7028 Programmierung 2

Prof. O. Nierstrasz

Sommersemester 1997

Table of Contents

C++ Programming Rules, Hints and Guidelines

1. P2 — Introduction to C++

Lecturer: Prof. Oscar Nierstrasz
Schützenmattstr. 14/103, Tel. 631.4618

Secretary: Frau I. Huber, Tel. 631.4692

Assistant: Franz Achermann, Manuel Guenter, Stefan Kneubuehl

WWW: http://iamwww.unibe.ch/~scg/Lectures/

Principle Text:

- Stanley B. Lippman, *C++ Primer, Second Edition*, Addison-Wesley, 1991.

Essential C++ Texts

- Magaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.
- Marshall P. Cline and Greg A. Lomow, *C++ FAQs*, Addison-Wesley, 1995.
- Scott Meyers, *Effective C++*, Addison-Wesley, 1992.
- James O. Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.
- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison Wesley, Reading, MA, 1995.
- David R. Musser and Atul Saini, *STL Tutorial and Reference Guide*, Addison-Wesley, 1996.

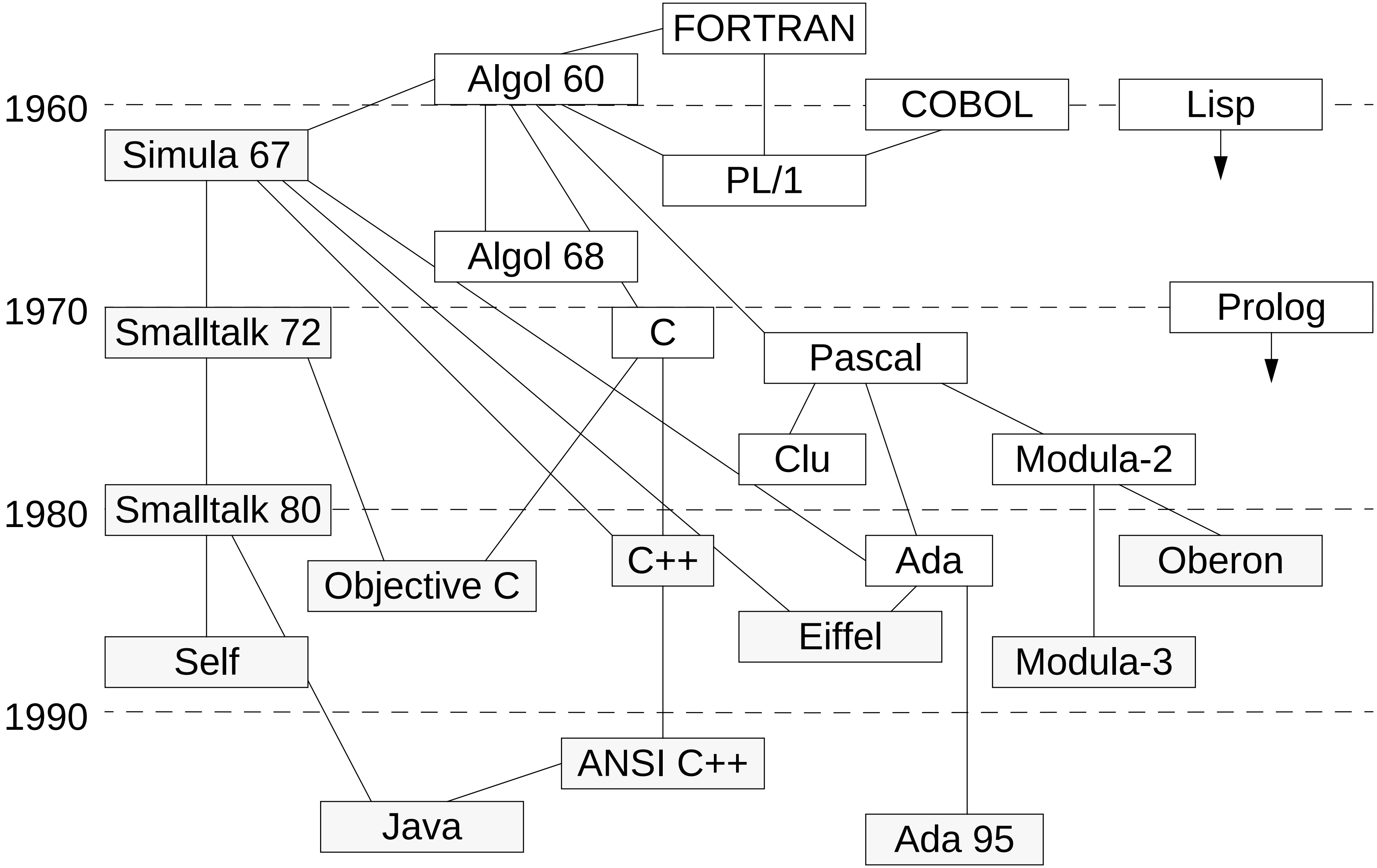
Overview

1.	21.03	Introduction
	28.03	*Good Friday — no lecture*
2.	04.04	A Taste of C++ — Comparison with Eiffel
3.	11.04	C++ Basic Language Features
4.	18.04	Decomposition and Recursion
5.	25.04	Specifying Classes
6.	02.05	Data Abstraction
7.	09.05	Managing Memory
8.	16.05	Inheritance
9.	23.05	Tools: makefiles, version control, debuggers, browsers ...
10.	30.05	C++ Design Rules
11.	06.06	Templates and other advanced features
12.	13.06	The Standard Template Library
13.	20.06	Two STL Examples
	27.06	*Final Exam*

What You Will Be Expected To Learn

- How to implement abstract data types with C++ classes
- How to use assertions and exceptions to develop correct programs
- How to use the C++ type system effectively
- How to use inheritance to support polymorphism and code reuse
- How to manage memory effectively
- How to organize C++ programs into source and header files
- How to use makefiles, debuggers and other basic tools

History


FORTRAN
Algol 60
1960
COBOL
Lisp
Simula 67
PL/1
Algol 68
1970
Prolog
Smalltalk 72
C
Pascal
Clu
Modula-2
1980
Smalltalk 80
C++
Ada
Oberon
Objective C
Eiffel
Self
Modula-3
1990
ANSI C++
Java
Ada 95

C++ Design Goals

"C with Classes" designed by Bjarne Stroustrup in early 1980s; grew into C++

- ❑ Originally a translator to C
  - ☞ Difficult to debug and potentially inefficient
- ❑ Mostly upward compatible extension of C
  - ☞ "As close to C as possible, but no closer"
  - ☞ Stronger type-checking
  - ☞ Support for data abstraction
  - ☞ Support for object-oriented programming
- ❑ Run-time efficiency
  - ☞ Language primitives close to machine instructions
  - ☞ Minimal cost for new features

Conflicts:

- ☞ Modern compiler optimization techniques are hard to apply because of low-level features (e.g.. arbitrary memory pointers)
- ☞ Software engineering principles require rigid discipline due to availability of inherited C features

C Features

C was developed in 1972 by Dennis Ritchie and Brian Kernighan as a systems language for Unix on the PDP-11. A successor to B [Thompson, 1970], in turn derived from BCPL.

C was designed as a general-purpose language with a very direct mapping from data types and operators to machine instructions. C can be seen as a "high-level assembler."

- *C preprocessor:* file inclusion, conditional compilation, macros
- *Data types:* char, short, int, long, double, float
- *Type constructors:* pointer, array, struct, union
- *Basic operators:* arithmetic, pointer manipulation, bit manipulation ...
- *Control abstractions:* if/else, while/for loops, switch, goto ...
- *Functions:* call-by-value, side-effects through pointers
- *Type operations:* typedef, sizeof, explicit type-casting and coercion

Prime advantage: programmers have direct control over the execution cost of programs

Prime disadvantages: few opportunities for optimization; hard to debug

C++ Features

C++ is an evolving language ...

C with Classes

- Classes as structs
- Inheritance; virtual functions
- Inline functions

C++ 1.0 (1985)

- Strong typing; function prototypes
- `new` and `delete` operators

C++ 2.0

- Local classes; protected members
- Multiple inheritance

C++ 3.0

- Templates
- Exception handling

ANSI C++

- Proposed standard

“Hello World”

Pre-processor directive: look in the system include directory for the header file “iostream.h” declaring the interfaces to the standard I/O library.

A comment: may also be written:

```
/*
   My first C++ program!
*/
```

```
#include <iostream.h>
// My first C++ program!
void main(void)
{
   cout << ”Hello world!” << endl;
}
```

Function definition: there is always a “main” function

String constant: an array of 13 chars (not 12!)

Global variable: `cout` is the “standard output stream”

Operator overloading: two *different* << operators are disambiguated by their argument types!

C++ Storage Classes

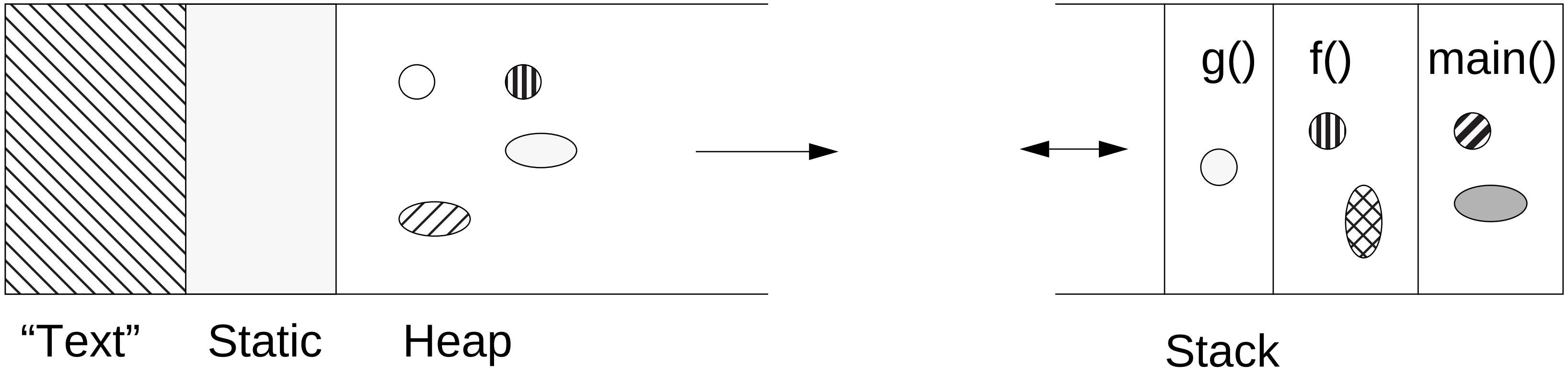
C++ requires that you explicitly manage storage space for objects

1. Static
   - ☞ static objects exist for the entire life-time of the process
   - ☞ scope may be local, global or class-specific

2. Automatic
   - ☞ only live during function invocation on the “run-time stack”

3. Dynamic
   - ☞ dynamic objects live between calls to `new` and `delete` (or `malloc` and `free`)
   - ☞ their lifetimes typically extend beyond their scope

Memory Layout



The address space available to a running process consists of (at least) four conceptually different parts:

1. Text: the executable program text (not writable)
2. Static: static global data
3. Heap: dynamically allocated global memory (grows upward)
4. Stack: local memory (stack frames) for function calls (grows downward)

The total number of memory pages available to a process varies at run-time according to need (i.e., function calls and requests to increase the heap).

Stack memory is automatically reclaimed when a function call returns; heap memory must be explicitly managed by the program!

Declarations and Definitions

- A ***declaration*** of a variable (or function) announces that the variable (function) exists and is defined somewhere else.
- A ***definition*** of a variable (function) causes storage to be allocated

☞ In C++ a variable *must* be declared or defined before it is used

C++ does not support an explicit *module* concept — instead one may break a program into separate *source* and *header* files. The source files typically contain definitions that may be separately compiled. The header files contain declarations that allow other parts of the program to know about and use the variables and functions exported by a given "module."

```
extern int size;            // declaration
void hello(void);           // declaration (function prototype)

int size;                   // definition
void hello(void) {          // definition
   cout << "hello!" << endl;
}
```

Hello World Project

prog.cpp contains main program:

```
#include "hello.h"          // needed to declare hello()

int main(void)
{
   hello();
   return 0;
}
```

hello.h declares all functions defined and exported from hello.cpp:

```
void hello(void);           //
```

hello.cpp contains definitions of library functions:

```
#include <iostream.h>       // needed to declare cout and endl
#include "hello.h"          // needed to declare functions defined here

void hello (void)
{
   cout << "hello world" << endl;
}
```

Compiling C++ Programs

Single file compilation:

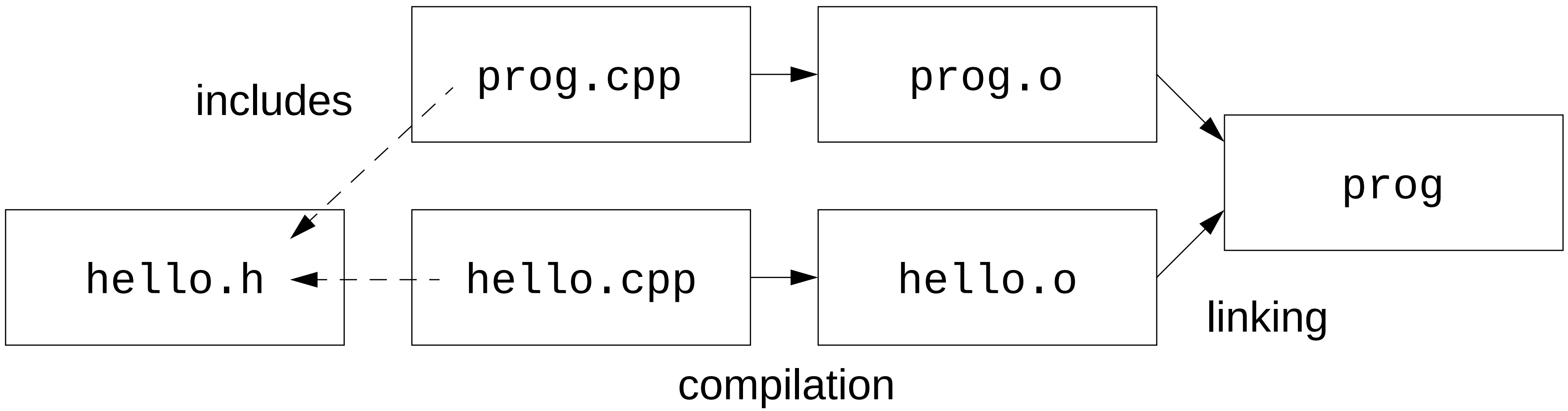
- ☞ CC prog.cpp — generates executable *a.out*
- ☞ CC -o prog prog.cpp — generates executable *prog*

Multi-file compilation:

- ☞ CC -o prog prog.cpp hello.cpp

Library pre-compilation:

- ☞ CC -c hello.cpp — generates object code *hello.o*
- ☞ CC -o prog prog.cpp hello.o — compiles *prog.cpp* and links *hello.o*



Header files contain declarations needed to link separately compiled “modules”

Basic Makefile

A Basic Makefile consists of comments, macros, dependency lines and commands:

```
# Version of the C++ compiler; link and compile options:

CXX         = CC
LFLAGS      = -L/opt/SUNWspro/SC3.0.1/lib
CFLAGS      = -O

# Object files needed to create prog:

PROGO = prog.o hello.o

# prog is made by linking together the object files:

prog : ${PROGO}
	${CXX} ${LFLAGS} ${PROGO} -o prog

# prog.o and hello.o each depend on a source file and a header file:

prog.o : prog.cpp hello.h
	${CXX} ${CFLAGS} -c prog.cpp

hello.o : hello.cpp hello.h
	${CXX} ${CFLAGS} -c hello.cpp

clean :
	rm -rf *.o
```

Summary

You should know the answers to these questions:

- ❑ What were the design goals of C++?
- ❑ What improvements did C++ introduce to C?
- ❑ What is an “include file”?
- ❑ What is the structure of a C++ program?
- ❑ What kinds of storage classes exist in C++, and what are they for?
- ❑ What is meant by “separate compilation”?

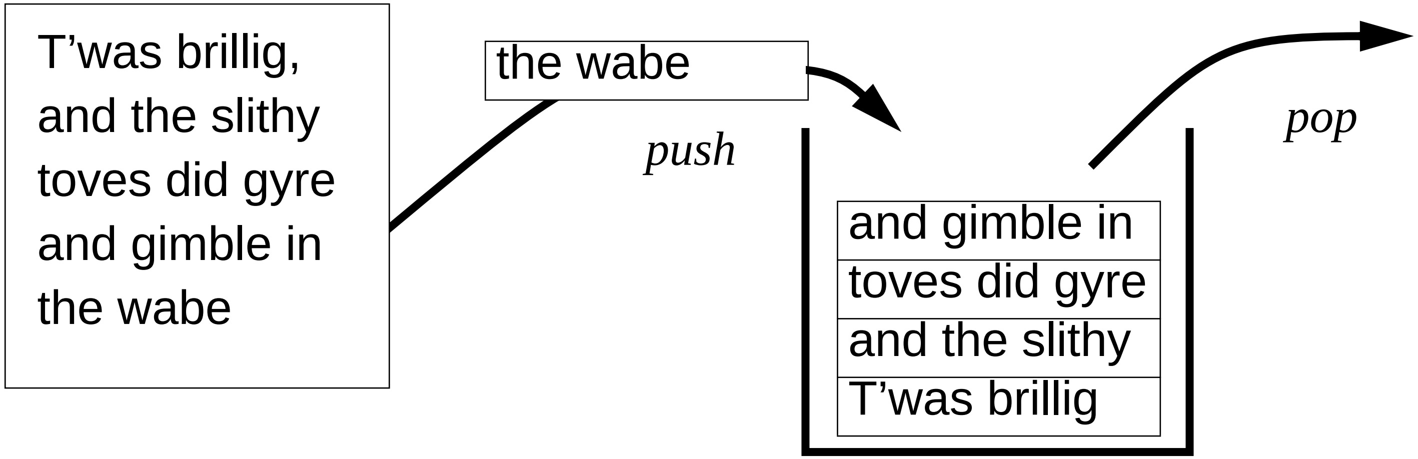
Can you answer the following questions?

- ✎ *When is C++ a good (resp. bad) choice to program in?*
- ✎ *What is meant by “overloaded operators”?*
- ✎ *Why does C++ require functions to be declared before they are used?*
- ✎ *What are the dangers of using* `new` *and* `delete`*?*
- ✎ *What are positive and negative aspects of separate compilation?*

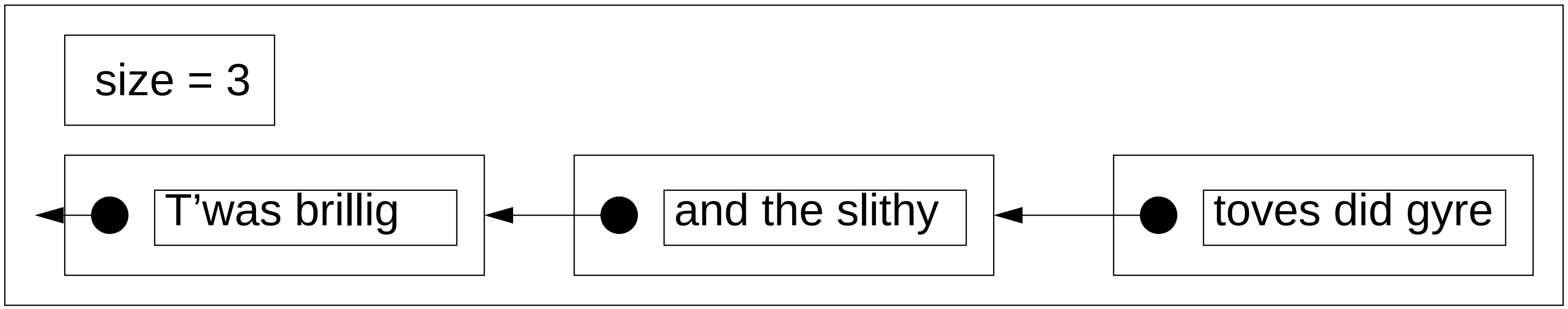
2. A Taste of C++ — Comparison with Eiffel

- ❑ Example: reversing lines of a file
- ❑ Implementation in Eiffel using a dynamic stack
- ❑ Equivalent implementation in C++
  - ☞ Differences between Eiffel and C++
- ❑ Software reuse with templates
- ❑ C implementation (without data abstraction)
- ❑ Recursive implementation (functional paradigm)
- ❑ Perl implementation (specialized language)
- ❑ Timing differences

Data Abstraction — Line Reverser Example



We can implement our Stack
abstraction as a linked list of Strings.



Eiffel Line Reverser

```
-- File: erev.e
--
-- Reverses the order of lines in the input
-- using a dynamic stack.
-- The stack is implemented as a linked list.

class EREV

creation { ANY }
    make

feature { NONE }

    ioStack : DYNAMICSTACK [STRING]

    make is
        do
            !!ioStack.make

            -- Push input lines onto stack
            from
                io.readline
            until
                 io.input.end_of_file
            loop
                io.readline
                ioStack.push(io.last_string)
                io.readline
            end

            -- Pop them off in reverse order
            -- and print them all
            from
            until  ioStack.empty
            loop
                io.putstring(ioStack.top)
                io.new_line
                ioStack.pop
            end
        end -- make
end -- class EREV
```

An Eiffel Stack Implementation

```
-- File: dynamicStack.e
class DYNAMICSTACK [T]

creation { ANY }
    make

feature { NONE }
    topCell : MYCELL [T]
    size : INTEGER

    make is
        do
            size := 0 ; !!topCell.make
        end -- make

feature { ANY }
    count : INTEGER is
        do
            Result := size
        end -- count

    empty : BOOLEAN is
        do
            Result := (size = 0)
        end -- empty

    top : T is
        require
            not empty
        do
            Result := topCell.value
        end -- top

    push (x : T) is
        local
            newCell : MYCELL [T]
        do
            size := size + 1
            !!newCell.make
            newCell.setValue(deep_clone (x))
            newCell.setNext(topCell)
            topCell := newCell
        ensure
            not empty
            deep_equal(top,x)
            size = old size + 1
        end -- push

    pop is
        require
            not empty
        do
            -- don't deference if empty!
            topCell := topCell.next
            size := size - 1
        ensure
            size = old size - 1
        end -- pop

    invariant
        0 <= size

end -- class DYNAMICSTACK
```

The (Hidden) Eiffel Stack Cells

```
-- File: mycell.e

class MYCELL [T]
-- structure for implementation of linked lists
-- NB: an exception will be raised if next is dereferenced
-- without first being set

creation { DYNAMICSTACK }
   make

feature { NONE }

   make is
      do
      end  -- make

feature { DYNAMICSTACK }

   value : T

   next : like Current

   setValue (v : T) is
      do
         value := v
      end -- setValue

   setNext (n : like Current) is
      do
         next := n
      end -- setNext

end -- class MYCELL
```

A C++ Line Reverser

```
// File: cpprev.cpp
//
// Reverses the order of lines in the input
// using a dynamic stack.
// The stack is implemented as a linked list.

#include <iostream.h>
#include <exception.h>
#include "dstack.h"

const int bufSize = 256;
```

```
int main(void)
{
   DStack ioStack;
   char * buf;

   try {
      buf = new char[bufSize];

      // Push input lines onto stack
      while (!cin.getline(buf, bufSize).eof()) {
         ioStack.push(buf);
         buf = new char[bufSize];
      }

      // Pop them off in reverse order
      // and print them all
      while (ioStack.count() != 0) {
         cout << ioStack.top() << endl;;
         delete [] ioStack.top();
         ioStack.pop();
      }
   }
   catch (xmsg &err) {
      cout << "Exception: "
         << err.why() << endl;
      return -1;
   }
   return 0;
}
```

A C++ Stack Interface

```
// File: dstack.h
//
// An absolutely minimal stack interface
// using linked lists.

#ifndef DSTACK_H
#define DSTACK_H

#include <exception.h>

typedef char* Item; // Redefine as necessary

class DStack
{
public:
    DStack(void);
    ~DStack(void);

    // inline functions:
    int count(void) { return size; };
    int empty(void) { return size == 0; };

    // NB: pop() does not return a value
    // use top() before pop() to retrieve
    // the value

    void push(Item item) throw();
    Item top(void) throw(xmsg);
    void pop(void) throw(xmsg);

private:
    // NB: The Cell interface is only
    // visible within DStack.
    class Cell
    {
    public:
        Item value;
        Cell *next;
    };

    Cell *topCell;
    int size;
};

#endif
```

A C++ Stack Implementation

```
// File: dstack.cpp
//
// An absolutely minimal stack implementation
// using linked lists.

#include "dstack.h"

// constructor for an empty stack:
DStack::DStack (void)
   : size(0), topCell(0)
{
}

// destructor pops all cells:
DStack::~DStack (void)
{
   while (!this->empty()) {
      this->pop();
   }
}

// this is the only way to get values
// from the stack:
Item
DStack::top (void) throw(xmsg)
{
   if (this->empty()) {
      throw(xmsg("Empty stack has no top!"));
   }
   return this->topCell->value;
}
```

```
// push makes a new top cell holding the new
// value and pointing to the existing cells:
void
DStack::push (Item item) throw()
{
   Cell *newCell;
   newCell = new Cell;
   newCell->value = item;
   newCell->next = this->topCell;
   this->topCell = newCell;
   size++;
}

// deallocates the top cell and resets the top:
void
DStack::pop (void) throw(xmsg)
{
   if (this->empty()) {
      throw(xmsg("Can't pop an empty stack!"));
   }
   Cell *oldTop = topCell;
   topCell = topCell->next;
   delete oldTop;
   size--;
}
```

Differences Between Eiffel and C++

Eiffel	*C++*
"Pure" object-oriented language	Hybrid language (global variables ...)
Uniform type system	Baroque type specifications Header files; declaration vs. definition Explicit type casting
Generic types; "like current" type	Templates (purely syntactic)
Feature visibility	Public/private/protected declarations; "friends"; nested classes
Automatic garbage collection Only object creation is specified	Explicit delete operator, destructors Can implement own memory management
Safe object identifiers	Object/pointer distinction; pointer arithmetic
Assertions to support "design by contract"	Exception handling; exception values
Automatic inlining	Explicit inlining, virtual declarations

Myth: *C++ is inherently more "efficient" than Eiffel.*

Fact: *C++ gives the programmer more control than Eiffel.*

A C++ Template Line Reverser

```
// File: rwrev.cpp
//
// Rogue Wave template implementation of line reverser.

#include <iostream.h>
#include <exception.h>

#include <rw/cstring.h>
#include <rw/tstack.h>
#include <rw/tvdlist.h>

typedef RWTStack<RWCString, RWTValDlist<RWCString> > IOStack;

int main (void)
{
   RWCString buf;
   IOStack ioStack;

   // Push input lines onto stack
   while (buf.readLine(cin, FALSE)) // don't ignore white space!
   {
      ioStack.push(buf);
   }

   // Pop them off in reverse order and print them all
   while (ioStack.entries() != 0)
   {
      cout << ioStack.pop() << endl;;
   }
   return 0;
}
```

A C Line Reverser

```
/*
   File: crev.c
   A C implementation of the line reverser program.
*/

#include <stdlib.h>
#include <stdio.h>

int main (void)
{
   const int bufSize = 256, stackSize = 32000;
   char *buf, **stack;
   int top=0;

   stack = (char**) malloc(sizeof(char*) * stackSize);
   buf = (char*) malloc(sizeof(char) * bufSize);
   stack[top] = buf;

   while (fgets(buf, bufSize, stdin) != NULL) {
      if (top>stackSize) {
         fprintf(stderr, "frev: buffer overflow!!!\n");
         exit(-1);
      }
      buf = (char*) malloc(sizeof(char) * bufSize);
      stack[++top] = buf;
   }

   /*
      don't use last allocated
      top since not null-terminated
   */
   free(stack[top--]);
   while (top>=0) {
      printf("%s", stack[top]);
      free(stack[top--]);
   }
   return 0;
}
```

A Recursive Line Reverser

```
// File: hrev.cpp
// A hybrid (recursive) line reverser.

#include <iostream.h>
#include <rw/cstring.h>

void recrev(void);

int main (void)
{
    recrev();
    return 0;
}

void recrev()
{
    RWCString buf;

    if (buf.readLine(cin, FALSE)) {          // read a line
        recrev();                            // reverse the rest of the input
        cout << buf << endl;                 // now output this line
    }
}
```

A Perl Line Reverser

```
#! /usr/local/bin/perl
#
# File: prev
#
# A Perl line reverser

while (<>) {                        # or simply: @file = <>;
    push(@file, $_);
}

while ($#file>=0) {
    print pop(@file);
}
__END__
```

Some Timing Differences

Input file: 20960 lines, 366167 characters

	Real Time	*User Time*	*System Time*
rwrev (RW Templates)	4.6	3.3	1.1
erev (Eiffel)	4.6	3.2	1.1
hrev (C++)	4.3	2.9	1.2
cpprev (C++)	3.2	1.6	1.4
prev (Perl)	2.0	1.2	0.5
crev (C)	1.9	0.9	0.7

What are the reasons for the differences in execution speed?
(Probably not what you think!)

Summary

You should know the answers to these questions:

- ❑ What are the essential differences between Eiffel and C++?
- ❑ What is a “function prototype”?
- ❑ What is the difference between a declaration and a definition?
- ❑ What is a header file for?
- ❑ What is a “destructor” and why do we need them?

Can you answer the following questions?

- ✎ *What does it mean to allocate objects “on the stack” or “on the heap”?*
- ✎ *When is an object paradigm better than a procedural or functional paradigm?*
- ✎ *What are the tradeoffs between programmer productivity and program performance?*

3. C++ Basic Language Features

C++ is a complex and evolving language.
This lecture gives an overview of the basic language features.

- Symbols and Keywords
- Comments and commenting conventions
- Built-in data types
- Expressions and operator precedence
- Arrays, pointers, references and strings
- Assignment — lvalues and rvalues
- Statements and control flow
- Enumeration types
- “Functions” (i.e., procedures)

Not covered yet:

☞ classes, inheritance, exceptions, templates, overloading ...

Symbols

C++ programs are built up from *symbols:*

- **Names:** `main`, `IOStack`, `_store`, `x10`
  { alphabetic or underscore } *followed by*
  { alphanumerics or underscores }
- **Keywords:** **`const`**, **`int`**, **`if`**, **`throw`**
- **Constants:** `"hello world"`, `'a'`, `10`, `077`, `0x1F`, `1.23e10`
- **Operators:** `+`, `>>`, `::`, `*`, `&`
- **Punctuation:** `{`, `}`, `,`

Keywords

<u>asm</u>	**continue**	*float*	<u>new</u>	*signed*	**try**
auto	**default**	**for**	*operator*	<u>sizeof</u>	*typedef*
break	<u>delete</u>	*friend*	*private*	*static*	*union*
case	**do**	**goto**	*protected*	*struct*	*unsigned*
catch	*double*	**if**	*public*	**switch**	*virtual*
char	**else**	*inline*	*register*	*template*	*void*
class	*enum*	*int*	**return**	<u>this</u>	*volatile*
const	*extern*	*long*	*short*	**throw**	**while**

C++ has a large number of keywords, including all those inherited from C.
Italic keywords are use in type declarations. Keywords in **bold** affect control flow.
<u>Underlined</u> keywords are used in statements and expressions.

Comments

Two styles:

```
/*
 * C-style comment pairs are generally used
 * for longer comments that span several lines.
 */


// C++ comments are useful for short comments to end-of-line
```

Be careful! Comment pairs do not nest!!!

```
/* Don't need these variables for now:
   int opt;      /* keep track of the current options */
   char *optDesc;/* a description of the current option */
*/
```

Only the first of the two variables has been commented out!

✔ *Use // comments exclusively within functions so that any part can be commented out using comment pairs.*

Commenting Conventions

Use comments for:

1. each source file stating, e.g., file name, purpose, author, manual references, hints for maintenance, etc.
2. classes and templates
3. every non-trivial function stating its purpose, algorithm used (unless this is obvious), and any assumptions about its environment
4. global variables
5. any non-obvious or non-portable code
6. little else

✔ *Use meaningful names to make your code as self-documenting as possible.*

✔ *DON'T use comments to restate what is obvious from the source code.*

✔ *DO use comments to improve the readability of your programs.*

[Stroustrup, C++ 2nd edn., p. 105]

Built-In Data Types

Data type	*No. of bits*	*Minimal value*	*Maximal value*
signed char	8	-128	127
signed short	16	-32768	32767
signed int	16 / 32	-32768 / -2147483648	32767 / 214748647
signed long	32	-2147483648	214748647
unsigned char	8	0	255
unsigned short	16	0	65535
unsigned int	16 / 32	0	65535 / 4294967295
unsigned long	32	0	4294967295

Data type	*No. of bytes*	*Min. exponent*	*Max. exponent*	*Decimal accuracy*
float	4	-38	+38	6
double	8	-308	+308	15
long double	8 / 10	-308 / -4932	+308 / 4932	15 / 19

Expressions

```
int a, b, c;
double d;
float f;

a = b = c = 7;           // assignment:          a == 7; b == 7; c == 7
a = (b == 7);            // equality test:       a == 1 (7 == 7)
b = !a;                  // negation:            b == 0 (!1)
a = (b>=0) && (c<10);    // logical AND:         a == 1 ((0>=0)&&(7<10))
a *= (b += c++);         // increment:           a == 7; b == 7; c == 8

a = 11 / 4;              // integer division:    a == 2
b = 11 % 4;              // remainder:           b == 3

d = 11 / 4;              //                      d == 2.0 (not 2.75!)
f = 11.0 / 4.0;          //                      f == 2.75

a = b|c;                 // bitwise OR:          a == 11 (03|010)
b = a^c;                 // bitwise XOR:         b == 3 (013^010)
c = a&b;                 // bitwise AND:         c == 3 (013&03)
b = a<<c;                // left shift:          b == 88 (11<<3)

a = (b++,c--);           // comma operator:      a == 3; b == 89; c == 2
b = (a>c)?a:c;           // conditional operator:b == 3 ((3>2)?3:2)
```

✔ *Avoid cryptic expressions! Use comments to explain mysterious code.*

Operator Precedence and Associativity

Level	*Operator*	*Function*
17R 17L	`::` `::`	global scope (unary) class scope (binary)

16L	`->, .`
	`[]`
	`()`
	`()`
		member selectors
		array index
		function call
		type construction

15R	`sizeof`
	`++, --`
	`~`
	`!`
	`+, -`
	`*, &`
	`()`
	`new, delete`
		size in bytes
		increment, decrement
		bitwise NOT
		logical NOT
		unary plus, minus
		dereference, address-of
		type conversion (cast)
		free store management

14L	`->*, .*`	member pointer selectors

<table>
<tr><th>Level</th><th>Operator</th><th>Function</th></tr>
<tr><td>13L</td><td><code>*, /, %</code></td><td>times, divide, remainder</td></tr>
<tr><td>12L</td><td><code>+, -</code></td><td>add, subtract</td></tr>
<tr><td>11L</td><td><code><<, >></code></td><td>bitwise shift left/right</td></tr>
<tr><td>10L</td><td><code><, <=, >, >=</code></td><td>comparisons</td></tr>
<tr><td>9L</td><td><code>==, !=</code></td><td>equality, inequality</td></tr>
<tr><td>8L</td><td><code>&</code></td><td>bitwise AND</td></tr>
<tr><td>7L</td><td><code>^</code></td><td>bitwise XOR</td></tr>
<tr><td>6L</td><td><code>|</code></td><td>bitwise OR</td></tr>
<tr><td>5L</td><td><code>&&</code></td><td>logical AND</td></tr>
<tr><td>4L</td><td><code>||</code></td><td>logical OR</td></tr>
<tr><td>3L</td><td><code>?:</code></td><td>arithmetic if (ternary)</td></tr>
<tr><td>2R</td><td><code>=, *=, /=, %=,
+=, -=, <<=, >>=,
&=, |=, ^=</code></td><td>assignment operators</td></tr>
<tr><td>1L</td><td><code>,</code></td><td>comma operator (eval left to right)</td></tr>
</table>

C++ Arrays

Arrays are fixed sequences of homogeneous elements

- `Type a[n];` defines a one-dimensional array a in a contiguous block of `(n*sizeof(Type))` bytes
- n must be a compile-time constant
- Arrays bounds run from `0` to `n-1`
- Size cannot vary at run-time
- No range-checking is performed at run-time:

```
{
   int a[10];
   for (int i=0; i<=10; i++)
      a[i] = 0; // disaster! a[10] is not part of the array!
}
```

- Can be initialized at compile time:

```
int eightPrimes[8] = { 2, 3, 5, 7, 11, 13, 17, 19 };
int idMatrix[2][2] = { { 1, 0 }, { 0, 1 } };
```

Pointers

A *pointer* is a variable that can hold the address of another variable:

```
int i = 10;
int *ip = &i;
```

- Pointers can be used to indirectly access and update variables:

```
*ip = *ip + 1;              // increment i
```

- Array variables are treated as pointers to their first element

```
int *ep = eightPrimes;
```

- Pointers can be treated like arrays:

```
ep[7] = 23;                 // update 8th element of eightPrimes[]
```

- But have differents sizes:

```
sizeof(eightPrimes) == 32   // 8 * 4 bytes
sizeof(ep) == 4             // pointer is 4 bytes
```

- new and `delete` respectively return and operate on object pointers
- A pointer to an unknown data type can be declared `void*`:

```
void *vp = ep;
```

- But must be *typecast* to the appropriate type before it is used:

```
((int*)vp)[7] = 29;         // update 8th element of eightPrimes[]
```

References

A *reference* is an alias for another variable:

```
int i = 10;
int &ir = i;
ir = ir + 1;            // increment i
```

- ❑ Once initialized, references cannot be changed
- ❑ References are most useful in procedure calls to avoid the overhead of passing arguments by value, without the clutter of explicit pointer dereferencing

```
void refInc(int &n)     // compare with Pascal's "var" declaration
{
   n = n+1;             // increment the variable that n refers to
}
```

✔ *References should generally be preferred to pointers except when:*

- ☞ using arrays
- ☞ manipulating dynamically allocated objects (i.e., using `new`)
- ☞ a variable must range over a *set* of objects

Strings

A string is a pointer to a NULL-terminated (i.e., ‘\0’) character array:

```
char *cp;                  // uninitialized string (pointer to a char)
char *hi = "hello";        // initialized string
char hello[6] = "hello";   // initialized char array

cp = hello;                // cp now points to hello[]
cp[1] = 'u';               // cp and hello now point to “hullo”
cp[4] = NULL;              // cp and hello now point to “hull”


sizeof(cp)     == 4        // a char pointer
strlen(cp)     == 4        // four characters in string “hull”

sizeof(hi)     == 4        // another char pointer
strlen(hi)     == 5        // five characters in string “hello”

sizeof(hello) == 6         // array of six chars
strlen(hello) == 4         // four characters in string “hull”
```

Various standard string manipulation routines (including strlen() and strcpy()) are declared in the header file <string.h> (usually in the directory /usr/include)

✔ *It is generally better to use a C++ string class instead of built-in char arrays!*

Assignment — lvalues and rvalues

An assignment expression is valid only if the left hand side is a modifiable lvalue:

```
lvalue = rvalue
```

- “An *‘object’* is a region of storage”
- “An *lvalue* is an expression referring to an ‘object’ or function”
  - e.g., variable names, *ptr, array[n]
- “An lvalue is *modifiable* if it is not a function name, an array name or `const`”

```
int x, y[10];
x = x + 1;      // ok -- x is a variable name
x+1 = x;        // not ok -- x+1 does not refer to storage
*(y+1) = x;     // ok -- same as: y[1] = x;
```

Statements

Expressions and Blocks:

```
{ int a=7; a++; }          // a block is a statement with its own scope
```

Iteration:

```
for (i=0; i<n; i++) { ... }// init, control and update are any expressions
while (notDone) { ... }   // can also break out of or continue loop
do { ... } while (notDone);// loop executed at least once
```

Conditional:

```
if (a>b) { ... }           // NB: any int can be used as a boolean
else { ... }               // else part is optional
```

Multi-case statement:

```
switch (i) {               // integer or expression that may be cast to int
case 0:  x = 0;            // constant expression to compare to
         break;            // break to end of block (else fall through)
case 1:                    // can group cases together
case 2:  x = 1;
         y = 2;
         break;
default: x = -1;           // at most one of these
}
```

Enumeration Types

An enumeration type declares a set of symbolic integer constants:

```
enum Colour { red, green, blue };       // red == 0; green == 1; blue == 2
```

An instance of an enumeration type can (normally) only be set or tested:

```
Colour c;
c = red;                                 // ok; but not: c = 0 etc.
cout << “colour “ << int(c) << “ is “;   // can convert to int if necessary
switch (c) {
case red :
   cout << “red” << endl;
   break;
case green :
   cout << “green” << endl;
   break;
case blue :
   cout << “blue” << endl;
   break;
default :                                // should never happen!
   cout << “unknown colour!” << endl;
   break;
}
```

Functions

Functions must be either declared or defined before they are used

```
int fact (int n);          // declaration only; parameter name n is optional


int fact (int n)           // definition
{
   if (n==0)
      return 1;            // obligatory if return type is not void
   else
      return n*fact(n-1); // NB: return value may be an expression
}
```

To be covered later:

- ☞ optional and default arguments
- ☞ overloading
- ☞ scope resolution
- ☞ static variables

Summary

You should know the answers to these questions:

- What are the built-in data types of C++?
- What does operator << do? In which contexts?
- Why do operators have different levels of precedence?
- What happens when you assign an array to a pointer variable? Vice versa?
- What type of value does `new` return?
- What is the difference between a reference and a pointer?
- What is the difference between an lvalue and an rvalue?

Can you answer the following questions?

- *Why was the language called C++ and not ++C?*
- *What does this statement do?:* `for(i=0; n >= 1<<i; i++);`
- *Can you assign the value of one array variable to another?*
- *Why does C++ have both references and pointers?*
- *Why do C++ strings have to end with a NULL character?*

4. Decomposition and Recursion

- Divide and Conquer: principle of recursion
- Documenting assumptions: assertions, invariants and exceptions
- Iteration vs. Recursion
- Binary search
- Tail recursion and iteration
- Merge sort
- A faster merge sort

Document Assumptions

- ✔ *Use descriptive names for variables; use short names only when their purpose is obvious from the context.*
- ✔ *Always state explicitly all pre- and post-conditions.*
- ✔ *Document all assumptions.*

```
// Requires:   s1[] holds NULL-terminated string;
//             s2[] is long enough to hold a copy of s1[]
// Ensures:    s2[] will hold a copy of s1[]
void strCopy(char s1[], char s2[])
{
   int i = 0;
   while (s1[i] != '\0') {   // Assume s1 is NULL-terminated!
      s2[i] = s1[i];         // Blithely assume s2 is big enough!
      i++;
   }
   s2[i] = '\0';
}
```

- ✔ *Avoid making assumptions that you can't check!*

Comment Selectively

- ✔ *Avoid complex or cryptic code; write code that is self-documenting.*
- ✔ *Use comments to explain any code that is not self-documenting.*

```
void strCopy2(char *s1, char *s2)
{
   while (*s2++ = *s1++); // copy string s1 to buffer s2 up to NULL character
}                         // assumes s2 is big enough!
```

It is easier to demonstrate that a *readable* program is correct than an unreadable one.

Although readability sometimes interferes with efficiency, it is clearly better to have a slow program that works correctly, than an fast program that is wrong!

- ✔ *Ensure your programs are correct before you try to optimize them.*
- ✔ *Never try to optimize code that is not a proven source of system inefficiency.*

Divide and Conquer

Recursion is a powerful technique for designing and implementing algorithms in a declarative, decompositional fashion.

- Determine how a complex instance of the problem can be solved by combining the solution to one or more simpler instances.
- Determine how the simplest (base) cases can be solved directly.
- Ensure that complex cases always reduce to simpler cases. (Otherwise the recursion may not terminate!)
- Implement the general solution by implementing the base cases directly, and the complex cases by recursion.

Recursion

Problem: find the minimum element of an array of integers.

```
// Requires:  num[] an array with length > 0
// Ensures:   result is smallest element of num[]

int findMin(int num[], int length)
{
   if (length <= 0) {
      throw(xmsg("findMin() called with empty array!"));
   } else if (length == 1) {
      // base case -- the only element is the smallest one:
      return num[0];
   } else {                             // now we know length >= 2
      int l1 = length/2;                // so l1 >= 1 but l1 < length
      int l2 = length - l1;             // and the same holds for l2
      int m1 = findMin(num, l1);        // call findMin() recursively
      int m2 = findMin(num+l1, l2);
      return (m1<m2) ? m1 : m2;         // result is min of m1 and m2
   }
}
```

✔ *If possible, check your assumptions, and raise exceptions when they are violated.*

Recursion — Pros and Cons

Pros:

- ❑ Recursive functions are *easy to develop* top-down,
- ❑ they are usually *easy to prove correct*, and
- ❑ they are often much *simpler* than equivalent iterative algorithms.

Cons:

- ❑ One must be *careful about base cases*.
- ❑ Recursion is typically *slower* than iteration (due to function call overhead).
- ❑ Recursive functions can *exhaust stack space* (if recursion is deep).
- ❑ Not all problems are inherently recursive.

✔ *If a problem is inherently recursive, implement a correct recursive solution before deciding whether a non-recursive solution is better.*

Iteration vs. Recursion

Sometimes iteration is more natural than recursion. Always adopt the simplest solution.

```
// Requires: num[] a non-empty array with size length > 0
// Ensures: result is min element of num[]
int findMin2(int num[], int length)
{
   if (length <= 0) {
      throw(xmsg("findMin() called with empty array!"));
   }
   int min = num[0];
   int i;
   for (i=1; i<length; i++) {
      min = (min < num[i]) ? min : num[i];
   }
   return min;
}
```

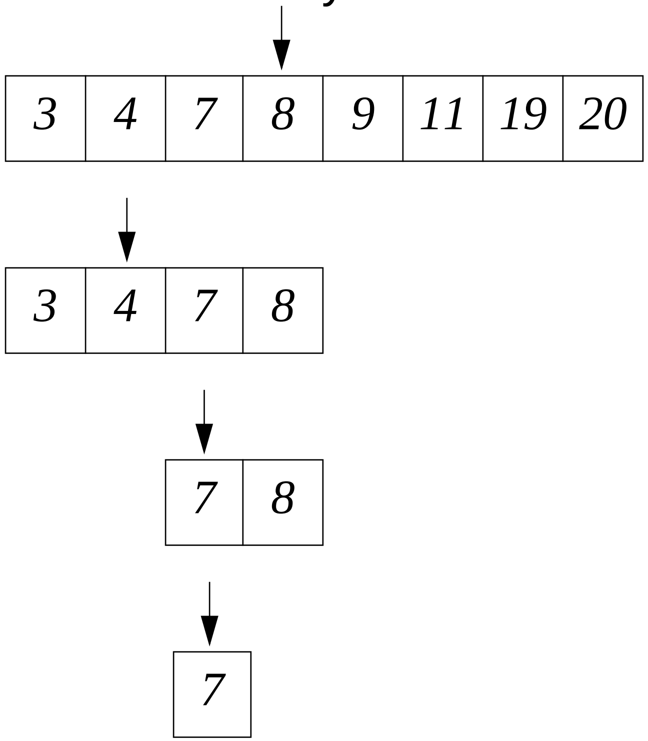
Binary Search

Problem: find a key element in a sorted array of integers.

Binary search is naturally expressed as a recursive algorithm:

If the array has more than one element,
then
split it in two,
eliminate the sub-array containing larger/smaller values.
Recurse on the other array.
else
check if the element is the one we are searching for

Search for key value 7:



Binary Search — Recursive Solution

```
// Requires:  num[] is sorted, high, low in range of num[]
// Ensures:   (result.keyFound == 0)
//         or ((result.keyfound == 1) and (num[result.index] = key))
keyIndex binSearch(int key, int num[], int low, int high)
{
   keyIndex result(0,0);
   if (low > high) {                 // Base case 1: empty range
      return keyIndex(0,0);          // not found
   } else if (low == high) {         // Base case 2: range of size 1
      if (key == num[high]) {
         return keyIndex(1,high);    // found at position high (== low)
      } else {
         return keyIndex(0,0);       // not found
      }
   } else {                          // high > low
      int mid = (high+low)/2;        // => mid < high
      if (key <= num[mid]) {         // Two recursive cases ...
         return binSearch(key, num, low, mid);
      } else {
         return binSearch(key, num, mid+1, high);
      }
   }
}
```

Records as Objects

BinSearch returns a pair of values. Since tuples are not a primitive in C++, we must encode the pair of values as an object:

```
class keyIndex {
   public :
      keyIndex(int k, int i) { keyFound = k; index = i; }
      int keyFound;     // == 0 or 1
      int index;        // if keyFound == 1, then should be a valid index
};
```

KeyIndex has a constructor that allows a new instance to be initialized with a given pair of integers.

Tail Recursion

A function is *tail-recursive* if it calls itself recursively only when returning its result:

```
int rfactorial(unsigned int n)
{
   if (n==0) {
      return 1;
   } else {
      return n*rfactorial(n-1);
   }
}
```

Tail-recursion can easily be transformed into iteration:

```
int ifactorial(unsigned int n)
{
   int result = 1;
   while (n != 0) {       // terminate loop with base case(s)
      result = n*result;
      n--;                // loop instead of calling recursively with n-1
   }
   return result;
}
```

Binary Search — Iterative Solution

Since binSearch() is tail-recursive, it is easy to transform:

```
keyIndex ibinSearch(int key, int num[], int low, int high)
{
   keyIndex result(0,0);
   while (low <= high) {                  // terminate if range is empty
      if (low == high) {                  // Base case 2: range size 1
         if (key == num[high]) {
            return keyIndex(1,high);      // found at position high (== low)
         } else {
            return keyIndex(0,0);         // not found
         }
      } else {                            // high > low
         int mid = (high+low)/2;          // => mid < high
         if (key <= num[mid]) {           // Two complex cases
            high = mid;                   // loop instead of recursing
         } else {
            low = mid+1;                  // loop instead of recursing
         }
      }
   }                                      // Base case 1: empty range
   return keyIndex(0,0);                  // not found
}
```

Sorting

Problem: sort an array of integers

The "obvious" solution — insertion sort — is not trivial to implement correctly, and is inherently slow (N elements will be sorted in $O(N^2)$ time).

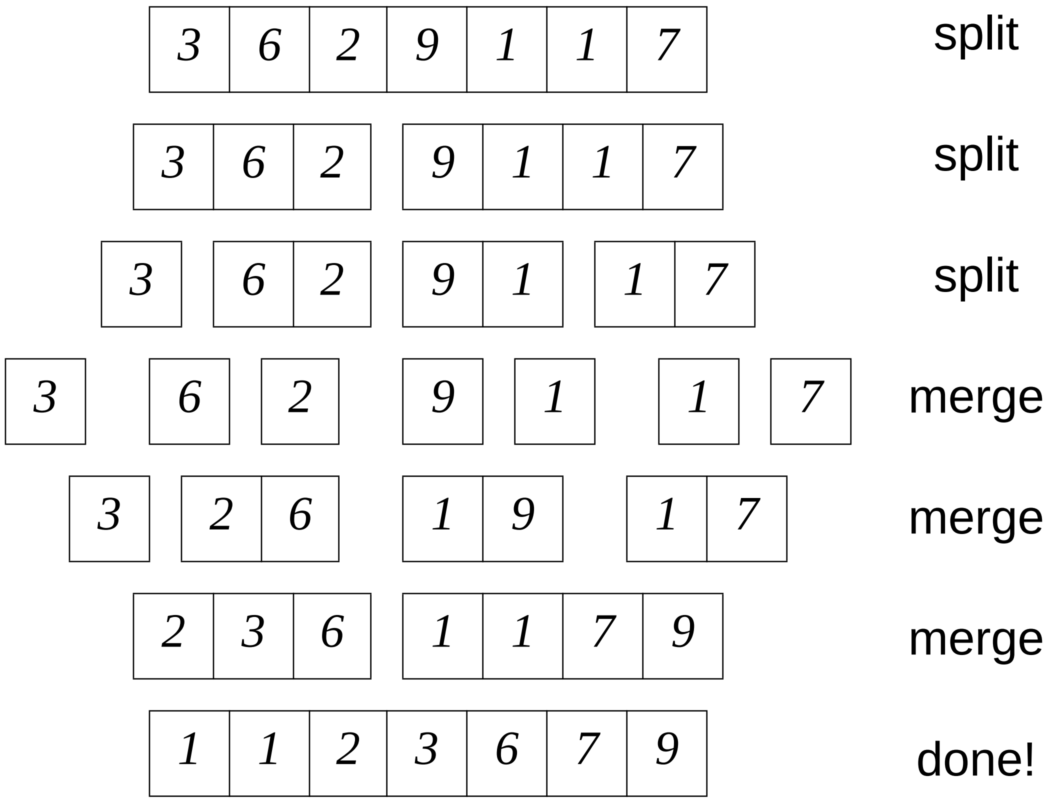
The principle of divide and conquer leads to an efficient, recursive solution:

- ❑ We want to sort an array of integers
- ❑ Split the array into two smaller arrays, and sort those
- ❑ Merge the two sorted arrays into one

Two questions remain:

- ❑ What are the base cases?
  - ☞ arrays of length 0 or 1 are trivially sorted
- ❑ How can we merge two sorted arrays into one?
  - ☞ in the obvious way!

MergeSort Example



Merge Sort

✔ *A function or procedure should always have a clear responsibility; promote readability by decomposing complex algorithms into helper functions.*

```
// Requires: a is an array of ints, length len
// Ensures:  a will be sorted
void mergeSort(int a[], int len)
{
   if (len <= 1) {
      return;                  // trivially sorted!
   }
   int *a1 = a;                // a1 points to the first half of a
   int l1 = len/2;             // len >= 2, so l1 >= 1
   mergeSort(a1, l1);          // a1 is now sorted
   int *a2 = a + l1;           // a2 points to the second half of a
   int l2 = len - l1;          // l1 < len, so l2 >= 1
   mergeSort(a2, l2);          // a2 is now sorted
   int *b = new int[len];      // need a buffer to merge into
   merge(a1, l1, a2, l2, b);   // merging is done by a separate function
   int i;
   for (i=0; i<len; i++) {     // copy result from b back to a
      a[i] = b[i];             // this is a serious source of inefficiency
   }                           // since each recursive call copies its arguments
   delete [] b;                // don't forget to delete b!
}
```

Merge

✔ *State loop invariants explicitly, and check that they hold through all execution paths.*

```
// Requires: a1 and a2 are sorted arrays of length l1 and l2 resp
// Ensures:  b will contain sorted merge of a1 and a2
void merge(int a1[], int l1, int a2[], int l2, int b[])
{
   int i1 = 0;
   int i2 = 0;
   int len = l1 + l2;
   int i;
   for (i=0; i<len; i++) {                // Invariant: (i == i1 + i2) && (len = l1 + l2)
      if (i1 < l1) {                      // a1 not exhausted
         if (i2 >= l2) {                  // but a2 is exhausted
            b[i] = a1[i1++];              // so copy rest of a1 to b
         } else if (a1[i1] <= a2[i2]) {   // a2 not exhausted, so compare
            b[i] = a1[i1++];              // a1[i1] smaller
         } else {
            b[i] = a2[i2++];              // a2[i2] smaller
         }
      } else {                            // a1 is exhausted
         b[i] = a2[i2++];                 // so copy rest of a2 to b
      }
   }                                      // Done when (i == len) && (i1 == l1) && (i2 == l2)
}
```

Refactoring Merge()

✔ *Eliminate duplicate code through refactoring or reorganizing.*

```
// Requires: a1 and a2 are sorted arrays of length l1 and l2 resp
// Ensures: b will contain sorted merge of a1 and a2
void merge(int a1[], int l1, int a2[], int l2, int b[])
{
   int i1 = 0;
   int i2 = 0;
   int len = l1 + l2;
   int i;
   for (i=0; i<len; i++) {       // Invariant: (i == i1 + i2) && (len = l1 + l2)
      if ((i1 < l1) && ((i2 >= l2) || (a1[i1] <= a2[i2]))) {
         b[i] = a1[i1++];        // a2 exhausted, or a1[i1[ is smaller
      } else {
         b[i] = a2[i2++];        // a1 is exhausted, or a2[i2] is exhausted
      }
   }
}
```

Optimizing MergeSort ...

Our mergesort() has O(N*log(N)) complexity, which is good, but copies the result of each merge back to the original array, which adds a fixed overhead.

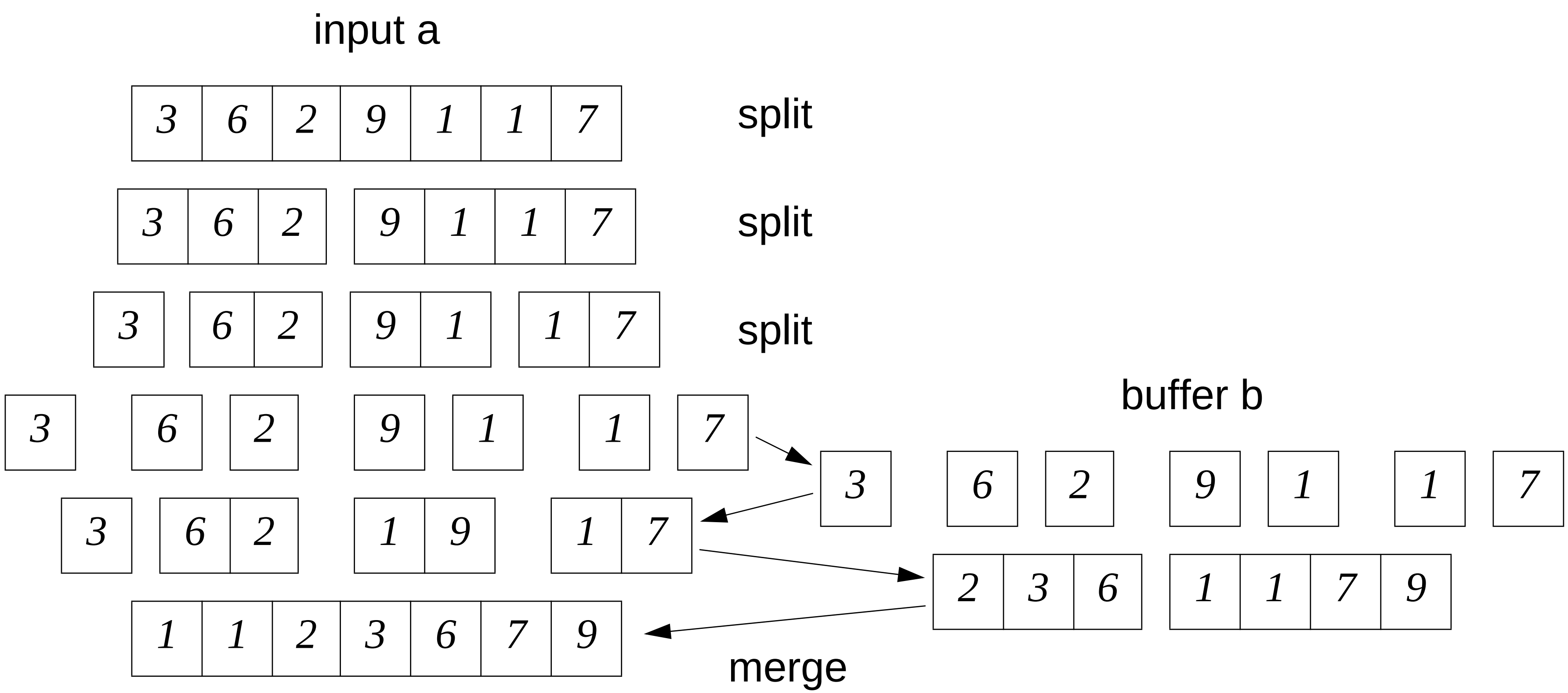
We can improve the performance, but make the program more complex.

Idea:

- allocate a fixed buffer for merging into
- define an auxiliary mergeSort function ms2b() that delivers the sorted array directly into the buffer
- define another function ms2a() that sorts the array with the help of ms2b()
- define mergeSort() with the help of ms2a() and ms2b()

MergeSort with a Fixed Buffer

Instead of each instance of mergeSort() allocating its own buffer, two versions of mergeSort() cooperate, either merging into the fixed buffer, or back to the original argument array:



A Faster MergeSort

We can improve the performance of MergeSort, at the cost of readability ...

```
void mergeSort2(int a[], int len)
{
    int *b = new int[len];
    ms2a(a, b, len);
    delete [] b;
}

// Ensures: a will be sorted into a
void ms2a(int a[], int b[], int len)
{
    if (len <= 1) {
        return;
    }

    int *a1 = a;
    int *b1 = b;
    int l1 = len/2;
    ms2b(a1, b1, l1);

    int *a2 = a + l1;
    int *b2 = b + l1;
    int l2 = len - l1;
    ms2b(a2, b2, l2);

    merge(b1, l1, b2, l2, a);
}
```

```
// Ensures: a will be sorted into b
void ms2b(int a[], int b[], int len)
{
    if (len <= 1) {
        if (len == 1) {
            b[0] = a[0];
        }
        return;
    }

    int *a1 = a;
    int *b1 = b;
    int l1 = len/2;
    ms2a(a1, b1, l1);

    int *a2 = a + l1;
    int *b2 = b + l1;
    int l2 = len - l1;
    ms2a(a2, b2, l2);

    merge(a1, l1, a2, l2, b);
}
```

Summary

You should know the answers to these questions:

- ❑ When can you implement algorithms with recursion?
- ❑ Why should you explicitly state pre- and post-conditions?
- ❑ When should you raise an exception?
- ❑ What is tail recursion? How can you eliminate it?
- ❑ What are loop invariants? Why are they important?
- ❑ When should you start optimizing your program?

Can you answer the following questions?

- ✎ *Our mergeSort() will crash if the argument array is shorter than the advertised length; how can we fix this?*
- ✎ *How would you implement mergeSort() without recursion?*
- ✎ *Why is code duplication a Bad Thing?*

5. Specifying Classes

- ❑ Abstract Data Types, Contracts and Invariants
- ❑ C++ Classes:
  - ☞ `public`, `protected` and `private` members
- ❑ Example of data abstraction:
  - ☞ a `TicTacToe` object
- ❑ Exceptions:
  - ☞ `try`, `catch` and `throw`
- ❑ Restricting visibility and write access:
  - ☞ `static` and `constant` declarations

Abstract Data Types and Invariants

Why do we need ADTs?

- to program at a *higher level of abstraction*
- to program with *reusable* software components
- to maintain program *invariants* (ensure server data consistency)
- to encapsulate and maintain client/server *contracts*
- to protect clients from *variations* in implementation

Contrast C++ (cpprev) and C (crev) stack implementations in lecture 2!

Design guidelines:

- What abstractions do you need? (i.e., abstract services/contracts)
- What are the program invariants? (i.e., consistency rules)
- Which data belong together? (i.e., via invariants and operations)

Example: Tic Tac Toe

Requirements specification: [Random House Dictionary of the English Language]

> *“A simple game in which one player marks down only crosses and another only ciphers [zeroes], each alternating in filling in marks in any of the nine compartments of a figure formed by two vertical lines crossed by two horizontal lines, the winner being the first to fill in three of his marks in any row or diagonal.”*

Explicit invariants:

- turn (current player) is either X or O
- X and O swap turns (turn never equals previous turn)
- game state is 3×3 array marked X, O or blank
- winner is X or O iff winner has three in a row

Implicit invariants:

- initially winner is nobody; initially it is the turn of X
- game is over when all squares are occupied, or there is a winner
- a player cannot mark a square that is already marked

Contracts:

- the current player may make a move, if the invariants are respected

C++ Classes

C++ classes are an extension to the C `struct` type constructor for records.

Class members are data and "functions" with varying levels of information hiding.

```
class ClassName {
public:
   // Data and methods accessible to clients, including constructors & destructors
protected:
   // Data and methods accessible to class methods, derived classes and friends only
private:
   // Data and methods accessible to class methods and friends only
}
```

Automatic (stack) instantiation:

```
ClassName oVal;         // Constructor called; destroyed when scope ends
```

Dynamic (heap) instantiation:

```
ClassName *oPtr;        // Pointer, so no constructor called
oPtr = new ClassName;   // Constructor called; must be explicitly deleted
```

Designing a Tic Tac Toe Game

tttMain.cpp:

- ❑ Driver — responsible for interacting with user
- ❑ Creates and destroys instances of TicTacToe game

TicTacToe.h:

- ❑ Abstract interface to TicTacToe game (*header file*)
- ❑ Declares public/private methods
- ❑ Shared by both driver and game implementation

TicTacToe.cpp:

- ❑ Includes needed libraries
- ❑ Implementation of TicTacToe game
- ❑ Responsible for maintaining game invariants during instantiation and updates

What should be the interface?

- ☞ Top-down strategy: *consider abstract services needed by driver*
- ☞ Bottom-up strategy: *consider game invariants and services*

Desired Interaction

```
Welcome to Tic Tac Toe!
Would you like to play a game? (y/n): y

   1|2|3
   -----
   4|5|6
   -----
   7|8|9

X plays: 5

   1|2|3
   -----
   4|X|6
   -----
   7|8|9

O plays: 5
Error: Square already occupied

   1|2|3
   -----
   4|X|6
   -----
   7|8|9

O plays: 0
Error: Move out of range 1-9
```

```
   1|2|3
   -----
   4|X|6
   -----
   7|8|9

O plays: 1

   The game continues ...

X plays: 9

   O|X|O
   -----
   X|X|O
   -----
   X|O|X

Nobody wins!!!

Would you like to play another game? (y/n): n
Goodbye!
```

The Tic Tac Toe Driver

```
/*
   File: tttMain.cpp
   Author: Oscar Nierstrasz 29.2.96
   Driver for Tic Tac Toe program
*/

#include <iostream.h>                          // Declare cout and endl
#include "TicTacToe.h"                         // Declare TicTacToe class

void playTicTacToe (void);
// void playTicTacToe(void) { cout << "not implemented yet" << endl; } // for testing

int main (void)
{
   cout << "Welcome to Tic Tac Toe!" << endl;
   cout << "Would you like to play a game? (y/n): ";

   char reply;
   cin >> reply;                               // Read from standard input stream
   while (reply == 'y') {
      playTicTacToe();
      cout << "Would you like to play another game? (y/n): ";
      cin >> reply;
   }
   cout << "Goodbye!" << endl;
   return 0;                                   // Unix process terminates without error
}
```

✔ *Prototyping strategy: always work with a running, if incomplete program, and incrementally “grow” the full version.*

Determining the Interface

✔ *Describe services at highest level of abstraction possible. Determine who is responsible for what!*

Always ask yourself, “can the object perform this task or is it my job?”

```
void playTicTacToe (void)
{
    TicTacToe game;                                  // new local instance
    int move;
    while (game.notover()) {                         // Describe driver in abstract terms!
        game.print();
        cout << game.turn() << " plays: ";
        cin >> move;
        try {                                        // This could fail!
            game.play(move);                         // Whose responsibility is it to check?
        }
        catch (xmsg &err) {                          // Standard class in <exception.h>
            cout << "Error: " << err.what() << endl; // Or possibly err.why()
        }
    }
    game.print();
    cout << game.winner() << " wins!!!" << endl << endl;
}
```

Exceptions

A server (i.e., a function, typically a member function of an object) may *throw* an exception if it cannot provide the requested service:

- ☞ the request was invalid (contract violated by client)
- ☞ the server failed (abnormal situation, e.g., out of memory)

The server should:

1. attempt to restore the invariant, and
2. inform the client by returning a suitable exception value

An *exception* is a value thrown by server to client:

- ☞ a number, an enum value, a string, an xmsg instance
- ☞ an instance of a specially designed exception class

Client may *catch* an exception and take appropriate action using try/catch construct.

✔ *Exceptions should only be used to signal abnormal situations, not normal flow of control.*

Specifying the Interface

```
/*
   File: TicTacToe.h
   Author: Oscar Nierstrasz 29.2.96
   Tic Tac Toe interface
*/

#ifndef TICTACTOE_H                // Include at most once!
#define TICTACTOE_H

#include <exception.h>             // Declare xmsg class (needed for interface of play())

class TicTacToe {
public :
   TicTacToe(void);                // Constructor

   int notover (void);             // True if game is not over
   const char *winner (void);      // Winner is "X", "O" or "Nobody"
   char turn (void);               // Whose turn is it?

   void play (int move)            // Current player marks a square
      throw(xmsg);                 // Invalid move raises exception

   void print (void);              // Pretty-print the current state

 private :                         // Private instance variables, types and methods ...
};

#endif // TICTACTOE_H
```

Instance Variables

Instance variables are needed to provide the services and induce the invariants.

Often most of these can be determined by considering the specification.

```
class TicTacToe {
public :                                // As before ...

private :
    enum Player { nobody, X, O };       // Symbolic names for players

    // Private instance variables
    Player _winner;                     // Initially nobody
    Player _turn;                       // Initially X
    int squaresLeft;                    // Initially 9
    Player square[9];                   // Initially all nobody
};
```

Remaining instance variables and other private members will be “discovered” during implementation ...

✔ *Use symbolic names and enumerated types to make your code as self-documenting as possible.*

Implementing the Constructor

```
/*
   File: TicTacToe.cpp
   Author: Oscar Nierstrasz 29.2.96

   Tic Tac Toe implementation
*/

#include <iostream.h>                // Declare cout and endl
#include "TicTacToe.h"               // Declare everything to be defined here

// Implementations of public and private methods ...

// Constructor:
TicTacToe::TicTacToe (void) :        // NB: TicTacToe() is within scope of TicTacToe class
   _winner(nobody),                  // Member initialization list
   _turn(X),                         // Whenever possible, initialize members here!
   squaresLeft(9)
{
   for (int i=0; i<9; i++)           // Cannot be initialized in MI list, so done in body
      square[i] = nobody;
}
```

- ☞ Constructors may have arguments, but never a return value
- ☞ Multiple constructors may be defined for different kinds of initializers

Implementing the Game

```
int
TicTacToe::notover (void)
{
   return (squaresLeft > 0) && (_winner == nobody);
}

const char *                  // Result string may not be modified by clients!
TicTacToe::winner (void)
{
   return winners[_winner];   // String representation of winner
}

char
TicTacToe::turn (void)
{
   return player[_turn];      // Char representation of current player
}

// Char and string names of players -- share one constant copy for all game instances!
// Oops! We should add their declarations to the list of private members in TicTacToe.h!
// Initialization of constant static members:
const char TicTacToe::player [3] = { ' ', 'X', 'O' };
const char * TicTacToe::winners [3] = { "Nobody", "X", "O" };

enum Player {                 // Oops! Now we need to change this type definition!
   nobody = 0,                // Representation fixed so we can index player[] and winners[]
   X      = 1,                // This goes in TicTacToe.h
   O      = 2
};
```

Static Declarations

```
class TicTacToe {
public :     // as before ...
private :    // as before ...
   static const char player [3];     // Only one, unmodifiable local copy of these arrays.
   static const char * winners [3]; // Both are indexed by Player values.
};
```

A static local variable has *class scope*, and persists across invocations

A static global variable has *file scope*, and is invisible outside file scope

NB: A static class member must be initialized just once!

Warning! Two *separate* but interacting meanings of `static`: [ARM p. 98]

- ☞ "allocated *once* at a fixed address"
- ☞ "*local* to a translation unit"

We could also have defined `player` and `winners` as static globals outside the class:

```
// Global variables are declared "static" so they are private to this module
static const char player [] = { ' ', 'X', 'O' };
static const char * winners [] = { "Nobody", "X", "O" };
```

Constant Declarations

`const` declarations are an important part of specifying class interfaces:

Function promises not to modify arguments:

```
void printGame (const TicTacToe&);// won't modify referenced game
```

Client promises not to modify return results

```
const char *winner (void);         // client won't change string
```

Object promises not to modify itself

```
const char *winner (void) const;   // can be safely applied to const game!
```

Inconsistent use of const variables is detected by the compiler:

```
char * s = game.winner();          // illegal conversion to non-const!
const char * s = game.winner();
*s = '*';                          // illegal assignment to constant string!
                                   // s is not constant; only what it points to!
```

Be careful exactly what is being declared constant!

```
char * const hi = "Hello world";   // hi is constant, but not what it points to!
hi[0] = 'B';                       // OK, since string is not constant
hi = "oh no!";                     // illegal assignment to constant!
```

Playing the Game

```
/*
   Current player makes a move by marking a square from 1-9.
   An exception is raised if the square is out of range or is already marked.
*/
void
TicTacToe::play (int move) throw(xmsg)
{
   if (!notover()) {                          // In Eiffel, these would be assertions!
      throw(xmsg("This game is already over!"));
      return;
   }
   if ((move<1) || (move>9)) {
      throw(xmsg("Move out of range 1-9"));
      return;
   }
   move--;                                    // OK, so decrement (index square from 0-8)
   if (square[move] == nobody) {              // Not already marked
      square[move] = _turn;                   // Mark the square
      squaresLeft--;
      _turn = (_turn == X) ? O : X;           // Switch current player
      checkWinner();                          // Need helper function to maintain invariants!
   } else {
      throw(xmsg("Square already occupied"));
   }
}
```

Printing the Game

```
// Pretty print the current state of the game:
void
TicTacToe::print(void)
{
    cout << endl;
    for (int row=0; row<3; row++) {                 // Print the game row by row
        int first = 3*row;
        cout << '\t'
            << showSquare(first) << '|'             // Need another helper function!
            << showSquare(first+1) << '|'
            << showSquare(first+2) << endl;
        if (row < 2)
            cout << "\t-----" << endl;
    }
    cout << endl;
}
/*
    Helper function for TicTacToe::print()
    Return ascii char for squares 0-8
    Returns 'X' or 'O' if occupied; otherwise square number as ascii char
*/
char
TicTacToe::showSquare(int m)
{
    Player state = square[m];
    return (state == nobody)?('1'+m):player[state];
}
```

The Complete TicTacToe Interface

```
class TicTacToe {
public :
   TicTacToe(void);                  // Constructor
   int notover (void);               // Public methods
   const char *winner (void);
   char turn (void);
   void play (int move) throw(xmsg);
   void print (void);
private :
   enum Player {    nobody = 0,      // Local type
                    X = 1,
                    O = 2
   };
   Player _winner;                   // Instance variables
   Player _turn;
   int squaresLeft;
   Player square[9];
   static const char player [3];     // Only one, unmodifiable local copy of these arrays.
   static const char * winners [3];  // Both are indexed by Player values.
   char showSquare(int);             // Local helper functions ...
   void checkWinner(void);           // Check for a winner (uses matchThree())
   int matchThree(int,int,int);      // Check for three in a row, and set _winner if there is
};
```

Summary

You should know the answers to these questions:

- What are invariants? How do they help in class design?
- What can one specify as public or private class members?
- How are object created?
- How do exceptions work?
- What belongs in a header file?
- What are `static` and `const` declarations for?

Can you answer the following questions?

- *When and how are objects destroyed?*
- *What belongs in the member initialization list (resp. body) of a constructor?*
- *Can you implement the missing helper functions for TicTacToe?*
- *Does it make sense to declare a function as* `static`*?*

6. Data Abstraction

- Run-time Stacks; Stacks as Data Abstractions
- Using a Stack to Interpret Postfix Expressions
- Stacks, Queues and Linked Lists
- Class Invariants
- Implementing the Linked List Abstraction
- Implementing Stacks
- Using a Stack to Balance Parentheses
- C++ trap: Shallow Copying and Call by Value

The Run-time Stack

The stack is a fundamental data structure used to record a context that will be returned to at a later point in time. Most programming languages use a run-time stack:

```
void main (void) { cout << "fact(5) = " << fact(5) << endl; }

int fact (int n) {
   if (n==0)   return 1;
   else        return n*fact(n-1);
}
```

The stack grows with each function call ...

main ...				
main; fact(3)=?	fact(3) ...			
main; fact(3)=?	fact(3); fact(2)=?	fact(2) ...		
main; fact(3)=?	fact(3); fact(2)=?	fact(2); fact(1)=?	fact(1) ...	
main; fact(3)=?	fact(3); fact(2)=?	fact(2); fact(1)=?	fact(1); fact(0)=?	fact(0) ...
main; fact(3)=?	fact(3); fact(2)=?	fact(2); fact(1)=?	fact(1); fact(0)=?	fact(0); return 1
main; fact(3)=?	fact(3); fact(2)=?	fact(2); fact(1)=?	fact(1); return 1	
main; fact(3)=?	fact(3); fact(2)=?	fact(2); return 2		
main; fact(3)=?	fact(3); return 6			
main; fact(3)=6				

... and shrinks with each return.

Stack as a Data Abstraction

✔ *Always encapsulate data structures as data abstractions.*

```
class Stack
{
public:
   Stack(void);                    // Construct an empty Stack
   ~Stack(void);                   // Destroy the Stack and its contents

   int count(void);                // Return how many Items the Stack holds
   int empty(void);                // Is the Stack empty?

   void push(Item item);           // Push an Item on top of the Stack
   Item top(void) throw(xmsg);     // Return value of the top Item
   void pop(void) throw(xmsg);     // Pop off the top Item
                                   // If empty, raise an exception

private:
   // Somehow, keep track of the state of the Stack ...
};
```

A naked data structure is easily corrupted. Only by defining an abstract interface can you ensure that your data will remain consistent independent of the rest of your program.

Postfix Expressions

A *Stack Machine* is a simple architecture for evaluating arithmetic expressions. Expressions written in *postfix form* are easy to interpret with a stack:

Example: 6 7 3 + 2 * -

Operation		*Stack*		
push	*6*	*6*		
push	*7*	*6*	*7*	
push	*3*	*6*	*7*	*3*
apply	+	*6*	*10*	
push	*2*	*6*	*10*	*2*
apply	*	*6*	*20*	
apply	-	*14*		

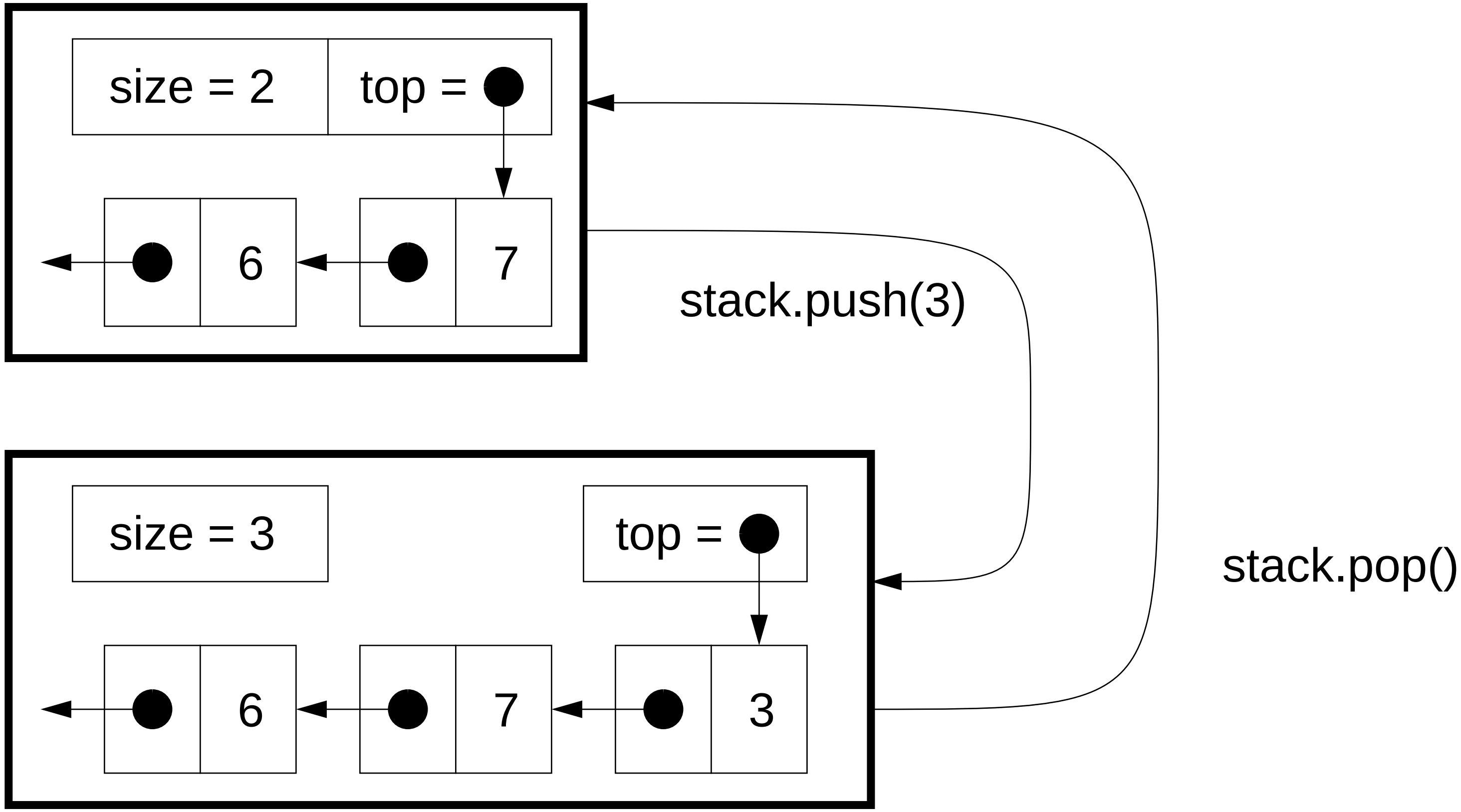
A Postfix Expression Interpreter

A postfix expression interpreter is straightforward to implement with a Stack:

```
void postfix(void) {
   Stack intStack;
   char c = ' ';
   cout << "Enter postfix expressions (\".\" to stop!)" << endl;
   while (c != '.') {
      try {
         int arg1, arg2;
         cin >> c;
         if (('0'<=c) && (c<='9')) {                      // push digits
            intStack.push(c - '0');
         } else {
            switch (c) {                                  // or apply operator to top numbers
            case '+':
               arg1 = intStack.top(); intStack.pop();
               arg2 = intStack.top(); intStack.pop();
               intStack.push(arg1 + arg2);
               cout << arg1 << " + " << arg2 << " = " << (arg1+arg2) << endl;
               break;
            // add other operators here ...
            default:
               cerr << "Invalid char " << c << " ignored" << endl;
               break;
            }
         }
      } catch (xmsg &err) { cout << "Exception: " << err.why() << endl; }
   }
}
```

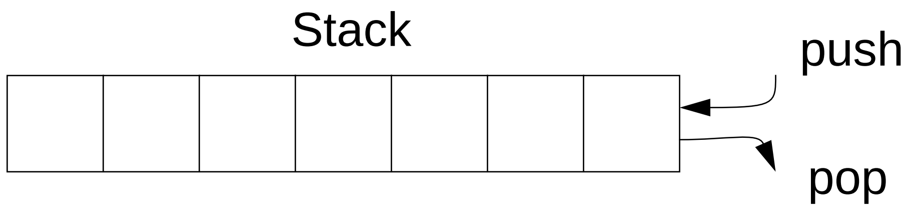
Stacks as Linked Lists

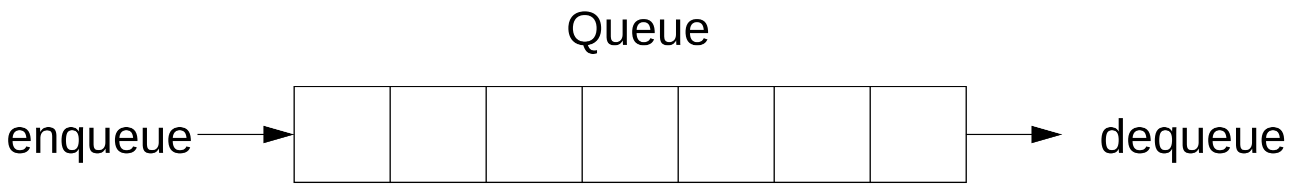
A Stack can easily be implemented using a linked data structure:



Stacks, Queues and Linked Lists

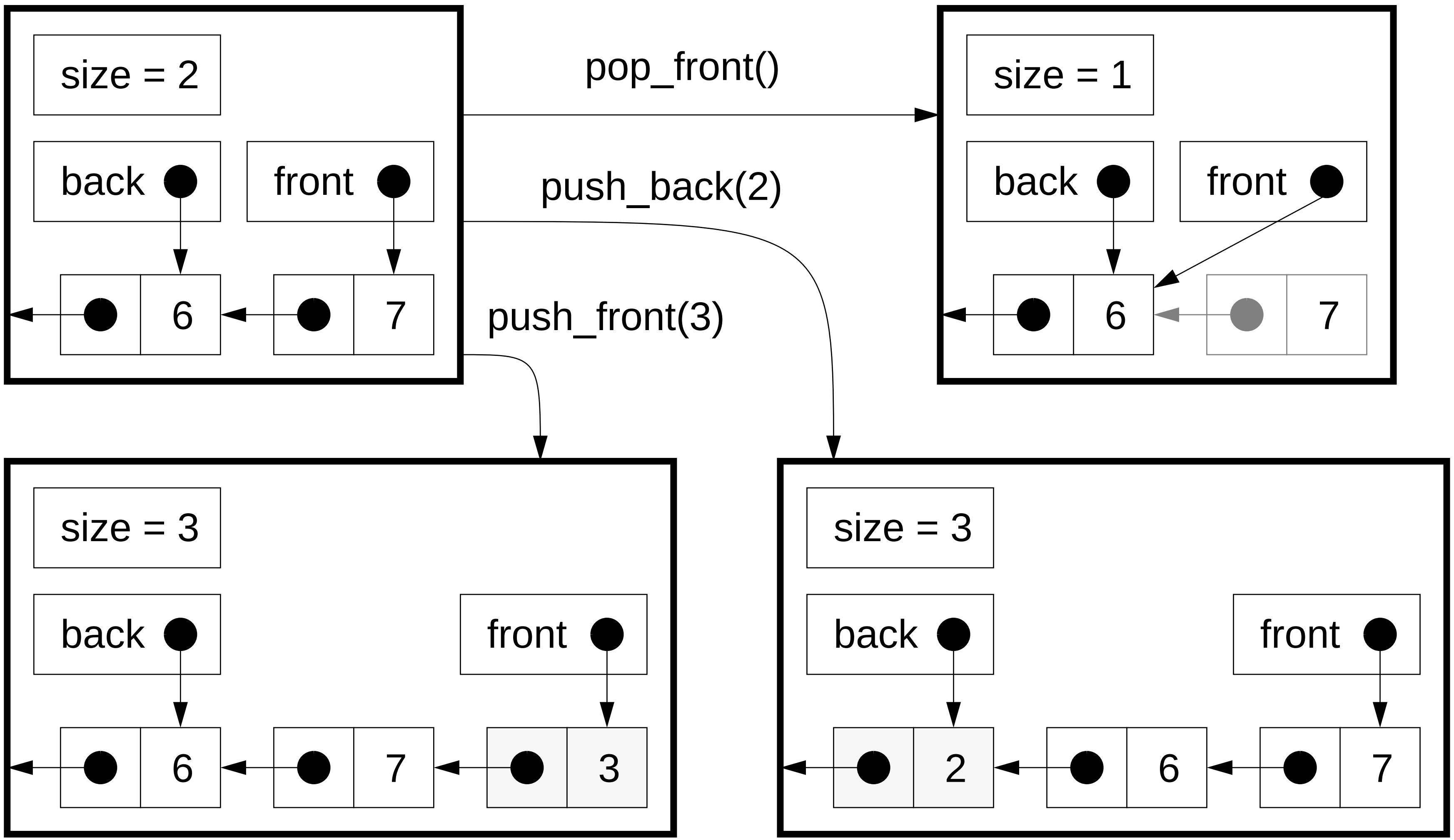
Stacks and Queues are both dynamic data abstractions that can be implemented using linked data structures.





This suggests that we should develop a separate Linked List abstraction that can be used to implement both Stacks and Queues.

Linked List Operations



Class Invariants

Recall that we implement data abstractions as classes — the class constructor must create instances that establish the class invariant, and each public method is responsible for maintaining the invariant.

A valid linked list instance has a size, front and back pointers, and a set of linked cells, such that:

- ❑ Initially size is zero; front and back point nowhere.
- ❑ When size is n > 0, there are n linked cells; front points to the first cell, and each cell points to the next; the back cell points nowhere.
- ❑ In case size = 1, front and back point to the same cell.

LList Declaration

```
class LList {                              // Declared in llist.h
public:
   LList(void);                            // Make an empty list
   ~LList(void);                           // Destroy the list and its contents!

   int count(void) { return size; };
   int empty(void) { return size == 0; };

   void push_front(Item item);             // Add item to front
   Item front(void) throw(xmsg);           // Return front item, if not empty
   void pop_front(void) throw(xmsg);       // Remove front item, if not empty

   void push_back(Item item);              // Add item to back
   Item back(void) throw(xmsg);            // Remove back item

   void print(ostream &os);                // Output a representation of the list on os

private:
   class Cell {                            // Private class (record) to link items
   public:
      Cell(Item val, Cell *nxt) { value = val; next = nxt; } // Constructor
      Item value;
      Cell *next;                          // Is zero if and only if this is the back cell
   };

   // Invariant: If size == 0, then frontCell == 0 and backCell == 0
   // else if size == 1, then frontCell == backCell and backCell->next == 0
   // else frontCell->...->next == backCell and backCell == 0

   Cell *frontCell;    // initially 0
   Cell *backCell;
   int size;           // >= 0

};                                         // WARNING:
```

Implementing List Methods

Recall that functions and procedures should always have a clear responsibility.

✔ *A method should always do one thing well; don't mix up responsibilities.*

Methods, like procedures, should be written at as high a level of abstraction as possible.

✔ *Methods should be short and easy to read.*

Rules of thumb:

- ❑ An ordinary method is typically 5 to 10 lines of code.
- ❑ A method that implements an algorithm might be 20 to 25 lines of code.

Complex methods should be decomposed using private helper methods.

List Constructor and Destructor

The constructor establishes the invariant:

```
// constructor for an empty stack:
LList::LList (void)
   : size(0), frontCell(0), backCell(0)
{
}
```

The destructor empties the stack so it can be cleanly deleted:

```
// destructor pops all cells:
LList::~LList(void)
{
   while (!this->empty()) {
      this->pop_front();        // If we don’t do this, the Cells will persist
   }                            // after the Stack is gone!
}
```

The C++ run-time will *only* delete automatic values (on the stack);
the destructor of a class is responsible for freeing all dynamic values.

Growing the List

Each method can assume that the object is in a valid state.

The state may be temporarily inconsistent inside the method, but the invariant must be re-established when the method terminates.

```
void
LList::push_front(Item item)      // Assume only that invariant holds
{
   Cell *newCell;
   newCell = new Cell(item, this->frontCell);
   // NB: the new Cell now points to frontCell,
   // even if frontCell == 0
   this->frontCell = newCell;     // Always do this
   if (this->empty()) {           // Handle special case of invariant!
      this->backCell = newCell;   // no longer empty, so set backCell
   }                              // to point here too
   size++;                        // Always do this
}
```

✎ *Can you implement pop_front( ) and push_back( )?*

Checking Pre-conditions

Remember to check pre-conditions, and raise an exception if they are violated.

```
// Requires: stack is non-empty
Item
LList::front(void) throw(xmsg)
{
   if (this->empty()) {
      throw(xmsg("Empty list has no front!"));
   }
   return this->frontCell->value;
}
```

Implementing a Stack with a Linked List

```
#ifndef STACK_H                  // NB: this implementation is a header file
#define STACK_H

#include <iostream.h>            // Declare ostream
#include <exception.h>           // Declare xmsg
#include "llist.h"

typedef int Item;                // Redefine as necessary ...

class Stack {                    // NB: all methods are inline
public:
   Stack(void)          { };     // Empty default constructor
   ~Stack(void)         { };     // Empty destructor

   int count(void)  { return myList.count(); };
   int empty(void)  { return myList.empty(); };

   void push(Item item)            { myList.push_front(item); }
   Item top(void) throw(xmsg)      { return myList.front(); }
   void pop(void) throw(xmsg)      { myList.pop_front(); }

   void print(ostream &os)         { myList.print(os); }

private:
   LList myList;                 // All methods implemented here
};

#endif // STACK_H
```

Example: Balancing Parentheses

Problem: Determine whether an expression containing parentheses, brackets and braces (i.e., ( ), [ ], and { }) is correctly balanced.

Example: “([[ ]] { ({ [ ] ( ) }) [ ] })” is balanced, “] {“ is not.

Approach: Push each left parenthesis on a stack, and pop it off when a right parenthesis is encountered. If the parentheses match, and the stack is empty at the end, the whole expression is balanced.

Example: “([{ }]]”

push(“(”)	→	“(”
push(“[”)	→	“([”
push(“{”)	→	“([{”
“{” matches “}” so pop( )	→	“([”
“[” matches “]” so pop( )	→	“(”
“(” doesn’t matches “]” so *not balanced*		

Parenthesis balancer

```
int balanced(char s[]) throw(xmsg)   // Assume s[] is a null-terminated ASCII string
{
    Stack myStack;
    int i = 0;
    while (s[i] != '\0') {
        switch (s[i]) {
        case '(':
            myStack.push(')');           // Push the matching parenthesis,
            break;                       // so we just need to test for equality
        case '[':
            myStack.push(']');
            break;
        case '{':
            myStack.push('}');
            break;
        case ')':
        case ']':
        case '}':
            if (myStack.empty())              { return 0; }       // Too many right parens
            else if (s[i] == myStack.top())   { myStack.pop();}   // OK, so continue
            else                              { return 0; }       // Mismatch
            break;
        default:
            break;
        }
        i++;
    }
    return myStack.empty();              // Equal number of matching left and right parens
}
```

Implementing a Queue with a Linked List

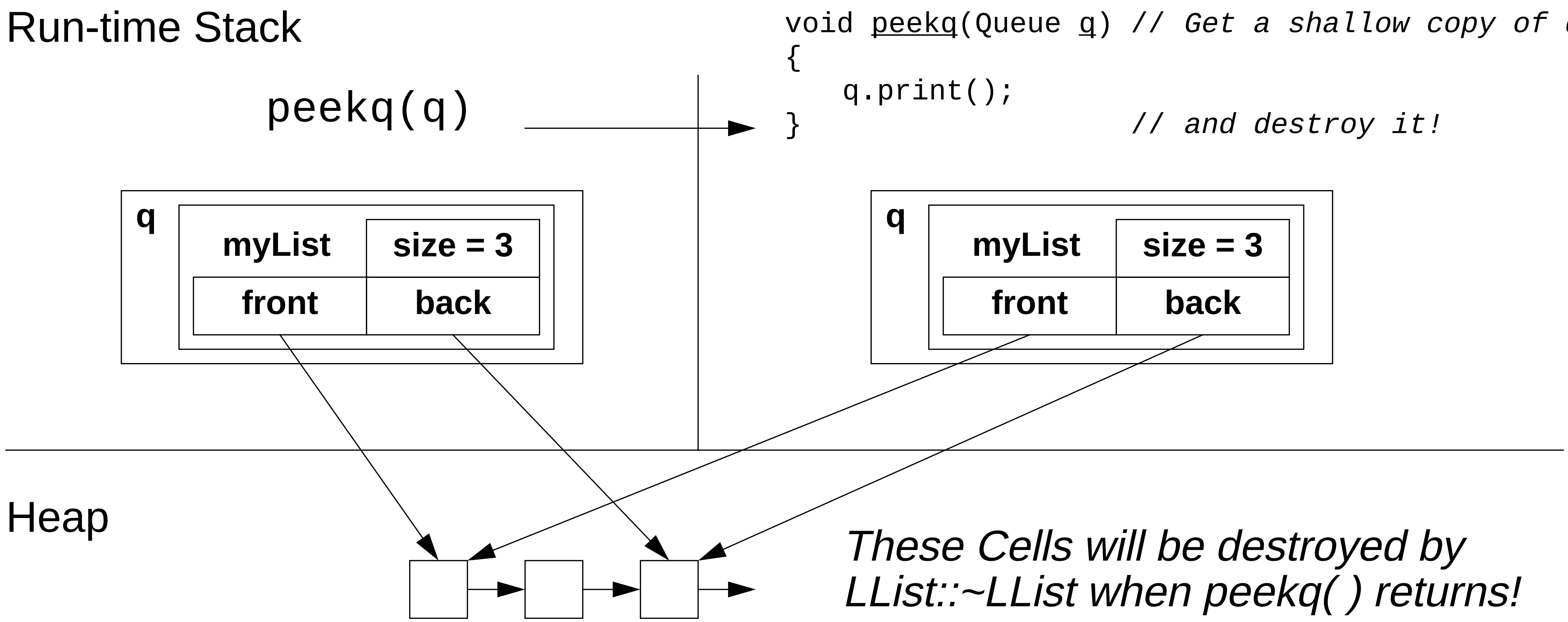
We can also implement a Queue as a wrapper around a linked list:

```
class Queue {                   // NB: all methods are inline functions
public:
   Queue(void)        { };
   ~Queue(void)       { };
   int count(void)    { return myList.count(); };
   int empty(void)    { return myList.empty(); };

   // join queue at tail with enqueue()

   Item tail(void) throw(xmsg)     { return myList.back(); }
   void enqueue(Item item)         { myList.push_back(item); }

   // leave queue at head with dequeue()

   Item head(void) throw(xmsg)     { return myList.front(); }
   void dequeue(void) throw(xmsg)  { myList.pop_front(); }

   void print(ostream &os)         { myList.print(os); }

private:
   LList myList;
};
```

The Dangers of Call by Value

Our LList class has a serious flaw. Parameters, by default, are passed by value. Since C++ does not know about the dynamic data your class may have allocated, only a *shallow copy* is passed. The copy's destructor will be called when the function returns.

```
void peekq(Queue q) // Get a shallow copy of q
{
   q.print();
}                   // and destroy it!
```



These Cells will be destroyed by LList::~LList when peekq( ) returns!

Guard Against Shallow Copies

If instances create and delete dynamic data as part of their state, you must guard against shallow copies being made when they are passed by value.

There are two possible solutions:

1. Implement a *copy constructor* that builds a copy correctly
2. Declare a copy constructor as `private`.
   - ☞ Instances can then only be passed by reference.
   - ☞ Attempts to pass instances by value will cause a compile-time error.

✔ *Declare a* `private` *copy constructor, if your objects should not be passed by value.*

```
class LList {
public: ...
private:
   LList(const LList&);// not implemented
   ...
};
LList::LList(const LList& arg) { // throw an exception if accidentally called
   throw(xmsg("LList::LList(LList&) not implemented"));
}
```

Summary

You should know the answers to these questions:

- ❑ What is the purpose of the run-time stack?
- ❑ What are typical applications of stacks?
- ❑ How can a stack or a queue be implemented with a linked list?
- ❑ Why is it important to encapsulate data structures within classes?
- ❑ What is a class invariant? Why is it important to specify?
- ❑ How can call by value invalidate a class invariant?

Can you answer the following questions?

- ✎ *Can you implement the missing methods of* `LList`*?*
- ✎ *How could you implement LList with only one pointer instead of two?*
- ✎ *How would you implement copying correctly for the* `LList` *class?*
- ✎ *Why can't C++ copy objects by value correctly?*

7. Managing Memory

- Orthodox Canonical Form
- Copy Constructors
- new and delete
- Assignment operators
- Inline functions
- Conditional compilation
- Operator overloading
- Friends
- IO Stream operators

Sources:

- Stanley B. Lippman, *C++ Primer, Second Edition*, Addison-Wesley, 1991.
- James O. Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.

Orthodox Canonical Form

Most of your classes should look like this:

```
class myClass
{
   public:
      myClass (void);                         // default constructor
      myClass (const myClass& copy);          // copy constructor
      ...                                     // other constructors
      ~myClass (void);                        // destructor

      myClass& operator= (const myClass&);    // assignment operator
      ...                                     // other public member functions
   private:
      ...
};
```

✔ *Use the orthodox canonical form for any non-trivial class whose objects will be copied or assigned to.*

Example: A String Class

C-style strings are inherently unsafe:

- strings are indistinguishable from char pointers
- string updates may cause memory to be corrupted

We would like to implement a String abstraction that protects us from common errors.
Should support:

- creation and destruction
- initialization from char arrays
- copying
- safe indexing
- safe concatenation and updating
- input and output
- length, comparison and other common functions

First version of String.h

Our String class will provide an interface to a hidden char array:

```
#ifndef STRING_H
#define STRING_H

#include <iostream.h>                                 // declare istream and ostream
#include <exception.h>                                // declare xmsg

class String
{
   public:
      String (void);                                  // default constructor
      String (const String& copy);                    // copy constructor

      String (const char* s);                         // char* constructor
      ~String (void);                                 // destructor

      String& operator= (const String&);              // assignment operator

      int strlen (void);                              // number of non-NULL chars

      char& operator[] (const int n) throw (xmsg);// safely return nth element
      int getline (istream&);                         // read into istream -- return 0 upon eof
      void print (ostream&);                          // print onto ostream

   private:
      // invariant: _s points to a NULL-terminated string on the heap
      char *_s;
};

#endif // STRING_H
```

Default Constructors

The default constructor for a class is called when a new instance is declared without any initialization parameters:

```
String anEmptyString;           // String::String() is called
String stringVector[10];        // String::String() is called ten times
```

Each constructor is responsible for properly initializing the state of a new object (i.e., establishing the *class invariant*):

```
String::String (void)           // NB: no member initialization list needed
{
   _s = new char[1];            // allocate a char array of length 1 on the heap
   _s[0] = '\0';                // make sure the string is NULL terminated
}
```

✔ *Decide what your class invariant is and make sure that each constructor correctly establishes the invariant.*

Automatic and Dynamic Objects

Recall that objects can either be allocated "on the stack" or "on the heap":

- Automatic objects are local to functions
  - constructors are called for objects where they are defined
  - destructors are called when functions exit
  - can only be returned "by value" (i.e., copying)

```
void f (void) {
   String s;          // constructor String::String() is called
}                     // destructor String::~String() is called
```

- Dynamic objects reside in global memory
  - created and destroyed by explicit calls to new and `delete`
  - may be shared by pointers or references

```
String* g (void) {
   String * s;        // just a pointer; no constructor is called
   s = new String;    // constructor String::String() is called
   return(s);         // client obtains a pointer to the new String
}                     // no destructor is called
```

Destructors

Dynamic objects are only needed if your objects must persist across function calls.

☞ A class constructor may need to allocate an array of new objects for its internal representation if the number of elements is not known in advance.

A single instance may be destroyed with a call to `delete`:

```
void h(void) {
   String * s = g();          // g() constructs a new instance of String
   delete s;                  // String::~String() is called here
}
```

An array of instances *must* be destroyed with a call to `delete[]`:

```
String::~String (void)
{
   delete [] _s;               // NB: an array, so not just "delete _s"!!!
}
```

✔ *If you use new, make sure that there will be exactly one matching delete!*

✔ *Destructors should deallocate all memory belonging to an object's private state.*

Copy Constructors

It can be very convenient to construct a new object from an existing instance.

A *copy constructor* takes an existing instance as an argument:

```
#include <string.h>                        // declare strcpy() ...

String::String (const String& copy)
{
   _s = new char[copy.strlen() + 1];       // leave room for NULL at end
   if (_s == 0)                            // new might fail!!!
      throw(xmsg("can't allocate string"));
   ::strcpy(_s, copy._s);                  // want strcpy() in the global scope
}
```

NB:

- If we do not declare copy as const, we cannot construct copies of const Strings
- If we declare copy as `String` rather than `String&`, a new copy will be made before it is passed to the constructor!
  - ☞ Functions arguments are always passed by value in C++
  - ☞ The "value" of a reference or a pointer is a pointer!
- Within a single class, all private members are visible (as is copy._s)

Other Constructors

Class constructors may have arbitrary arguments, as long as their signatures are unique and unambiguous:

```
String::String (const char* s)                    // initialize from ordinary string
{
   _s = new char[::strlen(s) + 1];                // must use global strlen()!
   if (_s == 0)
      throw(xmsg("can't allocate string"));
   ::strcpy(_s, s);
}
```

Since the argument is not modified, we can declare it as `const`. This will allow us to construct String instances from constant char arrays.

The implementation of this constructor is uncomfortably similar to that of the copy constructor. Factoring out the common parts will give us less code to maintain!

Refactoring Common Code

Helper functions are often implemented as private member functions:

```
String::String (const String& copy)           // copy constructor
{
   become (copy._s);
}

String::String (const char* s)                // char* constructor
{
   become (s);
}

void
String::become (const char* s) throw(xmsg)
{
   // Establishes, but does not assume class invariant:
   // The caller must ensure that _s is currently unassigned,
   // or that its previous value is deleted!
   _s = new char[::strlen(s) + 1];
   if (_s == 0)
      throw(xmsg("can't allocate string"));      // cleanup needed?
   ::strcpy(_s, s);
}
```

✔ *Clearly document whether helper functions assume or ensure class invariants!*

Assignment Operators

Assignment is different from the copy constructor because an instance already exists:

```
String&
String::operator= (const String& copy)
{
   if (this != &copy) {       // copying self would lead to an inconsistent state!
      delete [] _s;           // be sure to delete the previous value!
      become(copy._s);        // (re-)initialization is the same as before
   }
   return *this;              // return a reference, not a copy!
}
```

NB:

- ❑ Return `String&` rather than `void` so the result can be used in an expression
- ❑ Return `String&` rather than `String` so the result won't be copied!
- ❑ `this` is a pseudo-variable whose value is a pointer to the current object
  - ☞ so `*this` is the value of the current object, which is returned by reference

✔ *An assignment operator should always test for copying of self*

Shallow and Deep Copying

If you do not define a copy constructor or assignment operator for your class, the C++ compiler will automatically generate one for you.

- ☞ The default copy semantics is a *shallow copy:* the values of each data member are copied from one instance to another
- ☞ If some of the data members are pointers, only the pointers will be copied, not the objects pointed to.

If we do not define our own assignment operator, instances of String will share the same representation after an assignment!

- ☞ Modifying one String will also cause the other to be changed since they now share the same representation
- ☞ Worse, if either object is destroyed, the other will be left in an inconsistent state.

A *deep copy* causes data members to be recursively copied. In general it is not possible for the compiler to tell whether deep or shallow copying is required, so you should always implement your own copying functions for non-trivial objects.

Inline Functions

An inline function is like a macro: its body is copied wherever it is called rather than generating a run-time function call.

☞ An inline declaration is only a “hint” to the compiler, and may be ignored!

Inline class member functions can be declared directly in the header file:

```
inline int strlen (void) const { return ::strlen(_s); }
```

Note that strlen() is declared as const, so it can be applied to constant String objects

☞ if it is not declared const, a compiler error will be generated when it is applied to a constant String!

✔ *Don’t bother declaring inline functions unless (or until) you can be sure you will get a real improvement in performance.*

✔ *Short, frequently called functions may be good candidates for inlining.*

Using the Constructors

Default constructor:

```
String str;                 // initialized to empty string by String::String()
```

Char array constructor:

```
String hi ("howdy!");       // initialized by String::String(char*)
```

Copy constructor:

```
String hello (hi);          // initialized from hi by String::String(String&)
```

Assignment operator:

```
str = hi;                   // copies value from hi using operator=
```

Warning:

```
String s ();                // not a constructor call -- declares a function s()
```

Implicit Conversion

When an argument of the “wrong” type is passed to a function, the C++ compiler looks for a constructor that will convert it to the “right” type:

```
str = "hello world";   // Oops -- String& String::operator=(char*) not defined!
```

is implicitly converted to:

```
str = String("hello world");
```

since `String::operator=` expects a `String` argument and there is a constructor `String::String(char*)` that can be used to convert a `char*` to a `String`

NB: ☞ *A new String object will be created from the “hello world” char array, used to assign its value to str, and then destroyed.*

✔ *Don’t worry too much about unnecessary copying, but be aware of its overhead in computationally intensive code!*

Conditional Compilation

We can use conditional compilation to turn debug messages on and off:

In String.cpp:

```
// Comment out the following line to turn off debug msgs:
#define DEBUG
#include "Debug.h"

String::String (const String& copy)
{
   debug("Made a new string = ");          // let me know whenever a new String is constructed
   debug(copy._s);
   become(copy._s);
}
```

In Debug.h:

```
inline void debug (const char*);           // function prototype

#ifdef DEBUG
inline void
debug (const char * msg)                   // debug messages are printed if DEBUG is defined
{
   cerr << "DEBUG> " << msg << endl;       // print to standard error stream
}
#else
inline void debug (const char * msg) { ; } // else an empty statement is inlined ...
#endif
```

Operator Overloading

Not only assignment, but other useful operators can be "overloaded" provided their signatures are unique:

```
char&
String::operator[] (const int n) throw(xmsg)  // safely return the nth element
{
   if ((n<0) || (strlen()<=n)) {              // complain if index is invalid
      throw(xmsg("array index out of bounds"));
   }
   return _s[n];
}
```

NB: A *reference* to the nth element is returned, so it can be used as an lvalue in an assignment expression:

```
str[0] = 'X';       // will raise an exception if str has length 0
```

To *prohibit* String instances from being updated by indexing, we can declare:

```
const char& String::operator[] (const int n) { ... }
```

Overloadable Operators

The following operators may be overloaded:

<table>
<tr><th colspan="8">Overloadable Operators</th></tr>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td><td>&</td><td>|</td></tr>
<tr><td>-</td><td>!</td><td>,</td><td>=</td><td><</td><td>></td><td><=</td><td>>=</td></tr>
<tr><td>++</td><td>--</td><td><<</td><td>>></td><td>==</td><td>!=</td><td>&&</td><td>||</td></tr>
<tr><td>+=</td><td>-=</td><td>/=</td><td>%=</td><td>^=</td><td>&=</td><td>|=</td><td>*=</td></tr>
<tr><td><<=</td><td>>>=</td><td>[]</td><td>()</td><td>-></td><td>->*</td><td>new</td><td>delete</td></tr>
</table>

- It is not possible to introduce new operators (i.e., such as `**` for exponentiation)
- Operator precedence is fixed by the language
- The arity may not be changed (i.e., unary operators like `!` cannot be overloaded with a binary definition)
- Class member functions always take `this` as an implicit argument

Friends

Instead of:

```
cout << "str = ";
str.print();
cout << endl;
```

We would like to say:

```
cout << "str = " << str << endl;
```

So ... we need a binary function << that takes a `cout` and a `String` as arguments, prints the string, and returns the value of cout.

- Can't be a member function of `String` since target is `cout`
- But must have access to `String`'s implementation

Solution: declare this foreign function as a "friend" of String:

```
class String
{
      friend ostream& operator<<(ostream&, const String&);
   public:    // ...
   private:   // ...
};
```

IOStream Operators

The binary `operator<<` is a member of neither String nor ostream:

```
ostream& operator<< (ostream& outStream, const String& s)
{
   return outStream << s._s;    // only friends can access _s
}
```

Friend functions can often be avoided by:

1. providing a class member function that does most of the work
2. defining a binary function that reverses the arguments

```
inline
istream& operator>> (istream& inStream, String& s)
{
   s.getline(inStream);         // getline() updates the String
   return inStream;             // now we can write: cin >> str >> ...
}
```

Dynamic Memory Management

```
int
String::getline (istream& in)                  // dynamically read string from input stream
{
    char c;                                    // last char read
    int curLen = 0, maxLen = strlen();         // current string length and buffer available
    while ((c = in.get()) != EOF) {            // read to end of file or next newline
        if (curLen == maxLen) {                // oops -- out of space: need some more!
            _s[curLen] = '\0';                 // sanity: current string must be NULL-terminated
            maxLen = (maxLen==0)?2:(maxLen*2); // well, let's just double the current size
            grow(maxLen);                      // call helper function to double size
        }
        if (c == '\n') {                       // got end of line, so clean up and return
            _s[curLen] = '\0';
            return 1;                          // return 1 (true) is all is OK
        }
        _s[curLen++] = c;                      // remember the char read
    }
    return 0;                                  // hit end of file, so return 0 (false)
}

void
String::grow (int newSize) throw(xmsg)         // make a new String object of length newSize
{
    char * old = _s;
    _s = new char[newSize];
    if (_s == 0)
        throw(xmsg("can't allocate string"));
    ::strcpy(_s, old);
    delete [] old;
}
```

The Final String.h

```
#ifndef STRING_H
#define STRING_H

#include <iostream.h>   // declare istream and ostream
#include <exception.h> // declare xmsg

class String
{
        friend ostream& operator<<(ostream&, const String&);

    public:
        String(void);                                               // default constructor
        ~String (void);                                             // destructor
        String (const String& copy);                                // copy constructor

        String (const char*s);                                      // char* constructor

        String& operator= (const String&);                          // assignment

        inline int strlen (void) const { return ::strlen(_s); }    // current length
        char& operator[] (const int n) throw (xmsg);                // safe indexing
        String& operator+= (const String&) throw (xmsg);            // concatenation (exercise)
        int getline (istream&);                                     // read state from input stream

    private:
        char *_s;

        void become (const char*) throw (xmsg);                     // internal copy function
        void grow (int) throw (xmsg);                               // helper for getline()
};

#endif // STRING_H
```

Summary

You should know the answers to these questions:

- ❑ When should you use the “orthodox canonical form”?
- ❑ When are the different kinds of constructors called?
- ❑ When do you need `new` and `delete`?
- ❑ What is the difference between `delete` and `delete[]`?
- ❑ How do the copy constructor and the assignment operator differ?
- ❑ When should you use `inline` functions?
- ❑ How can you overload operators?
- ❑ What are `friend` declarations useful for?

Can you answer the following questions?

- ✎ *Why would you overload operator()? new? delete? ....*
- ✎ *Is it always possible to design classes so that `friend`s are not necessary?*
- ✎ *Can you define in-place concatenation as an operator+= member function?*
- ✎ *Can you define general concatenation as a global operator+ function?*

8. Inheritance

- Uses of inheritance
- Polymorphism and `virtual` member functions
- Default function arguments
- Public inheritance
- Base class initialization
- Function pointers

The Board Game

Tic Tac Toe is a pretty dull game, but there are many other interesting games that can be played by two players with a board and two colours of markers.

Example: Go-moku [Random House Dictionary of the English Language]

> *"A Japanese game played on a go board with players alternating and attempting to be first to place five counters in a row."*

☞ We would like to implement a program that can be used to play several different kinds of games using the same game-playing abstractions

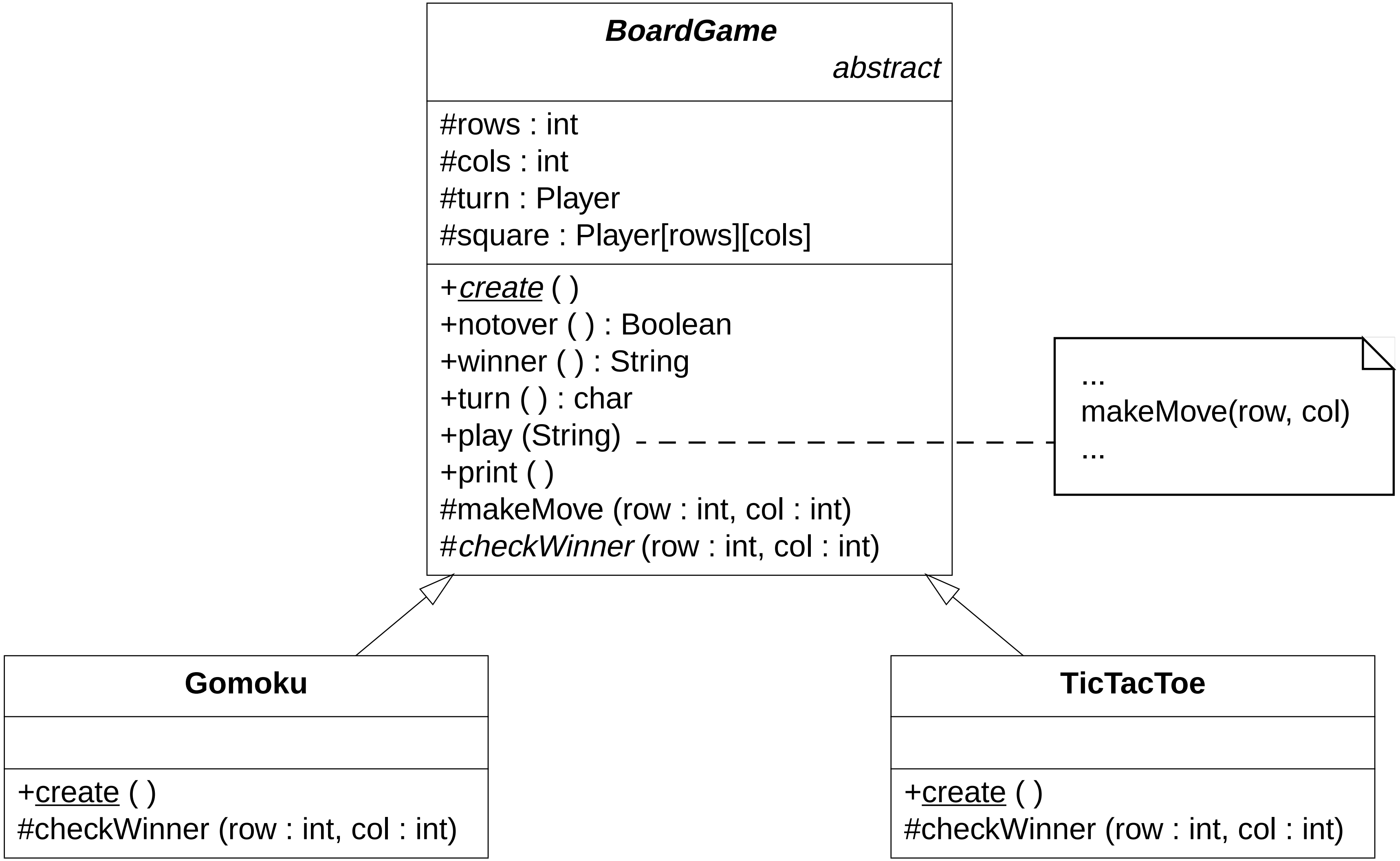
To start with, our program will let us play either Go-moku or Tic Tac Toe. We hope to use our experience implementing Tic Tac Toe to factor out the common abstractions as an abstract BoardGame class ...

Interaction

We will have to change the display and interaction to handle larger board games:

```
Welcome to The Board Game!
Would you like to play a game? (y/n): y
What game would you like to play?
Tic Tac Toe (t) or Go-moku (g)?: t

             A   B   C
           +---+---+---+
         a |   |   |   |
           +---+---+---+
         b |   |   |   |
           +---+---+---+
         c |   |   |   |
           +---+---+---+


X plays: bB
             A   B   C
           +---+---+---+
         a |   |   |   |
           +---+---+---+
         b |   | X |   |
           +---+---+---+
         c |   |   |   |
           +---+---+---+
O plays: q
Are you sure you want to quit this game? (y/n):y
Would you like to play another game? (y/n): n
Goodbye!
```

Class Hierarchy



Uses of Inheritance

Inheritance in object-oriented programming languages can be used for (at least) three different, but closely related purposes:

Conceptual hierarchy:

- Go-moku *is-a* kind of Board Game; Tic Tac Toe *is-a* kind of Board Game

Polymorphism:

- Instances of `Gomoku` and `TicTacToe` can be uniformly manipulated as instances of `BoardGame` by a client program

Software reuse:

- `Gomoku` and `TicTacToe` reuse the `BoardGame` interface
- `Gomoku` and `TicTacToe` reuse and extend the `BoardGame` representation and the implementations of its operations

Polymorphism

playGame() becomes more generic by making the abstract game a parameter.

```
void
playGame (BoardGame& game)    // Can be called with an instance of either Gomoku or TicTacToe
{
    String move;
    while (game.notover()) {
        cout << game << game.turn() << " plays: ";
        cin >> move;
        try {
            // Here we should check if the player wants to quit the game ...
            game.play(move);
        }
        catch (xmsg &err) {
            cout << "Error: " << err.why() << endl;
        }
    }
    cout << game << game.winner() << " wins!!!" << endl << endl;
}
```

Polymorphic Destruction

The main program is now responsible for creating and destroying BoardGame instances:

```
BoardGame * game;                // abstract, so we can only declare a pointer
game = makeGame();               // we get a pointer to some kind of game
playGame(*game);                 // we can play it
delete game;                     // and ask it to destroy itself
```

Only one function needs to know the concrete subclasses of BoardGame:

```
BoardGame*                       // Return type is abstract class
makeGame (void)
{
   cout << "What game would you like to play?" << endl;
   cout << "Tic Tac Toe (t) or Go-moku (g)?: ";
   String reply;
   cin >> reply;

   switch (reply[0]) {           // What happens if reply.strlen() == 0?
    case 't' :
      return new TicTacToe;      // We call new, so the client must call delete
      break;
    case 'g' :
      return new Gomoku;         // Either TicTacToe or Gomoku instances can be returned
      break;
    default:
      cout << "Hm ... I guess you want to play Tic Tac Toe ..." << endl;
      return new TicTacToe;
   }
}
```

The BoardGame Interface

```
class BoardGame {
   public :
      BoardGame (int rows = 8, int cols = 8)          // Constructor with default arguments
                throw(xmsg);                           // Out of memory raises exception
      virtual ~BoardGame (void) = 0;                   // Pure, virtual destructor

      int notover (void);                              // True if game is not over
      const char *winner (void);                       // Winner is "X", "O" or "Nobody"
      char turn (void);                                // Whose turn is it? 'X' 'O' or ' '

      void play (String move)                          // Current player marks a square
                throw (xmsg);                          // Invalid move raises exception

      void print(void);                                // Pretty-print the current state

   protected :
      enum Player { nobody = 0, X = 1, O = 2 };

      const int rows, cols;                            // Shape of the board
      Player _winner;                                  // Initially nobody
      Player _turn;                                    // Initially X
      int squaresLeft;                                 // Initially rows*cols
      Player ** square;                                // The board; initially all nobody

      static const char player[3];                     // Char names of the players
      static const char * winners [3];                 // Char* names of the players

      virtual void makeMove (int row, int col)         // Current players makes a move
                throw (xmsg);                          // Exception if invalid
      virtual void checkWinner (int row, int col) = 0; // Check if the last move wins

      int inRange (int row, int col);                  // (row,col) are in range for this board
};
```

Virtual Members

Data and methods that will be accessible to, or redefined by, subclasses should be declared as `protected`, not `private`.

Member functions that may be redefined by subclasses should be declared `virtual`:

- ☞ Calls to virtual functions will be dynamically resolved to the correct implementation (or “method”) defined for the target instance
- ☞ Any function that might be redefined should be virtual
- ☞ Constructors *cannot* be declared virtual
- ☞ Destructors should *always* be virtual

Member functions that *must* be redefined should be declared *pure virtual:*

- ☞ Classes with pure virtual functions are *abstract*, and cannot be instantiated
- ☞ Pure virtual *destructors* must nevertheless be defined!

✔ *A subclass should only redefine a member function if it has been declared virtual!*

Default Initializers

Default values may be specified for any function:

- ❑ When the function is called with missing arguments, default values are taken
  - ☞ e.g., `f()` is the same as `f(3)` if we declare `void f(int n = 3);`
- ❑ Arguments with default initializers *must* follow those without
  - ☞ if we declare `void nonsense (int x = 1, int y);`
    then what does it mean to call `nonsense(5)`?!
- ❑ Default initializers effectively declare several functions with different signatures
  - ☞ i.e., we now have both `void f(int);` and `void f(void);`
- ❑ Default initializers must appear in the *declaration* of a function, not in its definition
  - ☞ i.e., in the header file, not the implementation

✔ *Be sure that the implicit signatures of functions with default initializers do not overlap with those of other declared functions!*

Arrays of arrays

```
BoardGame::BoardGame (int rs, int cs) throw(xmsg) :            // Boardgame constructor
    rows(rs), cols(cs),                             // Initialize constant data members
    _winner(nobody),
    _turn(X),
    squaresLeft(rs*cs)                              // NB: can use expressions to initialize members
{
    debug("calling BoardGame constructor");     // Notify when constructor/destructor is called
    square = new Player* [rows];                    // square now points to an array of rows pointers
    if (square == 0)                                // Might fail for a ridiculously large board!
        throw(xmsg("Can't allocate board"));
    for (int r=0; r<rows; r++) {
        square[r] = new Player [cols];              // Each row pointer now points to cols Players
        if (square[r] == 0)
            throw(xmsg("Can't allocate board"));
        for (int c=0; c<cols; c++)
            square[r][c] = nobody;                  // Should explicitly initialize, even if nobody = 0
    }
}


BoardGame::~BoardGame (void)                        // BoardGame destructor
{
    debug("calling BoardGame destructor");
    for (int r=0; r<rows; r++)
        delete [] square[r];                        // Delete the array pointed to by square[r]
    delete [] square;                               // And delete the array pointers too!
}
```

Non-Virtual Functions

Do not declare base class functions virtual if they will never be overridden:

```
int
BoardGame::notover (void)            // Game isn't over if squares are left and there is no winner
{
   return (squaresLeft > 0) && (_winner == nobody);
}
const char *
BoardGame::winner (void)             // Return the char * name of the winner
{
   return winners[_winner];
}
char
BoardGame::turn (void)               // Return the char name of the current player
{
   return player[_turn];
}
int
BoardGame::inRange (int row, int col)// (row,col) are valid on this board
{
   return (0<=row) && (row<rows) && (0<=col) && (col<cols);
}
```

✎ *Are any of these functions good candidates to be declared* `virtual`*?*

Using Virtual Functions

Virtual functions are useful for parameterizing generic procedures

```
/*
   Current player makes a move by marking a square labelled aA-zZ.
   An exception is raised if the square is out of range or if the move is invalid.
*/
void
BoardGame::play (String move) throw(xmsg)
{
   if (!notover()) {
      throw(xmsg("This game is already over!"));
      return;
   }
   if (move.strlen() != 2) {
      throw(xmsg("Improper response: please give coordinates [a-z][A-Z]"));
   }
   // Check if move is in range, and convert to index into square[][]
   int row = move[0] - 'a';
   int col = move[1] - 'A';
   if (!inRange(row, col)) {
      throw(xmsg("Row out of range"));
   }
   makeMove (row, col);                // Try to make the requested move (might throw exception)
   checkWinner (row, col);             // Check if this is a winning move (if so, set _winner)
}
```

Defining Virtual Functions

Virtual functions can implement default behaviour:

```
/*
   The default implementation assumes you can mark any empty square.
   (This is all you need for Tic Tac Toe or Go-moku.)
   Override this to implement a different logic for valid moves.
*/
void
BoardGame::makeMove (int row, int col)
{
   if (square[row][col] == nobody) {                 // If square not already marked
      square[row][col] = _turn;                      // then mark the square
      squaresLeft--;
      _turn = (_turn == X) ? O : X;                  // and switch current player
   } else {
      throw(xmsg("Square already occupied"));
   }
}
```

Pure virtual functions are declared but not defined.

- ☞ Pure virtual destructors, on the other hand, *must* be defined, since they will be called when instances of derived classes are destroyed.

Public Inheritance

A new class can be *derived* from an existing *base class* by inheritance.

The derived class may introduce new features or override inherited features.

- ☞ If the derived class is to be concrete, pure virtual must be redefined
- ☞ Only virtual base member functions should be overridden!
- ☞ The derived class should always define its own constructors, destructors and (if needed) the assignment operator.

Derived class functions may access all public and protected features of the base class.

```
class Gomoku : public BoardGame {                  // Public members of BoardGame will stay public
   public :
      Gomoku (int r=19, int c=19, int ws=5)        // Constructor with default arguments
                throw(xmsg);                       // Can fail if base constructor fails
      virtual ~Gomoku (void);                      // Virtual destructor needed for derived class
   protected :
      const int winningScore;                      // New data member (instance variable)
      virtual void checkWinner (int row, int col);// Inherited pure virtual must be overridden
      void checkScore (int row,                    // New function member
                       int col,
                       void (*thisMove) (int&,int&),
                       void (*thatMove) (int&, int&));
};
```

Protected or private inheritance causes inherited features to be reclassified accordingly

Base Class Initialization

Abstract classes *must* have constructors since they are called by derived classes:

```
Gomoku::Gomoku (int r, int c, int ws) throw(xmsg) :
   BoardGame(r,c),                     // Base members are initialized with r rows and c columns
   winningScore(ws)
{
   debug("calling Gomoku constructor"); // Notify when base/derived constructors are called
}

Gomoku::~Gomoku (void)
{
   debug("calling Gomoku destructor");  // Base destructor will be automatically called!
}
```

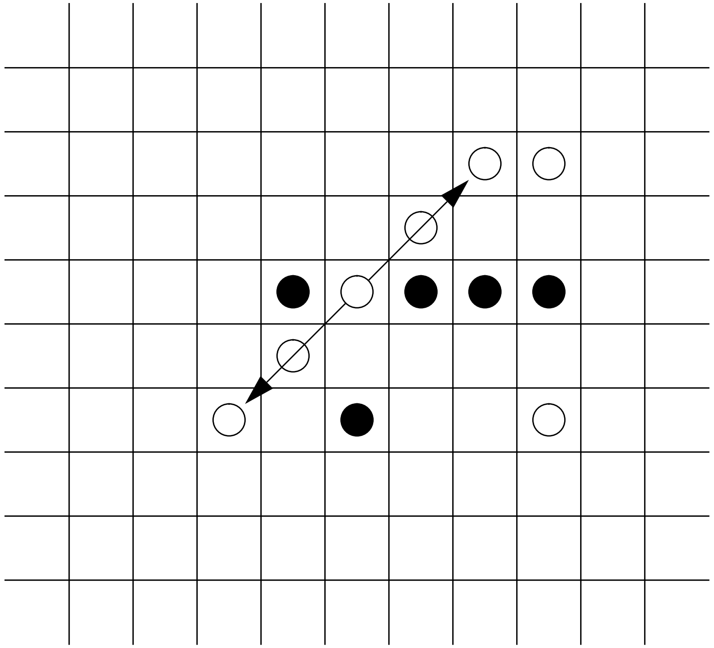
Base members are constructed before and destructed after derived members:

```
Welcome to The Board Game!
Would you like to play a game? (y/n): y
What game would you like to play?
Tic Tac Toe (t) or Go-moku (g)?: g
DEBUG> calling BoardGame constructor
DEBUG> calling Gomoku constructor
            A   B   C   D   E ...
          +---+---+---+---+---
        a |   |   |   |   | ...
          +---+---+---+---+ ...
X plays: q
Are you sure you want to quit this game? (y/n):y
DEBUG> calling Gomoku destructor
DEBUG> calling BoardGame destructor
```

Keeping Score

The Go board is too large to search it exhaustively for a winning Go-moku score.

Instead, we know a winning sequence must include the last square marked, so we search in all directions starting from that square to see if we find 5 in a row:



We must do the same thing in four directions.

How can we parameterize the algorithm by the directions to search?

Using Function Pointers

```
void
Gomoku::checkWinner (int row, int col)
{
   checkScore(row, col, right, left);          // Factor out the common algorithm
   checkScore(row, col, up, down);             // Apply in the four directions
   checkScore(row, col, northeast, southwest); // right, left etc. are function pointers
   checkScore(row, col, northwest, southeast);
}

void
Gomoku::checkScore (int row, int col,
   void (* thisMove) (int&, int&),             // Value passed is the address of a function!
   void (* thatMove) (int&, int&))             // Not the same as: void *thatMove(int&, int&)!
{
   int score = 1, r=row, c=col;                // Score is 1 at current location

   thisMove(r,c);                              // Increment in this direction
   while (inRange(r,c) && square[r][c] == square[row][col]) {
      score++;                                 // Neighbour is same as me, so increase score
      thisMove(r,c);                           // NB: same as (*thisMove)(r,c)
   }
   r = row; c = col;                           // Go back to starting square
   thatMove(r,c);                              // Continue in opposite direction
   while (inRange(r,c) && square[r][c] == square[row][col]) {
      score++;
      thatMove(r,c);
   }
   if (score >= winningScore)                  // We found 5 in a row!
      _winner = square[row][col];              // so current player is the winner.
}
```

Using Static Functions

Static functions are private to a file and cannot clash with similarly-named functions that might be defined in other files:

```
static void right (int&, int&);                 // Declared and defined in Gomoku.cpp
static void left (int&, int&);
static void up (int&, int&);
static void down (int&, int&);
static void northeast (int&, int&);
static void southwest (int&, int&);
static void northwest (int&, int&);
static void southeast (int&, int&);


void right (int& row, int& col) { col++; }      // Boring functions, but they make checkWinner()
void left (int& row, int& col) { col--; }       // much easier to define and maintain!

void up (int& row, int& col) { row++; }
void down (int& row, int& col) { row--; }

void northeast (int& row, int& col) { row++; col++; }
void southwest (int& row, int& col) { row--; col--; }

void northwest (int& row, int& col) { row++; col--; }
void southeast (int& row, int& col) { row--; col++; }
```

Implementation Inheritance

Tic Tac Toe is just Go-moku on a 3x3 board with a winning score of 3 instead of 5.

TicTacToe.h *must* declare a new constructor and destructor:

```
#ifndef TICTACTOE_H
#define TICTACTOE_H

#include "Gomoku.h"

class TicTacToe : public Gomoku {
   public :
      TicTacToe (void) throw(xmsg);
      virtual ~TicTacToe (void);
} ;

#endif // TICTACTOE_H
```

TicTacToe.cpp just overrides the default initializers of the Gomoku constructor:

```
#include "TicTacToe.h"

TicTacToe::TicTacToe (void) throw(xmsg) :
   Gomoku(3,3,3)                             // 3x3 board with winning score of 3
{
   debug("calling TicTacToe constructor");    // NB: This will be called after
}                                            // BoardGame and Gomoku constructors

TicTacToe::~TicTacToe (void)                 // Nothing new to destruct!
{
   debug("calling TicTacToe destructor");     // Destruction is in reverse order
}
```

Summary

You should know the answers to these questions:

- ❑ How does polymorphism help in writing generic code?
- ❑ How can you use inheritance and virtual functions to realize polymorphism?
- ❑ What are pure virtual functions?
- ❑ When should features be declared protected rather than public or private?
- ❑ What features can and should a derived class define?
- ❑ Why should destructors be virtual, but not constructors?
- ❑ When can you use function pointers to avoid duplicating code?

Can you answer the following questions?

- ✎ *Can you implement* `BoardGame::print()` *and* `operator<<`*?*
- ✎ *How can we improve* `BoardGame`*'s protected interface? Why should we?*
- ✎ *Can you specify the invariants maintained by* `BoardGame`*? By* `Gomoku`*?*
- ✎ *Should we have defined a copy constructor and operator= for* `BoardGame`*?*
- ✎ *Should* `Gomoku::winningScore` *and* `Gomoku::checkScore()` *have been declared* `private` *instead of* `protected`*?*

9. Tools

- ❑ Makefiles: manage file dependencies
- ❑ Version Control: manage multiple versions of files
- ❑ Debuggers: explore state of running program
- ❑ Profilers: analyze call graph of an execution instance
- ❑ SNiFF+: browse and navigate source code
- ❑ Purify: monitor memory accesses
- ❑ Other tools ...

Sources

- ❑ “UNIX in a Nutshell,” O’Reilly, 1994

Makefiles

Make is a tool for updating generated files (e.g., object files and executable programs) when files they depend on are modified.

☞ Make uses a user-specified list of dependencies and update commands defined in a *makefile* to compute the minimum set of files to regenerate.

```
# Makefile for prog

# prog depends on two object files:

prog :      prog.o mylib.o
            CC prog.o mylib.o -o prog

# prog.o and mylib.o each depend on a source file and a header file:

prog.o :    prog.cpp mylib.h
            CC -c prog.cpp

mylib.o :   mylib.cpp mylib.h
            CC -c mylib.cpp
```

Running ‘make’ with no arguments will create the first target (prog).

If any of the dependent files have been modified, the appropriate commands are run

✎ *What happens if mylib.cpp is modified? What about mylib.h?*

Make Options

Usage:

make [*options*] [*targets*]

Options:

-f *makefile*	Use *makefile* as the description file.
-n	Print commands but don't execute them.
-s	Execute, but do not display command lines.
-t	Touch the target files, causing them to be updated.

Run 'man make' for further options ...

✔ *Always define makefiles, even for your most trivial projects.*

Description File Lines

Blank lines are ignored

Comment lines:

- ❑ Everything following a ‘#’ is ignored.

Dependency lines:

- ❑ The target should be regenerated if any prerequisite is newer.

  `targets : prerequisites`

  NB: dependency lines must never start with a tab!

Suffix rules:

- ❑ All files with the first suffix are prerequisites for those with the second.

  `.suffix.suffix:`

Commands:

- ❑ Command lines start with a tab, following a dependency line or a suffix rule.

  If the line starts with “-”, errors are ignored; with “@”, echoing is suppressed

Macros:

- ❑ Macros have the form `name = string` and are referenced by either `$(name)` or `${name}`

Macros and Special Targets

Internal Macros

- $@ The current target.
- $? The list of prerequisites that are newer than the target.
  Can't be used in suffix rules.
- $< The name of the current prerequisite newer than the target.
  Only in suffix rules.
- $* Like $<, but with the suffix removed.
  Only in suffix rules.

Special Target Names:

- .DEFAULT: What to make if the request target has no rules.
- .IGNORE: Ignore error codes (same as -i option).
- .SILENT: Execute but don't echo commands (same as -s).
- .SUFFIXES: Recognize the following suffixes as targets in suffix rules.

Gomoku Makefile

```
# Make macros:

GMKO = gmkMain.o BoardGame.o Gomoku.o TicTacToe.o String.o

CXX          = CC
LFLAGS       = -L/opt/SUNWspro/SC3.0.1/lib
CFLAGS       = -O

# Suffix rules:

.SUFFIXES: .cpp .C
.cpp.o:
   $(CXX) $(CFLAGS) -c $<

.C.o:
   $(CXX) $(CFLAGS) -c $<

all : gomoku

gomoku : ${GMKO}
   $(CXX) ${GMKO} ${LFLAGS} -o $@

clean :
   rm -rf *.o

gmkMain.o :    TicTacToe.h Gomoku.h BoardGame.h String.h Debug.h
BoardGame.o :  BoardGame.h String.h Debug.h
Gomoku.o :     Gomoku.h BoardGame.h String.h Debug.h
TicTacToe.o :  TicTacToe.h Gomoku.h BoardGame.h String.h Debug.h
String.o :     String.h
```

Makefile for g++

A properly parameterized Makefile can easily be adapted to a different compiler:

```
# Compile .cxx files with g++:
CXX         = g++
LFLAGS      = -lstdc++
CFLAGS      = -fhandle-exceptions -O
```

Makedepend

Dependencies between files can be automatically generated and updated by running a tools like `makedepend`

- Dependencies must be listed at the end of the makefile
- Intermediate files are generated by suffix rules
- Dependencies are generated by recursively parsing source and header files

Version Control

A version control system keeps track of multiple file revisions:

- ❑ check-in and check-out of files
- ❑ logging changes (who, where, when)
- ❑ merge and comparison of versions
- ❑ retrieval of arbitrary versions
- ❑ “freezing” of versions as releases
- ❑ reduces storage space (manages sources files + multiple “deltas”)

✔ *You should use a version control system for any project that is non-trivial, developed by a team, or delivered to multiple clients*

SCCS and RCS are two popular version control systems for UNIX.

RCS

Overview of RCS commands:

ci	Check in revisions
co	Check out revisions
rcs	Set up or change attributes of RCS files
ident	Extract keyword values from an RCS file
rlog	Display a summary of revisions
merge	Incorporate changes from two files into a third
rcsdiff	Report differences between revisions
rcsmerge	Incorporate changes from two RCS files into a third
rcsclean	Remove working files that have not been changed
rcsfreeze	Label the files that make up a configuration

RCS Usage

When *`file`* is checked in, and RCS file called *`file,`*v is created in the RCS directory:

```
mkdir RCS             # create subdirectory for RCS files
ci file               # put file under control of RCS
```

Working copies must be checked out and checked in.

```
co -l file            # check out (and lock) file for editing
ci file               # check in a modified file
co file               # check out a read-only copy (i.e., for compiling, etc.)
ci -u file            # check in file, but leave a read-only copy (= ci/co)
rcsdiff file          # report changes between working copy and latest revision
```

Additional RCS Features

Keyword substitution

- Various keyword variables are maintained by RCS:

`$Author$`	who checked in revision (username)
`$Date$`	date and time of check-in
`$Log$`	description of revision (prompted during check-in)

  and several others ...

Revision numbering:

- Usually each revision is numbered *release.level*
- Level is incremented upon each check-in
- A new release is created explicitly: `ci -r2` ***file***

Debuggers

A debugger is a tool that allows you to examine the state of a running program:

- ❑ step through the program instruction by instruction
- ❑ view the source code of the executing program
- ❑ execute up to a specified *breakpoint*
- ❑ set and unset breakpoints anywhere in your program
- ❑ display values of variables in various formats
- ❑ manually set the values of variables
- ❑ examine the state of an aborted program (in a “core file”)

Various debuggers are available for UNIX: gdb, sdb, dbx

☞ To use a debugger effectively, you must compile with the `-g` option

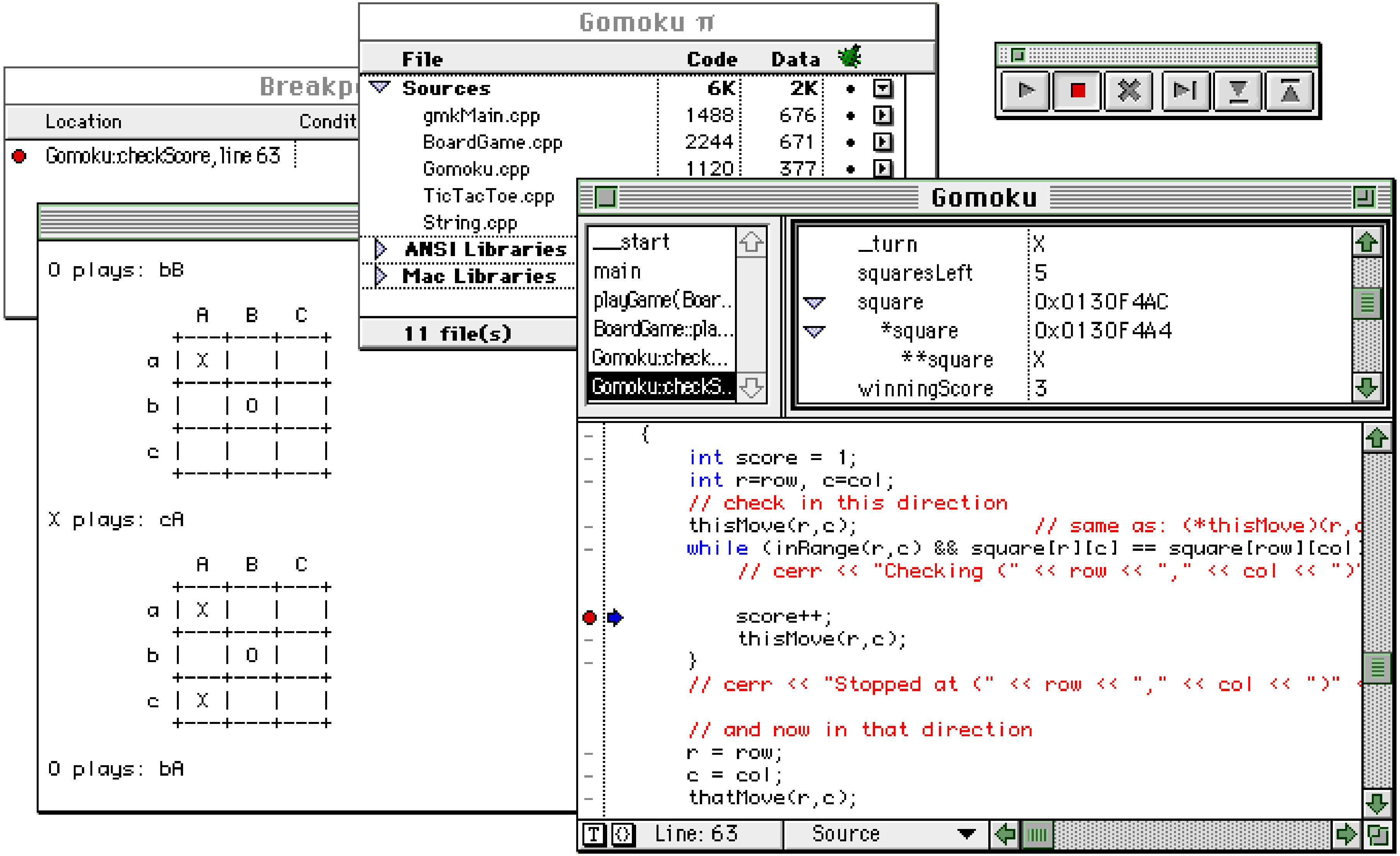
NB: debuggers are object code specific, so can only be used with programs compiled with compilers generating compatible object files. (sdb and dbx for CC; gdb for g++)

✔ *Use a debugger whenever you are unsure why your program is not working.*

Using dbx

```
oscar@pogo 1: dbx gomoku
Reading symbolic information for gomoku
   ...
(dbx) stop inmethod checkWinner
(2) stop inmember checkWinner
(dbx) run
Running: gomoku
(process id 27536)
Welcome to The Board Game!
Would you like to play a game? (y/n): y
What game would you like to play?
Tic Tac Toe (t) or Go-moku (g)?: t

          A   B   C
        +---+---+---+
      a |   |   |   |
        +---+---+---+
      b |   |   |   |
        +---+---+---+
      c |   |   |   |
        +---+---+---+

X plays: aA
stopped in Gomoku::checkWinner (optimized)
   at line 44 in file "Gomoku.cpp"
(dbx) where

=>[1] Gomoku::checkWinner(this = 0x3d0c8,
      row = 0, col = 0) (optimized),
      at 0x16a20 (line ~44) in "Gomoku.cpp"

  [2] BoardGame::play(this = ???, move = CLASS)
      (optimized), at 0x16180 (line ~81)
      in "BoardGame.cpp"

  [3] playGame(game =  CLASS) (optimized),
      at 0x15888 (line ~67) in "gmkMain.cpp"

  [4] main() (optimized), at 0x154f8 (line ~19)
      in "gmkMain.cpp"

(dbx) cont

          A   B   C
        +---+---+---+
      a | X |   |   |
        +---+---+---+
      b |   |   |   |
        +---+---+---+
      c |   |   |   |
        +---+---+---+

O plays: q
Are you sure you want to quit this game? (y/n):y
Would you like to play another game? (y/n): n
Goodbye!

execution completed, exit code is 0
(dbx) exit
```

GUI Debuggers — CodeWarrior



Profilers

A profiler can be used to display the call graph profile data of an executed program

- ❑ the program must be compiled with a special flag (e.g., -pg) that will cause profile data to be generated when the program is run
- ❑ profile data is generated in a special file (e.g., gmon.out)
- ❑ the profiler (e.g., gprof, lprof or prof) is run with the profile data and the object file (containing the symbol table) as arguments
- ❑ the call graph can be displayed in various formats (e.g., by decreasing total time, by decreasing number of calls, by symbol name, by symbol address ...)

✔ *Use a profiler to gain insight into where your program is spending most of its time.*

- ☞ Never try to “optimize” your program without profiling it first!
- ☞ Use a profiler to check which functions have been “exercised”.

Using gprof

Profilers can generate statistics in a variety of formats ...

```
granularity: each sample hit covers 2 byte(s) for 50.00% of 0.02 seconds
    %cumulative     self              self    total
 time    seconds  seconds  calls  ms/call  ms/call    name
 50.0       0.01     0.01                             filebuf::overflow(int) [1]
 50.0       0.02     0.01                             ostream::tellp(void) [2]
  0.0       0.02     0.00     57     0.00     0.00    BoardGame::inRange(int, int) [58]
  0.0       0.02     0.00     24     0.00     0.00    Gomoku::checkScore(int, int, void (*)
                                                      (int&, int&), void (*)(int&, int&)) [59]
  0.0       0.02     0.00     20     0.00     0.00    String::operator [](const int) [60]
  0.0       0.02     0.00     13     0.00     0.00    BoardGame::notover(void) [61]
  0.0       0.02     0.00      9     0.00     0.00    String::grow(int) [62]
  0.0       0.02     0.00      9     0.00     0.00    String::getline(istream&) [63]
  0.0       0.02     0.00      8     0.00     0.00    down(int&, int&) [64]
  0.0       0.02     0.00      7     0.00     0.00    up(int&, int&) [65]
  0.0       0.02     0.00      7     0.00     0.00    operator <<(ostream&, BoardGame&) [66]
  0.0       0.02     0.00      7     0.00     0.00    BoardGame::print(void) [67]
  0.0       0.02     0.00      6     0.00     0.00    left(int&, int&) [68]
  0.0       0.02     0.00      6     0.00     0.00    right(int&, int&) [69]
  0.0       0.02     0.00      6     0.00     0.00    northeast(int&, int&) [70]
  0.0       0.02     0.00      6     0.00     0.00    northwest(int&, int&) [71]
  0.0       0.02     0.00      6     0.00     0.00    southeast(int&, int&) [72]
  0.0       0.02     0.00      6     0.00     0.00    southwest(int&, int&) [73]
  0.0       0.02     0.00      6     0.00     0.00    Gomoku::checkWinner(int, int) [74]
  0.0       0.02     0.00      6     0.00     0.00    BoardGame::play(String&) [75]
  0.0       0.02     0.00      6     0.00     0.00    BoardGame::turn(void) [76]
  0.0       0.02     0.00      6     0.00     0.00    BoardGame::makeMove(int, int) [77]
...
```

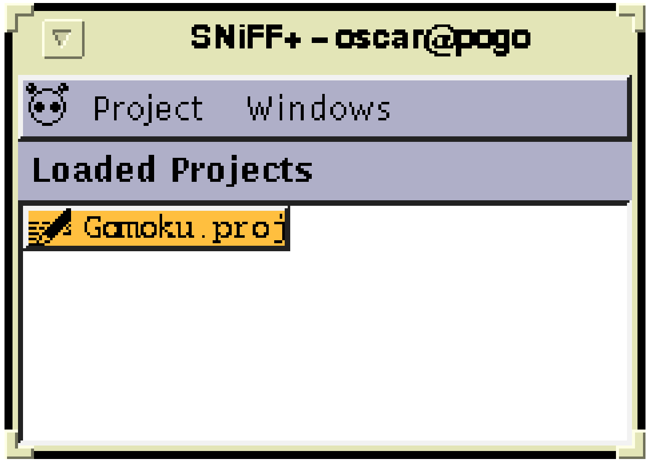
SNiFF+

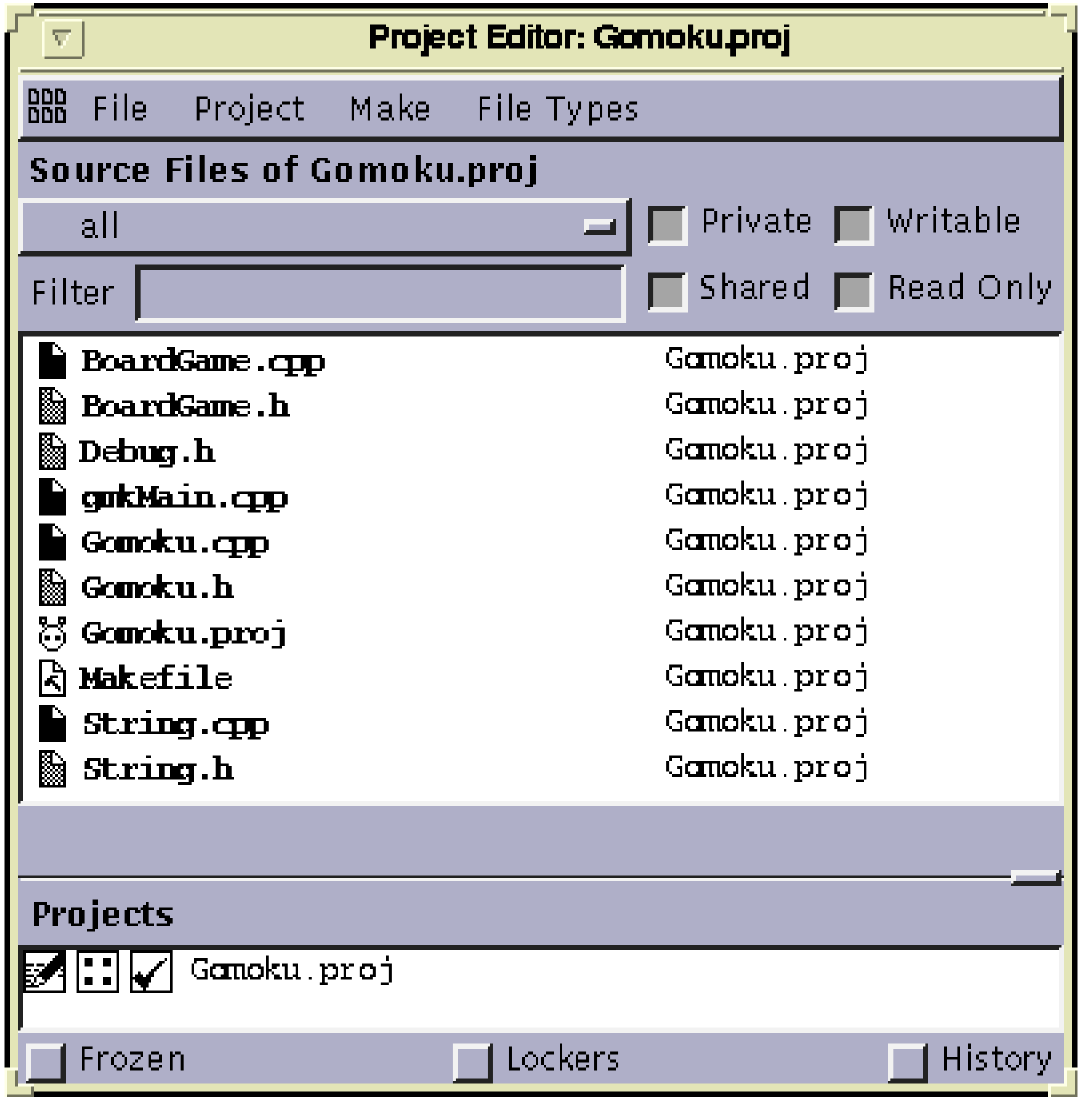
SNiFF+ is an integrated environment for C++ development:

- ❑ project management
- ❑ hierarchy browser
- ❑ class browser
- ❑ symbol browser
- ❑ cross referencer
- ❑ source code editor (using emacs, etc.)
- ❑ version control with RCS
- ❑ compiler error parsing (g++)
- ❑ integrated make facility

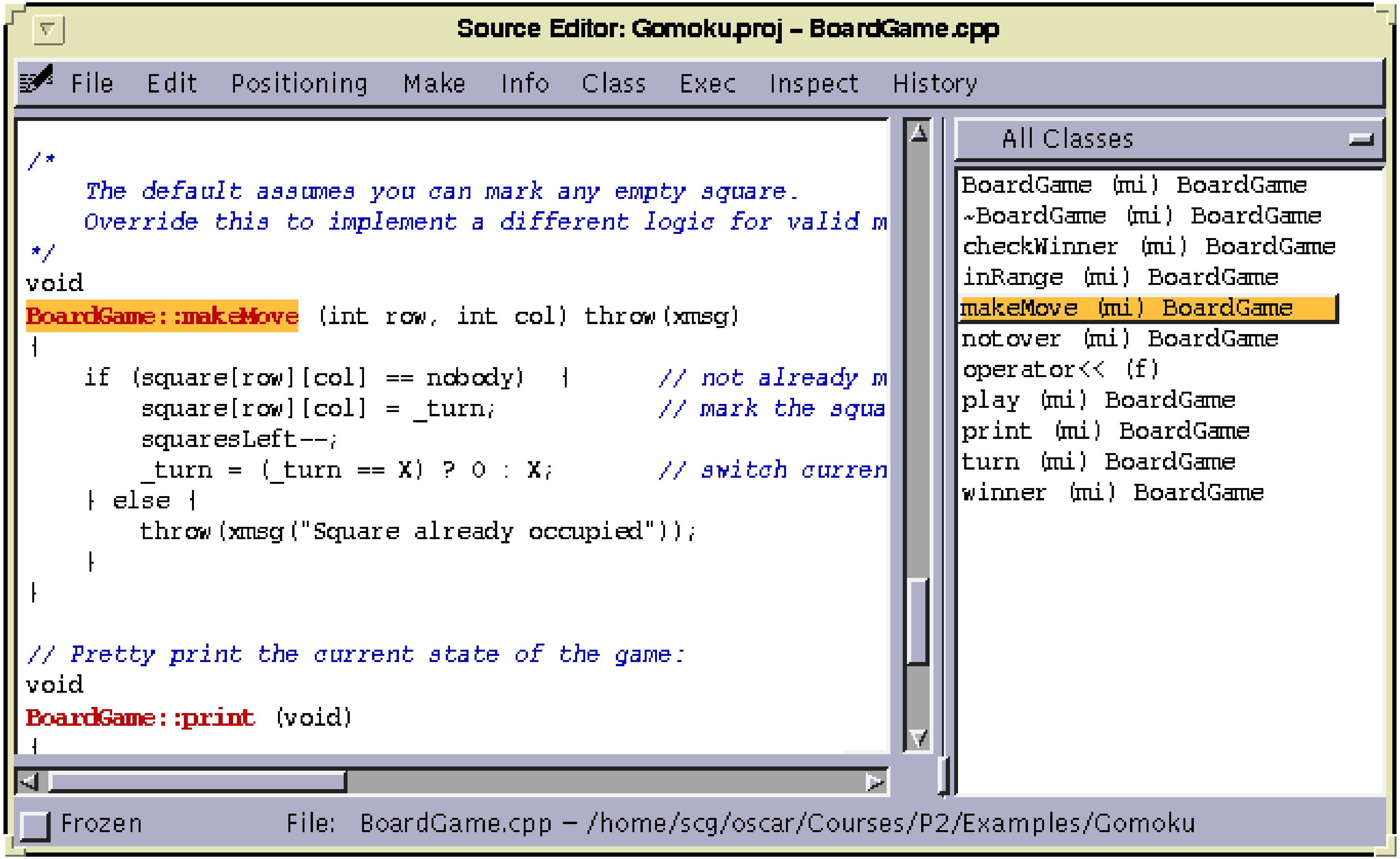
✔ *Always use an integrated programming environment if one is available!*

Using SNiFF+

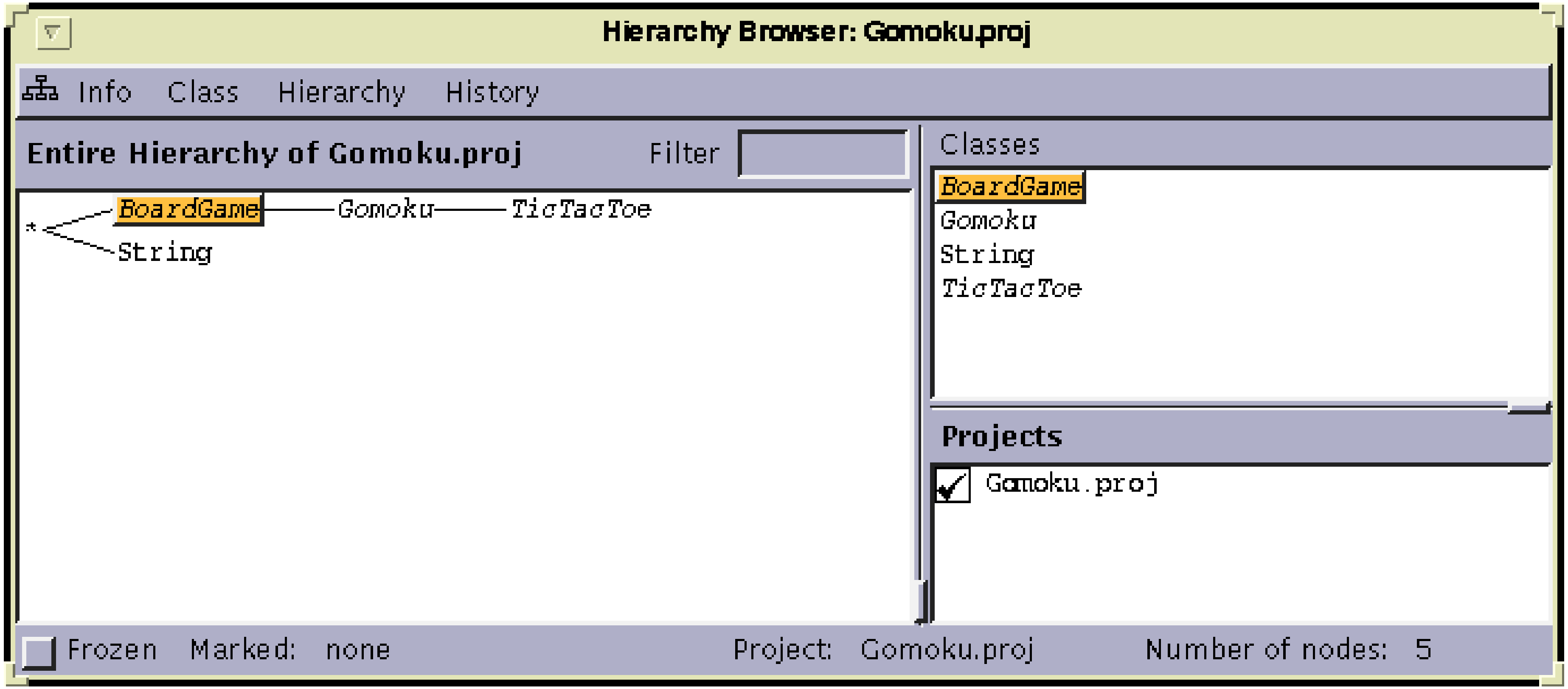

SNiFF+ – oscar@pogo
Project Windows
Loaded Projects
Gomoku.proj


Project Editor: Gomoku.proj
File Project Make File Types
Source Files of Gomoku.proj
all
Private
Writable
Filter
Shared
Read Only
BoardGame.cpp Gomoku.proj
BoardGame.h Gomoku.proj
Debug.h Gomoku.proj
gmkMain.cpp Gomoku.proj
Gomoku.cpp Gomoku.proj
Gomoku.h Gomoku.proj
Gomoku.proj Gomoku.proj
Makefile Gomoku.proj
String.cpp Gomoku.proj
String.h Gomoku.proj
Projects
Gomoku.proj
Frozen
Lockers
History

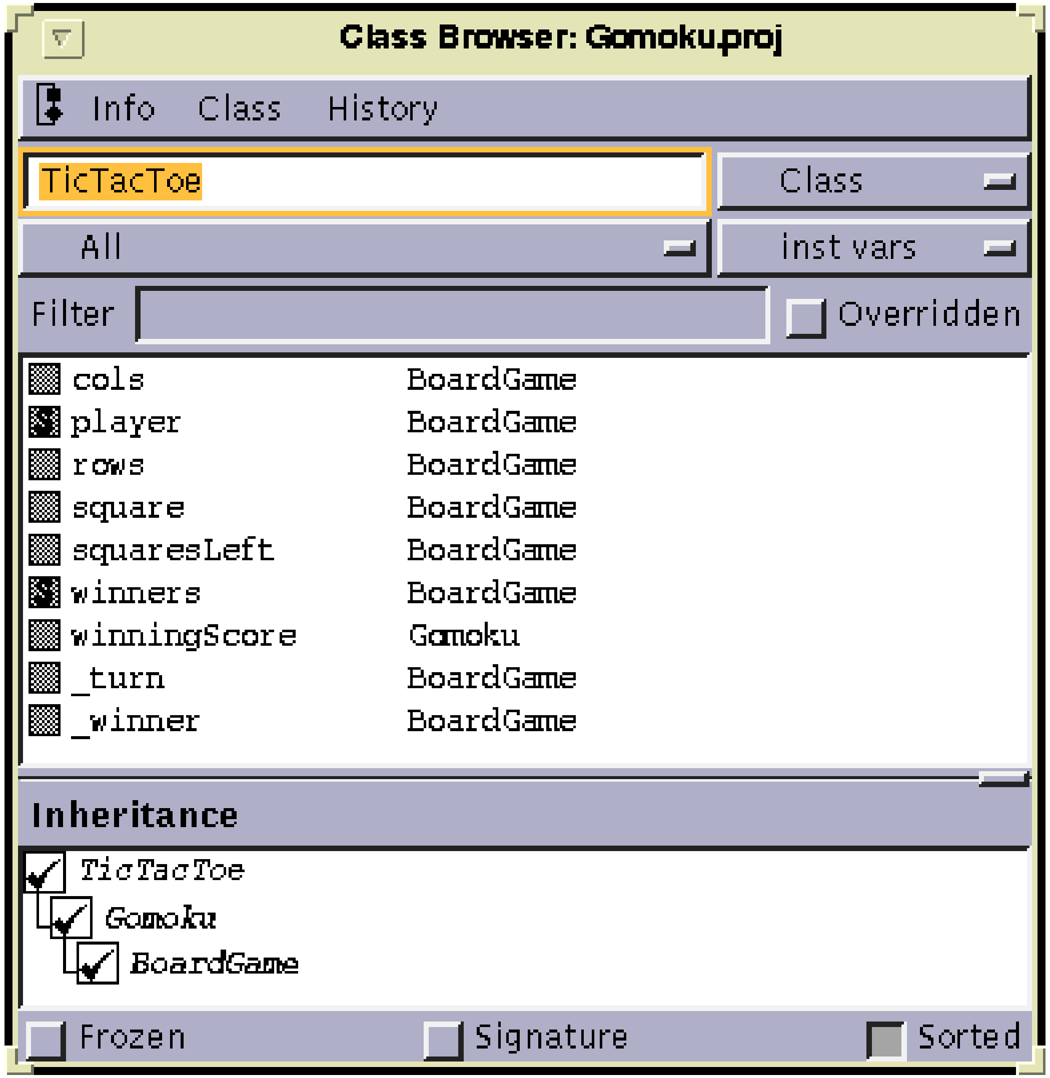
SNiFF+ Source Editor



SNiFF+ Hierarchy Browser



SNiFF+ Class Browser



Purify

Purify is tool to help detect run-time memory corruption and memory leaks

- ❑ Add `purify` to the link line in your Makefile, e.g.:

```
gomoku : ${GMKO}
   purify CC ${GMKO} ${LFLAGS} -o $@
```

  Purify will modify the object code at link time to add error-checking instructions.

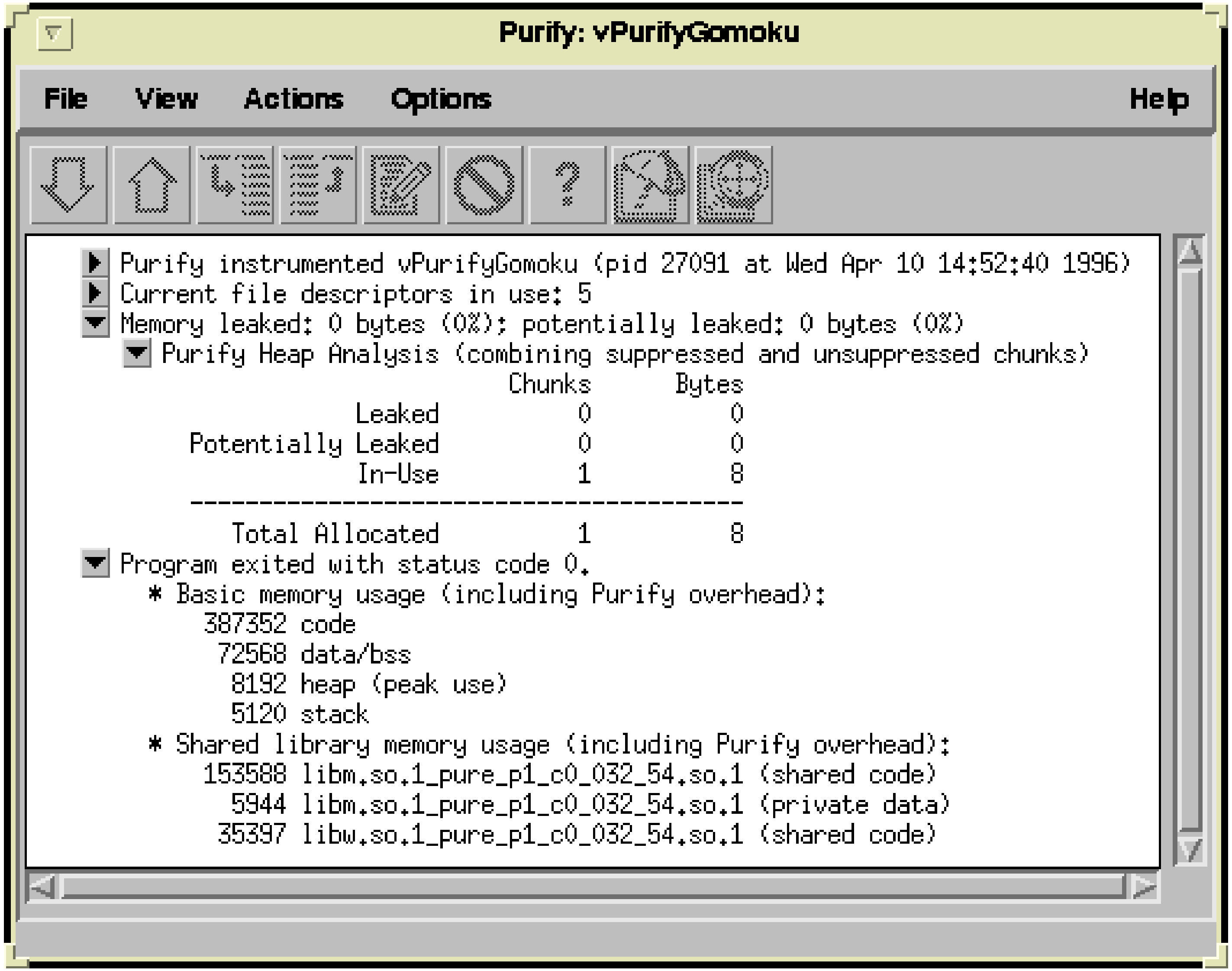
- ❑ Run your program as usual — a special window will open with error messages displayed as various abnormal conditions are detected
- ❑ Your program will (almost always) run exactly as it does without purify, except it will be about 3 to 5 times slower, and take about 40% more memory

✔ *Use purify (or an equivalent utility) while developing C++ programs to catch errors in managing memory.*

Remember, the most common C++ errors are invalid memory accesses!

Purify is a product of Pure Software Inc.

Using Purify



Other tools

Be familiar with the programming tools in your environment!

- **lint**: detect bugs, portability problems and other possible errors in C programs
- **strip**: remove symbol table and other non-essential data from object files
- **diff** and **patch**: compare versions of files, and generate/apply deltas
- **lex** and **yacc** [flex and bison]: generate lexical analysers and parsers from regular expression and context-free grammar specification files
- **awk**, **sed** and **perl**: process text files according to editing scripts/programs
- **tar**: stores files and directories as a "tape archive"
- **compress** and **uncompress** [gzip and gunzip]: compress files

Summary

You should know the answers to these questions:

- ❑ How are makefiles specified?
- ❑ What functionality does a version control system support?
- ❑ What are breakpoints? Where should you set them?
- ❑ When should you use a profiler?
- ❑ How can you catch memory leaks and invalid memory accesses?

Can you answer the following questions?

- ✎ *How can you force make to recompile programs even if they are not out-of-date?*
- ✎ *When should you specify a version of your project as a new "release"?*
- ✎ *When should you use a debugger instead of adding "print" statements to your program?*
- ✎ *When should you "strip" an executable program?*

10. Design Rules

- Using `new` and `delete`
- Initialization lists vs. assignment in constructors
- Virtual destructors
- Assignment and inheritance
- Class members, globals and friends
- `const` declarations
- References vs. values
- Overloading

Sources:

- Scott Meyers, *Effective C++*, Addison-Wesley, 1992.

Basic Rules

✔ *Use* `const` *and* `inline` *instead of* `#define`

- ☞ Constants are named and understood by the compiler; macros aren't
- ☞ Inline functions evaluate arguments once; macros are expanded literally

Recall the problems with the `badMin()` macro!

✔ *Prefer* `iostream.h` *to* `stdio.h`

- ☞ `scanf` and `printf` are not typesafe and not extensible

Deleting Objects

✔ *Use the same form in corresponding calls to* `new` *and* `delete`

- ☞ Delete objects with `delete`
- ☞ Delete arrays with `delete []`

If you try to delete an array with `delete`, you will only delete the first element!

✔ *Call* `delete` *on pointer members in destructors*

If your class has a pointer member, make sure that:

- ❑ The pointer is properly initialized within each constructor
  - ☞ If no memory is allocated, initialize the pointer to 0 (null)
- ❑ Existing memory is deleted and new memory assigned to the pointer in the assignment operator (i.e., operator=)
- ❑ Allocated memory is deleted in the destructor

  (NB: it is always safe to call delete on a null pointer)

Normally a class should *not* delete objects it did not create!

Running out of Memory

✔ *Check the return value of* `new`

When `new` cannot allocate the memory you need, it returns 0.

Alternatively, you can tell `new` to call an error handler that you supply:

```
void noMoreMemory(void)
{
   cerr << "Ran out of memory!" << endl;
   exit(1);
}
void memTest(void)
{
   set_new_handler(noMoreMemory);             // NB: #include <new.h>
   char *wayTooBig = new char[1000000000];    // Will cause new to call
}                                             // noMoreMemory()
```

Since `set_new_handler()` always returns the current handler, you can also locally set and restore handlers within classes.

Constructors

✔ *Define a copy constructor and an assignment operator for classes with dynamically allocated memory*

Use the orthodox canonical form — if you don't, C++ will silently generate for you copy constructors and assignment operators that perform shallow copies!

✔ *Prefer initialization to assignment in constructors*

```
class MyClass
{
   public:
      MyClass (const String& name);
   private:
      String myName;
};
```

```
MyClass::MyClass (const String& name)
   : myName(name)// initialization
{ }

MyClass::MyClass (const String& name)
{
   myName = name; //assignment
}
```

- ❑ Assignment adds overhead, since members must be first initialized and then assigned to
- ❑ `const` and reference members can only be initialized, never assigned!
- ❑ Use assignment only for algorithmic initialization (e.g., of arrays)

Initialization

✔ *List members in an initialization list in the order in which they are declared*

Class members are initialized in the order they are declared, not in the order they appear in the initialization list!

```
class Rectangle
{
   public:
      Rectangle (int initWidth);            // Construct as a square
      int width (void) { return w; }
      int height (void) { return h; }
   private:
      int h, w;                             // First construct h, then w
};


Rectangle::Rectangle (int initWidth)
   :  w(initWidth),
      h(w)                                  // WRONG! w is still undefined
{ }
```

Why? Because destructors destroy members in the reverse order they were constructed, so all constructors must create them in a consistent order ...

Virtual Destructors

✔ *Make destructors virtual in base classes*

If you make use of polymorphism, the only way you can be sure the correct destructor is called when an object is deleted is if the destructor is virtual in the base class.

Recall the polymorphic destruction of `BoardGame` *instances.*

But ... don't declare destructors virtual in classes that will never be inherited from!

Assignment

✔ *Have* `operator=` *return a reference to* `*this`

The result should be a reference to the object itself, so you can write statements like:

```
a = b = c ;
```

for arbitrary classes of objects.

✔ *Check for assignment to self in* `operator=`

Recall what would happen if our String class failed to check for this:

```
String&
String::operator= (const String& copy)
{
   if (this != &copy) {      // copying self would lead to an inconsistent state!
      delete [] _s;          // be sure to delete the previous value!
      become(copy._s);       // (re-)initialization is the same as before
   }
   return *this;             // return a reference, not a copy!
}
```

Assignment and Inheritance

✔ *Assign to all data members in* `operator=`

If a derived class does not have access to data members of the base class, it may be necessary to explicitly call operator= of the base class

```
class A
{
    public:
        A(int initVal) : x_(initVal) { } ;
        A& operator=(const A& rhs);
        int x(void) { return x_; }
    private:
        int x_;
};
```

```
class B : public A
{
    public:
        B(int initVal) : A(initVal), y_(initVal) { } ;
        B& operator=(const B& rhs);
        int y(void) { return y_; }
    private:
        int y_;
};
```

```
B& B::operator=(const B& rhs)
{
    if (this != &rhs) {
        y_ = rhs.y_;                // not enough -- need to also assign to (hidden) x_
        // x_ = rhs.x_;             // illegal access to private member!
        A::operator=(rhs);          // ok call to base operator=
        // ((A&) *this) = rhs;      // also ok, but more obscure ...
    }
    return *this;
}
```

Classes and Functions

✔ *Differentiate among member functions, global functions and friend functions*

```
// virtual functions must be members

if (f needs to be virtual)                           // e.g., BoardGame::checkWinner()
   make f a member function of C;

// operator>> and operator<< are never members

else if (f is operator>> or operator<<) {
   make f a global function                          // target is iostream
   if (f needs access to non-public members of C)
      make f a friend of C;
}

// only nonmembers can have type conversions on their left-hand argument

else if (f needs type conversions on its lhs) {  // e.g., "foo" + String("bar")
   make f a global function;
   if (f needs access to non-public members of C)
      make f a friend of C;
}

// everything else should be a member function

else
   make f a member function of C;
```

Class Interfaces

✔ *Avoid data members in the public interface*

- ❑ Clients don't have to remember whether to accessing members with or without parentheses (e.g., `p.x` vs. `p.x()`)
- ❑ You have more freedom to alter the implementation of your class without affecting clients

✔ *Use* `const` *wherever possible*

You can declare values, pointers, function arguments, return values and member functions as const; the compiler will ensure consistent usage of constant values.

How to declare `const` pointers:

What's pointed to is constant		*Pointer is constant*	
	`char *`		`p = "Hello";`
`const`	`char *`		`p = "Hello";`
	`char *`	`const`	`p = "Hello";`
`const`	`char *`	`const`	`p = "Hello";`

References and Values

✔ *Pass and return objects by reference instead of by value*

In C++ everything is passed by value. If you pass or return objects by value, the copy constructor will be called to create copies for every argument and return value.

✔ *Don't try to return a reference when you must return an object*

If a function creates a new object value from its arguments, then the result should be returned by value, not by reference.

Consider the global function:

```
String operator+ (const String& s1,const String& s2)
{
   String result = s1;       // call the String copy constructor
   return result += s2;      // return a copy of the result
}
```

It cannot return a reference since the result is not an existing object.

It also should *not* call `new` since the client cannot be expected to call `delete`!

Data Accessibility

✔ *Never return a reference to a local object or a dereferenced pointer initialized by* `new` *within the function*

Two bad ways to implement String concatenation:

```
String& operator+ (const String& s1,const String& s2)
{
   String result = s1;
   result += s2;
   return result;          // WRONG!!! never return a reference to a local object
}                          // result will be destroyed when function returns!

String& operator+ (const String& s1,const String& s2)
{
   String * result = new String(s1);
   *result += s2;
   return *result;         // Potential memory leak!!! Who will delete result?
}
```

✔ *Avoid member functions that return pointers or references to members less accessible than themselves*

Don't return non-`const` references or pointers to private data from public functions.

Const Member Functions

✔ *Avoid returning "handles" to internal data from* `const` *member functions*

If an object is declared `const`, then all its `const` member functions should be safe.

But if these functions may return non-`const` pointers to private data, the "constant" object may be modified by unexpected side effects:

```
char&
String::operator[] (const int n) const          // safe to use on const Strings
{
   if ((n<0) || (strlen()<=n))
      throw(xmsg("array index out of bounds"));
   return _s[n];                                // oops -- returns a reference!
}
```

Now the following code is unsafe:

```
const String cs = "I'm constant";
// cs = "hello world";                          // illegal implicit pointer cast
cout << "First char is: " << cs[0] << endl;     // ok -- operator[] is const
cs[0] = 'A';                                    // oops -- we just changed cs!
```

Overloading vs. Default Parameters

✔ *Choose carefully between function overloading and parameter defaulting*

```
void f(void);
void f(int x);          // f is overloaded
f();                    // calls f(void)
f(10);                  // calls f(int)
void g(int x=0);        // g has a default parameter
g();                    // calls g(0)
g(10);                  // calls g(10)
```

So, what's the difference?

Ask yourself:

- ❏ Is there a sensible default parameter?
- ❏ Is there a common algorithm?

Unless the answer to both of these questions is "yes", you should probably declare overloaded functions rather than default parameters.

Ambiguous Overloading

✔ *Avoid overloading on a pointer and a numerical type*

```
void f(int x);
void f(char * p);
f(0);                    // calls f(int) or f(char*)?
```

Since `0` is a literal integer constant, `f(int)` will be called, but this is not always what you want!

Common Errors

Watch out for these common errors:

- ❑ Forgetting to end a class declaration with a semi-colon
  - ☞ the compiler will generate non-intuitive errors concerning the code immediately following the class declaration
- ❑ Forgetting parentheses when calling class members (e.g., game.notover())
  - ☞ the function will never be called, but instead the value of the function pointer will be used

Summary

You should know the answers to these questions:

- Where and when should you use `new` and `delete`?
- When should you (not) use initialization lists in constructors?
- How should you define `operator=`?
- How can you update private inherited data members in a derived class?
- When should a function be global rather than a class member?
- When should you use `const` declarations?
- When should a function return a reference? A value?

Can you answer the following questions?

- *How does* `delete[]` *know how many items to destroy?*
- *Why can't you initialize references by assignment?*
- *Why shouldn't you always declare destructors* `virtual`*?*
- *Why should* `operator=` *return* `*this` *instead of simply* `this`*?*
- *What will happen if you return a reference to an automatic variable?*

11. Templates

- Function Templates
- Class Templates
  - Implementing templates: a resizeable Array template class
  - Using Templates
  - Reusing Templates: Stacks and Matrices
- Templates and Inheritance
- Developing Templates
- Templates vs. other abstraction mechanisms

What are Templates?

A template is a generic specification of a function or a class, parameterized by one or more types used within the function or class.

- ❑ Use templates whenever the parameters do not inherit from a common parent
  - ☞ functions that only assume basic operations of their arguments (comparison, assignment ...)
  - ☞ "container classes" that do little else but hold instances of other classes

- ❑ Templates are essentially glorified macros
  - ☞ like macros, they are compiled only when instantiated (and so are defined exclusively in header files)
  - ☞ unlike macros, templates are not expanded literally, but may be intelligently processed by the C++ compiler

Function Templates

Templates are preceded by the declaration:

```
template <class Type1, class Type2 ...>
```

for as many parameter types as needed.

The following declares a generic `min()` function that will work for arbitrary, comparable elements:

```
template <class Item>
inline const Item&
min (const Item& a, const Item& b)
{
   return (a<b) ? a : b;
}
```

Templates are automatically instantiated by need:

```
cout << "min(3,5) = " << min(3,5) << endl;
// automatically instantiates: inline const int& min(int&, int&);
cout << "min('a','c') = "<< min('a','c') << endl;
// automatically instantiates: inline const char& min(char&, char&);
```

Templates vs. Macros

Macros are a poor substitute for template functions:

```
#define badMin(a,b) ((a<b) ? a : b)
```

A function template works correctly with expressions as arguments:

```
int a, b;
cout << "Enter two numbers:";
cin >> a >> b;
cout << "min(" << a << "," << b << ") = " << min(a,b) << endl;
cout << "min(++" << a << ",++" << b << ") = ";
cout << min(++a,++b) << endl;           // works like a call-by-value function
```

But a macro is expanded literally:

```
cout << "badMin(" << a << "," << b << ") = " << badMin(a,b) << endl;
cout << "badMin(++" << a << ",++" << b << ") = ";
cout << badMin(++a,++b) << endl;        // don't get what we expect!
```

This last statement expands to:

```
cout << ((++a<++b) ? ++a : ++b) << endl;// a and b are incremented twice!
```

Class Templates

Class templates are declared just like function templates:

- A template class `MyClass` can be declared with a formal parameter:

```
template <class Param>
class MyClass { ... }
```

- Every usage of `MyClass` as a class name within the declarations and definitions must be *bound* to the formal parameter `Param`:

```
MyClass(const MyClass<Param>&); // declaration of copy constructor
```

- Every member function of a template class is a template function:

```
template <class Param>
MyClass<Param>::MyClass (const MyClass<Param>& copy)
{
   // definition of copy constructor ...
}
```

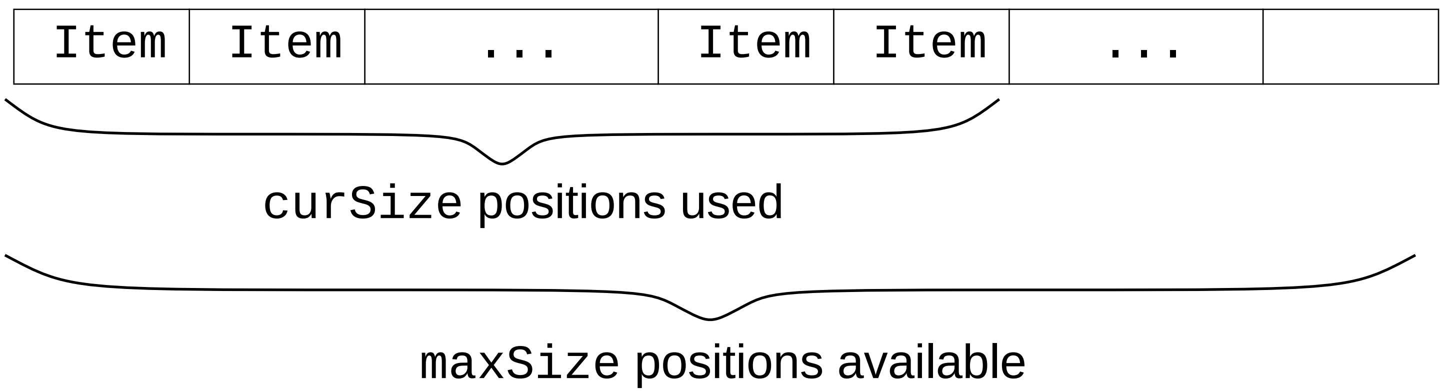
- Template classes are instantiated by binding the formal parameter:

```
MyClass<String> myObject;
```

Example: A Resizeable Array

We would like a generic Array abstraction that will:

- hold arbitrary kinds of elements
- support indexing, copying and assignment
- protect us from invalid indices
- dynamically grow or shrink upon `resize` requests



Idea: an `Array` instance can hold up to `maxSize` elements

If it needs to grow beyond this size, we can transparently allocate and a larger buffer and copy the old values. [Compare with our earlier `String` class]

The Array Interface

NB: both declarations *and* definitions are in Array.h

```
template <class Item>
class Array
{
public:
   Array (int initSize=0);                                  // default constructor; optional size
   Array (int initSize, const Item& initVal);               // initialize all values to initVal
   Array (const Array<Item>& copy);                         // copy constructor
   Array (int size, const Item* items);                     // initialize from C array
   ~Array (void);                                           // destructor
   Array<Item>& operator= (const Array<Item>& val)          // assignment
      throw(xmsg);
   Item& operator[] (const int n) throw(xmsg) const;        // index
   int size (void) const { return curSize; }                // current size
   void resize (int newSize) throw(xmsg);                   // change current size
protected:
   int curSize;                                             // current size
   int maxSize;                                             // maximum allocated size
   Item * arrayRep;                                         // pointer to allocated array
   // Helper functions for constructors and for resize()
   Item * makeArray(int n);                                 // make a new array
   void become(int initSize, int n, const Item* values);// become a new array of size n
};
```

Array Constructors

```
template <class Item>
Array<Item>::Array (int initSize) :                      // default constructor
   curSize(initSize),
   maxSize(initSize),
   arrayRep(makeArray(maxSize))                          // use default Item constructor
{
}

template <class Item>
Array<Item>::Array (int initSize, const Item& initVal) : // initializing constructor
   curSize(initSize),
   maxSize(initSize),
   arrayRep(makeArray(maxSize))
{
   for (i=0; i<curSize; i++)
      arrayRep[i] = initVal;
}

template <class Item>
Array<Item>::Array (const Array<Item>& copy) :           // copy constructor
   arrayRep(0)                                           // precondition for become()
{
   become(copy.maxSize, copy.curSize, copy.arrayRep);    // initialize state from copy
}

template <class Item>
Array<Item>::Array (int size, const Item* items) :       // array constructor
   arrayRep(0)
{
   become(size, size, items);                            // initialize from C array
}
```

Array Copy Functions

```
// requires: arrayRep is initialized to some array (or to 0)
// ensures:  class invariant (valid arrayRep, curSize and maxSize)
template <class Item>
void Array<Item>::become (int availSize, int newSize, const Item* values)
{
   if (newSize > 0 && values == 0)                // private function, so should never happen!
      throw(xmsg("Array::become: newSize>0 but values=0!"));
   if (availSize < newSize)                       // we assume availSize >= 0
      availSize = newSize;                        // sanity check
   Item * oldRep = arrayRep;                      // save just in case values overlaps arrayRep!
   arrayRep = makeArray(availSize);               // NB: this might fail
   maxSize = availSize;                           // now it is safe to update the size
   curSize = newSize;
   for (int i=0; i<newSize; i++)
      arrayRep[i] = values[i];
   delete [] oldRep;                              // don't forget to delete what you create!!!
}
template <class Item>
Item * Array<Item>::makeArray (int size)          // throws exception if new fails
{
   if (size < 0)
      throw(xmsg("Array::makeArray: Can't allocate negative sized array!"));
   Item * newArray = new Item[size];              // NB: will call Item constructor size times
   if (newArray == 0)
      throw(xmsg("Array::makeArray: Couldn't allocate enough space"));
   return newArray;
}
```

Resizing an Array

```
template <class Item>
void
Array<Item>::resize (int newSize)
{
    if (newSize < 0) {
        throw(xmsg("Array::resize: cannot take negative size"));

    if (newSize <= maxSize) {
        // newSize is smaller than maxSize, so we have plenty of room ...
        // but if we are too small now, we should reclaim space
        if (maxSize > 4*newSize) {
            become(newSize, newSize, arrayRep);                 // really shrink
        } else {
            curSize = newSize;                                  // just change logical size
        }
        return;
    } else {
        // newSize > maxSize, so we need to become larger
        int newMax = (maxSize>0) ? maxSize : 2;                 // start with some positive size
        while (newSize > newMax) {
            newMax = 2*newMax;                                  // select newMax >= newSize
        }

        become(newMax, curSize, arrayRep);                      // copy current state
        curSize = newSize;                                      // and now we have room to resize
        return;
    }
}
```

Completing the Array

The remaining member functions are straightforward:

```
template <class Item>
Array<Item>::~Array (void)                                   // destructor
{
   delete [] arrayRep;
}


template <class Item>
Item&
Array<Item>::operator[] (const int n) const                  // index -- doesn't modify state
{
   if ((n<0) || (n >= curSize))
      throw(xmsg("Array::operator[]: index out of range"));
   return arrayRep[n];                                       // NB: result can be modified!
}


template <class Item>
Array<Item>&
Array<Item>::operator= (const Array<Item>& val)              // assignment
{
   if (this != &val)                                         // NB: always check for this!
      become(val.maxSize, val.curSize, val.arrayRep);
   return *this;                                             // NB: not just "return this"!
}
```

Instantiating and Using the Array

Array can be instantiated with arbitrary argument types

☞ `Array<Item>` will be freshly compiled for each unique value of `Item`

```
Array<int> a(4);                    // new uninitialized int Array of length 4
for (int i=0; i<a.size(); i++)      // safe access and update
   a[i] = i*i;                      // NB: an updatable reference is returned
a.resize(10);                       // a grows to length 10
for (i=0;i<a.size();i++)
   a[i] = i*i;
a.resize(3);                        // and shrinks to length 3
Array<int> b(a);                    // copy constructor is used to initialize b from a
Array<int> d;                       // new int Array of length 0
d = a;                              // d becomes a copy of a by assignment
Array<char> c(5);                   // new char Array of length 5
for (i=0; i<c.size(); i++)
   c[i] = 'a' + i;
int primes[5] = { 2, 3, 5, 7, 11 }; // an ordinary C array
Array<int> e(5,primes);             // a new int Array is constructed and initialized
printArray(cout, "e", e);
Array<String> sa(2);                // a new String Array is created (default String constructor)
sa[0] = "hello";                    // new String is constructed and assigned to sa[0]
cout << "Enter your name:";
sa[1].getline(cin);                 // sa[1] dynamically grows from input
```

Implementing a Generic Stack

Template classes can be implemented using other templates

A generic Stack can easily be implemented using a resizeable Array:

```
template <class Item>
class AStack
{
   public:
      AStack (void) : stack(0) { } ;
      ~AStack (void) { } ;

      // inline functions:
      int count (void) { return stack.size(); };
      int empty (void) { return stack.size() == 0; };

      void push (Item& item) throw(xmsg);
      Item& top (void) throw(xmsg) { return stack[stack.size() - 1]; }
      void pop (void) throw(xmsg) { stack.resize(stack.size() - 1); }

   private:
      Array<Item> stack;
};

template <class Item>
void
AStack<Item>::push (Item& item)
{
   stack.resize(stack.size() + 1);
   top() = item;                                    // NB: top returns an updatable reference
}
```

Reimplementing the Line Reverser

```
#include <iostream.h>
#include <exception.h>

#include "AStack.h"
#include "String.h"

typedef AStack<String> IOStack;                  // assign a name to our template type

int main (void)
{
    IOStack ioStack;                             // a new zero-length IOStack
    String buf;                                  // and a zero-length String

    try {
        while (buf.getline(cin)) {               // initialize buf from cin
            ioStack.push(buf);                   // and copy buf to the top of ioStack
        }

        while (ioStack.count() != 0) {
            cout << ioStack.top() << endl;
            ioStack.pop();                       // top will get deleted when ioStack shrinks
        }
    }
    catch (xmsg &err) {
        cout << "Exception: " << err.why() << endl;
        return -1;
    }
    return 0;
}                                                // buf, ioStack and ioStack strings are deleted
```

Matrices as Arrays of Arrays

Another example: a generic Matrix can be implemented as an Array of generic Arrays:

```
template <class Item>
class Matrix
{
public:
   Matrix (int rows=0, int cols=0);                         // need new constructor
   Matrix (int rows, int cols, const Item& initVal);        // and another with initial value

   // the automatically generated copy constructor and operator= are adequate ?

   ~Matrix (void) { } ;                                     // destructor has nothing extra to do!

   Array<Item>& operator[] (const int n)                    // most operators are inline
      throw(xmsg) const { return matrixRep[n]; }

   int rows (void) const { return matrixRep.size(); }
   int cols (void) const { return matrixRep[0].size(); }

   void resize (int rows, int cols) throw(xmsg);            // this is the only interesting one

protected:
   Array<Array<Item> > matrixRep;                           // NB: space is significant!
};
```

Implementing the Matrix Operations

```
template <class Item>
Matrix<Item>::Matrix (int rows, int cols)                      // default constructor
{
   this->resize(rows, cols);                                   // make an uninitialized Matrix
}

template <class Item>
Matrix<Item>::Matrix (int rows, int cols, const Item& initVal) // initializing constructor
{
   this->resize(rows, cols);
   for (int r=0; r<rows; r++)
      for (int c=0; c<cols; c++)
         matrixRep[r][c] = initVal;                            // initialize all to initVal
}
template <class Item>
void
Matrix<Item>::resize (int rows, int cols)                      // resize rows and columns
{
   matrixRep.resize(rows);
   for (int r=0;r<rows;r++)
      matrixRep[r].resize(cols);
}
```

Gomoku using Matrix for the Board

We can now use our `Matrix` class to simplify the `BoardGame` class:

```
class BoardGame {
public :
   // as before ...
protected :
   // as before, except:
   Matrix<Player> square;                    // instead of: Player ** square;
};
```

The `BoardGame` constructor is similarly simplified:

```
BoardGame::BoardGame (int rs, int cs) throw(xmsg) :
   rows(rs), cols(cs),
   _winner(nobody),
   _turn(X),
   squaresLeft(rs*cs),
   square(rs,cs,nobody)                      // initialize all squares to nobody
{
}
```

Template Inheritance

Template classes can inherit from other template classes:

```
template <class Item>
class MyArray : public Array<Item>        // I.e., every MyArray<Item> inherits from Array<Item>
{
public:
   // default constructors, destructors, operator=, will be generated ... is this ok?!
   MyArray (const Array<Item>& copy) : Array<Item>(copy) { };  // construct from Array instance
   void print (void);
};

template <class Item>
void MyArray<Item>::print (void)          // pretty print array contents
{
   cout << "<" << this->size() << "> { ";  // print <size> ...
   for (int i=0; i<this->size(); i++)
      cout << (*this)[i] << " ";          // ... followed by { elements ... }
   cout << "} " << endl;
};
```

Sample usage:

```
void
printArray (Array<int> a)
{
   MyArray<int> mine(a);                  // construct a MyArray instance from an Array
   mine.print();                          // and print it using the derived print method
}
```

Developing Templates

Templates can be hard to debug since the code to compile is generated by need.

✔ *Use typedef declarations to prototype your generic classes before promoting them to template classes.*

```
typedef char* Item;                              // Test with different values of Item

class MyStack
{
public:
   DStack(void);
   ~DStack(void);

   int count(void) { return size; };
   int empty(void) { return size == 0; };

   void push(Item item) throw();
   Item top(void) throw(xmsg);
   void pop(void) throw(xmsg);

private:
   Cell *topCell;
   int size;
};
```

Start with concrete member function definitions in a source (.cpp) file, before converting them to template functions in the header file.

Forms of Reuse

C++ provides various mechanisms for factoring out common abstractions:

Abstraction	*Common*	*Variable*
Macros	syntax	syntactic parameter
Global Variables	state	client requests
Functions	algorithm	value parameter
Function pointers	algorithm	function parameter
Templates	function, class	type parameter
Classes	interface representation methods	state
Inheritance	interface methods	extended or overridden interface and methods
Virtual members	polymorphic clients and inherited methods	methods (glorified function pointers)

Summary

You should know the answers to these questions:

- ❑ How do you specify template functions and classes?
- ❑ Why are templates preferable to macros?
- ❑ Why are templates defined only in header files?
- ❑ How can you compose template classes from other templates?
- ❑ How should you use `typedef` declarations together with templates?
- ❑ When and how can you combine templates and inheritance?

Can you answer the following questions?

- ✎ *When should you use templates?*
- ✎ *Why aren't templates compiled until their parameters are bound?*
- ✎ *Are there situations where you might use either templates or inheritance to solve the same problem?*

12. The Standard Template Library

- STL Overview
- Example: STL line reverser
- Containers and Iterators — managing and traversing lists
- Generic Algorithms — parameterized by Containers and object types
- Function Objects — wrapping functions as objects
- Adaptors — altering interfaces to promote reuse

Sources:

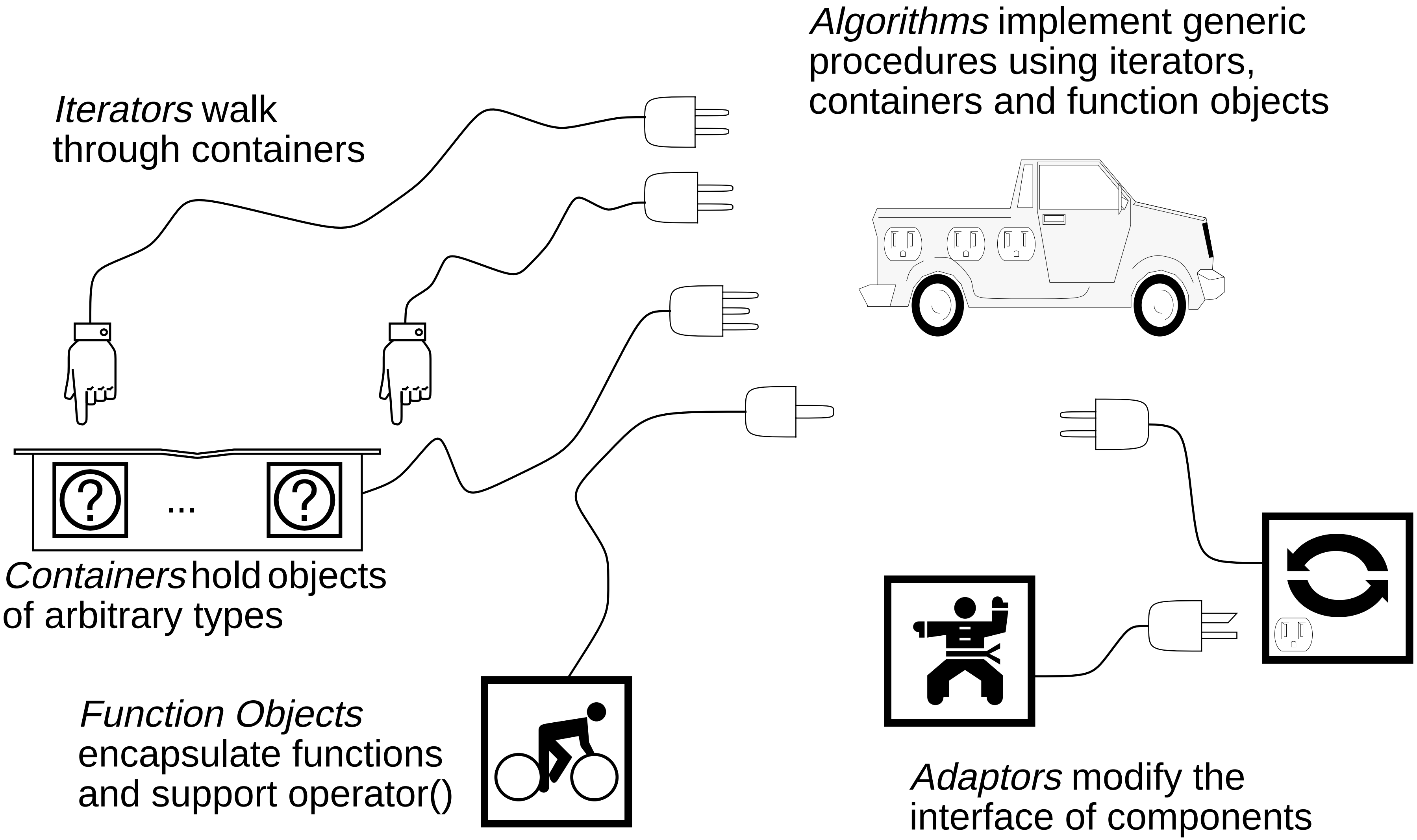
- Alexander Stepanov and Meng Lee, “The Standard Template Library,” Hewlett-Packard Laboratories, 1995
- David Musser and Atul Saini, *STL Tutorial and Reference Guide*, Addison-Wesley, 1996

What is STL?

STL is a general-purpose C++ library of generic algorithms and data structures.

- Generic:
  - All algorithms and data structures are parameterized (templates)
- Transparent:
  - Most of STL will work both with STL ADTs as well as with C++ primitive types (i.e., arrays and pointers)
- Efficient:
  - Assumptions, costs and complexity are part of components' contracts
  - Heavy use of in-line functions
- Composable:
  - Components are designed to be combined as easily as possible
- Extendible:
  - You can add your own components that can be combined with standard STL components
  - Existing components can be integrated by means of adaptors

Plug Compatibility in STL



STL Components

STL contains five main kinds of components:

1. *Containers* store collections of objects
   - ☞ `vector`, `list`, `deque`, `set`, `multiset`, `map`, `multimap`
2. *Iterators* traverse containers
   - ☞ random access, bidirectional, forward/backward, input/output
3. *Function Objects* encapsulate functions as objects
   - ☞ arithmetic, comparison, logical, and user-defined ...
4. *Algorithms* implement generic procedures
   - ☞ `search`, `count`, `copy`, `random_shuffle`, `sort`, `permute` ...
5. *Adaptors* provide an alternative interface to a component
   - ☞ `stack`, `queue`, `reverse_iterator`, ...

Example: STL Line Reverser

```
#include <iostream.h>
#include <vector.h>                                    // use vector template
#include <stack.h>                                     // use stack container adaptor

#include "String.h"

void rev(void);

int main(void)
{
   rev();
   return 0;
}

void rev(void)
{
   typedef stack<vector<String> > IOStack ;// container adaptor (stack) with container (vector)

   IOStack ioStack;
   String buf;

   while (cin >> buf) {
      ioStack.push(buf);                               // push is translated by stack to vector::push_back
   }

   while (ioStack.size() != 0) {
      cout << ioStack.top() << endl;
      ioStack.pop();
   }
}
```

Containers

Vector

- constant time random access; constant time insertions/deletions at end

Deque

- constant time random access; constant time insertion/deletion at either end

List

- bidirectional iterators; constant time insertion anywhere

Set

- “fast retrieval” of unique keys (must support comparison)

Multiset

- fast retrieval of multiple keys

Map

- fast retrieval of unique values by keys

Multimap

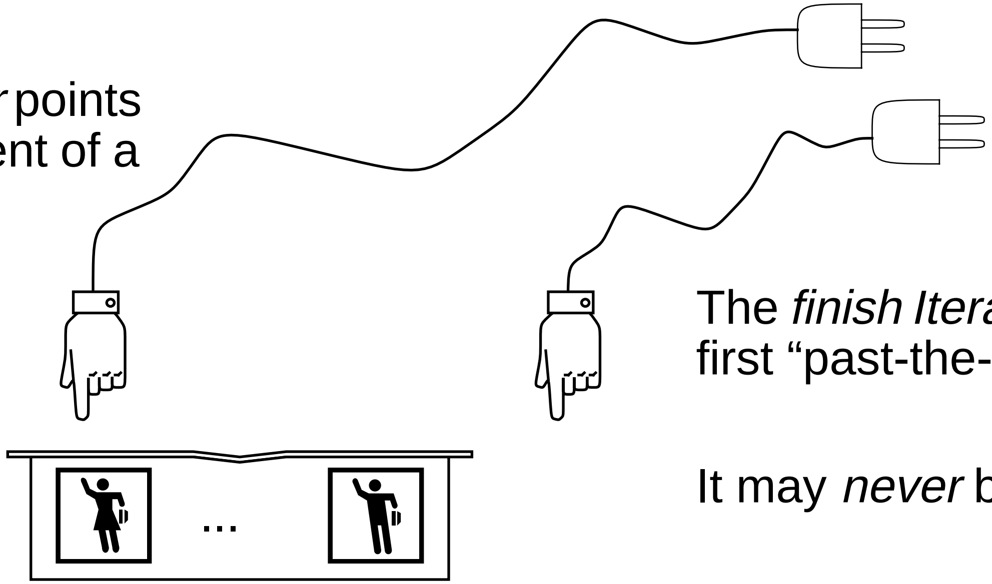
- fast retrieval of multiple values by keys

Containers and Iterators

An Iterator is a "smart pointer" that knows how to traverse a given Container.

Containers have special iterators associated with the start and finish of allocated space:

The *start Iterator* points to the first element of a Container



The *finish Iterator* points to the first "past-the-end" element.

It may *never* be dereferenced!

By convention, many algorithms return a "past-the-end" iterator under special circumstances (e.g., a value is not found in the container) — the client must check before dereferencing the iterator!

Note that a container is empty if start == finish!

Iterators

An Iterator is a “smart pointer”: Iterators are objects that support some or all of the operations defined for pointers, but operate on containers, not arrays.

Input iterators:

- Support copy constructor, ==, !=, and ++
- Dereferencing as rvalue only (*a is constant)

Output iterators:

- Support copy constructor and ++
- Dereferencing as lvalue only (*a = t)

Forward iterators:

- Support default and copy constructors, ==, !=, =, * and ++

Bidirectional iterators:

- Additionally support --

Random access iterators:

- Additionally support +, +=, -, -=, [ ], <, <=, >, >=

The Vector Template

The interface to Vector resembles that of the other containers:

```
template <class T>
class vector {
public:
    typedef Allocator<T> vector_allocator ;
    typedef T value_type ;
    typedef vector_allocator::pointer pointer ;
    typedef vector_allocator::pointer iterator ;
    typedef vector_allocator::const_pointer const_iterator ;
    // other locally defined vector types ...
protected:
    static Allocator<T> static_allocator;                      // handles memory allocation
    iterator start;                                            // start of vector
    iterator finish;                                           // "past the end of" vector
    // other protected variables ...
public:
    vector() : start(0), finish(0), end_of_storage(0) {}  // default constructor
    vector(size_type n, const T& value = T()) { ... }
    vector(const vector<T>& x) ;                           // copy constructor
    vector(const_iterator first, const_iterator last);    // copy from container range
    ~vector();                                             // destructor
    vector<T>& operator=(const vector<T>& x);             // assignment
```

The Vector Template ...

```
    iterator begin() { return start; }                     // provide a start iterator
    iterator end() { return finish; }                      // provide a "past the end" iterator

    // reverse iterators go backwards and forwards instead of forwards and backwards ...

    reverse_iterator rbegin() { return reverse_iterator(end()); }
    reverse_iterator rend() { return reverse_iterator(begin()); }

    size_type size() const { return size_type(end() - begin()); }
    size_type max_size() const { return static_allocator.max_size(); }

    size_type capacity() const { return size_type(end_of_storage - begin()); }
    bool empty() const { return begin() == end(); }

    reference operator[](size_type n) { return *(begin() + n); }

    reference front() { return *begin(); }                 // reference to first element
    reference back() { return *(end() - 1); }

    void push_back(const T& x);                            // insert at end

    iterator insert(iterator position, const T& x);        // insert at position; return iterator
                                                           // pointing to inserted element

    // other insert functions ...
    // other member functions ...
};
```

Generic Algorithms

Generic Algorithms work uniformly on all containers and iterators that satisfy their assumptions:

```
// copy elements from first to last into result

template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result) {
   while (first != last) *result++ = *first++;
   return result;
}
```

Sample Usage:

```
vector<int> a(10), b(5);                  // define integer vectors of length 10 and 5

for (int i=0; i<10; i++) {
   a[i] = i*i;                            // NB: vectors support array operations
}

// Copy a[5] through a[9] to b[0] through b[4]:

copy(a.begin()+5, a.end(), b.begin());    // NB: iterators support pointer operations

for (int j=0; j<5; j++) {
   cout << "b[" << j << "] = " << b[j] << endl;
}
```

Kinds of Generic Algorithms

STL provides a large number of generic algorithms for most common operations on containers:

Non-mutating sequence operations:

- for_each, find, adjacent find, count, mismatch, equal

Mutating sequence operations:

- copy, swap, transform, replace, fill, generate, remove, unique, reverse, rotate, random_shuffle, partition

Sorting:

- sort, nth_element, binary_search, merge
- Set operations: includes, set_union, set_intersection, set_difference
- Heap operations: make_heap, push_heap, pop_heap, sort_heap
- minimum, maximum, permutation

Generalized numeric operations:

- accumulate, inner_product, partial_sum, adjacent_difference

Checking Past-the-End

Many algorithms that return iterators expect the client to check that the result is valid before dereferencing it:

```
// return an iterator pointing to an element equal to value in range [first, last)
template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value) {
   while (first != last && *first != value) ++first;
   return first;                              // NB: result == last if element not found
}
```

A failed search yields a “past-the-end” iterator:

```
void findValue(vector<int> vec, int val)
{
   vector<int>::iterator it;                  // NB: vector<int>::iterator is a typedef
   it = find(vec.begin(), vec.end(), val);
   if (it != vec.end()) {                     // check iterator validity
      cout << "found value " << val << endl;
   } else {
      cout << "value " << val << " not found" << endl;
   }
}
findValue(a, 25);                             // “found value 25”
findValue(b, 99);                             // “value 99 not found”
```

Function Objects

Function Objects are just objects with `operator()` defined.

Many STL algorithms that take functions as arguments will work either with function objects or function pointers:

```
// Apply f() to each element in range [first, last):
template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f) {
    while (first != last) f(*first++);
    return f;
}
```

Function objects differ from function pointers in important ways:

- function objects can be dynamically instantiated with different states
- `operator()` can be inlined
- function objects can be bound to algorithms at compile time

Adaptors

Adaptors modify interfaces so that components can be used in new contexts

Container adaptors:

- `stack`, `queue` and `priority_queue`

Iterator adaptors:

- reverse iterators — reverse direction
- insert iterators — to insert/append vs. overwrite
- raw storage iterator — to write to uninitialized memory

Function adaptors:

- negators — negate unary/binary predicates
- binders — convert binary to unary functions
- pointers to functions — convert functions to function objects

Programmers may also define their own adaptors to integrate existing components ...

The Stack Template

A stack is just an interface to a container:

```
/* Copyright (c) 1994 Hewlett-Packard Company */

template <class Container>
class stack {
   friend bool operator==(const stack<Container>& x, const stack<Container>& y);
   friend bool operator<(const stack<Container>& x, const stack<Container>& y);
public:
    typedef Container::value_type value_type ;
    typedef Container::size_type size_type ;
protected:
    Container c;
public:
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    value_type& top() { return c.back(); }
    const value_type& top() const { return c.back(); }
    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_back(); }
};

template <class Container>
bool operator==(const stack<Container>& x, const stack<Container>& y) { return x.c == y.c; }

template <class Container>
bool operator<(const stack<Container>& x, const stack<Container>& y) { return x.c < y.c; }
```

Using Iterator Adaptors

We can implement the line reverser by copying the input stream to a vector, and then copying our vector back to the output stream in reverse order:

- use input/output stream adapters to make cin and cout look like containers
- use an insert iterator to append rather than overwrite
- use reverse iterators to copy back the vector in reverse order

```
#include <vector.h>
#include <algo.h>                                      // use copy function template

void rev(void)
{
   typedef vector<String> IOVector ;
   typedef istream_iterator<String, ptrdiff_t> String_in ;

   IOVector ioVec;

   copy(
      String_in(cin),                                  // input stream iterator
      String_in(),                                     // dummy "past the end" iterator for input
      back_inserter(ioVec));                           // append instead of overwrite

   copy(
      ioVec.rbegin(),                                  // reverse direction: begin at end
      ioVec.rend(),                                    // end at beginning
      ostream_iterator<String>(cout, "\n"));           // output stream iterator with separator
}
```

Sorting

Here is a complete sort program using STL:

```
#include <iostream.h>
#include <vector.h>
#include <algo.h>
#include "String.h"

int main(void)
{   typedef vector<String> IOVector ;
    typedef istream_iterator<String, ptrdiff_t> String_in ;

    IOVector ioVec;

    copy(String_in(cin), String_in(), back_inserter(ioVec));

    sort(ioVec.begin(), ioVec.end());                  // NB: assumes < defined for String

    copy(ioVec.begin(), ioVec.end(), ostream_iterator<String>(cout, "\n"));
}
```

Using Function Objects

A variant of the sort algorithm allows the user to supply the comparison operator as a Function object:

```
// Function object to compare Strings in reverse direction
class revStrCmp
{
public:
   bool operator() (const String& s1, const String& s2) { return s2 < s1; }
};

void revSort(void)
{
   typedef vector<String> IOVector ;
   typedef istream_iterator<String, ptrdiff_t> String_in ;

   IOVector ioVec;

   copy(String_in(cin), String_in(), back_inserter(ioVec));

   sort(ioVec.begin(), ioVec.end(),
      revStrCmp());                          // NB: pass an instance of revStrCmp

   copy(ioVec.begin(), ioVec.end(), ostream_iterator<String>(cout, "\n"));
}
```

Why Use STL?

Good points:

- covers most of the basic data structures and algorithms
- it’s “standard” (part of ANSI draft C++ standard library)
- fast and efficient
- component-oriented and genuinely reusable
- extensible

Bad points:

- steep learning curve
- no exceptions; no error checking
- not (yet) supported by all compilers
- based on templates (cf. “code bloat” and strange compile-time errors)
- String class not included

Summary

You should know the answers to these questions:

- ❑ What are the five kinds of components in STL?
- ❑ What kinds of components “plug into” generic algorithms?
- ❑ What kinds of iterators are there?
- ❑ What is a “past-the-end” iterator?
- ❑ What kind of adaptors are there?
- ❑ Why is stack an adaptor instead of a container?
- ❑ What is a function object, and why would you use one?

Can you answer the following questions?

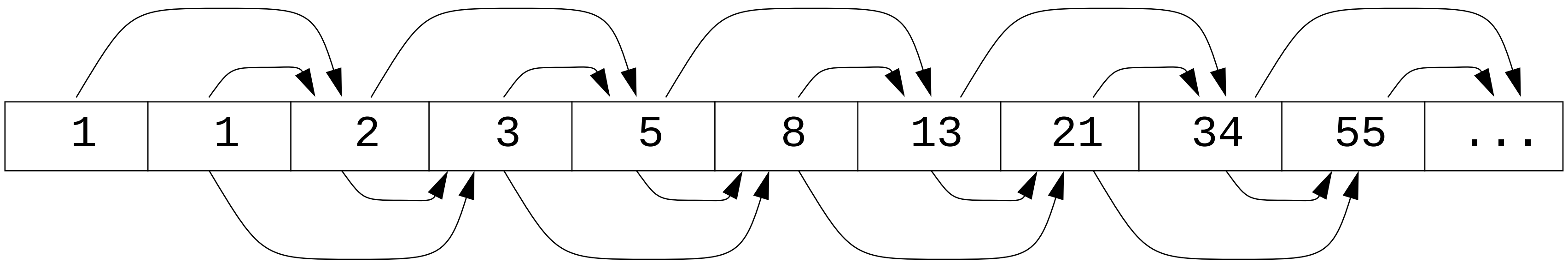
- ✎ *Why isn’t there just one kind of container?*
- ✎ *Why does STL catch attempts to dereference “past-the-end” iterators?*
- ✎ *Why isn’t there just one kind of iterator?*
- ✎ *Why is operator overloading indispensable for STL?*
- ✎ *How can templates cause “code bloat”?*

13. Two STL Examples

- Fibonacci Numbers:
  - Problems with naive recursion
  - Lazy Lists as caches for computed values
  - A `LazyList` function object class
- The Jumble Puzzle
  - A naive solution — looking up permutations
  - An efficient solution — looking up keys
  - Implementing an Unjumble program with STL components

Fibonacci Numbers

The Fibonacci sequence is a classic example of a recursively defined series of numbers:



Naive Recursion

A straightforward implementation of a function to compute the Fibonacci numbers can be defined recursively:

```
int
rfib (int n)                                // Compute the Fibonacci function recursively
{
    switch (n) {
    case 0 :                                // Our sequence starts with rfib(0), not rfib(1)
    case 1 :
        return 1;
        break;
    default :
        return rfib(n-1) + rfib(n-2);       // Naive recursion!
        break;
    }
}
```

Problems with Pure Functions

Pure functions are good for reasoning about programs because they have no side effects: one can always replace a function call by its value within an expression, or vice versa, so evaluation order does not matter [cf. the *Church-Rosser* property].

- ❑ But pure functions can be inefficient because every time they are called with the same value, the *same* computation will be performed.
- ❑ Worse, multiply recursive functions must be re-written to avoid exponential complexity:
  - ☞ To compute rfib(9), rfib(8) will be called twice, rfib(7) three times, rfib(6) five times ...

✎ *How could you re-write rfib( ) to work in linear time?*

✎ *How many times will rfib(0) be called in the computation of rfib(n)?*

Benevolent Side Effects

We can make pure functions more efficient by *caching* values when they are computed.

The next time the function is called with the same argument, the cached value can be returned instead of recomputing the function body. Because of the Church-Rosser property, it is safe to return the value instead of computing it!

Note that storing computing values *is* a side effect, but because this side effect is guaranteed to be invisible to clients (except for execution speed), it is considered “benevolent.”

Lazy Lists

A "Lazy List" represents an *infinite* list of values that are computed by some function.

The nth element of the Lazy List is only computed when it is needed, however, so its representation is always finite:

```
Fibonacci numbers =
```

1	1	2	3	5	8	Use rfib() to compute missing values ...

If the function used to compute the values is defined recursively, it can use the values already computed in the lazy list as a cache.

Lazy Lists:

- ❑ behave like caches, so values are only computed once
- ❑ reduce complexity of multiply recursive functions by caching computed values

Example: Lazy Lists as Function Objects

In C++, a Lazy List can be implemented as a function object:

```
class LazyList {
public:
    // constructor requires pointer
    // to function that computes f():
    LazyList(int (*f) (int, LazyList&));

    // operator() returns nth value:
    int operator() (int);

private:
    vector<int> _cache; // caches f(0) to f(n)
    // function that computes f(n)
    int (*_f) (int, LazyList&);

private:
    // hide default constructor etc.
    LazyList(void);
    LazyList(LazyList&);
    LazyList& operator=(LazyList&);
};

// constructor just remembers pointer to f():

LazyList::LazyList(int (*f) (int, LazyList&)) :
    _f(f)
{ }
```

```
// Return f(n) if already cached, else generate
// all missing values up to and including f(n)

int
LazyList::operator() (int n)
{
    // NB: if _cache.size() > n
    // then _cache[n] is already defined
    int i;
    for (i=_cache.size(); i<=n; i++) {
        _cache.push_back(_f(i,*this));
    }
    return _cache[n];
}

// Note that f() is able to use all the values
// currently stored in this LazyList!
```

Example: A Fibonacci Function Object

Now we can reimplement our Fibonacci function as a Lazy List with minimal changes:

```
int computeFib(int n, LazyList& fibList);           // declaration for function pointer
LazyList lfib(computeFib);                          // construct our function object
int
computeFib(int n, LazyList& fibList)                // NB: need extra argument
{
    switch (n) {
    case 0 :
    case 1 :
        return 1;
        break;
    default :
        return fibList(n-1) + fibList(n-2);         // call lazy list instead of self
        break;
    }
}
```

Note that the `lfib` function object now computes Fibonacci numbers in linear time (and returns pre-computed values in constant time!).

Function Objects as Functions

In (rare) situations where you really need a function instead of a function object, you can wrap the function object as a static variable:

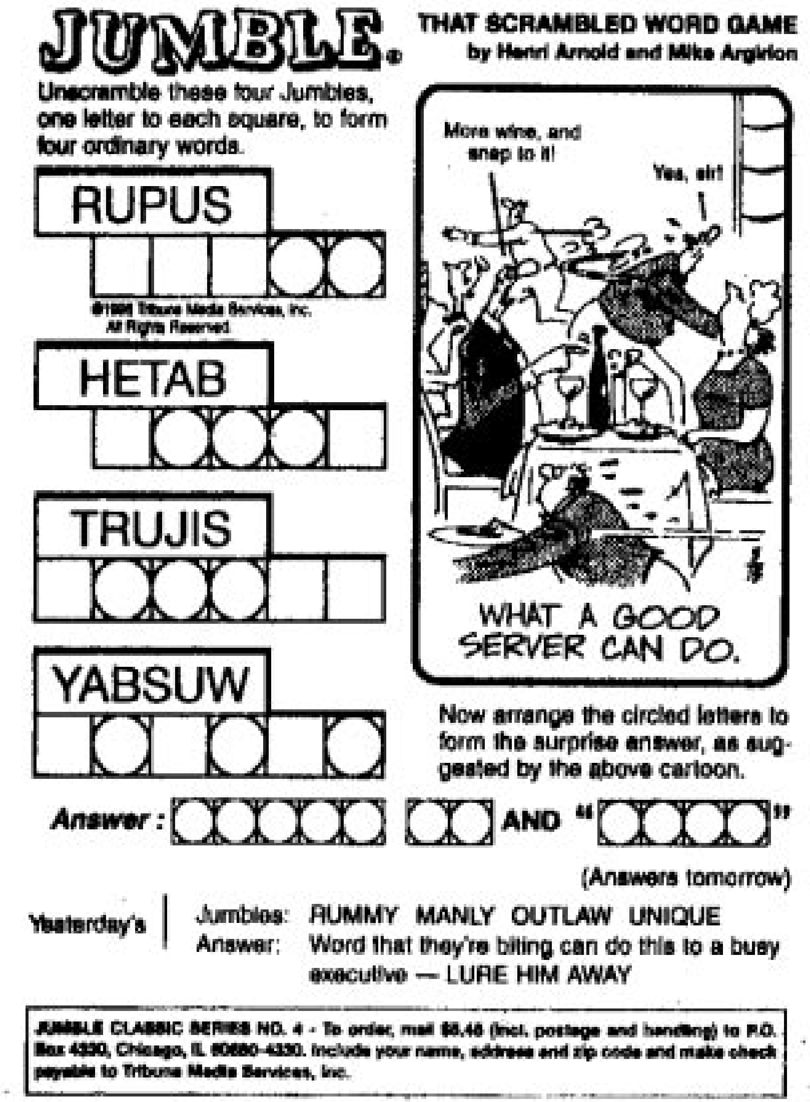
```
int
lfib (int n)                                  // want a real function, not a function object
{
   static LazyList fibList(computeFib);       // this LazyList will persist between calls!
   return fibList(n);
}
```

The Jumble Puzzle

The Jumble Puzzle tests your English vocabulary by presenting four jumbled, ordinary words.

The circled letters of the unjumbled words represent the jumbled answer to a cartoon puzzle.

Since the jumbled words can be found in an electronic dictionary, it should be possible to write a program to automatically solve the first part of the puzzle (unjumbling the four words).



Naive Solution

Generate all permutations of the jumbled words:	rupus urpus uprus purus pruus rpuus ruups urups ...	For each permutation, check if it exists in the word list:	abacus abalone abase abash ... zounds zucchini Zurich zygote

The obvious, naive solution is extremely inefficient: a word with *n* characters may have up to n! permutations. A five-letter word may have 120 permutations and a six-letter word may have 720 permutations. “rupus” has 60 permutations.

✎ *Exactly how many permutations will a word have in general?*

Rethinking the Jumble Problem

Observation: if a jumbled word (e.g. “rupus”) can be unjumbled to a real word in the list, then these two words are *jumbles of each other* (i.e. they are anagrams).

☞ Is there a fast way to tell if two words are anagrams?

Two words are anagrams if they have the same numbers of the same letters.

☞ Each word has a unique “key” consisting of its letters in sorted order

The key for “rupus” is “prsuu”.

☞ Two words are anagrams if they have the same key

We can unjumble “rupus” by looking for a word with the same key.

An Efficient Solution

1. Build an associative array of keys and words for every word in the dictionary:
2. Generate the key of a jumbled word:
   key("rupus") = "prsuu"
3. Look up and return the words with the same key.

Key	*Word*
aabcsu	abacus
aabelno	abalone
aabes	abase
aabhs	abash
...	...
dnosuz	zounds
cchiinuz	zucchini
chiruz	zurich
egotyz	zygote

We already have STL implementations of associative arrays (`multimap`) and a generic `sort` algorithm, so we should be able to use these to implement our unjumbler.

Generating Keys

Two words are anagrams if they have the same key:

```
String makeKey(const String& word)
{
   String key = word;
   sort(key.begin(), key.end());
   return(key);                    // NB: returns a String instance, not a reference
}
```

We must adapt our String class so we can generate keys:

```
class String
{
   // rest as before ...
public:
   // These return iterators (pointers) for use with STL algorithms:
   char * begin(void) { return _s; }
   char * end(void) { return _s+strlen(); }
private:
   char *_s;                       // String representation, as before ...
   // no changes ...
};
```

Initializing the Dictionary

The dictionary will be hold pairs of Strings (keys and words:

```
typedef multimap<String, String, less<String> > Dict ;
typedef Dict::value_type StringPair ;      // What you can store in a Dict
```

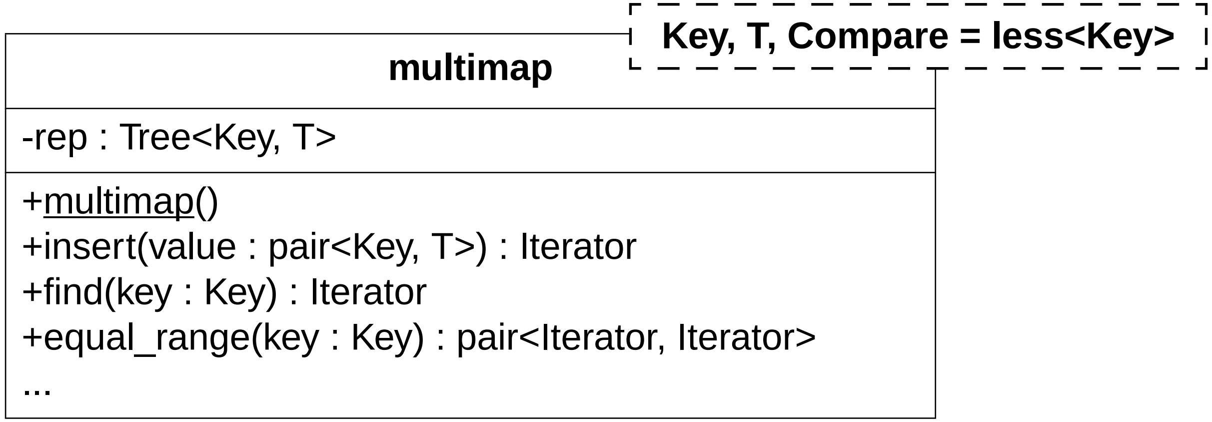
The dictionary is initialized from a predefined word list:

```
void loadDict(Dict& dict)
{
   String word;
   ifstream wordStream;                      // Input File Stream

   wordStream.open(wordFile);                // I.e., /usr/dict/words
   if (!wordStream) {                        // Can we open the file of words?
      cerr << "Invalid wordStream" << endl;
      return;
   }

   while (word.getline(wordStream)) {        // Make an entry for each word
      dict.insert(StringPair(makeKey(word), word));
   }
}
```

NB: We will also need to define `String::operator<( )` *so that our instance of multimap can compare words with* `less<String>`*.*

The STL Multimap Container

The multimap container implements an associative array over arbitrary types using a Tree of Keys and Values. Keys must support a comparison operator, which is also supplied as a parameter.



A multimap stores `pairs` of keys and values. `pair` is a basic STL component for representing arbitrary pairs of values as a single entity.

Since multiple words may have the same key, we need a `multimap` rather than a simple `map`, and we should use `equal_range` to return the set of matching words, rather than the simpler `find` member function.

The Multimap Class Template

```
#include <tree.h>

template <class Key, class T, class Compare>
class multimap {
public:
    typedef Key key_type ;
    typedef pair<const Key, T> value_type ;

private:
    typedef rb_tree<key_type, value_type, select1st<value_type, key_type>, key_compare> rep_type;
    rep_type t;             // red-black tree representing multimap

public:
    multimap(const Compare& comp = Compare()) : t(comp, true) { }
    // ...

    iterator begin() { return t.begin(); }
    iterator end() { return t.end(); }
    bool empty() const { return t.empty(); }
    // ...

    iterator insert(const value_type& x) { return t.insert(x).first; }
    // ...

    iterator find(const key_type& x) { return t.find(x); }
    typedef pair<iterator, iterator> pair_iterator_iterator;
    pair_iterator_iterator equal_range(const key_type& x) { return t.equal_range(x); }
    // ...
};
```

The Pair Template

Pairs of arbitrary values are constructed with the pair class template:

```
/* Copyright (c) 1994 Hewlett-Packard Company */

#ifndef PAIR_H
#define PAIR_H

#include <bool.h>

template <class T1, class T2>
struct pair {                    // NB: a struct is just a class whose members are all public
    T1 first;
    T2 second;
    pair() : first(), second() {}
    pair(const T1& a, const T2& b) : first(a), second(b) {}
};

template <class T1, class T2>
inline bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y) {
    return x.first == y.first && x.second == y.second;
}

template <class T1, class T2>
inline bool operator<(const pair<T1, T2>& x, const pair<T1, T2>& y) {
    return x.first < y.first || (!(y.first < x.first) && x.second < y.second);
}

template <class T1, class T2>
inline pair<T1, T2> make_pair(const T1& x, const T2& y) {
    return pair<T1, T2>(x, y);
}

#endif
```

Using the Pair Template

```
#include <pair.h>                        // use STL pair template
#include <assert.h>                      // use assert() -- a (poor) alternative to exceptions

template <class Pair>
void printPair(char* name, Pair p)       // function template
{
   cout << name << " = pair(" << p.first << "," << p.second << ")" << endl;
}

void pairTest(void)                      // example function to test the pair template
{
   pair<int,int> rect(3,5), rect2(2,6);
   printPair("rect", rect);              // instantiate printPair<pair<int,int> >()
                                         // prints: “rect = pair(3,5)”
   assert(rect2 < rect);                 // OK, < is defined for ints

   typedef pair<int, String> intStrPair ;
   intStrPair red(1, "red"), blue(2, "blue"), colour;

   printPair("red", red);                // instantiate printpair<intStrPair>()
                                         // prints: “red = pair(1,red)”
   colour = blue;                        // OK, = defined for int and for String
   printPair("colour", colour);          // prints: “colour = pair(2,blue)”

   assert(colour == blue);               // OOPS -- operator== not defined for our String!
}


class String {                           // Need to add this to our String class ...
public:
   int operator== (const String& s1) const { return ::strcmp(_s, s1._s) == 0; };
};
```

Jumble Declarations

```
#include <iostream.h>        // cout
#include <fstream.h>         // ifstream

#include <algo.h>            // sort, for_each
#include <multimap.h>        // multimap

#include "String.h"

const char* const wordFile = "words";      // File containing word list

// The dictionary will hold pairs of Strings:
typedef multimap<String, String, less<String> > Dict ;
typedef Dict::value_type StringPair ;

void unJumble(void);
void loadDict(Dict& dict);
String makeKey(const String& word);

int main(void)
{
   unJumble();
   return 0;
}

class printWord {   // Function object to print selected words in dictionary
   public:
      void operator() (StringPair& item) { cout << item.second << endl; }
};
```

UnJumble

```
void unJumble(void)                 // Prompt user for words to unjumble ...
{
   String word, key;
   Dict dict;
   pair<Dict::iterator, Dict::iterator> range;
   loadDict(dict);                                // Initialize the dictionary
   while (true) {
      cout << "Unjumble: " << flush;
      cin >> word;
      if (word.strlen() == 0) {                   // Quit on empty input
         cout << "Bye!" << endl;
         return;
      }
      key = makeKey(word);                        // Generate a key
      range = dict.equal_range(key);              // Find matching words
      if (range.first == range.second) {
         cout << "Can't unjumble " << word << endl;
      } else {
         cout << "Found:" << endl;
         for_each(range.first, range.second, printWord());
      }
   }
}
```

Summary

You should know the answers to these questions:

- ❑ Why should you beware of multiply recursive functions?
- ❑ What is a "benevolent side effect"?
- ❑ How can a "lazy list" make functions more efficient?
- ❑ What is an associative array?

Can you answer the following questions?

- ✎ *How can you compute Fibonacci numbers in linear time with pure, recursive functions?*
- ✎ *Exactly how many times does* `rfib()` *call itself to compute* `rfib(`*n*`)` *for some n?*
- ✎ *How would you make the* `LazyList` *class work for other types of values?*
- ✎ *When should (or shouldn't) you use lazy lists?*
- ✎ *How many permutations does an arbitrary word have?*
- ✎ *How would you define* `String::operator<( )`*?*
- ✎ *How would you define the* `less` *class template?*