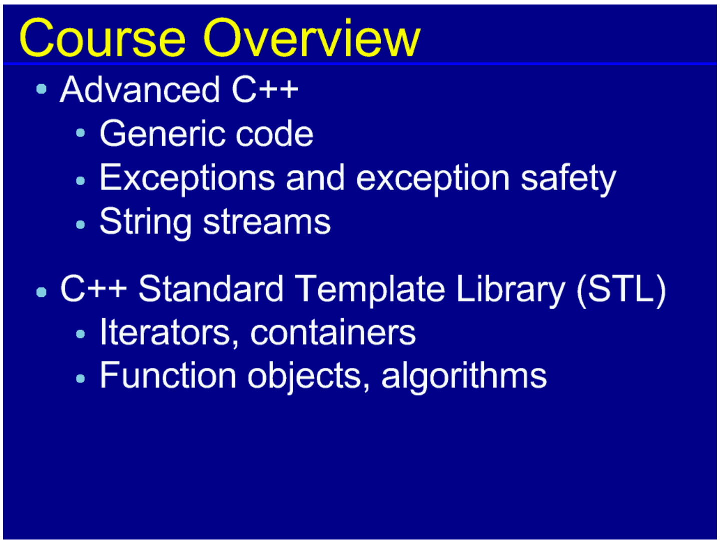
Course Overview

- Basic C++
- Object-oriented C++
- Unix development tools
- Advanced C++
- C++ Standard Template Library (STL)



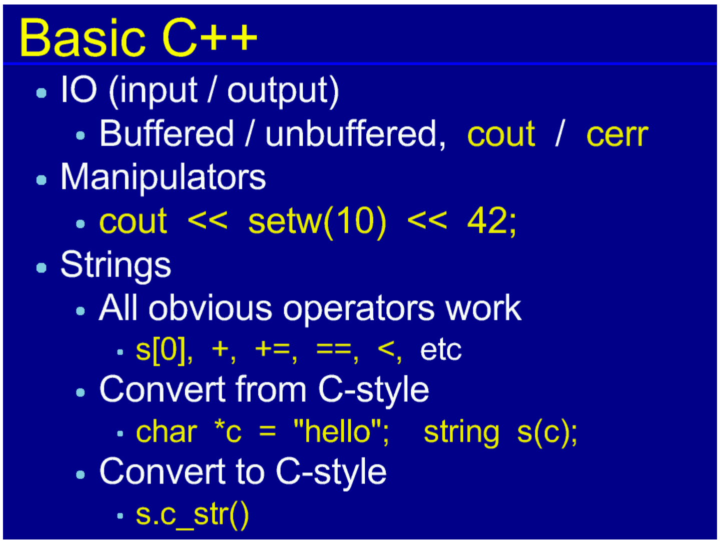
Course Overview

- Basic C++
  - Simple syntax, input/output, strings, vectors, etc
  - Pointers, arrays, memory allocation
- Object-oriented C++
  - Classes, operator overloading
  - Inheritance, namespaces
- Unix development tools
  - Shell, make, gdb, revision control



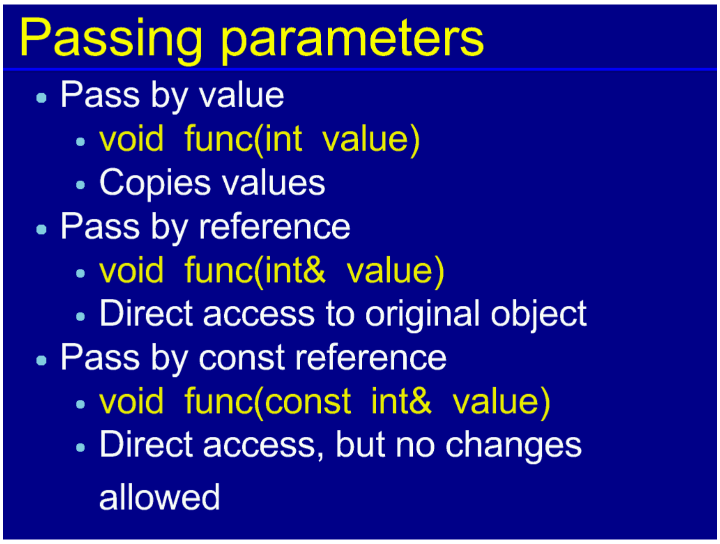
Course Overview

- Advanced C++
  - Generic code
  - Exceptions and exception safety
  - String streams
- C++ Standard Template Library (STL)
  - Iterators, containers
  - Function objects, algorithms



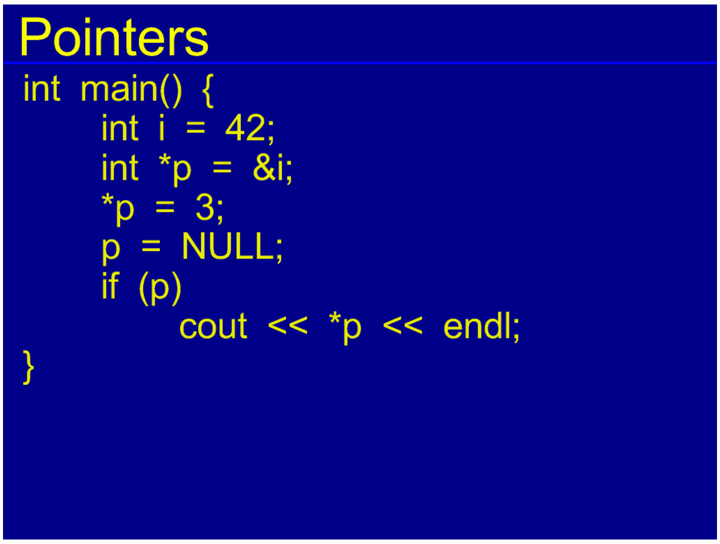
Basic C++

- IO (input / output)
  - Buffered / unbuffered, cout / cerr
- Manipulators
  - cout << setw(10) << 42;
- Strings
  - All obvious operators work
    - s[0], +, +=, ==, <, etc
  - Convert from C-style
    - char *c = "hello"; string s(c);
  - Convert to C-style
    - s.c_str()



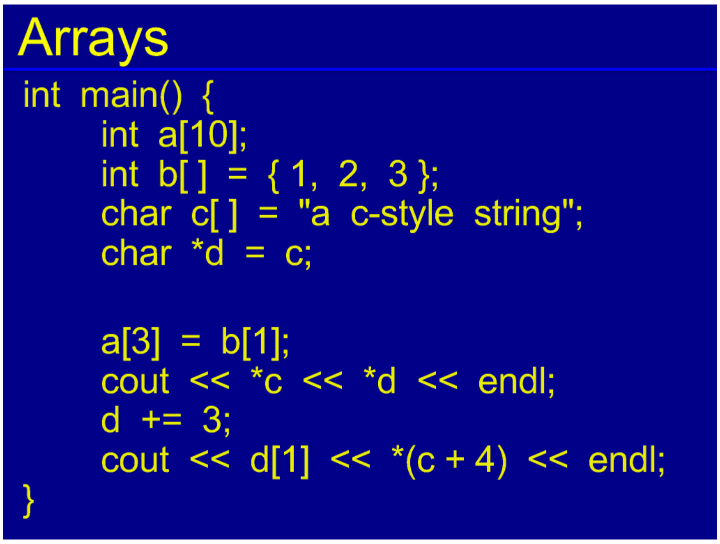
Passing parameters

- Pass by value
  - void func(int value)
  - Copies values
- Pass by reference
  - void func(int& value)
  - Direct access to original object
- Pass by const reference
  - void func(const int& value)
  - Direct access, but no changes allowed



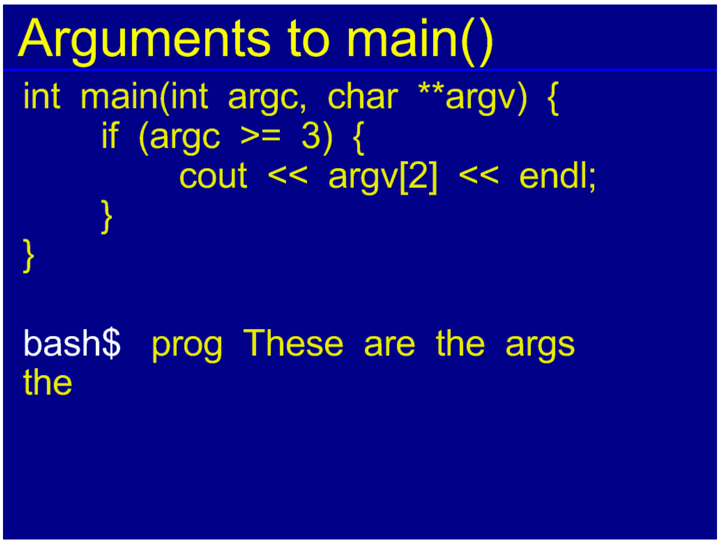
Pointers

```
int main() {
    int i = 42;
    int *p = &i;
    *p = 3;
    p = NULL;
    if (p)
        cout << *p << endl;
}
```



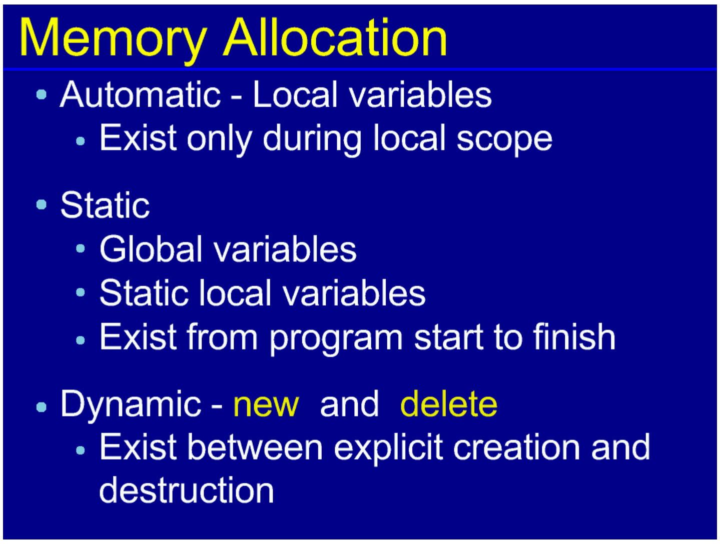
Arrays

```
int main() {
    int a[10];
    int b[ ] = { 1, 2, 3 };
    char c[ ] = "a c-style string";
    char *d = c;

    a[3] = b[1];
    cout << *c << *d << endl;
    d += 3;
    cout << d[1] << *(c + 4) << endl;
}
```



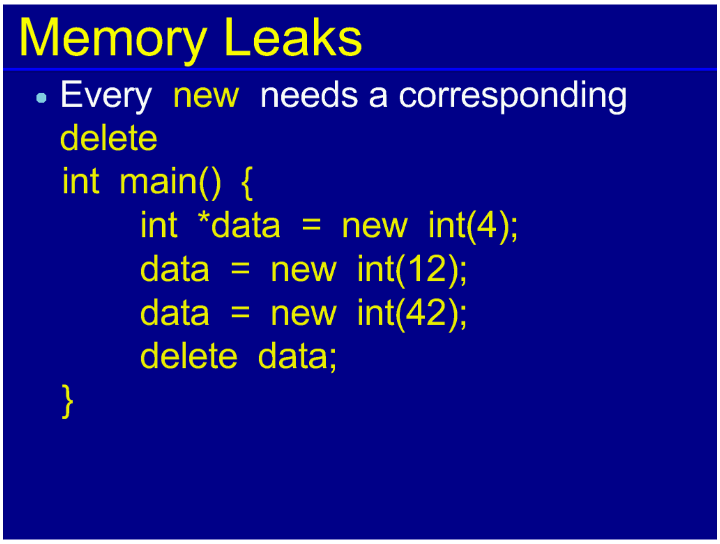
Arguments to main()

```
int main(int argc, char **argv) {
    if (argc >= 3) {
        cout << argv[2] << endl;
    }
}

bash$  prog These are the args
the
```



Memory Allocation

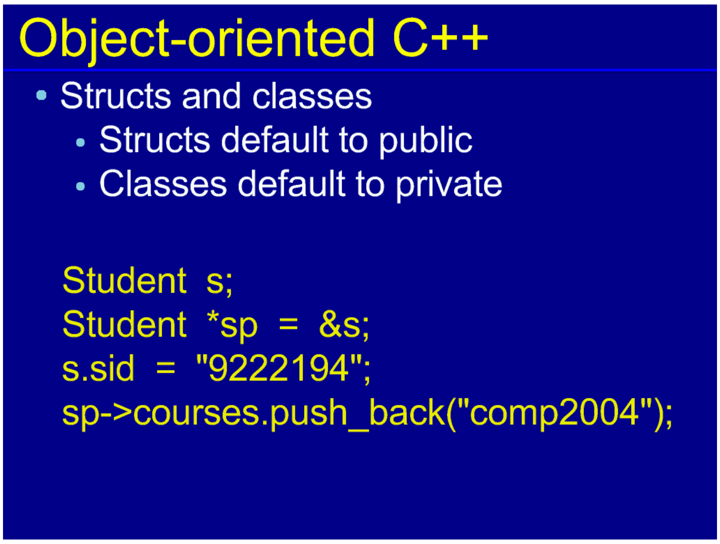
- Automatic - Local variables
  - Exist only during local scope
- Static
  - Global variables
  - Static local variables
  - Exist from program start to finish
- Dynamic - new and delete
  - Exist between explicit creation and destruction



Memory Leaks

- Every new needs a corresponding delete

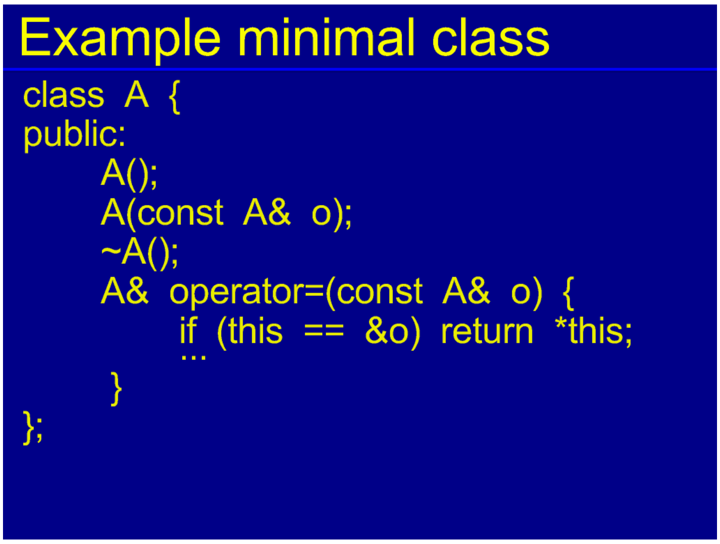
```
int main() {
    int *data = new int(4);
    data = new int(12);
    data = new int(42);
    delete data;
}
```



Object-oriented C++

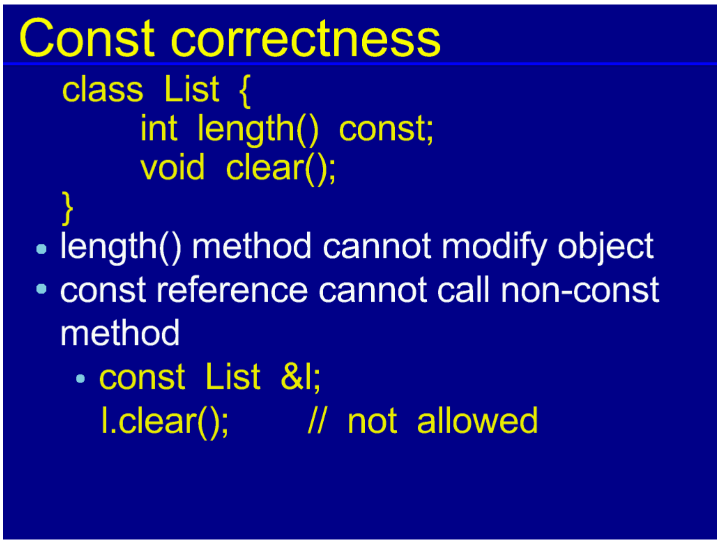
- Structs and classes
  - Structs default to public
  - Classes default to private

```
Student s;
Student *sp = &s;
s.sid = "9222194";
sp->courses.push_back("comp2004");
```



Example minimal class

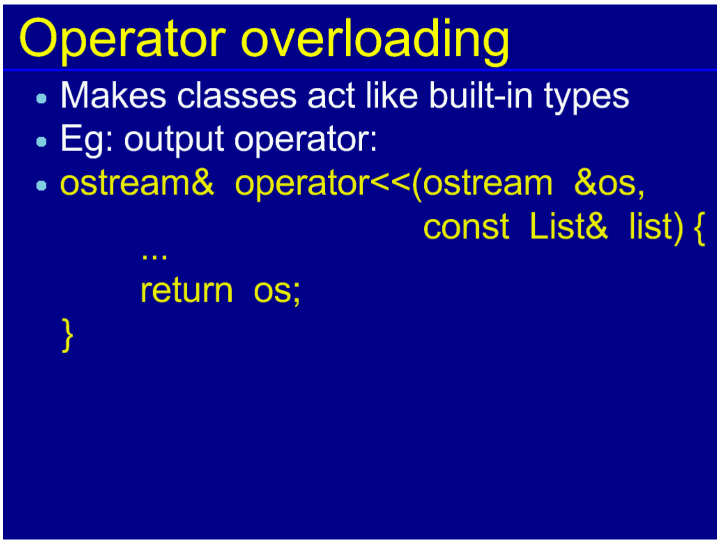
```
class A {
public:
      A();
      A(const A& o);
      ~A();
      A& operator=(const A& o) {
            if (this == &o) return *this;
            …
      }
};
```



Const correctness

```
class List {
      int length() const;
      void clear();
}
```

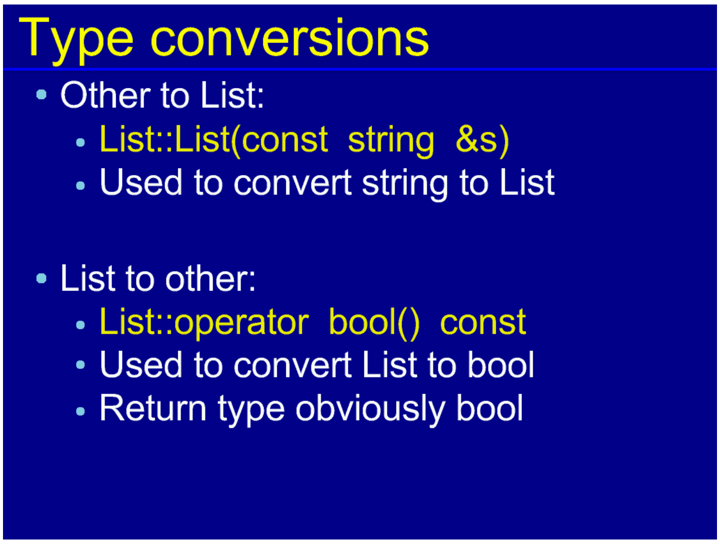
- length() method cannot modify object
- const reference cannot call non-const method
  - `const List &l;`
    `l.clear();      // not allowed`



Operator overloading

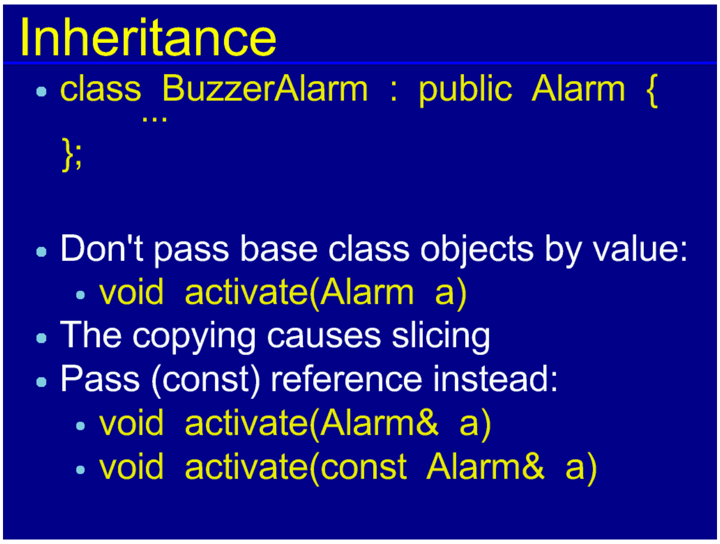
- Makes classes act like built-in types
- Eg: output operator:
- `ostream& operator<<(ostream &os,`

```
                              const List& list) {
        ...
        return os;
}
```



Type conversions

- Other to List:
  - `List::List(const string &s)`
  - Used to convert string to List
- List to other:
  - `List::operator bool() const`
  - Used to convert List to bool
  - Return type obviously bool



Inheritance

- `class BuzzerAlarm : public Alarm {`

```
      …
};
```

- Don't pass base class objects by value:
  - `void activate(Alarm a)`
- The copying causes slicing
- Pass (const) reference instead:
  - `void activate(Alarm& a)`
  - `void activate(const Alarm& a)`

Accessibility

- public members visible to all
- private members hidden from children
- protected members visible to children, but still hidden from others

- Use public to provide interface to others
- Use protected to provide interface to children

Virtual methods

```
void activate(Alarm& a) {
    a.turn_on();
}
BuzzerAlarm b;
activate(b);
```

- If turn_on() is not virtual:
  - Calls turn_on() in Alarm
- If turn_on() is virtual:
  - Calls turn_on() in BuzzerAlarm
- Virtual methods require virtual destructor

Pure virtual methods

```
class Alarm {
    virtual void turn_on() = 0;
};
```

- Alarm is abstract
  - Cannot create Alarm objects
- Child classes must redefine turn_on()

Namespaces

```
namespace lib {
    string func() { ... }
}

cout << lib::func() << endl;
using namespace lib;
cout << func() << endl;
```

Unix development tools

- make
  - Automatic code compilation
- gdb
  - Debugging
- shell
  - Simple text-manipulation programs
  - Bourne shell vs bash
  - Many other utilities

make

```
OBJS = main.o student.o course.o
CXXFLAGS = -Wall -g
LDFLAGS = -lm

all: prog
prog: $(OBJS)
   $(CXX) $(CXXFLAGS) $(LDFLAGS) \
                            -o prog $(OBJS)
clean:
   rm -f $(OBJS)
```

gdb

- Compiled with `-g`
- Crashed with
  `Segmentation fault (core dumped)`
- Run: `gdb prog core`
- Common commands
  - `bt`
    - Get a stack backtrace
    - Shows where the crash happened
  - `p head`
    - Print value of variable head

Simple shell

- Simple shell commands
  - `cd, cp, mv, rm`
  - `echo` - Prints parameters to stdout
  - `cat` - Prints files (or stdin) to stdout
- Redirection
  - `echo "foo" > foo.txt`
  - `echo "bar" >> foo.txt`
  - `cat < hello.txt`
- Redirection caveat
  - `prog file.txt > file.txt`

Shell / environment variables

- Assign:
  - `variable_name="value of variable"`
- Use:
  - `echo "$variable_name"`
- Shell variable -> environment variable:
  - `export variable_name`

If

- Exit status:
  - 0 is true, everything else is false
  - `exit n` command in shell

```
if cmp file1.txt file2.txt; then
   echo "files same"
elif diff -i file1.txt file2.txt; then
   echo "files same except case"
else
   echo "files different"
fi
```

Alternate if

- `if [ "$somevar" = "hello" ]; then`
- `if [ "$somevar" != "hello" ]; then`
- `if [ "$num" -lt 42 ]; then`
- `if [ "$num" -ge 42 ]; then`
- `if [ -r "$fname" ]; then`
- `if [ "$fname" -nt "file.txt" ]; then`
- and so on...

stdout -> shell variable

- Use backticks or `$()` in bash:

```
echo "f1.h f2.h f3.h" > file.lst
filelist=`cat file.lst`
cat $filelist
cat `cat file.lst`
cat $(cat file.lst)
```

- Shell arithmetic (`expr` in Bourne, `$(())` in bash):
  - ``result=`expr 3 + 8 \* 2` ``
  - `result=$((expr 3 + 8 * 2))`

while

```
count=1
while [ $count -le 10 ]; do
        echo "$count"
        count=$(($count + 1))
done

while :; do
        if [ -e "file.txt" ]; then
                break
        fi
        sleep 1
done
```

Command line parameters

- The shell script sample:

```
#!/gnu/usr/bin/bash
while [ $# -ge 2 ]; do
        echo "$1" "$2"
        shift
done

bash$ sample a bc defgh
a bc
bc defgh
```

case

```
case "$1" in
        -d*)
                cmd="diff"
                ;;
        -c*)
                cmd="cmp"
                ;;
        *)
                exit 1
                ;;
esac
$cmd file1.txt file2.txt
```

for and pipes

```
#!/gnu/usr/bin/bash
for infile in tests/*.in; do
        name=`basename $infile .in`
        if prog < $infile | cmp - \
                        tests/$name.out; then
                echo "$name : passed"
        else
                echo "$name : failed"
        fi
done
```

Useful Unix utilities

- sed - stream editor
- awk - processing columnar data
- sort - sorts lines in a file
- uniq - removes duplicate lines
- head - output first n lines
- tail - output last n lines
- cut - extracts columns of characters
- join, paste - merges files by columns

Course survey

- Voluntary and anonymous
  - So don't write your name on it
- 15 minutes
  - I won't be here
  - Student representative collects and seals forms
- Comments are useful for improving all courses
- Unit of Study = Programming Practice
- Unit of Study Code = COMP2004

Advanced C++

- Generic code
  - Template functions
  - Template classes
- Exceptions
- Exception Safety
- String streams

Template Functions

```
template<typename T>
typename vector<T>::size_type
find(const vector<T> &v,
                        const T &value) {
     for (typename vector<T>::size_type
               i = 0; i < v.size(); ++i)
          if (v[i] == value) return i;
     return v.size();
}
vector<int> vi;
find(vi, 42);
```

Template Classes

```
template<typename T>
class Node {
     T value;
     Node<T> *next;
     ...
};
```

- Heavy reliance on operators
- All template code in .h files
  - Only time this is allowed

Exceptions

- Seperate error detection from handling
- Any object can be exception
- First catch type matched is used (including inheritance)

- `void func() throw (e1, e2)`
- `void func() throw ()`
- `void func()`

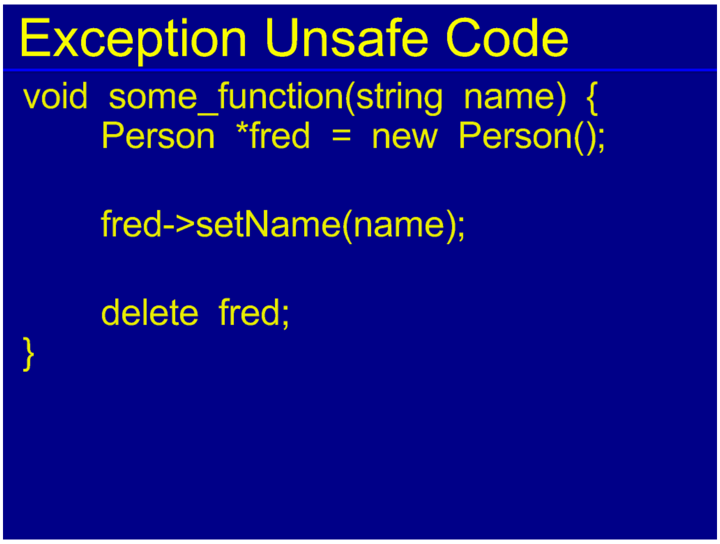
- Constructors can throw exception
- Copy constructors and destructors shouldn't

Exceptions

```
struct Error { int i;
     Error(int i) : i(i) {} };
struct OtherError { };

try {
     throw Error(42);
} catch (Error e) {
     cout << e.i << endl;
     throw OtherError();
} catch (...) {
     throw;
}
```

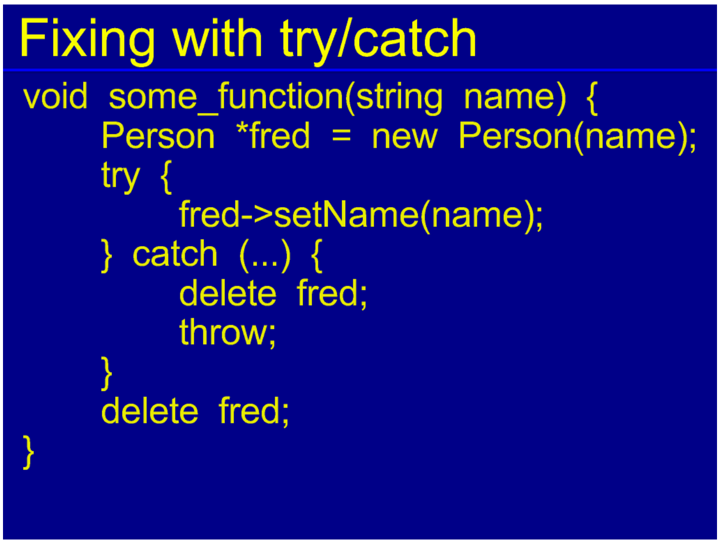
Exception Safety

- Each function must:
- Basic Guarantee
  - Resources not leaked, objects still usable
- Strong Guarantee
  - Program state is as before the call
- Nothrow
  - The function will never throw



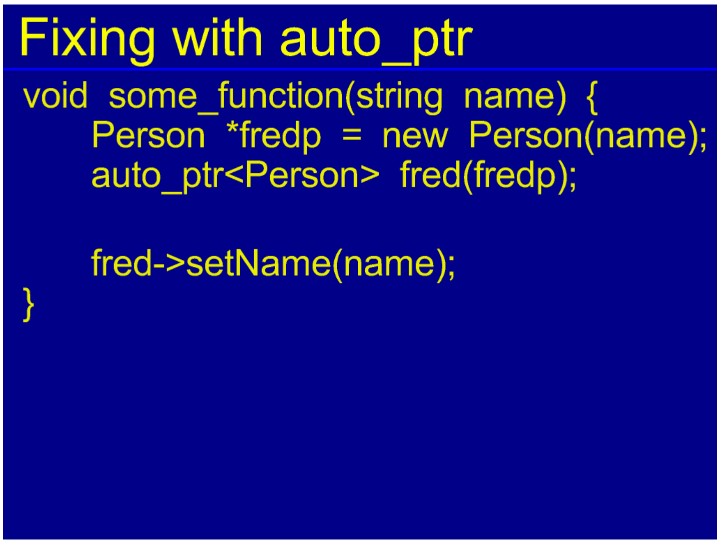
Exception Unsafe Code

```
void some_function(string name) {
	Person *fred = new Person();

	fred->setName(name);

	delete fred;
}
```



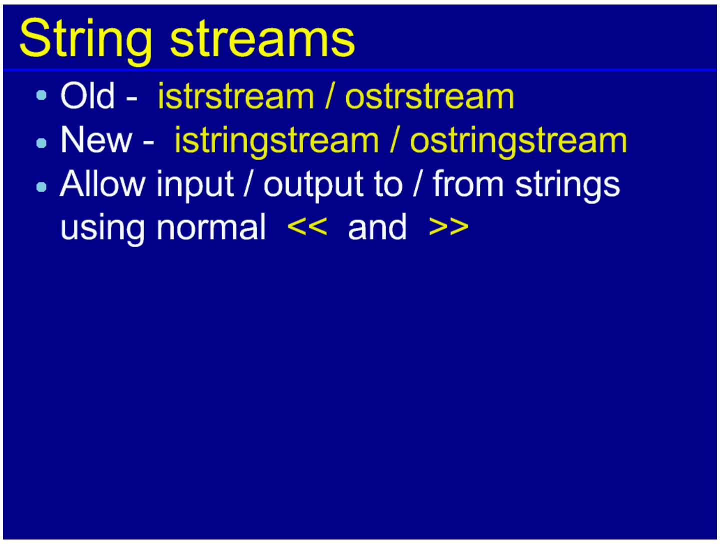
Fixing with try/catch

```
void some_function(string name) {
	Person *fred = new Person(name);
	try {
		fred->setName(name);
	} catch (...) {
		delete fred;
		throw;
	}
	delete fred;
}
```



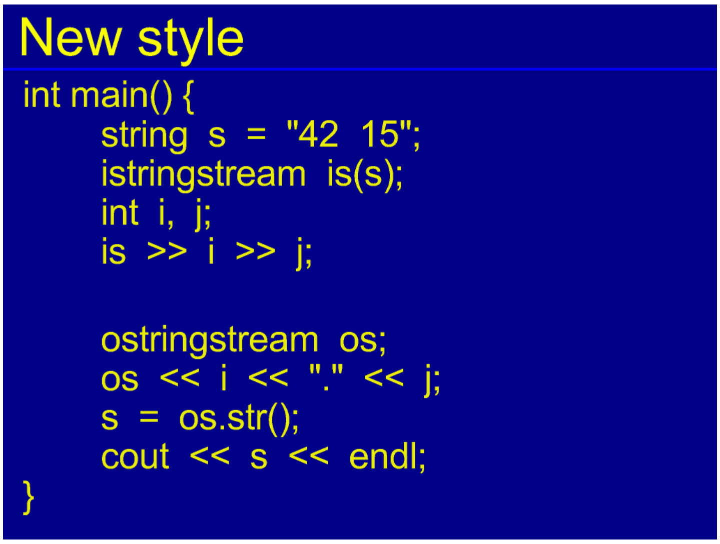
Fixing with auto_ptr

```
void some_function(string name) {
	Person *fredp = new Person(name);
	auto_ptr<Person> fred(fredp);

	fred->setName(name);
}
```



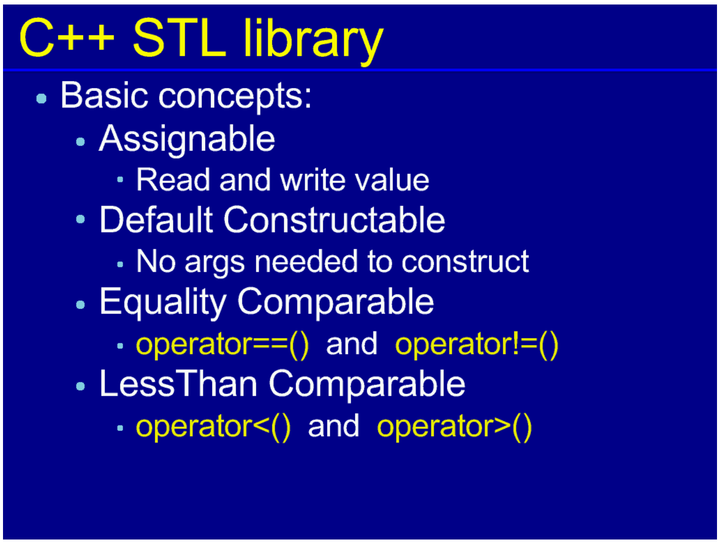
String streams

- Old - istrstream / ostrstream
- New - istringstream / ostringstream
- Allow input / output to / from strings using normal << and >>



New style

```
int main() {
	string s = "42 15";
	istringstream is(s);
	int i, j;
	is >> i >> j;

	ostringstream os;
	os << i << "." << j;
	s = os.str();
	cout << s << endl;
}
```



C++ STL library

- Basic concepts:
  - Assignable
    - Read and write value
  - Default Constructable
    - No args needed to construct
  - Equality Comparable
    - operator==() and operator!=()
  - LessThan Comparable
    - operator<() and operator>()

Iterators

- Restricted pointers
- Input / Output Iterator
  - Equality Comparable, Assignable, Dereference read / write, Increment
- Forward Iterator
  - Input and Output, no alternating
- Bidirectional Iterator
  - Forward, Decrement
- Random Access Iterator
  - Bidirectional, random access

Iterator Ranges

- begin and end iterators
- Range is [begin, end)
  - includes begin
  - but not end

Common output iterators

- ostream_iterator<T>(cout)
  - Outputs to ostream
- istream_iterator<T>(cin)
  - Inputs from istream
- istream_iterator<T>()
  - End-of-input iterator
- back_inserter(c), front_inserter(c)
  - Requires Front / Back Insertion Sequence

Containers

- Container
  - One Input Iterator, begin(), end()
- Forward Container = Container + Forward Iterators
- Reversible Container = Forward Container + Bidirectional Iterators, rbegin(), rend()
- Random Access Container = Reversible Container + Random Access Iterators

STL Container definitions

- Typedefs:
  - value_type
  - reference, const_reference
  - pointer, const_pointer
  - iterator, const_iterator
  - difference_type, size_type
- Methods:
  - a.size(), a.max_size()
  - a.empty(), a.swap(b)
  - a.begin(), a.end()

Container abstractions

- Sequence
  - Forward container, no reordering, add / delete anywhere
  - Front Insertion Sequence
    - push_front(), pop_front(), insert / access front quickly
  - Back Insertion Sequence
    - push_back(), pop_back(), insert / access last element quickly

Sequence types

- vector
  - Random Access Container
  - Back Insertion Sequence
  - Insertion can invalidate iterators
- list
  - Reversible Container
  - Front and Back Insertion Sequence
- deque
  - Random Access Container
  - Front and Back Insertion Sequence

Container abstractions

- Associative Container
  - Elements add / delete / access via keys
  - Unique vs Multiple
  - Simple vs Pair
  - Sorted vs Hashed

Associative Container types

- For all, deletion invalidates only deleted
- set
  - Unique, Simple, Sorted
- multiset
  - Multiple, Simple, Sorted
- map
  - Unique, Pair, Sorted
- multimap
  - Multiple, Pair, Sorted

Adapter containers

- stack
  - LIFO: only use top element
  - Use Back Insertion Sequence
- queue
  - FIFO: push back, pop front
  - Use Front and Back Insertion Seq
- priority_queue
  - Access top (largest) element
  - Use Random Access Container
  - Use LessThan Comparable elems

Function Objects

- Can be called like a function
  - Class overloads operator()
- Generator: 0 args
- Unary Function: 1 arg
- Unary Predicate: 1 arg, return bool
- Binary Function: 2 args
- Binary Predicate: 2 args, return bool
- Strict Weak Ordering: eg. less than

STL Function Objects

- Binary functions: plus, minus, ...
- Binary predicates: logical_and, logical_or, less, greater, equal_to, ...
- Unary predicates: logical_not, ...
- Unary functions: negate, ...

Adapter Function Objects

- Convert function objects, uses helpers
- bind1st(), bind2nd()
  - Convert binary function to unary
  - Allow constant value for one arg
- not1(), not2()
  - Logical not unary / binary predicate
- compose1(), compose2()
  - Composes unary / binary functions
- mem_fun_ref(), mem_fun()
  - Uses member function

Algorithms

- find - linear search for value
- find_if - linear search using predicate
- adjacent_find - linear search for adj
- find_first_of - first of possible values
- search - linear search for subrange
- find_end - like search but backwards
- search_n - first consecutive n of value
- count - counts occurances of value
- count_if - counts matches

Algorithms

- for_each - apply unary function
- accumulate - sum values
- equal - compare two ranges
- mismatch - find first difference
- lexicographical_compare -
- max_element - finds largest element
- min_element - finds smallest element

Mutating Algorithms

- Ranges are fixed
- Variants _copy, _if where appropriate
- Notables:
  - copy
  - transform
    - like for_each(), saves return values
  - replace
  - remove, unique
    - only rearranges, returns new end iter
    - use c.erase() to actually remove

Mutating Algorithms

- Notables:
  - reverse
  - random_shuffle
  - sort, partial_sort, is_sorted, merge

Exam information

- Location: MacLaurin Hall,
Main Quadrangle
- Duration: 2 hrs (10 mins reading)
- Date: Friday 15 February 2002
- Time: 9:20am to 11:30am
- Format: Closed book

- Clash => see or email me ASAP

- Good luck!