

CS 342: Object-Oriented Software Development Lab  
**C++ Support for Abstract Data Types**

David L. Levine  
 Christopher D. Gill  
 Department of Computer Science  
 Washington University, St. Louis  
 levine,cdgill@cs.wustl.edu

<http://classes.cec.wustl.edu/~cs342/>

## Data Abstraction

- *Organizing Principle:*
  - ‘Decide which types you want; provide a full set of operations for each type’
  - \* *i.e.*, make an abstract data type (ADT), where user-defined types act like built-in types
  - Note, when only one object of a type is required the Data Hiding approach suffices
- *Organizing Language Construct:*
  - User-defined data types plus access control . . .

## Describing Objects Using ADTs

- An ADT is a set of objects and an associated set of operations on those objects
- ADTs support *abstraction*, *encapsulation*, and *information hiding*
  - Basically, enhance representational independence . . .
- 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`, `dispose`, `push`, `pop`, `is_empty`, `is_full`, *etc.*
- queue
  - *Values*: Queue elements, *i.e.*, queue of  $X$  . .
  - *Operations*: `create`, `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.*

## Avoiding Over-Specification

- *Goal*:
  - We want complete, precise, and unambiguous descriptions and specifications of software components
- *Problem*:
  - We do *not* want to be dependent on physical representation, because that's not portable, and too closely tied to implementation
- *Solution*
  - Use ADTs
    - \* ADTs capture essential properties without over-specifying their internal realizations
    - \* ADT interfaces provide a list of *operations* rather than an implementation description, *i.e.*, *what* rather than *how*

## Over-Specification Examples

```
int buffer[100], last = -1;
buffer[++last] = 13;

struct Node {
    int item_;
    Node *next_;
} *p, *first = 0;
// . . .
p = new Node; p->next_ = first; p->item_ = 13; first = p;

template <class T, int SIZE>
class Stack {
// . . .
Stack<int, 100> int_stack;
int_stack.push (13);
```

## C++ Support for ADTs

- *C++ Classes*
- *Automatic Initialization and Termination*
- *Assignment and Initialization*
- *Parameterized Types*
- *Exception Handling*
- *Iterators*

## 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*:

```
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)

- Each access control section is optional, repeatable, and sections may occur in any order
- Note, access control section order may affect storage layout for classes and structs:
- C++ only guarantees that consecutive fields appear at ascending addresses *within* a section, not *between* sections, *e.g.*,

```
class Foo { /* Compiler may not rearrange these! */
int a_;
char b_;
};
class Foo { /* Compile may rearrange these! */
public: int a_;
public: char b_;
};
```

## 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
  - *i.e.*, it is just a cookie cutter . . .
  - Remember this when we talk about nested classes . . .
  - Note, a class with virtual methods will allocate at least one *vtable* to store virtual method definitions

## C++ Class Components

- *Nested classes, structs, unions, and enumerated types*
  - Versions of AT&T cfront translator later than 2.1 enforce proper class nesting semantics
- *Data Members*
  - Including both built-in types and user-defined class objects
- *Methods*
  - Also called *member functions*, only these operations (and friends) may access private class data and operations

## C++ Class Components (cont'd)

- The *this* pointer
  - Used in the source code to refer to a pointer to the object for which the method is called
- *Friends*
  - Non-class functions granted privileges to access internal class information, typically for efficiency reasons

## Nested Classes *et al.*

- Earlier releases of C++, *i.e.*, cfront versions pre-2.1, did not support nested classes, *i.e.*, nesting was only a syntactic convenience
- This was a problem because it prevented control over name space pollution of type names
  - Compare with static for functions and variables
- It is now possible to fully nest classes and structs
  - Class visibility is subject to normal access control . . .
- Note, the (newer) C++ namespace feature is a more general solution to this problem . . .

## Nested Classes *et al.* (cont'd)

```
class Outer {
public:
    class Visible_Inner { /* . . . */ };
private:
    class Hidden_Inner { /* . . . */ };
};
```

```
Outer outer; /* OK */
Hidden_Inner hi; /* ERROR */
Visible_Inner vi; /* ERROR */
Outer::Visible_Inner ovi; /* OK */
Outer::Hidden_Inner ohi; /* ERROR */
```

## Nested Classes *et al.* (cont'd)

- Note,
  - Nesting is purely a visibility issue, it does not convey additional privileges on Outer or Inner class relationships
    - \* *i.e.*, nesting and access control are separate concepts
  - Also, inner classes do *not* allocate any additional space inside the outer class

## Class Data Members

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

```
#include "Vector.h"
template <class T>
class Bounded_Stack {
public:
    Bounded_Stack (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
  - e.g., there are less efficiency surprises this way when changes are made
- Base/member initialization section only applies to constructors

## Base/Member Initialization Section

- Five mandatory cases for classes:
  - Initializing base classes (whose constructors require arguments)
  - Initializing user-defined class data members (whose constructors require arguments)
  - Initializing reference variables
  - Initializing `consts`
  - Initializing virtual base class(es), in most derived class (when they don't have default constructor(s))
- One optional case:
  - 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

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

## Class Method Categories

- Four types of methods
  - Manager functions* (constructors, destructors, and operator=)
    - Allow user-defined control over class creation, initialization, assignment, deallocation, and termination
  - Helper functions*
    - “Hidden” functions that assist in the class implementation
  - Accessor functions*
    - Provide an interface to various components in the class’s state
  - Implementor functions*
    - Perform the main class operations

## Class Method Categories (cont'd)

```
template <class T>
class Vector
{
public:
    // manager
    Vector (size_t len_ = 100);

    // manager
    ~Vector (void);

    // accessor
    size_t size (void) const;

    // implementor
    T &operator[] (size_t i);

private:
    // helper
    bool in_range (size_t i) const;
};
```

## The `this` Pointer

- `this` is a C++ reserved keyword
  - It valid only in non-static method definitions
- `this` textually identifies the pointer to the object for which the method is called

```
void String::print (void) {
    puts (this->str_); // same as puts (str_);
}
```

## The `this` Pointer (cont'd)

- The `this` pointer is most often used explicitly to
  - Pass the object (or a pointer or reference to it) to another function
  - Return the object (or a pointer or reference to it) to another function, *e.g.*,

```
#include <ctype.h>
class String {
public:
    // ...
    String &upper_case (void);
    void print (void) const;
private:
    char *str_;
};
```

## The `this` Pointer (cont'd)

```
String &String::upper_case (void) {
    for (char *cp = this->str_; *cp != 0; cp++)
        if (islower (*cp)) *cp = toupper (*cp);
    return *this;
}
int main (int, char *[]) {
    String s ("hello"); // this == &s
    s.upper_case ().print ();
    /* Could also be:
    s.upper_case ();
    s.print ();
    compare with:
    cout << s.upper_case (); */
    return 0;
}
```

## Friends

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

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

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

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

```
Vector &product (const Vector &v, const Matrix &m) {
    int vector_size = v.size_;
    // . . .
```

## Friends (cont'd)

- A class may confer friendship *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:

```
#include <iostream.h>
class String {
    friend ostream &operator << (ostream &,
                                String &);
private:
    char *str_;
    // . . .
};
ostream &operator << (ostream &os,
                    String &s) {
    os << s.str_;
    return os;
}
```

## 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 . . .

### Class Vector Example

```
// File Vector.h (correct wrt initialization
template <class T> // and assignment)
class Vector
{
public:
    ~Vector (void);
    Vector (size_t len = 100, const T init);
    size_t size () const;
    T &operator[] (size_t i);
    /* New functions */
    Vector (const Vector<T> &v); // Copy constructor
    // Assignment operator
    Vector<T> &operator= (const Vector<T> &v);
protected:
    T &elem (size_t i);
private:
    size_t size_;
    size_t max_;
    T *buf_;
    bool in_range (size_t i);
};
```

- This class solves previous problems with aliasing and deletion . . .

## Initialization and Termination

- Automatic initialization and termination activities are supported in C++ via constructors and destructors
- Constructors*

```
– Allocate contained data objects upon creation
– Initialize class data members, e.g.,
template <class T>
Vector<T>::Vector (size_t len, const T init)
    : size_ (len), max_ (len)
{
    if (this->size_ == 0) throw Vector<T>::range_error ();
    this->buf_ = new T[this->size_];

    while (--this->size_ >= 0) this->buf_[this->size_] = init;
}
```

## Initialization and Termination (cont'd)

- Destructors*
  - Deallocate data allocated by the constructor
  - Perform any other class-specific tasks associated with object termination

```
template <class T>
Vector<T>::~~Vector (void) {
    delete [] this->buf_;
}
```

## Initialization and Termination (cont'd)

- Without exceptions, handling constructor or destructor failures is very difficult and/or ugly, *e.g.*,
  1. Abort entire program
  2. Set global (or class instance) flag
  3. Return reference parameter (works for constructors, but not destructors)
  4. Log message and continue . . .
- However, exceptions have their own traps and pitfalls . . .

## 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)

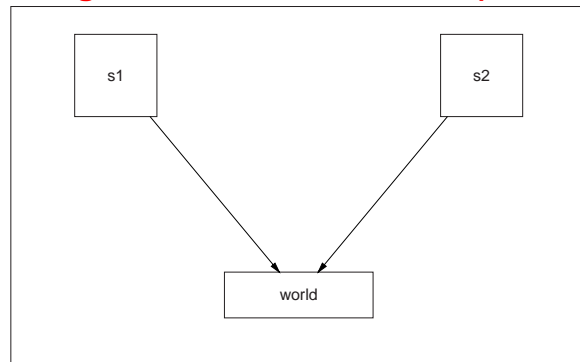
```
class String {
public:
    String (const char *t)
        : len_ (t == 0 ? 0 : ::strlen (t)) {
        if (this->len_ == 0)
            throw range_error ();
        this->str_ = ::strcpy (new char [len_ + 1], t);
    }
    ~String (void) { delete [] this->str_; }
    // . . .
private:
    size_t 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 `s1.s` and `s2.s` point to the dynamically allocated buffer storing `world` (this is known as *aliasing*)

## Assignment and Initialization (cont'd)

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

```

#include "Vector.h"
extern 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.,

```

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; }
  
```

## Assignment and Initialization (cont'd)

- Both 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 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 operator +=, which need have nothing to do with operator =

### Vector Usage Example

```
#include "Vector.h"

int main (int argc, char *argv[]) {
    int size = argc > 1
        ? :atoi (argv[1]) : 10;
    Vector<int> v1 (size); // defaults to 10
    Vector<int> v2 (v1);

    ::srandom (::time (0L));

    for (size_t i = 0;
         i < v1.size ();
         i++)
        v1[i] = v2[i] = ::random ();

    Vector<int> v3 (v1.size (), -1);
    v3 = v1; // Perform a Vector assignment

    for (size_t i = 0; i < v3.size (); i++)
        cout << v3[i];
    return 0;
}
```

## Restricting Assignment and Initialization

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

```
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!

```
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

```
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 . . .

```
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.*,

```
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:

```
extern int divide (double, double);
extern double divide (double, double);
```

## Overloading (cont'd)

- `const` and `non-const` functions are different functions, so constness 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.*,

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

## Overloading (cont'd)

```
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)

- There are two issues to consider when *composing* overloaded operators in expressions, *e.g.*,
- Two issues to
    1. *Memory Management*
      - \* Creation and destruction of temporary variables
      - \* Where is memory for return values allocated?
    2. *Error Handling*
      - \* *e.g.*, what happens if a constructor for a temporary object fails in an expression?
      - \* This requires some type of exception handling

## 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:

```
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:
 

```
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, *e.g.*,

```
Matrix a, b, c, d;
. . .
Matrix t (b);
t.add (c);
t.mult (d);
a = t;
```

- or define and use the short-hand operator x= instead, *e.g.*,  
a = b + c \* d; can be represented by:

```
Matrix a (c);
a *= d; a += b;
```

## 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)

- Here's the Vector class again (this time using a default parameter for the type)

```
template <class T = int>
class Vector
{
public:
    Vector (size_t len): size_ (len),
        buf_ (new T[size_ < 0 ? 1 : size_]) {}
    T &operator[] (size_t i) { return this->buf_[i]; }
    // . . .
private:
    size_t size_; /* Note, this must come first!!! */
    T *buf_;
};
```

## Parameterized Types (cont'd)

- And examples of its usage . . .

```
Vector<> v1 (20); // int by default . . .
Vector<String> v2 (30);
typedef Vector<Complex> Complex_Vector;
Complex_Vector v3 (40);
v1[1] = 20;
v2[3] = "hello";
v3[10] = Complex (1.0, 1.1);
v1[2] = "hello"; // ERROR!
```

## Parameterized Types (cont'd)

```
Vector<int> *foo (size_t size) {
    Vector<double> vd (size); // An array of doubles

    // A dynamically allocated array of chars
    Vector<char> *vc = new Vector<char>(size);

    // "size" arrays of 100 ints
    Vector<int> *vi = new Vector<int>[size];

    delete vc; /* Destructor for vc called */
    return vi; // vi won't be deallocated automatically!
    /* Destructor called for auto variable vd */
}
Vector<int> *va = foo (10); assert (va[1].size () == 100);
delete [] va; /* Call 10 destructors */
```

## Parameterized Types (cont'd)

- Note that we could also use templates to supply the size of a vector at compile-time (more efficient, but less flexible)

```
template <class T = int, size_t SIZE = 100>
class Vector
{
public:
    Vector (void): size_ (SIZE) {}
    T &operator[] (size_t i) { return this->buf_[i]; }
private:
    size_t size_;
    T buf[SIZE];
};
```

- This would be used as follows: `Vector<double, 1000> v;`

## Parameterized Types (cont'd)

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

```
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;
}
```

## Exception Handling Overview

- Exception handling provides a disciplined way of dealing with erroneous run-time events
- When used properly, exception handling makes functions easier to understand because they separate out error code from normal control flow
- C++ exceptions may throw and catch arbitrary C++ objects
  - Therefore, an unlimited amount of information may be passed along with the exception indication
- The *termination* (rather than *resumption*) model of exception handling is used

## Limitations of Exception Handling

- Exception handling may be costly in terms of time/space efficiency and portability
  - *e.g.*, it may be inefficient even if exceptions are not used or not raised during a program's execution
- Exception handling is not appropriate for all forms of error-handling, *e.g.*,
  - If immediate handling or precise context is required
  - If "error" case may occur frequently
    - \* *e.g.*, reaching end of linked list
- Exception handling can be hard to program correctly

## Exception Handling Examples

- Without exceptions:

```
Stack s;
int i;
// . . .
if (!s.is_full ()) s.push (10);
else /* . . . */
// . . .
if (!s.is_empty ()) i = s.pop ();
else /* . . . */
```

## Exception Handling Examples (cont'd)

- Versus

```
Stack s;
int i;
try {
    s.push (10);
    // . . .
    i = s.pop ();
}
catch (Stack::UNDERFLOW &e) { /* . . . */ }
catch (Stack::OVERFLOW &e) { /* . . . */ }
```

## Another C++ Exception Handling Example

- Note the subtle chances for errors . . .

```
class xxii {
public:
    xxii (const String &r): reason_ (r) {}
    String reason_;
};
int g (const String &s) {
    String null ("");
    if (s == null) throw xxii ("null string");
    // destructors are automatically called!
    // . . .
}
```

## Another C++ Exception Handling Example

```
int f (const String &s) {
    try {
        String s1 (s);
        char *s2 = new char[100]; // careful . . .
        // . . .
        g (s1);
        delete [] s2;
        return 1;
    }
    catch (xxii &e) {
        cerr << "g () failed, " << e.reason_; return 22; }
    catch (...) {
        cerr << "unknown error occurred!"; return -1; }
}
```

## 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)

- 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

## Pointer to Function Iterator Example

```
#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

```
#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 (void) { return this->i_++ < this->vr_.size (); }
        T value (void) { 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 Vector\_iterator be declared as a friend of class Vector
  - However, for efficiency reasons this is often necessary in more complex ADTs

## Iterator Style, conventional

```
Vector_iterator<int> i (v);

while (!i.done ())
{
    int *item = 0;
    iterator2.next (item);
    cout << *item << endl;

    iterator2.advance ();
}
```

## Iterator Style, STL

```
#include <vector>
#include <iterator>

vector<int> v;
v.push_back (1); v.push_back (2); v.push_back (3);

vector<int>::iterator i = v.begin ();

while (i != v.end ()) {
    cout << *i++ << endl;
    // Equivalent to: *(i++). The iterator is advanced
    // first. But, the post-increment operation returns
    // the value before advancement. The dereference
    // operation (*) on the iterator returns the item.
}
```

## Miscellaneous ADT Issues in C++

- New (ANSI) casts
- References
- const methods
- static methods
- static data members
- mutable Type Qualifier
- Arrays of class objects

## 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)

- `reinterpret_cast` converts types, possibly in an implementation-dependent manner
  - `long random = reinterpret_cast<long> (&func);`

- `dynamic_cast` casts at run-time, using RTTI

```
void func (Base *bp) {
    Derived *dp = dynamic_cast<Derived *> (bp);
    if (dp)
        // bp is a pointer to a Derived object
}
```

## 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)

- Note, there is a trade-off between indirection vs. copying

```
struct Huge { int size_; int array_[100000]; };
int total (const Huge &h) {
    int count = 0;
    for (int i = 0; i < h.size_; i++)
        count += h.array_[i];
    return count;
}
Huge h;

int main (int, char *[]) {
    // Small parameter passing cost . . .
    int count = total (h);
}
```

## References (cont'd)

- The following behaves like Pascal's VAR parameter passing mechanism (a.k.a. *call-by-reference*):

```
double square (double &x) { return x *= x; }
int bar (void) {
    double foo = 10.0;
    square (foo);
    cout << foo; // prints 100.0
}
```

## References (cont'd)

- In C this would be written using explicit dereferencing:

```
double square (double *x) { return *x *= *x; }
int bar (void) {
    double foo = 10.0;
    square (&foo);
    printf ("%f, foo); /* prints 100.0 */
}
```

- Note, reference variables may lead to subtle aliasing problems when combined with side-effects:

```
cout << (square (foo) * foo);
// output result is not defined!
```

## References (cont'd)

- A function can also return a reference to an object, *i.e.*, an *lvalue*

- Avoids cost of returning by an object by value
  - Allows the function call to be an *lvalue*
- ```
Employee &boss_of (const Employee &);
Employee smith, jones, vacant;
if (boss_of (smith) == jones)
    boss_of (smith) = vacant;
```
- Note, this is often done with `operator[]`, *e.g.*,

```
Vector<int> v (10);
v[3] = 100; // v.operator[] (3) = 100;
int i = v[3]; // int i = v.operator[] (3);
```

## 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)

## 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.*,

```
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
```

## 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.*,

```
class Foo {
public:
    int a_;
private:
    // {\em Must} be defined exactly once outside header!
    // (usually in corresponding .C file)
    static int s_;
};
Foo x, y, z;
```

## Static Data Members (cont'd)

- Note:
  - There are three distinct addresses for `Foo::a`, *i.e.*, `&x.a_`, `&y.a_`, and `&z.a_`.
  - 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

```
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)

- The following calls are legal:

```
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:

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

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

## Mutable Type Qualifier

- The constness of an object's storage is determined by whether the object is constructed as `const`
- An attempt to modify the contents of `const` storage (via casting of pointers or other tricks) results in undefined behavior
  - It is possible (though not encouraged) to “cast-away” the constness of an object. This is not guaranteed to be portable or correct, however!
 

```
const int i = 10;
//. . .
* (int *) &i = 100; // Asking for trouble!
```
- If a data member is declared with the storage class `mutable`, then that member is modifiable even if the containing object is `const`

**class Foo { Mutable Type Qualifier (cont'd)**

```
public:
    Foo (int a, int b): i_ (a), j_ (b) {}
    mutable int i_;
    int j_;
};
const Foo bar;

// the following must be written in a context with
// access rights to Foo::i_ and Foo::j_.

bar.i_ = 5;           // well formed and defined
bar.j_ = 5;           // not well-formed
*(int *)&bar.j_ = 5; // well-formed; undefined behavior!
// better style, but still undefined behavior
if (int *i = const_cast<int *>(&bar.j_)) i = 5;
```

**Mutable Type Qualifier (cont'd)**

- A consequence of mutable is that a object is ROMable if
  1. Its class doesn't have any mutable data members
  2. The compiler can figure out its contents after construction at compile time
  3. The compiler can cope with any side effects of the constructor and destructor
    - or can determine that there aren't any

**Arrays of Objects**

- In order to create an array of objects that have constructors, one constructor must take no arguments
  - Either directly or via default arguments for all formal parameters
  - e.g.,
 

```
Vector<Vector<int> > vector_vector1;
Vector<int> vector_vector2[100];
Vector<int> *vector_vector_ptr = new Vector<int>[size]
```
- The constructor is called for each element
- If array created dynamically via new, then delete must use an empty []
  - This instructs the compiler to call the destructor the correct number of times, e.g., `delete [] vector_vector_ptr;`

**Unions**

- A union is a structure who member objects all begin at offset zero and whose size is sufficient to contain any of its member objects
  - They are often used to save space
- A union of the form `union { member-list— };` is called an *anonymous union*; it defines an unnamed object
  - The union fields are used directly without the usual member access syntax, e.g.,
 

```
void f (void) {
    union { int a_; char *p_; };
    a_ = 1; p_ = "Hello World\n";
    // a_ and p_ have the same address!
    // i.e., &a_ == &p_
}
```

## Unions (cont'd)

- Here's an example that illustrates a typical way of using unions, *e.g.*,

```
struct Types {
    enum Type {int, double, char} type_;
    union { int i_; double d_; char c_; };
} t;
if (t.type_ == Types::double) t.d_ = 100.02;

// Q: ``what is the total size of struct Types?``
// Q: ``What if union were changed to struct?``
```

- Note that C++ provides other language features that makes unions less necessary (compared to C)
  - *e.g.*, inheritance with dynamic binding

## Unions (cont'd)

- Some restrictions apply:
  - *Unions in general*
    - \* A union may not be used as a base class and can have no virtual functions
    - \* An object of a class with a constructor or destructor or a user-defined assignment operator cannot be a member of a union
    - \* A union can have no static data members
  - *Anonymous unions*
    - \* Global anonymous unions must be declared static
    - \* An anonymous union may *not* have private or protected members
    - \* An anonymous union may not have methods

## Summary

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