
Object Oriented Programming and Design with C++

Aaron Naiman

Jerusalem College of Technology

naiman@jct.ac.il

<http://jct.ac.il/~naiman>

Copyright ©2012 by A. E. Naiman

Overview of Classes

- ⇒ Motivation
 - Five Stages
 - Constructors and Destructor
 - Other Member Functions
 - Inheritance

Class Aggregation

- Classes provide method for logical grouping
- Groupings are of both data and functionality
- *Data members* = objects (aka: variables, fields), perhaps themselves of other classes
- *Member functions* = actions or operations
 - * usually applied to data members
 - * *important*: called by objects
- New defined class = *abstract data type*
- Built by class library programmers
- Can build in protection; conceal from application programmer

Let's look at some examples,

Triangles in 3-Space

- Without classes:

```
float vertex_1_x, vertex_1_y, vertex_1_z,  
      vertex_2_x, vertex_2_y, vertex_2_z,  
      vertex_3_x, vertex_3_y, vertex_3_z;
```

- We need all this *for each* triangle
- 9 assignments to assign one triangle to another !!
- Likewise, 9 function arguments !!
- Advanced: What about automatic functions for:
 - * the area of a triangle
 - * drawing triangles (maybe even cout)

We'll return to coord and triangle classes.

Database of Student Information

```
const int db_size = 10000, id_size = 10;
char *db_names[db_size], db_ids[db_size][id_size];
double db_weights[db_size];    // okay, maybe floats :-)
short db_num_courses[db_size], db_gpas[db_size];
// ...
```

- More logical to group all info per student
 - * All “db_”s and “db_size”s tell of repeated code, need to associate
 - * Should group even for different types of info
- Nice if free store allocation were automated (where?)
- How about an understood “cout <<” call?
- Can we retain confidentiality of db_gpas []?

Complex Numbers

- Lack of data type is serious deficiency in C and C++ (vs. Fortran)
- Want to *bind* real and imaginary parts into one object \Rightarrow allow use of `c2 = c5;`
- Desired mathematical functionality for application programmer:
 - * simple arithmetic operations, e.g.: `+`, `-`, `*`, `/`, `...`
 - * other capabilities: `sin()`, `cos()`, `rho()`, `dist()`, `...`
- Also: nice I/O interface: `(4.2, -39.34i)`
- Note: no pointer fields

Let's look into this example some more

Overview of Classes

- Motivation
- ⇒ Five Stages
- Constructors and Destructor
- Other Member Functions
- Inheritance

The int Class

```
{ /* ... */ int i; /* ... */ }
```

- Two steps:
 - * At definition of `i`: allocation of object `i` of class `int`
 - * At “}”: `i` “leaves scope”, deallocated

```
{ /* ... */ int i = 3; /* ... */ }
```

- Additional step of initialization right after allocation
- For classes
 - * this initialization includes other initial setting up = *constructors*
 - * there is also a final clean up step *prior* to deallocation = *destructor*

The Life of an Object

This all happens *to an object* of the class (note the symmetry):

- [1] allocation of data members
- [2] construction
- [3] usage
- [4] destruction
- [5] deallocation of data members

An example from real life,

Band Performance

- [1] arrival: take over the stage
- [2] setup: check electricity, sound, tune-up, ...
- [3] jam: take requests, ignore them, play music
- [4] breakdown: detach equipment, pack up (get paid?)
- [5] departure: release stage (for monkey act, jugglers, ...)

Back to the complex class library ...

Overview of Classes

- Motivation
- Five Stages
- ⇒ Constructors and Destructor
- Other Member Functions
- Inheritance

complex Class Skeleton

```
class complex {    // new type with name "complex"
public:           // public interface, can be
    // ...       // accessed by all functions
protected:     // hidden info/functions for use
    float re, im; // by member functions only
};              // note ";", vs. end of function
```

- public:/protected: is *only* a permissions issue
- public: adds an additional layer between application programmer and class implementation (i.e., data members)
- Reasons for information hiding—later (any guesses?)
- What functions should be in public:?

Let's start with creation and extinction.

Class Constructors

- Responsibility of *all* constructors
 - * initialization of *all* data members
 - * perhaps: set globals, e.g., number of objects
 - * perhaps: special I/O—for debug only (why?)
- *Important*: constructor is “called” by an object and effects the object (like all member functions)
- Has the same name as class
- Does not specify a return type or return a value
- Is not (usually) called explicitly, but automatically (Stage 2) after creation of new object (Stage 1)
- Two special constructors: *default* and *copy*

complex Default Constructor—Take I

- *Default* constructor = willing to take no arguments

```
// application usage
```

```
complex c1, *cp = &c1;    // just as with an int
```

```
// ... outside class def
```

```
const float def_re = 0.f, def_im = 0.f;
```

```
// ... in public: section (BTW: sizeof(c1) == ??)
```

```
complex(void) {re = def_re; im = def_im;}
```

- Rule: member function definition included in class def. \Rightarrow automatically (suggested to be) `inline`
- `re` and `im`: declared by and belong to calling object (`c1` above)
- Note: constructor *not* called for `cp`
- Construction is enough for `const` object initialization

How about constructors with initialization?

complex Default Constructor—Take II

```
complex c2, c3(3.5f),    // NOT: c2()
        c4 = -1.f, c5(-7.f, .2f);
// ... definition
complex(const float& in_re = def_re,
        const float& in_im = def_im) {
    re = in_re; im = in_im;
}
```

- Due to responsibility, supply args for all data members
- Note: `c2()` declares a function which returns a `complex`
- Any constructor call with single argument can use “=” form
- How about same-class, object-to-object initialization, e.g.:

```
int i, j = i;
```

complex Copy Constructor

- Purpose: to initialize with another complex object

```
complex c6(c3);    // just like: int i(-4);
```

```
complex c7 = c2;   // just like: int j = 3;
```

definition:

```
complex(const complex& rhs) {    // why reference type?  
    re = rhs.re; im = rhs.im;    // "." for member access  
}
```

- Again: c6 and c7 are *calling* the constructors
c3 and c2 are the arguments
- rhs.re and rhs.im
 - * since member function, access even to different object's protected members
 - * but rhs. necessary to specify other object

complex Destructor

```
~complex(void) {} // again ...
```

- Placed in `public:` section
- Has the same name as class, prepended with `~`
- Does not return a value
- Is *not* called explicitly, but automatically (Stage 4) *prior* to deallocation of an `complex` object (Stage 5)
- Primary purpose/responsibility: cleanup (nothing needed for `complex`)

What do we have so far?

complex Con/Destructors

```
const float def_re = 0.f, def_im = 0.f; // all in complex.h
                                         // with COMPLEX_H
class complex {                           // envelope ...
public:
    complex(const float& in_re = def_re,
            const float& in_im = def_im) {
        re = in_re; im = in_im;
    }
    complex(const complex& rhs) {
        re = rhs.re; im = rhs.im;
    }
    ~complex(void) {}
    // ...
protected:
    float re, im;
};
```

But this is kind of boring, ...

Overview of Classes

- Motivation
- Five Stages
- Constructors and Destructor
- ⇒ Other Member Functions
- Inheritance

View Member Functions

- To view (i.e., read from) data members:

```
float f = c2.real(), g = cp -> imag();
```

```
// definitions:
```

```
float real(void) const {return re;}
```

```
float imag(void) const {return im;}
```

- Required, as application programmer has no access to protected re
- `const` refers to calling object (more later); use when possible
- Again: “.” for member access
- For member access, and for pointers to objects:
 $a \rightarrow b \equiv (*a).b$ (note: $(*a).b \neq *a.b = *(a.b)$)
- Choose user-friendly names for functions (more so than for data members)

How about member functions to modify?

Modify Member Functions

- To modify (i.e., write to) data members:

```
c5.imag(c1.real());
```

```
// definitions:
```

```
void real(const float& in_re) {re = in_re; return;}
```

```
void imag(const float& in_im) {im = in_im; return;}
```

- Exploiting function overloading
 - * for maintainability, consider `_real` and `_imag` as fields
- Functions clearly cannot be `const`
- How about object-to-object assignment? Involves:
 - * 2 reads
 - * 2 writes

How will assignment look?

complex Assignment

```
c1.operator=(c6);    // or more commonly, b|c operator:
c1 = c6;           // c1 calls function with argument of c6
// definition:
complex& operator=(const complex& rhs) {
    if (this == &rhs) return *this; // time-saving self-test
    real(rhs.real()); imag(rhs.imag());
    return *this; // copy constr. can also use real() ...
}
```

- Function name: operator=
 - Prefer real() over re—see later why (any guesses?)
 - complex (vs. void) returned for daisy-chaining; ref. for speed
- ```
c6 = c5 = c3; // c6.operator=(c5.operator=(c3));
 // match return value and argument types
```

Operator overloading is powerful and helpful.

# Assignment Self-Test

---

- Reference argument for speed and to enable self-test
- `this` contains address of calling object (`this == &c1` above)
- Reasons for self-test
  - \* speed
  - \* another reason later

- Possible self-assignment situations

```
complex& c8 = c3, *pc = &c2;
// later ...
c3 = c8; // or viceversa
*pc = c2; // ditto
```

*Don't* forget the operator=() self-test!

# Copy Constructor vs. Assignment Operator

---

- Copy construction: `complex c9 = c2;`
- Assignment: `c9 = c6;`

|                          | new object around | can daisy-chain | self-assignment check | returns reference to <code>*this</code> |
|--------------------------|-------------------|-----------------|-----------------------|-----------------------------------------|
| copy constructor         | ✓                 | ×               | ×                     | ×                                       |
| <code>operator=()</code> | ×                 | ✓               | ✓                     | ✓                                       |

Aside from viewing and modifying, . . . .

# Equality Test Operator

---

- To test equality to another `complex` object
- Meanwhile, just a field-wise “and” test
- Use the `==` operator

```
if (c7 == c2)
 cout << "Yup, they're equal!" << endl; // definition:
bool operator==(const complex& rhs) const {
 return ((real() == rhs.real()) &&
 (imag() == rhs.imag()));
}
```

- `rhs.real()` vs. `rhs.re`
  - \* safe public interface
  - \* `inline`, so no slowdown
  - \* `real()` and `imag()` are predictably `const`  $\Rightarrow$

no problem for `const` calling object or `rhs`

Use public interface when possible and not deleterious.

# Math Member Operators

---

- Addition:

```
c6 = c1 + c8;
```

```
// definition:
```

```
complex operator+(const complex& rhs) const {
 return complex (real() + rhs.real(),
 imag() + rhs.imag());
}
```

- Who's the "caller", and who's the argument?  
*Hint: Write function calls explicitly.*
- Constructor called (note: explicitly) for temporary object
- Note: both `operator==(())` and `operator+(())` are `const`
- Do the same for `-`, `*`, `...`

Current tally (all in `complex.h`) . . . .

## complex Class Definition

---

```
const float def_re = 0.f, def_im = 0.f;

class complex { // comments missing for space conservation
public:
 complex(const float& in_re = def_re,
 const float& in_im = def_im) { /* ... */ }
 complex(const complex& rhs) { /* ... */ }
 ~complex(void) {}
 float real(void) const {return re;}
 float imag(void) const {return im;}
 void real(const float& in_re) {re = in_re; return;}
 void imag(const float& in_im) {im = in_im; return;}
 complex& operator=(const complex& rhs) { /* ... */ }
 bool operator==(const complex& rhs) const { /* ... */ }
 complex operator+(const complex& rhs) const { /* ... */ }
protected:
 float re, im;
};
```

## main() Using complex Class

---

```
#include <iostream>
#include "complex.h"
using namespace std; // how much memory is de/allocated? where?
int main(void) {
 complex radar_info[] = { /* ... */ }, // init. syn. later
 assumed_average = radar_info[0];
 const int collected_data = // note maintainability
 sizeof(radar_info) / sizeof(radar_info[0]);
 bool verify_if_average(const complex *ca, int arr_len,
 const complex& average);

 cout << verify_if_average(radar_info, collected_data,
 assumed_average) << endl;
 return 0;
}
```

## verify\_if\_average() Using complex Class

---

```
bool verify_if_average(const complex *ca, int arr_len,
 const complex& average) {
 // being (needlessly) explicit
 complex sum(0.f, 0.f);

 // operator+=() would be nice here
 for (int i = 0; i < arr_len; i ++)
 sum = sum + ca[i];

 // operator/ (int) needs to be defined above
 sum = sum / arr_len;

 return (sum == average);
}
```

# Overview of Classes

---

- Motivation
  - Five Stages
  - Constructors and Destructor
  - Other Member Functions
- ⇒ Inheritance

# To Inherit or not to Inherit?

---

- Goal: to define a special type (class) of a class
- E.g., for the complex class
  - \* unit complex numbers:  $\{x \in \mathcal{C} \mid |x| = 1\}$ —requires special functions (e.g.?)
  - \* named numbers—requires an extra field as well
- Important: for inheritance, objects of subclass have an *isA* relationship to base class
- Non-inheritance of coord
  - \* e.g.: triangle, polygon, (positioned) sphere
  - \* note inappropriate operator+()
  - \* likely will *include* (list of) coord

Do the *isA* test!

# Subclass Definition

---

```
#include "complex.h" // this code placed in unit_complex.h

class unit_complex : public complex { // built on top of complex
public:
 // new constructors needed at least
protected:
 // new protected stuff (if any)
};
```

- Building on previous work  $\Rightarrow$  more maintainable
- Inherits (almost) all members and permissions from `complex`
  - \* can pass `unit_complex` object as a `complex` argument
- Still need to
  - \* define constructors
  - \* re-define member functions which have
    - ★ changed
    - ★ same overloaded name as one which has changed

# New Subclass Functions

---

```
public: // need also copy constructor
 unit_complex(const float& in_re = def_re,
 const float& in_im = def_im) :
 complex(in_re, in_im) { // more on initializers
 normalize(); // later
}
unit_complex operator+(const unit_complex& rhs) const {
 return unit_complex (real() + rhs.real(),
 imag() + rhs.imag());
} // also functions for mixed types (complex, floats, ...)
protected:
void normalize(void) { // helper member function
 real(real() / length()); // length():
 imag(imag() / length()); // 1) assume in complex
 return; // 2) need to worry
} // about length == 0
```

# Pointer Data Members

---

- ⇒ Motivation
- Constructors and Destructor
  - Accessing Pointer Members
  - Other Member Functions

# Pointers vs. Non-pointers

---

- Free store allocation (and deallocation)
  - \* pointers usually indicate this
  - \* who should do this, and keep track?
  - \* how about the constructors and destructor?
- Pointer value vs. what it points to
  - \* for object assignment, do we want memberwise assignment of pointer fields? (no!)
  - \* what should `operator==( )` mean?
  - \* for security and confidentiality—address *both*

Let's look at the popular `cstring` class.

# Strings

---

- We are constantly bitten by memory allocation issues
  - \* unallocated space
  - \* memory leaks
  - \* string copying
- Can we implement `operator=()` to perform `strdup()`?
- Operators (or at least member functions) for
  - \* `strlen()`
  - \* `strcmp()`
  - \* `strcat()`

would be nice (which operators?)

Let's crack open the `cstring` class.

# Pointer Data Members

---

- Motivation
- ⇒ Constructors and Destructor
- Accessing Pointer Members
- Other Member Functions

# The Life of a `cstring` Object

---

- [1] allocation of memory for address (`sizeof(char *)`)
- [2] perhaps `new()`
- [3] perhaps other free storing (`op=()`, ...)
- [4] perhaps `delete()`
- [5] deallocation of memory for address

Goal: conceal free store business from application programmer.

# cstring Class Skeleton

---

```
class cstring {
public:
 // ... // constructors, destructor, ...
protected:
 void set_s(const char *in_s) { /* ... */ }
 char *s;
};
```

- For information hiding, don't let application programmer know value of `s` to change it
- What functions should be in `public`?:
- `set_s()` is a helper function, similar to `strdup()`

What does `set_s()` look like?

## set\_s()

```
#include <assert> // consider being more user-friendly
#include <string>
using namespace std;

// safe and sound, all in one place (in cstring definition)
void set_s(const char *in_s) {
 s = (char *) 0; // either way, ...
 if (in_s) { // for safety
 int len = strlen(in_s);
 s = new char[len + 1]; // allocating
 assert(s != 0); // checking
 strcpy(s, in_s); // copying (NULL too)
 }
 return;
}
```

Used by constructors and others, .....

# Class Constructors

---

```
cstring(const char *in_val = 0) { // default ctor
 set_s(in_val);
}
```

```
cstring(const cstring& rhs) { // copy ctor
 set_s(rhs.s);
}
```

- Recall
  - \* constructors responsible for initialization
  - \* preferred pointer states/initialization
    - ★ allocate memory from free store
    - ★ set to address of existing object
    - ★ set to 0

# Constructor Invocations and Destructor

---

```
#include "cstring.h"

int main(void) {
 cstring s_a, s_b(s_a), s_c = "I am s_c",
 s_d = s_c; // uh-oh
 // ...
 return 0;
}
```

- When `cstring` object leaves scope, destructor (responsibility: cleanup) is called  $\Rightarrow$

Rule: call `delete` for each pointer member

```
~cstring(void) {delete [] s;}
```

- What's the problem with `s_d`? Okay syntax, but identity crisis
- How do we see it? How do we correct it?

Reading from and writing to pointer field . . . .

# Pointer Data Members

---

- Motivation
- Constructors and Destructor
- ⇒ Accessing Pointer Members
- Other Member Functions

# Pointer Field Viewing

---

```
const char *get_s(void) const {
 return (const char *) s;
}
```

- With `complex's real()`, a *copy* of `re` is returned
- So to with `get_s()`, a *copy* of `s` is returned
- *But*, `copy` refers to address of char array itself
- Application programmer can change char array indirectly! Oy!
- Returning `const` ensures that this will not happen

```
char *cs1 = s_b.get_s(); // error (or warning)
const char *cs2 = s_a.get_s(); // okay
cout << s_d.get_s() << endl; // okay, but: I am s_c?
```

How do we fix `s_d`?

# Pointer Assignment

---

```
float *fp1, *fp2;
fp1 = new float[42]; // area is allocated
fp2 = fp1; // _address_ is copied over
delete [] fp1; // area is deallocated
cout << fp2[7] << endl; // area is accessed, but undefined
```

- General hazard: two pointers to same free store allocation
- Usually, we want copy of info pointed to, not address
- Therefore
  - \* new allocation
  - \* (assertion of success)
  - \* copy info over

How about our `cstring`?

# Pointer Field Modification

---

```
s_d.modify_s("I am not s_c");
```

```
void modify_s(const char *in_s) { // poor implementation
 delete [] s; // delete to 0, no problem
 set_s(in_s);
 return;
}
```

- delete ∴ good thing s always initialized or zeroed
- Caveat: deleting before new succeeded
- For cstring, field assignment *is* object assignment ⇒  
Can we exploit operator overloading?
- What if cstring &s\_e = s\_b;, and later:

```
s_b.modify_s(s_e.get_s());
```

A more natural and safe operator for assigning, . . . .

# String to cstring Assignment

---

```
s_d = "I am indeed s_d";
```

```
cstring& operator=(const char *rhs) {
 if (s == rhs)
 return *this;
 delete [] s; // delete to 0, no problem
 set_s(rhs);
 return *this;
}
```

- Earlier reason for early-return: speed
- Now also: avoid catastrophic delete before set\_s()
- Recall: reference returned for daisy-chaining

```
s_d = s_c = s_b = "something new"; // compiles?
```

Are first two assignments defined?

# cstring Assignment Operator

---

```
// usage:
```

```
s_a = s_b;
```

```
cstring& operator=(const cstring& rhs) {
 if (this == &rhs)
 return *this;
 delete [] s; // alt. for these 2 lines:
 set_s(rhs.s); // *this = rhs.s;
 return *this;
}
```

- Note: self-test
  - \* previous: on (char \*) rvalues
  - \* here: on object lvalues

What have we learned?

# operator=() Lessons

---

- Do not forget the self-test
- Have operator=() assign to all data members (just like constructors)
- Vs. copy constructor (recall earlier table)  
    additional concern: to delete old free store allocation
- After operator=() invocation
  - \* corresponding pointer fields are *not* the same, but
  - \* point to copies of same data
- Without copy constructor and operator=() definitions
  - \* compiler *will* perform memberwise copying (bitwise)
  - \* leads to: memory leaks and redundant pointers

∃ pointer fields ⇒ define copy constr. and operator=().

# Pointer Data Members

---

- Motivation
  - Constructors and Destructor
  - Accessing Pointer Members
- ⇒ Other Member Functions

# Other cstring Capabilities

---

- Obvious one: `strlen()`:  

```
int length(void) const {return s ? strlen(s) : 0;}
```
- Equality testing (`operator==(())`)
- String concatenation (`operator+()`, `operator+=()`)
- Finding a char in a cstring
- Finding a cstring (or string) in a cstring
- Change to upper/lower case

Let's define the first two operators.

# Equality Test Operator

---

- `operator=()` is different for a pointer field  $\Rightarrow$   
What about `operator==(())`?
- Definition: equality if dereferenced data is the same
- Note: should have as well:

```
bool operator==(const char *rhs) const { ... }
```

```
bool operator==(const cstring& rhs) const {
 if (this == &rhs) return true; // time savers
 if (length() + rhs.length() == 0) return true;
 if (length() != rhs.length())
 return false;
 return !strcmp(s, rhs.s); // note "!"
}
```

# Concatenation Operator

---

```
cstring operator+(const cstring& rhs) const {
 cstring result; // creating new object for result
 const int total_length = length() + rhs.length() + 1;

 if (total_length == 1) return result;

 result.s = new char[total_length];
 assert(result.s); // or something more graceful ...

 length() && strcpy(result.s, s);
 rhs.length() && strcpy(result.s + length(), rhs.s);

 return result;
}
```

```
// a calls with argument b
cstring a = "Hi, ", b = "Mom!", c = a + b;
```

# cstring.h

---

```
#include <assert>
#include <string>
using namespace std;

class cstring { // missing documentation
public:
 cstring(const char *in_val = 0) {set_s(in_val);}
 cstring(const cstring& rhs) {set_s(rhs.s);}
 ~cstring(void) {delete [] s;}
 const char *get_s(void) const {return (const char *) s;}
 cstring& operator=(const char *rhs) {/* ... */}
 cstring& operator=(const cstring& rhs) {/* ... */}
 int length(void) const {return s ? strlen(s) : 0;}
 bool operator== const(const cstring& rhs) {/* ... */}
 cstring operator+ const(const cstring& rhs) {/* ... */}
protected:
 void set_s(const char *) {/* ... */}
 char *s;
};
```

# Using the cstring Class

---

```
#include <iostream>
#include "cstring"
using namespace std;

int main(void) {
 cstring red = "red", blue = "blue",
 purple = red, clear;

 // oops, fixing
 purple = "purple";
 if ((red + blue) == purple)
 cerr << "It's a MIRACLE! How did you get " <<
 purple.get_s () << "?" << endl;

 return 0;
}
```

# Additional Class Features

---

- ⇒ File Organization and Scope
  - Friends and Object I/O
  - Hidden Data and Static Members
  - Construction and Initialization
  - References Between Functions
  - Member Access Functions

# Inline Member Functions—Revisited

---

- Inline for (short, sweet) often-used functions
- Member function definition in
  - \* `.h`  $\Leftrightarrow$  inline (and static)  $\Rightarrow$   
automatic if in class definition
  - \* `.C`  $\Leftrightarrow$  not inline
- Otherwise, in
  - \* `.C` and inline  $\Rightarrow$   
no expansion definition for compilation
  - \* `.h` and not inline  $\Rightarrow$   
> 1 definition upon linking

Recall and beware: `inline` is only a suggestion.

# Scope and Related Globals Functions

---

- Class scope: members (outside of class definition) associated to class with scope operator “::”

```
inline complex::complex(const complex& rhs) { /* ... */ }
```

- E.g., if `op+=()` built on *global* `op+()` (see later)  $\Rightarrow$   
declare in class and define (with “::”) outside

- Recall: building a class library for others to use

- \* `.h` `#included` for interface

- \* `.C` `#includes` `.h` and is compiled away for linking

- Therefore, other related global functions, e.g.,

```
complex sin(const complex&);
```

- \* declared in `.h`

- \* defined in `.C` (again, unless `inline`)

# File Organization of Class Information

---

- `foo.h`
  - \* class definition
  - \* definitions of `inline` functions
  - \* declarations of non-`inline` functions
- `foo.C`: definitions of non-`inline` functions (`#include foo.h`)
- Lib. prog. compilation: `g++ -Wall -c foo.C (→ foo.o)`
- Later application prog. compiles (`main()` in) `prog.C`:  
`g++ -Wall -c prog.C (→ prog.o)`
- And finally application programmer links:  
`g++ -Wall -o prog prog.o foo.o`
- Or in one shot: `g++ -Wall -o prog prog.C foo.o`

# Additional Class Features

---

- File Organization and Scope
- ⇒ Friends and Object I/O
- Hidden Data and Static Members
- Construction and Initialization
- References Between Functions
- Member Access Functions

# Preference of Member Functions

---

- Situation: designing new function related to class
- Question: make the function global or a member?
- Answer in general: make it a member  $\Rightarrow$   
keep things object oriented and neat
- E.g., for matrix multiplication:  $C_{l \times n} = A_{l \times m} \times B_{m \times n}$   
matrix A(3, 2), B(2, 7);  
// ... later:  
matrix C(A.rows(), B.cols());  
C = A \* B; // alt.: matrix C = A \* B;
- Neatness: object *calls* instead of being an argument
- Compatibility: won't conflict with `image::rows()` or `::rows()`

But, in two situations this fails . . . .

# Back to `complex::operator+`

---

- We currently have

```
inline complex complex::operator+(const complex& rhs)
 const {
 return complex(real() + rhs.real(),
 imag() + rhs.imag());
}
```

// ... called by:

```
c6 = c1 + c8; // fine and dandy
```

- What if we want mixed-type addition?

```
c6 = c1 + 19; // still okay, or
```

```
c6 = 19 + c1; // error!
```

Why isn't addition commutative?

# Plight of operator+()'s LHS Object

---

- To understand, look at explicit function call

```
// implicit (int -> complex) conversion
c6 = c1.operator+(19);
// no int class, so no member function
c6 = 19.operator+(c1);
// alt., but no global function which takes complex
c6 = operator+(19, c1);
```

- Implicit conversion possible via (default) constructor

```
complex((float) 19) // int -> float -> complex
```

- money class (int dollars; float cents) ⇒

def. ctor is wrong (why?), add ctor: money(float)

- *Rule*: no implicit conversions on invoking object

Solution: define global function for complex addition.

# Global complex Addition

---

- Global form for addition (*replaces* member `op+()` – why?)

```
complex operator+(const complex& lhs, const complex& rhs) {
 return complex(lhs.real() + rhs.real(),
 lhs.imag() + rhs.imag());
}
```
- Note single argument of member `op+()`, and two here
- Now both arguments of `operator+()` can be converted
- Other member functions should be changed too (which ones?)
- Sometimes inhibit *object op object* (e.g., `time * time`)
  - \* but still allow both mixed-types (how?)
- Reminder: even though global, declare in `complex.h` and define in `complex.C` (unless `inline`)

# Friend Functions—Motivation

---

- What if  $\nexists$  `complex::real()` and `complex::imag()`?
- Alternate: use `complex::re` and `complex::im`
- Problem
  - \* data members are (properly) hidden in `protected`:
  - \* our function is now global  $\Rightarrow$  no access
- Solution: declare our global function to be a friend

How do I apply for friendship?

# Friend Functions

---

- Remain global functions
- Have access to members in `protected:` (and `private:`)
- Declared “friend” in class (e.g., in `complex`)  $\Rightarrow$   
usually all friends together at beginning
- “`friend class foo;`” within class `bar`’s definition
  - \* befriends `foo`’s member functions (but *not* `foo`’s friends) to `bar`
  - \* opposite is not true

And for a second reason for a non-member function, . . . .

# Proper Pretty-Printing

---

- Goal: print objects
  - \* to appear in a natural format (user-friendly)
  - \* in an easy-to-use fashion  
(application-programmer-friendly)
- Appearance: depends on object
  - \* `complex` (or coordinates): parenthesized, comma-separated list: `(-4.3, 94.3 i)`
  - \* `cstring`: just the pointer field
  - \* student info: itemization of fields
- Ease-of-use: can we get something like:  

```
cout << "This is c3: " << c3 << endl;
cout << " and s_b: " << s_b << endl; // no .get_s()
```

Let's try as a member function.

## `complex::operator<<()`—Wrong

---

- Here is the prototype (i.e., declaration)  
`ostream& complex::operator<<(ostream& output) const;`
- `ostream` reference returned for daisy-chaining (see later)
- Recall: as a member function, `complex` object invokes (calls) the function
- Therefore: function invocation  
`c3 << cout; // most unnatural!`
- Again we want object to be an argument (and *not* to invoke)
- Note: we cannot define (why?)  
`* ostream::operator<<(const complex& rhs)`

Therefore, make `operator<<()` global for `complex` objects.

# Global complex operator<<()

---

```
ostream& operator<<(ostream& output, const complex& rhs) {
 return output << "(" << rhs.real() <<
 ", " << rhs.imag() << " i)";
}
```

- ostream ref. for daisy-chaining; the following are equivalent:

```
cout << a << b; // standard form
((cout << a) << b); // left to right evaluation
(op<<(cout, a) << b); // replace infix ops
op<<(op<<(cout, a), b); // with prefix ops
```

- Again, if no `complex::real()` (and `imag()`)  $\Rightarrow$   
need to make this friendly
- Note: ostream argument is not const (why?)

# Other I/O Considerations

---

- Non-operator form
  - \* definition: `ostream& pretty_print( ...`
  - \* invocation: `cout << pretty_print(cout, a) << ...`
  - \* will print address of `ostream` (why? what changed?)
- What would `operator>>()` look like?
- Tip: code `operator<<()` before `operator>>()`
- Note: no `cout` in `operator>>()`
  - \* `istream` might be a file, not `cin`
  - \* can have additional function for interactive input
- How about both operators for `cstring` objects?

So, in summary ...

# Algorithm for Function Type Decision

---

- In general: keep functions as members if possible
- Reason for non-member function  $\Rightarrow$   
do not want object to invoke, rather be an argument
- Reason for global function to be `friendly`  $\Rightarrow$   
access to hidden data
- For defining function `func` for objects of class `cfoo`  
if (`func` needs type conversion on left-most argument) or  
    (`func` is `operator>>()` or `operator<<()`) {  
    make `func` global;  
    if (`func` needs access to non-public members of `cfoo`)  
        make `func` a friend;  
    }  
} else  
    make `func` a member;

And why is it so important to hide the data?

# Additional Class Features

---

- File Organization and Scope
- Friends and Object I/O
- ⇒ Hidden Data and Static Members
- Construction and Initialization
- References Between Functions
- Member Access Functions

# Reasons for Data Out of `public`:

---

- Bottom line
  - \* application-programmer-access only through
  - \* library-programmer-supplied member functions
- Advantages w.r.t. application programmer
  - \* Simplicity: need only learn to use member functions
    - ★ can ignore data implementation details
  - \* Uniformity: *always* accesses members via functions
    - ★ `complex::real()`, not `complex::real`
  - \* Protection: can be disallowed access (read/write/both)
    - ★ no writing to `cstring::s` without proper allocation
  - \* Restriction: even with write access, can be limited
    - ★ modifying `unit_complex` data members

... and for redesigning ...

# Reasons for Data Out of `public`: (cont.)

---

- Forward compatibility: future class library changes localized to member functions  $\Rightarrow$ 
  - will not need to change application code
  - \* errors (Well, not in *your* code, of course.)
  - \* data member name changes (`complex::re`  $\rightarrow$  `complex::_re`)
  - \* Algorithm changes: e.g., `length` vs. `length()` for `cstring`
    - \* save on computation (e.g., `op==()`), vs.
    - \* save on storage
- Advanced: define class-specific `enums` in class
  - \* in `protected`, if possible
  - \* but *before* being used

This is the reason for the extra “layer”.

# Static Data Members—Motivation

---

- How about a global variable for a class (*not* an array of objects), e.g.:
  - \* `_num_numbers` for `complex` (probably in/decremented in `con/destructors`)
  - \* `max_length` for `cstring` (probably set once at beginning of `main()`)
  - \* `average_life_span` for `person`
  - \* common lookup table (tournament for class `game`)
  - \* or: a running average, object state (ordered? normalized?), which algorithm to use, etc.
- How does one associate it with the class?
- And making sure to have only one copy? Otherwise
  - \* wasteful, error-prone

Answer: static data members.

# Static Data Members

---

- New (third) type of static
- $\left. \begin{array}{l} \text{Non-static} \\ \text{Static} \end{array} \right\}$  members belong to  $\left\{ \begin{array}{l} \text{objects} \\ \text{class} \end{array} \right.$
- *Declaration*: in class definition (in `complex.h`)  
`static int _num_numbers; // for complex class`
- *Definition*: needed elsewhere (in `complex.C`, *not* in `complex.h`—why?)  
`// note superfluous initialization (why?)`  
`int complex::_num_numbers = 0;`
- Is just a global variable
- Has protection like any other data member  $\Rightarrow$   
keep it out of `public`:

# Static Member Functions

---

- For accessing class static information *only*
  - \* also good idea for class help/info to user
- In class definition only: add `static` to beginning of function
- Invocation

```
cout << "max: " << s_b.max_length(); // okay, but ...
int objects = complex::num_numbers(); // more sensible
```
- Not associated with any specific object
  - \* if object constructed just to invoke function ⇒  
make function `static`
- Ramifications
  - \* `this` is not defined
  - \* not allowed to access non-static info
  - \* cannot be `const` (why not?)
    - ★ note: some things remain unprotected (what?)

# Additional Class Features

---

- File Organization and Scope
- Friends and Object I/O
- Hidden Data and Static Members
- ⇒ Construction and Initialization
- References Between Functions
- Member Access Functions

# Construction: Member Initialization

---

```
class colored_triangle {
 // ...
 colored_triangle(const coord& in_c1,
 const coord& in_c2, const coord& in_c3,
 int in_color) : c1(in_c1), c2(in_c2),
 c3(in_c3), color(in_color) {}
 // ...
 coord c1, c2, c3;
 int color;
};
```

- Construction proceeds in two phases
  - \* data member initialization *in declaration order*
    - ★ from member initialization list ∴ retain order, or
    - ★ default construction of data member
  - \* constructor function body

# Member Initialization—Features

---

- Initialization list format
  - \* similar to: `int i(7);`
  - \* variable must be data member (later re: inheritance)
  - \* can be complicated expression in parentheses
  - \* in definition only, not declaration
- Savings
  - \* if done as above: one copy construction for `c1`, vs.
  - \* `c1 = in_c1;` in function body causes
    - ★ data member initialization: default construction
    - ★ assignment: `coord::operator=()`
  - \* a good habit, but only an issue for non-built-in types
- Advanced: `const` and reference data members
  - \* example of `const`: student's birthday
  - \* un-assignable  $\Rightarrow$  initialization is *required*

# Object Array Initialization

---

```
int main(void) {
 complex a, b[20], c[] = {
 4.3f, // def. constructor
 complex(6.f, -22.0f), // def. constructor
 a, // copy constructor
 b[13] // copy constructor
 };

 // ...
}
```

- Use braces as usual for array initialization
- Individual element initialization (any constructor)
  - \* single argument: as is
  - \* multiple arguments: use constructor form

# Con/destructors and the Free Store

---

- Often con/destructors call `new()/delete()` (e.g., for `cstring`)
- Opposite is true as well: often `new()/delete()` call con/destructors
- In general, symmetrically we have
  - \* `new()` allocates (Stage 1) and constructs (Stage 2)
  - \* `delete()` destructs (Stage 4) and deallocates (Stage 5)
- Additional benefits over C's `malloc()` and `free()`

# Additional Class Features

---

- File Organization and Scope
- Friends and Object I/O
- Hidden Data and Static Members
- Construction and Initialization
- ⇒ References Between Functions
- Member Access Functions

# Passing Objects by Reference

---

- Passing an object by value, to or from a function, invokes copy constructor
- *And*, a constructor call will (eventually) be followed by a destructor invocation

```
class coord { ... float x, y, z;};
class triangle { ... coord v1, v2, v3;};
```

```
triangle return_triangle(triangle t) {return t;}
```

- Eight (copy) ctors and eight dtors called (where?)
- Solution: pass by reference (0 additional invocations)

```
triangle& return_triangle(const triangle& t) {return t;}
```

Passing by value can be very wasteful. But, . . . .

# Returning by Reference—Meaning

---

```
complex operator+(const complex& lhs, const complex& rhs);
```

- Returning by value method
  - \* copy constructor invoked to duplicate to stack
- Why does `operator+()` return by value?
- Recall: a reference is a name for another *existing* object
- Reason for return by value: it cannot return by reference
- Nowhere does the sum exist in an object
- Possible solution: create such an object and return its reference
- Wrong! Constructors not avoided, and . . .

# Forbidden Reference Returns

---

```
complex& operator+(/* ... lhs, ... rhs */); // WRONG!
```

- Two possible creation methods
  - \* local object
  - \* free store allocated object
- Downfalls
  - \* local object: leaves scope too early
  - \* free store: who will perform `delete()`s of `a + b + c`?
- Rule: do not return reference to (or address of)
  - \* local object
  - \* object allocated in function from free store (exception, e.g.: `free_matrix()` will perform `delete()`)

When new object is needed, return it by value.

# Redundant Object Specification

---

```
triangle francine;
```

```
francine.move(4, 8);
```

```
francine.enlarge(3);
```

```
francine.show();
```

```
francine.rotate(45);
```

```
francine.show();
```

- Redundancy unsightly for application programmer
- Occurs in non-operator, member functions
- Cause: above functions return void

How can we contract this code?

# Returning Reference to `*this`

---

```
francine.move(4, 8).enlarge(3).show();
```

```
francine.rotate(45).show();
```

- Desired interface: daisy-chaining
- Required: `francine.move(4, 8)` to “leave” `francine` on the line when done
  - \* enable invocation of `.enlarge(3)`
- Solution: as with `operator=()`, have functions return
  - \* `*this` (instead of `void`)
  - \* by reference (to enable subsequent changes)
- Can do this for all modify functions as well!

# Additional Class Features

---

- File Organization and Scope
  - Friends and Object I/O
  - Hidden Data and Static Members
  - Construction and Initialization
  - References Between Functions
- ⇒ Member Access Functions

# Constant Member Functions

---

- Pointers: a `const long *`
  - \* cannot change what it points to, therefore
  - \* is only kind of pointer a `const long` can be assigned to
- This allows compiler to help enforce a constraint
- Similarly a constant member function
  - \* cannot change data of the invoking object, therefore
  - \* is only member function a `const` object can invoke  $\Rightarrow$   
not just a good idea—it's the LAW!, e.g.

```
float complex::real(void) const {return re;}
```

```
complex operator+(const complex& lhs, const complex& rhs) {
 return complex(lhs.real() + rhs.real(), /* ... */);}
```

- Need “`const`” in declaration and definition

Note the many different forms of `const`.

# Current View and Modify Functions

---

```
class coord {
public:
 // ...
 double x(void) const {return _x;} // returns a _copy_
 void x(const double& in_x) {_x = in_x; return;}
 // ... same for y ...
protected:
 double _x, _y;
};
```

// usage ...

```
coord a, b;
b.x(a.y());
```

- View pro: `a._y` *cannot* be changed

- View con: large data member  $\Rightarrow$  wasteful, slow, and ...

# Class Layering

---

```
class triangle {
public:
 triangle(void) : _v1(1., 1.), _v2(2., 2.), _v3(3., 3.) {}
 // ...
 coord v1(void) const {return _v1;}
 void v1(const coord& in_v1) {_v1 = in_v1; return;}
 // ... same for v2 and v3 ...
protected:
 coord _v1, _v2, _v3;
};
```

- Consistent, object-oriented design
- Layering of classes  $\Rightarrow$  daisy-chaining of access functions

# Data Member Daisy-Chaining

---

```
// ... usage ...
```

```
triangle t;
cout << t.v3().x() << " ";
t.v3().x(7.);
cout << t.v3().x();
```

- Output: 3 3 (!!)
- Problem: `.v3()` leaves a *copy* of `t._v3`
  - \* this temporary copy has its `_x` set to 7.
- Note: some regard these as confusing run-on sentences

# Viewing a const Reference

---

```
// ...
const double& x(void) const {return _x;}
```

```
// ...
const coord& v1(void) const {return _v1;}
```

```
// ... usage: same
```

- Pro: not wasteful, quick
- Pro: still pretty safe—2 consts for each (what for?)
- Con: *could* be typecast to remove constness, e.g.:  
    double &xr = (double &) t.v3().x(); xr = -3.14;
- Con: data member daisy-chaining no longer compiles (why?)

Does not handle all cases of object access.

# View/Modify a Reference

---

```
const double& x(void) const {return _x;}
double& x(void) {return _x;} // not const, to disambiguate
// ... same for y, v1, v2, ...
// ... usage: same, plus

// invocation based on object const-ness (which calls which?)
b.x() = a.y();
```

- Can overload member functions based on constness alone
  - \* note: this will not work for class static info
- Can rid of original modify-only form
  - \* *but*: modification correctness check is inhibited
- Pro: daisy-chaining okay now
- Con: “()” on lhs—somewhat unconventional
  - \* returning non-const more for op[]() (see iarray later)

# Array Redesign

---

- ⇒ Motivation
  - Constructor Declarations
  - M/G Function Declarations
  - Implementation: Data Members
  - M/G/F Function Definitions
  - Class Templates

# Redesign Issues of Arrays

---

- The size must be a constant
  - \* not decided by user even once
  - \* certainly not changed later
  - \* cannot free up once not needed
- The array does not know its own size
  - \* we carry around additional variable with size
- Cannot assign: `array1 = array2;` // error
- Other possible ops/functions: `array1 * array2`, printing, ...
- No range checking:

```
cout << array1[-3]; // compiles! Oy!
```

Want what arrays can do, plus more ....

# Class Design—Approach

---

- Functions declarations first, definitions later
- Application and library programmers (in whose office?)
  - \* Start with public interface—prototypes only
    - \* first constructor forms (default, copy, other?)
    - \* other related functions (member, global?)
      - \* start with views, modifies if appropriate
      - \* assignment, operators, etc.
- Library programmer only
  - \* Determine internal representation, i.e., data members
  - \* Define the functions
    - \* which should be `inline`?
    - \* members: which should be `const`? `static`?
    - \* globals: which must be `friends`?

Let's proceed, warp factor 2.

# Array Redesign

---

- Motivation
- ⇒ Constructor Declarations
- M/G Function Declarations
- Implementation: Data Members
- M/G/F Function Definitions
- Class Templates

# iarray Constructor Purposes

---

- Usage by application programmer: to specify length
  - \* len: not const, not known at compile time

```
cin >> len; iarray ia1(len); // mtg: appli. prog. writes
```
- Constructor given length:

```
iarray(int input_size); // mtg: lib. prog. writes, etc.
```

or combining with default constructor:

```
const int def_iarray_size = 7; // outside class def
// ... in public: section
iarray(int default_size = def_iarray_size);
```
- Allocate additional memory for the array *safely*
- Zero out (or initialize to default value) the iarray

How about constructors which *really* initialize?

# Two More Constructors

---

- *Copy* construction: usage

```
iarray ia2(ia1), ia3 = ia2;
```

declaration:

```
iarray(const iarray&);
```

- Prototype to initialize with “normal” array

```
int heights[] = {-4, 5, /* ... */, 84};
```

```
iarray ia4(heights, sizeof(heights) / sizeof(heights[0]));
```

declaration:

```
iarray(const int *, int); // more overloading
```

What do we have so far?

# iarray Constructors

---

- Could have ctor to read from file (ifstream or const char \*)

```
const int def_iarray_size = 7;
```

```
class iarray {
public: // note: this section first, as with design
 iarray(int default_size = def_iarray_size);
 iarray(const iarray&);
 iarray(const int *, int);
 // ...
protected:
 // ...
};
```

On to other function declarations, ...

# Array Redesign

---

- Motivation
- Constructor Declarations
- ⇒ M/G Function Declarations
- Implementation: Data Members
- M/G/F Function Definitions
- Class Templates

# Assignment and Multiplication Declarations

---

- Assignment invocation and prototype

```
ia1 = ia4; // ia1 calls function with argument of ia4
ia1.operator=(ia4); // equivalent (but ugly) form
// ... declared by (ret. non-const to allow: (i = j) ++):
iarray& operator=(const iarray& rhs);
```

- iarray element-wise multiplication (not global for us ... b|c?)

```
ia2 = ia1 * ia3; // who invokes? who is the argument?
// ... declared by (return by value!):
iarray operator*(const iarray& rhs) const;
```

- According to our goals, still need access functions for
  - \* accessing a single element, just as any array can
  - \* viewing the size of the iarray

Last two declarations . . . .

# iarray Indexing and Size

---

```
ia4[2] = ia3[5]; // ia4.operator[](2) = ia3.operator[](5);
// calls the following twice:
```

```
int& operator[](int) const; // why is "const" okay?
```

- Function name: `operator[]`
- Return value: a *reference* to an `int` (why? look at the lhs)
- More like a modify: separate to:

```
 const int& operator[](int) const; // view
```

```
 int& operator[](int); // modify
```

```
cout << "size of ia2: " << ia2.size() << endl;
```

```
// ... declared by:
```

```
int size(void) const;
```

- Also a member function, declared in class definition
- View allows an `iarray` object to know its own size

Current tally ...

# iarray Function Declarations

---

```
const int def_iarray_size = 7;
```

```
class iarray {
public:
 // constructors
 iarray& operator=(const iarray& rhs);
 iarray operator*(const iarray& rhs) const;
 const int& operator[](int) const;
 int& operator[](int);
 int size(void) const;
protected:
 // ...
};
```

Okay, enough theory, where's the beef?

# Array Redesign

---

- Motivation
- Constructor Declarations
- M/G Function Declarations
- ⇒ Implementation: Data Members
- M/G/F Function Definitions
- Class Templates

# Information to be Stored

---

- What should internal representation look like?

protected:

```
int *_info, _size;
```

- protected  $\Rightarrow$  access only allowed to
  - \* member functions
  - \* friend functions
- What have we paid? Eight extra bytes
  - \* this is what `sizeof()` reports, even with 1000 internal elements
- What have we gained?
  - \* extra four bytes, can be assigned and reassigned  $\Rightarrow$  flexibility of pointers over arrays
  - \* another four bytes  $\Rightarrow$  retain size information

# Array Redesign

---

- Motivation
- Constructor Declarations
- M/G Function Declarations
- Implementation: Data Members
- ⇒ M/G/F Function Definitions
- Class Templates

# A Helping Function for Constructors

---

- Motivation: all 3 constructors do similar things
- For assistance in defining other constructors (and assignment)
- `∴` declared in `protected`: section of `iarray` definition

```
void init(const int *, int);
```
- Arguments
  - \* optionally take an `int` array for initialization
  - \* target array size
- Responsibilities: initialize (two) fields, including
  - \* free store allocate memory for new array
  - \* check for success
  - \* optionally initialize

The envelope please . . .

## Definition of `iarray::init()`

---

```
#include <string> // for memcpy() and memset()
#include <assert> // consider cerr-notifying and
 // retrying with less, ...
using namespace std;
inline void iarray::init(const int *array, int sz) {
 _info = 0; _info = new int[_size = sz]; // whose _info?
 assert(_info); // quit if new() failed

 if (array) // did we receive an initialization array?
 memcpy((void *) _info, (const void *) array,
 _size * sizeof(int)); // or: for-loop
 else
 memset((void *) _info, 0, _size * sizeof(int));
 return;
}
```

And the constructors ... (any guesses?)

# iarray Constructors—Definitions

---

```
iarray::iarray(int sz = def_iarray_size) {
 init((const int *) 0, sz); // what if def sz == 0?
}
iarray::iarray(const iarray& rhs) {
 init(rhs._info, rhs._size);
}
iarray::iarray(const int *array, int sz) {
 init(array, sz);
}
```

- Calling-object's `init()` invoked, `∴` no “.”
- Recall: in copy constructor: `rhs._info` and `rhs._size`
  - \* since member function, access even to different object's protected members
  - \* but `rhs.` necessary to specify other object

# Inline Member Functions

---

- Let's make constructors `inline` for speed
- In `public:` section of class definition:

```
iarray(int sz = def_iarray_size) {
 init((const int *) 0, sz);
}
```
- Recall: only an *inline suggestion* to compiler
- Note: `init()` can be `inline` as well

Still need the “cleanup” function, ...

# iarray Destructor

---

- Recall: constructors called just after allocation of data members, i.e., `_info` and `_size`
- Symmetrically, destructors called just *before* deallocation of data members
- Primarily to deallocate free store memory; not called explicitly
- Aside from these (how many?) bytes, what else needs be done? Freeing up memory of free store allocated array  

```
~iarray(void) {delete [] _info;} // notice []
```
- Note: also made inline for speed

And for the other member functions, ...

## iarray operator=() Definition

---

```
// ref returned for daisy-chaining
iarray& iarray::operator=(const iarray& rhs) {
 if (this == &rhs) return *this; // self-test
 delete [] _info; // free up current memory
 init(rhs._info, rhs._size); // copy rhs to lhs
 return *this;
}
```

- Recall: this contains the address of calling object, e.g.:  
    ia4 = ia1; // in iarray::operator=(), this == &ia4

- If self-test is true, return early (as with cstring)
  - \* save time (nothing to do)
  - \* avoid catastrophic self-assignment

```
 iarray& ia5 = ia3, *pia = &ia2;
 // later ...
 ia3 = ia5; // or viceversa
 *pia = ia2; // ditto
```

## iarray::operator\*() Definition

---

```
iarray operator*(const iarray& rhs) const {
 int min_size = (size() < rhs.size()) ?
 size() : rhs.size();
 iarray result(min_size);

 for (int index = 0; index < min_size; index++)
 result[index] = (*this)[index] * rhs[index];
 return result;
}
```

- `result` returns by value, as information is new
- Note: other definitions possible for mis-matched lengths
- Note use of `this`—how else could this be coded?

# operator[] () and size() Definitions

---

```
const int& operator[](int index) const { // view, and
 return _info[index];
} // modify:
int& operator[](int index) {return _info[index];}
int size(void) const { // view
 return _size;
}
```

- Recall: operator[] () return a reference for speed and to allow, e.g., ia3[4] --;
- Inlining:
  - \* operator=()—probably okay, worth a try
  - \* operator\*(), probably
  - \* operator[] () and size()—yes, definitely

What might prog.C look like?

## main() Using iarray Class

---

```
#include "iarray.h" // locate:
 // * memory allocation (and how much)
int main(void) { // * function calls
 int grades[] =
 {47, 94, 68, 32, 87, 82}; // probably read in
 const int num_students = sizeof(grades) / sizeof(grades[0]);
 iarray all_grades(grades, num_students),
 passing_grades = all_grades;
 void remove_failures(iarray&);

 remove_failures(passing_grades);
 return 0;
}
```

## remove\_failures() Using iarray Class

---

```
void remove_failures(iarray& grades) {
 const int passing_grade = 60;
 int num_passing = 0, student;

 for (student = 0; student < grades.size(); student ++)
 if (grades[student] >= passing_grade)
 num_passing ++;

 iarray temp_grades(num_passing);
 num_passing = 0;
 for (student = 0; student < grades.size(); student ++)
 if (grades[student] >= passing_grade)
 temp_grades[num_passing ++] = grades[student];

 grades = temp_grades;
 return;
}
```

# Array Redesign

---

- Motivation
  - Constructor Declarations
  - M/G Function Declarations
  - Implementation: Data Members
  - M/G/F Function Definitions
- ⇒ Class Templates

# Class Templates—Motivation

---

- Fine for ints, but what about floats, shorts, chars, etc.?
- Copying everything to farray, sarray, carray, etc.
  - \* wasteful
  - \* error-prone
- Use class templates, as with function templates
- Can instantiate various types of class array on the fly:

```
array<int> ia6(16);
array<char> ca1; // what's wrong with type char?
```
- Different classes ∴ different static members
- `remove_failures()`: 1) `array<int>&` argument or 2) templated
- Class template definition placed in `array.h`

What does it look like?

# Class Templates—Definition

---

```
const int def_array_size = 7;

template <class Type>
class array { // in class def., "array" is
public: // short for "array<Type>"
 array(int sz = def_array_size) :
 _info(new Type[sz]), _size(sz) {}
 ~array(void) {delete [] _info;}

 int size(void) const {return _size;}
 const Type& operator[](int index) const // view
 {return _info[index];}

protected:
 Type *_info;
 int _size;
};
```

- Egcs g++: friends require <> after function name

Note: initialization order, missing assert(), ...

# STL

---

- Standard Template Library (“STL”) is major part of C++ R&D
- Now being standardized (what to include, interface, etc.)
  - \* somewhat antiquates pointers and arrays
- Open issues for templates in general:
  - \* are even non-`inline` functions in the header files?
    - ★ adds to compilation time
  - \* how does one avoid multiple definitions?
  - \* what about static data members of class templates
    - ★ where do the definitions go?
    - ★ note: not implemented by all compilers

Templates are very powerful, and are the wave of the future.

# Inheritance

---

- ⇒ Motivation
  - Basic Elements
  - What Needs to be New
  - Polymorphism
  - Subclass Templates

# What has `iarray` bought us?

---

- The size of the `iarray` need not be constant — ✓
- The `iarray` knows its own size — ✓
- One can: `ia4 = ia6;` — ✓
- Also possible: `ia5 * ia3` — ✓
- Range checking — ×  
`cout << ia2[-3]; // still compiles! oy vey!`

What are our options?

# Options for `iarray` Range Checking

---

- Add this capability to `iarray::operator[]()`

- \* problem: this may be slow, not always desired, e.g.:

```
for (int i = 0; i < ia3.size(); i ++)
 ia3[i] = 0; // nothing unsafe here
```

vs.

```
cin >> iarray_index; // need to check user input
cout << ia1[iarray_index];
```

- Define a totally new class `safeiarray`
  - \* problem: this will repeat lots of code, error-prone
- Create a subclass, ...

Enter: inheritance.

# Public Inheritance—The Basic Idea

---

- Goal: a new, *derived* class whose objects are both
  - \* objects of the *base* class, and also
  - \* specific, specialized objects of the derived class
- Rule: a derived object *isA* base object, but *not* vice-versa
- Also: justify separate classes
  - \* perhaps base class objects should be of derived type  $\Rightarrow$   
only one class needed
- Can think of sets and Venn diagrams (e.g.,  $R \subset \mathcal{C}$ )

Always ask the “*isA*” question.

# Public Inheritance—Permissions

---

- Derived class inherits (almost) all members of base class
  - \* note: friendship is *not* inherited
- (Not true for `private`: members—not for us)
- Permissions: anything one can do to/with a base object, can be done with a derived object
- `static` members (data and functions)
  - \* base class members are inherited
  - \* *no new* members for derived class
  - \* derived objects behaves like base objects (*isA* lives!)
- Note: base class (`base.[h|C]`) basically
  - \* does not change
  - \* does not know about derived class
    - \* *never* include `derived.h` (design bug)
  - \* should compile separately, runnable with driver (do it!)

# airplanes and bombers

---

```
class airplane { /* ... */ };
class bomber : public airplane { /* ... */ };

void fly(const airplane& a); // Qs: 1) members?
void attack(const bomber& b); // 2) do you agree?

airplane a;
bomber b;

fly(a); // "a" is an airplane
fly(b); // "b" is a bomber, and "b" isA airplane
attack(b); // "b" is a bomber
attack(a); // error! "a" is not a bomber (necessarily)
```

C++ permissions follow *isA* sensibility.

# The triangle Base Class

---

- Possible derived classes
  - \* particular characteristic: `right_triangle`
  - \* additional field: `colored_triangle`

⇒ they pass the *isA* test
- Not possible derived classes
  - \* particular characteristic: `trianglar_pipe` (*isA* pipe, but not a triangle)
  - \* additional field: `raised_triangle` (is not 2-D, but a triangle is)

⇒ they fail the *isA* test
- Note: `trianglar_pipe` and `raised_triangle` will probably *include* triangle data members

# Inheritance

---

- Motivation
- ⇒ Basic Elements
- What Needs to be New
- Polymorphism
- Subclass Templates

# The safeiarray Subclass

---

```
#include "iarray.h"
```

```
class safeiarray : public iarray { // built on top of iarray
public:
 // new constructors needed at least
protected:
 // new protected stuff
};
```

- Placed in safeiarray.h
- Important: a safeiarray object isA iarray object, conceptually and re: permissions—but a special case
- safeiarray *inherits* everything from iarray (except constructors)

So what's so special about an safeiarray object?

# Inheritance

---

- Motivation
- Basic Elements
- ⇒ What Needs to be New
- Polymorphism
- Subclass Templates

# New safeiarray Functions

---

```
public: // which constructor is missing?
 safeiarray(int sz = def_iarray_size) :
 iarray(sz) {} // base name, not base members
 safeiarray(const int *array, int sz) :
 iarray(array, sz) {}
 const int& operator[](int) const;
 int& operator[](int);
```

- base construction first, with constructor from
  - \* initialization list if present, otherwise: default
- Note: nothing else need be constructed
- operator[]()
  - \* we like and keep declaration/interface
  - \* must match return value as well, but
  - \* definitions needs reworking

How do we range-check?

## safeiarray::operator[] () Definition

---

```
#include <assert> // or consider something "nicer"
#include "safeiarray.h"
using namespace std;

const int& safeiarray::operator[](int index) const {
 assert(index >= 0 && index < _size);
 return _info[index];
} // same for modify op[] () ...
```

- Placed in safeiarray.C (or .h if inline)
- Recall: redeclared member functions hide *all* base forms with same overloaded name
- Recall: a safeiarray object can be sent to `remove_failures()`, hmmm

How will `operator[] ()` behave on a `safeiarray` object there?

# Inheritance

---

- Motivation
- Basic Elements
- What Needs to be New
- ⇒ Polymorphism
- Subclass Templates

# Making Member Functions Virtual

---

- Problem = “slicing” of subclass features: within `remove_failures()`, `iarray::operator[]()` will be called
- Solution: `virtual iarray::operator[]()` (by base only)
  - \* subclass function need not be virtual (why?)
- Method: all `iarray::operator[]()` calls are dynamically bound
  - \* check what object really is = *polymorphism*
  - \* but note: specific object always calls one or the other
- `remove_failures()` correctly received grades by reference
  - \* another reason for not passing by value (which *copies*)
- Note: `remove_failures()` does not change, per se, just calls to `iarray::operator[]()`
- Trade-off: `iarray::op[]()` can no longer be `inline` (why?)

# Working With Virtuality

---

- Needed *types*: pointers and references to base class objects

```
void foo(base *okay1, base& okay2, base not_okay);
```

  - \* even if pointing/referring to subclass object
  - \* object themselves behave monomorphically
- Correct object type: via virtual function invocation
  - \* Rule: in general, avoid base/derived class type checking
- Forcing the issue: explicit casting

```
hw *w = new nurse; // okay due to isA
cout << *w; // slicing! only hw info printed
nurse n1 = *w, // 2 casts (where?), n1 only gets hw info
n2 = *((nurse *) w); // somewhat ugly, but okay
```
- Advanced: RTTI (RunTime Type Identification)
  - \* dynamic casts—when one *needs* the type information

# Object Identity Crisis

---

- Case 1
  - \* desired: object of either class or subclass type (perhaps user defined)
  - \* snag: type not known at compile time, which to use?
- Case 2
  - \* desired: array of base class *and* subclass objects intermingled
  - \* snag: arrays can be of only one type
- Solutions
  - \* Case 1: pointer to *base* class object (why base?)
  - \* Case 2: array of pointers to *base* class objects

# Object Virtuality

---

```
hw *hwp, *ahwp[42];
```

- Pointers  $\Rightarrow$  free store allocation (probably in a switch)

```
hwp = new hw;
```

```
// or:
```

```
ahwp[19] = new doctor;
```

- Result: subsequent calls to `virtual` functions act appropriately

```
hwp -> send_holiday_bonus();
```

```
(ahwp[25]) -> send_holiday_bonus();
```

- Pointer type

- \* remains (`hw *`), but `virtual` functions check dereferenced object type

Gives more pause as to designing virtual functions.

# Virtual Functions—Design Features

---

- Subclass function needs identical prototype to base virtual function (including return value)
- Adding fields to subclass: likely `op=` and `op==` will change  $\Rightarrow$  but not `virtual` due to different arguments
- When possible, build on base virtual function, e.g., invoke:

```
hw::print(); // from within nurse::print(),
 // doctor::print(), admin::print(), etc.
 // why is hw:: required?
```
- Make `dtor` `virtual` in base class (even though different name)
  - \* derived `dtor` *always* calls base `dtor` afterwards (symmetric to `ctors`)
  - \* therefore correctly *start* with subclass `dtor` (then base's)

```
hw *hwp = new admin; /* ... */ delete hwp;
```

# Virtual Functions—Design Features (cont.)

---

- Internal objects table is smart about types
  - \* when storing in “dumb” file, tag the type
- Need for `virtual`: another reason for member vs. global function
- To make a global function (e.g., `op<<()`) act virtual
  - \* have it invoke a virtual member function (e.g., `print()`)
- Some subclasses may inherit definition; others redefine
- Note: retracting from earlier base class individuality claim
  - \* base class acknowledges possible derived classes

Virtual functions supply subclasses with default definitions.

# Pure Virtual Functions

---

- What if  $\nexists$  default definition?

```
class pet {
public:
 virtual void feed(void) = 0; // pure virtual
 // ...
};
```

```
class dog : public pet { /* ... */ };
class snake : public pet { /* ... */ };
```

- pet is an *abstract base class* (ABC,  $\geq 1$  pure virtual function[s])
- subclasses (dog and snake) must
  - \* define pure virtual functions, or
  - \* inherit them, and become an ABC

# ABC — Purposes

---

- Cannot create objects of class `pet`
  - \* but pointers and references to `pet` are okay
    - ★ for use, e.g., in function arguments
    - ★ will *only* point/refer to subclass object
- Why declare pure virtual at all in base?
  - force interface inheritance, allowing:

```
for (int this_pet = 0; this_pet < num_pets; this_pet ++)
 pet_array[this_pet] -> feed();
```
- Treat ABC like any other base class
  - \* place all common members there
  - \* even define constructors and destructors (*why?*)
- Note: retracting further from base class individuality claim
  - \* ABC created only as building block for derived classes
  - \* ABC knows of their existence—but not *what* they are

# Member Function Virtuality

---

- Member function declarations—inherited
- Member function definitions—inheritance
  - \* non-virtual: ✓
    - ★ do *not* redefine (if redefined, will slice)
  - \* non-pure virtual: ✗
    - ★ by default, if not redefined
  - \* pure virtual: ×

Function virtuality is important C++ feature.

# Inheritance

---

- Motivation
  - Basic Elements
  - What Needs to be New
  - Polymorphism
- ⇒ Subclass Templates

# Subclass Templates

```
#include <assert> // again, perhaps be user-friendlier
#include "array.h"

template <class T>
class safearray : public array<T> {
public:
 safearray(int sz = def_array_size) : array<T>(sz) {}
 const T& operator[](int index) const { // view
 assert(index >= 0 && index < _size);
 return _info[index];
 } // same for modify op[]() ...
};
```

- Instantiation: `safearray<short> ssa(14);` causes base and derived class generation
- g++ option `-fpermissive` may be necessary
- Q: How will `remove_failures()` look? templated? Its argument?

# Linked Lists

---

- ⇒ Motivation and Design
  - Constructor Declarations
  - M/G Function Declarations
  - Implementation: Data Members
  - M/G/F Function Definitions
  - Sample Driver Program
  - Unique List Derivation

# Linked Lists—Motivation

---

- Aggregation, for grouping objects

| <i>issues</i>              | <i>implementations</i> |
|----------------------------|------------------------|
| random access              | array                  |
| variable length            | iarray                 |
| add/deleting in the middle | linked list            |
| direction bias             | doubly linked list     |
| pseudo-uniform access      | additional links       |
| special structure/search   | binary trees           |
| others                     | others                 |

- Tradeoffs: flexibility, vs. storage, speed
- For heterogeneous (base/derived) lists
  - \* allow for linked list of base pointers
  - \* two paradigms
    - ★ relate to dereferenced object (for free store, comparisons, printing, etc.)—need to record this
    - ★ or not—this is the case here (arguably inferior)

# Class Implementation—Motivation

---

- Exploit data hiding and `const` member functions
- Functions can be complex  $\Rightarrow$   
supply them to application programmer
- Squirrel away free store interactions
- Exploit operator overloading for ease-of-use
- Note: model will not contain a “current” node
- Build for general type of list nodes  $\Rightarrow$  use templates
  - \* for us: *all* template functions in `.h`, automatically `static`
- A basic STL container class

# Linked Lists

---

- Motivation and Design
- ⇒ Constructor Declarations
- M/G Function Declarations
- Implementation: Data Members
- M/G/F Function Definitions
- Sample Driver Program
- Unique List Derivation

# list Default Constructor

---

- Usage by application programmer:

```
list<short> list1; // mtg: appli. prog. writes
with declaration:
```

```
// ... in public: section
```

```
list(void); // mtg: lib. prog. writes, etc.
```

- Purpose: start out a fresh, new, empty list
- Other possible lists
  - \* `list<char>`, `list<complex>`, ...
  - \* `list<complex *>`
  - \* `list< list<unsigned> >` (ooh!) ... (note needed space)

How about constructors which initialize?

# Two More Constructors

---

- *Copy* construction

```
list<short> list2(list1), list3 = list2;
```

declared by (in class temp. def., list is short for list<T>):

```
list(const list& rhs); // overloading here and below
```

- Prototype to initialize with “normal” array

```
short sa[] = {-7, -16, /* ... */, 6};
```

```
list<short> list4(sa, sizeof(sa) / sizeof(sa[0]));
```

declared by:

```
list(const T *larray, int arr_length);
```

What do we have so far?

# list Constructors

---

- Could have ctor to read from file (ifstream or const char \*)

```
template <class T>
class list {
public: // note: this section first, as with design
 list(void);
 list(const list& rhs);
 list(const T *larray, int arr_length);
 // ...
protected:
 // ...
};
```

On to other function declarations, ...

# Linked Lists

---

- Motivation and Design
- Constructor Declarations
- ⇒ M/G Function Declarations
- Implementation: Data Members
- M/G/F Function Definitions
- Sample Driver Program
- Unique List Derivation

# list Assignment Declaration

---

```
list1 = list4; // list1 calls function with argument of list4
list1.operator=(list4); // equivalent form
// ... declared by:
list& operator=(const list& rhs);
```

Recall from complex:

- Function name: `operator=`
- Reference argument for speed and self-test
- Reference (i.e., lvalue) returned for speed and daisy-chaining:

```
list2 = list4 = list3;
// equiv.: list2.operator=(list4.operator=(list3));
// match return value and argument types
```

- Recall: vs. copy constructor, no new object just defined

# list Node Insertion

---

- For now, just prepending and appending
- We could use member functions `prepend()` and `append()`  $\Rightarrow$   
how about some operators?
- Appending

```
list3 = list1 + (short) 7;
list2 += short (-4); // saves time vs. previous (why?)
// declared by:
list operator+(const T& rhs) const;
void operator+=(const T& rhs);
```
- Prepending
  - \* want syntax of: `list3 = (short) 7 + list1;`
  - \* `list` object does *not* invoke
  - \* same situation as `operator<<`
  - \* therefore make global (and maybe: `friend`, `inline`)

```
template <class T>
list<T> operator+(const T& lhs, const list<T>& rhs) { ...
```

# list Node Removal and Querying

---

- Removal definition: all occurrences of specified value (other definitions?)

```
list1 = list4 - (short) 24;
```

```
list3 -= (short) -16;
```

```
// declared by:
```

```
list operator-(const T& rhs) const;
```

```
void operator-=(const T& rhs);
```

```
* note lack of: list1 = (short) 7 - list2;
```

```
* recall paradigm for list of pointers
```

```
 * not handling dereferenced free store
```

- Querying emptiness and presence (could return a count)

```
bool is_empty(void) const; // suggestions for an op()?
```

```
bool is_present(const T& lvalue) const;
```

```
* can add is_present_ptr() for searching dereferenced
objects
```

- Also possible: dealing with  $i^{\text{th}}$  node (which op()?)

# list Output and Number of lists

---

- Output: global (maybe friend)

```
// can drop first line (and <T>) from declaration if friend
template <class T>
ostream& operator<<(ostream& output, const list<T>& rhs) {
 // ...
}
```

- \* again, can add function for printing dereferenced objects

- Number of lists “alive”

```
static int num_lists(void);
```

- \* a “view” to access and return static info only
  - \* not associated with class object  $\Rightarrow$  no `this` or `const`
  - \* recall: not implemented for template classes by all compilers (e.g., old versions of `g++`)

Current tally ...

# list Function Declarations

---

```
template <class T>
class list { // doc deleted for space
public:
 // constructors ...
 list& operator=(const list& rhs);
 list operator+(const T& rhs) const;
 void operator+=(const T& rhs);
 list operator-(const T& rhs) const;
 void operator-=(const T& rhs);
 bool is_empty(void) const;
 bool is_present(const T& lvalue) const;
 static int num_lists(void);
protected:
 // ...
};
```

## list Function Declarations (cont.)

---

```
// some of these may have to be friends
```

```
template <class T>
```

```
list<T> operator+(const T& lhs, const list<T>& rhs) {
```

```
 // ...
```

```
}
```

```
template <class T>
```

```
ostream& operator<<(ostream& output, const list<T>& rhs) {
```

```
 // ...
```

```
}
```

- Interface done  $\Rightarrow$  lib. programmer to his/her office
  - \* what about application programmer?
  - \* note: appl. prog. can compile but not link (2 whys?)

Now for library's internal representation . . . .

# Linked Lists

---

- Motivation and Design
- Constructor Declarations
- M/G Function Declarations
- ⇒ Implementation: Data Members
- M/G/F Function Definitions
- Sample Driver Program
- Unique List Derivation

# Split Abstraction

---

- Break apart node abstraction from list concept
- Node
  - \* information (short, complex, ...)
  - \* link (pointer) to other node (or nothing if tail node)
- List: pointer to head of list (first node)
- Node features
  - \* *auxiliary* or *utility* class for list
  - \* hidden from application programmer
    - \* s/he know T and list<T>, but not node<T>
  - \* no public interface—even constructors
  - \* ∴ list is a friend of node
- Also retain a static number of lists

# Structure of Classes

---

- Include in `list.h`, before `list` template

```
template <class T> // forward declaration
class list;
```

```
template <class T>
class node {
friend class list<T>;
protected:
 T value;
 node *next;
};
```

- In `list`'s protected section

```
node<T> *head;
static int number_of_lists;
```

# Linked Lists

---

- Motivation and Design
- Constructor Declarations
- M/G Function Declarations
- Implementation: Data Members
- ⇒ M/G/F Function Definitions
- Sample Driver Program
- Unique List Derivation

# list Default Constructor

---

- Defines an empty list
  - Remember to initialize *all* data members
    - \* Includes adjusting `number_of_lists` accordingly
    - \* Added advantage of `(head == 0)` signifying empty list
- ```
// in list def., therefore inline
list(void) : head(0) {
    number_of_lists ++;
}
```
- Other constructors are for non-empty lists
 - First we need some building block functions

On to single node construction.

node Constructor

- In any list building, we will be adding nodes

- Need to create a new node \Rightarrow need node constructor

```
template <class T>    // note: could be placed in node def.
inline node<T>::node(const T& in_value,
                    const node<T> *in_next) :
    value(in_value), next((node<T> *) in_next) {
}
```

- Declared in node's protected: section
- Declare with 0 default value for pointer
- Function name does not get "<T>"
- First argument not sent by value
 - * wasteful for large nodes (lots of info)

Other node issues; Empty list Check

- Recall: member initialization—preferred over assignment
 - * determines which constructor to use for data (copy)
 - * otherwise, default constructor invoked
 - ★ necessitates assignment
 - ★ slower due to extra member function call
- Interestingly, no node destructor ⇒ deallocation handled by list
- To check for an empty list (inline and const)

```
template <class T>
inline bool list<T>::is_empty(void) const {
    return !head;
}
```

We start with self-appending.

Self-Appending Design

- `operator+=()` is a building block
- Two options for algorithm for appending
 - * start from `head` and go through list
 - * have a pointer to the `list`'s tail
- Note: implementation hidden from application programmer \Rightarrow will not change public interface
- Without tail pointer: $\text{time}_{\text{appending}} \gg \text{time}_{\text{prepending}} \Rightarrow$
 - * add to protected section: `node<T> *tail;`
 - * initialize `tail` to 0 in default constructor

And for the algorithm

Self-Appending Components

- Done "in place" \Rightarrow will be much quicker than `operator+()` which returns new list
- `inline`— \surd , `const`— \times
- Allocate from the free store
- Assert success of allocation
- Need to check for special case of empty list
- Update `tail`

And for the code,

Self-Appending operator+=()

```
template <class T>
inline void list<T>::operator+=(const T& rhs) {
// not const b/c assigning to "head" and "tail"
  const node<T> *pnode = new node<T>(rhs);
  assert(pnode);      // or friendlier notification,
                      //      retrying, ...

  if (is_empty())
    head = (node<T> *) pnode;
  else
    tail -> next = (node<T> *) pnode;

  tail = (node<T> *) pnode;

  return;
}
```

Note how many times this will be used.

list Copy Constructor

- Calls lower level member function `operator+=()` for copying from `list` to `list` (note overloading!)
- `operator+=()`
 - * used by a number of functions (thusly created)
 - * but only by member functions \Rightarrow protected:

```
template <class T>
inline list<T>::list(const list<T>& rhs) : head(0), tail(0) {
// op+=() calls op+=(const T&) which reads "head"
    (*this) += rhs;    // could use op=() instead
    number_of_lists ++;
}
```

And here is `operator+=()`.

Copying from list to list

```
template <class T>
inline void list<T>::operator+=(const list<T>& copy_from) {
    const node<T> *pnode = copy_from.head;

    while (pnode) {
        *this += pnode -> value;
        pnode = pnode -> next;
    }

    return;
}
```

- Two consts used for *reading* source info
- Note call to `operator+=(const T&)`
- Our first `while` loop, concealed from application programmer
- (Probably) can be `inline`, otherwise
 - * would be defined in `list.C`, if not template

Constructing From an Array; Destructor

- All the same except: while → for loop

```
list<T>::list(const T *larray, int arr_length) :
    head(0), tail(0) {    // op+=() checks "head"
    for (int index = 0; index < arr_length; index ++ )
        *this += larray[index];
                                // loop == op+=(const list&)
    number_of_lists ++;
}
```

- Destructor calls lower level `clear()`, to which the application programmer *will* have access

```
inline list<T>::~~list(void) {
    clear();
    number_of_lists --;    // for bean counter
}
```

Clearing a list

```
// list wiped clean, but not necessarily deleted
template <class T>
inline void list<T>::clear(void) {
    node<T> *pnode;

    // do not care about direction of deleting, as nodes
    // may have been appended, then prepended, etc.
    while (head) {
        pnode = head;
        head = head -> next;
        delete pnode;
    }

    tail = 0;    // do not forget this, as it may be
    return;     // checked someday
}
```

- delete comment: since normally done in LIFO fashion

list Assignment

```
// reference returned for daisy-chaining
template <class T>
inline list<T>& list<T>::operator=(const list<T>& rhs) {
    if (this == &rhs)    // self-test
        return *this;

    clear();    // free up current memory

    (*this) += rhs;    // copy rhs to lhs
    return *this;
}
```

- With previous functions, this is trivial
- Straightforward: self-check, deallocation, copying
- Imagine extra work if rhs sent by value!

Non-self Appending

```
template <class T>
inline list<T> list<T>::operator+(const T& rhs) const {
    list<T> temp_list(*this);    // call to copy constructor

    temp_list += rhs;

    return temp_list;
}
```

- Invocation—only on right hand side

```
list3 = list1 + (short) 87;
// equivalent to:
// list3.operator=(list1.operator+((short) 87))
```

Non-self Appending Features

- `New list` returned \Rightarrow
returned by value, again calling copy constructor
- Note amount of work vs. `operator+=(const T&)`
 - * `operator+=()` no longer just a shorthand
- 2 constructor calls \Rightarrow 2 destructor calls
 - * 1 pair is implicit, by compiler
 - * find all four!
- `inline`— \checkmark , `const`— \checkmark

Similar story for prepending

Prepending

```
template <class T>
inline list<T> operator+(const T& lhs, const list<T>& rhs) {
    list<T> temp_list;

    temp_list += lhs;    // prepending value first
    temp_list += rhs;

    return temp_list;
}
```

- Recall: lhs operand not list \Rightarrow global
- Still inline
- Calls protected: `op+=()` \Rightarrow friend
- Move declaration into `list` class definition

Checking for Value Presence

```
template <class T>
inline bool list<T>::is_present(const T& lvalue) const {
    const node<T> *pnode = head;

    while (pnode) {
        if ((pnode -> value) == lvalue)    // deref. both for
            return true;                  // is_present_ptr()
        pnode = pnode -> next;
    }

    return false;
}
```

- By nature, `is_present()` is readonly \Rightarrow 3 consts for
 - * test value `lvalue`
 - * object's data members `head` and `tail`
 - * all dereferenced info between `head` and `tail`
- Probably can be `inline`

On to list printing.

Printing Out lists

- As usual, `operator<<()` will be global
- Will traverse `list` similar to `is_present()`
- `∴` will access `head` \Rightarrow make it a friend
- Will need access to `node` information
 - * `value` for printing
 - * `next` for traversing `list`
- **But:** friendship of `list` to `node`
 - * is for `list`'s member functions, e.g., `is_present()`
 - * is *not* for `list`'s friends (i.e., non-transitive)

Need `list` member functions for `operator<<()` to use.

Getting node Data

```
template <class T>
inline const T& list<T>::value(const node<T> *pnode) {
    return pnode -> value;
}
```

```
template <class T>
inline const node<T> *list<T>::next(const node<T> *pnode) {
    return pnode -> next;
}
```

- These functions are only for above accessibility reasons
- No reason to declare them in `public`:
- Note: they use nothing of `list` at all
 - * declare as class static

And for friend operator<<()

operator<<()

```
template <class T>
inline ostream& operator<<(ostream& output, const list<T>& rhs) {
    const node<T> *pnode = rhs.head;

    output << '<';
    while (pnode) { // deref. output for list of pointers
        output << list<T>::value(pnode);
        pnode = list<T>::next(pnode);
        if (pnode)
            output << ", ";
    }
    output << '>';

    return output;
}
```

Understood—right?

The Number of Lists

- `number_of_lists` needs to be defined (in `list.C` for non-template class)

```
template <class T>    // global -> initialized to zero
int list<T>::number_of_lists;
```

- Only this global definition allowed for protected `number_of_lists`

- For viewing its value

```
template <class T>
inline int list<T>::num_lists(void) {
    return number_of_lists;
}
```

- `operator-()` and `operator-=()` are left to the reader as a trivial exercise :-) — NOT!

Linked Lists

- Motivation and Design
- Constructor Declarations
- M/G Function Declarations
- Implementation: Data Members
- M/G/F Function Definitions
- ⇒ Sample Driver Program
- Unique List Derivation

Beginning of main()

```
#include <iostream>
#include <complex>    // do not forget to link with -lcomplex
#include "list.h"
#include "list-manipulate.h"
using namespace std;

int main(void) {
//  testing shorts

    short array1[] = {5, -7, 8};
    const int size1 = sizeof(array1) / sizeof(array1[0]);
    const short new_value1 = -19;

    list<short> short_list(array1, size1);
    list_manipulate(short_list, new_value1);
```

Where is the free store involved? How many bytes?

Manipulating the list

```
static const char *yes_or_no(bool value) {  
    if (value)  
        return "yes";  
    else  
        return "no";  
}
```

```
template <class T>  
void list_manipulate(list<T>& some_items, const T& new_value) {
```

Note: already type-free.

Printing, Testing and Appending

Code:

```
cout << "The list to begin with: " << some_items << "."
      << endl << endl;

cout << "The value of " << new_value << " is present: " <<
      yes_or_no(some_items.is_present(new_value)) <<
      "." << endl;

cout << "Appending the value: " << new_value << "." << endl;
some_items += new_value;
```

produces:

The list to begin with: <5, -7, 8>.

The value of -19 is present: no.

Appending the value: -19.

Note how natural `op<<()` and `op+=()` behave for application.

... and Prepending

Code:

```
cout << "The new list: " << some_items << "." << endl;
cout << "The value of " << new_value << " is present: " <<
    yes_or_no(some_items.is_present(new_value)) <<
    "." << endl << endl;

cout << "And prepending the value of: " << new_value <<
    "." << endl;
some_items = new_value + some_items;
cout << "The new list: " << some_items << "." << endl;
```

produces:

The new list: <5, -7, 8, -19>.

The value of -19 is present: yes.

And prepending the value of: -19.

The new list: <-19, 5, -7, 8, -19>.

Overloading of operator+() picks up correct function.

Emptying and Verifying

Code:

```
cout << "The list is empty: " <<
    yes_or_no(some_items.is_empty()) << "." << endl;
cout << "Clearing the list." << endl;
some_items.clear();
cout << "The new list: " << some_items << "." << endl;
cout << "The list is empty: " <<
    yes_or_no(some_items.is_empty()) << "." << endl;
```

produces:

The list is empty: no.

Clearing the list.

The new list: <>.

The list is empty: yes.

Everything okay so far.

The Number of lists

Code:

```
cout << "The number of lists is: " <<
      list<T>::num_lists() << "." << endl;
```

produces:

The number of lists is: 2.

- Ouch!
- Where did the second list come from?!
- Answer: a number of *temporary* list objects were created
 - * destroyed at end of statement (;) or block (})
- Global prepending (operator+()) return value remains
 - * can check by surrounding invocation with braces

A complex list

Back in main()

```
complex array2[] = {0, 0};  
const int size2 = sizeof(array2) / sizeof(array2[0]);  
const complex new_value2(12.3, -4.56);  
  
array2[0] = 2 * new_value2;  
  
list<complex> complex_list(array2, size2);  
list_manipulate(complex_list, new_value2);
```

Note how consistent code is for application programmer.

Output from complex list

The list to begin with: `<(24.6, -9.12), (0, 0)>`.

The value of `(12.3, -4.56)` is present: no.

Appending the value: `(12.3, -4.56)`.

The new list: `<(24.6, -9.12), (0, 0), (12.3, -4.56)>`.

The value of `(12.3, -4.56)` is present: yes.

And prepending the value of: `(12.3, -4.56)`.

The new list: `<(12.3, -4.56), (24.6, -9.12), (0, 0),
(12.3, -4.56)>`.

The list is empty: no.

Clearing the list.

The new list: `<>`.

The list is empty: yes.

The number of lists is: 2. // why is 2 correct?

Note how consistent output is for user.

Linked Lists

- Motivation and Design
 - Constructor Declarations
 - M/G Function Declarations
 - Implementation: Data Members
 - M/G/F Function Definitions
 - Sample Driver Program
- ⇒ Unique List Derivation

Motivation

- Want a `list` without repeated nodes (only first occurrence)
- Other possible derivations: sorted list, unique/sorted list, ...
- As always: build on previous work
- Leads to ... a subclass template: `unique_list`
- What can/cannot be inherited?
- How else can we build on previous work?

Proper design → robust, concise subclass code.

Standard Two Constructors

```
#include "list.h"    // all in unique-list.h

template <class T>
class unique_list : public list<T> {
public:
    unique_list(void) {}    // list<T>() called by def.
    unique_list(const unique_list& rhs) : list<T>(rhs) {}
```

- Recall: constructors are not inherited
- But with initializers, can build on previous code
- Not much to do in default constructor
- Copy constructor initializes with `list<T>` copy constructor
 - * rightfully assumes `rhs` is properly unique
- Recall: no new static members for derived `unique_list`
 - * static counter handled by `list<T>` constructors

Constructing from an Array—Take I

```
// incorrect implementation
unique_list(const T *larray, int arr_length) :
    list<T>(larray, arr_length) {}
```

- This seems like the natural way to proceed
- Problem: during `list<T>` initialization, `unique_list` object is a `list` object
- Outcome: `unique_list` object with repeated nodes
- Possible solution: immediately remove repeated nodes in function body
- Alternatively, never allow repeated nodes to be added

Recall how nodes are added ...

unique_list Self-Appending

```
void operator+=(const T& rhs) {
    if (!is_present(rhs))
        this -> list<T>::operator+=(rhs);

    return;
}
```

- Again, build on previous code when possible
- Do not forget to add "virtual" to `list<T>::operator+=(const T&)` declaration
- Note need for explicit invocation of `operator+=()` form
- Explicit class specification to prevent infinite recursion

Back to third constructor ...

Constructing from an Array—Take II

```
unique_list(const T *larray, int arr_length) {  
    // list<T>() called by default  
    for (int index = 0; index < arr_length; index ++)  
        *this += larray[index]; // correct op+=() called  
}
```

- Note: identical to third list constructor (aside from static object counter)
 - * define base helper function for both
- Alternate design: do not invoke virtual functions from constructor
 - * allow derived class to initialize with repeats
 - * remove repeated nodes in ctor body
- Subclass design: done!! A cinch due to `operator+=(const T&)`
 - * used as common, efficient building block
 - * its *virtuality*

unique_list Test Code

```
#include <iostream>
#include "unique-list.h" // #include's list.h
#include "list-manipulate.h"
using namespace std;

int main(void) {
// testing floats

float array1[] = {6.f, -7.1f, 8.f, -7.1f};
const int size1 = sizeof(array1) / sizeof(array1[0]);
const float new_value1 = 8.f;

unique_list<float> float_ulist(array1, size1);
list_manipulate(float_ulist, new_value1);

return 0;
}
```

- virtuality → list_manipulate() need not change

unique_list Test Output

The list to begin with: <6, -7.1, 8>.

The value of 8 is present: yes.

Appending the value: 8.

The new list: <6, -7.1, 8>.

The value of 8 is present: yes.

And prepending the value of: 8.

The new list: <8, 6, -7.1>.

The list is empty: no.

Clearing the list.

The new list: <>.

The list is empty: yes.

The number of lists is: 2.