Plan of the day

Where are we at?

- session 1: basic language constructs
- session 2: pointers and functions
- session 3: basic class and operator overloading

Today

- design of two types of container classes
- templates
- friend
- nested classes

`SimpleFloatArray` Class

Design and implement an array class with

- run time sizing
- access to element with `x[i]`
- automatic memory management
- automatic copy of array elements
- automatic copy upon assignment
- set all elements of array to a value
- find the current size
- dynamic resizing

Each requirement leads to a member function

There will be some technical issues to learn

Warning: this will not be a production quality class

Why an array class?

Replace these parts of linefit.C

```
    cin >> n;
    float* x = new float[n];
// munch munch
    sx += x[i];
    delete [] x;
```

with

```
    cin >> n;
    SimpleFloatArray x(n);
// munch munch
    sx += x[i];
//    delete [] x;
```

- to avoid pointers
- to get automatic deletion
- to show how to be able to do

```
SimpleFloatArray x(n);
SimpleFloatArray y = x;
SimpleFloatArray z;
//
z = x;   // copy array
x = 0.0; // clears the array
```

SimpleFloatArray Class Declaration

The header file

```
class SimpleFloatArray {
public:
  SimpleFloatArray(int n);            // init to size n
  SimpleFloatArray();                 // init to size 0
  SimpleFloatArray(const SimpleFloatArray&); // copy
  ~SimpleFloatArray();                     // destroy
  float& operator[](int i);                // subscript
  int numElts();
  SimpleFloatArray& operator=(const SimpleFloatArray&);
  SimpleFloatArray& operator=(float);     // set values
  void setSize(int n);
private:
  int num_elts;
  float* ptr_to_data;
  void copy(const SimpleFloatArray& a);
};
```

- `~SimpleFloatArray()` is the *destructor* member function and is invoked when object is deleted
- `float& operator[](int i)` is the member function invoked when the operator `[]` is used
- `operator=()` is member function invoked when doing assignment: the *copy* assignment
- note private member function

Constructor Implementations

Constructors

```
SimpleFloatArray::SimpleFloatArray(int n) {
    num_elts = n;
    ptr_to_data = new float[n];
}

SimpleFloatArray::SimpleFloatArray() {
    num_elts = 0;
    ptr_to_data = 0; // set pointer to null
}

SimpleFloatArray::SimpleFloatArray(const SimpleFloatArray& a) {
    num_elts = a.num_elts;
    ptr_to_data = new float[num_elts];
    copy(a); // Copy a's elements
}
```

- by implementing the default constructor, we ensure that every instance is in well defined state before it can be used
- must implement copy constructor else the default behavior is member-wise copy which would lead to two array objects sharing the same data

Can you think of alternative coding?

copy Implementation

Terse implementation

```
void SimpleFloatArray::copy(const SimpleFloatArray& a) {
  // Copy a's elements into the elements of our array
  float* p = ptr_to_data + num_elts;
  float* q = a.ptr_to_data + num_elts;
  while (p > ptr_to_data) *--p = *--q;
}
```

- uses pointer arithmetic
- uses prefix operators

Fortran style implementation

```
void SimpleFloatArray::copy(const SimpleFloatArray& a) {
  // Copy a's elements into the elements of *this
  for (int i = 0; i < num_elts; i++ ) {
    ptr_to_data[i] = a.ptr_to_data[i];
  }
}
```

- uses array notation on pointer
- uses postfix operator

Destructor Member Function

Implementation

```
SimpleFloatArray::~SimpleFloatArray() {
    delete [] ptr_to_data;
}
```

- one and only one destructor
- function with same name as class with `~` prepended
- no arguments, no return type
- invoked automatically when object goes out of scope
- invoked automatically when object is deleted
- usually responsible for cleaning up any dynamically allocated memory or other resources

`operator[]` Member Function

Implementation

```
float& SimpleFloatArray::operator[](int i) {
    return ptr_to_data[i];
}
```

- overloads what `[]` means for object of this type
- returns *reference* to element in array
- since it is a reference, it can be used on right-hand or left-hand side of assignment operator
- this snippet of code will work (`ch4/linefit.C`)

```
     int n;
     cin >> n;
     SimpleFloatArray x(n);
     SimpleFloatArray y(n);

     for (int i = 0; i < n; i++) {
         cin >> x[i] >> y[i];
     }
     double sx  = 0.0, sy  = 0.0;
     for (i = 0; i < n; i++) {
         sx += x[i];
         sy += y[i];
     }
```

- remember, a reference is not a pointer

`operator=` Member Function

Implementation

```
SimpleFloatArray&
SimpleFloatArray::operator=(const SimpleFloatArray& rhs) {
    if ( ptr_to_data != rhs.ptr_to_data ) {
        setSize( rhs.num_elts );
        copy(rhs);
    }
    return *this;
}
```

- `if()` statements tests that array object is not being assigned to itself.
- `this` is a pointer to the object with which the member function was called.
- must implement else default is member-wise copy leading to two objects sharing the same data
- is the behaviour what we expected?
  - what if lhs is smaller than rhs?
  - what if lhs is larger than rhs?

Assignment versus Copy

Copy Constructor

```
SimpleFloatArray::SimpleFloatArray(const SimpleFloatArray& a) {
    num_elts = a.num_elts;
    ptr_to_data = new float[num_elts];
    copy(a); // Copy a's elements
}
```

Assignment operator

```
SimpleFloatArray&
SimpleFloatArray::operator=(const SimpleFloatArray& rhs) {
    if ( ptr_to_data != rhs.ptr_to_data ) {
        setSize( rhs.num_elts );
        copy(rhs);
    }
    return *this;
}
```

Use

```
SimpleFloatArray x(n);
SimpleFloatArray y = x;    // copy constructor
SimpleFloatArray z;
//
z = x;   // copy array     // assignment
```

- should not implement one without the other

Scaler assignment

Implementation

```
SimpleFloatArray& SimpleFloatArray::operator=(float rhs) {
    float* p = ptr_to_data + num_elts;
    while (p > ptr_to_data) *--p = rhs;
    return *this;
}
```

- set all elements of array to a value
- invoked by

```
SimpleFloatArray a(10);
a = 0.0; // assignment
```

- not

```
SimpleFloatArray a(10) = 0.0;
```

  which attempts to do both construction and assignment

- might add another constructor function to allocate and assign

```
SimpleFloatArray a(10, 0.0);
```

The remaining implementation

Implementation

```
int SimpleFloatArray::numElts() {
    return num_elts;
}

void SimpleFloatArray::setSize(int n) {
    if (n != num_elts) {
         delete [] ptr_to_data;
         num_elts = n;
         ptr_to_data = new float[n];
    }
}
```

- nothing special here.
- can’t resize (no `realloc()`)
- could save old data with re-write of class.
- But do we really want this member function?

Key points

- must supply destructor function so object can delete memory it allocated before it gets deleted itself
- must supply copy constructor and `operator=()` if member-wise copy is not what we want
- should return reference in case where object could be on left hand side of assignment

Compiler will never tell you that you forgot any of the above.

List of compiler supplied member functions

- default constructor, if no other constructor exits
- copy constructor
- destructor
- assignment operator

Class explosion?

Suppose we want `SimpleIntArray`?

Could copy `SimpleFloatArray`, edit everywhere we find `float` and save to create new class

- tedious work
- duplicate code
- we'll want to the same for `double`, `Hep3Vector`, *etc*.

Could use `void *` instead of `float` and then cast return values.

- only C programmers know what I'm talking about
- bad idea because we lose type safety

If we have `n` data types and `m` things to work with them, we don't want to have to write `n x m` classes

Enter *template* feature of C++ to solve this problem

`SimpleArray` Template Class

Class declaration

```
template<class T>
class SimpleArray {
public:

    SimpleArray(int n);
    SimpleArray();
    SimpleArray(const SimpleArray<T>&);
    ~SimpleArray();
    T& operator[](int i);
    int numElts();
    SimpleArray<T>& operator=(const SimpleArray<T>&);
    SimpleArray<T>& operator=(T);
    void setSize(int n);
private:
    int num_elts;
    T* ptr_to_data;
    void copy(const SimpleArray<T>& a);
};
```

- `template<class T>` says what follows is a template for producing a class
- `<class T>` is the template argument
- `T` is arbitrary symbol for some type, either built-in or programmer defined (not necessarily a class)
- line breaking is a style issue

Use of Class Template

Line fit with template class

```
void linefit() {

    int n;
    cin >> n;
    SimpleArray<float> x(n);
    SimpleArray<float> y(n);

    // Read the data points
    for (int i = 0; i < n; i++) {
        cin >> x[i] >> y[i];
    }
    // the rest is the same as before
```

- `SimpleArray<float>` is now a class
- `float` replaced `class T`
- use a template class like any other class
- any type can be used

  ```
  SimpleArray<Hep3Vector> x(n);
  ```

  provided that all member functions used by the template exist in the type.

Function Templates

Suppose we have

```
inline double sqr(double x) {
    return x * x;
}
```

Templated version

```
template<class T>
inline
T sqr(T x) {
    return x * x;
}
```

Now we can do

```
   int i = 1;
   float f = 3.1;
   Hep3Vector v(1, 1, 1);

   cout << sqr(i) << endl;
   cout << sqr(f) << endl;
   cout << sqr(v) << endl;
```

- using the templated function generates one of the correct type
- square of a `Hep3Vector` makes no sense

List or Array?

`SimpleArray` is fixed in size once created or reassigned

Sometimes what we really want is a `List`

- add incrementally objects to a list
- remove objects from a list
- list should resize itself automatically
- provide a means to iterate through the list
- find member of a list
- insert an object at particular point in the list
- sort a list
-
-
-
-

Use of a `List`

Normalizing some numbers to minimum value

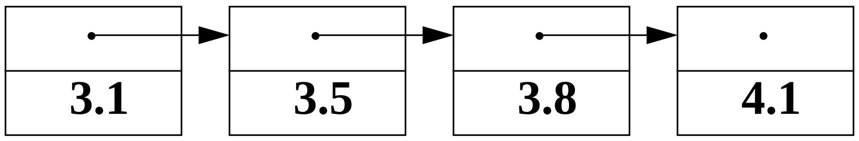
```
int main() {
    // Read list of values and find minimum.
    List<float> list;
    float val;
    float minval = FLT_MAX; // from <float.h>
    while ( cin >> val) {
        if (val < minval) minval = val;
        list.add(val);
    }

    // Normalize values and write out.
    for (ListIterator<float> i(list); i.more(); i.advance()) {
        cout << i.current() - minval << endl;
    }
    return 0;
}
```

- reads until end of file
- finds minimum value
- adds to list
- iterate through list to normalize

Linked List

A popular data structure



Advantages of a list compared to an array

- fast to re-size
- fast to insert

Disadvantage of a list compared to an array

- more memory per element
- slow to random access

The `Node` and `List` classes

Declaration and implementation

```
template <class T>
class Node {
private:
    Node(T x) : link(0), datum(x) {}
  // perhpas more not shown
    Node* link;
    T     datum;
};
```

- uses initializers

Declaration and implementation

```
template<class T>
class List {
public:
    List() : first(0), last(0) {}
    void add(T x) {
        if (first == 0) first = last = new Node(x);
        else last = last->link = new Node(x);
    }
private:
    Node* first;
    Node* last;
};
```

- data members point to first and last nodes in order to quickly add a node to end of list

Problems

Some design issues

- If `Node` class will only be used by `List`, then should it take such a simple name?
- If we always use ListIterator to access data, then do we have to provide three accessor functions?

The answers makes use of two new features:

- nested classes
- `friend` declaration

Warning: this will not be production quality class

`List` with nested `node` class

Declaration and implementation

```
template<class T>
class List {
public:
    List() : first(0), last(0) {}
    void add(T x) {
        if (first == 0) first = last = new Node(x);
        else last = last->link = new Node(x);
    }
    friend class ListIterator<T>;
private:
    class Node {
    public:
        Node(T x) : link(0), datum(x) {}
        Node* link;
        T     datum;
    };
    Node* first;
    Node* last;
};
```

- not only nested, but private as well
- `Node` as a class name is not visible outside of `List`
- did not have to repeat `template` keyword
- `friend` keyword allows access of private data members to `ListIterator<T>` class

ListIterator class

Declaration and Implementation

```
template<class T>
class ListIterator {
public:
   ListIterator(const List<T>& list) : cur(list.first) {}

    bool    more()    const { return cur != 0;   }
    T       current() const { return cur->datum; }
    void    advance()       { cur = cur->link;   }

private:
    List<T>::Node* cur;
};
```

- violation of private parts?
- In `List` we had

```
   friend class ListIterator<T>;
```

- `List<T>::Node*` scoping is needed because `Node` as a class name is not visible even to a friend
- note that `List` was easier to implement than `SimpleArray`

Iterators

Compare

```
SimpleArray<float> a(n);
// ..
for (int i = 0; i < a.numElts(); i++) {
    sum += a[i];
}
```

with

```
List<float> list;
// ..
for (ListIterator<float> i(list); i.more(); i.advance()) {
    sum += i.current();
}
```

- `i` is the iterator in both cases
- both initialize `i` to first element
- both use `i` to test for completion
- both increment `i` to next element
- both use `i` to reference element
- the `ListIterator` version is more tolerant to changes
- need seperate object in case of nested loops

Homework

Write a `SimpleArrayIterator<>` class with

- template class to work with `SimpleArray<>` class
- only four member functions: constructor, `advance()`, `current()` and `more()`

We know the behavior, but what are the data members?

Iterators++

Compare

```
SimpleArray<float> a(n);
// ..
for (int i = 0; i < n; i++) {
    sum += a[i];
}
```

with

```
List<float> list;
// ..
for (ListIterator<float> i(list); i.more(); i++) {
    sum += *i;
}
```

- implement `operator++()`
- implement the deference operator
- make interator look like pointers

Use of Containers

Chamber containing layers

```
class Chamber {
//
private:
    SimpleArray<Layer> layers;
// ...
}
```

- size is known at compile time

Event containing tracks

```
class Event {
//
private:
    List< Track * > tracks;
// ...
}
```

- size not known at compile time

Why use pointers?

- avoid copying object into list
- needed when same object is reference by multiple lists, *e.g.* tracks can share hits
- but must be careful of memory management

CLHEP containers

`HepAList<class T>`

- a templated class
- used by HEP experiments that were early adoters of C++
- stores pointers to objects, *i.e.* does not copy objects, but since this is not visible from declaration it is probably a bad idea
- try to behave like both list and array, which is also probably a bad idea
- has associated iterator, that’s good
- not bad for work done in the early 90’s
- being phased out in favor of standards

Standard Template Library (STL)

Features

- a minimual set of templated containers
- very much iterator based
- supplies functions that can work with most kinds of containers
- extremely well designed

Status

- contributed by HP labs, Palo Alto
- part of the draft standard since July 1994
- at that time under UNIX, HP reference version compiled only with IBM’s xlC
- (Hanover) hacked version worked with gcc
- today its well supported
- books have been written about it (for example, Musser and Saini)
- could spend one session on just it, but we don’t have time