Mopping Up

- static and global – scoping rules
- pointers to functions, void*
- inline
- recursion
- other types: String, enum
- nested classes
- typedef, struct
- multiple inheritance; “is a” and “has a”; container classes
- exceptions

Introduction

- To finish off, we look at many of the “small” rules that are useful, important, or simply annoying, but didn’t merit their own section.
- I won’t be “inclusive” – there will be several small details I will leave as “exercises for the student”
- See other books, ANSI standard, etc.

static and global – scoping rules

- So far, variables have been
  - class member data
  - local variables of limited scope – within `{}`
- If the variable goes out of scope, or the object goes out of scope, the variable is gone.
- To prevent this, we can declare a variable `static`.
- when part of a class, it becomes a *class variable*, rather than an *instance variable*.

Let's look at an example:

```
#ifndef __FLOAT_HH   // Float1.hh
#define __FLOAT_HH
#include <iostream.h>

class Float {
public:
  Float(float x=0.0);
  float operator()() const;
  friend ostream& operator<<(ostream&, const Float&);
private:
  float m_x;
  static unsigned long m_get;
};
#endif // __FLOAT_HH
```

This is a class that encapsulates a `float`, and uses a `static` to keep track of how many times the field is accessed.

```
#include "Float1.hh"   // Float1.cc

Float::Float(float x) : m_x(x) {}

float Float::operator()() const { m_get++; return m_x; }

unsigned long Float::m_get(0);

ostream& operator<<(ostream& os, const Float& f) {
  os << f.m_x << " (field accessed " << Float::m_get << " times)";
  return os;
}
```

```
#include "Float1.hh"   // Static1.cc

int main() {
  const Float a(3.14159);
  Float b(2.7);
  float x(0.0);
  for (int i=0; i<10; i++) {
    x += a() + b();
  }
  cout << "a=" << a << endl;
  a.~Float();
  Float c(1.414);
  cout << "x = " << x+c() << endl;
  cout << "c=" << c << endl;
  return 0;
}
```

Points to note:

- We overload `operator()()`, the function call operator, to provide a "natural" accessor function. Note that we must declare it `const`.
- The variable `m_get` is declared `static`. It is initialized in `Float.cc`
- In `ostream& operator<<()`, we refer to `f.m_x` (since it is for that particular object), but `Float::m_get` (since it is for the whole class).
- We explicitly call the destructor for `a`, to ensure that `a` is out of scope.
- Since `Float::m_get` is static, it exists without an object instance.

- What if we need to *access* static members?
- – if the member is `const` we can make it `public`
    (This is the exception to making all data members private)
  - otherwise, we must use a *static function*
    It has to be static to allow access without an object
- Using static class members almost completely removes the need for global data - with a bonus:

The class name removes global naming ambiguities

```
#ifndef __FLOAT_HH   // Float2.hh
#define __FLOAT_HH
#include <iostream.h>

class Float {
public:
  Float(float x=0.0);
  float operator()() const;
  friend ostream& operator<<(ostream&, const Float&);
  static unsigned long getStatic() { return m_get; }
  static const float PI;
private:
  float m_x;
  static unsigned long m_get;
};
#endif // __FLOAT_HH
```

```
#include "Float2.hh"   // Float2.cc

Float::Float(float x) : m_x(x) {}

float Float::operator()() const { m_get++; return m_x; }

unsigned long Float::m_get(0);
const float Float::PI(3.14159);

ostream& operator<<(ostream& os, const Float& f) {
  os << f.m_x << " (field accessed " << Float::m_get << " times)";
  return os;
}
```

```
#include "Float2.hh"   // Static2.cc

int main() {
  const Float a(1.414);
  Float b(2.7);
  float x = a() + b() + Float::PI;
  cout << "x = " << x << endl;
  cout << "field accessed: " << Float::getStatic() << endl;
  return 0;
}
```

File and Global Scope

- Sometimes, we have to make the scope of an object the whole file, or even the whole program.
- We can make an object `static` or `extern`

In C++ `extern` should be almost completely avoided. It can break encapsulation, and there are other, better ways

As an example, consider a program with 3 parts:

- Initialization
- A main loop
- Termination

```
#ifndef __JOB_HH   // Job.hh
#define __JOB_HH
#include <iostream.h>

class Job {
public:
  static Job* Instance();
  void begin();
  void middle();
  void end();
private:
  Job();
  static Job* m_instance;
};
#endif // __JOB_HH
```

This is a useful class that guarantees only one instance of an object.
It is called a Singleton class.

```
#include "Job.hh"   // Job.cc

Job* Job::m_instance(0);

Job* Job::Instance() {
  if (m_instance == 0) {
    m_instance = new Job();
  }
  return m_instance;
}

Job::Job() {}
```

First put all the functions in one file:

```
#include "Job.hh"   // Driver1.cc

static int global;

void Job::begin() { cout <<"begin: global=" << (global=0)<<endl; }
void Job::middle() { cout <<"middle: global=" <<++global <<endl; }
void Job::end() { cout << "end: global=" << global << endl; }

int main() {
  Job* j=Job::Instance();
  j->begin();
  for (int i=0; i<10; i++) j->middle();
  j->end();
  return 0;
}
```

Then put the “driver” program and each function in separate files:

```
#include "Job.hh"   // Driver2.cc

int main() {
  Job* j=Job::Instance();
  j->begin();
  for (int i=0; i<10; i++) { j->middle(); }
  j->end();
  return 0;
}
```

The global object (`global`) has to be defined once and once only, but declared `extern` wherever it is used.

```
#include "Job.hh"   // Job_begin.cc

int global(0);  // more usually would be an object

void Job::begin() {
  extern int global;
  cout <<"begin: global=" << global <<endl;
}
```

Points to note:

- We define a `Job` class that can only have one instance. We do this with:
  - a static private pointer to the instance
  - a private constructor
  - a public static function, `Instance`
- In `main`, we create a singleton `Job` instance, and call `begin`, `middle`, `end` as user-supplied member functions.
- `global` is declared and defined `static int` – it can then be used by all 3 functions.
- In the second case, the file containing `main` doesn't know about `global`, nor the function implementations
- Then `global` has to be *declared* `extern` wherever it is used, and *defined* exactly once.

pointers to functions, `void*`

- A function has an address – its code is somewhere in memory.
- We can use that address to pass a function name to another function.
- Using virtual functions, we don't have to do this very often

```
#include <iostream.h>   // Dispatcher2.cc

void f1(const int* p) { cout << "I am f1 " << *p << endl; }

void f2(const float* p) { cout << "I am f2 " << *p << endl; }

void dispatcher(void (*f)(const void*), const void* p) { (*f)(p); }

int main() {
  const int* i = new int(17);
  const float* x = new float(3.14159);
  dispatcher(f1, i);
  dispatcher(f2, x);
  return 0;
}
```

Points to note:

- The syntax gets messy (see K&R).
  - `void (*f)` is a *pointer* to a function which is void. This is *not* the same as:
  - `void* f` – which is a function that returns a `void*`
- In the prototype, we have to specify the number and types of arguments
- We use the special pointer `void*`. to mean "this is a pointer, but we don't know what type". *Eventually* we have to know the type (in function `f1` and `f2`
- Any pointer can be cast to `void*`

inline

- The `inline` keyword allows a function to be expanded "inline".
- The use of `inline` makes macros almost redundant (which is why I didn't tell you about them).
- A function which is *defined* with its class *declaration* is automatically inline.
- The definition must be in the header file (or where the code is used – how else could it be inlined?
- Access functions are often inlined.

> *Style*
>
> **Use `inline` sparingly. Look at the performance *first* before deciding to inline a function.**

```
#ifndef __POINT_HH   // Point1.hh
#define __POINT_HH
#include <math.h>
#include <iostream.h>

class Point {
public:
  Point(int initX=0, int initY=0);
  int x() { return m_x; }   // this will be inlined
  int y() { return m_y; }   // this will be inlined
  inline int r();  // as too will this
  friend ostream& operator<<(ostream& os, const Point&);
private:
  int m_x, m_y;
};
int Point::r() {
  return (int)sqrt( m_x*m_x + m_y*m_y );
}
#endif // __POINT_HH
```

The calling code is the same – it doesn't know whether or not a member function is `inline`.

```
#include "Point1.hh"   // Inline.cc

int main() {
  Point* p = new Point(3,4);
  cout <<"Point: "<< *p <<", r="<< p->r() << endl;
  return 0;
}
```

recursion

- C++ supports recursion – calling function `foo` from inside `foo`.
- All recursive functions must have a termination condition.

Recursion should be used carefully. Sometimes it is very efficient. Sometimes it is very inefficient.

```
#include <iostream.h>   // factorial.cc
#include <assert.h>

double factorial(int n) {
  assert(n>=0);
  if   (n<=1) return 1.0;
  else return n*factorial(n-1);
}

int main() {
  for (int i=0; i<100; i++) {
    cout << i << "! = " << factorial(i) << endl;
  }
  return 0;
}
```

other types: String

- We've already met the `String` class – declared in `String.h`
- In C, strings are represented by an array of `char`. `class String` is just an encapsulation of `char*` – with some member functions.
- Since `String` is fairly recent, you will see both `String` and `char*`.
- Member functions: look in `String.h`
  Note: `String.h` makes the old `string.h` *almost* redundant.

`class String` contains a pointer to a `char*`, so be careful when making classes containing `String` persistent.

other types: enum

- C++ supports an *enumeration* type, `enum` – a type that allows only certain integer values.
- It is often used in a way similar to `static` to define constant values for a class.
- It is a type in its own right – `int` cannot be cast to `enum` (but vice versa is OK).
- It can be used with, or without, declaring an `enum` type.

```
#ifndef __FONTSIZE_HH   // FontSize.hh
#define __FONTSIZE_HH

class FontSize {
public:
  enum { TINY, SMALL, NORMAL, LARGE, HUGE };
};
#endif // __FONTSIZE_HH
```

```
#include <iostream.h>   // FontSize.cc
#include "FontSize.hh"

int main() {
  cout << FontSize::SMALL << endl;
  return 0;
}
```

```
#ifndef __PIXEL_HH   // Pixel.hh
#define __PIXEL_HH
#include <iostream.h>
#include "Point1.hh"

class Pixel : public Point {
public:
  enum Color { BLACK, WHITE, RED, GREEN, BLUE };
  Pixel(int initX=0, int initY=0, Color initColor=BLACK);
  friend ostream& operator<<(ostream& os, const Pixel&);
private:
  Color m_color;
};
#endif // __PIXEL_HH
```

```
#include "Pixel.hh"   // Pixel.cc

Pixel::Pixel(int initX, int initY, Color initColor)
  : m_color(initColor), Point(initX, initY) {}

ostream& operator << (ostream& os, const Pixel& p) {
  os << (Point)p << ", color=" << p.m_color;
  return os;
}
```

Note:

- the use of the type `Color`
- the cast to `Point` for the `<<` argument

```
#include "Pixel.hh"   // Enum.cc

int main() {
  const kPixels(3);
  Pixel* p[kPixels];
  p[0] = new Pixel(3,4, Pixel::RED);
  p[1] = new Pixel(6,7, Pixel::GREEN);
  p[2] = new Pixel(5,9, Pixel::BLUE);
  for (int i=0; i<kPixels; cout<<*p[i++]<<endl ) {}
  return 0;
}
```

Note the use of an array of pointers to an object (deferring calling the constructor).

nested classes

- A class can be defined inside the scope of another class. This is a *nested* class.
- The access rules are the same as for any other member object:
  - the class can be `public` or `private`
  - the scope resolution operator, `::` is needed outside the class.
- This is useful if the nested class only has meaning in the context of the outer class.

```
class Outer {
public:
  ...
  class Inner {
  public:
    Inner();
    ...
  private:
    ...
  };
private:
  ...
};
```

```
#include "Nested.hh"   // Nested.cc

int main() {
  Outer a;
  Outer::Inner b(2.7);
  cout << "Outer = " << a << endl;
  cout << "Outer::Inner = " << b << endl;
  return 0;
}
```

- Since class `Inner` is a member of class `Outer`, we have to refer to it as `Outer::Inner`
- The previous `enum` example is really a nested class

typedef

- A `typedef` allows us to define a new type in terms of an old one.
- Syntax: `typedef float Float`
  makes the new type `Float` a synonym for `float`
- We can use more complicated declarations:

  ```
  typedef Stack<int> intStack
  ```
- In C, `typedef` was as good as could be done. In C++, we need it far less. They are most often encountered in standard header files.

struct

- Another hangover from C. In C, a *struct* was like a class with only public data members.
- In C++, a `struct` is *almost* like a `class` *except* that the default access is `public`.
- Unlike in C, a `struct` can also have member functions, inheritance, etc.

Don't use `struct` in C++. Always use `class`.

multiple inheritance

- C++ supports multiple inheritance (Java does not). A derived class inherits the members of *multiple* base classes.

Style	**Other things being equal, multiple inheritance should generally be avoided.**

- The inheritance family tree can get very knotted.
- Data and functions of independent base classes can interfere
- Often, it's not inheritance we need at all

```
#include <iostream.h>   // Multiple.hh

class Base1 {
public:
  Base1(int initX1) { m_x1=initX1; }
  int x1() const { return m_x1; }
private:
  int m_x1;
};

class Base2 {
public:
  Base2(float initX2) { m_x2=initX2; }
  float x2() const { return m_x2; }
private:
  float m_x2;
};
```

```
#include "Multiple.hh"   // Multiple.cc

class Derived : public Base1, public Base2 {
public:
  Derived(int initX1=0, float initX2=0.0f)
    : Base1(initX1), Base2(initX2) {}
  friend ostream& operator<<(ostream&, const Derived&);
};

ostream& operator<<(ostream& os, const Derived& d) {
  os << "(" << d.x1() << "," << d.x2() << ")";
  return os;
}

int main() {
  Derived a(42, 3.14159);
  cout << "a = " << a << endl;
  return 0;
}
```

“is a” and “has a”

The “litmus test” for whether to use inheritance is the “is a” vs. “has a” test.

- If an object of class `A` *is* an object of class `B`, then use inheritance.
- If an object of class `A` *has* an object of class `B`, then use a container class.

Unfortunately, like all definitive tests, this one isn’t. But it’s a good start.

container classes

Another version of the same test is:

- Could an object of class `A` have several objects of class `B`?

If so, then we almost certainly don't want to use inheritance, but rather a container class.

- A container class simply "contains" objects of other classes. The objects could be:
  - the objects themselves
  - pointers to the objects. Sometimes, it's useful to make a "wrapper" class for the pointer.

```
#ifndef __HOUSE_HH   // House.hh
#define __HOUSE_HH
#include <iostream.h>
#include <String.h>
class Room;
class House {
public:
  House(const String& name) : m_name(name), m_n(0), m_rooms(0) {}
  void addRoom(const Room&);
  ~House();
  friend ostream& operator<<(ostream&, const House&);
private:
  String m_name;
  int m_n;
  Room* m_rooms;
  House(const House&);      // don't allow copy constructor
  void operator=(const House&);  // nor assignment operator
};
#endif // __HOUSE_HH
```

```
#ifndef __ROOM_HH   // Room.hh
#define __ROOM_HH
#include <iostream.h>
#include <String.h>

class Room {
public:
  Room() {}
  Room(const String& name, float l=0.0f, float w=0.0f)
    : m_name(name), m_l(l), m_w(w) {}
  Room& operator=(const Room&);
  ~Room() {}
  float area() const { return m_l*m_w; }
  friend ostream& operator<<(ostream&, const Room&);
private:
  String m_name;
  float m_l;
  float m_w;
};
#endif // __ROOM_HH
```

```
#include "House.hh"   // House-Room.cc
#include "Room.hh"

int main() {
  House h("123 Any Street, Newtown");
  h.addRoom(Room("Living", 25, 20));
  h.addRoom(Room("Bedroom #1", 20, 17));
  h.addRoom(Room("Bedroom #2", 16, 12));
  h.addRoom(Room("Bedroom #3", 12, 8));
  h.addRoom(Room("Kitchen", 16, 13));
  cout << h << endl;
}
```

Points to note:

- the forward declaration of class `Room`.
- class `House` *contains* an array of rooms
- The copy constructor and `operator=()` are declared `private`, but *not* defined. This prevents their inadvertent use.
- We need a destructor (since we'll be dynamically allocating memory).
- class `Room` is the contained object. If it dynamically allocated memory, we'd need copy constructors, etc.
- We explicitly define the default constructor (for use by `new`).
- We add objects to `House` with:

  ```
  h.addRoom(Room("Living", 25, 20));
  ```

  (that's where the work is, but you know how to do that.)

wrapper classes

- A “bare” pointer can be dangerous, for all the usual reasons
- It’s often good to protect ourselves (this will be done repeatedly in STL) by “wrapping” the pointer in a class.
- By ensuring that this wrapper class has the usual army of:
  - copy constructor
  - default constructor
  - destructor
  - assignment operator

  we can make it “container safe”.
- The `String` class is such a class. We’ll do similar with class `Wrapper`.

```
#ifndef __WRAPPER_HH   // Wrapper.hh
#define __WRAPPER_HH
#include <iostream.h>
class Foo {
public:
  int m_size;  char* m_array;
};
class Wrapper {
public:
  Wrapper();
  Wrapper(char*);
  Wrapper(const Wrapper&);
  Wrapper& operator=(const Wrapper&);
  virtual ~Wrapper();
  friend ostream& operator<<(ostream&, const Wrapper&);
private:
  Foo* rep;
};
#endif // __WRAPPER_HH
```

```
#ifndef __ROOM_HH   // Room2.hh
#define __ROOM_HH
#include "Wrapper.hh"

class Room {
public:
  Room() {}
  Room(const Wrapper& name, float l=0.0f, float w=0.0f)
    : m_name(name), m_l(l), m_w(w) {cout<<"Room constructor"<<endl;}
  ~Room() {}
  Room(const Room&);
  float area() const { return m_l*m_w; }
  friend ostream& operator<<(ostream&, const Room&);
private:
  Wrapper m_name;
  float m_l;  float m_w;
};
ostream& operator<<(ostream& os, const Room& r) {
  os<<r.m_name<<", L="<<r.m_l<<", W="<<r.m_w<<", area="<<r.area();
  return os;
}
```

```
#include "Room2.hh"   // Room-Wrapper.cc

int main() {
  Room* a = new Room("room 1", 23, 17);
  cout << *a << " ----------------" << endl;
  Room b=*a;
  delete a;
  cout << b <<  " ----------------" << endl;
  for (int i=0; i<1; i++) {
    Room c("room", 12*(i+1), 8*(i+1));
    cout << c <<  " ----------------" << endl;
  }
  Room d(b);
  cout << d <<  " ----------------" << endl;
}
```

Points to note:

- The only data in class `Wrapper` is a pointer to `Foo`
- The data in `Foo` are public, but `Wrapper`'s instance (`rep`) is still private.
- `Wrapper` has the usual army of constructors, destructor, etc.
- We put print statements in `Wrapper`'s constructors, etc. to make it clear what's happening.
- Because the pointer is "wrapped", class `Room` can use the default constructors, etc. It thinks `Wrapper` is a "regular" class.
- We exercise the class with objects being created, deleted, going out of scope.

```
Wrapper::Wrapper(char* s)
Wrapper::Wrapper(const Wrapper& w)
Room constructor
Wrapper::~Wrapper()
room 1, length=23, width=17, area=391 ----------------
Wrapper::Wrapper(const Wrapper& w)
Wrapper::~Wrapper()
room 1, length=23, width=17, area=391 ----------------
Wrapper::Wrapper(char* s)
Wrapper::Wrapper(const Wrapper& w)
Room constructor
Wrapper::~Wrapper()
room, length=12, width=8, area=96 ----------------
Wrapper::~Wrapper()
Wrapper::Wrapper(const Wrapper& w)
room 1, length=23, width=17, area=391 ----------------
Wrapper::~Wrapper()
Wrapper::~Wrapper()
```

exceptions

- The idea behind exceptions is:
  - A *function* knows best how to *detect* an error.
  - The *calling* program knows best how to *handle* the error.
- We could simply call `exit`, or use `assert`, but that's often too drastic.
- Conventionally, we second guess all possible errors and *avoid* them. We usually can't anticipate *all* errors
- Instead, an *Exception Handler* lets the function detect the error, and the calling program handle it. This is what we want.

Syntax:

1. The function is called inside a `try` block.
2. If the function detects an error, it `throw`s an object.
3. This object is caught by the calling program inside a `catch` block.
4. There can be multiple `catch` blocks – each catching a different object.

Exceptions break the normal program flow control:

1. If no exception was thrown:
   - the function exits normally
   - the whole `try` block is executed
   - none of the `catch` block is executed
2. If an exception was thrown:
   - the function exits at the `throw` statement
   - the rest of the `try` block is skipped
   - the relevant `catch` block is executed

In both cases, control resumes after the last `catch`.

```
#include <iostream.h>   // Exception1.cc
#include <math.h>

double mySqrt(double x) {
  if ( x<0 ) { throw "argument must be >= 0"; }
  return sqrt(x);
}
int main() {
  double x;
  while (1) {
    cout << "Enter a number: " << ends;
    if ( !(cin>>x) ) { cout << endl; break; }
    try {
      cout << "sqrt(" << x << ") = " << mySqrt(x) <<endl;  }
    catch(char* message) {  cout << message << endl;  }
  }
  return 0;
}
```

Points to note:

- To enable exceptions with `g++`, we have to use the flag:

  `g++ -Wall -fhandle-exceptions`

- The exception object is a `char*` – which is a built-in type.

- More generally, the `throw` is calling the constructor for the thrown object.

- The `catch` looks just like a function, with the thrown object as its argument.

And now for a more complicated example:

```
class ArraySizeError {   // Exceptions.hh
public:
  ArraySizeError(int size) : m_size(size) {}
  int badArraySize() { return m_size; }
private:
  int m_size;
};
class AllocateError {
public:
  AllocateError() {}
  int badAllocate() { return -1; }
};
class SubscriptError {
public:
  SubscriptError(int i) : m_i(i) {}
  int badArraySubscript() { return m_i; }
private:
  int m_i;
};
```

```
#ifndef __EXCEPTIONARRAY_HH
#define __EXCEPTIONARRAY_HH
#include "Exceptions.hh"

template <class T>
class ExceptionArray {
public:
  ExceptionArray(int);
  virtual ~ExceptionArray();
  T& operator[](int);
private:
  int m_size;
  T*  m_array;
  ExceptionArray(const ExceptionArray&);            // disable
  ExceptionArray& operator=(const ExceptionArray&);  // disable
};

template <class T>
ExceptionArray<T>::ExceptionArray(int size) : m_size(size){
  if (size<=0) { throw ArraySizeError(size); }
  m_array = new T[m_size];
  if (m_array==0) { throw AllocateError(); }
}

template <class T>
ExceptionArray<T>::~ExceptionArray() { delete [] m_array; }

template <class T>
T& ExceptionArray<T>::operator[](int i) {
  if ( (i<0) || (i>=m_size) ) { throw SubscriptError(i); }
  return m_array[i];
}
#endif // __EXCEPTIONARRAY_HH
```

```
#include <iostream.h>   // ExceptionArray-Throw.cc
#include "ExceptionArray.hh"
main() {
  try {
    ExceptionArray<int> a(-7);
    cout << "we shouldn't get here" << endl;  }
  catch(ArraySizeError e) {
    cout <<"caught ArraySizeError: "<< e.badArraySize() <<endl; }

  try {
    ExceptionArray<int> b(6);
    cout << "b[13]=" << b[13] << endl;  }
  catch(ArraySizeError e) {
    cout <<"caught ArraySizeError: "<< e.badArraySize() <<endl; }
  catch(SubscriptError e) {
    cout <<"caught SubscriptError: "<< e.badArraySubscript()<<endl; }
  catch(AllocateError e) {
    cout <<"caught AllocateError: "<< e.badAllocate() <<endl; }
}
```