

# *7033 Programmierung 2*

Prof. O. Nierstrasz

Sommersemester 1998

# Table of Contents

Table of Contents	i	Summary	31	Merge Sort	64
C++ Programming Rules, Hints and Guidelines	iv	<b>3. C++ Basic Language Features</b>	<b>32</b>	Merge	65
<b>1. P2 — Introduction to C++</b>	<b>1</b>	Symbols	33	Refactoring Merge()	66
Essential C++ Texts	2	Keywords	34	Optimizing MergeSort ...	67
Overview	3	Comments	35	MergeSort with a Fixed Buffer	68
What You Will Be Expected To Learn	4	Commenting Conventions	36	A Faster MergeSort	69
History	5	Built-In Data Types	37	Summary	70
C++ Design Goals	6	Expressions	38	<b>5. Specifying Classes</b>	<b>71</b>
C Features	7	Operator Precedence and Associativity	39	Abstract Data Types and Invariants	72
C++ Features	8	C++ Arrays	41	Example: Tic Tac Toe	73
“Hello World”	9	Pointers	42	C++ Classes	74
C++ Storage Classes	10	References	43	Designing a Tic Tac Toe Game	75
Memory Layout	11	Strings	44	Desired Interaction	76
Declarations and Definitions	12	Assignment — lvalues and rvalues	45	The Tic Tac Toe Driver	77
Hello World Project	13	Statements	46	Determining the Interface	78
Compiling C++ Programs	14	Enumeration Types	47	Exceptions	79
Basic Makefile	15	Functions	48	Specifying the Interface	80
Summary	16	Summary	49	Instance Variables	81
<b>2. A Taste of C++ — Comparison with Eiffel</b>	<b>17</b>	<b>4. Decomposition and Recursion</b>	<b>50</b>	Implementing the Constructor	82
Data Abstraction — Line Reverser Example	18	Document Assumptions	51	Implementing the Game	83
Eiffel Line Reverser	19	Comment Selectively	52	Static Declarations	84
An Eiffel Stack Implementation	20	Divide and Conquer	53	Constant Declarations	85
The (Hidden) Eiffel Stack Cells	21	Recursion	54	Playing the Game	86
A C++ Line Reverser	22	Recursion — Pros and Cons	55	Printing the Game	87
A C++ Stack Interface	23	Iteration vs. Recursion	56	The Complete TicTacToe Interface	88
A C++ Stack Implementation	24	Binary Search	57	Summary	89
Differences Between Eiffel and C++	25	Binary Search — Recursive Solution	58	<b>6. Data Abstraction</b>	<b>90</b>
A C++ Template Line Reverser	26	Records as Objects	59	The Run-time Stack	91
A C Line Reverser	27	Tail Recursion	60	Stack as a Data Abstraction	92
A Recursive Line Reverser	28	Binary Search — Iterative Solution	61	Postfix Expressions	93
A Perl Line Reverser	29	Sorting	62	A Postfix Expression Interpreter	94
Some Timing Differences	30	MergeSort Example	63	Stacks as Linked Lists	95

Stacks, Queues and Linked Lists	96	Dynamic Memory Management	131	Debuggers	166
Linked List Operations	97	The Final String.h	132	Using dbx	167
Class Invariants	98	Summary	133	GUI Debuggers — CodeWarrior	168
LList Declaration	99	<b>8. Inheritance</b>	<b>134</b>	Profilers	169
Implementing List Methods	100	The Board Game	135	Using gprof	170
List Constructor and Destructor	101	Interaction	136	SNiFF+	171
Growing the List	102	Class Hierarchy	137	Using SNiFF+	172
Checking Pre-conditions	103	Uses of Inheritance	138	SNiFF+ Source Editor	173
Implementing a Stack with a Linked List	104	Polymorphism	139	SNiFF+ Hierarchy Browser	174
Example: Balancing Parentheses	105	Polymorphic Destruction	140	SNiFF+ Class Browser	175
Parenthesis balancer	106	The BoardGame Interface	141	Purify	176
Implementing a Queue with a Linked List	107	Virtual Members	142	Using Purify	177
The Dangers of Call by Value	108	Default Initializers	143	Other tools	178
Guard Against Shallow Copies	109	Arrays of arrays	144	Summary	179
Summary	110	Non-Virtual Functions	145	<b>10. An Introduction to Java</b>	<b>180</b>
<b>7. Managing Memory</b>	<b>111</b>	Using Virtual Functions	146	Java	181
Orthodox Canonical Form	112	Defining Virtual Functions	147	Java and C++ — Similarities and Extensions	182
Example: A String Class	113	Public Inheritance	148	Java and C++ — Simplifications	183
First version of String.h	114	Base Class Initialization	149	The “Hello World” Program	184
Default Constructors	115	Keeping Score	150	Packages	185
Automatic and Dynamic Objects	116	Using Function Pointers	151	Java Basics	186
Destructors	117	Using Static Functions	152	Classes and Objects	187
Copy Constructors	118	Implementation Inheritance	153	Garbage Collection	188
Other Constructors	119	Summary	154	Inheritance	189
Refactoring Common Code	120	<b>9. Tools</b>	<b>155</b>	Dynamic Binding	190
Assignment Operators	121	Makefiles	156	Downcasting	191
Shallow and Deep Copying	122	Make Options	157	Feature Visibility	192
Inline Functions	123	Description File Lines	158	Modifiers	193
Using the Constructors	124	Macros and Special Targets	159	Exceptions	194
Implicit Conversion	125	Gomoku Makefile	160	Defining Exceptions	195
Conditional Compilation	126	Makefile for g++	161	Multiple Inheritance	196
Operator Overloading	127	Version Control	162	Interfaces	197
Overloadable Operators	128	RCS	163	Overriding and Overloading	198
Friends	129	RCS Usage	164	Arrays	199
IOStream Operators	130	Additional RCS Features	165	Arrays and Generics	200

The Java API	201
Applets	202
The Hello World Applet	203
Frameworks vs. Libraries	204
Standalone Applets	205
Events	206
The Scribble Applet	207
Responding to Events	208
Running the Scribble Applet	209
Summary	210
<b>11. Design Rules</b>	<b>211</b>
Basic Rules	212
Deleting Objects	213
Running out of Memory	214
Constructors	215
Initialization	216
Virtual Destructors	217
Assignment	218
Assignment and Inheritance	219
Classes and Functions	220
Class Interfaces	221
References and Values	222
Data Accessibility	223
Const Member Functions	224
Overloading vs. Default Parameters	225
Ambiguous Overloading	226
Common Errors	227
Summary	228

# C++ Programming Rules, Hints and Guidelines

<b>C++ Basic Language Features</b> - - - - -	<b>32</b>
Use // comments exclusively within functions so that any part can be commented out using comment pairs. . . . .	35
Use meaningful names to make your code as self-documenting as possible. . . . .	36
DON'T use comments to restate what is obvious from the source code. . . . .	36
DO use comments to improve the readability of your programs. . . . .	36
Avoid cryptic expressions! Use comments to explain mysterious code. . . . .	38
References should generally be preferred to pointers except when: . . . . .	43
It is generally better to use a C++ string class instead of built-in char arrays! . . . . .	44
<b>Decomposition and Recursion</b> - - - - -	<b>50</b>
Use descriptive names for variables; use short names only when their purpose is obvious from the context. . . . .	51
Always state explicitly all pre- and post-conditions. . . . .	51
Document all assumptions. . . . .	51
Avoid making assumptions that you can't check! . . . . .	51
Avoid complex or cryptic code; write code that is self-documenting. . . . .	52
Use comments to explain any code that is not self-documenting. . . . .	52
Ensure your programs are correct before you try to optimize them. . . . .	52
Never try to optimize code that is not a proven source of system inefficiency. . . . .	52
If possible, check your assumptions, and raise exceptions when they are violated. . . . .	54
If a problem is inherently recursive, implement a correct recursive solution before deciding whether a non-recursive solution is better. . . . .	55
A function or procedure should always have a clear responsibility; promote readability by decomposing complex algorithms into helper functions. . . . .	64
State loop invariants explicitly, and check that they hold through all execution paths. . . . .	65
Eliminate duplicate code through refactoring or reorganizing. . . . .	66
<b>Specifying Classes</b> - - - - -	<b>71</b>
Prototyping strategy: always work with a running, if incomplete program, and incrementally "grow" the full version. . . . .	77
Describe services at highest level of abstraction possible. Determine who is responsible for what! . . . . .	78
Exceptions should only be used to signal abnormal situations, not normal flow of control. . . . .	79
Use symbolic names and enumerated types to make your code as self-documenting as possible. . . . .	81
<b>Data Abstraction</b> - - - - -	<b>90</b>
Always encapsulate data structures as data abstractions. . . . .	92
A method should always do one thing well; don't mix up responsibilities. . . . .	100
Methods should be short and easy to read. . . . .	100
Declare a <code>private</code> copy constructor, if your objects should not be passed by value. . . . .	109
<b>Managing Memory</b> - - - - -	<b>111</b>
Use the orthodox canonical form for any non-trivial class whose objects will be copied or assigned to. . . . .	112
Decide what your class invariant is and make sure that each constructor correctly establishes the invariant. . . . .	115
If you use <code>new</code> , make sure that there will be exactly one matching <code>delete</code> ! . . . . .	117
Destructors should deallocate all memory belonging to an object's private state. . . . .	117
Clearly document whether helper functions assume or ensure class invariants! . . . . .	120

An assignment operator should always test for copying of self . . . . .	121
Don't bother declaring inline functions unless (or until) you can be sure you will get a real improvement in performance. . . . .	123
Short, frequently called functions may be good candidates for inlining. . . . .	123
Don't worry too much about unnecessary copying, but be aware of its overhead in computationally intensive code! . . . . .	125
<b>Inheritance</b> . . . . .	<b>-134</b>
A subclass should only redefine a member function if it has been declared virtual! . . . . .	142
Be sure that the implicit signatures of functions with default initializers do not overlap with those of other declared functions! . . . . .	143
<b>Tools</b> . . . . .	<b>-155</b>
Always define makefiles, even for your most trivial projects. . . . .	157
You should use a version control system for any project that is non-trivial, developed by a team, or delivered to multiple clients. . . . .	162
Use a debugger whenever you are unsure why your program is not working. . . . .	166
Use a profiler to gain insight into where your program is spending most of its time. . . . .	169
Always use an integrated programming environment if one is available! . . . . .	171
Use purify (or an equivalent utility) while developing C++ programs to catch errors in managing memory. . . . .	176
<b>An Introduction to Java</b> . . . . .	<b>-180</b>
<b>Design Rules</b> . . . . .	<b>-211</b>
Use <code>const</code> and <code>inline</code> instead of <code>#define</code> . . . . .	212
Prefer <code>iostream.h</code> to <code>stdio.h</code> . . . . .	212
Use the same form in corresponding calls to <code>new</code> and <code>delete</code> . . . . .	213
Call <code>delete</code> on pointer members in destructors. . . . .	213
Check the return value of <code>new</code> . . . . .	214
Define a copy constructor and an assignment operator for classes with dynamically allocated memory . . . . .	215
Prefer initialization to assignment in constructors . . . . .	215
List members in an initialization list in the order in which they are declared . . . . .	216
Make destructors virtual in base classes . . . . .	217
Have <code>operator=</code> return a reference to <code>*this</code> . . . . .	218
Check for assignment to self in <code>operator=</code> . . . . .	218
Assign to all data members in <code>operator=</code> . . . . .	219
Differentiate among member functions, global functions and friend functions. . . . .	220
Avoid data members in the public interface. . . . .	221
Use <code>const</code> wherever possible. . . . .	221
Pass and return objects by reference instead of by value . . . . .	222
Don't try to return a reference when you must return an object. . . . .	222
Never return a reference to a local object or a dereferenced pointer initialized by <code>new</code> within the function. . . . .	223
Avoid member functions that return pointers or references to members less accessible than themselves. . . . .	223
Avoid returning "handles" to internal data from <code>const</code> member functions . . . . .	224
Choose carefully between function overloading and parameter defaulting . . . . .	225
Avoid overloading on a pointer and a numerical type . . . . .	226

# 1. P2 — Introduction to C++

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

**Secretary:** Frau I. Huber, Tel. 631.4692

**Assistants:** Franz Achermann, Stefan Kneubuehl, Matthias Scheidegger

**WWW:** <http://www.iam.unibe.ch/~scg>

**Principle Text:**

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

## *Essential C++ Texts*

- ❑ Margaret 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++*, 2d ed., Addison-Wesley, 1997.
- ❑ 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.



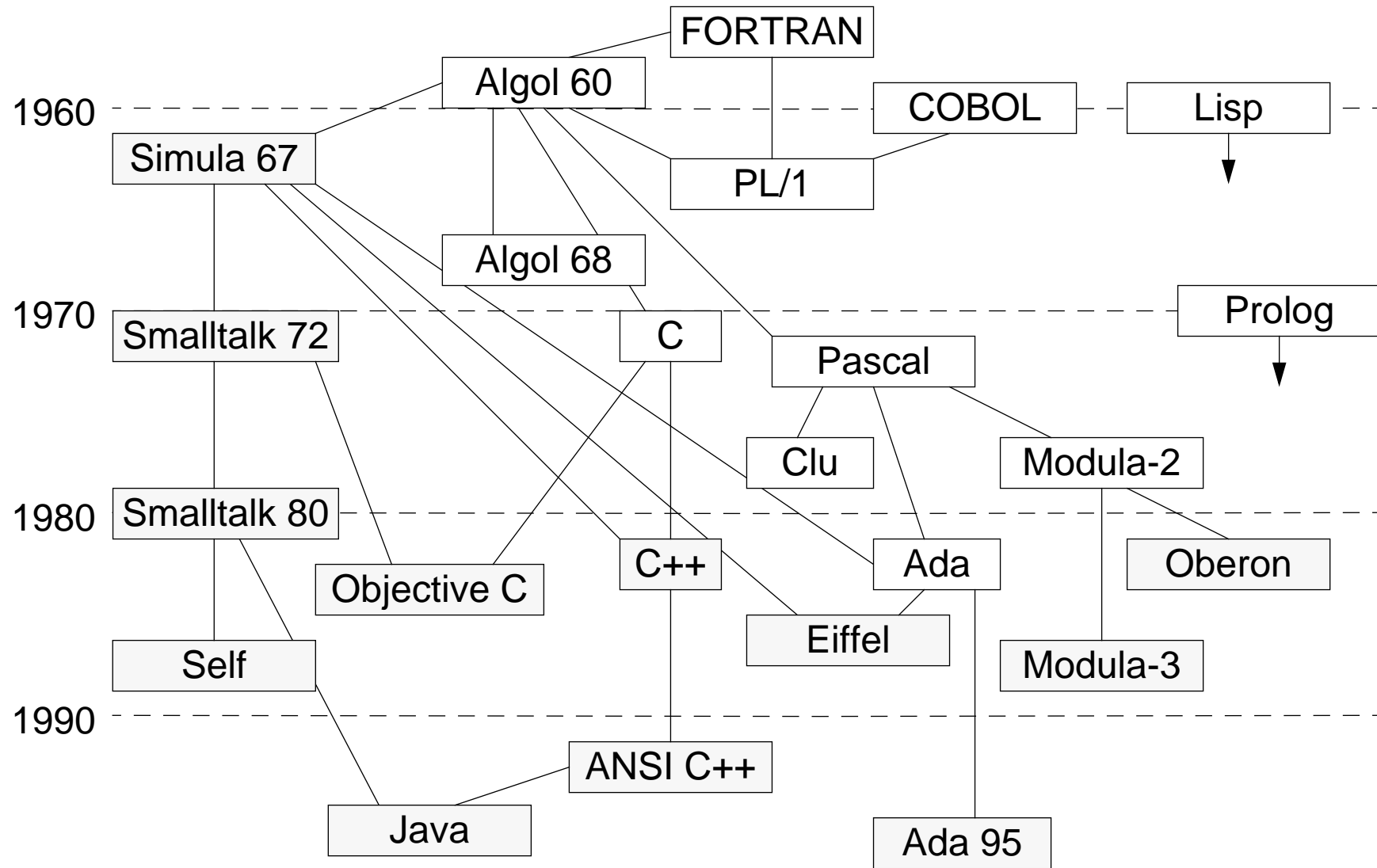
# Overview

1. 30.03 Introduction
2. 06.04 A Taste of C++ — Comparison with Eiffel  
13.04 *Easter Monday — no lecture*
3. 20.04 C++ Basic Language Features
4. 27.04 Decomposition and Recursion
5. 04.05 Specifying Classes
6. 11.05 Data Abstraction
7. 18.05 Managing Memory
8. 25.05 Inheritance  
01.06 *Whit Monday — no lecture*
9. 08.06 Tools
10. 15.06 An Introduction to Java  
22.06 *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



## 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!
*/
```

*Function definition:* there is always a "main" function

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

*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

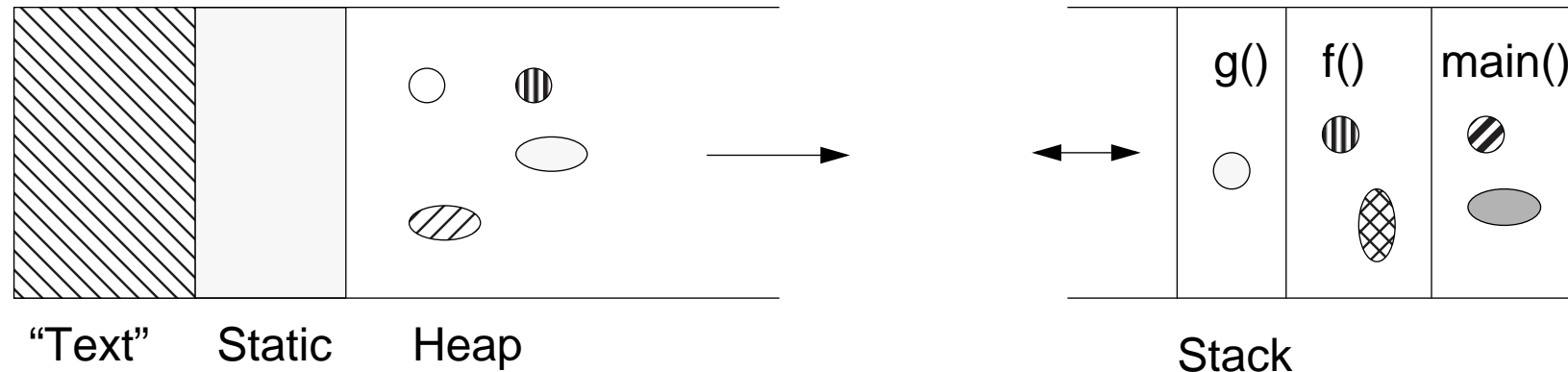
- ☞ only live during function invocation on the “run-time stack”

## 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:

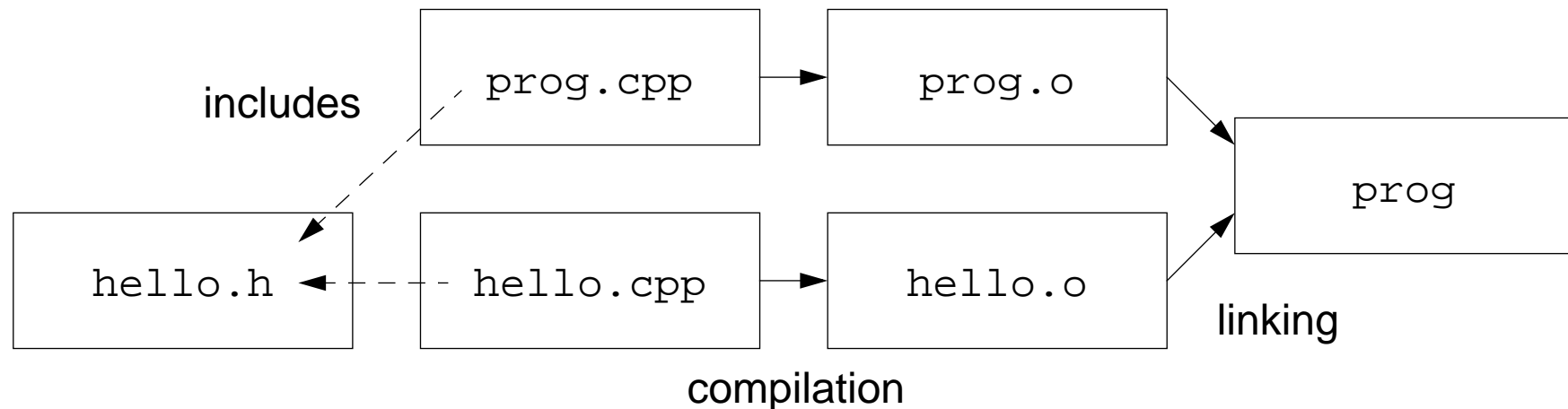
- ☞ CC prog.cpp — generates executable *a.out*
- ☞ CC -o prog prog.cpp — generates executable *prog*

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”?

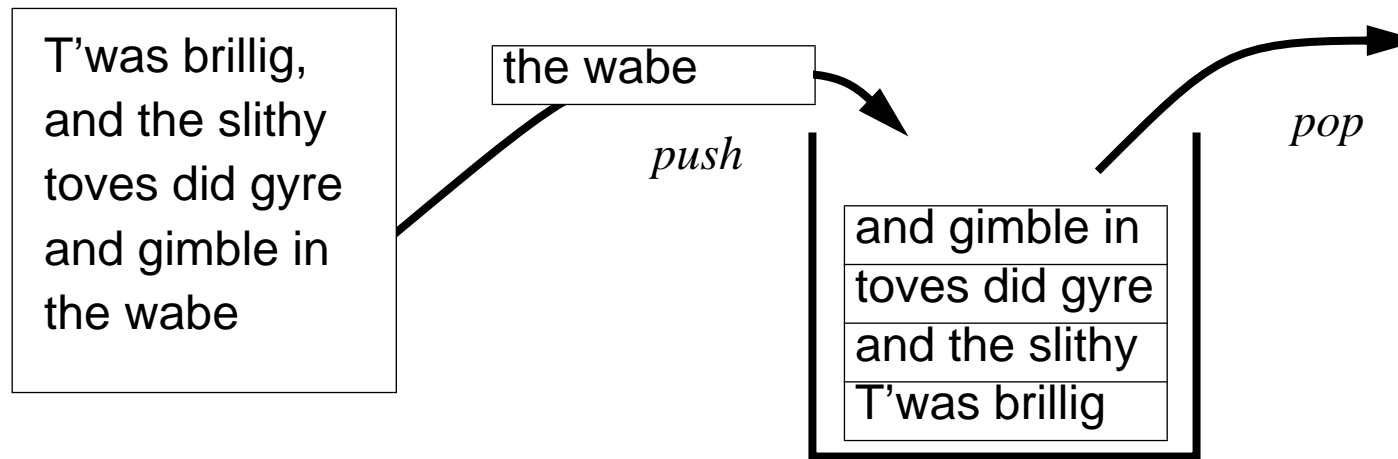
### Can you answer the following questions?

- ✎ *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?*

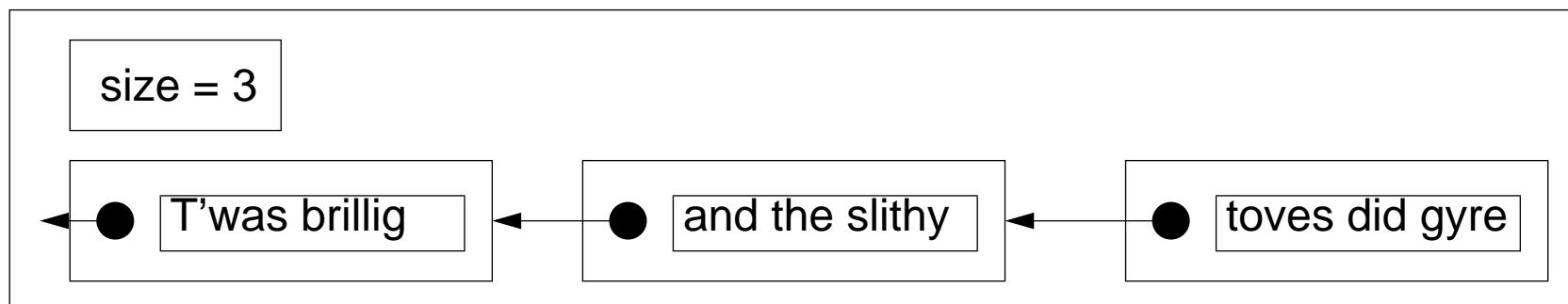
## 2. A Taste of C++ — Comparison with Eiffel

- ❑ 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
    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++*

<i>Eiffel</i>	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 (<>) {
    push(@file, $_);
}
while ($#file>=0) {
    print pop(@file);
}
__END__
```

## *Some Timing Differences*

Input file: 20960 lines, 366167 characters

	<i>Real Time</i>	<i>User Time</i>	<i>System Time</i>
<i>rwrev (RW Templates)</i>	4.6	3.3	1.1
<i>erev (Eiffel)</i>	4.6	3.2	1.1
<i>hrev (C++)</i>	4.3	2.9	1.2
<i>cpprev (C++)</i>	3.2	1.6	1.4
<i>prev (Perl)</i>	2.0	1.2	0.5
<i>crev (C)</i>	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>	<b>continue</b>	<i>float</i>	<u>new</u>	<i>signed</i>	<b>try</b>
<i>auto</i>	<b>default</b>	<b>for</b>	<i>operator</i>	<u>sizeof</u>	<i>typedef</i>
<b>break</b>	<u>delete</u>	<i>friend</i>	<i>private</i>	<i>static</i>	<i>union</i>
<b>case</b>	<b>do</b>	<b>goto</b>	<i>protected</i>	<i>struct</i>	<i>unsigned</i>
<b>catch</b>	<i>double</i>	<b>if</b>	<i>public</i>	<b>switch</b>	<i>virtual</i>
<i>char</i>	<b>else</b>	<i>inline</i>	<i>register</i>	<i>template</i>	<i>void</i>
<i>class</i>	<i>enum</i>	<i>int</i>	<b>return</b>	<u>this</u>	<i>volatile</i>
<i>const</i>	<i>extern</i>	<i>long</i>	<i>short</i>	<b>throw</b>	<b>while</b>

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.

Underlined 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

<i>Data type</i>	<i>No. of bits</i>	<i>Minimal value</i>	<i>Maximal value</i>
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

<i>Data type</i>	<i>No. of bytes</i>	<i>Min. exponent</i>	<i>Max. exponent</i>	<i>Decimal accuracy</i>
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

<i>Level</i>	<i>Operator</i>	<i>Function</i>
17R	::	global scope (unary)
17L	::	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

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

# C++ Arrays

## Arrays are fixed sequences of homogeneous elements

- ❑ `Type a[n];` defines a one-dimensional array `a` in a contiguous block of  $(n * \text{sizeof}(\text{Type}))$  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 different 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.

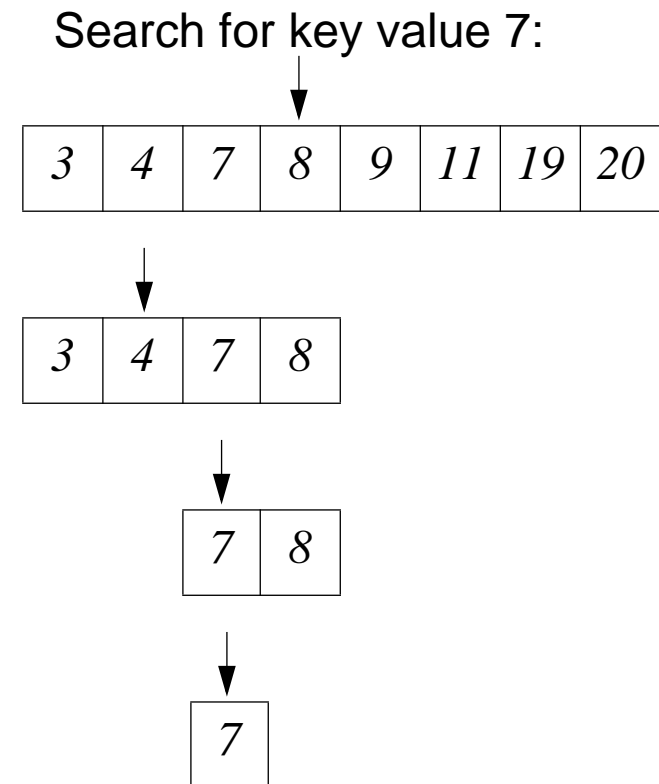
*Binary search is naturally expressed as a recursive algorithm:*

*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



## 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);
    }
} // recursive, but not tail-recursive
```

```
int trfactorial(unsigned int n,
               unsigned int result=1)
{
    if (n==0) {
        return result;
    } else {
        return trfactorial(n-1, n*result);
    }
} // tail recursive!
```

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)
{
    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).

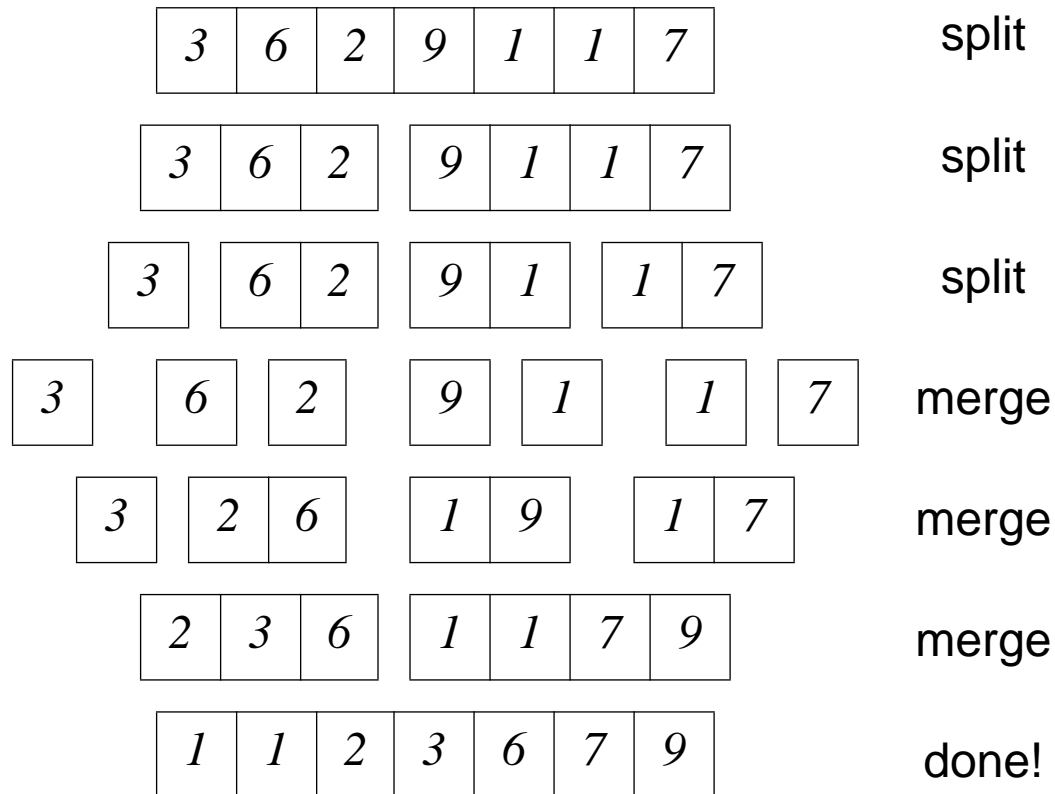
The principle of divide and conquer leads to an efficient, recursive solution:

- ❑ 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 {
                // a2[i2] smaller
                b[i] = a2[i2++];
            }
        } 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 smaller
        }
    }
}
```



## Optimizing MergeSort ...

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

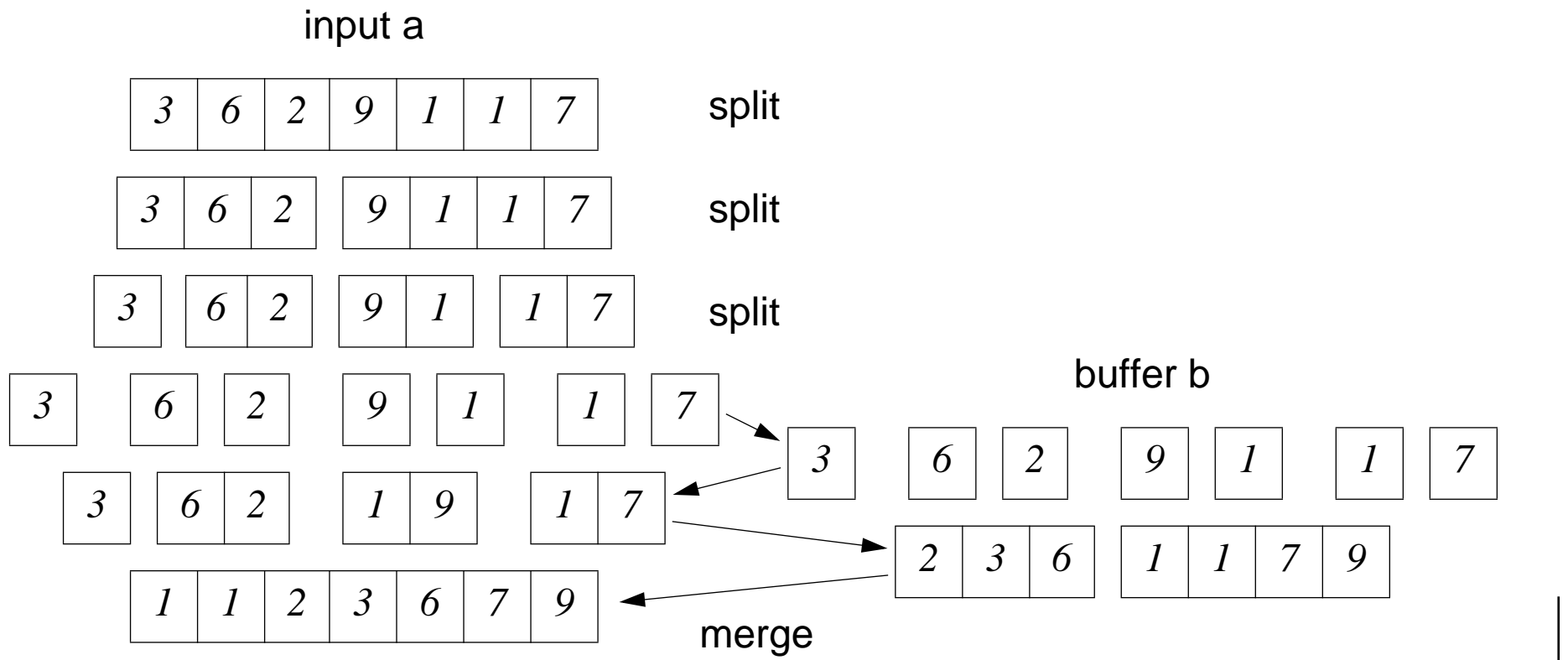
We can improve the performance, but make the program more complex.

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 * winner [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);
}
```

<b>The stack grows with each function call ...</b>				
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); 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 \* -

<i>Operation</i>		<i>Stack</i>		
<i>push</i>	6	6		
<i>push</i>	7	6	7	
<i>push</i>	3	6	7	3
<i>apply</i>	+	6	10	
<i>push</i>	2	6	10	2
<i>apply</i>	*	6	20	
<i>apply</i>	-	14		

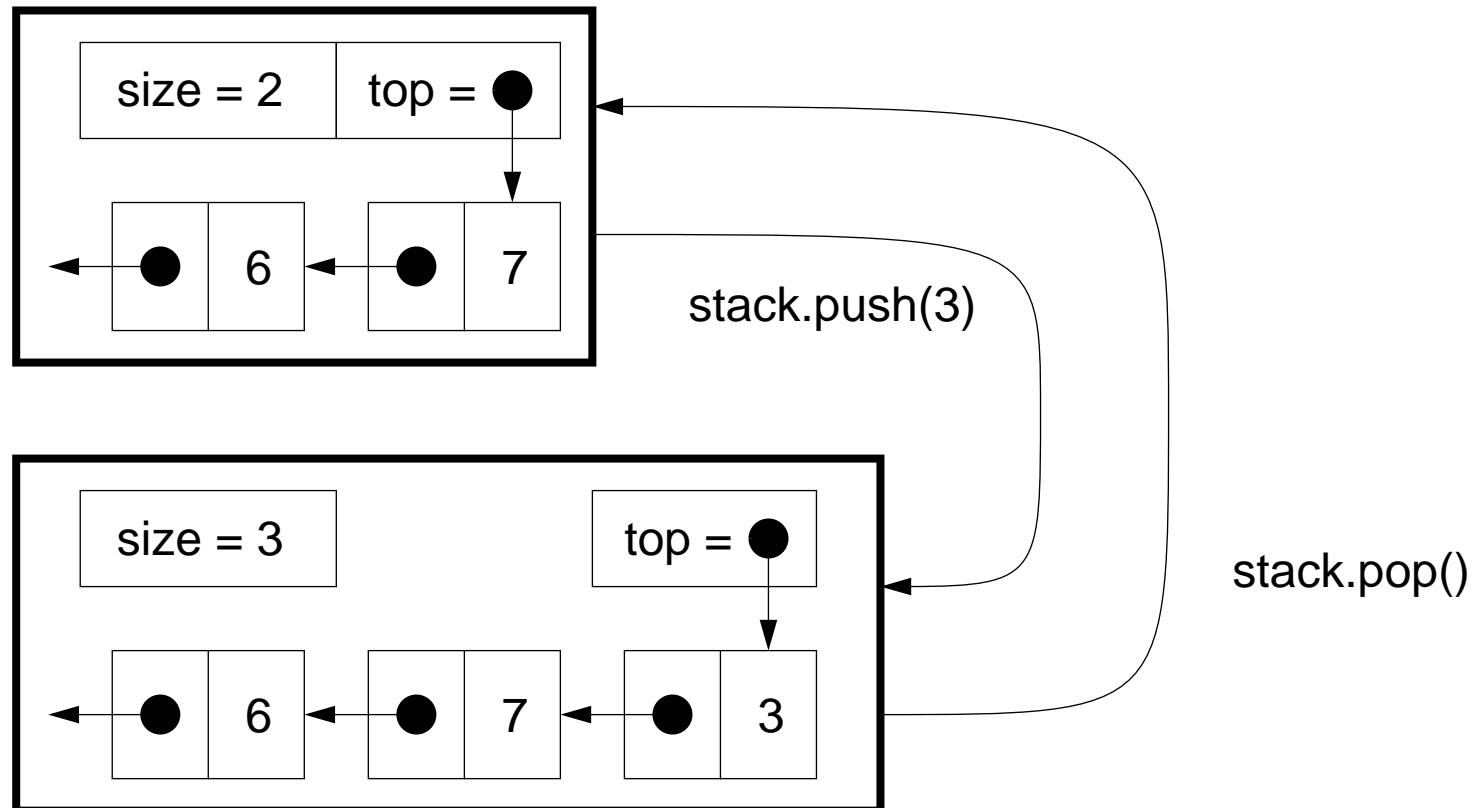
# *A Postfix Expression Interpreter*

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; }
    }
}
```

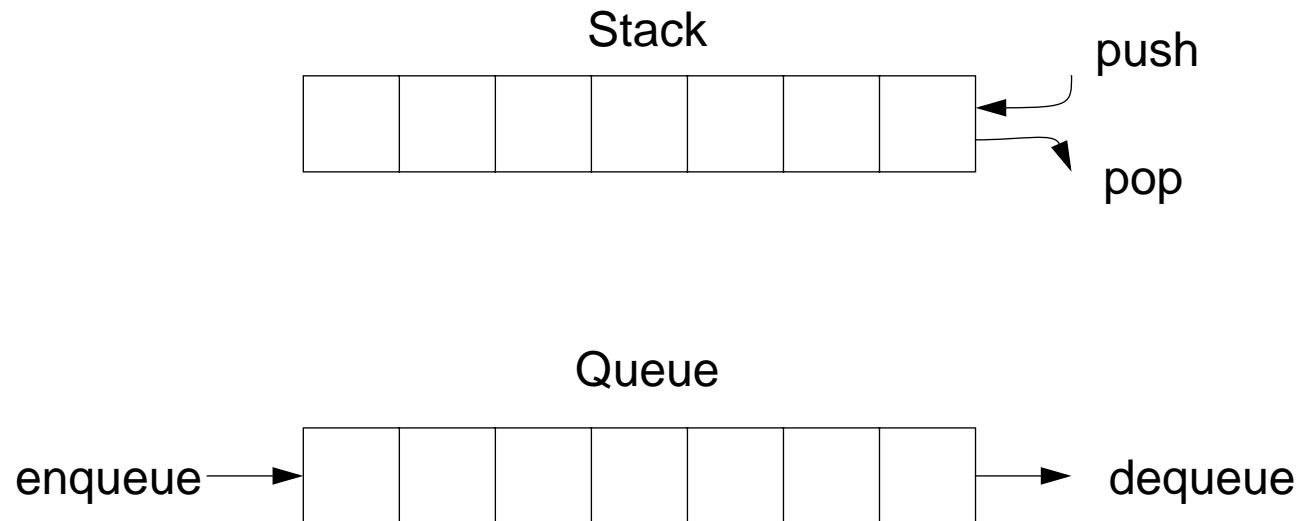
## Stacks as Linked Lists

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



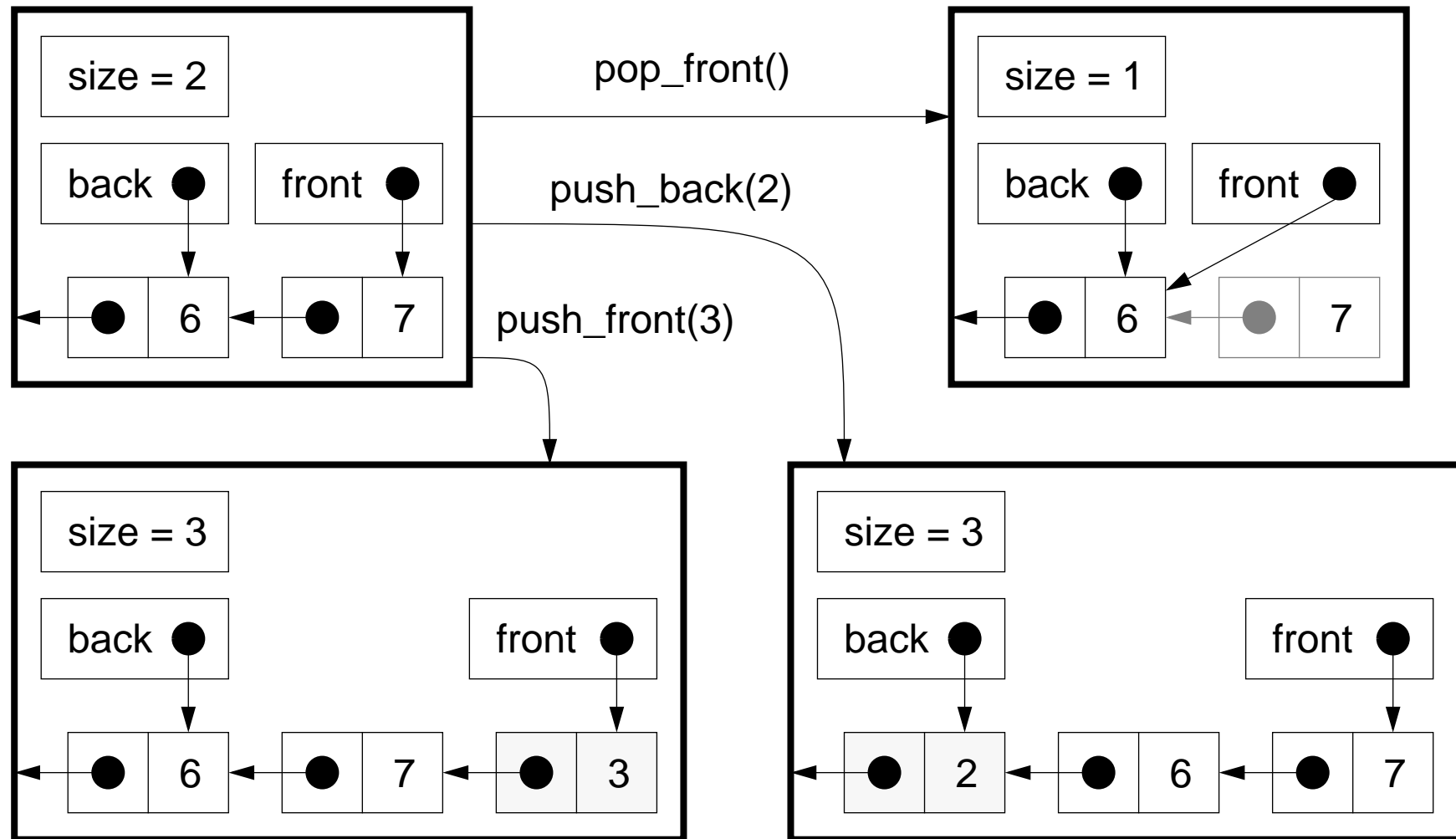
## Stacks, Queues and Linked Lists

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->next == 0

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

```

## *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 <i>not balanced</i>		

## 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); }

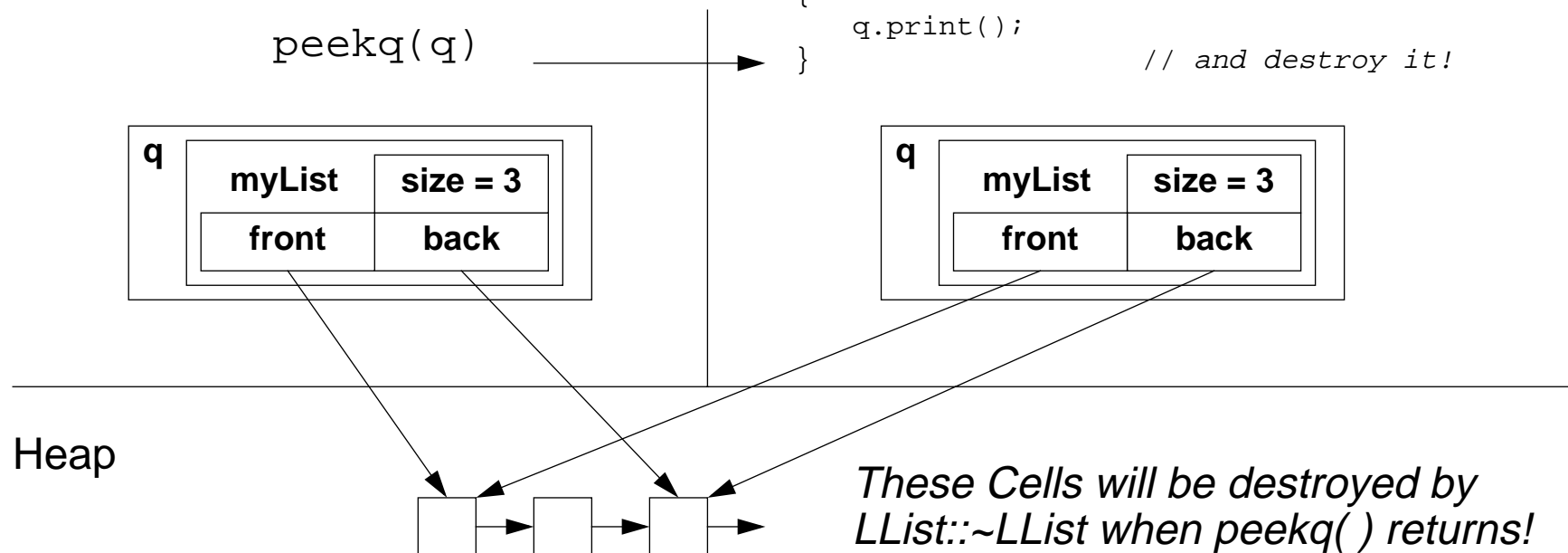
private:
    LList myList;
};
```

## 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.

Run-time Stack

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



## *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:

<i>Overloadable Operators</i>							
+	-	*	/	%	^	&	
-	!	,	=	<	>	<=	>=
++	--	<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=	=	*=
<<=	>>=	[ ]	( )	->	->*	new	delete

- ☞ 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: // ...  
};
```

## *IStream 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 `friends` 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 |   |   |   |
+---+---+---+
```

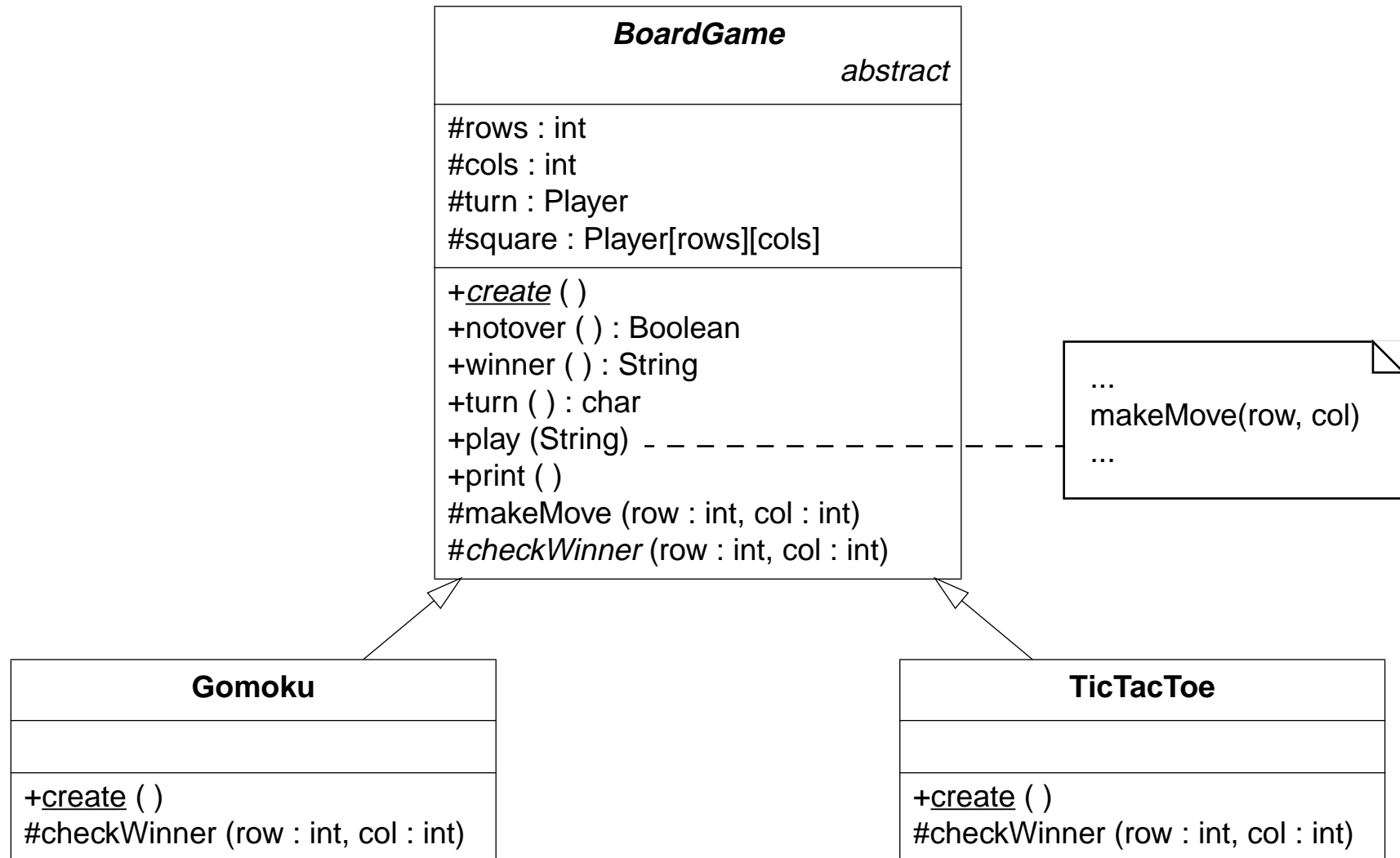
X plays: **bB**

```
      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 * winner [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:

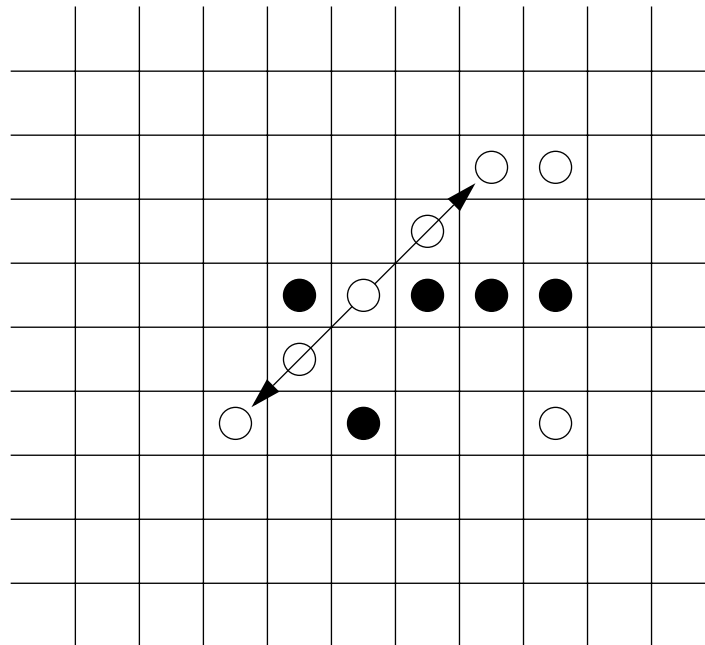
```
Gomoku::Gomoku (int r, int c, int ws) throw(xmsg) :
    BoardGame(r,c),           // Base members are initialized with r rows and c columns
    winningScore(ws)
{
    debug("calling Gomoku constructor"); // Notify when base/derived constructors are called
}
Gomoku::~Gomoku (void)
{
    debug("calling 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 <i>makefile</i>	Use <i>makefile</i> 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
- `rcc`                        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
- `rccmerge`               Incorporate changes from two RCS files into a third
- `rccclean`                Remove working files that have not been changed
- `rccfreeze`              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

The screenshot displays the CodeWarrior GUI debugger interface for a Gomoku game. The main window is titled "Gomoku" and shows the game board and the current state of the game.

**Breakpoint Window:** Shows a breakpoint set at "Gomoku:checkScore, line 63".

**File Explorer:** Shows the project structure for "Gomoku π", including source files (gmkMain.cpp, BoardGame.cpp, Gomoku.cpp, TicTacToe.cpp, String.cpp) and libraries (ANSI Libraries, Mac Libraries).

**Variable Watcher:** Shows the current state of variables:

Variable	Value
_turn	X
squaresLeft	5
square	0x0130F4AC
*square	0x0130F4A4
**square	X
winningScore	3

**Source Code Editor:** Shows the code for the checkScore function, with a breakpoint at line 63:

```

{
    int score = 1;
    int r=row, c=col;
    // check in this direction
    thisMove(r,c); // same as: (*thisMove)(r,c)
    while (inRange(r,c) && square[r][c] == square[row][col])
        // cerr << "Checking (" << row << ", " << col << ")"
        score++;
        thisMove(r,c);
}
// cerr << "Stopped at (" << row << ", " << col << ")"

// and now in that direction
r = row;
c = col;
thatMove(r,c);

```

**Game Board:** Shows the current state of the Gomoku board (3x3 grid) after 0 plays: bB and X plays: cA.

```

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

```

**Game Board:** Shows the current state of the Gomoku board (3x3 grid) after 0 plays: bA.

```

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

```

## 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]
...						



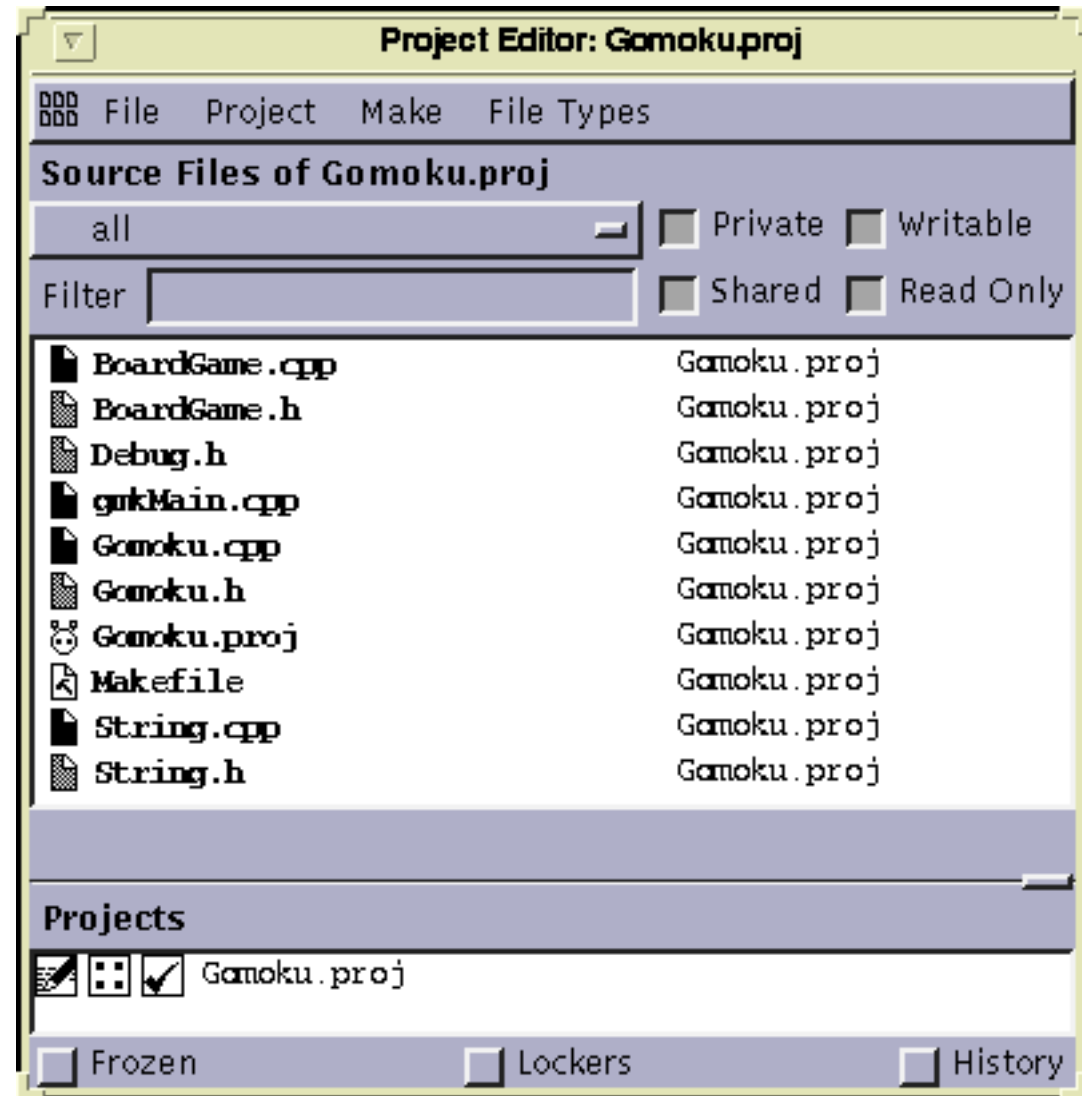
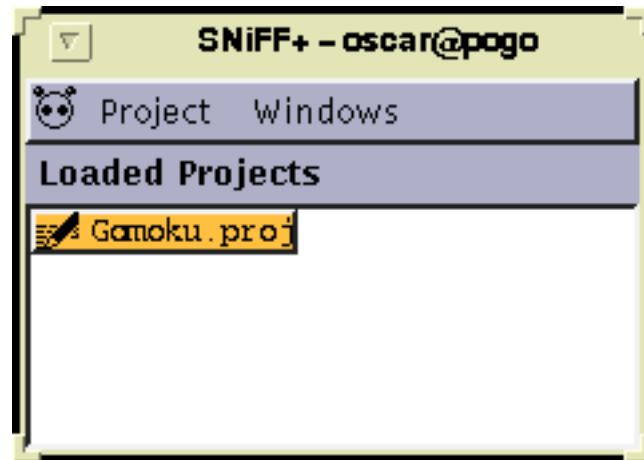
## SNiFF+

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+ Source Editor

```
Source Editor: Gomoku.proj - BoardGame.cpp
File Edit Positioning Make Info Class Exec Inspect History

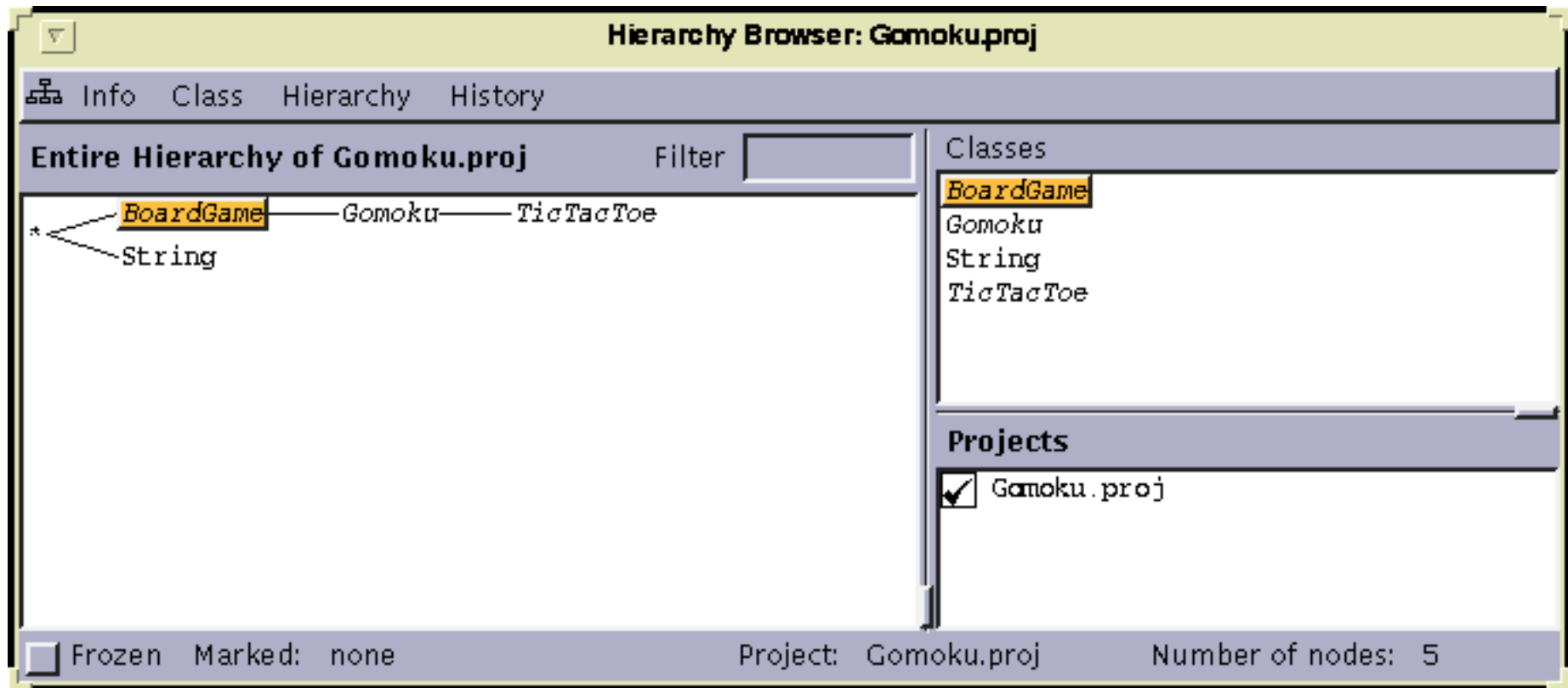
/*
   The default assumes you can mark any empty square.
   Override this to implement a different logic for valid m
*/
void
BoardGame::makeMove (int row, int col) throw (xmsg)
{
    if (square[row][col] == nobody) { // not already m
        square[row][col] = _turn; // mark the squa
        squaresLeft--;
        _turn = (_turn == X) ? O : X; // switch curren
    } else {
        throw (xmsg("Square already occupied"));
    }
}

// Pretty print the current state of the game:
void
BoardGame::print (void)
{

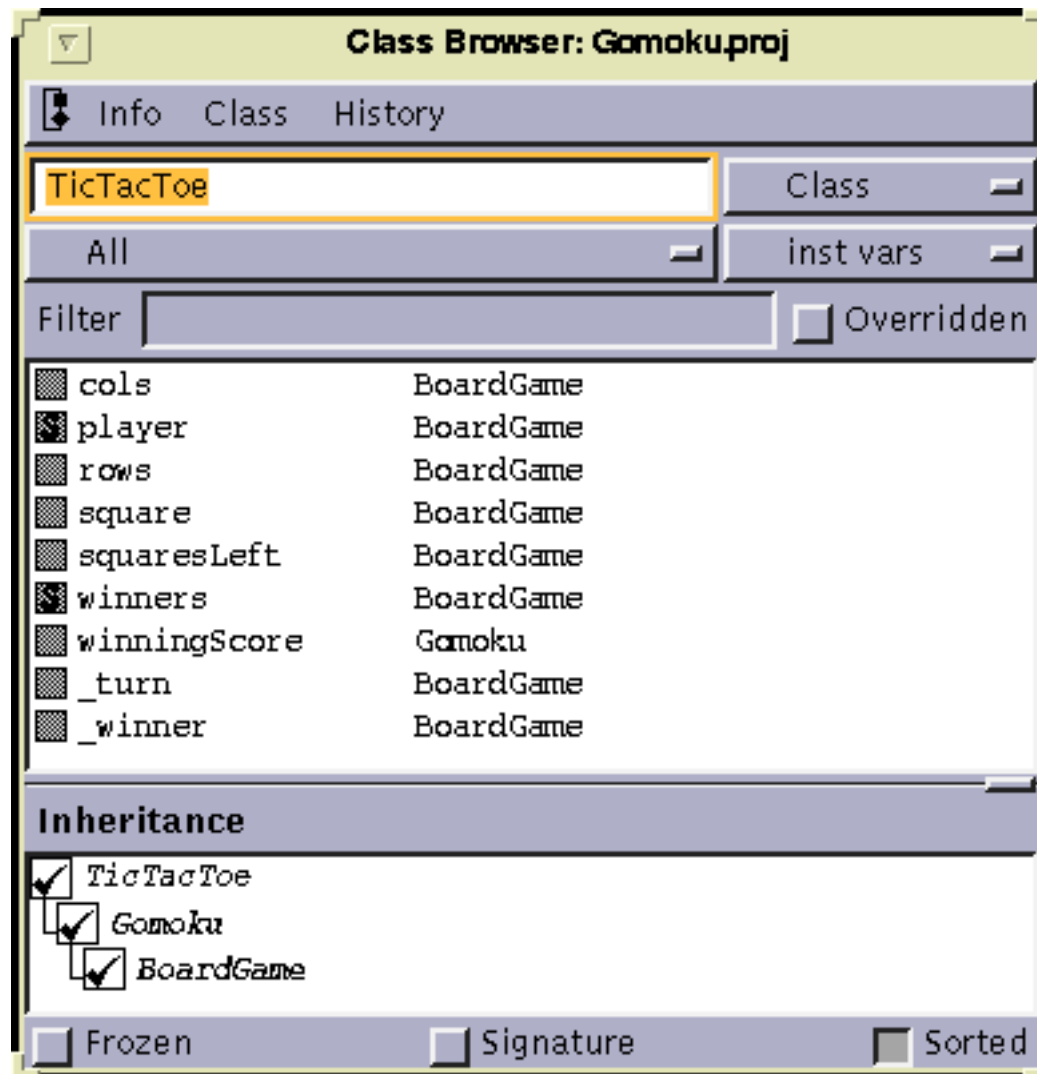
All Classes
BoardGame (mi) BoardGame
~BoardGame (mi) BoardGame
checkWinner (mi) BoardGame
inRange (mi) BoardGame
makeMove (mi) BoardGame
notover (mi) BoardGame
operator<< (f)
play (mi) BoardGame
print (mi) BoardGame
turn (mi) BoardGame
winner (mi) BoardGame

Frozen File: BoardGame.cpp - /home/scg/oscar/Courses/P2/Examples/Gomoku
```

## *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

The screenshot shows the Purify vPurifyGomoku application window. The title bar reads "Purify: vPurifyGomoku". The menu bar includes "File", "View", "Actions", "Options", and "Help". Below the menu bar is a toolbar with icons for home, back, forward, print, stop, help, and zoom. The main content area displays the following text:

```

▶ Purify instrumented vPurifyGomoku (pid 27091 at Wed Apr 10 14:52:40 1996)
▶ Current file descriptors in use: 5
▼ Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)
  ▼ Purify Heap Analysis (combining suppressed and unsuppressed chunks)
    Chunks      Bytes
    Leaked      0      0
    Potentially Leaked 0      0
    In-Use      1      8
    -----
    Total Allocated    1      8
  ▼ Program exited with status code 0.
    * Basic memory usage (including Purify overhead):
      387352 code
      72568 data/bss
      8192 heap (peak use)
      5120 stack
    * Shared library memory usage (including Purify overhead):
      153588 libm.so.1_pure_p1_c0_032_54.so.1 (shared code)
      5944 libm.so.1_pure_p1_c0_032_54.so.1 (private data)
      35397 libw.so.1_pure_p1_c0_032_54.so.1 (shared code)
  
```

## *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. An Introduction to Java

### Overview

- ❑ Java vs. C++
- ❑ Java language features: packages, classes, exceptions ...
- ❑ The Java API
- ❑ Applets

### Texts:

- ❑ David Flanagan, *Java in a Nutshell*, O'Reilly, 1996
- ❑ Mary Campione and Kathy Walrath, *The Java Tutorial*, The Java Series, Addison-Wesley, 1996

### On-line resources:

- ❑ Locally installed Java resources (on-line tutorial, language spec, etc):  
<http://www.iam.unibe.ch/~scg/Resources/Java/>

# Java

Language design influenced by existing OO languages (C++, Smalltalk ...):

- ❑ Strongly-typed, concurrent, pure object-oriented language
- ❑ Syntax, type model influenced by C++
- ❑ Single-inheritance but multiple subtyping
- ❑ Garbage collection

Innovation in support for network applications:

- ❑ Standard API for language features, basic GUI, IO, concurrency, network
- ❑ Compiled to bytecode; interpreted by portable abstract machine
- ❑ Support for native methods
- ❑ Classes can be dynamically loaded over network
- ❑ Security model protects clients from malicious objects

*Java applications do not have to be installed and maintained by users*

# *Java and C++ — Similarities and Extensions*

Java resembles C++ only superficially:

Similarities:

- primitive data types (in Java, platform independent)
- syntax: control structures, exceptions ...
- classes, visibility declarations (`public`, `private`)
- multiple constructors, `this`, `new`
- types, type casting

Extensions:

- garbage collection
- standard classes (Strings, collections ...)
- packages
- standard abstract machine
- final classes

## *Java and C++ — Simplifications*

Whereas C++ is a hybrid language, Java is a pure object-oriented language that eliminates many of the complex features of C++:

Simplifications:

- ❑ no pointers — just references
- ❑ no functions — can declare `static` methods
- ❑ no global variables — can declare `public static` variables
- ❑ no destructors — garbage collection and `finalize` methods
- ❑ no linking — dynamic class loading
- ❑ no header files — can define `interface`
- ❑ no operator overloading — only method overloading
- ❑ no member initialization lists — `super` constructor can be called
- ❑ no preprocessor — `static final` constants and automatic inlining
- ❑ no multiple inheritance — can implement multiple interfaces
- ❑ no structs, unions, enums — typically not needed
- ❑ no templates — but generics will likely be added ...

# The “Hello World” Program

helloWorld objects can be instantiated by any client

only classes can be declared (pure OO)

class methods behave like global functions

Every program must have a `main` method declared in some class

`String` is a standard class

```
// My first Java program!  
public class helloWorld {  
    public static void main (String argv[]) {  
        System.out.println("Hello World");  
    }  
}
```

a class in the package `java.lang`

a public class variable

a public method

# Packages

A Java program is a collection of classes organized into *packages*

- ❑ At least one class must have a `public static void main()` method
- ❑ The first statement of a source file may declare the package name:

```
package games.tetris;
```

- ❑ Source files (e.g., `helloWorld.java`) are compiled to bytecode files (e.g., `helloWorld.class`), one for each target class
- ❑ Class files must be stored in subdirectories corresponding to the package hierarchy
- ❑ When using classes, either the full package name must be given:

```
java.lang.System.out.println("Hello World");
```

or classes from the package may be *imported*:

```
import java.lang.*; // this package is always imported by default
```

- ❑ Class names are usually capitalized for readability:

```
a.b.c.d.e.f(); // which is the name of the class?!
```

# Java Basics

Java's primitive data types and control statements resemble those of C/C++:

## Primitive Data Types:

```
boolean byte char double float int long short void
```

## Literals:

```
false null true
```

## Control flow:

```
if ( boolean ) { Statements } else { Statements }  
for ( boolean ) { Statements }  
while ( boolean ) { Statements }  
do { Statements } while ( boolean )  
switch ( variable ) {  
    case label : Statements;  
        break; ...  
    default : ... break;  
}
```



# Classes and Objects

The encapsulation boundary is a class (not an object):

```
public class Point {
    private double x, y;    // not accessible to other classes (even subclasses)

    // constructors:
    public Point (double xCoord, double yCoord) { x = xCoord; y = yCoord; }
    public Point (Point p) { x = p.x; y = p.y; } // can access private data here

    // public methods:
    public double getX ( )           { return x; }
    public void   setX (double xCoord) { x = xCoord; }
    public double getY ( )           { return y; }
    public void   setY (double yCoord) { y = yCoord; }
    public double distance ( )       { return Math.sqrt(x*x + y*y); }
}
```

In pure OOLs, (non-primitive) objects are passed by reference, not by value:

```
int a = 3, b = 4;           // a and b are primitive objects
Point p1 = new Point(a,b); // p1 is a reference to an object (NB: a & b coerced!)

int c = a;                   // c gets value of a
c = 8;                        // c gets new value; a is unchanged

Point p2 = p1;               // p2 refers to p1
Point p3 = new Point(p1); // p3 is a copy of p1
p2.setX(c);                  // The object p1 and p2 refer to is modified
```

## *Garbage Collection*

In Java (as in Smalltalk and Eiffel), objects no longer referred to are automatically garbage-collected:

- ❑ no need to explicitly `delete` objects
- ❑ no destructors need to be defined
- ❑ no need to write reference-counting code
- ❑ no danger of accidentally deleting objects that are still in use

You can still exercise extra control:

- ❑ Cleanup activities can be specified in a `finalize` method
  - ☞ useful for freeing external resources (files, sockets etc.)
- ❑ Objects you no longer need can be explicitly “forgotten”
  - ☞ you can explicitly forget objects by assigning the value `null` to a variable (this is the initial value of declared, but unassigned variables)

# Inheritance

A subclass *extends* a superclass, inheriting all its features, and possibly overriding some or adding its own:

```
public class Circle extends Point {
    private double r;

    public Circle (double xCoord, double yCoord, double radius) {
        super(xCoord, yCoord);    // call Point constructor
        r = radius;
    }

    public Circle (Circle c) {
        super(c);                // call Point constructor with c as Point
        r = c.r;
    }

    public double getR ( )           { return r; }
    public void   setR (double radius) { r = radius; }
    public double distance ( )       { return super.distance() - r; }
}
```

Public superclass features can always be accessed, even if overridden.

## Dynamic Binding

One of the key features of object-oriented programming is *dynamic binding* — the actual method that will be executed in response to a request depends on the dynamic type of target, not the static type of the reference:

```
Point p = new Circle(5, 12, 4);  
System.out.println("p.distance() = " + p.distance());
```

*yields:*

```
p.distance() = 9
```

In pure OOLs, all methods are dynamically bound by default. Static binding is the exception:

- ❑ `static` methods belong to classes, so are statically bound
- ❑ `private` methods have purely local scope
- ❑ `final` methods cannot be overridden, so are statically bound

# Downcasting

Dynamic binding can cause type information to be lost:

```
Point p = new Circle(5, 12, 4);    // p refers to a Circle – upcast ok
Circle c1 = p;                    // compile-time error! – can't downcast
```

Type information can be recovered at run-time by explicit tests and casts:

```
if (p instanceof Circle) {        // run-time test
    c1 = (Circle) p;              // explicit run-time downcast ok
}
```

An attempt to cast to an invalid type will raise an exception at run-time:

```
p = new Point(3,4);
c1 = (Circle) p;                  // invalid downcast raises run-time exception
```

## Feature Visibility

Features can be declared with different degrees of visibility:

- ❑ `private` — accessible only within the class body
- ❑ `public` — accessible everywhere
- ❑ `protected` — accessible to subclasses *and* to members of the same package
  - ☞ allows access to cooperating classes
- ❑ default (no modifier) — accessible throughout the package only
  - ☞ allows package access but prevents all external access

## Modifiers

In addition to feature visibility, modifiers can specify several other important attributes of classes, methods and variables:

- ❑ `abstract` — unimplemented method; class must also be declared `abstract`
  - ☞ method signature is followed by semi-colon instead of body
- ❑ `final` — class/method/variable cannot be overridden by subclass
- ❑ `static` — method/variable belongs to class, not instances; implicitly `final`
- ❑ `native` — method implemented in some other language, usually C

# Exceptions

A class must declare which exceptions it throws, or it must catch them:

```
public class TryException {
    public static void main(String args[]) {
        try {
            alwaysThrow(0);           // NB: we never get past this point
            alwaysThrow("hello");
        } catch (NumException e) {
            System.out.println("Got NumException: " + e.getMessage());
        } catch (StringException e) {
            System.out.println("Got StringException: " + e.getMessage());
        } finally {
            System.out.println("Cleaning up");
        }
    }
    public static void alwaysThrow(int arg) throws NumException {
        throw new NumException("don't call me with an int arg!");
    }
    public static void alwaysThrow(String arg) throws StringException {
        throw new StringException("don't call me with a String arg!");
    }
}
```



## Defining Exceptions

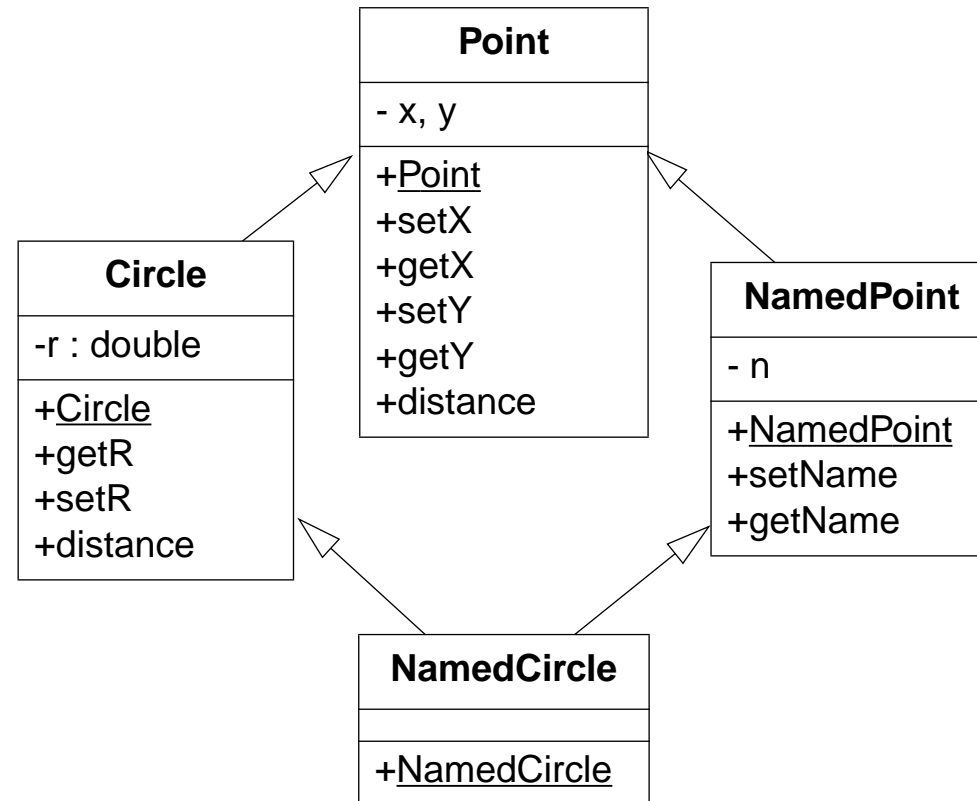
You can define your own exception classes that inherit from `Exception`. Typically, you will only define constructors:

```
// Most exception classes look like this:
public class NumException extends Exception {
    public NumException() { super(); }
    public NumException(String s) { super(s); }
}

public class StringException extends Exception {
    public StringException() { super(); }
    public StringException(String s) { super(s); }
}
```

# Multiple Inheritance

Although conceptually elegant, multiple inheritance poses significant pragmatic problems for language designers:



*Which version of distance() should be inherited by NamedCircle?*

# Interfaces

An interface declares methods but provides no implementation:

```
interface Named {  
    public void setName (String name);  
    public String getName ( );  
}
```

A Java class can extend at most one superclass, but may implement multiple interfaces:

```
public class NamedCircle extends Circle implements Named {  
    private NamedObject n; // object composition vs. inheritance  
    public NamedCircle (double xCoord, double yCoord, double radius, String name) {  
        super(xCoord, yCoord, radius); // call Circle constructor  
        n = new NamedObject(name); // compose a NamedObject instance  
    }  
    public void setName (String name) { n.setName(name); } // forwarding  
    public String getName ( ) { return n.getName(); }  
}
```

Reusable behaviour can be encapsulated as a separate class:

```
public class NamedObject implements Named {  
    private String n;  
    public NamedObject (String name) { n = name; }  
    public void setName (String name) { n = name; }  
    public String getName ( ) { return n; }  
}
```

# Overriding and Overloading

Overridden methods have the same name and argument types

Overloaded methods have the same name but different argument types

```
public class A {
    public void f (float x)    { System.out.println("A.f(float)"); }
    public void g (float x)    { System.out.println("A.g(float)"); }
}

public class B extends A {
    public void f (float x)    { System.out.println("B.f(float)"); }
    public void g (int x)      { System.out.println("B.g(int)"); }
}
```

Overloaded methods are disambiguated by their arguments:

```
B b = new B();    // both dynamic and static type B
A a = b;         // static type is A but dynamic type is B

b.f(3.14f);      // B.f(float) -- overridden
b.f(3);          // B.f(float) -- 3 is converted to 3.0
b.g(3.14f);      // A.g(float) -- not overridden
b.g(3);          // B.g(int) -- overloaded

a.f(3.14f);      // B.f(float) -- overridden
a.f(3);          // B.f(float) -- 3 is converted to 3.0
a.g(3.14f);      // A.g(float) -- not overridden
a.g(3);          // A.g(float) -- g(int) does not exist in SuperClass!
```

# Arrays

Arrays are polymorphic objects:

- ❑ Can declare arrays of any type

```
int[] array1;
```

```
MyObject s[];
```

- ❑ Can build array of arrays

```
int a[][] = new int[10][3];
```

```
a.length --> 10
```

```
a[0].length --> 3
```

Creating arrays

- ❑ An empty array:

```
int list[] = new int [50];
```

- ❑ Pre-initialized:

```
String names[] = { "Marc", "Tom", "Pete" };
```

- ❑ Cannot create static compile time arrays

```
int nogood[20]; // compile time error
```

# Arrays and Generics

Arrays are the only polymorphic containers in Java:

```
Point [] pa = new Point[3];
pa[0] = new Point(3,4);
pa[1] = new Point(5,12);
Point p = pa[0];           // ok -- pa is an array of Points
```

It is not possible to program other kinds of polymorphic containers:

```
Stack s = new Stack();           // defined in package java.util
s.push(pa[0]);
s.push(pa[1]);
// p = s.pop();                 // compile-time error -- s.pop() returns an Object
p = (Point) s.pop();            // ok -- run-time cast
```

## *The Java API*

**java.lang.** contains essential Java classes, including numerics, strings, objects, compiler, runtime, security, and threads. This is the only package that is automatically imported into every Java program.

**java.awt.** Abstract Windowing Toolkit

**java.applet.** enables the creation of applets through the Applet class.

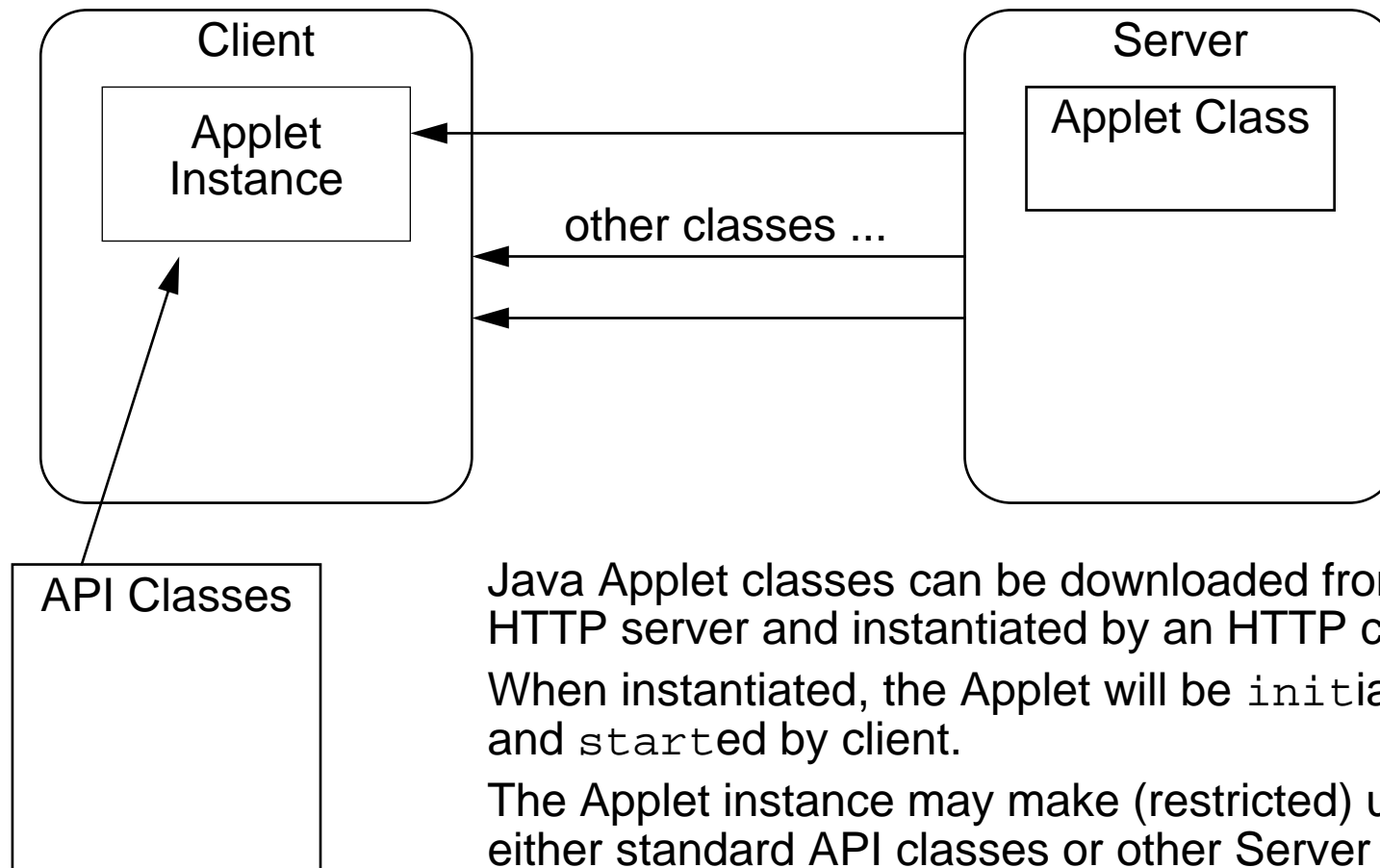
**java.io.** provides classes to manage input and output streams to read data from and write data to files, strings, and other sources.

**java.util.** contains miscellaneous utility classes, including generic data structures, bit sets, time, date, string manipulation, etc.

**java.net.** provides network support, including URLs, TCP sockets, UDP sockets, IP addresses, and a binary-to-text converter.

*And many others ...*

# Applets



Java Applet classes can be downloaded from an HTTP server and instantiated by an HTTP client. When instantiated, the Applet will be *initialized* and *started* by client.

The Applet instance may make (restricted) use of either standard API classes or other Server classes to be downloaded dynamically.

*NB:* objects are *not* downloaded, only classes!



## *The Hello World Applet*

### The simplest Applet:

```
// From Java in a Nutshell, by David Flanagan.
import java.applet.*; // To extended Applet
import java.awt.*;    // Abstract windowing toolkit

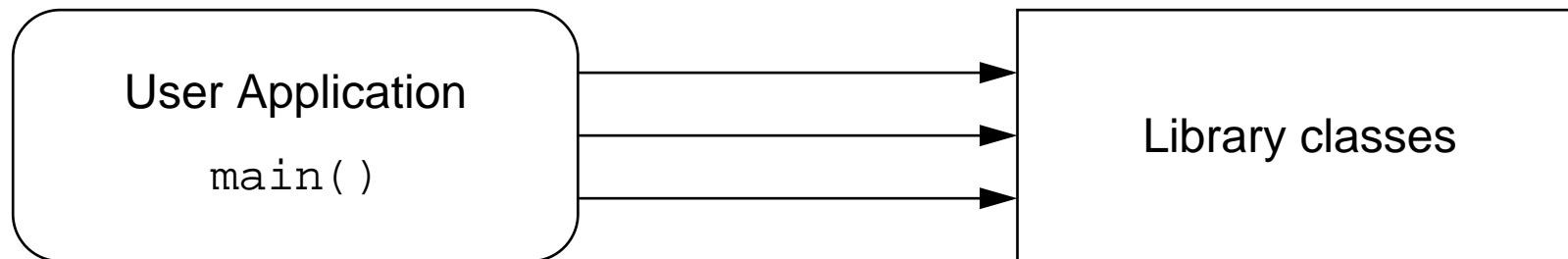
public class HelloApplet extends Applet {
    // This method displays the applet.
    // The Graphics class is how you do all drawing in Java.
    public void paint(Graphics g) {
        g.drawString("Hello World", 25, 50);
    }
} // NB: there is no main() method!
```

### HTML applet inclusion:

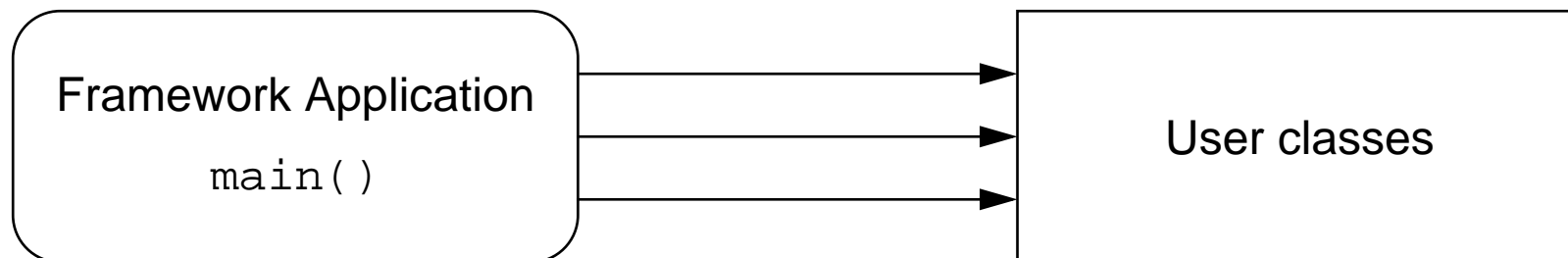
```
<title>Hello Applet</title>
<hr>
<applet codebase="HelloApplet.out" code="HelloApplet.class" width=200 height=200>
</applet>
<hr>
<a href="HelloApplet.java">The source.</a>
```

## Frameworks vs. Libraries

In traditional application architectures, user applications make use of library functionality in the form of procedures or classes:



A framework reverses the usual relationship between generic and application code. Frameworks provide *both* generic functionality *and* application architecture:



*Essentially, a framework says: "Don't call me — I'll call you."*

## *Standalone Applets*

An Applet is just a user object instantiated by the Applet framework:

```
// Adapted from Java in a Nutshell, by David Flanagan.
// A simple example of directly instantiating an Applet.

import java.applet.*;
import java.awt.*;

public class HelloStandalone {
    public static void main(String args[]) {
        Applet applet = new HelloApplet();
        Frame frame = new AppletFrame("Hello Applet", applet, 300, 300);
    }
}

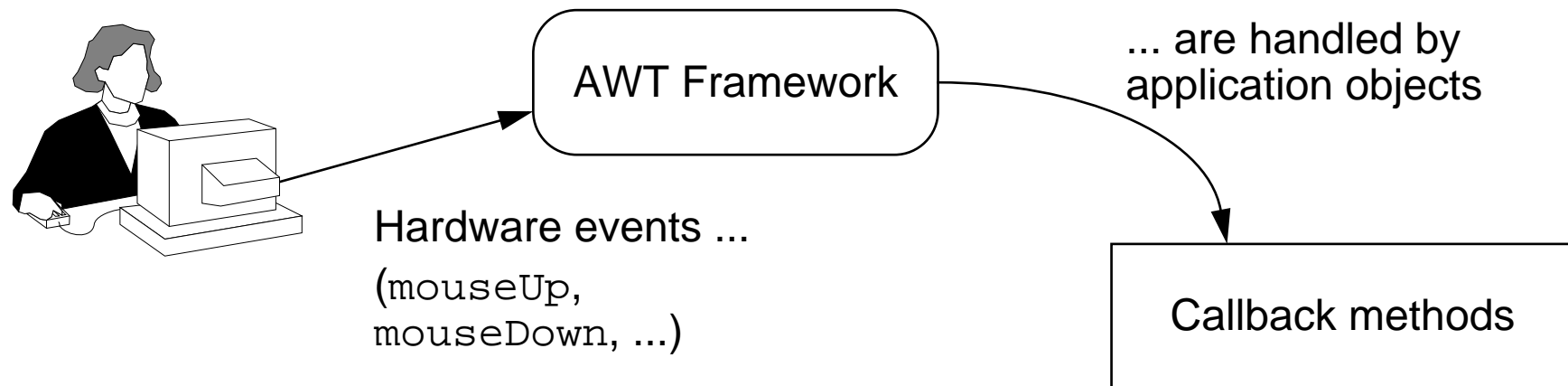
class AppletFrame extends Frame {
    public AppletFrame(String title, Applet applet, int width, int height) {
        super(title); // Create the Frame with the specified title.

        this.add("Center", applet); // Add the applet to the window.
        this.resize(width, height); // Set the window size.
        this.show(); // Pop it up.

        applet.init(); // Initialize and start the applet.
        applet.start();
    }
}
```

# Events

Instead of actively checking for GUI events, you can define callback methods that will be invoked when your GUI objects receive events:



`Component` is the superclass of all GUI components (including `Frame` and `Applet`) and defines all the callback methods that components must implement.

## *The Scribble Applet*

Scribble is a simple Applet that supports drawing by dragging the mouse:

*NB: This example uses the (deprecated) Java 1.0 event model!*

```
// Adapted from Java in a Nutshell, by David Flanagan.
import java.applet.*;
import java.awt.*;

public class Scribble extends Applet {
    private int last_x = 0;
    private int last_y = 0;
    private Button clear button;

    // Called to initialize the applet.
    public void init() {
        this.setBackground(Color.white);           // Set the background colour
        clear_button = new Button("Clear");        // Create a Button
        clear_button.setForeground(Color.black);
        clear_button.setBackground(Color.lightGray);
        this.add(clear_button);                    // Add it to the Applet
    }
}
```

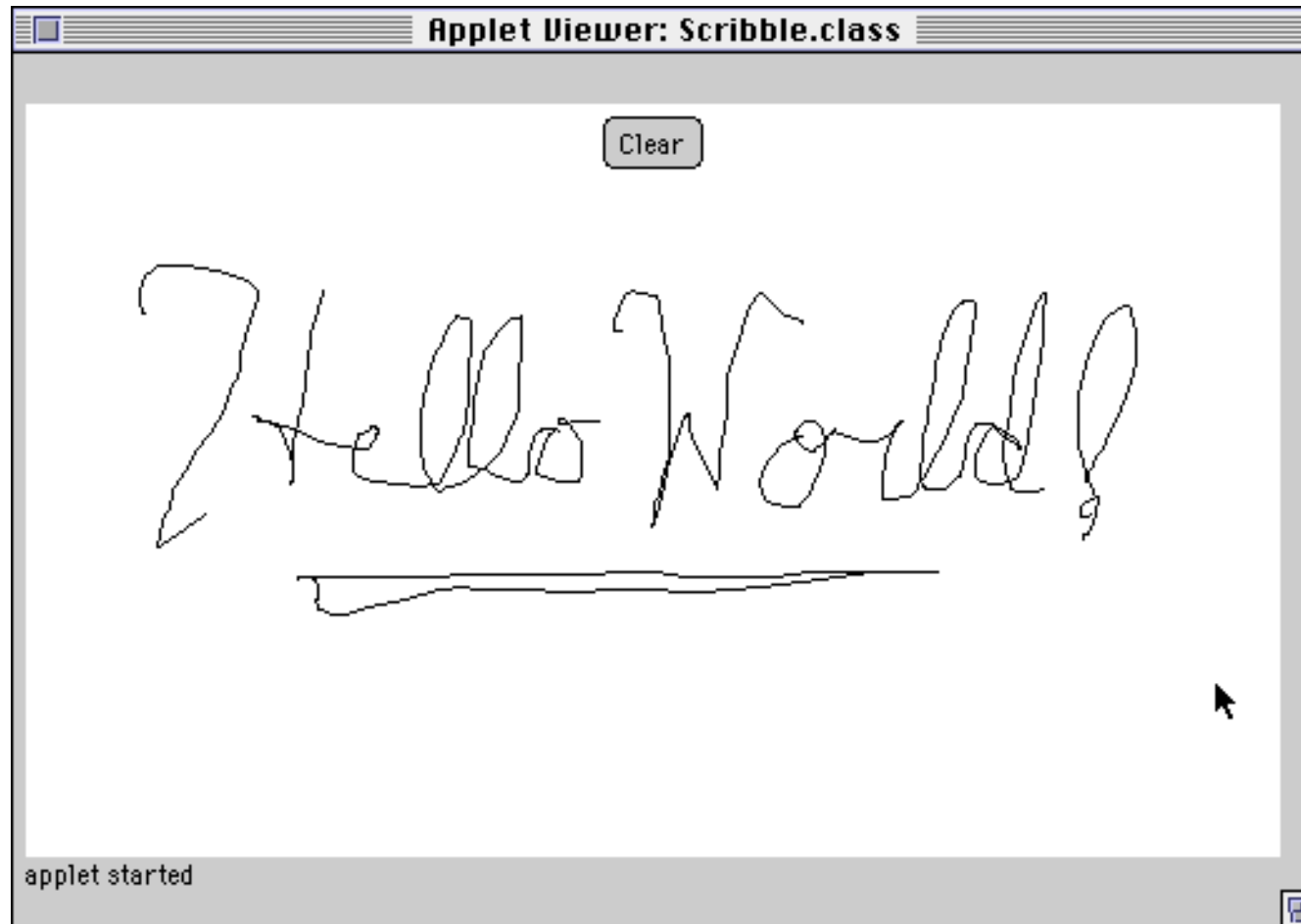
## Responding to Events

```
// Called when the user clicks the mouse to start a scribble
public boolean mouseDown(Event e, int x, int y) {
    last_x = x; last_y = y; return true; // Always return true if event handled
}

// Called when the user scribbles with the mouse button down
public boolean mouseDrag(Event e, int x, int y) {
    Graphics g = this.getGraphics();
    g.setColor(Color.black); g.drawLine(last_x, last_y, x, y);
    last_x = x; last_y = y; return true;
}

// Called when the user clicks the button
public boolean action(Event event, Object arg) {
    // If the Clear button was clicked on, handle it.
    if (event.target == clear_button) {
        Graphics g = this.getGraphics();
        Rectangle r = this.bounds();
        g.setColor(this.getBackground());
        g.fillRect(r.x, r.y, r.width, r.height);
        return true;
    } // Otherwise, let the superclass handle it.
    else return super.action(event, arg);
}
}
```

## Running the Scribble Applet



## Summary

### You should know the answers to these questions:

- What are the similarities and differences between Java and C++?
- What role do packages play in Java?
- Why should a subclass constructor call its `super` constructor?
- What is dynamic binding? Why are `static` methods not dynamically bound?
- What is the difference between overriding and overloading?
- Why doesn't an Applet need a `main()` method?
- What are events and callbacks?

### Can you answer the following questions?

- ✎ *How can an object gain access to a `private` instance variable of another object?*
- ✎ *Why does Java (need to) support explicit type-casting?*
- ✎ *What is the difference between an `interface` and an `abstract class`?*



# 11. 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:

<i>What's pointed to is constant</i>		<i>Pointer is constant</i>	
	<code>char *</code>		<code>p = "Hello";</code>
<code>const</code>	<code>char *</code>		<code>p = "Hello";</code>
	<code>char *</code>	<code>const</code>	<code>p = "Hello";</code>
<code>const</code>	<code>char *</code>	<code>const</code>	<code>p = "Hello";</code>

## 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?*