Applied Artificial Intelligence for Engineers
Course Goals

- Introduce several techniques from artificial intelligence and advanced computer science.
- Show how each of these techniques can be applied to engineering problems.
- Give students some practice applying each technique.
- Give an example of the current state-of-the-art.
- Identify the strengths and weaknesses of each technique.
- Teach students the fundamentals of object-oriented programming and give some practice in doing it.

Artificial Intelligence and Advanced Computer Science

- **Computer Science**
  - **Understand the physical and mathematical limits of computation**
  - **Improve the ways computers are applied**
- **Artificial Intelligence**
  - **Understand the mathematical and computational limits of "intelligent" behavior**
  - **Improve the ability of computers to behave "intelligently"**

What is "Intelligent" behavior?

If we ignore computational limitations, intelligent behavior is simply optimal behavior for the task at hand:

- **Chess:** Play perfect chess.
- **Medicine:** Prescribe the therapy that optimizes cost-effectiveness.
- **Engineering design:** Produce the design that simultaneously maximizes usability, functionality, safety, manufacturability, maintainability, and marketability while minimizing costs.

Sometimes it isn't easy to specify optimal behavior because of tradeoffs involving (a) lack of knowledge and (b) competing objectives.

What is “Intelligent” behavior? (continued)

Unfortunately, even with very fast computers, most problems are computationally intractable—we can’t compute optimal solutions.

Intelligent behavior therefore becomes “doing the best you can” with the available resources.

People are able to find very good solutions to many difficult problems (e.g., in design, in signal processing, etc.). Much work in AI is based on studying how people solve problems and then programming computers to use similar methods.

Methods and Problems

	Design	Diagnosis	Schedling	Pattern Recognition
State-Space Search	✓		✓	
Object-Oriented Programming	✓	✓		
Machine Learning			✓	✓
Bayes Networks		✓		

Course Management

- **Approximately two weeks will be devoted to each AI technique.**
- **In each unit, there will be three kinds of assignments:**
  - **Written Homework**
  - **Programming Assignment in C++**
  - **Assignment to practice using the technique**
- **Hardware and Software:**

  The official hardware resources for this course are the ENGR HP Unix machines (Dearborn 115, ECE 241, Rogers 338, Hovland 108). You may use other computers (e.g., PC's, Mac's, etc.) if you have access to them, but you will need to port the C++ code to work on your machine, and I can't help you do that.

  Software will be supported on the HP UNIX workstations using the GNU g++ compiler (which is free and can also be obtained for Windows and NT).

Course Management (2)

- **World Wide Web.** Assignments, handouts, solutions, software etc. will all be made available on our class web page. `http://www.cs.orst.edu/~tgd/classes/533/index.html`.
- **Newsgroup.** The Usenet newsgroup `orst.cs.533` will contain announcements of things like where the class will meet, changes in due dates, new information on the web pages, etc.

UNIT 1: Introduction to C++

- **An example program**
- **Data types, variables, initialization**
- **Control structures**
- **Input and output**
- **Function definition, parameter passing and return**
- **Classes: data members, function members**
- **Classes: `rational` example**
- **Classes: copy constructors**

An Example Program

```
#include <iostream.h>

int main()
{
    // read in two numbers and print their sum
    int  num1;
    int  num2;
    cout << "Enter an integer: ";
    cin >> num1;
    cout << "Enter a second integer: ";
    cin >> num2;
    cout << "The sum is " << num1 + num2 << endl;
    return 0;
}
```

Data types, variables, initialization

C++ has the following builtin data types:

```
int     integer (sometimes only 16 bits on PC's)
float   floating point number
double  double-precision floating point number
char    8-bit character
```

In addition, you can prefix these with the words `long` and `unsigned` to get longer versions or unsigned (non-negative) versions.

To define a variable, you give the type name, the variable name and a semicolon. You can also provide an initial value.

```
int   var1;              // an integer variable
float var2 = 5.23;       // a floating point variable initialized to 5.23
char  var3 = 'w';        // a character constant enclosed in single quotes
int   v1 = 12, v2 = 15;  // multiple variables in a single line
```

Simple arrays

C++ also supports simple arrays. No bounds checking is done on them. Arrays can be initialized by typing a list of values in brackets.

```
int  var1[20];  // an integer array with 20 elements
char var2[5] = {'a', 'b', 'c', 'd', 'e'};
char var3[10] = "abcdefghi";
```

A string constant with k characters in it is actually an array of $k+1$ characters—the last character is the null character (binary value 0). That is why I allocated 10 characters in `var3` for a 9-character string.

Special characters can be entered using the following backslash forms:

```
'\0'        null character
'\n'        newline character
'\t'        tab character
'\\'        backslash character
```

Array Access

```
char var2[5] = {'a', 'b', 'c', 'd', 'e'};

var2[0] = 'b';      // indexing is from 0
var2[4] = '\0';     // null terminate
cout << var2;       // character arrays can be printed if null terminated
```

Pointers

A pointer is a variable that contains a memory address (the address of some other item). Pointers are needed for constructing complex data structures such as lists and trees.

```
int int1;        // an integer
int * pint1;     // a pointer to an integer

int1 = 2;
pint1 = & int1; // prefix & returns the address of its argument
cout << "pint1 points to " << *pint1 << endl;
(*pint1) = 3;  // changes the value of int1 to be 3
```

The `&` operator returns a pointer to its argument.

The `*` operator *de-references* the pointer and returns the object that it points to.

References

A reference is another name for a data object. It is essentially a pointer whose value cannot be changed and that is automatically dereferenced. I like to use references so that I don't need to remember to dereference the pointer.

```
int int1;
int & rint1 = int1;     // a reference to an integer
// int1 and rint1 are two different names for the same memory cell
int1 = 2;
cout << "rint1 refers to " << rint1 << endl;
rint1 = 3;  // changes int1 to be 3
```

Arithmetic Operators

```
int int1, int2;
float f1, f2;

int1 + int2;  int1 - int2; int1 * int2; - int1;
  f1 + f2;      f1 - f2;     f1 * f2;   - f1;
int1 / int2;    // this returns an integer and discards remainder
int1 % int2;    // this returns the remainder as an integer
f1 / f2;        // returns floating point result

f1 + int1;      // int1 is converted to float.
int1 = int(f1);  // truncate f1 to get its integer part.
int1 / float(int2);  // returns floating point result.
```

Special Operations on Data Objects

C++ supports many shorthand forms of common operations on objects.

```
int var;
cout << var++;   // print value of var, then increment by 1
cout << ++var;   // increment by 1, then print value of var
cout << var--;   // post-decrement
cout << --var;   // pre-decrement
var += 5;        // increment by 5
var -= 2;        // decrement by 2
var *= 3;        // multiply by 3
var /= 2;        // divide by 2
```

Comparison Operations

```
int a, b;
if (a > b)  cout << "a is greater than b";
if (a >= b) cout << "a is greater than or equal to b";
if (a < b)  cout << "a is less than b";
if (a <= b) cout << "a is less than or equal to b";
if (a == b) cout << "a is equal to b";
if (a != b) cout << "a is not equal to b";
```

Logical (boolean) operations

In C++, there is no special data type for values of TRUE and FALSE. Instead, TRUE is any nonzero value, and FALSE is any zero value.

```
int a, b;
if (a && b) cout << "a and b are both true";
if (a || b) cout << "a is true or a is false and b is true";
if (!a)     cout << "a is false";
if (a)      cout << "a is true";
```

Control Structures

C++ has the usual control structures.

```
if (a > 5) {
  cout << "a is too big.\n";
}
else {
  cout << "a has a legal value.\n";
}
```

I recommend using the curly braces in all `if` statements. However, it is legal to write

```
if (a > 5) cout << "a is too big.\n";
else       cout << "a has a legal value.\n";
```

Note the semicolon after the `if` part! You need this if you are writing a single statement, but not if you use the curly braces.

Looping Structures

```
int i = 0;
while (i < 10) {
  cout << i << endl;
  i++;
}
// similar for statement.  Scope of i is only inside the loop.
for (int i = 0; i < 10; i++) {
   cout << i << endl;
}
// the do statement:  always executes the body at least once.
int i = 0;
do {
  cout << i << endl;
  i++;
} while (i < 10);
```

Loop Escapes

Searching an array for a value:

```
int  len = 100;
char  array[len];
char  target;
// OMITTED: fill array and target somehow.
int  i;  // note declaration OUTSIDE the for loop so that we can
         // refer to its value after the break.
for (i = 0; i < len; i++) {
    if (array[i] == target) break;
}
if (i < len) {
    cout << "Target found at position " << i << endl;
} else {
    cout << "Target not found" << endl;
}
```

Loop Skips

Skip certain array elements.

```
int  len = 100;
char  array[len];
char  target;
// OMITTED: fill array and target
int  i;
for (i = 0; i < len; i++) {
    if (array[i] == target) continue;
    cout << array[i];
}
```

Input and Output

```
#include <iostream.h>
#include <fstream.h>
int main()
{
  {
    ofstream  outputlog("log1");  // output file stream
    for (int i = 1; i < 100; i++) {
      outputlog << i << endl;
    }
  }   // outputlog is "closed" here automatically
  return 0;
}
```

There are also `ifstream` objects for input stream.

`"log1"` is an example of an *constructor argument* (see below).

Function definition and parameter passing and return

```
int  square(int arg)
{
  return arg * arg;
}

int main()
{
  cout << "The square of 3 is " << square(3) << endl;
  cout << "The square of this is " << square(square(3)) << endl;
  return 0;
}
```

Parameters (except arrays) are passed by *value*. In other words, each parameter is copied when it is passed. This means that the called function cannot change the value of the parameter that was passed. The return value is also copied when it is returned.

Reference Parameters

The most common use of references is for parameter passing and value return.

```
void double(int & arg)    // no return value
{
   arg = arg + arg;
   return;              // note that no value is given here.
}


int main()
{
    int  var1 = 5;
    double(var1);
    cout << "The value of var1 is now " << var1 << endl;
    return 0;
}
```

Reference parameters are not copied. They are like `VAR` parameters in Pascal. When the called routine changes the value of `arg`, this changes the value of `var1`.

Reference Values

Sometimes we want a function to return something that we can assign into.

```
int & find(int key, int arg[], int length)
{
  for (int i = 0; i < length; i = i + 1) {
    if (arg[i] == key) return arg[i];
  }
  return arg[0];
}

int main()
{
  int nums[10];
  // change the first occurrance of 5 in the nums array to be a 2.
  find(5, nums, 10) = 2;
  return 0;
}
```

Classes: data members and function members

Suppose we wanted to extend C++ to be able to process rational numbers. For example, we would like to be able to say something like

```
rational num1(3,5);   // the number 3/5
rational num2(4,3);   // the number 4/3
rational sum = num1 + num2;   // 29/15
cout << "The sum of " << num1 << " and " << num2 << " is " << sum
  << endl;
```

We can do this in C++ by defining a rational number class.

Rational Number Class

```
class rational {
public:
        // constructors
        rational    (int num = 0, int denom = 1);
        // accessor functions
        int     numerator    ();
        int     denominator    ();
        // assignments
        rational & operator = (rational &);
        // operations
        rational operator + (rational & right);
        rational operator - (rational & right);
        rational operator / (rational & right);
        rational operator * (rational & right);
        rational operator - ();
private:
    // data areas
    int     top;
    int     bottom;
    // operations used internally
    void     normalize    ();
};                        // note semicolon here.
```

Notes on the Class Declaration

The class contains functions and data items. Some of these are publicly accessible and others are private.

- `top` and `bottom` are private data members.
- `normalize()` is a private function that takes no arguments.
- `rational()` is the constructor (see below).
- `numerator()` and `denominator()` are accessor functions that return the `top` and `bottom`. This allows anyone to *read* these data fields but not write them.
- Six functions are defined that *overload* builtin C++ operators. These have the weird syntax `operator +`, `operator -`, and so on.

Class Implementation (1): Simple functions

In a separate file, we must provide an implementation for each member function declared in the class declaration. The syntax is the same as for defining an ordinary function except that the function name is prefixed by `rational::`.

```
int rational::numerator ()
{
  return top;
}
int rational::denominator ()
{
  return bottom;
}
```

We could invoke these functions by the following:

```
  rational a(6,10);   // create a rational and initialize to 6/10
  cout << "The numerator is " << a.numerator()
      << " and the denominator is " << a.denominator() << endl;
```

Member Access

Inside the member function definition, the names of any data or function member (e.g., `top`, `normalize()`) refer to the so-called `receiver`. In this case, the receiver is the rational number `a`. So `a.numerator()` returns the value of the `top` private data member of `a`.

The names of the private data members and member functions are only visible inside the bodies of other member functions for this class.

Class Implementation (2): The Constructor

The constructor is called automatically by the compiler whenever a new **`rational`** is created (e.g., whenever a variable of type **`rational`** is declared as in **`rational a(6,10);`** above.

```
rational::rational(int numerator, int denominator)
    : top(numerator), bottom(denominator)
{
    normalize();
}
```

This is like a function declaration except:

- There is no return type.
- The function header can be followed by a colon and a list of initialization forms.
- There is no return value.

At the time the constructor is invoked, memory has been allocated for the **`rational`**, but that memory has not been initialized.

In this example, we initialize the numerator and denominator and then invoke the private **`normalize`** function.

Class Implementation (3): Other member functions

```
rational rational::operator + (rational & right)
{
  rational result(numerator() * right.denominator() +
                  right.numerator() * denominator(),
                  denominator() * right.denominator());
  return result;
}
```

When we execute `a + b` (where `a` and `b` are `rational`s), this member function is invoked. `a` is the receiver and `b` is the `right` argument.

Hence, `numerator()` is the `top` field of `a`, while `right.numerator()` is the `top` field of `b`.

File Organization

The class declaration would normally be stored in a file called `rational.h`. This is called the *header file*. Every program that wants to use `rational` numbers should include this file by using

```
#include "rational.h"
```

at the top of the file.

The definitions of the various member functions would normally be stored in a file called `rational.C`, which is the *implementation file*. When it is compiled, the code implementing the member functions will be stored in `rational.o`.

A key idea in object-oriented programming is the separation between the interface (header file) and the implementation. Indeed, the implementation can be kept secret by distributing it only in compiled (`.o`) form.

The `make` facility

C++ supports separate compilation—that is, you can divide your program up into several files and compile each file separately. Then you can link the entire set of files together to create an executable program. The main advantage of this is that you can divide the program into conceptual chunks and when you change one chunk, you don't need to recompile all of the others.

However, sometimes one chunk depends on another. So when you edit one file, you need to recompile some other files as well. The **`make`** facility supports this. Suppose we have a program **`main.C`** that contains our **`main()`** function and that is using rational numbers.

Contents of the `Makefile`

```
CC = g++
CFLAGS = -g

main: main.o rational.o
	$(CC) $(CFLAGS) main.o rational.o -lm -o main

rational.o: rational.h rational.C
	$(CC) $(CFLAGS) rational.C -o rational.o

main.o: rational.h main.C
	$(CC) $(CFLAGS) main.C -o main.o
```

Note: The indented lines in this file are indented using a single TAB character, not spaces!

Makefile structure

The make file begins by defining two variables `CC` and `CFLAGS`. This makes it easy for us to change the compiler or the compiler flags later.

Then it contains a series of file commands. Each command occupies two lines. The first line declares the *dependencies*. For example, the line

```
main: main.o rational.o
```

says that the file `main` depends on the files `main.o` and `rational.o`. If either of those `.o` files changes, then `main` needs to be rebuilt.

The next line

```
        $(CC) $(CFLAGS) main.o rational.o -lm -o main
```

tells how to rebuild `main`: invoke the compiler and pass it `main.o`, `rational.o`, and the math library (`-lm`) and write the output on `main`.

Using the `make` facility

After you edit your source files, you can type

```
% make main
```

to the shell, and **`make`** will check all of the dependencies that lead to **`main`**. If any file is older than one of the files it depends upon, then that file is remade (and the test is repeated, recursively).

For example, suppose we edit **`rational.C`**. Then **`make`** will recompile **`rational.C`** to produce **`rational.o`** and then relink **`rational.o`** with **`main.o`** to produce **`main`**.

Copy Constructors

```
rational::rational (rational & old): top(old.top), bottom(old.bottom)
{}

rational square(rational r1)
{
  return r1 * r1;
}

// in main()
  rational r(2,3);
  cout << square(r);
```

The copy constructor is called whenever it is necessary to create a copy of a rational: when passing a rational by value and when returning a rational. If no copy constructor is defined for a class, then the default is to copy each component bit-by-bit (shallow copy).

Defining Input/Output Operations

Because the `<<` and `>>` functions take an `iostream` as their left argument, they cannot be members of the `rational` class.

```
#include <iostream.h>

ostream & operator << (ostream & out, const rational & value)
{   // stream output of rational number
    out << value.numerator() << " / " << value.denominator();
    return out;
}
```

Note that we return a *reference* to the stream. This permits chaining:

```
cout << "The sum of " << r1 << " and " << r2 << " is " << r1 + r2 << endl;
```

Dynamic Storage Allocation and Pointers

By default, all storage for variables is allocated on the runtime stack. This means that it is allocated upon entry to a function (or block) and released upon exit from the block. This makes it impossible (or at least very expensive) to allocate large objects in a function and then return them. Consider, for example, a function that creates a copy of a `State` object.

```
State & stateCopy (State & old)
{
    State answer;        // allocate a State on the stack.
    answer.p1 = old.p1;
    answer.p2 = old.p2;  // etc
    return answer;       // Returns a reference to object on stack.
    // storage for answer is released here.
}
// in main:
  State test;
  State & test2 = stateCopy(test, 20);
  // test2 does not refer to legal memory
```

Allocating New Memory

The special operator `new` allocates new space in a different part of main memory (referred to as the *heap*). `delete` releases this memory.

```
State * stateCopy (State & old)
{
   State * answer = new State;  // answer holds the address of a State
   answer->p1 = old.p1;         // -> follows the pointer and then
   answer->p2 = old.p2;         // accesses the p1 (or p2) fields.
   return answer;
}

// in main:
  State test;
  State * test2 = stateCopy(test, 20);
  // omitted: do something with test2
  delete test2;      // now release the storage for test2
```

An even better way

If `State` has a copy constructor defined, this could be written as

```
// in main:
  State test;
  State * test2 = new State(test);  // invokes copy constructor
  // omitted: do something with test2
  delete test2;   // return the storage for test2
```

Inheritance

Consider a class for representing points in the X-Y plane:

```
class point_t
{
 public:
   // constructors
   point_t(double,double);
   point_t (& point_t);
   // accessors
   double givx();
   double givy();
   void setx(double);
   void sety(double);
 protected:
   double x;
   double y;
} ;
```

The only unusual thing here is the `protected:` declaration. These are private members that can be accessed by *subclasses*.

A 3-dimensional Point Class

```
class point3_t: public point_t
{
  public:
    point3_t(double,double,double);

    double givz();
    void setz(double);

  protected:
    double z;
}
```

An object of type `point3_t` inherits all of the data and function members of the `point_t` class *except the constructors and destructors.*

Using the `point3_t` class

```
point3_t  p3(1.0,2.0,3.0);

p3.givx();      // evaluates to 1.0
p3.setx(4.0);   // p3 is now (4.0,2.0,3.0)
p3.setz(0.0);   // p3 is now (4.0,2.0,0.0)
```

Defining the Constructor for `point3_t`

```
point3_t::point3_t(double a, double b, double c): point_t(a,b), z(c)
{}
```

We invoke the constructor for **`point_t`**. The order in which the initializations appear is not important. All initializations of ancestor classes are performed first (with the most distant ancestors initialized before their children). Then the data member initializations are performed.

Destructors

C++ also permits user code to be executed when an instance of a class is about to be destroyed. This allows the code to clean up. For example, suppose we had a class to represent a linked list of integers:

```
class link
{
 public:
   link(int, link *);
   ~link();
   int a;
   link * next;   // pointer to the next element in the list, or 0
};
link::link(int i, link * ptr): a(i), next(ptr)
{}
link::~link()
{
  if (next) delete next;
}
```

Creating a Linked List

```
void main()
{
  // create a linked list of three links 1, 2, 3, allocated on the heap
  link * head = new link(1, new link(2, new link(3,0)));
  // real stuff skipped
  delete head;
}
```

When we delete the first link of the linked list, we need some way of freeing up all of the other elements in the list. This is accomplished by the *destructor*.

The Destructor (2)

Whenever storage for an instance of a class is about to be released (either on the stack or the heap), the *destructor* function for that class is invoked first. After the destructor returns, the memory is released.

In this case, the destructor recursively deletes all of the elements of the linked list, so all memory is released.

General Rule: If a class has a data member that stores a pointer to heap-allocated memory, then that class needs a destructor to release that heap-allocated memory.

Virtual Function Inheritance

```
class prim_t  // primitives
{
  public:
    prim_t * givnext();
    virtual void draw( scene_t &);
  protected:
    prim_t * next;  // ptr for linked list
};

class line_t: public prim_t  // lines
{
  public:
    void draw(scene_t &);
};

class circle_t: public prim_t  // circles
{
  public:
    void draw(scene_t &);
};
```

```
class scene_t   // a scene with lines and circles
{
  public:
    void draw();
  private:
    prim_t * head;   // head of linked list
};

void scene_t::draw()
{
  for(prim_t * ptr=head;
      ptr;
      ptr = ptr->givnext())
  {
    ptr->draw(*this);
  }
}
```

Virtual Function Inheritance (2)

Each type of object (lines and circles) indicates how to draw itself by redefining the `draw()` member function that is initially defined in the `prim_t` class. Suppose we create a scene named `sc` with a linked list of primitives that contains some circles and some lines. When we invoke

`sc.draw()`,

it loops through the primitives and invokes the `draw` function for each of them. Because `draw` is a virtual function, each instance contains information that indicates which class it belongs to. So when `ptr` points to a `circle_t`, then the `circle_t::draw()` function is invoked. But when `ptr` points to a `line_t`, the `line_t::draw()` function is invoked.

Template Functions

Sometimes we want to write a function that works for many different types of arguments. Consider `min(x,y)` which returns the minimum of its two arguments. We would like versions of `min` that work for `int`s, `float`s, and even `rational`s. But we would like to write the code for `min` just once. Template functions make this possible:

```
template <class T>
T min(T & arg1, T & arg2)
{
  if (arg1 < arg2) return arg1;
  else return arg2;
}
```

Templates (2)

The `template <class T>` defines a "type variable" `T` that can stand for any type. The function `min` takes two arguments of type `T` and returns a value of type `T`. When the compiler sees an instance of `min` applied to two `int`s, it creates a copy of this function definition with `T` replaced by `int`.

```
int min (int & arg1, int & arg2)
{
  if (arg1 < arg2) return arg1;
  else return arg2;
}
```

and arranges for it to be compiled. Similarly, when it sees `min` applied to two `rational`s, it creates a copy with `T` replaced by `rational`.

For this to work, the `rational` class must define `rational::operator <`. Otherwise, a compiler error will be given.

Template Classes

It is possible to create an entire class using template variables. Let's consider the linked list example from before. We defined a linked list of `int`s. But we could also define a linked list of any type `T`:

```
template <class T> class link
{
 public:
   link(T, link *);
   ~link();
   T a;
   link * next;   // pointer to the next element in the list, or 0
};

template <class T> link::link(T i, link * ptr): a(i), next(ptr)
{}
```

Creating an Instance of a Template Class

```
link<int> * head = new link<int>(1, new link<int>(2, new link<int>(3,0)));
```

This creates a linked list containing three `link` objects with `T` replaced by `int`.

C++ Class Libraries

In the programs we write in this class, we will use a class library developed by Professor Tim Budd of the CS Department.

The `vector` class:

```
vector<int>   v1(20);   // a vector of 20 integers
vector<float> v2(10,3.1416); // a vector of 10 floats, initialized to pi
```

These vectors behave very much like C++ arrays. You can index the i-th element as `v1[i]`. However, they provide three other services:

- **Bounds Checking**. An attempt to access an illegal index will result in the runtime error "`Assertion failed index < size`".
- **Size Information**. The member function `length()` returns the number of elements in the vector.
- **Expandable (or contractable)**. The member function `setSize(int newsize)` will change the number of elements. An optional second argument specifies the initial value for the new elements.

Linked Lists

```
list<int>   v1;      // a linked list of integers

cin >> v1;           // text format: ( 3 45 32 2 )
cout << "The list " << v1 << " contains " << v1.length() << " elements.";

v1.add(22);          // add 22 to the head of the list.
cout << "The first element is " << v1.firstElement();
if (v1.includes(0)) cout << "The list contains zero.";
v1.reverse();        // reverse the order of the list.
v1.removeFirst();    // delete the first element of the list.
if(v1.isEmpty()) cout << "No more elements in the list";
cout << "The 3rd element is " << v1[2];
```

Linked lists are useful for implementing search algorithms.

Heaps (Priority Queues)

```
heap<int>   h1;     // a heap of integers.

// Read 10 integers from cin and insert them into the heap
for (int i = 0; i < 10; i++) {
     int element;
     cin >> element;
     h1.add(element);
     }
cout << "The smallest integer in the list was " << h1.min();
cout << "In ascending order, the elements are: ";
while (! h1.isEmpty())
  cout << h1.deleteMin();
}
```

Heaps are useful for many algorithms.

Iterators

One of the most interesting concepts in object-oriented programming is the concept of an iterator. Iterators provide an abstract way of looping over all of the elements in an abstract data type. For example, the following loops over the elements in a linked list:

```
list<int>  thelist;              // a linked list
listIterator<int>  itr(theList); // an iterator over that list

cin >> thelist;
for(itr.init(); !itr; ++itr) {
  cout << "The next element of the list is " << itr() << endl;
}
```

Operations on Iterators

- **Initialization**. `itr.init()`
- **Test for end of collection**. `!itr`.
- **Advance to next element**. `++itr`.
- **Return a reference to current element**. `itr()`. This can be used to assign to the current element. `itr() = 5;`

Vector Iterators

There are two additional operations supported for vector iterators:

- **Access the current index** `itr.key()`.
- **Decrement** `--itr`.

```
vector<int> thevector;
vectorIterator<int> itr(thevector);

for(itr.init(); !itr; ++itr) {
  cout << "Element number " << itr.key() << " is " << itr() << endl;
}
```

List Iterators

There are three special operations on list iterators:

- **Remove current element:** `itr.removeCurrent()`
- **Insert element before current one:** `addBefore(newelem)`
- **Insert element after current one:** `addAfter(newelem)`