C++ Support for Abstract Data Types

Douglas C. Schmidt

Professor
d.schmidt@vanderbilt.edu
www.cs.wustl.edu/~schmidt/

Department of EECS
Vanderbilt University
(615) 343-8197
Topics

- Describing Objects Using ADTs
- Built-in vs. User-defined ADTs
- C++ Support
Describing Objects Using ADTs

- An ADT is a collection of data and associated operations for manipulating that data
- ADTs support abstraction, encapsulation, and information hiding
- They provide equal attention to data and operations
- Common examples of ADTs:
  - Built-in types: boolean, integer, real, array
  - User-defined types: stack, queue, tree, list
Built-in ADTs

- boolean
  - *Values*: true and false
  - *Operations*: `and`, `or`, `not`, `nand`, *etc.*

- integer
  - *Values*: Whole numbers between MIN and MAX values
  - *Operations*: `add`, `subtract`, `multiply`, `divide`, *etc.*

- arrays
  - *Values*: Homogeneous elements, *i.e.*, array of X . . .
  - *Operations*: `initialize`, `store`, `retrieve`, `copy`, *etc.*
User-defined ADTs

- **stack**
  - *Values*: Stack elements, *i.e.*, stack of \(X\) . . .
  - *Operations*: `create`, `destroy/dispose`, `push`, `pop`, `is_empty`, `is_full`, *etc.*

- **queue**
  - *Values*: Queue elements, *i.e.*, queue of \(X\) . . .
  - *Operations*: `create`, `destroy/dispose`, `enqueue`, `dequeue`, `is_empty`, `is_full`, *etc.*

- **tree search structure**
  - *Values*: Tree elements, *i.e.*, tree of \(X\)
  - *Operations*: `insert`, `delete`, `find`, `size`, `traverse` (in-order, post-order, pre-order, level-order), *etc.*
C++ Support for ADTs

- C++ Classes
- Automatic Initialization and Termination
- Friends
- Assignment and Initialization
- Overloading
- Parameterized Types
- Iterators
- Miscellaneous Issues
C++ Classes

• Classes are *containers* for state variables and provide operations, *i.e.*, *methods*, for manipulating the state variables

• A class is separated into three *access control sections*:

```cpp
class Classic_Example {
public:
    // Data and methods accessible to any user of the class
protected:
    // Data and methods accessible to class methods, derived classes, and friends only
private:
    // Data and methods accessible to class methods and friends only
};
```
C++ Classes (cont’d)

- A `struct` is interpreted as a class with all data objects and methods declared in the public section.
- By default, all class members are private and all struct members are public.
- A class definition does *not* allocate storage for any objects.
- Data members and member functions (i.e., methods)
The *this* pointer

- Used in the source code to refer to a pointer to the object on which the method is called

*Friends*

- Non-class functions granted privileges to access internal class information, typically for efficiency reasons
Class Data Members

- Data members may be objects of built-in types, as well as user-defined types, e.g., class Bounded_Strip

```cpp
#include "Vector.h"

template <class T>
class Bounded_Strip {
public:
    Bounded_Strip (int len) : stack_ (len), top_ (0) {}  
    // . . .
private:
    Vector<T> stack_;  
    int top_;  
};
```
Class Data Members (cont’d)

• Important Question: ‘How do we initialize class data members that are objects of user-defined types whose constructors require arguments?’

• Answer: use the base/member initialization section
  – That’s the part of the constructor after the ’:’, following the constructor’s parameter list (up to the first ‘{’)

• Note, it is a good habit to always use the base/member initialization section

• Base/member initialization section only applies to constructors
* Five mandatory cases for classes:

1. Initializing base classes (whose constructors require arguments)
2. Initializing user-defined class data members (whose constructors require arguments)
3. Initializing reference variables
4. Initializing `const`
5. Initializing virtual base class(es), in most derived class (when they don’t have default constructor(s))

* One optional case:

1. Initializing built-in data members
Base/Member Initialization Section (cont’d)

class Vector { public: Vector (size_t len); /* . . . */ };
class String { public: String (const char *str); /* . . . */
class Stack : private Vector // Base class
{
    public:
        Stack (size_t len, const char *name)
            : Vector (len), name_ (name),
                max_size_ (len), top_ (0) {}
    // . . .

    private:
        String name_; // user-defined
        const int max_size_; // const
        size_t top_; // built-in type
    // . . .
};
Base/Member Initialization Section (cont’d)

- References (and consts) must be initialized

```cpp
class Vector_Iterator {
public:
    Vector_Iterator (const Vector &v): vr_ (v), i_ (0) {} // . . .
    // . . .
private:
    Vector &vr_; // reference
    size_t i_;  
};
```
Friends

- A class may grant access to its private data and methods by including friend declarations in the class definition, e.g.,

```cpp
class Vector {
    friend Vector &product (const Vector &, const Matrix &);

private:
    int size_;  // . . .
};
```

- Function `product` can access `Vector`'s private parts:

```cpp
Vector &product (const Vector &v, const Matrix &m) {
    int vector_size = v.size_;  // . . .
```
Friends (cont’d)

- A class may confer friendship on entire classes, selected methods in a particular class, ordinary stand-alone functions.

- Friends allow for controlled violation of information-hiding
  - e.g., ostream and istream functions:

```cpp
#include <iostream.h>
class String {
    friend ostream &operator<< (ostream &, String &);

private:
    char *str_;  
    // . . .
};

ostream &operator<< (ostream &os, String &s) {
    os << s.str_;  
    return os;
}
```

Vanderbilt University
Friends (cont’d)

- Using friends weakens information hiding
  - In particular, it leads to tightly-coupled implementations that are overly reliant on certain *naming* and *implementation* details
- For this reason, friends are known as the ‘goto of access protection mechanisms!’
- Note, C++ inline (accessor) functions reduce the need for friends . . .
Assignment and Initialization

- Some ADTs must control all copy operations invoked upon objects
- This is necessary to avoid dynamic memory aliasing problems caused by “shallow” copying
- A String class is a good example of the need for controlling all copy operations . . .
Assignment and Initialization (cont’d)

class String {
    public:
        String (const char *t) 
            : len_ (t == 0 ? 0 : strlen (t)) { 
            if (this->len_ == 0) 
                throw range_error ();
            this->str_ = strcpy (new char [len_ + 1], t);
        }
        ~String (void) { delete [] this->str_; } 
    // . . .
    private:
        size_t len_;
        char *str_;
};
Assignment and Initialization (cont’d)

void foo (void) {
  String s1 ("hello");
  String s2 ("world");

  s1 = s2; // leads to aliasing
  s1[2] = 'x';
  assert (s2[2] == 'x'); // will be true!
  // . . .
  // double deletion in destructor calls!
}
Assignment and Initialization (cont’d)

- Note that both \texttt{s1.s} and \texttt{s2.s} point to the dynamically allocated buffer storing world (this is known as \textit{aliasing})
Assignment and Initialization (cont’d)

• In C++, copy operations include assignment, initialization, parameter passing and function return, *e.g.*, 

```cpp
#include "Vector.h"
Vector<int> bar (Vector<int>);

void foo (void) {
    Vector<int> v1 (100);

    Vector<int> v2 = v1; // Initialize new v2 from v1
    // Same net effect as Vector v2 (v1);

    v1 = v2; // Vector assign v2 to v1

    v2 = bar (v1); } // Pass and return Vectors
```

• Note, parameter passing and function return of objects by *value* is handled using the initialization semantics of the *copy constructor*
Assignment and Initialization (cont’d)

- Assignment is different than initialization because the left hand object already exists for assignment.
- Therefore, C++ provides two different operators, one for initialization (the copy constructor, which also handles parameter passing and return of objects from functions)...

```
template <class T>
Vector<T>::Vector (const Vector &v)
  : size_ (v.size_), max_ (v.max_), buf_ (new T[v.max_])
{
  for (size_t i = 0; i < this->size_; i++)
    this->buf_[i] = v.buf_[i];
}
```
Assignment and Initialization (cont’d)

• . . . and one for assignment (the assignment operator), e.g.,

```cpp
template <class T>
Vector<T> &Vector<T>::operator= (const Vector<T> &v) {
    if (this != &v) {
        if (this->max_ < v.size_) {
            delete [] this->buf_;
            this->buf_ = new T[v.size_];
            this->max_ = v.size_;
        }
        this->size_ = v.size_;
    }
    for (size_t i = 0; i < this->size_; i++)
        this->buf_[i] = v.buf_[i];
    return *this; // Allows v1 = v2 = v3; }
```

Vanderbilt University
Assignment and Initialization (cont’d)

- Constructors and `operator=` must be class members and neither are inherited
  
  - Rationale
    * If a class had a constructor and an `operator=`, but a class derived from it did not, what would happen to the derived class members which are not part of the base class?!
  
  - Therefore
    * If a constructor or `operator=` is not defined for the derived class, the compiler-generated one will use the base class constructors and `operator=`’s for each base class (whether user-defined or compiler-defined)
    * In addition, a memberwise copy (e.g., using `operator=`) is used for each of the derived class members
Assignment and Initialization (cont’d)

- Bottom-line: define constructors and \texttt{operator=} for almost every non-trivial class . . .
  - Also, define destructors and copy constructors for most classes as well . . .
- Note, you can also define compound assignment operators, such as \texttt{operator +=}, which need have nothing to do with \texttt{operator =}
Restricting Assignment and Initialization

- Assignment, initialization, and parameter passing of objects by value may be prohibited by using access control specifiers:

```cpp
template <class T> class Vector {
public:
    Vector<T> (void); // Default constructor

private:
    Vector<T> &operator= (const Vector<T> &);
    Vector<T> (const Vector<T> &);
};

void foo (Vector<int>); // pass-by-value prototype
Vector<int> v1;
Vector<int> v2 = v1; // Error
v2 = v1; // Error
foo (v1); // Error
```
Restricting Assignment and Initialization (cont’d)

- A similar idiom can be used to prevent static or auto declaration of an object, i.e., only allows dynamic objects!

```cpp
class Foo { public: void dispose (void);
    private: ~Foo (void); // Destructor is private ...
};
Foo f; // error
```

- Now the only way to declare a Foo object is off the heap, using operator new, `Foo *f = new Foo;`
  - Note, the delete operator is no longer accessible

```cpp
delete f; // error!
```

- Therefore, a `dispose` function must be provided to delete the object, `f->dispose ();`
Restricting Assignment and Initialization (cont’d)

- If you declare a class constructor protected then only objects derived from the class can be created
  
  - Note, you can also use *pure virtual functions* to achieve a similar effect, though it forces the use of virtual tables . . .

```cpp
class Foo { protected: Foo (void); };  
class Bar : private Foo { public Bar (void); };  
Foo f; // Illegal  
Bar b; // OK
```

- Note, if Foo’s constructor is declared in the private section then we can not declare objects of class Bar either (unless class Bar is declared as a friend of Foo)
Overloading

- C++ allows overloading of all function names and nearly all operators that handle user-defined types, including:
  - the assignment `operator =`
  - the function call `operator ()`
  - the array subscript `operator []`
  - the pointer `operator ->()`
  - the sequence (comma) `operator ,`
  - the ternary `operator ? :`
  - the auto-increment `operator ++`

- You may not overload:
  - the scope resolution `operator ::`
  - the member selection (dot) `operator .`
Overloading (cont’d)

• Ambiguous cases are rejected by the compiler, *e.g.*,

```c++
int foo (int);
int foo (int, int = 10);
foo (100); // ERROR, ambiguous call!
foo (100, 101); // OK!
```

• A function’s return type is not considered when distinguishing between overloaded instances

  – *e.g.*, the following declarations are ambiguous to the C++ compiler:

```c++
int divide (double, double);
double divide (double, double);
```
Overloading (cont’d)

- `const` and `non-const` functions are different functions, so const-ness may be used to distinguish return values, *e.g.*,

  ```
  char &operator[] (unsigned int);
  const char &operator[] (unsigned int) const;
  ```
Overloading (cont’d)

- Function name overloading and operator overloading relieves the programmer from the lexical complexity of specifying unique function identifier names. *e.g.*,

```cpp
class String {
    // various constructors, destructors,
    // and methods omitted
    friend String operator+ (const String&, const char *);
    friend String operator+ (const String&, const String&);
    friend String operator+ (const char *, const String&);
    friend ostream &operator<< (ostream &, const String &);
};
```
Overloading (cont’d)

```cpp
String str_vec[101];
String curly ("curly");
String comma (", ");
str_vec[13] = "larry";
String foo = str_vec[13] + ", " + curly
String bar = foo + comma + "and moe";
/* bar.String::String (  
    operator+ (operator+ (foo, comma), "and moe")); */

void baz (void) {
    cout << bar << "\n";
    // prints larry, curly, and moe
}
```
Overloading (cont’d)

- Overloading becomes a hindrance to the readability of a program when it serves to remove information.
  - This is especially true of overloading operators!
    * e.g., overloading operators `+=` and `-=` to mean push and pop from a Stack ADT.

- For another example of why to avoid operator overloading, consider the following expression:

```cpp
Matrix a, b, c, d;
// . . .
a = b + c * d; // *, +, and = are overloaded
// remember, standard precedence rules apply . . .
```
Overloading (cont’d)

- This code will be compiled into something like the following:

```cpp
Matrix t1 = c.operator* (d);
Matrix t2 = b.operator+ (t1);
a.operator= (t2);
destroy t1;
destroy t2;
```

- This may involve many constructor/destructor calls and extra memory copying . . .
Overloading (cont’d)

- So, do not use operator overloading unless necessary!

- Instead, many operations may be written using functions with explicit arguments, \textit{e.g.},

  \begin{verbatim}
  Matrix b, c, d;
  ... Matrix a (c);
  a.mult (d);
  a.add (b);
  \end{verbatim}

- or define and use the short-hand operator \( x = \) instead, \textit{e.g.},

  \begin{verbatim}
  a = b + c * d; \text{ can be represented by:}
  Matrix a (c);
  a *= d; a += b;
  \end{verbatim}
Parameterized Types

- Parameterized types serve to describe general container class data structures that have identical implementations, regardless of the elements they are composed of.

- The C++ parameterized type scheme allows “lazy instantiation”
  - *i.e.*, the compiler need not generate definitions for template methods that are not used (or non-template methods).

- ANSI/ISO C++ allows a programmer to explicitly instantiate parameterized types, *e.g.*, `template class Vector<int>;`
Parameterized Types (cont’d)

- C++ templates may also be used to parameterize functions. The compiler generates all the necessary code!

```cpp
template <class T> inline void
swap (T &x, T &y) {
    T t = x; x = y; y = t;
}

int main (int, char **[]) {
    int a = 10, b = 20;
    double d = 10.0, e = 20.0;
    char c = 'a', s = 'b';

    swap (a, b); swap (d, e); swap (c, s);
    return 0;
}
```
Parameterized Types (cont’d)

- C++ standard library provides standard containers, algorithms, iterators and functors. The library is generic in the sense that they are heavily parameterized.
  - Containers - e.g., vectors, list, map, queue etc.
  - Algorithm - e.g., copy, sort, find, count etc.
  - Iterators - e.g., Input, Output, Forward, BiDirectional, Random Access and Trivial
  - Function Objects or Functors - e.g., plus, minus, multiplies etc.

- They were called STL in earlier versions of C++
Template Metaprograms

- Make the compiler act as an interpreter.
- Made possible by C++ template features.
- These programs need not be executed. They generate their output at compile time.

```cpp
template<int N> class Power2 {
public:
    enum { value = 2 * Power2<N-1>::value };
};
class Power2<1> { 
public:
    enum { value = 2 };
};
```
Template Metaprograms (cont’d)

- Very powerful when combined with normal C++ code.
- A hybrid approach would result in faster code.
- Template metaprograms can be written for specific algorithms and embedded in code.
- Generates useful code for specific input sizes during compile times.
- Basically, it is an extremely early binding mechanism as opposed to traditional late binding used with C++.
- Can torture your compiler, and not many compilers can handle this.
Template Metaprograms (cont’d)

- A simple do while loop

```cpp
template<int I>
class loop {
    private: enum { go = (I-1) != 0 }; 
    public:  static inline void f() {
        // Whatever needs to go here
        loop<go ? (I-1) : 0>::f(); 
    }
};

class loop<0> {
    public:
        static inline void f() {
        }
    };

loop<N>::f();
```
Iterators

- Iterators allow applications to loop through elements of some ADT without depending upon knowledge of its implementation details
- There are a number of different techniques for implementing iterators
  - Each has advantages and disadvantages
- Other design issues:
  - ‘Providing a copy of each data item vs. providing a reference to each data item’?
  - ‘How to handle concurrency and insertion/deletion while iterator(s) are running‘
Iterators (cont’d)

- Iterators are central to generic programming

1. *Pass a pointer to a function*
   - Not very OO . . .
   - Clumsy way to handle shared data . . .
2. *Use in-class iterators* (a.k.a. *passive* or *internal* iterators)
   - Requires modification of class interface
   - Generally not reentrant . . .
3. *Use out-of-class iterators* (a.k.a. *active* or *external* iterator)
   - Handles multiple simultaneously active iterators
   - May require special access to original class internals . . .
   - *i.e.*, use *friends*
Iterators (cont’d)

- Three primary methods of designing iterators

1. *Pass a pointer to a function*
   - Not very OO . . .
   - Clumsy way to handle shared data . . .
2. *Use in-class iterators* (a.k.a. *passive* or *internal* iterators)
   - Requires modification of class interface
   - Generally not reentrant . . .
3. *Use out-of-class iterators* (a.k.a. *active* or *external* iterator)
   - Handles multiple simultaneously active iterators
   - May require special access to original class internals . . .
   - *i.e.*, use *friends*
#include <stream.h>

template <class T>
class Vector { 
  public: 
    /* Same as before */ 
    int apply (void (*ptf) (T &)) { 
      for (int i = 0; i < this->size (); i++) 
        (*ptf) (this->buf[i]); 
    } 
  };

  template <class T> void f (T &i) { cout << i << endl; } 

vector<int> v (100); 
// . . .
  v.apply (f);
In-class Iterator Example

#include <stream.h>

template <class T>
class Vector {
    public:
        // Same as before
        void reset (void) {this->i_ = 0;}
        int advance (void) {return this->i_++ < this->size ();}
        T value (void) {return this->buf[this->i_ - 1];}
    private:
        size_t i_
    
    Vector<int> v (100);
    // . . .
    for (v.reset (); v.advance () != 0; )
        cout << "value = " << v.value () << "\n";
Out-of-class Iterator Example

```cpp
#include <stream.h>
#include "Vector.h"

template <class T> class Vector_Iterator {
 public:
   Vector_Iterator(const Vector<T> &v) : vr_(v), i_(0) {}
   int advance() {return this->i_++ < this->vr_.size();}
   T value() {return this->vr_[this->i_ - 1];}

 private:
   Vector<T> &vr_;  
   size_t i_; 
};

Vector<int> v (100);
Vector_Iterator<int> iter (v);
while (iter.advance () != 0)
   cout << "value = " << iter.value () << "\n";
```
Out-of-class Iterator Example (cont’d)

- Note, this particular scheme does not require that VectorIterator be declared as a friend of class Vector
  - However, for efficiency reasons this is often necessary in more complex ADTs
Miscellaneous ADT Issues in C++

- const methods
- New (ANSI) casts
- References
- static methods
- static data members
Const Methods

- When a user-defined class object is declared as const, its methods cannot be called unless they are declared to be const methods
  - *i.e.*, a const method must *not* modify its member data directly, or indirectly by calling non-const methods
Const Methods (cont’d)

- This allows read-only user-defined objects to function correctly, *e.g.*,

```cpp
class Point {
public:
    Point (int x, int y): x_ (x), y_ (y) {} 
    int dist (void) const {
        return ::sqrt (this->x_ * this->x_ + this->y_ * 
                      this->y_); }
    void move (int dx, int dy) { this->x_ += dx; 
                              this->y_ += dy; }
private:
    int x_, y_; 
};
const Point p (10, 20); int d = p.dist (); // OK 
p.move (3, 5); // ERROR
```
New (ANSI) casts

- `static_cast` performs a standard, nonpolymorphic cast
  
  ```
  unsigned int invalid = static_cast<unsigned int> (-1);
  ```

- `const_cast` removes const-ness

  ```
  void Foo::func (void) const
  {
    // Call a non-const member function from a
    // const member function. Often dangerous!!!!
    const_cast<Foo *> (this)->func2 ();
  }
  ```
New (ANSI) casts, (cont’d)

- \texttt{reinterpret\_cast} converts types, possibly in an implementation-dependent manner

\begin{verbatim}
  long random = reinterpret\_cast<long> (&func);
\end{verbatim}

- \texttt{dynamic\_cast} casts at run-time, using RTTI

\begin{verbatim}
void func (Base *bp) {
    Derived *dp = dynamic\_cast<Derived *> (bp);
    if (dp)
        // bp is a pointer to a Derived object
}
\end{verbatim}
References

- Parameters, return values, and variables can all be defined as “references”
  - This is primarily done for efficiency

- *Call-by-reference* can be used to avoid the run-time impact of passing large arguments by value
References (cont’d)

- References are implemented similarly to const pointers. Conceptually, the differences between references and pointers are:
  - **Pointers are first class objects, references are not**
    * e.g., you can have an array of pointers, but you can’t have an array of references
  - References must refer to an actual object, but pointers can refer to lots of other things that aren’t objects, e.g.,
    * Pointers can refer to the special value 0 in C++ (often referred to as NULL)
    * Also, pointers can legitimately refer to a location one past the end of an array
- In general, use of references is safer, less ambiguous, and much more restricted than pointers (this is both good and bad, of course)
Static Data Members

- A static data member has exactly one instantiation for the entire class (as opposed to one for each object in the class), e.g.,

```cpp
class Foo {
public:
    int a_; // Must be defined exactly once outside header!
    // (usually in corresponding .C file)
    static int s_; // (usually in corresponding .C file)
};

Foo x, y, z;
```
Static Data Members (cont’d)

• Note:
  – There are three distinct addresses for Foo::a, i.e., &x.a_, &y.a_, &z.
  – There is only one Foo::s, however . . .

• Also note:

  &Foo::s_ == (int *);
  &Foo::a_ == (int Foo::*); // pointer to data member
Static Methods

- A static method may be called on an object of a class, or on the class itself *without supplying an object* (unlike non-static methods . . .)

- Note, there is no `this` pointer in a static method
Static Methods (cont'd)

- *i.e.*, a static method cannot access non-static class data and functions

```cpp
class Foo {
public:
    static int get_s1 (void) {
        this->a_ = 10; /* ERROR! */; return Foo::s_; 
    }
    int get_s2 (void) {
        this->a_ = 10; /* OK */; return Foo::s_; 
    }
private:
    int a_; static int s_; 
};
```
Static Methods (cont’d)

• Most of the following calls are legal:

```cpp
Foo f;
int i1, i2, i3, i4;
i1 = Foo::get_s1 ();
i2 = f.get_s2 ();
i3 = f.get_s1 ();
i4 = Foo::get_s2 (); // error
```

• Note:

```cpp
&Foo::get_s1 == int (*)(());

// pointer to method
&Foo::get_s2 == int (Foo::*)(());
```
Summary

- A major contribution of C++ is its support for defining abstract data types (ADTs), *e.g.*,
  - Classes
  - Parameterized types

- For many systems, successfully utilizing C++’s ADT support is more important than using the OO features of the language, *e.g. ,
  - Inheritance
  - Dynamic binding