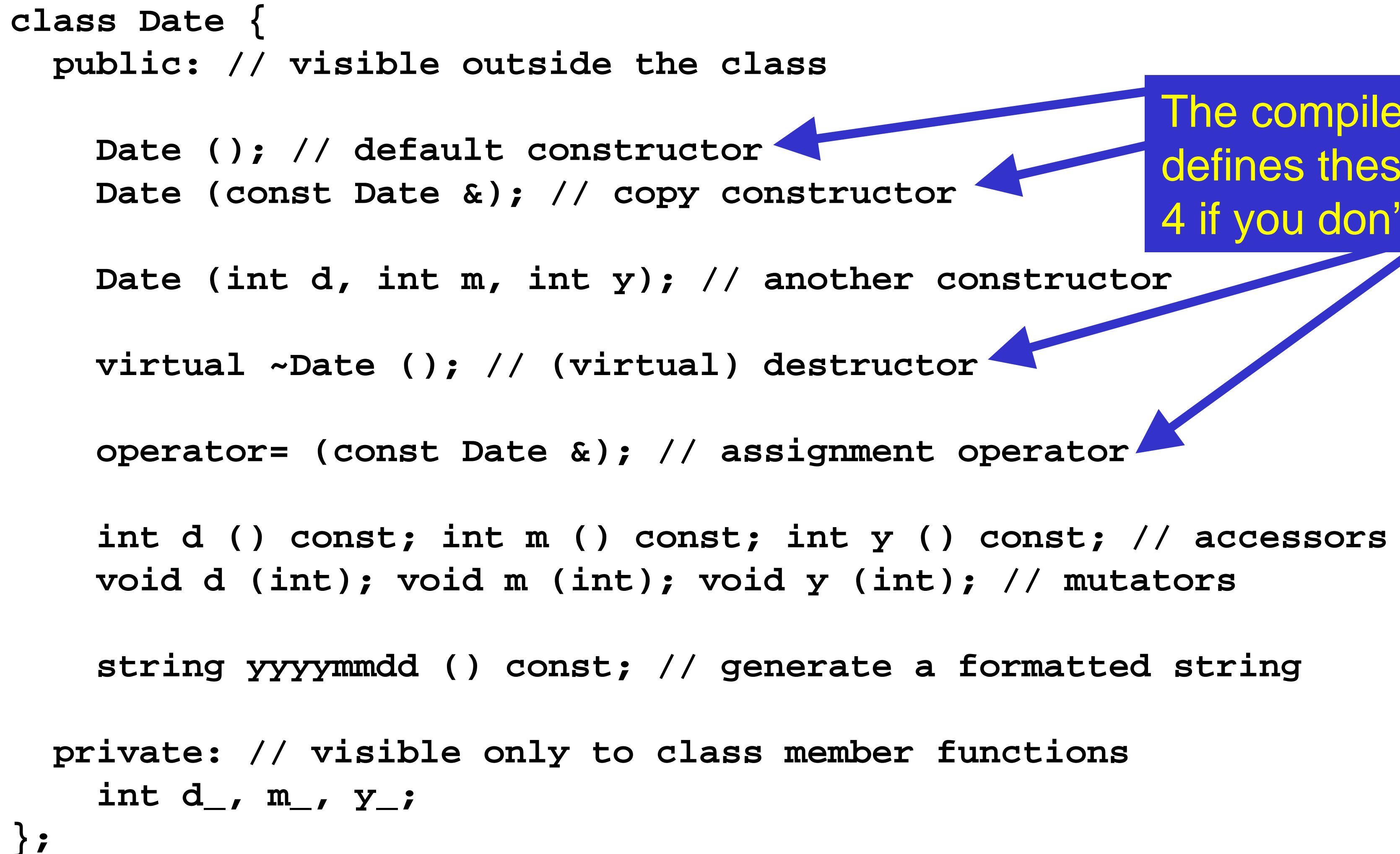
From Procedural to Object-oriented Programming

- Procedural programming
  - Functions have been the main focus so far
    - Parameters, return values, call stack
  - Mostly viewed data, functions as being separate

- Object-oriented programming
  - Allows us to package data and functions together
    - Makes data more interesting (adds behavior)
    - Makes functions more focused (restricts data scope)
  - Next 3 lectures will focus on OO programing
    - Using classes/structs, member functions/variables
    - New ideas like inheritance, polymorphism, substitution

Structure of a Simple C++ Class




```
class Date {
  public: // visible outside the class

    Date (); // default constructor
    Date (const Date &); // copy constructor

    Date (int d, int m, int y); // another constructor

    virtual ~Date (); // (virtual) destructor

    operator= (const Date &); // assignment operator

    int d () const; int m () const; int y () const; // accessors
    void d (int); void m (int); void y (int); // mutators

    string yyyymmdd () const; // generate a formatted string

  private: // visible only to class member functions
    int d_, m_, y_;
};
```

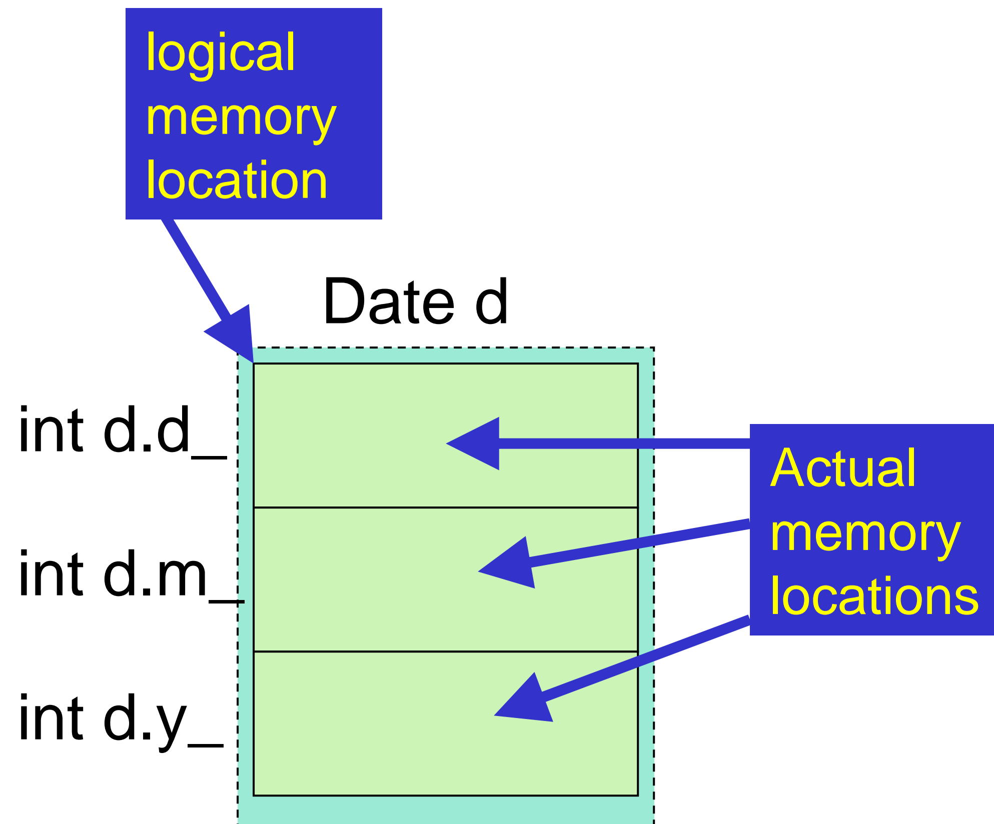
Class Members and Memory Locations

```
class Date {

public:
  Date ();
  Date (const Date &);
  Date (int d, int m, int y);
  virtual ~Date ();
  operator= (const Date &);
  int d () const;
  int m () const;
  int y () const;
  void d (int);
  void m (int);
  void y (int);
  string yyyymmdd () const;

private:
  int d_, m_, y_;
};
```

class members live in whichever memory segment an object (a *class instance*) was created



Access Control

- Declaring access control scopes within a class
  - `public`: visible outside the class
  - `protected`: visible to derived classes (more later)
  - `private`: visible only within the class
  - – Access control in a class is `private` by default
    - but, better style to label access control scopes explicitly
- A `struct` is the same as a `class`, except
  - – Access control for a struct is `public` by default
  - – Usually used for things that are “mostly data”
  - – Versus classes, that have both data and behavior

Issues with Encapsulation in C++

- Sometimes two classes are closely tied
  - For example, a container and its iterators
  - One needs direct access to the other's internal details
  - But other classes shouldn't have such direct access
- Poses interesting design forces
  - How should iterator access members of container?
  - Making container members public violates encapsulation
    - Any class, not just iterator could modify them
  - Make protected, derive iterator from container?
    - Could work: inheritance for implementation
    - But, may prove awkward for a number of other reasons
  - Could have lots of fine-grain accessors and mutators
    - Functions to get and set value of each member variable
    - But that would clutter the interface for other classes

Friends

- Offer a limited way to open up class encapsulation
- C++ allows a class to declare its "friends"
  - Give access to specific classes or functions
    - A "controlled violation of encapsulation"
  - Keyword `friend` is used in class declaration
- Properties of the friend relation in C++
  - Friendship gives complete access
    - Friend methods/functions behave like class members
    - public, protected, private scopes are all accessible by friends
  - Friendship is asymmetric and voluntary
    - A class gets to say what friends it has
    - But one cannot "force friendship" on a class
  - Friendship is not inherited
    - Specific friend relationships must be declared by each class
    - "Your parents friends are not necessarily your friends"

Friends Example

```
class Foo {
  friend ostream &operator<<(ostream &out, const Foo &f);
public:
  Foo(int) {}
  ~Foo() {}

private:
  int baz;
};

ostream &operator<<(ostream &out, const Foo &f) {
  out << f.baz;
  return out;
}
```

Class declares operator<< as a friend

- Gives access to member variable baz

Can now print Foo like a built-in type

```
Foo foo;
cout << foo << endl;
```

Constructors

```
class Date {
  public:
    Date ();
    Date (const Date &);
    Date (int d, int m, int y);
    // ...
private:
    int d_, m_, y_;
};

Date::Date ()
 : d_(0), m_(0), y_(0)
{}

Date::Date (const Date &d){
 : d_(d.d_), m_(d.m_), y_(d.y_)
{}

Date::Date (int d, int m, int y)
 : d_(d), m_(m), y_(y)
{}
```

- A constructor has the same name as its class
- Establishes invariants for the class instances (objects)
  - Properties that always hold
  - Like, no memory leaks
- Notice parameters given in base/member initialization list
  - Members constructed in order they were declared
    - List should follow that order
  - Set up invariants before the constructor body is run
  - Help avoid/fix constructor failure

More About Default and Copy Constructors

- Usually have default and copy constructors
  - Compiler provides them if you don’t
  - Defined as member-wise default or copy construction (respectively)
  - Declare private, don’t define if you don’t want them
- Default / copy construction of built-in types
  - Default construction does nothing (uninitialized)
  - Copy construction fills in the value given
- Default constructor takes no arguments
  - Can supply default values via member list
  - Must do this for const and reference members

Destructors

```
class Date {
  public:
    virtual ~Date (); // ...
  private:
    int d_, m_, y_;
};

Date::~Date () {
  // nothing to do here
}

class Calendar {
  public:
    virtual ~Calendar (); // ...
  private:
    size_t size; Date * dates;
};

Calendar::~Calendar () {
  delete [] dates;
}
```

- Constructors initialize objects
  - At start of object's lifetime
  - implicitly called when object is created (can call explicitly)
- Destructors clean up afterward
  - At end of object's lifetime
  - implicitly called when object is destroyed (can call explicitly)
  - Compiler provides if you don't
    - Defined as member-wise destruction
- Common scenario with dynamic memory management
  - Constructor sets pointer to 0
  - Constructor or another function allocates (calls new or new [])
  - If another function deallocates, sets pointer back to 0 when appropriate
  - Destructor deallocates if needed (calls delete or delete [])

More on Initialization and Destruction

- Initialization follows a well defined order
  - Base class constructor is called
    - That constructor recursively follows this order, too
  - Member constructors are called
    - In order members were declared
    - Good style to list in that order (compiler may warn if not)
  - Constructor body is run
- Destruction occurs in the reverse order
  - Destructor body is run
  - Member destructors are called
  - Base class destructor is called
    - That destructor recursively follows this order, too
- Declare destructor as virtual (more on this next week)

Tips for Initialization and Destruction

- Use base/member initialization list if you can
  - To set pointers to zero, etc. before body is run
  - To initialize safe things not requiring a loop, etc.
- Do things that can fail in constructor body
  - Rather than in the initialization list
  - For example, memory allocation, etc.
- Never let an exception leave a constructor
  - Otherwise in a partially initialized (“zombie”) state
  - No guarantee destructor will be called if destroyed
- Use try/catch, set object to default state if it fails

Self Reference

- All non-static member functions are invoked with implicit pointer to object on which function was called
  - non-const member function: `X * const this`
  - const member function: `const X * const this`
- The "this" pointer is implied when member variables are accessed from within a member function
- If passed a pointer or reference as a parameter
  - May need to compare against this (see assignment, next)
- You can use this to return a reference or pointer to the object on which the member function was called

```
Date& Date::add_year(int n)
{
  if (…) {…}         // check for leap year
  _y += n;           // this-> implied
  return *this;
}

// allows Date d; d.add_year(3).add_year(5);
```

Assignment Operator

```
class Date {
  public:
    operator= (const Date &);
    // ...
private:
    int d_, m_, y_;
};

Date::operator= (const Date &d){
  d_ = d.d_;
  m_ = d.m_;
  y_ = d.y_;
}

int main (int, char *[])
{
  Date a; // default constructor
  Date b(a); // copy constructor
  Date c = b; // copy constructor
  a = c; // assignment operator
}
```

- Compiler supplies if you don't
  - Does member-wise assignment
- Similar to copy constructor
  - But must deal with existing value
  - And no initialization list
- Watch out for self-reference
  - Assignment to self

    s = s; // perfectly legal
  - Efficiency, correctness issues
- Watch out for aliasing/copying
  - Copying an `int` vs. an `int *`
  - Copying an `int` vs. an `int &`
  - Copying an `int` vs. an `int []`
  - More on this throughout semester

Accessor and Mutator Functions

```
class Date {

  public:
    // ...
    int d () const;
    int m () const;
    int y () const;
    void d (int);
    void m (int);
    void y (int);
    // ...

private:
    int d_, m_, y_;
};
```

- May want to give access to private class members
  - Public scope is common with structs
  - Private scope is common with classes
    - With non-member operators as friends
- Accessors
  - Member functions to "read" values
  - Return by value or by const reference
  - Functions often const (see next slide)
- Mutators
  - Member functions to "write" values
  - Return type is often void
  - Parameter same type as member variable
- Often see one of two main styles

```
int d () const; void d (int);
int get_d () const; void set_d (int);
```

Const Member Functions

```
class Date {

public:
   // ...
   int d () const;
   int m () const;
   int y () const;
   string yyyymmdd () const;
   // ...

private:
     int d_, m_, y_;
};

int main (int, char *[])
{
   const Date c (4, 10, 2005);
   cout << c.yyyymmdd () << endl;
   return 0;
}
```

- Can declare member functions as being “const”
  - Promise not to modify member variables when called
- Const member functions can be called on const objects
  - Can’t call non-const member functions on const objects
- Const rule doesn’t apply to constructors or destructor
  - Const objects are still created and destroyed
  - Just can’t be modified between creation and destruction

Static Member Variables and Functions

- A static member is part of class but not of any particular object (instance of the class)
- Declaration
  - Use *static* keyword before variable or function in .h file (not in .cc file)
- Memory Segment
  - Global, one instance of static per class (not per object)
- Access
  - Non-static member functions can access static member functions and static member variables
  - Static member functions cannot access non-static member functions or non-static member variables
  - Static variable or function accessed from outside class using class name
  - From inside class, can just use its name (but can’t precede with this->)

Static Class Members

```
class Date {
public:
  // ...
  static void set_default(int,int,int);
private:
  int _d, _m, _y;
  static Date default_date;
};

Date::Date ()
 : _d (default_date._d),
   _m (default_date._m),
   _y (default_data._y)
{}

Date::operator= (const Date &d){
  this->d_ = d.d_;
  this->m_ = d.m_;
  this->y_ = d.y_;
}
```

```
Date Date::default_date(01,01,2004);

void Date::set_default(int m, int d, int y)
{
    Date::default_date = Date(m, d, y);
}
```

- Must define static members, usually outside of class
  - Initialized before any functions in same compilation unit are called
- Static member functions don't get implicit this parameter
  - Can't see non-static class members
  - But non-static member functions can see static members

Initializing Static Member Variables

- Static integral constants may be initialized within class declaration by a constant expression

```
static const int MAX = 100;
```

  - you must then also (only) define it outside of class

```
const int MAX; // don't reinitialize
```

- More common to just declare static member in the class

```
static Date default_date;
```

  - and then initialize at its definition outside the class

```
Date Date::default_date(01,01,2004);
```

For Next Time

- Topic: C++ Classes++
- Discussion
  - Dynamic memory management
  - Access control tricks with constructors, destructors
  - Temporary variables and object lifetimes
  - Introduction to inheritance
  - Classes, inheritance, and exceptions