Introduction

Background on C++

C++ vs. C

C++ vs. Java

What is C++?

In fifteen words or less:

> A superset of C that supports type extensibility and object-oriented programming.

Bjarne Stroustrup, the designer of C++, says:

> "C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer."
>
> "C++ is designed to:
>
> Be a better C
>
> Support data abstraction
>
> Support object-oriented programming"
>
> "As close to C as possible, but no closer."

What is C++?, continued

C++ is designed to handle large, complex systems.

The primary tools in C++ for coping with complexity are strong compile-time type checking and encapsulation of data inside objects.

C is a language that's close to the machine. C++ is designed to be close to the problem to be solved, to allow a direct and concise solution.

With respect to C, C++ has relatively few new keywords, but has a great deal of new syntax.

A driving factor in the design of C++ is that you "pay" for only what you use.

Why choose C++?

C++ provides strong support for object-oriented programming.

Because C++ is roughly a superset of C, it builds on existing C language skills.

C++ fits well with existing C programming environments, especially with respect to libraries.

A C++ program can be as fast and memory efficient as an equivalent program in C.

C++ is a proven language. It has been used successfully for many large applications.

C++ is well documented. Many good books on C++ have been published. There's a vast amount of information on the web about C++.

The C++ time line

May 1979: Bjarne Stroustrup, a researcher at Bell Labs, took a number of ideas from Simula-67 and produced a dialect of C called "C with Classes".

August 1983: First C++ implementation in use.

December 1983: Name "C++" coined by Rick Mascitti.

February 1985: First external release of C++ (version "e").

October 1985: Version 1.0 of C++ (cfront) released; 1st edition of Stroustrup's C++ book published.

June 1989: Version 2.0 of cfront released.

1989: ANSI XJ316 formed to begin standardization.

1991: Version 3.0 of C++ released.

1995: Draft ANSI standard completed.

1998: ISO/IEC standard approved (14882:1998). Sometimes called C++98.

2003: Technical Corrigendum 1 for standard due (14882:2003)

200X: C++0x

C++ evolved informally and pragmatically. The design was driven to a large extent by user feedback.

The Bad News about C++

C++ is a chameleon of a language. It tries to:

Be fast

Be memory-efficient

Be close to the machine

Be close to the problem to be solved

Support user-defined types

Support object-oriented programming

Support development of very large systems

It succeeds at all of these goals, but at the cost of complexity.

Some say that C++ has a fractal-like quality.

The C++ Standard Library has a very limited scope.

C++ vs. C

C++ is in essence a superset of C.

C++ uses C's:

    Data types
    Operators
    Control structures
    Preprocessor
    And more...

Most C code will compile as C++.

Almost everything you know about C is directly applicable in C++.

The executable instructions generated for a body of C++ code are generally as fast and memory-efficient as the same code in C.

Just like C, C++ source files are compiled into object files that are then linked to produce an executable program.

The name C++ was chosen to signify the evolutionary nature of the changes from C. C++ was not called "D" because it is an extension of C and doesn't try to remedy problems in C.

C++ vs. Java

In 1990 Sun Microsystems formed a group called the Green project. The initial focus was to create a software development environment for consumer electronics products.

C++ was the initial choice for a language for Green but frustration with C++ led to a new language, Oak, designed by James Gosling.

Oak is now called Java.

Java borrows heavily from C++ in many ways. Among them:

- Class definition syntax
- Class/object relationship
- Data types
- Operators
- Control structures
- Compile-time type checking philosophy

C++ vs. Java, continued

Here are some things from Java that you'll probably miss in C++:

Speedy compilation

Garbage collection

Vast standard library

Easy to use 3rd-party libraries

Language support for multi-threading

Reflection capabilities

Security model

.class files

Class loading

C++ vs. Java, continued

Here are some things about C++ that you may like better than Java:

- Faster execution (as a rule)
- Few compromises on encapsulation and type safety
- Classes and functions with parameterized types
- Operator overloading for user-defined types
- Multiple inheritance
- Readily usable with C libraries
- The IO Streams facility
- A very interesting set of container classes
- Everything you "love" from C, including:
    - Closeness to the hardware
    - Global functions and variables
    - Preprocessor
    - No restrictions on file names and directory structure

Class and Object Basics in C++

Class definition

Working with objects

class vs. struct

this

Source file organization

Class definition in C++

The form of class definitions in C++ is very similar to Java. Here is a trivial Java class representing a "counter":

```
public class Counter {
    private int itsCount = 0;
    private String itsName;

    public Counter(String name) { itsName = name; }
    public void bump()              { itsCount++; }
    public void print()
    {
        System.out.println(itsName + "'s count is " + itsCount);
    }
}
```

An equivalent class definition in C++:

```
class Counter {
  private:
    int itsCount;
    string itsName;           // 'string' is from std. library
  public:
    Counter(string name) { itsCount = 0; itsName = name; }
    void bump()              { itsCount++; }
    void print()  {
        printf("%s's count is %d\n", itsName.c_str(), itsCount);
    }
};
```

Class definition in C++, continued

For reference:

```
class Counter {
  private:
    int itsCount;
    string itsName;
  public:
    Counter(string name) { ... }
    void bump() { itsCount++; }
    void print()   { ... }
};
```

Points to note:

itsCount and itsName are called *data members*.

bump() and print() are called *member functions*.

A public: or private: access specification applies to all following members up to the next access specification, if any.

There may be any number of public and private sections, and in any order. If no specifiers, all members are private.

Unlike Java, there are no class-level modifiers.

C++ places no requirements on source file names.

Note that a class definition must end with a semicolon.

Class definition in C++, continued

Just as in Java, a C++ class definition establishes the rules for creating and interacting with instances of a class.

Public and private specifications have the same meaning as in Java:

> Public members can be accessed by any code.
>
> Private members can only be accessed by code in member functions of the same class.

Source code that violates the rules established by a class definition generates a compilation error.

Working with objects in C++

In Java, objects are created with a new expression and reside in the heap. Variables of class type hold *references* to objects.

```
Counter c1 = new Counter("#1");  // Java

Counter c2;
c2 = new Counter("two");
```

In C++ an object can be created on the stack, in the heap, or in a global data area.

Consider the following C++ function:

```
void f()
{
   int i = 7;
   Counter c1("#1");
   ...
}
```

When f is called, two variables are created:

A variable named i of type int that is initialized with the value 7.

A variable named c1 of type Counter that is initialized with the value "#1".

Both i and c1 reside on the stack. After f returns, the memory provided for both i and c1 is available for reuse.

Working with objects in C++, continued

For variables of class type, member functions are invoked using the "." operator:

```
Counter c1("#1");
Counter c2("two");

c1.print();
c1.bump();
c2.bump();
c2.bump();
c1.print();
c2.print();
```

Output:

```
#1's count is 0
#1's count is 1
two's count is 2
```

Note that `c1` and `c2` are objects, not references to objects. Their address and size can be computed:

```
printf("&c1 = %X, &c2 = %X, sizeof(c1) = %d\n",
          &c1, &c2, sizeof(c1));
```

Output:

```
&c1 = 22FEB8, &c2 = 22FE98, sizeof(c1) = 8
```

Working with objects in C++, continued

In some cases one must reference an object using a pointer to it rather than the name of the object.

Imagine a routine that returns a pointer to a Counter:

```
Counter *findCounter(...);
```

Given that routine, one might write this:

```
Counter *cp = findCounter(...);
cp->bump();
```

Here is a routine that prints each Counter referenced in a zero-terminated array of Counter pointers:

```
void printAll(Counter *counters[ ])
{
    for (int i = 0; counters[i] != 0; i++) {
        Counter *cp = counters[i];
        cp->print();
        }
}
```

Usage:

```
Counter a('a'), b('b'), c('c');
Counter *cs[ ] = { &a, &b, &c, 0 };
printAll(cs);
```

Working with objects in C++, continued

Consider this routine:

```
Counter *makeLoadedCounter(string name, int count)
{
    Counter c(name);
    while (count--)
        c.bump();

    return &c;
}
```

and an invocation:

```
Counter *cp = makeLoadedCounter("loaded", 5);
cp->print();
```

Are there any problems with it?

Sidebar: class vs. struct

The C++ syntax for member function invocation is obviously an extension of the C syntax for structure references:

```
struct Point {
    int x, y;
    };

int main()
{
    struct Point pt;
    struct Point *p;

    pt.x = 30;
    pt.y = 40;

    p = &pt;

    printf("x = %d, y = %d\n", p->x, p->y);
}
```

Note that "class" is "syntactic sugar". The declaration

```
class X {
      ...declarations...
      };
```

is exactly equivalent to:

```
struct X {
   private:
       ...declarations...
      };
```

The special pointer variable this

Inside every member function C++ makes available a variable named 'this'. It contains the address of the object whose member function is being invoked. It is very similar to 'this' in Java.

Here is a new version of bump() for Counter:

```
void bump()
{
    printf("Bumping Counter at %X\n", this);
    itsCount++;
}
```

Usage:

```
Counter c("c");

printf("c is at %X\n", &c);
c.bump();
```

Output:

```
c is at 22FEB8
Bumping Counter at 22FEB8
```

this, continued

For member functions of a class X, the type of 'this' is "X *const". (The const specification prevents modifications to the value of this.)

If desired, we can reference members using this:

```
void bump()
{
    printf("Bumping Counter at %X\n", this);
    this->itsCount++;
}
```

Usage of this in C++ programs is usually for the same reasons as in Java, such as an object registering itself with an observer, or an object needing to identify itself in a data structure containing like objects.

Counter in C

To better understand how C++ works, it is useful to consider how a Counter "class" might be approached in C:

```
typedef struct {
    char itsName;  // 'char' to keep things simple in C
    int  itsCount;
    } Counter;

Counter_init(Counter *this, char name)
{
    this->itsCount = 0;
    this->itsName = name;
}

void Counter_bump(Counter *this)
{
    this->itsCount++;
}

void Counter_print(Counter *this)
{
    printf("%c's count is %d\n", this->itsName, this->itsCount);
}

int main()
{
    Counter a;
    Counter_init(&a, 'a');

    Counter_bump(&a);
    Counter_print(&a);
}
```

The machine code generated by a C++ compiler for Counter is very similar to the code a C compiler generates for the above.

Source file organization

Member function definitions do not need to appear in the class definition itself. One alternative is to place them in a separate source file.

One possible partitioning of code would produce this Counter.h:

```
#ifndef _Counter_h_
#define _Counter_h_       // Handle multiple inclusion
using namespace std;     // Use standard namespace
#include <string>         // Standard library string class header
class Counter
{
  private:
    int itsCount;
    string itsName;
  public:
    Counter(string);      // Note: no parameter name, only type
    void bump();
    int getCount();
    void print();
};
#endif
```

Unlike Java but just like C, C++ has a notion of a *translation unit*. A translation unit is a source file with #includes expanded and appropriate processing of directives like #ifdef.

A translation unit must include an appropriate declaration or definition of identifiers before code references them. For example, Counter.h needs to be #included in a source file before any members of Counter are referenced.

Source file organization, continued

The other piece of Counter is Counter.cc: (or .cpp, .cxx, .C, etc.)

```
#include <cstdio>
#include "Counter.h"

Counter::Counter(string name)
{
    itsCount = 0;
    itsName = name;
}

void Counter::bump() { itsCount++; }

int Counter::getCount() { return itsCount; }

void Counter::print()
{
    printf("%s's count is %d\n", itsName.c_str(), itsCount);
}
```

The scope resolution operator (::) is used to associate the functions with the Counter class.

Each function designated as a member of Counter must correspond to a declaration in Counter.h, which is included.

Member function definitions can be distributed across any number of source files. A missing definition manifests itself as an unresolved symbol when linking.

Note that C++ does not require "Counter" to appear in the name of the files that define the class.

Source file organization, continued

The third piece of the picture is code that makes use of Counter. Here's a test program, ctest.cc:

```
#include "Counter.h"

int main()
{
    Counter c1("#1");
    Counter c2("two");

    c1.print();
    c1.bump();
    c2.bump();
    c2.bump();
    c1.print();
    c2.print();
}
```

An executable is produced by compiling Counter.cc and ctest.cc, and linking them together. Here's one way:

```
% g++ ctest.cc Counter.cc
```

Here's another way:

```
% g++ -c Counter.cc
% g++ -c ctest.cc
% g++ -o ctest ctest.o Counter.o
```

The discussion of in-line functions will raise some additional issues with source organization.

Common compilation problems

Here are some common errors when coding in C++.

Missing semicolon at end of class declaration:

```
class X {
    int itsValue;
    }
```

This might generate an error about "multiple types in one declaration", or "too many types", "can't define type X here".

If the class declaration is the last thing in a header file, such problems turn up in the including file or the next included file.

Omission of scope resolution operator:

```
double getArea()  // Should be Rectangle::getArea()
{
    return itsWidth * itsHeight;
}
```

This might generate an error claiming that itsWidth and itsHeight are undeclared identifiers.

Common compilation problems, continued

Use of C++ keywords as identifiers:

```
if (typename == 0)
        ...
```

This might generate "parse error before '==' token" or "type expected".

Mismatching declaration of member function:

```
class X {
        ...
        int print();
        };

void X::print() ...
```

This might generate an error claiming that print is not a member of X.

Missing parentheses in member function invocation:

```
area = r.getArea;
```

This might produce an error about "member function must be called or address taken" or "argument of type 'int (Rectangle::)()' does not match 'int'.

More on Classes and Objects

More on constructors and destructors

Construction and global objects

Interesting uses for destructors

Dynamic memory management

Static members

In-line functions

Default arguments

Constructors

In C++, as in Java, constructors specify what data must be supplied to create a new instance of a class and how to initialize that new instance.

In Java, the predominant use of constructors is to initialize objects created with `new` expressions.

In C++, constructors are used in several contexts. One use of constructors is to support type extensibility—the ability to define new types that are as easy to use as built-in types such as `int` and `float`.

The compiler "knows" the definition "`int i = 7;`" indicates that:

(1) Memory to hold an integer should be set aside

(2) The memory should be initialized with the value 7

(3) The memory will be referred to as `i`

For a definition such as '`Counter c("x")`' the compiler knows to set aside memory to hold a `Counter` and that it will be referred to as `c`, but it doesn't know how to initialize `c` with the value `"x"`.

The constructor(s) for a class extend the compiler's repertoire by describing, in terms of C++ code, how to initialize a new instance of a type.

As in Java, member functions having the same name as the class are considered to be constructors.

Constructors, continued

Recall the constructor for Counter:

```
Counter::Counter(string name)
{
    itsCount = 0;
    itsName = name;
}
```

Just as in Java, constructors can be overloaded. Here's a second constructor; it provides for an initial count for a Counter:

```
Counter::Counter(string name, int count)
{
        itsCount = count;
        itsName = name;
}
```

With the second constructor in hand, the compiler is able to generate code for this definition:

```
Counter a("a",5), b("b",10), c("c");
```

Will the following line compile?

```
Counter a("a"), b("b",10), counters[10];
```

Default constructor

In Java, a *default constructor* is one that is supplied by the compiler.

In C++ a *default constructor* is a constructor that requires no arguments.

Here is a C++ class whose instances can be created with or without an integer initializer:

```
class X {
    public:
        X(int);      // Note: no parameter name – it's optional
        X();
        };
```

If an initializing value is specified, X(int) is called:

```
X a(1);
X pair[2] = { 7, 11 };
```

If no initializing value is specified, X() is called:

```
X a;
X xlist[10];
```

The compiler will generate a default constructor for a class iff no constructors have been specified for the class. Generated default constructors are public.

Will the following line of code compile?

```
X pair[2] = { 7 };
```

Details on constructors

In C++ as in Java...

Conceptually, every class has a constructor.

Conceptually, a constructor is always called whenever an object comes into existence. **Always**.

A constructor can do whatever it wants. It might initialize all, some or none of the data members. It might call other functions.

Constructors may be private.

A very important difference from Java:

Scalar data members are not zeroed as part of object creation—the value of uninitialized members is unpredictable. (Exception: memory for globals is zeroed.)

It is important to note that the definition

```
X a = 10;
```

is valid, but is NOT equivalent to

```
X a(10);  // "direct initialization"
```

The former causes a *copy constructor* to be invoked. Copy constructors are discussed later.

Details on constructors, continued

It is possible, and often convenient, to use temporary objects.

Examples:

```
int day = Date("7/4/04").day_of_week();  // not in std. library...

int span = Range(x, y, 'a').span();
```

"Temporary objects are destroyed as the last step in evaluating the full expression that (lexically) contains the point where they were created."—ISO C++ Standard

Destructors

The counterpart of a constructor is a *destructor*.

The destructor for a class X is a member function named ~X.

The destructor of a class is automatically called when the lifetime of an instance is over.

One situation in which objects are destroyed is when a block is exited: objects with local scope (automatic variables) are destroyed.

Example:

```
void f()
{
   Point p1(3,4);      // (A)

   ... computation ...

   if (...)
      return;

   Point p2(5,6);      // (B)

      ... more computation ...

}
```

p1 is created when execution reaches (A). p2 is created when/if execution reaches (B).

p1, and p2 if created, are destroyed when the routine returns.

Destructors, continued

The Java counterpart for a destructor is a *finalizer*, a method denoted by its name: finalize(). A finalizer is called when the memory of an object is about to be reclaimed by the garbage collector.

Java finalizers are often of little practical use because there is no guarantee that a finalizer will ever be called.

Destructors, continued

"Instrumenting" constructors and destructors with output expressions can aid understanding:

```
class X {
    public:
        X(char tag);
        ~X();
    private:
        char itsTag;
    };
X::X(char tag) { itsTag = tag; printf("X(%c)\n", itsTag); }

X::~X() { printf("~X(%c)\n", itsTag); }

int main()
{
    printf("...1...\n");
    X a('a');
    printf("...2...\n");
    X b('b');
    printf("...3...\n");
}
```

Output:

```
...1...
X(a)
...2...
X(b)
...3...
~X(b)
~X(a)     (Note LIFO ordering...)
```

Destructors, continued

The relationship between constructors and destructors is not exactly symmetrical—a constructor initializes an object but a destructor "salvages" still-useful resources when the object is destroyed.

Here's a start at a very simple string class. The constructor allocates memory and stores a copy of the string:

```
class String {
    public:
        String(char *s);
        ~String();
    private:
        char *itsPtr;
    };

String::String(char *s)
{
    itsPtr = (char*)malloc(strlen(s)+1);
    strcpy(itsPtr, s);
}
```

Usage:

```
String s("abc");
```

The destructor needs to free the allocated memory:

```
String::~String()
{
    free(itsPtr);
}
```

Construction and global objects

Constructors for global (file scope) objects in a file are guaranteed to be called before any routine in the file. Destructors for global objects are called when main() returns or when exit() is called.

Example:

```
X g1("g1");
X g2("g2");

int main()
{
        printf("main entered\n");

        X a("a");
        {
                X b("block 1");
                {
                        X b("block 2");
                }
        }

        X b("b");

        printf("exiting main\n");
}

X g3("g3");
```

Output with instrumented constructors and destructors:

```
X(g1), X(g2), X(g3), main entered, X(a), X(block 1)
X(block 2), ~X(block 2), ~X(block 1), X(b),
exiting main, ~X(b), ~X(a), ~X(g3), ~X(g2), ~X(g1)
```

Interesting uses for destructors

Problem: Imagining a graphical application, speculate on the purpose of the object hg in this sketch of code:

```
void render()
{
    Hourglass hg;

    ...a long and involved computation, but no use of 'hg'...
}
```

Dynamic memory management

In C, responsibility for providing explicit memory management is placed on the C library, which typically provides the functions malloc, free, and others.

C++ has language facilities for explicit memory management through the new and delete operators.

The new operator has several forms. This is the form that is most similar to Java:

```
new type ( initializing value(s) )
```

Example:

```
Counter *cp = new Counter("#1");  // C++
```

Three things happen:

(1) Sufficient memory to hold a Counter is allocated from the heap.

(2) The constructor Counter(string) is invoked, passing a char* for "#1". It initializes the data members.

(3) The memory address of the new Counter is the result of the new expression. The value is assigned to cp.

The analogous Java code is this:

```
Counter c = new Counter("#1");
```

Dynamic memory management, continued

If a class has a default constructor the initializing value(s) may be omitted:

```
X *p = new X;
```

It is possible to create an array of objects:

```
X *xs = new X[10];
```

In this case, the end result is that `xs` will hold the address of an array of ten initialized `X`s.

The type named in a `new` expression may be a scalar type. This expression allocates space for an array of 100 characters:

```
char *str = new char[100];
```

If desired, space can be allocated for a single scalar value. An initializer can be specified, too:

```
int *ip = new int;
double *dp = new double(12.34);
```

Dynamic memory management, continued

In general terms, here are the three commonly used forms of the new operator:

```
new T
new T ( initializers )
new T [ number-of-elements ]
```

In all cases the result type of a new expression is T*.

Will the following line compile?

```
X* p = new X*[10];
```

Dynamic memory management, continued

The counterpart of new is delete. Here is one of the two commonly used forms of the delete operator:

```
delete pointer-to-object
```

Example:

```
Counter *cp = new Counter("#1");

cp->bump();

cp->print();

delete cp;
```

If the object being deleted is of class type, the first action is to invoke its destructor. The next step is to deallocate the memory, making it available for subsequent allocation.

Dynamic memory management, continued

Here is the other common form of delete:

```
delete [ ] pointer-to-array of objects
```

This form should be used if the pointer references an array:

```
Counter *counters = new Counter[10];
char *p = new char[100];
...
delete [ ] counters;
delete [ ] p;
```

For an array of objects, such as counters above, the destructor is called for each of the objects before the memory is released.

The behavior of mixing an array allocation with a non-array delete is not defined by the standard. One common behavior is that if the array is of class type, only the first object in the array has its destructor called.

Question: Why does delete have differing forms for the two cases?

Dynamic memory management, continued

Problem: Write code that allocates an array of ten pointers to Counter and then populates the array with the addresses of ten new Counters, using the default constructor for each.

Problem: Write code that destroys the above-created Counters and appropriately deallocates memory.

Dynamic memory management, continued

new and delete may make use of malloc() and free() in the C library, but it is an error to mix and match them, calling free() with a value produced by new, for example.

It is permitted to call delete with the value zero:

```
    delete 0;   // No problem...
```

The new and delete operators can be overridden both globally and/or on a class by class basis.

In some cases it is useful to direct new to place an object at a particular location. The *placement syntax* accommodates that need, but is not discussed here.

Last but not least...

> *The absence of garbage collection in C++ raises the possibility of the same types of memory management errors that can occur when working in C.*

Static members

Just as Java, C++ provides a way to associate data and methods with a class itself rather than each instance of a class.

Here is a C++ class that maintains a count of the number of instances that exist:

```
// File: X.h
class X {
  public:
    X()     { theInstanceCount++; }
    ~X()   { theInstanceCount--; }

    static int getInstances() { return theInstanceCount; }

  private:
    static int theInstanceCount;
    };
```

Just as in Java, "static" is used to indicate that a data member or member function is associated with the class rather than an instance.

The scope resolution operator is used to associate a class with a static member:

```
int n = X::getInstances();
```

The Java equivalent:

```
int n = X.getInstances();
```

Static members, continued

Example:

```
#include "X.h"

int main()
{
  printf("[1]: %d Xs exist\n", X::getInstances());

  {
    X a, b, c;

    X *xs = new X[5];

    printf("[2]: %d Xs exist\n", X::getInstances());
  }

  printf("[3]: %d Xs exist\n", X::getInstances());
}
```

Output:

```
[1]: 0 Xs exist
[2]: 8 Xs exist
[3]: 5 Xs exist
```

Static members, continued

The preceding example hides a detail: Compiling and linking the main program shown produces an error saying that X::theInstanceCount is an unresolved symbol.

A static data member in C++ requires a definition for the data member that is external to the class definition.

In this case the solution is a third source file: X.cc.

```
//----- X.h -----
class X {
  public:
    X()     { theInstanceCount++; }
    ~X()    { theInstanceCount--; }
    static int getInstances() { return theInstanceCount; }
  private:
    static int theInstanceCount;  // declares theInstanceCount
    };

// ----- X.cc -----
int X::theInstanceCount = 0;     // defines theInstanceCount

// ----- xtest.cc -----
int main()
{
    int n = X::getInstances();
    ...
}
```

Note that the definition of theInstanceCount can't include static.

Static members, continued

In Java there is no notion of global functions but an equivalent effect is provided by static methods such as Math.sqrt().

In C++ most library functions with C equivalents are global. For example, there is a global sqrt() in <cmath>.

Just as in Java, class libraries often use static members to congregate data and functions. For example, imagine a Geometry class:

```
// --- Geometry.h ---
class Geometry {
  public:
        static double PI;
        static double GoldenRatio;
        static double Slope(Point p1, Point p2);
        static double SphericalVolume(double radius);
        ...
   private:
        Geometry();  // Can't make a Geometry...
        };

// --- Geometry.cc ---
double Geometry::PI = 3.141592653589793;
double Geometry::GoldenRatio = 1.618033988749895;

// ... method bodies ...
```

Usage:

```
area = Geometry::PI * radius * radius;
volume = Geometry::SphericalVolume(...);
```

In-line functions

For a given function it is possible to indicate that the function's code should be placed "in-line" rather than be called as a separate routine.

Given this declaration in a header file,

```
inline int abs(int i)
{
        if (i >= 0)
                return i;
        else
                return -i;
}
```

a use such as

```
int a = abs(b);
```

will cause code to be generated that performs the calculation "in-line"—no function call takes place.

The net result is as if this,

```
int a = (b >= 0) ? b : -b;
```

had been written instead.

In C++, in-lining is preferred over a preprocessor macro because inline functions have full function call semantics.

In-line functions, continued

Specifying a method body in a class definition implicitly indicates that the method is to be in-lined.

```
// Rectangle.h
class Rectangle {
    public:
        Rectangle(double width, double height);
        ...
        double getArea() {
            return itsWidth * itsHeight;
            }
    private:
        double itsWidth, itsHeight;
        };
```

getArea is implicitly declared as inline because its method body appears in the class definition.

Given Rectangle r(3,4), the statement

```
int a = r.getArea();
```

results in code generated as if this had been written instead:

```
int a = r.itsWidth * r.itsHeight;
```

In-line functions, continued

The inline keyword can be applied to member functions defined outside the class declaration:

```
// Rectangle.h
class Rectangle {
    public:
        Rectangle(double width, double height);
        ...
        double getArea();
    private:
        double itsWidth, itsHeight;
        };

inline double Rectangle::getArea()
{
    return itsWidth * itsHeight;
}
```

The result is completely equivalent to placing the function body in the class definition. This form is sometimes used to make a class definition easier to read.

Questions:

What happens if "inline" on the getArea() definition is omitted?

What happens if "Rectangle::" is omitted?

What happens if the above definition of getArea() is placed in Rectangle.cc instead of Rectangle.h?

In-line functions, continued

The benefit of in-lining:

> *In-line methods provide access that is just as fast as directly referencing the members, but without loss of encapsulation.*

Some things to note about in-lining:

Can lead to "code bloat"
Creates additional dependency on header files
Can complicate debugging
<u>A request to in-line a routine might not be honored</u>

Rule of thumb:

Keep inline functions trivial (e.g., "setters" and "getters") until performance requirements dictate a change.

Default arguments

Default arguments can provide a concise alternative to overloading.

Recall the example with two constructors for Counter:

```
class Counter
{
   Counter(string nm) { itsName = nm; itsCount = 0; }
   Counter(string nm, int ct) { itsName = nm;  itsCount = ct; }
   ...
};
```

Here's an alternative that uses a default argument:

```
Counter(string name, int count = 0)
{
   itsCount = count;
   itsName = name;
}
```

A declaration like this:

```
Counter c("loops");
```

is treated as if it were this:

```
Counter c("loops", 0);
```

Default arguments, continued

A further step is to supply a default for the name:

```
Counter(string nm = "<unknown>", int ct = 0)
{
    itsCount = ct;
    itsName = nm;
}
```

This single constructor allows a Counter to be created in three different ways:

```
Counter a, b("b"), c("c", 7);
```

Default arguments in C++ are often used in situations where Java constructors call "this(...)". For comparison, here's how the same problem might be approached in Java:

```
class Counter {
    public Counter()             { this ("<unknown>", 0); }
    public Counter(String nm) { this (nm, 0); }
    public Counter(String nm, int ct)
                                 { itsName = nm; itsCount = ct; }
    ...
    }
```

There is no C++ equivalent to calling this(...) in Java.

Default arguments, continued

Default arguments are not limited to constructors—they can be used in any function. Another example:

```
string TrimChars(string s, char what = ' ');

String s = "aaabbb   ";
s = TrimChars(s);       // now "aaabbb"
s = TrimChars(s, 'b');   // now "aaa"
```

A default value specification for an argument can appear only once in a translation unit. The usual practice is to specify default arguments in a header file:

```
// --- strutils.h ---
string TrimChars(string s, char what = ' ');

// --- strutils.cc ---
string TrimChars(string s, char what)
{
    ...processing...
}
```

The body of a function having default arguments often has no evidence of defaults being present.

Although literal values are most commonly specified for defaults, an arbitrary expression can be used. (Several rules apply, however.)

Miscellany

References

The const qualifier

NULL vs. 0

bool

The friend specifier

Copy constructors

References

The declaration `int x;` creates an integer object with the name `x`.

C++ provides a way to create a *reference* to an object, which is an alternative name, or *alias*, for the object.

Example:

```
int x = 1;

int& xref = x;

xref = 2;

printf("x = %d, xref = %d, &x = %X, &xref = %X\n",
    x, xref, &x, &xref);
```

Output:

```
x = 2, xref = 2, &x = EFFFFBE4, &xref = EFFFFBE4
```

References must always be initialized:

```
int& intref;  // Invalid -- no initialization!
```

References, continued

A reference may name an object with no prior name:

```
Rectangle *rp;

rp = FindRectangle();

Rectangle& r = *rp;

double a = r.getArea();
```

References cannot be changed. (And even if they could be changed, special syntax would be needed—think about it!)

References, continued

Consider a C routine to swap the value of two ints:

```
void swap(int *ap, int *bp)
{
        int tmp = *ap;
        *ap = *bp;
        *bp = tmp;
}
```

Its usage:

```
int i = 5, j = 10;
swap(&i, &j);    // sets i to 10, j to 5

int v[2] = { 3, 4 };
swap(&v[0], &v[1]);
```

Using references, swap can be implemented like this:

```
void swap(int& a, int& b)
{
        int tmp = a;
        a = b;
        b = tmp;
}
```

Its usage:

```
swap(i, j);
swap(v[0], v[1]);
```

The code generated for reference parameters is essentially the same as parameters that are pointers.

References, continued

The most common use of references in C++ is to reference instances of classes:

```
double maxArea(Rectangle& a, Rectangle& b)
{
    if (a.getArea() >= b.getArea())
        return a.getArea();
    else
        return b.getArea();
}
```

Usage:

```
Rectangle a(3,4), b(5,6);

int max = maxArea(a, b);
```

Same routine, but with pointers:

```
int maxArea(Rectangle* ap, Rectangle* bp)
{
    if (ap->getArea() >= bp->getArea())
        return ap->getArea();
    else
        return bp->getArea();
}
```

Usage:

```
Rectangle a(3,4), b(5,6);
int max = maxArea(&a, &b);
```

References, continued

Note that the potential of references means that you can't tell whether a function call might modify a scalar parameter.

Consider this call:

```
    int n = f(i);
```

Does `f()` change `i`?

Although references are used in a variety of ways in C++, the language feature that "closed the deal" to include references is operator overloading.

The const qualifier

const is a declaration of invariability.

const can be applied to simple variables:

```
const int couple = 2;  // integer constant

couple = 3;  // compilation error — couple can't be modified
```

The Java counterpart for const is final:

```
final int couple = 2;
```

const can be applied to the object referenced by a pointer:

```
const char *p;
   // p points to characters that are not to be modified

p = "abc";        // OK — modifies p
*p = '?';         // compilation error — uses p to change
                  // a character
```

const can be applied to a pointer:

```
char buf[ ] = "Testing";
char *const q = &buf[2];
   // q can't be changed; what q points to can

*q = 'x';         // OK — changes the 's' to an 'x'
q++;              // compilation error — would change q
```

The const qualifier, continued

const can be applied to both a pointer and what it references:

```
const char *const r = "abcd";
   // constant pointer to constant characters
```

Recall that in a member function for a class X the variable this has the type "X *const", i.e.:

```
X *const this;
```

A const static member may include an initialization. Example:

```
class Geometry {
  public:
      const static double PI = 3.141592653589793;
      const static double GoldenRatio = 1.618033988749895;
      static double Slope(Point p1, Point p2);
      static double SphericalVolume(double radius);
      //...
   private:
      Geometry();
      };
```

The const qualifier, continued

const can be applied to member functions. Imagine a class that represents a list of integers:

```
class IntList {
    public:
        IntList();
        void addValue(int value);
        int getLength() const;
        ...
    };
```

The const specification for the getLength() member function specifies that getLength will change no data members.

Having no const specification, addValue() is free to change data members.

const can be applied to parameters to indicate that the parameter should not be modified:

```
void f(const IntList& ilist)
{
    int len = ilist.getLength(); // OK

    ilist.addValue(7);  // compilation error — ilist is const
}
```

Note that inside a const method for class X, this is treated as

```
const X *const this;
```

What benefit is provided by const member functions?

The const qualifier, continued

Here's some code from a Java class:

```
//
// isDrainable determines whether water will fully drain from gs.
// NOTE: The GutterSystem is not modified!
//
boolean isDrainable(GutterSystem gs) { ...lots of code... }
```

Does isDrainable() cause any changes in the state of a GutterSystem?

Here's the signature of an equivalent method in C++:

```
//
// isDrainable determines whether water will fully drain from gs.
//
bool isDrainable(const GutterSystem& gs);
```

Does isDrainable() cause any changes in the state of a GutterSystem?

The combination of const member functions and const reference parameters provides two benefits:

The developer of a routine can be sure that the code is not inadvertently modifying a parameter that should not be changed.

The user of a routine can be sure that it won't modify a parameter.

The const qualifier, continued

Problem: Appropriately apply const to this Rectangle class:

```
//
// Rectangle.h
//
class Rectangle {
    public:
        Rectangle(double width, double height);
        double getArea();
        double getPerimeter();
        double getWidth();
        double getHeight();
        void     print();
    private:
        double itsWidth, itsHeight;
        };

//
// Rectangle.cc
//
double Rectangle::getWidth()
{
    return itsWidth;
}
double Rectangle::getArea()
{
    return itsWidth * itsHeight;
}
...and more...
```

Logical vs. physical const-ness

Consider this simple class and a function that uses it:

```
class X {
  public:
    X(int val) { itsValue = val; }
    int getValue() const { return itsValue; }
  private:
    int itsValue;
  };

int f(const X& x)
{
    int v = x.getValue();
    ...
}
```

Consider X augmented with an access count:

```
class X {
  public:
    X(int val) { itsValue = val; itsAccCnt = 0; }

    int getValue() const {
        itsAccCnt++;
        return itsValue;
        }
  private:
    int itsValue;
    int itsAccCnt;
    };
```

Does X still compile? If not, how can we fix it?

Logical vs. physical const-ness, continued

At hand:

```
int getValue() const {
    itsAccCnt++;
    return itsValue;
    }
```

The problem with getValue() is that it maintains logical constancy but not physical constancy.

The mutable type specifier designates that a data member is allowed to be changed in a const method.

Here's a solution for the getValue() problem:

```
class X {
    ...
  private:
    int itsValue;
    mutable int itsAccCnt;
    };
```

As a rule, mutable data members are used to hold data that has no direct external manifestation but that aids with things such as performance monitoring and caching.

NULL vs. 0

In C, NULL is a macro that expands to an implementation-defined null pointer constant, commonly ((void*)0), but is not guaranteed that NULL is numerically a zero.

In C, the recommended practice is to use NULL to represent a null pointer:

```
Node *next = NULL;
```

(C code that doesn't use NULL to represent a null pointer is considered to be non-portable.)

For C++, it is explicitly stated that the literal 0 (zero) can be used as a null pointer constant, as well as being an int literal.

A common practice in C++ is to use 0 to represent a null pointer, but NULL is OK, too:

```
Node *next = 0;          // Very common
Node *last = NULL;       // Also common (but be consistent!)
```

Visual C++ .NET defines NULL as 0.

g++ defines NULL as _ _null, a zero value but with pointer type, which causes a statement like this,

```
int i = NULL;
```

To generate a warning:

```
initialization to non-pointer type `int' from NULL
```

The bool type

The bool type in C++ is used to represent Boolean values.

There are two bool literals: true and false

In C, operators such as <, ==, &&, and ! yield an int result that is 0 or 1.

In C++ those same operators yield a bool result that is either true or false.

Any arithmetic (numeric) or pointer value can be implicitly converted to a bool value. A zero numeric value or a null pointer is converted to false. All other values are converted to true.

A bool value can be converted to an arithmetic type, producing either 0 or 1.

Problem: What is the value of j after the execution of this code?

```
int n = 10, m = 20;
bool a = n < m;
bool b = true;
int i = a < b;
bool c = 1.2 || false;
int j = !i + c;
```

The bool type, continued

The condition expressions for control structures (like if and while) and the operands of logical operators like == and ! are implicitly converted to type bool, producing an end result that is the same as C: A non-zero value indicates true and a zero indicates false.

Two examples:

```
//
// Print "Hello!" ten times
//
int i = 10;
while (i--)                              // Java: while (i-- != 0)
    puts("Hello!");

//
// Walk a linked list, printing the value in each node
//
for (node *p = first; p; p = p->next)
    printf("Value: %d\n", p->value);
```

The boolean and bool types in Java and C++, and their contexts of usage, are largely identical, essentially differing only by the automatic conversions in C++, but that difference has great effect.

The friend specifier

C++ has the concept of *friends* of a class. A friend is a function that is not a method of the class but is permitted access to the private members of the class.

Example:

```
class X {
  public:
        X(int val) { itsValue = val; itsAccCnt = 0; }

        int getValue() const {
                itsAccCnt++;
                return itsValue;
                }

  private:
        int itsValue;
        mutable int itsAccCnt;
  friend void Xamine(const X& theX);
        };

void Xamine(const X& theX)
{
        printf("The X at %x has an access count of %d\n",
                &theX, theX.itsAccCnt);
}
```

Being a friend of X, the function Xamine() can do its job, but there's no general exposure of the access count.

friend, continued

A class can name other classes as friends. Specific member functions of classes may be named as well.

```
class X {
    friend class Y;
    friend int Z::q(int);
    ...
    };
```

The first friend declaration makes all member functions of Y friends of X. Private data members and member functions of X can be accessed in any member function of Y.

The second friend declaration makes one member function of class Z a friend, too.

Some points about friendship in C++:

Friendship is granted, not taken.

A friend of a class should be thought of as part of the abstraction of that class.

"Without friends you expose too much". — Booch

Copy constructors

In certain situations in C programs, a value is initialized using an existing value of the same type. One situation is a variable definition with an initializer:

```
int i = 3;
int j = i + 10;
```

Both i and j have no previous value and are initialized with an int value.

Another situation arises in passing arguments to functions:

```
int add(int a, int b)
{
	return a + b;
}
```

Given a call such as add(i +2, j), the value of i + 2 is computed and used to initialize the parameter a. The value of j is used to initialize b.

A class may define a *copy constructor*, which describes how to initialize a new instance of the class with an existing instance of that class.

The copy constructor is another component of C++'s support for type extensibility.

Copy constructors, continued

Recall the data members of the simple rectangle class:

```
class Rectangle {
        ...
    private:
        double itsWidth, itsHeight;
  };
```

Imagine a routine that returns the larger of the areas of two rectangles:

```
double largerArea(Rectangle a, Rectangle b)
```

It might be used like this:

```
Rectangle r1(7,8) , r2(5,12);
double largest = largestArea(r1, r2);
```

The type of the parameters, simply Rectangle, indicate the arguments are to be passed by value. This is a case where a copy constructor is used: the parameters are initialized with values of the same type.

Copy constructors, continued

Because Rectangle does not define a copy constructor the compiler automatically generates one. Generated copy constructors use *memberwise copy* and are public.

Here's an approximation of the generated copy constructor:

```
Rectangle(const Rectangle& r)
{
    itsWidth = r.itsWidth;
    itsHeight = r.itsHeight;
}
```

The generated copy constructor works just fine. There's no reason to write one ourselves, except perhaps for debugging output.

In what situations will a generated copy constructor be inadequate?

Aggregations of Objects

Aggregation using pointers

Aggregation by value

Member initializers

Aggregation using references

Choosing representation of aggregation

Aggregations of objects

In Java there is only one way to represent an aggregation of objects: an aggregate holds references to elements.

In Java we might represent 2D points and lines like this:

```
class Point {
   public Point(int x, int y) { itsX = x; itsY = y; }
   private int itsX, itsY;
   }

class Line {
    public Line(Point A, Point B) { itsA = A; itsB = B; }
    private Point itsA, itsB;
    }
```

Usage:

```
Point origin = new Point(0,0);
Point p1 = new Point(3,4);

Line L1 = new Line(origin, p1);

Line L2 = new Line(new Point(7,11), new Point(5,10));
```

Aggregation is sometimes called the "has-a" relationship. For example, a line has two points.

Aggregation using pointers

In C++ there are three distinct ways to represent aggregation. One way is to use pointers. Here are Point and Line in C++:

```
class Point {
  public:
      Point(int x, int y) { itsX = x; itsY = y; }
  private:
      int itsX, itsY;
  };

class Line {
   public:
      Line(Point *p1, Point *p2) { itsP1 = p1; itsP2 = p2; }
   private:
      Point *itsP1, *itsP2;
   };
```

Usage:

```
Point origin(0,0);
Point p1(3,4);

Line L1(&origin, &p1);
```

How does the above code compare to the Java code?

How about the following alternative for L1?

```
Line *L1 = new Line(&origin, &p1);
```

Aggregation using pointers, continued

Recall the second Line created in Java:

```
Line L2 = new Line(new Point(7,11), new Point(5,10));
```

Is the following a suitable C++ analog?

```
Line L2(new Point(7,11), new Point(5,10));
```

Are there any problems with the following routine?

```
Line f()
{
    Point p1(1,1);
    Point p2(2,2);

    Line L(&p1, &p2);

    return L;
}
```

Aggregation by value

Another way to represent aggregation in C++ is to have objects physically contain other objects. Sometimes this is called *composition* or *composition by value*, or *containment by value*.

Example:

```
class Point {
    public:
        Point(int x, int y) { itsX = x; itsY = y; }
    private:
        int itsX, itsY;
    };

class Line {
    public:
        Line(Point p1, Point p2) { itsP1 = p1; itsP2 = p2; }
    private:
        Point itsP1, itsP2;
    };
```

Point physically contains two ints. Line physically contains two Points. If sizeof(int) is 4, then sizeof(Point) is 8 and sizeof(Line) is 16.

Usage:

```
Point origin(0,0);
Point p1(3,4);

Line L1(origin, p1);
Line L2(Point(7,11), Point(5,10));
```

Aggregation by value, continued

At hand:

```
class Point {
    public:
        Point(double x, double y) { itsX = x; itsY = y; }
    private:
        double itsX, itsY;
    };

class Line {
    public:
        Line(Point p1, Point p2) { itsP1 = p1; itsP2 = p2; }
    private:
        Point itsP1, itsP2;
    };
```

Only one problem: It doesn't compile. Here's what `g++` says:

```
agg2.cc: In constructor `Line::Line(Point, Point)':
agg2.cc:10: error: no matching function for call to
`Point::Point()'
agg2.cc:1: error: candidates are: Point::Point(const Point&)
agg2.cc:3: error:                 Point::Point(double, double)
agg2.cc:10: error: no matching function for call to
`Point::Point()'
agg2.cc:1: error: candidates are: Point::Point(const Point&)
agg2.cc:3: error:                 Point::Point(double, double)
```

Aggregation by value, continued

If an object contains other objects by value, the contained objects are constructed first and in the order they appear as data members in the containing object. Then, the constructor for the containing object is called. (A postorder tree traversal, in essence.)

```
class Milk        {  public: Milk() { puts("Milk"); }  };

class Bread       {  public: Bread() { puts("Bread"); } };

class Yolk        {  public: Yolk() { puts("Yolk"); }  };

class Egg {
   public:    Egg()  { puts("Egg"); }
   private:   Yolk  itsYolk;
   };

class CartonOfEggs {
   public:    CartonOfEggs() { puts("CartonOfEggs"); }
   private:   Egg  itsEggs[6];
   };

class Groceries {
   public:
      Groceries() { puts("Groceries"); }
   private:
      Milk            itsMilk;
      Bread           itsBread[2];
      CartonOfEggs    itsEggs;
      };
int main() {  Groceries g;  }
```

Output: (compressed)

```
Milk, Bread, Bread, Yolk, Egg, Yolk, Egg, Yolk, Egg, Yolk, Egg,
Yolk, Egg, Yolk, Egg, CartonOfEggs, Groceries
```

Member initializers

The correct way to write the constructor for Line is to use *member initializers*.

Instead of this:

```
Line(Point p1, Point p2) { itsP1 = p1; itsP2 = p2; }
```

Use this:

```
Line(Point p1, Point p2) : itsP1(p1), itsP2(p2) { }
```

Member initializers provide a way to associate initializing values with members. In this case the copy constructor for Point is used to initialize itsP1 and itsP2 using the values of p1 and p2.

A rule:

> *If an instance of class* X *is a data member then either (1) the data member must have a member initializer, or (2)* X *must have a default constructor.*

The constructor for Line has an empty body, but it need not.

The need for member initializers is due to the strong distinction between initialization and assignment in C++.

Member initializers, continued

Here is another constructor for Line:

```
Line(int x1, int y1, int x2, int y2)
 : itsP1(x1, y1), itsP2(x2, y2) { }
```

This declares that the members itsP1 and itsP2 should be initialized with the values x1, y1 and x2, y2, respectively.

Member initializers can used with scalar data members, too:

```
Point(int x, int y)  : itsX(x), itsY(y) { }
```

The expressions used for member initialization may be of arbitrary complexity.

A member initializer is the only way to initialize a non-static const data member:

```
class X {
   public:
      X(int N) { itsN = N; q = new int[N]; }       // Compilation error
      X(int N) : itsN(N), q(new int[N])  { }        // OK
   private:
      const  int  itsN;
      int  *const  q;
   };
```

Aggregation using references

The third way to represent aggregation in C++ is to use references.

Example:

```
class CounterPair {
  public:
    CounterPair(Counter& c1, Counter& c2)
    : itsA(c1), itsB(c2) { }

    void bump() { itsA.bump();  itsB.bump(); }

    void print(char *label)
    {
        printf("%s", label);
        itsA.print();
        itsB.print();
    }

  private:
    Counter& itsA;  // Alternative: Counter &itsA, &itsB;
    Counter& itsB;
  };
```

Using references to represent aggregation implies that the objects always exist and don't vary (i.e., are not swapped in and out).

Note that member initializers are required for `itsA` and `itsB`.

Internally, a data member of reference type is represented with a pointer.

Aggregation using references, continued

Usage:

```
Counter a("a"), b("b");

CounterPair p1(a, b);
p1.bump();
p1.print("p1:\n");

Counter *cp = new Counter("c");

CounterPair p2(b, *cp);
p2.bump();

p2.print("p2:\n");
```

Output:

```
p1:
a's count is 1
b's count is 1
p2:
b's count is 2
c's count is 1
```

Choosing representation of aggregation

C++ provides three ways to represent aggregation:

An object can physically contain other objects
(Aggregation by value)

An object can hold pointers to other objects
(Aggregation with pointers)

An object can hold C++ references to other objects
(Aggregation with references)

A single class might use all three.

Here are some guidelines for selection of a representation:

Objects that have no existence beyond that of the aggregate suggest aggregation by value.

Aggregation by value creates a header file dependency. For example, `Line.h` needs to include `Point.h` if composition is used but if not, a forward declaration (`class Point;`) would suffice.

Independent lifetimes suggests use of pointers or references.

A varying number of contained objects suggests use of pointers.

An object present in more than one aggregation requires representation using a pointer or a reference.

A choice in aggregation

Problem: Ignoring the lack of a constructor that takes two points, comment on the merit of this implementation of Line:

```
class Line {
  public:
    Line(double x1, double y1,
           double x2, double y2)
    : itsP1(new Point(x1, y1)), itsP2(new Point(x2, y2)) { }

    ~Line() { delete itsP1;  delete itsP2; }

  private:
    Point *itsP1, *itsP2;
    };
```

Type Extensibility and Operator Overloading

Overload resolution

Operator overloading basics

Operators as member functions

Choice in overloading

Overloading assignment

A simple string class

Default arguments

Operator overloading basics

Conversion operators

Review of constructors, destructors, and assignment

Overload resolution

C++ allows both functions and operators to be *overloaded*.

A simple example of function overloading:

```
int max(int a, int b)
{
    return (a > b ? a : b);
}

double max(double a, double b)
{
    return (a > b ? a : b);
}
```

Selection between overloaded functions is based on which function best matches the supplied arguments.

```
max(1, 2);              // calls max(int, int)
max(3.4, 3.5);          // calls max(double, double)
```

In both cases there is an exact match between the supplied arguments and a version of min.

The process of determining which overloaded function should be selected is called *overload resolution*.

Overload resolution, continued

At hand:

```
int max(int a, int b);

double max(double a, double b);
```

Here's a call that doesn't exactly match either function:

```
max('a', 'b');
```

C++ will apply conversions to match a call with a function. In this case the standard conversion of integral promotion is applied to convert the two char values into two int values, and then match the max(int, int) form.

Here's a call that is said to be *ambiguous*; it will not compile:

```
max(3.4, 4);
```

One way to produce a match would be to convert 4 to a double. Another way to produce a match would be to convert 3.4 to an int. *C++ considers those two conversions to be of equivalent merit and will not choose between them.*

We can eliminate the ambiguity with either of two casts:

```
max(3.4, (double)4);      //  calls max(double, double)
max((int)3.4, 4);         //  calls max(int, int)
```

Overload resolution, continued

In addition to standard conversions, C++ will also apply one user-defined conversion to match a call to a function. Construction is one example of a user-defined conversion.

Consider this class:

```
class X {
    public:
        X(double);
    };
```

In addition to telling the compiler what's required to make an X and how to do it, the class defines this conversion:

If you have a double *and need an* X*, call this constructor.*

Here's a function that requires an X as its argument:

```
void f(X x) { }
```

All of these calls are valid:

```
f(1);               // Converts int to double, calls f(X(double))
f('a');             // Promotes char to int, converts int to double,
                    //     calls f(X(double))
f(1.2);             // Calls f(X(double))

X x1(2.0);
f(x1);              // Exact match – no conversion
```

Overload resolution, continued

At hand:

```
class X {
    public:
        X(double);
    };

void f(X x) { }
```

Let's add another class and also overload f:

```
class Y {
    public:
        Y(double);
    };

void f(Y y) { }
```

The call f(1) is now ambiguous. The compiler can't choose between

Convert int to double, call f(X(double))

and

Convert int to double, call f(Y(double))

Is f(1.2) ambiguous?

Overload resolution, continued

In some cases, treating a constructor as a user-defined conversion creates headaches.

Adding the `explicit` specifier to a constructor indicates that only explicit calls of the constructor are permitted; such a constructor is not considered to specify a user-defined conversion.

Example:

```
class X {
    public:
        X(double);
    };

class Y {
    public:
        explicit Y(double);
    };

void f(X x) { }
void f(Y y) { }
void g(Y y) { }
```

Calls:

```
f(1);           // Unambiguous
f('a');         // Ambiguous or OK?
f(1.2);         // Ambiguous or OK?

g(1.0);         // Ambiguous or OK?
g(Y(1.0));      // Ambiguous or OK?
```

Overload resolution, continued

C++ will not consider a series of conversions that requires more than one user-defined conversion.

Two trivial classes and two functions:

```
class A {
    public:
        A(int);
    };

class B {
    public:
        B(A);
    };

void f(A) { }
void g(B) { }
```

The two classes define two user-defined conversions:

An `A` can be made from an `int`

A `B` can be made from an `A`

Which of following calls are valid? Why or why not?

```
f(1);
f('a');
g(1);
g('a');
g(A(1));
g(A('a'));
```

Operator overloading

It is possible to overload the operators of the C++ language so that they have meaning for user-defined types.

A type to represent complex numbers:

```
    Complex a(1,0), b(2,-3), p, q;

    p = a + b;
    q = (a + b) / (-p * 5);
```

A type to represent character strings:

```
    String first = "John", last = "Smith";

    String name = first + " " + last; // produces "John Smith"
```

Types for times and durations:

```
    Time FirstArrival("12/31/2002 18:00");
    Time LastDeparture("1/1/2003 04:27");

    Duration PartyLength =   LastDeparture - FirstArrival;
        // Represents 10 hours, 27 minutes
```

Operator overloading is another aspect of C++'s support for type extensibility.

Ground rules for operator overloading

By convention, operators have an expected interpretation, but that is left to the discretion of the programmer. A class designer homesick for Icon might do this:

```
int n = *segmentList;  // produces the number of segments
```

Operator overloading in C++ is not as flexible as in some languages:

No new operators can be defined. For example, you can't define an operator /\ to represent a logical conjunction, such as P /\ Q.

Operator/operand type combinations that already have a meaning can't be redefined. For example, the meaning of i + j, where i and j are ints, can't be changed. ("C++ should be extensible, but not mutable."—Stroustrup)

The precedence and "arity" of operators cannot be changed. Two examples:

^ can be overloaded to mean exponentiation but x*y^z would mean (x*y)^z, not x*(y^z).

A unary | operator can't be defined.

Operator overloading basics

Here is an ordinary function that "sums" two Rectangles by adding their widths and heights:

```
Rectangle Sum(Rectangle a, Rectangle b)
{
        double new_w = a.getWidth() + b.getWidth();
        double new_h = a.getHeight() + b.getHeight();

        Rectangle newRect(new_w, new_h);

        return newRect;
}
```

It might be used like this:

```
Rectangle x(3, 4);
Rectangle y(5,10);

Rectangle z = Sum(x, y);
z.print('z');  // 'z' labels output

x = Sum( Sum(x,y), z);
x.print('x');
```

To produce this output:

```
Rectangle 'z': 8 x 14
Rectangle 'x': 16 x 28
```

Operator overloading basics, continued

If instead of a functional form, an operator form is desired for producing the sum of two rectangles, we can say:

```
Rectangle operator+(Rectangle a, Rectangle b)
{
        double new_w = a.getWidth() + b.getWidth();
        double new_h = a.getHeight() + b.getHeight();

        Rectangle newRect(new_w, new_h);

        return newRect;
}
```

This declares (to the compiler):

> *If two* Rectangle*-valued expressions are the operands of* +*, call this routine and for a result, use the value it returns.*

Rectangles can now be "added" using operator syntax:

```
Rectangle x(3,4);
Rectangle y(5,10);

Rectangle z = x + y;

x = x + y + z;
```

Note that providing an overloaded definition for + does not imply a definition for +=.

Operator overloading basics, continued

For reference:

```
Rectangle operator+(Rectangle a, Rectangle b) ...
```

Passing a and b by value is inefficient. It is better to pass const references. There are no changes aside from the parameter list:

```
Rectangle operator+(const Rectangle& a, const Rectangle& b)
{
        double new_w = a.getWidth() + b.getWidth();
        double new_h = a.getHeight() + b.getHeight();

        Rectangle newRect(new_w, new_h);

        return newRect;
}
```

More Rectangle operators

```
// Compare two rectangles
//
bool operator==(const Rectangle& a, const Rectangle& b)
{  return  a.getWidth() == b.getWidth()
        && a.getHeight() == b.getHeight();  }

// Scale a rectangle by a factor n:
//
Rectangle operator*(const Rectangle& a, double n)
{  return Rectangle(a.getWidth() * n, a.getHeight() * n);  }

// "Rotate" a rectangle 90 degrees
//
Rectangle operator-(const Rectangle& a)
{   return Rectangle(a.getHeight(), a.getWidth());  }
```

Usage:

```
Rectangle a(3,4), b(1,2);

Rectangle c = b * 3;
c.print('c');

Rectangle d = -c;
d.print('d');

if (-(Rectangle(2,3) * 3) == d + Rectangle(3,3))
        printf("Works!\n");
```

Output:

```
Rectangle 'c': 3 x 6
Rectangle 'd': 6 x 3
Works!
```

Operators as member functions

The preceding slides show overloaded operators implemented as free-standing functions—they are in no way part of the Rectangle class.

Given the preceding definition for operator+, if a and b are Rectangles, the expression a+b is treated as this:

```
operator+(a,b)
```

Alternatively, operators may be defined as member functions. In such a case, a+b would be treated as this:

```
a.operator+(b)
```

Operators as member functions

At hand: If operator+ is a member function, then

```
a+b
```

is treated as:

```
a.operator+(b)
```

Here is operator+ defined as a member function of Rectangle:

```
class Rectangle {
  public:
        Rectangle operator+(const Rectangle& rhs) const {
                double new_w = itsWidth + rhs.itsWidth;
                double new_h = itsHeight + rhs.itsHeight;

                Rectangle newRect = Rectangle(new_w, new_h);
                return newRect;
                }
       ...
        };
```

A member function for an N-ary operator has N-1 parameters.

Question: This implementation of operator+ uses itsWidth and itsHeight while the free-standing function uses getWidth() and getHeight(). Why?

Operators as member functions, continued

Here are more Rectangle operators in the form of member functions:

```
class Rectangle {
  public:
        ...
        Rectangle operator*(double rhs) const;
        Rectangle operator-() const;
        bool operator==(const Rectangle& rhs) const;
        bool operator!=(const Rectangle& rhs) const;
        };

Rectangle Rectangle::operator*(double rhs) const
{
    return Rectangle(itsWidth * rhs, itsHeight * rhs);
}

Rectangle Rectangle::operator-() const
{
    return Rectangle(itsHeight, itsWidth);
}
```

Operators as member functions, continued

For comparison, here are statements in infix form and their interpretation both with operators as free-standing functions and with operators as member functions.

```
Rectangle x = a + b + c;

    Rectangle x = operator+(operator+(a, b), c);

    Rectangle x = a.operator+(b).operator+(c);


Rectangle e = -d * 3;

    Rectangle e = operator*(operator-(d),3);

    Rectangle e = d.operator-().operator*(3);
```

Choice in overloading

A trivial wrapper class for ints:

```
class Num {
    public:
        Num(int i) : value(i) { }
        int getValue() const { return value; }
    private:
        int value;
    };
```

Addition is overloaded via a free standing function:

```
Num operator+(const Num& lhs, const Num& rhs)
{
    return Num(lhs.getValue() + rhs.getValue());
}
```

These statements compile:

```
Num a(5);
Num b(7);

Num c = a + b;
Num d = c + 2;
Num e = 5 + d;
```

The first addition is matched directly by operator+.

Why do the second and third additions work?

Choice in overloading, continued

Let's add subtraction via a member function:

```
class Num {
    public:
        Num(int i) : value(i) { }
        int getValue() const { return value; }
        Num operator-(const Num& rhs) const {
            return Num(getValue() - rhs.getValue());
            }
    private:
        int value;
    };

Num operator+(const Num& lhs, const Num& rhs)
{
    return Num(lhs.getValue() + rhs.getValue());
}
```

It almost works:

```
Num a(5);
Num b(7);

Num c = a + b;
Num d = c + 2;
Num e = 5 + d;
Num f = a - b;
Num g = f - 2;
Num h = 5 - g;        // Error: no match for 'operator-' in '5 - g'
```

What's the problem?

Choice in overloading, continued

At hand:

```
class Num {
    public:
        Num operator-(const Num& rhs) const {
            return Num(getValue() - rhs.getValue());
            }
        ...
    };

Num operator+(const Num& lhs, const Num& rhs)
{
    return Num(lhs.getValue() + rhs.getValue());
}
```

Usage:

```
Num e = 5 + d;        // OK
Num h = 5 - g;        // Error
```

Addition works because the conversion `Num(int)` can be applied to 5 and then the call matches `operator+(Num, Num)`.

C++ simply does not consider treating `5-g` as `Num(5).operator-(g)`.

As a rule of thumb, overload binary operators with free-standing functions to avoid asymmetries.

Question: Operators that are member functions can access private data. How can that same access be provided to operators that are free-standing functions?

Choice in overloading, continued

At hand:

```
class Num {
    public:
        Num(int i) : value(i) { }
        int getValue() const { return value; }
        ...
    private:
        int value;
    };

Num operator+(const Num& lhs, const Num& rhs)
{
    return Num(lhs.getValue() + rhs.getValue());
}
```

The friend specifier can be used to allow a free-standing function operator+ to access private data:

```
class Num {
    public:
        Num(int i) : value(i) { }
        int getValue() const { return value; }
    friend Num operator+(const Num&, const Num&);
    private:
        int value;
    };

Num operator+(const Num& lhs, const Num& rhs)
{
    return Num(lhs.value + rhs.value);
}
```

Overloading assignment

By default, if one instance of a class X is assigned to another, *memberwise assignment* is performed.

For example, the result of the assignment r2 = r1 in:

```
Rectangle r1(3,4), r2(1,2);
r2 = r1;
```

is as if these two statements had been executed:

```
r2.itsWidth = r1.itsWidth;
r2.itsHeight = r1.itsHeight;
```

If an object contains others objects, memberwise assignment is recursively applied. For example, if L1 and L2 are Lines, then L2 = L1 causes

```
L2.itsP1 = L1.itsP1;
```

which in turn causes

```
L2.itsP1.itsX = L1.itsP1.itsX;
L2.itsP1.itsY = L1.itsP1.itsY;
```

Resuming at the level of L2 = L1,

```
L2.itsP2 = L1.itsP2;
```

in turn causes

```
L2.itsP2.itsX = L2.itsP2.itsX;
L2.itsP2.itsY = L2.itsP2.itsY;
```

Overloading assignment, continued

Memberwise assignment happens to be satisfactory for Rectangle, but we can provide an overloaded assignment operator:

```
void Rectangle::operator=(const Rectangle& rhs)
{
        itsWidth = rhs.itsWidth;
        itsHeight = rhs.itsHeight;
}
```

The language definition requires assignment to be implemented as a member function; it cannot be implemented as a free-standing function.

Overloading assignment, continued

The current state of Rectangle:

```
class Rectangle {
  public:
      Rectangle(double w, double h);

      void operator=(const Rectangle& rhs) {
          itsWidth = rhs.itsWidth;
          itsHeight = rhs.itsHeight;
          }
      ...
  private:
      double itsWidth;
      double itsHeight;
      };
```

Unfortunately, if r1, r2, and r3 are instances of Rectangle, our current implementation doesn't allow this:

```
r1 = r2 = r3;
```

(With functional syntax:)

```
r1.operator= ( r2.operator= ( r3 ) ) ;
```

Why doesn't it?

Overloading assignment, continued

At hand: The statement

```
r1 = r2 = r3;
```

does not compile

Solution:

```
Rectangle& operator=(const Rectangle& rhs)
{
    itsWidth = rhs.itsWidth;
    itsHeight = rhs.itsHeight;

    return *this;
}
```

The above routine is essentially what's generated if no assignment operator is specified in the definition of Rectangle.

Overloading assignment, continued

Suppose it is decided that assigning a double value to a Rectangle is meaningful, and that it means the width and height of the Rectangle should be set to the given value.

Usage:

```
    Rectangle r(3,4);

    r = 10;
    r.print('r');
```

Output:

```
    Rectangle 'r': 10 x 10
```

Implementation:

```
    Rectangle& Rectangle::operator=(double side)
    {
        itsWidth = itsHeight = side;

        return *this;
    }
```

A simple string class

It is rare to a find a program that doesn't make some use of character strings.

C has no string data type but it has a very strong convention: Strings are represented by a null-terminated array of characters.

String handling in C is tedious and error prone, and can lead to shortcuts such as hoping the result of a concatenation will not overrun a fixed length buffer.

Many languages have a built-in string data type that allows strings to be manipulated in a very natural fashion.

Java has a built-in string type but it is no gem: It is immutable and the only operator available is concatenation.

The C++ language itself has no string data type, but the standard library includes a `string` class. It is mutable, and supports a reasonable set of operators.

`string` is built using the type extensibility mechanisms of C++.

Building a simple string type from scratch is a good exercise in type extensibility. Our string type will be called `String`.

String

String has one data member, a pointer to a null-terminated array of characters in allocated memory (a C-style string):

```
class String {
    private:
        char *itsPtr;
    };
```

What constructors are needed to support the following definitions?

```
String s1("This is s1")
String s2;
String s3('x');
String names[10];
```

Does String need a destructor?

String, continued

Here are constructors, a destructor, and a "dump" method:

```
class String {
    public:
        String(const char *s) {
            itsPtr = new char[strlen(s) + 1];
            strcpy(itsPtr, s);
            }

        String() { itsPtr = new char[1]; itsPtr[0] = '\0';  }

        ~String() { delete [ ] itsPtr; }

        void dump(char *label)
        { printf("%s: '%s' (at %x)\n", label, itsPtr, itsPtr); }

    private:
        char *itsPtr;
    };
```

Usage:

```
String s1("This is s1"), s2;

s1.dump("s1");
s2.dump("s2");
```

Output:

```
s1: 'This is s1' (at a0416a8)
s2: '' (at a0416b8)
```

String, continued

No assignment operator is defined for String and therefore, assigning one String to another is done with memberwise assignment.

Unfortunately, the result of memberwise assignment is a shared pointer:

```
String s1("This is s1"), s2;

s1.dump("s1");
s2.dump("s2");

s2 = s1;    // Result: s2.itsPtr = s1.itsPtr;  (bad!!!)
puts("---");

s2.dump("s2 after 's2 = s1'");
```

Output:

```
s1 = 'This is s1' (at a0416a8), s2 = '' (at a0416b8)
---
s2 = 'This is s1' (at a0416a8)
```

Note that s1 and s2 reference the same piece of memory. A change in the contents of one will be reflected in the other.

That behavior is not what's typically desired—strings should have *value semantics*, like scalar types, not *reference semantics*.

String, continued

Here is a method that simply overwrites a string with a given character:

```
void String::overwrite(char c) {
    for (char *p = itsPtr; *p; p++)
        *p = c;
    }
```

We can use it to demonstrate the inappropriate sharing of data that results from assignment using the generated assignment operator:

```
String s1("This is s1"), s2;

s2 = s1;

s2.overwrite('X');

s1.dump("s1 after overwrite");
s2.dump("s2 after overwrite");
```

Output:

```
s1 after overwrite: 'XXXXXXXXXX' (at a041b70)
s2 after overwrite: 'XXXXXXXXXX' (at a041b70)
```

Are there other ill-effects in addition to this unwanted sharing?

String, continued

Memberwise assignment is obviously not suitable for String—an assignment operator must be written. A first cut:

```
String& String::operator=(const String& rhs)
{
        delete [ ] itsPtr;

        itsPtr = new char[strlen(rhs.itsPtr) + 1];
        strcpy(itsPtr, rhs.itsPtr);

        return *this;
}
```

Recall that `s1 = s2` is equivalent to `s1.operator=(s2)`.

Are the any problems with the following code?

```
String s("abc");
s = s;
```

String, continued

At hand—self assignment:

```
String s("abc");
s = s;
```

Solution:

```
String& String::operator=(const String& rhs)
{
        if (this != &rhs) { // know myself
                delete [ ] itsPtr;

                itsPtr = new char[strlen(rhs.itsPtr)+1];
                strcpy(itsPtr, rhs.itsPtr);
                }
        return *this;
}
```

Thus, self-assignment is a "no-op".

String, continued

Remember that there is a distinction between initialization and assignment.

This is *initialization* of two Strings, s1 and s2:

```
String s1("This is s1");
String s2(s1);
```

s1 is initialized with the character string "This is s1", of type char *. The initialization is handled by String(const char *).

In the second case, s2 is initialized with s1. What constructor handles the initialization?

A copy constructor for String

To initialize an object of type X with another object of type X, C++ uses the *copy constructor* of X.

If no copy constructor is declared, one is generated by the compiler. The generated copy constructor for String is equivalent to this:

```
    String(const String& s) : itsPtr(s.itsPtr) { }
```

The result is two Strings with the same itsPtr value.

This is a suitable copy constructor for String:

```
    String::String(const String& s)
    {
        itsPtr = new char[strlen(s.itsPtr) + 1];
        strcpy(itsPtr, s.itsPtr);
    }
```

Contrast with assignment:

```
    String& String::operator=(const String& rhs)
    {
        if (this != &rhs) { // know myself
            delete [ ] itsPtr;
            itsPtr = new char[strlen(rhs.itsPtr)+1];
            strcpy(itsPtr, rhs.itsPtr);
            }
        return *this;
    }
```

Why doesn't the copy constructor need to free memory, too?

A copy constructor for String, continued

The copy constructor is also used when passing an object to a function by value, and when returning an object by value.

Example:

```
String trivial(String a)
{
   String b("xyz");

   return b;
}

void f()
{
   String s1("abc");

   String s2 = trivial(s1);
}
```

More on copy constructors

A copy constructor for a class X usually has this form:

```
X(const X&)
```

What's wrong with the following?

```
class X {
    public:
        X(X value);
        ...
    };
```

If it makes no sense to copy an object, or you don't want to worry about it (yet) for a class, declare a copy constructor and make it private.

A detail on initialization

A String can be initialized like this:

```
String s = "testing";
```

It seems reasonable for that to be equivalent to this,

```
String s("testing");
```

but it is not guaranteed.

The first form may generate two constructor calls, equivalent to this:

```
String T("testing");
String s(T);
```

The form String s("testing"), with parentheses, is called *direct initialization*.

Note that if String(const char*) is made explicit,

```
explicit String(const char *s) { ... }
```

then String s = "abc"; won't compile but String s("abc"); will.

Scalars may be initialized using the direct initialization form:

```
int i(7);          // equivalent to  int i = 7;
char c('x');       // equivalent to  char c = 'x';
int i = int();     // equivalent to i = 0; (what does 'int i();' mean?)
```

String: concatenation

Consider overloading '+' to allow concatenation of two strings:

```
String a = "xyz";
String b = "pdq";
String c = a + b; // c is "xyzpdq"

String c = a.operator+(b); // equivalent, as a member function
```

Implementation:

```
class String {
  public:
        ...
        String operator+(const String& rhs);
        ...
        };

String String::operator+(const String& rhs)
{
        int len = strlen(itsPtr) + strlen(rhs.itsPtr);

        char *p = new char[len + 1];
        strcpy(p, itsPtr);
        strcat(p, rhs.itsPtr);

        String r(p);  // Invokes String(const char *)
        delete [ ] p;
        return r;
}
```

String: subscripting

Consider overloading '[ ]' to provide array-like access to individual characters:

```
String s("aeiou");

char c = s[0];   // c is 'a'

s[2] = 'I';      // changes s to "aeIou"
```

Another example of usage—print the index, address, and value of each character in a String:

```
String s = "smudge";
s.dump("s");

for (int i = 0; i < s.getLength(); i++) {
    char *p = &s[i];
    printf("s[%d] at %x is '%c'\n",
              i,     p,     *p);
    }
```

Output:

```
s: 'smudge' (at a041cf0)
s[0] at a041cf0 is 's'
s[1] at a041cf1 is 'm'
s[2] at a041cf2 is 'u'
s[3] at a041cf3 is 'd'
s[4] at a041cf4 is 'g'
s[5] at a041cf5 is 'e'
```

String: subscripting, continued

Implementation:

```
char& String::operator[ ](int pos)
{
        assert(pos >= 0 && pos < strlen(itsPtr));

        return itsPtr[pos];
}
```

Note that because a reference is returned it is possible to change contained characters and/or obtain their address.

It would be trivial to add meaning for negative subscripts.

The assert macro terminates execution if the subscript is out of range. A better choice would be to throw an exception.

This simple String class provides value semantics, worry-free concatenation and subscripting, but also provides C-like semantics with the ability to get the address of an individual character.

Conversion operators

A *conversion operator* is a specialized member function that defines how an object can create an instance of another type that is a representation of itself.

For example, here is a conversion operator for Rectangle:

```
class Rectangle {
  public:
        ...
        operator double() { return getArea(); }
  private:
        ...
        };
```

This declares (to the compiler):

> *If you have a* Rectangle *and need a* double, *call this function and use the value it returns.*

Note the general form:

```
operator type-name() { ... }
```

Example:

```
Rectangle r(3,4);
double a = r;                          // assigns 12 to a

double b = Rectangle(5,6) / 3;         // assigns 10 to b
```

Conversion operators are another type of user-defined conversion.

Conversion operators, continued

Another example:

```
class String {
        public:
                String(char *s);
                ~String() { delete [ ] itsPtr; }
                operator const char *() const { return itsPtr; }
                operator char *() const {
                        char *p = new char[strlen(itsPtr) + 1];
                        strcpy(p, itsPtr);
                        return p;
                        }
        private:
                char *itsPtr;
        };
```

Usage:

```
String s("testing");

const char *p1 = s;  // refs same data as in s

char *p2 = s;           // refs allocated data,
*p2 = 'x';              //  must be freed
...
delete [ ] p2;
```

A class may have any number of conversion operators.

It is easy to get carried away with conversion operators—use them with caution.

Review—ctors, dtors, and assignment

Whenever an instance of a class is created, an appropriate constructor is called. The task of a constructor is to appropriately initialize a block of memory that is to represent an object.

The implementor of a class defines what constructors do; the compiler determines when constructors are called; the run-time system determines where objects reside in memory.

Distinguished types of constructors:

Default constructor: A constructor that requires no parameters.

Used to initialize an object if no initializing values are supplied.

Examples:

```
X x1, x2;

X *xp = new X;

X xs[10];

class Y { X itsX; };
```

If a class has no constructors, a public default constructor is supplied.

Review—ctors, dtors, and assignment, cont.

Copy constructor:

A constructor of the form `X(X&)` or `X(const X&)`.

Used to initialize a new instance of a class given an existing instance.

A copy constructor using memberwise copy is generated if no copy constructor is specified.

Examples:

```
X x3 = x2;
X x4(x3);

X *xp = new X(x3);

void f(X a, X b);
f(x3, *xp);
```

Ordinary constructor:

Neither a default or copy constructor.

Selected based on types of initializing values.

Examples:

```
X x5(1);
X x6("abc", 'a', 10);
```

Review—ctors, dtors, and assignment, cont.

Destructors:

A destructor is responsible for salvaging any reusable resources immediately before an object ceases to exist.

Local variables of class type are destroyed when they go out of scope. Objects occupying memory allocated from the heap are destroyed immediately before that memory is freed due to a call to `delete`.

Conceptually, every class has a destructor. If a destructor is not defined by the implementor of a class, one is generated that essentially does nothing.

A class never has more than one destructor.

Assignment:

The assignment operator is used to change the contents of an existing object based on a given value.

If a class `X` has no assignment operator defined that accepts an object of type `X`, an assignment operator using memberwise assignment is generated.

Do not confuse initialization with assignment. Assignment is used to change the contents of an already existing object.

Review—ctors, dtors, and assignment, cont.

Consider the following class definition and function declarations:

```
class X {
  public:
        X(int);
        X(X&);
        X& operator=(X& rhs);
    };
void f(X val);
void g(X& val);
void h(X* valp);
```

What operations would be invoked for each of the following statements?

```
X x1(1);

X x2 = x1;

X *xp; xp = new X(x1);

x2 = *xp;

f(x2);

f(*new X(x2));

g(x2);

h(xp);

X x3 = 3;
```

IO Streams

Basics of stream I/O

Inserters for user-defined types

Extractors for user-defined types

IO Streams

There are some big problems with I/O via printf (et al.) in the C library:

Not typesafe—prone to mismatch errors
Not extensible—there's no support for user-defined types

As in C, the C++ language itself has no I/O facilities, but the "IO Streams" library is provided as an alternative to C-style I/O.

The IO Streams library overloads the operators << and >> to have additional meaning in C++.

But, the entire C "stdio" library is available as well.

The terms "IO Streams", "I/O Streams", "Stream I/O", and just "Streams" all mean the same thing.

IO Streams, continued

A sample program:

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello, World!" << endl;

    for (int i = 1; i <= 3; ++i)
        cout << "i = " << i << endl;

    cout << "Length and width? " << flush;

    int length, width;
    cin >> length >> width;

    cout << "The area is " << length * width << endl;
}
```

The `<iostream>` header declares `cin` and `cout` as an `istream` and `an ostream`, respectively. Initially, `cin` is associated with standard input and `cout` is associated with standard output.

Using `<<` is called *insertion*. Using `>>` is called *extraction*.

```
Hello, World!
i = 1
i = 2
i = 3
Length and width? 8 13
The area is 104
```

Manipulators

Manipulators are used to cause various changes in the state of a stream.

Print x, y, and z on three separate lines:

```
cout << x << endl << y << endl << z << endl;
```

Prompt for name and don't print a newline:

```
cout << "Name? " << flush;
```

Print every tenth value from 0 to 100 in decimal and hexadecimal:

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    for (int i = 0; i < 100; i += 10)
        cout << dec << setw(3) << i << " "
             << hex << setw(2) << i << endl;
}
```

Output:

```
  0  0
 10  a
 20 14
 30 1e
...
```

IO manipulators, continued

Note that with streams, it usually takes more to say something,

```
for (int i = 0; i < 100; i += 10)
    cout << dec << setw(3) << i << " "
           << hex << setw(2) << i << endl;
```

than with printf:

```
for (int i = 0; i < 100; i += 10)
    printf("%3d %2x\n", i, i);
```

Evaluation of insertion expressions

Consider:

```
int i = 10;  double x = 1.5;  Point p(3,4);

cout << "i = " << i << ", x = " << x << ", p = " << p << endl;
```

Output:

```
i = 10, x = 1.5, p = (3,4)
```

Evaluation:

```
cout << "i = "
```

> Call: `ostream& op<<(ostream&, char*)`
> Side effect: output of `"i = "`
> Return: A reference to `cout` (just a pointer, internally)

```
cout << i
```

> Call: `ostream& op<<(ostream&, int)`
> Side effect: output of `"10"`;
> Return: A reference to `cout`

```
cout << ", x = "
```

```
cout << x
```

> Call: `ostream& op<<(ostream&, double)`
> Side effect: output of `"1.5"`

```
cout << ", p = "
```

```
cout << p
```

> Call: `ostream& op<<(ostream&, const Point&)`

```
cout << endl
```

> `Calls:  ostream& op<< (ostream&, ostream& (f)(ostream&));`
> Side effect: output of newline; flushes buffer

Inserters for user-defined types

Imagining Line::getSlope(), consider this example:

```
Point a(1,1), b(4,2);
Line ln(a,b);

cout << "The slope of the line from " << a << " to " << b
      << " is " << ln.getSlope() << "." << endl;
```

Output:

```
The slope of the line from (1,1) to (2,4) is 3.
```

An overloaded definition of << for Point is required:

```
ostream& operator<< (ostream& o, const Point& p)
{
      o << "(" << p.getX() << "," << p.getY() << ")";
      return o;
}
```

Inserters for user-defined types, continued

An inserter for Line that uses the inserter for Point:

```
ostream& operator<<(ostream& o, const Line& line)
{
    o << "[ " << line.getP1()  << ", " << line.getP2() << " ]";
    return o;
}
```

Now we can write:

```
Point a(1,1), b(4,2);
Line L(a,b);

cout <<  "L = " << L << endl;
cout << Line(Point(0,0), Point(100,50) ) << endl;
```

Output:

```
L = [ (1,1), (4,2) ]
[ (0,0), (100,50) ]
```

What's the Java counterpart for user-defined stream inserters?

Inserters for user-defined types, continued

The nearest analog in Java for user-defined stream inserters is to override `Object.toString()`:

```
class Point {
    ...
    public String toString() {
        return "(" + getX() + "," + getY() + ")";
        }
  }

class Line {
    ...
    public String toString() {
        return "[ " + getP1()  + ", " + getP2() + " ]";
        }
    }
```

Usage:

```
Point a = new Point(1,1), b = new Point(4,2);
Line L = new Line(a,b);

System.out.println("L = " + L);
System.out.println(
    new Line(new Point(0,0), new Point(100,50)));
```

Output:

```
L = [ (1.0,1.0), (4.0,2.0) ]
[ (0.0,0.0), (100.0,50.0) ]
```

Inserters for user-defined types, continued

Imagine an inserter for our String class:

```
String a = "purple";
String b = "parsnips";

cout << "a = " << a << ", b = " << b << endl;
cout << "a + b = " << a + " " + b << endl;
```

Output:

```
a = purple, b = parsnips
a + b = purple parsnips
```

Problem: Write it!

Sidebar: A handy macro

This preprocessor macro works with inserters to conveniently produce labeled output.

```
#define ShowVal(x) #x " = " << (x) << "; "
```

Usage:

```
int i = 7;
double x = 3.4;
Point p(5,10);

cout << ShowVal(i) << ShowVal(x) << ShowVal(p) << endl;
```

Output:

```
i = 7; x = 3.4; p = (5,10);
```

In contrast:

```
cout << "i = " << i << "; x = " << x << "; p = " << p
     << ";" << endl;
```

Note that the macro relies on the unary # preprocessor operator and the fact that adjacent string literals are concatenated.

Extractors for user-defined types

Providing an extractor for Point allows this:

```
Point p1, p2; // assumes default constructor

cin >> p1 >> p2;
```

A simple extractor that considers a Point to be two numbers separated by whitespace:

```
istream& operator>>(istream& i, Point& p)
{
    double x, y;

    i >> x >> y;

    p = Point(x,y);

    return i;
}
```

Extractors for user-defined types, continued

A loop to read and write Points:

```
while (true) {
    cout << "Point? " << flush;
    Point p;
    cin >> p;
    if (!cin)
        break;
    cout << "p = " << p << endl;
    }
```

Interaction:

```
Point? 3 4
p = (3,4)
Point? 1.2 3.4
p = (1.2,3.4)
Point? 10
                        (carriage return)
20
p = (10,20)
```

Writing an extractor that handles input such as "(2.3   , 4.5)" is more involved.

Question: What's going on with if (!cin) ... ?

Inheritance

Basics of inheritance in C++

Virtual member functions

Abstract classes and methods

Virtual destructors

Base class initialization

Inserters and inheritance

The protected access specifier

Inheritance basics

In general, inheritance in C++ is very similar to Java.

In Java, inheritance is indicated with the keyword extends:

```
class Clock { }

class AlarmClock extends Clock { }
```

In C++, inheritance is indicated by following the class name with a colon and a superclass specification:

```
class Clock { };

class AlarmClock : public Clock { };
```

Unlike Java, C++ supports three forms of inheritance: public, private, and protected. Public inheritance in C++ is essentially equivalent to inheritance in Java.

C++ programmers commonly use the term "base class" as a synonym for "superclass", and "derived class" as a synonym for "subclass".

Inheritance basics, continued

A fundamental language design choice in Java is that every class is a direct or indirect subclass of Object. If a Java class doesn't name a superclass, Object is assumed. For example, the class declaration

```
    class Clock { }
```

is equivalent to

```
    class Clock extends Object { }
```

The result of having Object as a direct or indirect superclass of every class is of course that every instance of every class can be treated as an Object.

This allows great generality when coding: A variable of type Object can refer to an instance of any class; an array of type Object[ ] can hold instances of any combination of classes; methods such as toString() can be invoked on any object, etc.

Inheritance basics, continued

The language design choice made in C++ is the opposite of Java: there is no common base class.

In the early days of C++ some class libraries borrowed ideas from Smalltalk and used a common base class such as Object. Classes such as String, List, and Date were derived from Object. Working with those libraries was somewhat similar to working with Java today.

The C++ Standard Library does not introduce a common base class. By far the most common situation is that a C++ system is composed of a forest of class trees[1] rather than a single class tree as in Java.

Having a common base class provides many advantages. Why was that route was not taken in C++?

[1] Actually, due to multiple inheritance in C++, it is a forest of directed graphs.

Inheritance basics, continued

Public inheritance in C++ has the same essential property as inheritance in Java: A derived class inherits the member functions and data members of the base class.

Example:

```
class Clock {
  public:
    Clock();
    void setTime(Time);
    Time getTime();
  private:
    Time itsTime;
    };

class AlarmClock: public Clock {
  public:
    AlarmClock();
    void setAlarmTime(Time);
    void setAlarm(bool);
  private:
    Time  itsAlarmTime;
    bool   isAlarmSet;
    };
```

Instances of AlarmClock have three data members: itsTime, itsAlarmTime, and isAlarmSet.

Along with setAlarmTime() and setAlarm(), an AlarmClock can respond to setTime() and getTime().

As in Java, derived class member functions do not have access to private members of a base class.

Inheritance basics, continued

We can create instances of derived classes and work with them just like any other class in C++:

```
AlarmClock ac;

ac.setTime(Time("now") + Duration("3m"));
ac.setAlarmTime(Time("6:00am"));
ac.setAlarm(true);

AlarmClock *acp = new AlarmClock;
acp->setAlarmTime(Time("now") + Duration("5h"));

AlarmClock alarm_battery[5];
```

Note that Time and Date are imagined for this example. They are not in the standard library.

Inheritance basics, continued

A very useful rule:

> If B and D are classes and B is the base class of D, we may reference instances of D using a pointer of type B*.

Example:

```
Clock *cp = new AlarmClock;  // Clock is B; AlarmClock is D

cp->setTime(Time("8:00am"));
```

Keeping in mind that a Java variable of class type in Java is essentially a pointer, here's the Java version:

```
Clock c = new AlarmClock();

c.setTime(new Time("8:00am"));
```

Because the type of cp is Clock*,

```
cp->setAlarm(true);     // Compilation error
```

won't compile, even though the referenced object really is an AlarmClock.

If we cast cp, the call is permitted:

```
((AlarmClock*)cp)->setAlarm(true);
```

Inheritance basics, continued

Reminder:

> If B and D are classes and B is the base class of D, we may reference instances of D using a pointer of type B*.

A similar rule applies to references:

> If B and D are classes and B is the base class of D, a B& may refer to an instance of D.

Example:

```
AlarmClock ac;

Clock& c = ac;
c.setTime(Time("8:00am"));
```

Just as with pointers, we can't call an AlarmClock method using c unless we cast:

```
c.setAlarm(true);                   // Compilation error

((AlarmClock&)c).setAlarm(true);    // OK
```

Inheritance basics, continued

Because an instance of AlarmClock occupies more memory than an instance of Clock, it is almost always a Bad Idea to assign an instance of AlarmClock to an instance of Clock, even though the language allows it:

```
Clock c;
AlarmClock ac;

c = ac;                    // It does compile...
```

This is called *slicing* or *shearing*, because the AlarmClock portion is lost.

In this case, c is a valid Clock but that's not true in general. For example, a pointer in the base class portion may refer to a data member in the derived class portion.

Of course, if we slice and then cast back to the derived class, the best we can hope for is a program fault sooner, rather than later:

```
Clock c;
AlarmClock ac;

c = ac;
((AlarmClock*)&c)->setAlarm(true);       // Probably clobbers
                                         //  something. With luck
                                         //  it blows up now.
```

Inheritance basics, continued

In short, if we want to treat an instance of a derived class as an instance of a base class, we must refer to the instance using a pointer or a reference.

Consider an array that is to hold an arbitrary mixture of a varying number of Clocks and AlarmClocks. Additionally, the array may need to also hold Clock-derived classes that are currently not imagined.

The <u>only</u> choice is an array of Clock pointers:

```
Clock *clocks[MAXCLOCKS];
```

Here's a routine that sets a number of clocks to (about) the same time:

```
void setClocks(Clock *clocks[ ], const Time& t)
{
   for (int i = 0; clocks[i] != 0; i++)   // Assumes 0-terminated
      clocks[i]->setTime(t);
}
```

Usage:

```
Clock* clocks[MAXCLOCKS];
clocks[0] = new Clock;
clocks[1] = new AlarmClock;
clocks[2] = new Clock;
clocks[3] = 0;

setClocks(clocks, Time("12:00"));
```

Problem: Explain why Clock clocks[N], AlarmClock clocks[N], and Clock& clocks[N] are all unsuitable choices.

Virtual member functions

Consider a Java class hierarchy to represent geometric shapes:

```
class Shape {
    public double getArea() { return 0; } // Should be abstract...
    }

class Rectangle extends Shape {
    public Rectangle(double w, double h) { itsW = w; itsH = h; }
    public double getArea() { return itsW * itsH; }
    public double itsW, itsH;
}
```

An attempted analog in C++:

```
class Shape {
    public:
        double getArea() { return 0; }
    };
class Rectangle: public Shape {
    public:
        Rectangle(double w, double h) : itsW(w), itsH(h) { }
        double getArea() { return itsW * itsH; }
    private:  double itsW, itsH;
};
```

Test code: (it reports `area = 0!`)

```
Shape *sp = new Rectangle(3,4);
cout << "area = " << sp->getArea() << endl;
```

What's wrong?

Virtual member functions, continued

At hand:

```
class Shape {
    public:
        double getArea() { return 0; }
    };
class Rectangle: public Shape {
    public:
        Rectangle(double w, double h) : itsW(w), itsH(h) { }
        double getArea() { return itsW * itsH; }
    private:  double itsW, itsH;
};
```

By default, C++ does not use *virtual dispatch* (also called *dynamic binding*) for member functions.

In contrast, Java uses virtual dispatch unless a method is declared to be final. Consider this Java code:

```
Shape s = new Rectangle(3,4);  // Java
double a = s.getArea();
```

The idea of virtual dispatch is that the exact routine that will be called by s.getArea() is not known until execution. All that is assumed at compile time is that s will reference an instance of Shape or a subclass of Shape.

When the code is executed, the object referred by s is examined to determine which getArea() should be called. In the case above it is Rectangle.getArea().

Why does C++ not use virtual dispatch by default?

Virtual member functions, continued

At hand: *Why does C++ not use virtual dispatch by default?*

If virtual dispatch is used by default, then every instance of every class (that has any methods) must contain enough information to support run-time lookup of methods, and that lookup would be done on every call.

The overhead to support virtual dispatch is actually very small—typically one more word of memory per object and one pointer dereference per call, but imposing that default overhead would conflict with the C++ philosophy of not imposing overhead for features you don't use.

Virtual member functions, continued

The solution is simple: add the virtual specifier to Shape::getArea():

```
class Shape {
    public:
        virtual double getArea() { return 0; }
    };
class Rectangle: public Shape {
    ...no changes...
};
```

The virtual specifier indicates that virtual dispatch is to be used for calls to that member function.

If a class has any virtual functions then every instance will have the extra data, typically only a pointer to a *virtual table* (or *vtbl*), required to dynamically bind the call.

Only calls to virtual functions will incur run-time overhead.

Virtual member functions, continued

Summary:

Unless a method is final, Java uses virtual dispatch, deferring until execution the decision of which routine to invoke.

If a C++ member function is virtual, virtual dispatch is used.

If a member function is not virtual, the routine to call is determined at compile time, based on the class type of the expression referencing the method. (*static binding*)

A boiled-down example of third point:

```
class B {
  public: void f() { cout << "B::f()" << endl; }
    };

class D: public B {
  public: void f() { cout << "D::f()" << endl; }
    };
```

Usage:

```
D d;

B *bp = &d;
bp->f();          // Calls B::f() because bp is B*

D* dp = &d;
dp->f();          // Calls D::f() because dp is D*
```

Abstract classes and methods

Logically, the getArea() method of Shape should be abstract—we never intend to create a Shape. Instead we intend to create instances of derived classes such as Rectangle and Circle.

In Java, the abstract keyword expresses those two points:

```
abstract class Shape {
   abstract public double getArea();
   }
```

There is no abstract keyword in C++. Instead:

```
class Shape {
   public:
      virtual double getArea() = 0;
   };
```

The '= 0' indicates that getArea() is a *pure virtual method*—C++ lingo for an abstract method.

Note that '= 0' has nothing to do with the return type. It is simply the syntactic mechanism used in C++.

C++ has no class-level specification that a class is abstract. A C++ class is considered abstract iff it defines at least one pure virtual method, or if it inherits one that has not been overridden.

More on Shape et al.

Here is a more complete version of the shape hierarchy:

```
class Shape {
  public:
        Shape() { }
        virtual double getArea() const = 0;
        virtual double getPerimeter() const = 0;
        virtual double getBoundingBoxArea() const = 0;
    };

class Rectangle: public Shape {
  public:
        Rectangle(double w, double h) : itsW(w), itsH(h) { }

        double getArea() const { return itsW * itsH; }

        double getPerimeter() const {
                return 2 * (itsW + itsH);
                }

        double getBoundingBoxArea() const {
                return getArea();
                }

  private:
        double itsW, itsH; // width and height
};
```

More on Shape et al.

```
class Circle: public Shape {
  public:
        Circle(double radius) : itsR(radius) { }

        double getArea() const {
                return Geometry::PI * itsR * itsR;
                }

        double getPerimeter() const {
                return Geometry::PI * (itsR * 2);
                }

        double getBoundingBoxArea() const {
                return Rectangle(itsR*2, itsR*2).getArea();
                }
  private:
        double itsR;
        };
```

Constructors and destructors

When an instance of a derived class is constructed, the base class part is built first and then the derived class.

The order is reversed for destruction.

A simple inheritance hierarchy:

```
class Base { };

class Derived: public Base { };

class MoreDerived: public Derived { };
```

Assuming the presence of instrumented constructors and destructors, here's what we'd see:

```
Code:     {  Base b;  puts("---");  }
Output:   Base(), ---,  ~Base()

Code:     {  Derived d;  puts("---");  }
Output:   Base(), Derived(), ---, ~Derived(), ~Base()

Code:     {  MoreDerived m;  puts("---");  }
Output:   Base(), Derived(), MoreDerived()
          ---
          ~MoreDerived(), ~Derived(), ~Base()
```

An ugly detail: virtual destructors

Consider the following code:

```
Base *b;

b = new Derived;
cout << "deleting..." << endl;
delete b;
```

The output, assuming instrumented constructors and destructors:

```
Base()
Derived()
deleting...
~Base()
```

The destructor for Derived *is not called!*

The solution:

```
class Base {
    public:
        Base() { }
        virtual ~Base() { }
    };
```

By default, destructors are not virtual. By making ~Base() virtual, when an object referenced by a Base* is destroyed, virtual dispatch is used to call the destructor.

Why not make destructors implicitly virtual?

Base class initialization

Consider a modification to Shape that associates a one-character tag with each shape:

```
class Shape {
  public:
    Shape(char tag) : itsTag(tag) { }
    virtual double getArea() = 0;
    virtual double getPerimeter() = 0;
    virtual double getBoundingBoxArea() = 0;
    char getTag() { return itsTag; }
  private:
    char itsTag;
    };
```

Problem: How can the tag be communicated to the base class constructor via a constructor call such as the following?

```
Rectangle r(3,4,'r');
```

In Java, the solution is a call to super:

```
public Rectangle(double w, double h, char tag) {
    super(tag);
    itsW = w; itsH = h;
    }
```

Base class initialization, continued

In C++ the member initialization list is used to pass values to a base class constructor:

```
class Shape {
  public:
    Shape(char tag) : itsTag(tag) { }
    ...
 private:
    char itsTag;
    };

class Rectangle: public Shape {
  public:
    Rectangle(double w, double h, char tag)
        : Shape(tag), itsW(w), itsH(h) { }
        ...
  private:
    double itsW, itsH; // width and height
};
```

Inserters and inheritance

Consider this inserter for Rectangle,

```
ostream& operator<<(ostream&, const Rectangle&);
```

which works fine with this,

```
Rectangle r(3,4, 'r');

cout << "r = " << r << endl;
```

but not with this:

```
Shape& s = r;

cout << "s = " << s  << endl;

Shape *sp = &r;
cout << "*sp = " << *sp  << endl;
```

Compilation errors are produced:

```
ShapeIOErr.cpp:11: error: no match for 'operator<<' in '
  std::operator<<(std::basic_ostream<char, _Traits>&, const char*) [with
  _Traits = std::char_traits<char>]((&std::cout), "s = ") << s'
/usr/include/c++/3.3.1/bits/ostream.tcc:63: error: candidates are:
[...about 100 more lines of output...]
ShapeIOErr.cpp:14: error: no match for 'operator<<' in '
  std::operator<<(std::basic_ostream<char, _Traits>&, const char*) [with
  _Traits = std::char_traits<char>]((&std::cout), "*sp = ") << *sp'
/usr/include/c++/3.3.1/bits/ostream.tcc:63: error: candidates are:
[...about 100 more lines of output...]
```

Inserters and inheritance, continued

The problem is that virtual dispatch does not come into play with overloaded operators. This code:

```
Shape& s = r;

cout << "s = " << s  << endl;

Shape *sp = &r;
cout << "*sp = " << *sp  << endl;
```

needs a Shape inserter.

Inserters and inheritance, continued

Solution: Provide a pure virtual method print(ostream&) in Shape and override it in derived classes. Call print(...) in an inserter for Shape.

```
class Shape {
  public:
    ...
    virtual void print(ostream&) const = 0;
    };

void Rectangle::print(ostream& o) const {
    o << "Rectangle(" << getTag() << "), "
       << itsW << "x" << itsH << ", area = " << getArea();
    }

void Circle::print(ostream& o) const {
        o << "Circle(" << getTag() << "), r = " << itsR << ", area = "
           << getArea();
    }

ostream& operator<<(ostream& o, const Shape& s)
{
    s.print(o);
    return o;
}
```

What will the inserters for Rectangle and Circle look like?

Complete source for Shape hierarchy

```
#include <iostream>

using namespace std;

class Shape {
  public:
    Shape(char tag) : itsTag(tag) { }
    virtual double getArea() const = 0;
    virtual double getPerimeter() const = 0;
    virtual double getBoundingBoxArea() const = 0;
    char getTag() const { return itsTag; }
    virtual void print(ostream&) const = 0;
  private:
    char itsTag;
    };

ostream& operator<<(ostream& o, const Shape& s)
{
    s.print(o);
    return o;
}
```

Complete source for Shape hierarchy, continued

```
class Rectangle: public Shape {
  public:
    Rectangle(double w, double h, char tag)
    : Shape(tag), itsW(w), itsH(h) { }

    double getArea() const { return itsW * itsH; }

    double getPerimeter() const {
        return 2 * (itsW + itsH);
        }

    double getBoundingBoxArea() const {
        return getArea();
        }

    void print(ostream& o) const {
        o << "Rectangle(" << getTag() << "), "
            << itsW << "x" << itsH << ", area = " << getArea();
        }

  private:
    double itsW, itsH;
};
```

Complete source for Shape hierarchy, continued

```
class Circle: public Shape {
 public:
   Circle(double radius, char tag) : Shape(tag), itsR(radius) { }

   double getArea() const {
       return Geometry::PI * itsR * itsR;
       }

   double getPerimeter() const {
       return Geometry::PI * (itsR * 2);
       }

   double getBoundingBoxArea() const {
       return
         Rectangle(itsR*2, itsR*2, 't').getArea();
       }

   void print(ostream& o) const {
       o << "Circle(" << getTag()
           << "), r = " << itsR << ", area = " << getArea();
       }

 private:
   double itsR;
   };
```

Working with Shapes

```
//
// Calculate the sum of the areas of a list of Shapes.
//
double SumOfAreas(Shape *shapes[ ])
{
        double area = 0.0;

        for (int i = 0; shapes[i] != 0; i++) {
                Shape *sp = shapes[i];
                area += sp->getArea();
                }

        return area;
}

//
// Find the shape with the largest area in a list of Shapes.
//
Shape* Biggest(Shape *shapes[ ])
{
    Shape *bigp = shapes[0];

    for (int i = 0; shapes[i] != 0; i++) {
        Shape *sp = shapes[i];
        if (sp->getArea() > bigp->getArea())
            bigp = shapes[i];
        }
    return bigp;
}
```

Note that we don't need to modify, or even recompile SumOfAreas and Biggest to handle future subclasses of Shape. (Just like Java.)

Working with Shapes, continued

```
int main()
{
    Rectangle a(1,1,'a'), b(3,4,'b'), c(5,10,'c');
    Circle d(1,'d'), e(2,'e'), f(3,'f');

    Shape *shapes[ ] =   { &a, &b, &c, &d, &e, &f, 0};

    cout << "Shapes:" << endl;

    for (Shape **sp = shapes; *sp; sp++) {
        cout << **sp << endl;
        }
    cout << endl;

    cout << "Total area: " << SumOfAreas(shapes)  << endl;

    Shape *bp = Biggest(shapes);
    cout << "Biggest shape: " << *bp << endl;
}
```

Output:

```
Shapes:
Rectangle(a), 1x1, area = 1
Rectangle(b), 3x4, area = 12
Rectangle(c), 5x10, area = 50
Circle(d), r = 1, area = 3.14159
Circle(e), r = 2, area = 12.5664
Circle(f), r = 3, area = 28.2743

Total area: 106.982
Biggest shape: Rectangle(c), 5x10, area = 50
```

The protected access specifier

C++ has a protected access specifier that has the same meaning as in Java: only member functions of derived classes may access protected members.

Recall the "tag" in Shape:

```
class Shape {
  public:
    Shape(char tag) : itsTag(tag) { }
    ...
    char getTag() { return itsTag; }
  private:
    char itsTag;
    };
```

As is, getTag() can be called from anywhere. itsTag can only be accessed in Shape and not in Rectangle or Circle.

If we desire to expose getTag() only to derived classes, we make it protected:

```
class Shape {
  public:
    Shape(char tag) : itsTag(tag) { }
    ...
  protected:
    char getTag() const { return itsTag; }
  private:
    char itsTag;
    };
```

The protected access specifier, continued

Alternatively, we can dispense with getTag() and simply allow derived classes to directly access itsTag:

```
class Shape {
  public:
    Shape(char tag) : itsTag(tag) { }
    ...
  protected:
    char itsTag;
    };
```

Invocation of base class methods

The invocation of a virtual method is always resolved to the method in the most-derived class.

However, it is sometimes useful for a derived class method to invoke its overridden counterpart in a base class.

```
class Window {
    public:
        virtual void Draw() {
            cout << "Window::Draw()" << endl;
            }
        };

class ScrollingWindow: public Window {
    public:
        virtual void Draw() {
            Window::Draw();
            cout << "ScrollingWindow::Draw()" << endl;
            }
        };

int main()
{
    Window *w = new ScrollingWindow();
    w->Draw();
}
```

Output:

```
Window::Draw()
ScrollingWindow::Draw()
```

How is this effect achieved in Java?

Templates

Function templates

A template class: List

Nested class templates

A template class: Table

Inheritance and template classes

Function templates

Templates provide a means to parameterize a class or function with a type.

An example of a template function:

```
template <typename T>
T min(T a, T b)
{
	if (a < b)
		return a;
	else
		return b;
}
```

The function min can be called for any type T for which T < T is valid. (i.e. operator<(T, T) is defined.)

Examples:

```
int i = 5, j = 10;
int minint = min(i, j);

String s1("just"), s2("testing");
String minstr = min(s1, s2);

Point p1(3,4), p2(5,10);
Point minpt = min(p1, p2);
```

Generics in Java 1.5 and C# (Whidbey), are a similar language facility.

Function templates, continued

For reference:

```
template <typename T>  // (old) equivalent:  template <class T>
T min(T a, T b)
{
        if (a < b)
                return a;
        else
                return b;
}
```

Code such as

```
int minint = min(5, 10);
```

causes *template instantiation*. The end result of template instantiation is a *specialization*—a version of the function with the appropriate type(s) plugged in.

The above call to min() would produce this specialization:

```
int min(int a, int b)
{
        if (a < b) return a;
        else       return b;
}
```

Note that template <typename T> simply indicates that entity that follows is a templated class or function and that in it, T refers to a template parameter.

A list class using templates

Entire classes may be parameterized on a type. This is often seen with *container classes*, whose primary purpose is to hold instances of another class and provide access to them.

This program uses a templated class called List to accumulate a sequence of integers and then print them out:

```
int main()
{
    List<int> ilist;

    int i;
    while (cin >> i)
        ilist.add(i);

    cout << ilist.length() << " elements in list: ";

    for (i = 0; i < ilist.length(); i++)
        cout << ilist[i] << " ";  // What operators are overloaded?

    cout << endl;
}
```

Interaction:

```
5 7 7
6 4 3 1
^Z
7 elements in list: 5 7 7 6 4 3 1
```

Note that List<int> is a type name, just like Point and Rectangle.

What would be necessary to handle Points instead of ints?

A list class, continued

A list of Strings:

```
List<String> L;

L.add("ab");
L.add(L[0] + "xyz");
L.add(L[1] / 2);

String s = L[0];
L.add(s * 2);

String s2 = L[0] + L[1] + L[L[0].length()];

cout << L << endl;

List<String> a[2];

a[0].add(s2 * (a[1].length()+1));

cout << a[0] << "," << a[1] << endl;

f(L, a[0], a[1]);
```

Output:

```
[ ab abxyz ab abab ]
[ ababxyzab ],[ ]
```

Note that there's no casting whatsoever.

A list class, continued

Implementation of List:

```
template <typename T> class List {
  public:
    List();
    void add(T);
    T operator[ ](int) const;
    int length() const { return itsLength; }
    int capacity() const { return CAPACITY; }
  private:
    static const int CAPACITY = 100;
    T itsValues[CAPACITY];
    int itsLength;
    };

template <typename T> List<T>::List() : itsLength(0) { }

template <typename T> void List<T>::add(T newValue)
{
    if (itsLength >= CAPACITY)
        return;

    itsValues[itsLength++] = newValue;
}

template <typename T> T List<T>::operator[ ](int index) const
{
    return itsValues[index];
}
```

What are the restrictions on the type held by an instance of List?

Is an assignment operator and/or copy constructor needed?

A more flexible version of List

A class can be templated on the value of a scalar type. In this version of List, the template instantiation can specify an optional capacity for the list, which defaults to 20.

```
template <typename T, int CAPACITY = 20> class List {
  public:
        List();
        void add(T);
        T operator[ ](int) const;
        int length() const { return itsLength; }
        int capacity() const { return CAPACITY; }
  private:
        T itsValues[CAPACITY];
        int itsLength;
        };

template <typename T, int CAPACITY>
List<T, CAPACITY>::List() : itsLength(0) { }

template <typename T, int CAPACITY>
void List<T, CAPACITY>::add(T newValue)
{
        if (itsLength >= CAPACITY)
                return;
        itsValues[itsLength++] = newValue;
}

template <typename T, int CAPACITY>
T List<T, CAPACITY>::operator[ ](int index) const
{ return itsValues[index]; }
```

Usage:

```
List<int> L1;                   // capacity is 20
List<int, 1000> L2;             // capacity is 1000
```

Nested templates

Just as int is a type, so is List<int> and therefore a list of List<int>s can be created:

```
List<int> odds;
for (int i = 1; i <= 10; i += 2)
    odds.add(i);

List<int> evens;
for (int i = 2; i <= 10; i += 2)
    evens.add(i);

cout << "odds:  " << odds << endl;
cout << "evens: " << evens << endl;

List< List<int> > both;
both.add(odds);
both.add(evens);

cout << "both:  " << both << endl;
```

Output: (with an upcoming inserter)

```
odds:  [ 1 3 5 7 9 ]
evens: [ 2 4 6 8 10 ]
both:  [ [ 1 3 5 7 9 ] [ 2 4 6 8 10 ] ]
```

Note that the above code shows a workaround for lexical bug in C++. Instead of this,

```
List<List<int>> both;
```

we must say this:

```
List<List<int> > both; // Note the space in '> >'
```

Nested templates, continued

An inserter for List<T>:

```
template<typename T> ostream&
    operator<<(ostream& o, const List<T>& list)
{
    o << "[ ";

    for (int i = 0; i < list.length(); i++)
        o << list[i] << " ";

    o << "]";
    return o;
}
```

What template specializations would result from the following code?

```
List< List< List<char> > > x;

cout << x;
```

What would sizeof(x) produce?

A template class: Table

Consider a class Table that is similar to List, but can be indexed by values (keys) of any type, not just integers.

To construct a Table, the type of the keys and the type of the value must be specified. This declares a Table indexed by Strings and holding ints:

```
Table<String, int> group_sizes;
```

We now add key/value pairs to the table and print it:

```
group_sizes.add("duo", 2);        // String(const char *) is used
group_sizes.add("trio", 3);
group_sizes.add("quartet", 4);
group_sizes.add("dozen", 12);

cout << group_sizes << endl;
```

Output:

```
[ ( duo -> 2) ( trio -> 3) ( quartet -> 4) ( dozen -> 12) ]
```

An individual value can be accessed via an overloaded indexing operator:

```
int trio = group_sizes["trio"];
int doz  = group_sizes["dozen"];

cout << ShowVal(trio) << ShowVal(doz) << endl;
```

Output:

```
trio = 3; doz = 12;
```

Implementation of Table

```
template <typename K, typename V> class Table {
  public:
    Table();
    Table(V defval);
    void add(K, V);
    V operator[ ](K) const;
    int size() const { return itsSize; }

  private:
    static const int CAPACITY = 100;
    struct Entry {
        K itsKey;
        V itsValue;
        } itsEntries[CAPACITY];
    int itsSize;
    V itsDefaultValue;

    friend ostream& operator<< <>(ostream& o, // Note <>
                                      const Table<K,V>& table);
    };

template <typename K, typename V>
Table<K,V>::Table() : itsSize(0) { }

template <typename K, typename V>
Table<K,V>::Table(V defval)
 : itsSize(0), itsDefaultValue(defval) { }
```

Implementation of Table, continued

```
template <typename K, typename V>
void Table<K,V>::add(K key, V value)
{
    if (itsSize >= CAPACITY)
        return;

    for (int i = 0; i < itsSize; i++)
        if (itsEntries[i].itsKey == key) {
            itsEntries[i].itsValue = value;
            return;
            }

    itsEntries[itsSize].itsKey = key;
    itsEntries[itsSize].itsValue = value;
    itsSize++;
}

template <typename K, typename V>
V Table<K,V>::operator[ ](K key) const
{
    for (int i = 0; i < itsSize; i++)
        if (itsEntries[i].itsKey == key)
            return itsEntries[i].itsValue;

    return itsDefaultValue;
}
```

What requirements does Table place on keys? How about values?

Another instance of Table

Consider a table containing strings and indexed by points:

```
Table<Point, String> point_names("<unknown>");

point_names.add(Point(0,0),           "lower left");
point_names.add(Point(0,100),         "upper left");
point_names.add(Point(100,100),       "upper right");
point_names.add(Point(100,0),         "lower right");

Point which;

while (cout << "Point? " << flush && cin >> which) {
    String name = point_names[which];
    cout << "That point is named " << name << endl;
    }
```

Interaction:

```
Point? 0 0
That point is named lower left
Point? 100 100
That point is named upper right
Point? 5 10
That point is named <unknown>
```

Problem: Describe the data structure represented by x in this declaration:

```
Table<String, Table<String, int> > x;
```

Templates and inheritance

A templated class can be derived from another templated class. Consider OrderedList, a subclass of List that maintains elements in order from smallest to largest:

```
template <typename T> class OrderedList: public List<T> {
    public:
        OrderedList() { }
        virtual void add(T);
        };

template <typename T> void OrderedList<T>::add(T newValue)
{
    if (itsSize >= CAPACITY)
        return;

    int i;
    for (i = 0; i < itsSize; i++) {
        if (newValue < itsValues[i]) {
            //
            // newValue should go in itsValues[i].  Make space
            // there by pushing the other values back one.
            for (int j = itsSize-1; j >= i; j--) {
                itsValues[j+1] = itsValues[j];
                }
            break;
            }
        }

    itsValues[i] = newValue;
    itsSize++;
}
```

Note: The code assumes itsValues is protected.

Templates and inheritance, continued

Usage of OrderedList:

```
OrderedList<char> letters;

for (char *p = "tim korb"; *p; p++)
        letters.add(*p);

cout << letters << endl;
```

Output:

```
[   b i k m o r t ]
```

OrderedList can be used anywhere List can be used.

The C++ Standard Library

- The string class
- The Standard Template Library (STL)
  - The vector class
  - Iterators with vector
  - Algorithms
  - Function objects
  - Algorithms with plain pointers
  - More on iterators and algorithms
  - Constant iterators
  - Iterator adapters
  - The map class
  - The set class

The string class

The C++ Standard Library provides a string class to represent character strings of arbitrary length. NULs are not accommodated.

string provides several constructors and many operators, including assignment, comparison, concatenation, and indexing. There are a variety of member functions for searching and producing substrings.

Strings have *value semantics*—assigning one string to another doesn't result in a shared value.

Unlike Java, strings are mutable—the characters in a string can be changed.

string is defined in the <string> header.

The following slides show a handful of the many operations provided by string.

Example: parse_path

Imagine a routine that breaks a file name such as /home/whm/jtc/survey.cc into three components: directories, basename, and extension. The routine is parse_path:

```
void parse_path(const string& fullpath,
                        string& dirs, string& base, string& ext)
```

Some test code:

```
string line;
while (cout << "Path? " << flush, getline(cin, line)) {
    string dirs, base, ext;
    parse_path(line, dirs, base, ext);
    cout << sq(dirs) << sq(base) << sq(ext) << endl;
    }
```

Interaction:

```
Path? /home/whm/jtc/surveys.cc
dirs = '/home/whm/jtc'
base = 'surveys'
ext = 'cc'

Path? /etc/passwd
dirs = '/etc'
base = 'passwd'
ext = ''

Path? jtcsli.wpd
dirs = ''
base = 'jtcsli'
ext = 'wpd'
```

parse_path

```
void parse_path(const string& fullpath,
        string& dirs, string& base, string& ext)
{
    //
    // Isolate directories
    //
    string::size_type lastslash = fullpath.rfind('/');

    if (lastslash != string::npos)  // string::npos -> not found
        dirs = fullpath.substr(0, lastslash);
    else
        dirs = "";

    string fname =
        fullpath.substr(lastslash+1);  // 2nd arg defaults to npos

    //
    // Isolate base and extension
    //
    string::size_type dotpos = fname.rfind('.');
    if (dotpos != string::npos) {
        base = fname.substr(0,dotpos);
        ext = fname.substr(dotpos+1);
        }
    else {
        base = fname;
        ext = "";
        }
}
```

Note that values are "returned" via reference arguments.

string vs. C-strings

A non-explicit string constructor takes a const char *, enabling a pointer to a C-style string to be used anywhere a string is required.

Instead of a char * or const char * conversion operator string has this member function:

```
        const char *c_str() const;   (slightly simplified)
```

Example:

```
        string snooze(20, 'z');  // Twenty occurrences of 'z'
        const char *p = snooze.c_str();
        cout << snooze << endl;
```

Output:

```
        zzzzzzzzzzzzzzzzzzzz
```

The pointer returned by c_str() references memory managed by the string; do not deallocate it! The contents are only valid while the string exists.

Example: changing a filename extension

This program changes the extension of a file. For example,

```
% chext  Hello.c  cc
```

would be equivalent to "mv Hello.c Hello.cc".

```
#include <cstdio>        (for rename(...))
#include <string>
#include <iostream>
using namespace std;

int main(int argc, char **argv)
{
   string file(argv[1]);
   string new_ext(argv[2]);

   string dirs, base, ext;
   parse_path(file, dirs, base, ext);

   string new_name = base + "." + new_ext;
   rename(file.c_str(), new_name.c_str());
}
```

Note that the file is assumed to be in the current directory, and there's no error handling such as checking the argument count or success of the operation.

ostringstream

Many string synthesis problems are best solved with ostringstream, a ostream subclass that "outputs" to a string.

Imagine a function that forms a name like "H.J.Strappman" from a first, middle, and last name:

```
mk_name("Hanley","James","Strappman"); // "H.J.Strappman"
mk_name("Hanley", "", "Strappman");         // "H.Strappman"
mk_name("", "", "Strappman");                // "Strappman"
```

It's easy with an output string stream:

```
string mk_name(const string& first, const string& middle,
                        const string& last)
{
    ostringstream s;

    if (first.length() != 0)
        s << first[0] << '.';

    if (middle.length() != 0)
        s << middle[0] << '.';

    s << last;

    return s.str();
}
```

ostringstream is defined in the <sstream> header.

C++ Old-Timers should note that ostringstream is preferred over ostrstream.

ostringstream, continued

Here is a templated function that converts values of various types to string:

```
template<class T>
string toString(T x)
{
    ostringstream oss;

    oss << x;
    return oss.str();
}
```

Usage:

```
int i = 73;
double a = 123.456;
Point p(3,4);

string s1 = toString(i);     //     "73"
string s2 = toString(a);     //     "123.456"
string s3 = toString(p);     //     "(3,4)"
```

The counterpart of ostringstring is istringstream, an input string stream.

string really is...

In fact, there is no class named string. string is a typedef for a template specialization:

```
typedef  basic_string<char>  string;
```

There is also a typedef for wstring, for strings of "wide" characters (e.g., 16-bits):

```
typedef basic_string<wchar_t> wstring;
```

The Standard Template Library

Much of the C++ Standard Library is the Standard Template Library, or STL. It is a collection of *containers*, *iterators*, *algorithms*, and *function objects*. It makes extensive use of templates.

There are a handful of containers:

vector	Generalized array. Similar to Java's Vector and ArrayList.
deque	Double ended queue.
list	Doubly linked list. Similar to LinkedList.
set	A sorted collection of unique values. Similar to TreeSet.
multiset	A sorted collection of not necessarily unique values; sometimes called a "bag".
map	An associative array that maintains keys in sorted order. Similar to TreeMap.
multimap	A map that allows duplicate keys

Java's container classes make extensive use of inheritance to produce polymorphic behavior. In contrast, the STL relies mainly on templates to achieve the same ends. The style of programming is often called "generic" programming.

The material here is only an introduction to the STL.

The vector class

A vector can contain elements of any type T that supports copy and assignment operations.

vector is designed to provide random access to elements in constant time (*O(1)*), just like an array. Additionally, elements can be added to the end of a vector in amortized constant time.

A rule of thumb is to use vector to hold a sequence of values unless there is good reason to use a deque or list instead.

vector is defined in the <vector> header.

The vector class, continued

The following code fills a vector with "words" read from standard input and prints the count when done.

```
vector<string> words;

string word;
while (cin >> word)  // whitespace delimited string, by default
    words.push_back(word);

cout << "Read " << words.size() << " words" << endl;
```

This routine produces a vector filled with powers of two:

```
vector<int> powers_of_two(int n)
{
    vector<int> vals;
    for (int i = 0; i < n; i++)
        vals.push_back(1 << i);

    return vals;
}
```

Usage:

```
vector<int> pows = powers_of_two(10);
```

Contents: (2^0 to 2^9)

```
1  2  4  8  16  32  64  128  256  512
```

The vector class, continued

Vectors have value semantics—assigning one to another doesn't produce a shared copy, and comparison is based on contained values.

Example:

```
vector<int> v1 = powers_of_two(10);
vector<int> v2(5, 3);  // Five 3's

cout << SV(v1 == v2) << endl;        // false

v2 = v1;

cout << "--- After v2 = v1 ---" << endl;

cout << SV(v1 == v2) << endl;        // true

v1.pop_back();

cout << "--- After v1.pop_back() ---" << endl;

cout << SV(v1 == v2) << endl;        // false
```

Output: (assuming cout << boolalpha)

```
v1 == v2 = false;
--- After v2 = v1 ---
v1 == v2 = true;
--- After v1.pop_back() ---
v1 == v2 = false;
```

Note that no iostream inserter or extractor is defined for vector.

vector, continued

A vector can be accessed like an array:

```
vector<int> pows = powers_of_two(10);

int i = pows[5];

pows[7] = i + 10;
pows[pows[3]] = pows[3] * pows[9];
```

Note that operator[ ] returns T& and therefore can be assigned to.

The at() method is a range-checked equivalent of operator[ ]:

```
pows = powers_of_two(10);

int i = pows.at(5);

pows.at(7) = i + 10;
pows.at(pows.at(3)) = pows.at(3) * pows.at(9);
```

An out of bounds access with both forms:

```
try {
    cout << pows[500] << endl;
    cout << pows.at(500) << endl;
    }
catch (exception& e) { cout << e.what() << endl; }
```

Output:

```
168043312
vector [ ] access out of range
```

Iterators with vector

Iterators can be used to navigate in STL containers. An iterator to be used with a vector<int> is declared like this:

```
vector<int>::iterator itr;   // nested class
```

One of several vector methods that produce an iterator is begin():

```
vector<int> v = powers_of_two(10);

itr = v.begin();
```

An iterator produced by a vector can be used much like a pointer:

```
cout << *itr << endl;        // prints 1       (pows[0])

++itr;

cout << *itr << endl;        // prints 2       (pows[1])

itr += 7;

cout << *itr << endl;        // prints 256     (pows[8])

itr -=3;
cout << *itr << endl;        // prints 32      (pows[5])

cout << (itr - v.begin()) << endl;      // prints 5

*itr = 20;
cout << *itr << endl;        // prints 20
```

Note that *itr is really itr.operator*().

Iterators with vector, continued

A loop that prints the contents of a vector<int>:

```
vector<int> v = powers_of_two(10);

for (vector<int>::iterator i = v.begin(); i != v.end(); ++i)
    cout << *i << " ";
```

Output:

```
1  2  4  8  16  32  64  128  256  512
```

Important: v.end() is "one past" the last element. Dereferencing v.end() is considered to be an error.

Iterators with vector, continued

For reference:

```
vector<int> v = powers_of_two(10);

for (vector<int>::iterator i = v.begin(); i != v.end(); ++i)
   cout << *i << " ";
```

We can (awkwardly) work backwards with begin() and end():

```
for (vector<int>::iterator i = v.end()-1; i >= v.begin(); --i)
   cout << *i << " ";
```

Output:

```
512  256  128  64  32  16  8  4  2  1
```

The better way to navigate from the rear to the front of a vector is to use rbegin() and rend(). They produce *reverse iterators*:

```
for (vector<int>::reverse_iterator i = v.rbegin(); i != v.rend(); ++i)
   cout << *i << " ";
```

The output is the same.

Note that incrementing a reverse iterator moves backwards.

Speculate: What does v.begin() == v.rend()-1 produce?

Algorithms

The STL includes a number of *algorithms* that are written in terms of iterators. Some are simple and others are sophisticated. STL algorithms are implemented as template functions.

One algorithm is reverse. It is a simply a function that takes two arguments: iterators naming the beginning and (one past) the end of a range of elements in a container. The order of the elements in the range are reversed; the reversal is in-place.

The header <algorithm> is required.

Example:

```
vector<int> v = powers_of_two(10);

reverse(v.begin(), v.end());
print(v, "reversed: ");              // utility routine; non-standard
```

Output:

```
reversed: 512  256  128  64  32  16  8  4  2  1
```

A string is a container. Consider this:

```
string s("Bjarne Stroustrup");
reverse(s.begin(), s.end());
cout << s << endl;
```

Output:

```
purtsuortS enrajB
```

Algorithms, continued

Another algorithm is random_shuffle:

```
random_shuffle(v.begin(), v.end());
print(v, "shuffled: ");

String s("Bjarne Stroustrup");

random_shuffle(s.begin()+1, s.end()-3);
cout << "shuffled: " << s << endl;
```

Output:

```
shuffled: 4  256  64  2  32  8  1  16  512  128

shuffled: Ba rusrjSttenorup
```

Note that the algorithms have no knowledge of the containers. An algorithm is written exclusively in terms of iterators.

If the implementor of a new container implements the appropriate iterators the container should work with any algorithm written in terms of those types of iterators.

This page is intentionally blank

Algorithms, continued

This program reads lines from standard input and prints them in reverse order on standard output:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
using namespace std;

int main()
{
    vector<string> lines;
    string line;

    while (getline(cin, line))
        lines.push_back(line);

    reverse(lines.begin(), lines.end());

    for (vector<string>::iterator i = lines.begin();
            i < lines.end(); i++)
        cout << *i << endl;
}
```

Can it be done without calling `reverse()`?

Function objects

It is possible to overload operator(). Example:

```
class Negate {
    public:
        int operator()(int x) { return -x; }
        };
```

Usage:

```
Negate mk_negative;

int a = mk_negative(3);  //  a = mk_negative.operator()(3);

cout << a << endl;          // prints -3
```

Imagine a function apply_and_print() that takes a vector<int> and an instance of a class like Negate, and prints the result of applying operator() to each value:

```
vector<int> vals = rand_ints(5, 20);  // not in standard library...
print(vals, "Random values:");

Negate mk_negative;
apply_and_print("Negation:", mk_negative, vals);
```

Output: (manually aligned)

```
Random values: 13   3   2   9  0
Negation:     -13  -3  -2  -9  0
```

Function objects, continued

For reference:

```
class Negate {
    public:
        int operator()(int x) { return -x; }
        };


vector<int> vals = rand_ints(5, 20);

Negate mk_negative;
apply_and_print("Negation:\n", mk_negative, vals);
```

Here is apply_and_print:

```
template<typename Function>
void apply_and_print(char *label, Function f, vector<int> v)
{
    cout << label;
    for (vector<int>::iterator i = v.begin(); i != v.end(); ++i) {
        int r = f(*i);              // Equivalent: f.operator()(*i)
        cout << r << "  ";
        }
    cout << endl;
}
```

An instance of a class like Negate is called a *function object* or a *functional*.

Function objects, continued

Here's another class whose instances are function objects:

```
class IsEven {
    public:
        bool operator()(int i) { return i % 2 == 0; }
    };
```

Usage:

```
vector<int> vals = rand_ints(5, 20);
print(vals, "Random values: ");

IsEven even_fcn;
apply_and_print("Even? ", even_fcn, vals);
```

Output: (manually aligned)

```
Random values: 13  3  2  9  0
Even?              0  0  1  0  1
```

More concise:

```
apply_and_print("Even? ", IsEven(), vals);
```

Function objects, continued

Many STL algorithms are written to employ function objects or have an alternate form that uses a function object. One of them:

```
int count(InIter first, InIter last, T value)

int count_if(InIter first, InIter last, Predicate pred)
```

A function object like `IsEven()`, which has a boolean result, is called a *predicate*.

Example:

```
vector<int> vals = rand_ints(10, 5);

print(vals, "Random values: ");

int n = count_if(vals.begin(), vals.end(), IsEven());

cout << n << " even values" << endl;
```

Output:

```
Random values: 3  2  1  1  4  1  1  3  0  3
3 even values
```

Function objects, continued

Here is a templated predicate:

```
template<typename T>
class IsNegative {
    public:
        bool operator()(T i) { return i < 0; }
        };
```

Usage:

```
n = count_if(vals.begin(), vals.end(), IsNegative<int>());
```

The STL includes a number of function objects. One of them is `greater`. Here is a simplified version of it:

```
template <typename T>
class greater {
    public:
        bool operator()(T x, T y)  { return x > y; }
    };
```

Function objects, continued

The STL sort algorithm has two forms:

```
void sort(RandIter first, RandIter last);
void sort(RandIter first, RandIter last, Pr pred);
```

The first form uses the < operator. The second form uses a predicate.

Note the difference between the result when using the first form of sort, and the second, which uses an instance of greater<int>):

```
print(vals, "Values: ");

sort(vals.begin(), vals.end());
print(vals, "Sorted with operator<: ");

sort(vals.begin(), vals.end(), greater<int>());
print(vals, "Sorted with greater<int>: ");
```

Output:

```
Sorted with operator<:    -3  -1  -1  -1  0   1  2  3  3  4

Sorted with greater<int>: 4  3  3  2  1  0  -1  -1  -1  -3
```

STL function objects are defined in the <functional> header.

For reference: print(vector<T>)

```
template <typename T>
void print(const vector<T>& c, char *label = "")
{
    typename vector<T>::const_iterator i;

    cout << label;

    for (i = c.begin(); i != c.end(); ++i)
        cout << *i << "  ";

    cout << endl;
}
```

Algorithms with plain pointers

Another type of container that's compatible with the STL algorithms is `T a[n]`—a plain old array.

Example:

```
char buffer[ ] = "Does it really work??";
int n = strlen(buffer);

cout << buffer << endl;         // Output: Does it really work??

reverse(buffer, &buffer[n]);    // buffer is &buffer[0]

cout << buffer << endl;         // Output: ??krow yllaer ti seoD

random_shuffle(&buffer[2], &buffer[n-2]);

cout << buffer << endl;   // Output: ??lte   eikowasylrroD

fill_n(buffer, 5, 'z');   // start at &buffer[0] and fill with 5 z's

cout << buffer << endl;   // Output: zzzzz   eikowasylrroD
```

More on iterators and algorithms

There are five categories of STL iterators: *input*, *output*, *forward*, *bidirectional*, and *random access*.

There is no class hierarchy for iterators. Instead, iterators are categorized by what they can do. For example, an *output iterator* must support the following operations:

```
*itr = value
++itr
itr++
```
copy constructor

As mentioned earlier, the STL algorithms are written in terms of iterators. The fill_n algorithm (simply a function) looks like this:

```
fill_n(OutIter first, Size n, const T& value)
```

fill_n starts at the position indicated by first, an output iterator, and stores value in each of the next n positions.

Are the four operations listed above sufficient to implement fill_n?

An example of fill_n with a vector:

```
vector<int> nums(10, 77);   // ten copies of 77

fill_n(nums.begin(), 3, 11);
print(nums, "After fill_n: ");
```

Output:

```
After fill_n: 11  11  11  77  77  77  77  77  77  77
```

More on iterators and algorithms, continued

Here are the operations that are required for an iterator to be considered an *input iterator*:

```
*itr            (fetch value)
itr->member     (access member)
++itr
itr++
itr1 == itr2
itr1 != itr2
```
copy constructor

The count algorithm looks like this:

```
int count(InIter first, InIter last, const T& value)
```

Example:

```
vector<int> vals = rand_ints(10, 3); // Not in standard library...

print(vals, "Random values: ");
int n = count(vals.begin(), vals.end(), 0);
cout << "Found " << n << " instances" << endl;
```

Output:

```
Random values: 2  0  2  2  0  0  1  0  2  1
Found 4 instances
```

What output iterator operations does count need to make use of?

More on iterators and algorithms, continued

The call to count is worth another look:

```
vector<int> vals = rand_ints(10, 3);

print(vals, "Random values: ");
int n = count(vals.begin(), vals.end(), 0);
```

The values produced by vals.begin() and vals.end() define a *range* of elements.

The specification of count says that it operates on this range:

```
[ vals.begin(), vals.end() )
```

The notation is borrowed from mathematics: the range [0.0, 1.0) includes 0.0 but stops an infinitesimal amount short of 1.0.

As applied to a container, the range includes the element referenced by vals.begin(), but stops just short of vals.end().

Here's an approximation of count:

```
int count(InIter first, InIter last, const T& value)
{
    int n = 0;
    for ( ; first != last; ++first)
       if (*first == value)
          ++n;
     return n;
}
```

More on iterators and algorithms, continued

The capabilities required of *forward iterators* are roughly a union of input and output iterators:

```
*itr
itr->member
++itr
itr++
itr1 == itr2
itr1 != itr2
```
default constructor
copy constructor
assignment operator

Bidirectional iterators are simply forward iterators that also support `--itr` and `itr--`.

Random access iterators have all the capabilities of bidirectional iterators and also provide pointer-like operations including subscripting, subtraction, comparison, and addition/subtraction of integers.

Here are three more algorithms to consider:

```
FwdIter min_element(FwdIter first, FwdIter last)

void random_shuffle(RandIter first, RandIter last)

void reverse(BiIter first, BiIter last)
```

More on iterators and algorithms, continued

Not all containers produce random access iterators. For example, list produces bidirectional iterators. An iterator produced by a list can't be used with an algorithm that counts on a random access iterator.

For example, this program won't compile:

```
int main()
{
    list<int> L;
    random_shuffle(L.begin(), L.end());
}
```

The error produced by g++ is triggered by the absence of an overloaded operator:

```
stl_algo.h:1643: error: no match for 'operator+' in  '__first + 1'
stl_algo.h:1644: error: no match for 'operator-' in  '__i - __first'
```

Constant iterators

Here is a reasonable first cut at the print() routine shown earlier:

```
template <typename T>
inline void print(const vector<T>& c, char *label = "")
{
    typename vector<T>::iterator i;

    cout << label;
    for (i = c.begin(); i != c.end(); ++i)
        cout << *i << "  ";
    cout << endl;
}
```

Unfortunately, it doesn't compile:

```
error: invalid conversion from `const int* const' to `int*'
```

Why?

Solution:

```
template <typename T>
inline void print(const vector<T>& c, char *label = "")
{
    typename vector<T>::const_iterator i;
                        ^^^^^^^^^^^^^^
    cout << label;
    for (i = c.begin(); i != c.end(); ++i)
        cout << *i << "  ";
    cout << endl;
}
```

Iterator adapters

One thing that can be done with the copy algorithm,

```
OutItr copy(InIter first, InIter last, OutIter result)
```

is this:

```
vector<int> nums(10, 0), fives(3, 5);

print(nums, "nums before: ");

copy(fives.begin(), fives.end(), &nums[4]);

print(nums, "nums after:  ");
```

Output:

```
nums before: 0  0  0  0  0  0  0  0  0  0
nums after:    0  0  0  0  5  5  5  0  0  0
```

Here's a copy call that doesn't do what's expected:

```
copy(fives.begin(), fives.end(), nums.end());  //  A Bad Thing
print(nums, "nums after(2):  ");
```

Output: (it didn't blow up, unfortunately)

```
nums after(2):  0  0  0  0  5  5  5  0  0  0
```

What's wrong with the call?

Iterator adapters, continued

A solution is provided with an *insert iterator*, which is one type of *iterator adapter*.

Instead of this,

```
copy(fives.begin(), fives.end(), nums.end());  //  A Bad Thing
```

do this:

```
copy(fives.begin(), fives.end(), back_inserter(nums));
```

Result: (with all prints)

```
nums before:   0  0  0  0  0  0  0  0  0  0
nums after:    0  0  0  0  5  5  5  0  0  0
nums after(2): 0  0  0  0  5  5  5  0  0  0  5  5  5
```

Here is back_inserter:

```
template<typename C>
back_insert_iterator<C> back_inserter(C& container) {
    return back_insert_iterator<C>(container);
    }
```

Iterator adapters, continued

Another type of iterator adapter is a *stream iterator*. Here's an iterator that turns assignments into output:

```
ostream_iterator<int> prt = ostream_iterator<int>(cout, ",\n");
*prt = 3;
*prt = 4;
*prt = 5;
```

Output: (exact)

```
3,
4,
5,
```

Another example:

```
vector<int> pows = powers_of_two(10);

copy(pows.begin(), pows.end(),
     ostream_iterator<int>(cout, "  "));
```

Output:

```
1  2  4  8  16  32  64  128  256  512
```

A *reverse iterator*, such as produced by `rbegin()` and `rend()`, is another example of an iterator adapter.

The map class

A map is an associative array that holds key/value pairs. The Table class studied in the template section is a very rudimentary equivalent.

Any type K that supports copy, assignment, and comparison can be a key. Any type V that supports copy and assignment can be a value. Keys in a map are unique.

Here is a simple word-occurrence counter:

```
int main()
{
    map<string, int> counts;

    string word;
    while (cin >> word)
        counts[word] += 1;

    map<string, int>::iterator i;

    for (i = counts.begin(); i != counts.end(); ++i) {
        cout << left << setw(15) << i->first
             << right << setw(5) << i->second << endl;
        }

}
```

The map iterator supports a member reference to access the first (key) and second (value) elements of the key/value pair. (Yes, operator-> is overloaded!)

Manipulators are used to produce aligned output.

The map class, continued

An input file:

```
to be or not to be is
not going to be the
question
```

Execution:

```
be                  3
going               1
is                  1
not                 2
or                  1
question            1
the                 1
to                  3
```

The set class

A set is a sorted collection of unique values. A set can contain values of any type T that supports copy, assignment, and comparison.

This program reads file names on standard input, perhaps piped from ls or find, and prints a list of unique file extensions:

```
int main()
{
    set<string> exts;

    string line;
    while (getline(cin, line)) {
        string dirs, base, ext;
        parse_path(line, dirs, base, ext);
        exts.insert(ext);
        }

    cout << exts.size() << " unique extensions:" << endl;
    for (set<string>::iterator i = exts.begin(); i != exts.end(); i++)
        cout << *i << endl;
}
```

Usage:

```
% ls | uniqexts
6 unique extensions:
cc
class
htm
icn
java
pdf
```

Multiple Inheritance

Basics

Multiple inheritance and Java

Ambiguity in multiple inheritance

Virtual base classes

Multiple inheritance basics

If a class has an is-a relationship with more than one class, the use of *multiple inheritance* **may** be appropriate.

Recall Clock:

```
class Clock {
    public:
        Clock();
        void setTime(Time);
        Time getTime();
    private: Time itsTime;
    };
```

Consider a new class, Radio:

```
class Radio {
    public:
        Radio();
        virtual ~Radio();
        void setFrequency(double);
        void setVolume(double);
    private: double itsFrequency, itsVolume;
    };
```

ClockRadio is derived from both Clock and Radio:

```
class ClockRadio: public Clock, public Radio  {
        public:
                ClockRadio();
                virtual ~ClockRadio();
        };
```

This is an example of *multiple inheritance*.

Multiple inheritance basics, continued

Multiple inheritance creates classes whose instances inherit the combined interface, structure, and behavior of two or more classes.

Instances of ClockRadio combine the structure and behavior of a Clock and a Radio:

```
ClockRadio cr;
cr.setTime("10:10");          // Clock::setTime
cr.setVolume(5);              // Radio::setVolume
cr.setFrequency(1000);        // Radio::setFrequency
```

ClockRadio instances have three data members: itsTime, itsFrequency, and itsVolume.

The potential presence of multiple inheritance implies that instead of inheritance relationships defining a tree of classes, they define a directed acyclic graph (DAG) instead.

There is no limit to the size and complexity of class structures built with multiple inheritance.

Multiple inheritance basics, continued

An important aspect of multiple inheritance is that an instance of a class with several base classes can be treated as an instance of any of those base classes.

A ClockRadio is-a Clock and it also is-a Radio. A ClockRadio may therefore be used anywhere either a Clock or a Radio is required.

Imagine a function to tune in a radio station currently playing a particular song:

```
FindSong(Song& song, Radio& radio)
```

FindSong can be used with either a Radio or a ClockRadio:

```
Song s("Chattanooga Choo Choo");

Radio r;
FindSong(s, r);

ClockRadio cr;
FindSong(s, cr);
```

Multiple inheritance basics, continued

ClockRadio pointers can be held in arrays of Clock pointers or Radio pointers:

```
Clock c1, c2;
ClockRadio cr1, cr2;
Radio r1, r2;

Clock* clocks[ ] = { &c1, &c2, &cr1, &cr2 };
Radio* radios[ ] = { &r1, &cr1, &r2, &cr2 };
```

An interesting consequence of multiple inheritance is that casting a pointer may cause adjustment in the value, not just the type:

```
ClockRadio cr1;

ClockRadio *crp = &cr1;
Radio *rp = (Radio*)crp;
Clock *cp = (Clock*)crp;

cout << SV(crp) << SV(rp) << SV(cp) << endl;
```

Output:

```
crp = 0x22fe78; rp = 0x22fe78; cp = 0x22fe90;
```

Note the difference between cp and rp. Note also that the Radio portion is first and the Clock portion second.

Multiple inheritance and Java

Early versions of C++ did not support multiple inheritance. The merit of supporting multiple inheritance was hotly debated. Many persons believe the additional complexity is not worth the benefit.

Java does not support multiple inheritance. It is interesting to consider how ClockRadio might be approached in Java.

One approach is to define a ClockRadio class that contains a Clock and a Radio. The combined set of methods is implemented by appropriately delegating calls to the Clock or the Radio:

```
class ClockRadio {
    private Clock itsClock = new Clock();
    private Radio itsRadio = new Radio();

    public void setTime(Time t)        { itsClock.setTime(t); }
    public Time getTime()              { return itsClock.getTime(); }
    public void setVolume(double f) { itsRadio.setVolume(f); }
    public void setFrequency(double f)
                                       { itsRadio.setFrequency(f); }
    }
```

What are the disadvantages of this approach?

Multiple inheritance and Java, continued

It may be the case that we really don't need to work with a ClockRadio as a Radio, but it would be very convenient to work with it as a Clock. If so, we might inherit from Clock and contain a Radio:

```
class ClockRadio extends Clock {
    private Radio itsRadio = new Radio();

    public void setVolume(double f) { itsRadio.setVolume(f); }
    public void setFrequency(double f)
                                             { itsRadio.setFrequency(f); }
    }
```

Multiple inheritance and Java, continued

We can match the behavior of the C++ ClockRadio by using a combination of interfaces and implementation classes:

```
interface Clock {
    void setTime(Time t);
    Time getTime();
    };

class ClockImpl implements Clock {
    public ClockImpl() { }
    public void setTime(Time t)    { itsTime = t; }
    public Time getTime()          { return itsTime; }
    private Time itsTime;
    };

interface Radio {
    void setFrequency(double f);
    void setVolume(double v);
    };

class RadioImpl implements Radio {
    public RadioImpl() { }
    public void setFrequency(double f)  { itsFrequency = f; }
    public void setVolume(double v)     { itsVolume = v; }
    private double itsFrequency, itsVolume;
    };
```

Multiple inheritance and Java, continued

The grand finale:

```
class ClockRadio implements Clock, Radio {
    private ClockImpl itsClock = new ClockImpl();
    private RadioImpl itsRadio = new RadioImpl();

    public void setTime(Time t) { itsClock.setTime(t); }
    public Time getTime()       { return itsClock.getTime(); }

    public void setFrequency(double f)
                                    { itsRadio.setFrequency(f); }
    public void setVolume(double f) { itsRadio.setVolume(f); }
    }
```

This matches the behavior of ClockRadio in C++: It combines the behavior of both Clock and Radio, and an instance of ClockRadio can be used anywhere an instance of Clock or Radio is required.

Here's the C++ version again:

```
class ClockRadio: public Clock, public Radio  {
        public:
                ClockRadio();
                virtual ~ClockRadio();
        };
```

Ambiguity in multiple inheritance

Multiple inheritance is very expressive but it comes with a cost: there are a number of potential conflicts and ambiguities that can arise. C++ has mechanisms to resolve those problems, but they are elaborate.

A simple example of ambiguity is an identical member function in two base classes:

```
class Clock {
    public:
        ...
        void reset();
    };

class Radio {
    public:
        ...
        void reset();
    };

class ClockRadio: public Clock, public Radio { ... };
```

An instance of ClockRadio can be created, but a call to ClockRadio::reset() is said to be ambiguous:

```
ClockRadio cr;    // OK
cr.reset();          // Ambiguous: Clock::reset() or Radio::reset()?
```

Ambiguity in multiple inheritance, continued

This ambiguity can be resolved by supplying ClockRadio::reset().

If the desired behavior of resetting a clock radio is to reset both the Clock and the Radio, then this is a solution:

```
class ClockRadio: public Clock, public Radio  {
    public:
        ...
        void reset() {
            Clock::reset();
            Radio::reset();
            }
    };
```

What are some alternatives?

Virtual base classes

Consider a skeletal set of classes for a windowing system:

```
class Window {
    public:
        Window(...) { itsWHnd = CreateWindow(...); }
         void setFgColor(...);
    protected:
        WinHandle itsWHnd;
        };

class GraphicalWindow: public Window { // full graphics
    public:
        GraphicalWindow(...);
        void drawRect(...);
        void drawCurve(...);
    };

class TextWindow: public Window {  // like an ASCII terminal
    public:
        TextWindow(...);
        void writeLine(...);
        void gotoRowCol(...);
    };
```

Usage:

```
GraphicalWindow gw;     // opens a window
gw.drawRect(...);       // draws a rectangle

TextWindow tw;          // opens another window
tw.writeLine(...);      // outputs a string as if dumb terminal
```

Virtual base classes, continued

During development we'd like to see debugging output in the window along with the graphics. Multiple inheritance seems to offer a simple solution:

```
class DebugWindow: public GraphicalWindow,
                   public TextWindow {
    ...
    };
```

Usage:

```
DebugWindow dw;

dw.drawRect(...);
dw.writeLine(...);
```

Will it work?

Virtual base classes, continued

Here is a representation of the current structure:

$$\begin{tikzcd}
\text{Window} & & \text{Window} \\
\text{GraphicalWindow} \arrow[u] & & \text{TextWindow} \arrow[u] \\
& \text{DebugWindow} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

The problem is that both GraphicalWindow and TextWindow are Windows. The constructor for Window calls the OS service CreateWindow(). Constructing the GraphicalWindow portion of DebugWindow causes one OS window to be created. A second OS window results from constructing the TextWindow portion of DebugWindow.

We'd see graphical drawing in one window and terminal-like output in the other, instead of seeing both in one window.

Problem: Would the following code compile?

```
        DebugWindow dw;

        dw.setFgColor(...);
```

Virtual base classes, continued

The problem can be solved using a *virtual base class*:

```
class GraphicalWindow: public virtual Window { ... };
class TextWindow: public virtual Window { ... };

class DebugWindow:   // Unchanged
        public TextWindow, public GraphicalWindow { ... }
```

The result is that a DebugWindow contains one instance of Window, not two:

$$\begin{tikzcd}
 & \text{Window} & \\
\text{GraphicalWindow} \arrow[ur] & & \text{TextWindow} \arrow[ul] \\
 & \text{DebugWindow} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Stroustrup describes the effect of a virtual base specification like this:

> "Every virtual base of a derived class is represented by the same (shared) object."

Multiple inheritance: Worth the weight?

It is a fact that multiple inheritance is part of C++. It won't be going away.

The basic idea of multiple inheritance—allowing more than one base class—is very simple and powerful. However, liberal use of multiple inheritance can easily produce a class structure that is very difficult to understand.

It is not a bad idea for projects to adopt guidelines about how much use may be made of multiple inheritance. For example, a very conservative rule is to use multiple inheritance to provide only the functionality of Java interfaces.

A final note about multiple inheritance:

Be sure all base classes have a virtual destructor.

Exceptions

Basics

Objects as exceptions

Stack unwinding

Exception specifications

Inheritance and exception handling

The auto_ptr class

Exception handling basics

In general, the C++ exception handling mechanism is very similar to Java.

In Java an exception is thrown with a throw statement:

```
throw new IllegalArgumentException("positive value required");
```

Java requires the value thrown be assignable to Throwable.

C++ also uses a throw statement, but a value of <u>any type</u> can be thrown. These are all valid:

```
throw 1;

throw "x";

throw Rectangle(3,4);

throw new Rectangle(5,6);
```

Exception handling basics, continued

C++ has a try statement that is almost identical to Java. Example:

```
try {
    g();
    }

catch (int i) { cout << "Caught int = " << i << endl; }

catch (double) { cout << "Caught a double" << endl; }

catch (...)   { cout << "Caught something" << endl; }
```

If an int is thrown by g(), the first catch clause is selected. The value thrown is assigned to i, and it is printed.

If a double is thrown, the second clause is selected. As is the case with parameter lists, an identifier need not be specified if the value doesn't need to be referenced.

The third catch has an ellipsis (...) for the exception declaration. It is literally three periods. It catches any value. No identifier can be specified in conjunction with it. If used, it must be the last catch clause.

Just as in Java, a C++ exception will propagate upwards from an arbitrarily deep sequence of calls until it is caught or it propagates out of main. By default, if an exception propagates out of main (i.e., it was never caught), execution is terminated.

C++ has no counterpart for Java's finally clause.

Objects as exceptions

Although C++ allows values of any type to be thrown the common practice is to throw an instance of a class that specifically represents an exception.

With our String class in mind we might create a StringBounds class to represent a bounds error:

```
class StringBounds {
  public:
    StringBounds(int pos, const String& str)
        : itsPos(pos), itsStr(str) { }
    String getStr() { return itsStr; }
    int getPos() { return itsPos; }
  private:
    int itsPos;
    String itsStr;
    };
```

In String:

```
char& String::operator[ ](int pos)
{
    if (pos < 0 || pos >= strlen(itsPtr))
        throw StringBounds(pos, *this);

    return itsPtr[pos];
}
```

Objects as exceptions, continued

Test code:

```
String s = "test";

try {
    char c = s[10];
    cout << "c = " << c << endl;
    }
catch (StringBounds& sb) {
    cout << "Error: string position " << sb.getPos()
           << " out of bounds in '" << sb.getStr() << "'"  << endl;
    }
```

Execution:

```
Error: string position 10 out of bounds in 'test'
```

In the catch, a value is assigned to sb in roughly the same way as for a function call. Note that the catch names a StringBounds reference.

Stack unwinding

Stack unwinding is a key element of the exception handling mechanism in C++. It is an orderly deactivation of scopes (such as function calls) until a suitable exception handler is found.

```
int main()
{
    try { f();  }
    catch (...) { cout << "caught it!" << endl; }
}
void f()
{
    X x1(1);
    g();
}
void g()
{
    X x2(2);

    throw logic_error("oops"); // from <stdexcept>
}
```

With instrumented constructors and destructors, here's the output:

```
X(1), X(2), ~X(2), ~X(1), caught it!
```

Unwinding ensures that each object that was constructed on the stack is destroyed in the process of handling the exception.

How does stack unwinding compare to `setjmp/longjmp` in C?

Is stack unwinding important in Java?

Stack unwinding, continued

Another example of unwinding:

```
    int main() {
        try { f(); }
        catch (...)
        { cout << "Caught something in main()" << endl; }
    }
    void f() {
        X x("f");

        try { g(); }
        catch (const char *s)
        { cout << "caught '" << s << "' in f()" << endl; }
    }

    void g() {
        X x("g");

        if (time(0) % 2) {  // seconds since the epoch
            cout << "Throwing 'odd time'" << endl;
            throw "odd time!";
            }

        X x2("g2");

        cout << "Throwing 2.0" << endl;
        throw 2.0;
    }
```

One possibility:

X(f), X(g), Throwing 'odd time', ~X(g), caught 'odd time!' in f(), ~X(f)

Another possibility:

X(f), X(g), X(g2), Throwing 2.0, ~X(g2), ~X(g), ~X(f), Caught something in main()

Rethrowing an exception

The statement 'throw;' rethrows the current exception. It can be used to filter out exceptions of interest.

```
int main()
{
    for (int i = 1; i <= 12; i++) {
        try { f(i); }
        catch (int i) { cout << "main: Caught " << i << endl; }
        }
}
void f(int i)
{
    try { g(i); }
    catch (int i) {
        cout << "f: Caught " << i << endl;
        if ((i % 3) == 0)
            throw; // Rethrow the current exception
        }
}

void g(int i) { if ((i % 2) == 0) throw i; }
```

Execution:

```
f: Caught 2
f: Caught 4
f: Caught 6
main: Caught 6
f: Caught 8
f: Caught 10
f: Caught 12
main: Caught 12
```

Exception specifications

Java has a notion of checked and unchecked exceptions. If a method invokes a method that throws a checked exception the invoking method must either enclose the call in a try or specify the exception in a throws clause.

For example, a method creating a FileReader must do this:

```
public void f(String fname) {
    try {
        FileReader r = new FileReader(fname);
        ...
        }
    catch (FileNotFoundException e) { ... }
    }
```

or this:

```
public void f(String fname) throws FileNotFoundException {
    FileReader r = new FileReader(fname);
    ...
    }
```

Exception specifications, continued

C++ provides *exception specifications* which, if present, "limit" the exceptions that can be thrown by a routine.

For example, here is a routine f with an exception specification that indicates that only exceptions of type X (and subclasses of X) are expected to be thrown:

```
void f() throw (X)
{
}
```

Unlike Java, it is not guaranteed to be a compile time error to have code that throws an unexpected exception. g++ compiles the following code without complaint:

```
void f() throw(X)
{
    throw Y();
}
```

If f is called, however, the throw Y(); violates the specification and the global function unexpected() is called, which terminates execution, by default.

If no exception specification is present, any value can be thrown as an exception:

```
void f()
{
    throw Y();
}
```

Exception specifications, continued

An exception specification may name any number of types:

```
Window::Window() throw (NoDisplay, ServerFault, NoAccess)
{
    ...
}
```

An empty list indicates that no exceptions may be thrown:

```
void g() throw()
{
    throw X();
}
```

As with the earlier example, the violation might not be caught until execution.

Inheritance and exception handling

Just as in Java, a catch clause can discriminate between base and derived classes:

```
class OSError {
  public:
      OSError(int code);
      int getCode();
      ...
      };

class NetworkError: public OSError {
  public:
      NetworkError(int code, Interface);
      Interface getInterface();
      ...
      };

try { ...some code... }
catch (NetworkError& ne) {
    cout << "Network error; code is "
        << ne.getCode() << ", on interface "
        << ne.getInterface() << endl;
    }

catch (OSError& oserr) {
    cout << "General OS error; code: "
        << oserr.getCode() << endl;
    }
```

Note that if the OSError catch is first, the NetworkError catch is effectively unreachable.

C++ Standard Exceptions

The C++ Standard library defines a small inheritance hierarchy of exceptions: (inheritance is shown via indentation)

```
exception

    logic_error
        domain_error
        invalid_argument
        length_error
        out_of_range

    runtime_error
        overflow_error
        range_error
        underflow_error

    bad_alloc
    bad_cast
    bad_exception
    bad_typeid

    ios_base::failure
```

The exception classes are defined in the <exception> and <stdexcept> headers.

auto_ptr

Consider this routine:

```
void f()
{
    X *xp = new X;
    Y y;

    xp->g();

    delete xp;
}
```

It creates an instance of X and an instance of Y, does some processing, and then destroys the X explicitly. The Y is destroyed implicitly when f() returns and the lifetime of y ends.

If an exception is thrown during X::g(), y will be destroyed when the stack is unwound, but "delete xp" will not be done.

It can be said that the code above is <u>not</u> "exception safe". It is challenging to write exception safe code in C++ and much has been written about the problems involved.

How about wrapping the processing in a try block?

```
try { xp->g(); }
catch (...) { delete xp; throw; }

delete xp;
```

auto_ptr, continued

What's really needed is way to indicate that if a pointer goes out scope, the object it references, if any, is deleted. That's the idea of auto_ptr.

Example:

```
void f()
{
    auto_ptr<X> xp(new X);
    Y y;

    xp->g();
}
```

auto_ptr is a template class. xp is an auto_ptr<X> that holds the address of the X created in the heap. xp resides on the stack just like y.

Because xp is on the stack, ~auto_ptr<X>() is called when xp goes out of scope, either due to f() returning or an exception being thrown.

The auto_ptr destructor simply deletes the pointer it holds.

Note that both the original f() and the auto_ptr version make the same call: xp->g()

An auto_ptr is a "smart pointer". It overloads 'operator->' (a unary postfix operator) so that an expression like xp-> produces the stored value, of type X*. That value in turn is used to invoke X::g(). Think of xp->g() as being this:

```
(xp.operator->()) -> g()
```

auto_ptr, continued

A key property of auto_ptr is that, when used as intended, an object is always "owned" by exactly one auto_ptr. (Why?)

The auto_ptr copy constructor enforces the one owner rule: initializing an auto_ptr<X> with an auto_ptr<X> transfers ownership from the old one to the new one.

For example, the end result of this code,

```
auto_ptr<X> xp1(new X);
auto_ptr<X> xp2(xp1);
```

is that xp2 owns the object created by new X and xp1 can no longer be used—it holds the null pointer.

Example:

```
X* p = new X;
auto_ptr<X> xp1(p);

cout << SV(p) << SV(xp1.operator->()) << endl << endl;

auto_ptr<X> xp2(xp1);
cout << SV(p) << SV(xp1.operator->()) << endl;
cout << SV(p) << SV(xp2.operator->()) << endl;
```

Output:

```
p = 0xa0417e8; xp1.operator->() = 0xa0417e8;

p = 0xa0417e8; xp1.operator->() = 0;
p = 0xa0417e8; xp2.operator->() = 0xa0417e8;
```

auto_ptr, continued

Assignment also enforces the one owner rule:

```
auto_ptr<X> xp1(new X);
auto_ptr<X> xp2(new X);

xp2 = xp1;
```

When done, xp2 can be used to reference the X and xp1 holds a null pointer. *Additionally, the X originally referenced by xp2 was destroyed.*

The behavior of the auto_ptr copy constructor and assignment operator can be used to create a notion of "sources" and "sinks". A source is a routine that returns an auto_ptr instance. A sink uses the contained pointer.

Recall the Int class from the operator overloading section, with instrumentation added:

```
class Num {
    public:
        Num(int i) : value(i) { cout << "Num(" << value << ")" <<
endl; }

        ~Num() { cout << "~Num(" << value << ")" << endl; }

        int getValue() const { return value; }

    private:
        int value;
    };
```

auto_ptr, continued

Here is an auto_ptr source:

```
auto_ptr<Num> makeNum()
{
    static int n = 0;  // "serial number"
    return auto_ptr<Num>(new Num(n++));
}
```

Here is an auto_ptr sink:

```
void printNums()
{
    auto_ptr<Num> ip;

    for (int i = 1; i <= 3; i++)
    {
        ip = makeNum();
        cout << ip->getValue() << endl;
    }
}
```

Problem: Explain the output of printNums():

```
Num(0)
0
Num(1)
~Num(0)
1
Num(2)
~Num(1)
2
~Num(2)
...ad infinitum...
```

auto_ptr, continued

There is much more to auto_ptr (and the general topic of exception-safe code) than is discussed here. This material attempts to simply establish a foundation for further study. Two points, however, are extremely important:

> auto_ptr does not work properly with arrays. For example,
>
> ```
> auto_ptr<X> ap(new X[10]);
> ```
>
> has been observed to create an array of ten X instances, but only destroy X[0] when ap goes out of scope. (Officially, the behavior is undefined.)
>
> The one owner rule of auto_ptr creates fundamental problems with the idea of a container of auto_ptrs. For example,
>
> ```
> List<auto_ptr<Num> >
> ```
>
> would be a Bad Idea.

The semantics of auto_ptr were hotly debated during standardization and not everybody got their way.

Recommendation: A decision about the extent of auto_ptr usage should be made by a project's technical leadership.

Run-Time Type Information

The type_info class

The dynamic_cast operator

Other casting operators

Run-time type information (RTTI)

In Java a wealth of information about class types is available during execution via `Object.getClass()`, the reflection mechanism, and constructs such as `instanceof`.

Type information about C++ objects is available at run-time but it is far more limited than Java. Additionally, some aspects are implementation dependent.

RTTI, continued

A simple class hierarchy:

```
class Cycle { virtual void f() { } };   // 'virtual' probably required...
class Unicycle: public Cycle { };
class Bicycle: public Cycle { };
class TandemBicycle: public Bicycle { };
```

A simple usage of RTTI:

```
void DescribeCycle(Cycle *cp)
{
      cout << "It is a '" << typeid(*cp).name() << "'" << endl;
}
```

Usage:

```
Unicycle u;
TandemBicycle tb;

DescribeCycle(&u);
DescribeCycle(&tb);
```

Output: (with g++ 3.3.1; virtual method required)

```
It is a '8Unicycle'
It is a '13TandemBicycle'
```

Output: (with Visual C++ 5.0)

```
It is a 'class Unicycle'
It is a 'class TandemBicycle'
```

The type_info class

The typeid function returns a reference to a constant type_info object.

The definition of the type_info class is implementation-dependent but must support comparisons of type_info instances and be able to produce the name of a type.

An implementation's type_info is defined in the <typeinfo> header. Here is a representative type_info:

```
class type_info {
  public:
      virtual     ~type_info();
      int         operator==(const type_info&) const;
      int         operator!=(const type_info&) const;
      int         before(const type_info&) const;
      const char *name() const;
  private:
      type_info(const type_info&);
      type_info& operator=(const type_info&);
      ...data members not shown...
      };
```

typeid can be applied to non-class types: (SV is the ShowVal macro)

```
char c;
cout << SV(typeid(c).name()) << SV(typeid(3.4).name())
     << endl << SV(typeid(const char*).name())
     << SV(typeid(10U).name()) << endl;
```

Output:

```
typeid(c).name() = c; typeid(3.4).name() = d;
typeid(const char*).name() = PKc; typeid(10U).name() = j;
```

The type_info class, continued

This routine determines if two Cycles have the same structure by getting a type_info for each and comparing them:

```
bool Isomorphic(Cycle& c1, Cycle& c2)
{
   const type_info& t1 = typeid(c1);
   const type_info& t2 = typeid(c2);

   return t1 == t2;
}
```

Given:

```
Bicycle b, b2;
Unicycle u;
```

The expression...

```
Isomorphic(b, b2)              // produces true
Isomorphic(b, u)               // produces false
Isomorphic(b, (Bicycle&)u))    // produces false
```

The dynamic_cast operator

For reference: (Java code)

```
class Cycle { }
class Unicycle extends Cycle { }
class Bicycle extends Cycle { }
class TandemBicycle extends Bicycle { }

Cycle c = new Cycle();
Cycle u = new Unicycle();
Cycle b = new Bicycle();
Cycle tb = new TandemBicycle();
```

Java's instanceof operator is used to test whether a value is "assignment compatible" with a named type. Examples:

```
b instanceof Cycle          is true
u instanceof Bicycle        is false
u instanceof Unicycle       is true
b instanceof Unicycle       is false
tb instanceof Bicycle       is true
```

instanceof can be used to see if a cast will succeed or throw a ClassCastException.

Example:

```
Bicycle bike1 = (Bicycle)u;          // throws C.C.E.
Bicycle bike2 = (Bicycle)tb;         // OK
```

The C++ counterpart for instanceof is dynamic_cast<T>.

The dynamic_cast operator, continued

The dynamic_cast operator tries to convert a pointer of type Base* to a pointer of type Derived*, producing zero if the object pointed to is not an instance of a class derived from Base.

Problem: Given a list of pointers to Cycles how many of them are Bicycles?

A solution with dynamic_cast:

```
int CountBikes(Cycle *cycles[ ])
{
        int nbikes = 0;
        for (int i = 0; cycles[i] != 0; i++) {
                Bicycle *bp =
                        dynamic_cast<Bicycle*>(cycles[i]);
                if (bp != 0)
                        nbikes++;
                }

        return nbikes;
}
```

Invocation:

```
Cycle c; Unicycle u;  Bicycle b, b2;  TandemBicycle tb;

Cycle *cycles[ ] = { &c, &u, &b, &b2, &tb, 0 };
cout << "# of Bikes: " << CountBikes(cycles) << endl;
```

dynamic_cast is said to provide a *typesafe downcast*.

The dynamic_cast operator, continued

As a rule, downcasts are used far more frequently in Java than C++.

Downcasts are commonly used when working with Java classes like Vector, which work with Objects and have methods that return an Object, like Vector.elementAt().

Example:

```
Vector v = loadPoints();

for (int i = 0; i < v.size(); i++) {
    Point p = (Point)v.elementAt(i);
    p.translate(10,20);
    System.out.println(p);
    }
```

A C++ analog that uses our templated List class:

```
List<Point> pts = loadPoints();

for (int i = 0; i < pts.length(); i++) {
    Point p = pts[i];
    p.translate(10,20);
    cout << p << endl;
    }
```

Note that there's no need to cast in the C++ version.

As a rule of thumb, use of dynamic_cast may indicate that C++ facilities are not being fully utilized.

Other casting operators

There are three other casting operators that are similar in appearance to dynamic_cast. They are reinterpret_cast, const_cast, and static_cast.

reinterpret_cast<T>(e) allows any conversion allowed by (T)e. Example:

```
long v = 100;
char *p = reinterpret_cast<char *>(v);
```

const_cast<T>(e) removes the const-ness of the expression e. Example:

```
const *char p = String("xyz");
char *p2 = const_cast<char*>(p);
```

static_cast<T>(e) is intended as a replacement for (T)e where e is of type S and T can be converted to S implicitly. Example:

```
Cycle *cp = get_a_Bicycle();
Bicycle *bp = static_cast<Bicycle*>(cp);
```

Note that static_cast does not perform a run-time check of the type as dynamic_cast does.

Odds and Ends

Namespaces

Member pointers

Type-safe linkage

Reducing header inclusion

Namespaces

Imagine an architectural design application. The developers choose to using a GUI library from company A and some room modeling software from company B.

The GUI library has a key abstraction called Window that represents a window on the screen:

```
class Window { ... };
```

The room modeling software, a non-graphical set of classes that makes extensive use of computational geometry, has classes that represent entities found in buildings:

```
class Room { ... };

class Door { ... };

class Window { ... };
```

One day somebody does this:

```
#include "A.h"          // Headers for GUI library
#include "B.h"          // Headers for room modeling library

int main()
{
    Window w;
}
```

Will there be any problems?

Namespaces, continued

Both companies have certainly made a reasonable choice when naming their Window class. We could perhaps persuade one to supply a version that uses a different name, like a A_Window or attempt some magic with the preprocessor, but neither option is a good one.

The C++ *namespace* facility provides a solution for this problem. C++ namespaces provide an additional level of encapsulation and qualification for identifiers. They are somewhat like packages in Java.

Example:

```
// A.h
namespace A {
    class Window { };
    }

// B.h
namespace B {
    class Room { };
    class Door { };
    class Window { };
    }

#include "A.h"
#include "B.h"
int main()
{
    A::Window root;

    B::Window kitchen_sink;
    B::Room kitchen(kitchen_sink);
}
```

Namespaces, continued

Here is some code that will not compile:

```
#include "A.h"
int f()
{
   Window w;  // Error: Window is undefined
}
```

With clashing classes it's often the case that a given source file uses only one of the classes predominately.

A *using directive* tells the compiler to search the cited namespace in order to resolve a name. This works:

```
#include "A.h"

using namespace A;

int f()
{
   Window root;
}
```

A translation unit may contain any number of using directives, and they may appear anywhere in the file.

The C++ Standard Library is in the namespace std. It is very common to use this directive,

```
using namespace std;
```

to avoid code like this: `std::cout << "Hello!" << std::endl;`

Namespaces, continued

It's reasonable to think of this directive,

```
using namespace A;
```

as a rough analog to this Java import:

```
import com.A_Software.SuperWin.*;
```

One important difference between C++ namespaces and Java packages is that package membership plays a role in member accessibility. Namespaces do not carry a similar implication.

All the names in a namespace don't need to be in a single definition. Namespaces accumulate names and when an identifier is encountered in a translation unit, the then-current accumulation is used.

For example, the following series of namespace definitions is completely equivalent to the all-in-one definition of B used earlier.

```
// Room.h
namespace B {
    class Room { };
    }
// Door.h
namespace B {
    class Door { };
    }
// Window. h
namespace B {
    class Window { };
    }
```

Namespaces, continued

In some cases a `using` directive pulls in names that aren't needed and that cause other conflicts. A `using` declaration is useful in that case.

Example:

```
using namespace A;
void f()
{
   using B::Room;
   using B::Door;

   Window w;          // Window::A
   Door d;            // B::Door
   Room r;            // B::Room
}

Door front_door;      // Error: Not found
...
```

A `using` declaration can be placed at file scope, in which case it's very similar to an import:

```
import com.A_Software.SuperWin.Door;
```

Namespaces, continued

It is possible to create an *alias* for a namespace. One use of an alias is to make a long namespace name, perhaps that of a supplier, easier to deal with:

```
namespace Great_Solutions_Software_Of_North_Dakota {
    class Mosquito { };
    ...
}

namespace gss =
        Great_Solutions_Software_Of_North_Dakota;

gss::Mosquito m;
```

As a whole, the C++ namespace facility is rich, powerful, and complex, but it's not clear that all developers need a deep understanding of it. Having just one developer with broad knowledge of namespaces may be sufficient for a project.

Three of the namespace topics not covered here are *nested namespaces*, *unnamed namespaces*, and *Koenig lookup*.

Member pointers

C++ has the notion of a *member pointer* that can be used in conjunction with a class instance to reference a data member or member function.

```
struct X {
        int i, j;
        char *p1, *p2;
        };

int main()
{
        int    X::*PIMX;
        char  *X::*PCPMX;

        PIMX = &X::i;
        X anX;

        anX.*PIMX = 1; // sets anX.i to 1

        PIMX = &X::j;
        anX.*PIMX = 2; // sets anX.j to 2

        X *xp = &anX;
        PCPMX = &X::p1;
        xp->*PCPMX = "testing";
}
```

The type of PIMX is "pointer to int data member of X". The type of PCPMX is "pointer to char * data member of X".

A class instance <u>is not needed</u> to assign a value to a member pointer.

Member pointers, continued

Recall the print() and reset() methods from CounterGroup:

```
void CounterGroup::print(char *s)
{
    printf("%s", s);
    for (int i = 0; i < itsNumCounters; i++)
            itsCounters[i]->print();
}

void CounterGroup::reset()
{
    for (int i = 0; i < itsNumCounters; i++)
            itsCounters[i]->reset();
}
```

A better solution using a member pointer to reference a member function of Counter:

```
void CounterGroup::doAll(void (Counter::*f)())
{
    for (int i = 0; i < itsNumCounters; i++)
            (itsCounters[i]->*f)();
}

void CounterGroup::print(char *s)
{
    printf("%s", s);
    doAll(&Counter::print);
}

void CounterGroup::reset()
{
    doAll(&Counter::reset);
}
```

Member pointers, continued

Another example:

```
class Polygon {
        public:
                ...
                double getArea(); { return (this->*AreaFcnP)(); }
                enum AreaCalcType {Exact, Approx };
                void SetAreaCalc(AreaCalcType);
        private:
                double ExactArea();
                double ApproxArea();
                double (Polygon::*AreaFcnP)();
                };

void Polygon::SetAreaCalc(AreaCalcType t)
{
        if (t == Exact)
                AreaFcnP = &Polygon::ExactArea;
        else if (t == Approx)
                AreaFcnP = &Polygon::ApproxArea;
}
int main()
{
        Polygon p;

        p.SetAreaCalc(Polygon::Exact);
        cout << p.Area() << endl;

        p.SetAreaCalc(Polygon::Approx);
        cout << p.Area() << endl;
}
```

The type of Polygon::AreaFcnP is "pointer to Polygon member function with no parameters and returning a double".

Type-safe linkage

In addition to compile-time checking of type consistency via header file declarations, C++ provides *type-safe linkage*. Type-safe linkage ensures a match between the declared and defined signatures of a function.

Example:

a.cc:

```
	int f(char *, int);

	main()
	{
		f("a test", 10);
	}
```

b.cc:

```
	int f(int, char *)
	{
		...
	}
```

Compiling and then linking these files together will produce an error citing that the function `int f(char*, int)` is undefined.

Type-safe linkage, continued

The scheme used to provide type-safe linkage using current linker technology is called "name mangling".

The basic idea with name mangling is to transform the name of a function F into a new name, F', that encodes the types of the arguments.

Examples, with g++:

int FCN() encodes as __Z3FCNv

int FCN(int, int, char) encodes as __Z3FCNiic

int FCN(String, int*) encodes as __Z3FCN6StringPi

double Circle::getArea() encodes as __ZN6Circle7getAreaEv

String::String(char *) encodes as __ZN6Circle7getAreaEv

C functions can be called directly from C++ code, but an extern declaration is required to avoid name mangling:

```
extern "C" {
   void some_C_function(int);
   void another_one(char *, int);
   };

void g()
{
   some_C_function(1);
   another_one("x", 1);
}
```

Type-safe linkage, continued

Wrapping a C++ routine allows it to be called from C. Example:

```
// rectlib.cc
extern "C" {
double get_area_of_Rectangle(double w, double h)
{
    Rectangle r(w,h);

    return r.getArea();
}
};
```

A main program in C:

```
// rtest.c
#include <stdlib.h>
#include <stdio.h>
extern double get_area_of_Rectangle(double w, double h);
int main(int argc, char **argv)
{
    double w = atof(argv[1]);
    double h = atof(argv[2]);

    double a = get_area_of_Rectangle(w, h);

    printf("Area of %g x %g rectangle is %g\n", w, h, a);
}
```

Build it:

```
g++ -c rectlib.cc
gcc rtest.c rectlib.o
```

Reducing header inclusion

Compiling a typical C++ source file requires the inclusion of thousands of lines of headers. Unnecessary inclusion of header files, especially in other header files, can greatly increase compilation time.

The declaration of a class B only needs to see the declaration of a class A if B contains A by value or if it references a member of A.

This class declaration does not need a full declaration of A in order to be compiled:

```
class A;
class B {
        public:
                B(A a);
                A f();
                void g(A*);
                void h(A&);
        private:
                A* ptrToA;
                A& refToA;
        };
```

Any of these additions to B require a full definition of A:

```
class B {
        ...
        friend int A::x();
        int z() { return ptrToA->x(); }
        A itsA;
        };
```

If your compiler supports precompiled headers, take time to learn how they work.