

On to C++

Chapters 1 to 25

by

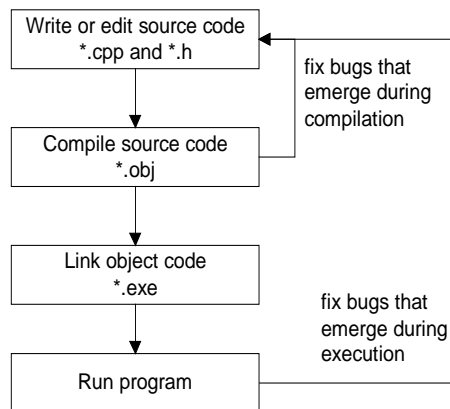
Advanced Object Technologies Limited

What is C++

- ▶ C++ is an object-oriented programming language; emphasise user-defined data types and data-type hierarchies.
- ▶ C++ descends from C, an already very popular programming language.
- ▶ C++, like C, is highly efficient and fast.
- ▶ C++ is supported on almost all hardware platforms.
- ▶ C++ can be more productive compared with C.
- ▶ A large supply of off-the-shelf C++ software modules.

How to Compile & Run a Simple Program

- ▶ Typical C++ development cycle:

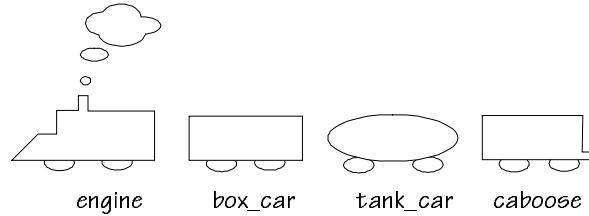


How to Compile & Run a Simple Program

- ▶ C++ programs contain many **function definitions**, each of which may contain **variable definitions**
- ▶ Often, functions and variables are grouped into one or more **class definitions**.
- ▶ Every C++ program must contain a definition for a function named **main**.

How to Compile & Run a Simple Program

- ▶ To compute the volume of a typical railroad box car:



```
void main () {
    11 * 9 * 40;
}
```

How to Compile & Run a Simple Program

- ▶ To print result to standard output:

```
#include <iostream.h>
void main() {
    cout << "The volume of the box car is ";
    cout << 11 * 9 * 40;
    cout << endl;
}
```

tell C++ compiler that
program will use standard
IO library

insertion operator

How to Compile & Run a Simple Program

- ▶ Consecutive calls to the insertion operator can be chained:

```
#include <iostream.h>
void main() {
    cout << "The volume of the box car is "
         << 11 * 9 * 40
         << endl;
}
```

How to Declare Variables

- ▶ C++ **identifier** is a name consisting of letters and digits, the first must be a letter (underscore is considered as a letter)
- ▶ A C++ **variable** is an identifier that serves as the name of a chunk of memory
- ▶ The variable's **data type** determines the size of the memory chunk and the way the bits in the chunk are interpreted

How to Declare Variables

▶ Declaring integer variables:

```
void main() {  
    int height;  
    int width;  
    int length;  
    // ...  
}
```



Box Car

How to Declare Variables

▶ Declaring integer variables:

```
void main() {  
    int height, width, length;  
    // ...  
}
```



Box Car

How to Declare Variables

- ▶ Declaring and initialising integer variables:

```
void main() {  
    int height = 11;  
    int width = 9;  
    int length = 40;  
    // ...  
}
```



How to Declare Variables

- ▶ Declaring and initialising integer variables:

```
void main() {  
    int height = 11, width = 9, length = 40;  
    // ...  
}
```



How to Declare Variables

- ▶ To change the value of a variable, use the assignment operator:

```
void main() {  
    int result, height = 11, width = 9, length = 40;  
    result = height * width * length;  
    cout << "The volume of the box car is " << result <<  
    endl;  
}
```

Assignment operator



Box Car

How to Declare Variables

- ▶ The size of a memory chunk depends on the data type:

The number of bytes for the `char` type

≤ The number of bytes for the `short` type

≤ The number of bytes for the `int` type

≤ The number of bytes for the `long` type

The number of bytes for the `float` type

≤ The number of bytes for the `double` type

≤ The number of bytes for the `long double` type

How to Write Arithmetic Expressions

▶ Example:

```
- 6 * 3 / 2 // Equivalent to ((-6) * 3)/2 = -9
```

▶ Precedence:

<u>Operators</u>	<u>Associativity</u>
+ (unary) - (negation)	right to left
* / % (modulus)	left to right
+ -	left to right
<<	left to right
=	right to left

How to Write Arithmetic Expressions

▶ Convert a value from one type to another:

```
(double) i // i was an int  
double(i)
```

```
(int) d // d was a double  
int(d)
```

How to Read from Keyboard

- ▶ To read in values of height, width, and length of a box car:

```
#include <iostream.h>
void main() {
    int result, height, width, length;
    cout << "Please type three integers." << endl;
    cin >> height;
    cin >> width;
    cin >> length;
    result = height * width * length;
    cout << "The volume of the box car is " << result <<
    endl;
}
```

extraction/input operator



Box Car

How to Read from Keyboard

- ▶ Alternatively, chain calls to the input operator together:

```
#include <iostream.h>
void main() {
    int result, height, width, length;
    cout << "Please type three integers." << endl;
    cin >> height >> width >> length;
    result = height * width * length;
    cout << "The volume of the box car is " << result <<
    endl;
}
```



Box Car

How to Define Simple Functions

- ▶ Lets define a function to compute volume of a box car:

```
#include <iostream.h>
void main() {
    cout << "The volume of the box car is "
         << boxCarVolume(11, 9, 40) << endl;
}
```

arguments



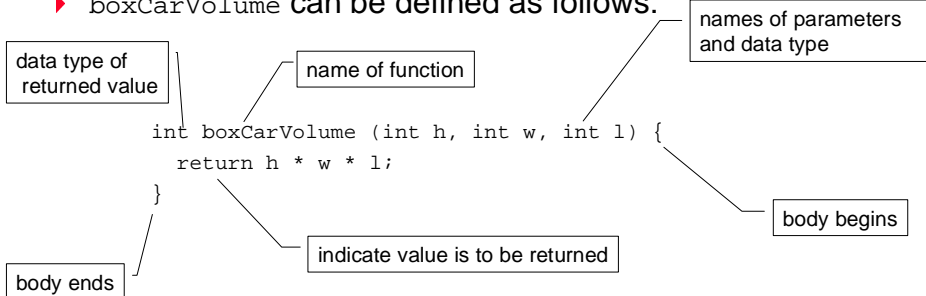
Box Car

How to Define Simple Functions

- ▶ Whenever a call to the `boxCarVolume` function appears, the C++ compiler must arrange for the following to be done:
 - ▶ reserve chunks of memory for the values of the argument expressions
 - ▶ write the values of those argument expressions into those memory chunks
 - ▶ identify the memory chunks with special variables, the **parameters** of the methods -- say, `h`, `w`, and `l`.
 - ▶ evaluate the expression $h*w*l$.
 - ▶ return the value of $h*w*l$ for use in other computations

How to Define Simple Functions

► `boxCarVolume` can be defined as follows:



Box Car

How to Define Simple Functions

► `boxCarVolume` must appear before `main`:

```
#include <iostream.h>
int boxCarVolume (int h, int w, int l) {
    return h * w * l;
}
void main() {
    int height = 11, width = 9, length = 40;
    cout << "The volume of the box car is "
         << boxCarVolume(height, width, length) << endl;
}
```

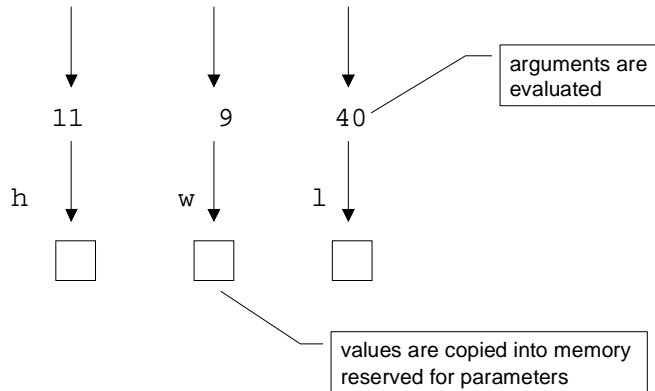


Box Car

How to Define Simple Functions

- ▶ C++ uses **call-by-value**, arguments are copied:

```
boxCarVolume(height, width, length);
```



How to Define Simple Functions

- ▶ C++ allows function **overloading** -- more than one definition for a function:

```
#include <iostream.h>
void displayBoxCarVolume (int h, int w, int l) {
    cout << "The integer volume of the box car is "
        << h * w * l << endl;
}
void displayBoxCarVolume (double h, double w, double l) {
    cout << "The floating-point volume of the box car is "
        << h * w * l << endl;
}
```



Box Car

Benefits of Procedure Abstraction

- ▶ Advantages of **procedure abstraction**:
 - ▶ easier to reuse programs
 - ▶ easier to read (push details out of sight and out of mind)
 - ▶ easier to debug
 - ▶ easier to augment
 - ▶ easier to improve
 - ▶ easier to adapt

Local and Global Variables

- ▶ The **extent** of a variable is the time during which a chunk of memory is allocated for that variable.
- ▶ The **scope** of a variable is that portion of a program where that variable can be evaluated or assigned.
- ▶ Consider **h**, **w**, and **l** in the following programs:

```
double boxCarVolume (double h, double w, double l) {  
    return h * w * l;  
}
```

Local and Global Variables

outside

boxCarVolume fence

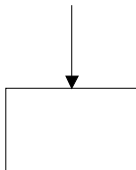
The value of h, w, and l inside this fence are isolated from the values outside

Function computes the value of $h*w*l$ using the values of h, w, l inside this fence

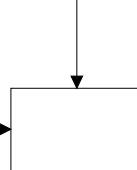
The value of h, w, and l outside the fence, if any, are not affected by the values inside

Variable Scope and Extent

Memory reserved for a variable in calling program



Memory supplying a value for a corresponding parameter in called program



Copy operation

C++ is a **call-by-value** language

Variable Scope and Extent

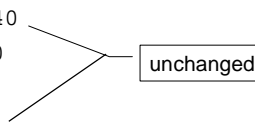
- ▶ To illustrate call-by-value and argument copying:

```
#include <iostream.h>
double boxCarVolume (double h, double w, double l) {
    cout << "Value of l inside boxCarVolume: " << l << endl;
    return h * w * l;
}
main ( ) {
    double l = 40.0;
    cout << "Value of l outside boxCarVolume: " << l << endl
        << "The volume is " << boxCarVolume(10.5, 9.5, l+10)<<
        endl
        << "Value of l after boxCarVolume: " << l << endl;
}
```

Variable Scope and Extent

- ▶ Result of executing program:

```
Value of l outside boxCarVolume: 40
Value of l inside boxCarVolume: 50
The volume is 4987.5
Value of l after boxCarVolume: 40
```



Variable Scope and Extent

- ▶ Program to illustrate scope of parameter values:

```
#include <iostream.h>
double multiplier ( ) {return h * w * l;}          // BUG!
double boxCarVolume (double h, double w, double l) {
    return multiplier ( );
}
main ( ) {
    cout << "The volume of the box car is "
         << boxCarVolume (10.5, 9.5, 40.0) << endl;
    cout << "The value of the parameters are "
         << h << ", " << w << ", and " << l          // BUG!
         << endl;
}
```

Variable Scope and Extent

- ▶ A **local variable** is a variable declared inside a block.
- ▶ C++ function parameters and local variables have **local scope**.
- ▶ Because memory for parameters and variables are reallocated as soon as corresponding method has finished executing, C++ parameters and variables have **dynamic extent**.
- ▶ A **global variable** is a variable define outside any block and has **universal scope** and **static extent**.

Variable Scope and Extent

- ▶ A global variable to store the mathematical constant π :

```

#include <iostream.h>
const double pi = 3.14159;

double tankCarVolume (double r, double l) {
    return pi * r * r * l;
}

main ( ) {
    cout << "The volume of a standard tank car is "
         << tankCarVolume (3.5, 40.0) << endl;
}

```

pi cannot be reassigned

global variable



Tank Car

Variable Scope and Extent

- ▶ A static global variable has a scope of one file:

```

#include <iostream.h>
static const double pi = 3.14159;

double tankCarVolume (double r, double l) {
    return pi * r * r * l;
}

main ( ) {
    cout << "The volume of a standard tank car is "
         << tankCarVolume (3.5, 40.0) << endl;
}

```

static keyword

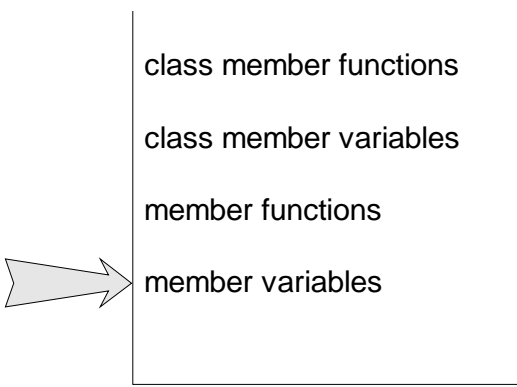
can only be used in this file



Tank Car

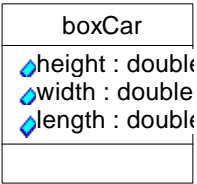
How to Create Classes and Objects

class definition

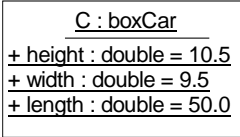
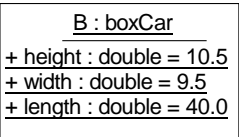
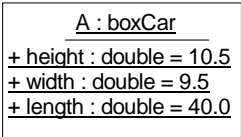


How to Create Classes and Objects

► Definition of box car class:



► Particular box car instances:



How to Create Classes and Objects

- ▶ Definition of box car with memory for height, width, and length:

```
class boxCar {
public:
    double height, width, length;
};
```

member variables



Box Car

How to Create Classes and Objects

- ▶ To create and use an instance of the box car class:

```
#include <iostream.h>
class boxCar {
public: double height, width, length;
};
double boxCarVolume (double h, double w, double l) {
    return h * w * l;
}
main ( ) {
    boxCar x;
    x.height = 10.5; x.width = 9.5; x.length = 40.0;
    cout << "The volume of the box car is "
        << boxCarVolume (x.height, x.width, x.length) <<
        endl;
}
```

create boxCar object

How to Create Classes and Objects

- ▶ Instead of passing three arguments, we can also pass just the object:

```
#include <iostream.h>
class boxCar {public: double height, width, length;};
double volume(boxCar b) {return b.height * b.width *
    b.length;}

main ( ) {
    boxCar x;
    x.height = 10.5; x.width = 9.5; x.length = 40.0;
    cout << "The volume of the box car is "
        << volume(x) << endl;
}
```

pass object to function



Box Car

How to Create Classes and Objects

- ▶ If we also have tank cars, the program can easily be extended:

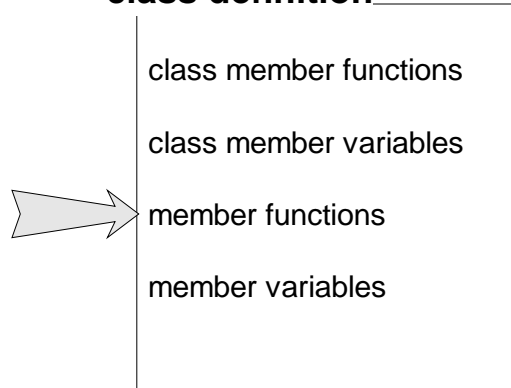
```
#include <iostream.h>
const double pi = 3.14159;
class boxCar {public: double height, width, length;};
class tankCar {public: double radius, length;};
double volume(boxCar b) {return b.height * b.width * b.length;}
double volume(tankCar t) {return pi * t.radius*t.radius*t.length;}

main ( ) {
    boxCar x; x.height = 10.5; x.width = 9.5; x.length = 40.0;
    tankCar y; y.radius = 3.5, y.length = 40.0;
    cout << "The volume of the box car is " << volume(x) << endl
        << "The volume of the tank car is " << volume(y) << endl;
}
```

function overloading

How to Define Member Functions

class definition



How to Define Member Functions

► Original version of box car class:

```
class boxCar {  
    public:  
        double height, width, length;  
};  
double volume(boxCar b) {  
    return b.height * b.width * b.length;  
}
```



Box Car

How to Define Member Functions

- ▶ Change volume to be a member function:

```
class boxCar {
public:
    double height, width, length;
    double volume( ) {
        return height * width * length;
    }
};
```

no arguments needed

assumes member variables



Box Car

How to Define Member Functions

- ▶ Original version of our program:



Box Car

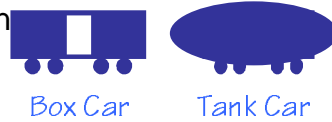


Tank Car

```
#include <iostream.h>
const double pi = 3.14159;
class boxCar {
public: double height, width, length;
};
class tankCar {
public: double radius, length;
};
double volume(boxCar b) {return b.height * b.width *
    b.length;}
double volume(tankCar t) {return pi
    *t.radius*t.radius*t.length;}
```

How to Define Member Functions

- ▶ New version with member function



```
#include <iostream.h>
const double pi = 3.14159;
class boxCar {
public:
    double height, width, length;
    double volume () {return height * width * length;}
};
class tankCar {
public:
    double radius, length;
    double volume () {return pi * radius * radius * length;}
};
```

How to Define Member Functions

- ▶ New version of main function:

```
main ( ) {
    boxCar x; x.height = 10.5; x.width = 9.5; x.length = 40.0;
    tankCar y; y.radius = 3.5, y.length = 40.0;
    cout << "The volume of the box car is " << x.volume( )
         << endl
         << "The volume of the tank car is " << y.volume( )
         << endl;
}
```

implicit parameter

How to Define Member Functions

- ▶ C++ allows you to separate the function definition from the class definition:

```
class boxCar {
public:
    double height, width, length;
    double volume ();
};
```

in boxCar.h

function prototype

```
double boxCar::volume () {
    return pi*radius*radius*length;
}
```

in boxCar.cpp

function definition

class-scope operator

no longer inline



Constructor Member Functions

- ▶ Constructor member functions are special member functions that are called to create class objects
- ▶ Constructor functions names are the same as the name of the class
- ▶ Constructor function has no return value data type
- ▶ In the absence of any constructors defined by you, C++ creates a default constructor, which has no parameters



Constructor Member Function

- ▶ Two constructor functions for tank car:

```
class tankCar {
public:
    double radius, length;
    tankCar ( ) {radius = 3.5; length = 40.0;}
    tankCar (double r, double l) {radius = r; length = l;}
    // . . .
};

main ( ) {
    tankCar t1;
    tankCar t2(3.5, 50.0);
    // . . .
}
```

default constructor

define typical values



Tank Car

Constructor Member Function

- ▶ Constructor for objects with 'const' attributes:

```
class Person {
public:
    const int eye;
    Person (int colour) {eye = colour;} // bug!
};
```

constant attribute

compile time error

- ▶ Correct version:

```
class Person {
public:
    const int eye;
    Person (int colour) : eye(colour) {}
};
```

initialization list

How to Define Getter Member Functions

- ▶ Can refer to member variables directly:

```
tankCar t(3.5, 50.0);  
t.radius
```

- ▶ Better to define getter (reader) member functions:

```
class tankCar {  
    public:  
        double radius, length;  
        double getRadius() const {return radius;}  
        // . . .  
};
```

no change to member variables

How to Define Setter Member Functions

- ▶ Similarly, it is better to define **setter** (writer) methods:

```
class tankCar {  
    public:  
        double radius, length;  
        void setRadius (double r) {radius = r;}  
        // . . .  
};
```



Tank Car

Getter and Setter Methods

- ▶ Can define **getter** and **setter** member functions for imaginary member variables:

```
class tankCar {
public:
    double radius, length;
    double getDiameter () {return radius * 2.0;}
    void setDiameter (double d) {radius = d/2.0;}
    // . . .
};
```



Tank Car

How to Benefit from Data Abstraction

- ▶ Change storage from radius to diameters
 - ▶ no program change needed if using setters and getters

```
. . . x.getRadius() . . . -> . . . x.getDiameter() . . .
.
. . . x.getDiameter() . . . -> . . . x.getDiameter() . . .
. . .
```

- ▶ without setters and getters

```
. . . x.radius . . . -> . . . x.diameter/2.0 . . .
. . .
. . . (x.radius * 2.0) . . . -> . . . x.diameter . . .
```

How to Benefit from Data Abstraction

- ▶ Advantages of data abstraction:
 - ▶ programs are easier to reuse
 - ▶ easier to read
 - ▶ easily to augment what a class provides
 - ▶ easily improve the way that data are stored

Protect Variables from Harmful Access

- ▶ Prevent direct member variable access by putting member variables in the **private** part of the class definition:

```

class tankCar {
  public:
    tankCar() {radius = 3.5; length = 40.0;}
    tankCar (double r, double l) {radius = r; length = l;}
    // . . .
  private:
    double radius, length;
};
  
```

public interface

make private

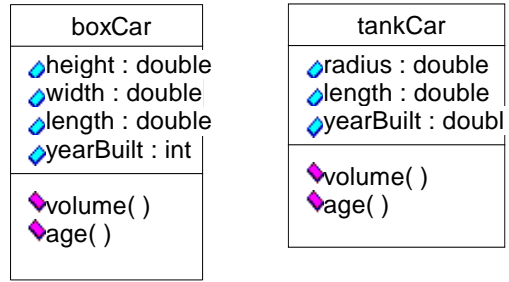


Tank Car

Introduction to C++

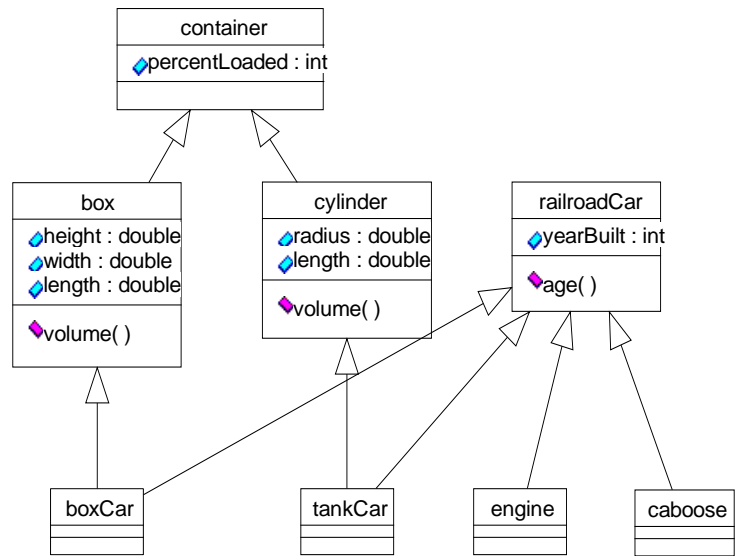
Define Classes with Inheritance

- ▶ Two train car types, with common similarities:



- ▶ better to form higher level abstraction and use inheritance

Define Classes with Inheritance



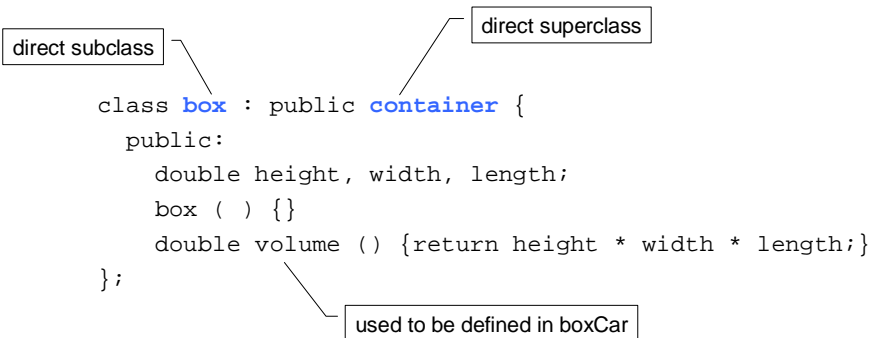
Define Classes with Inheritance

- ▶ Define the parent `container` and `railroadCar` class:

```
int current_year = 2001;
class container {
public:
    int percentLoaded;
    container () {}
};
class railroadCar {
public:
    int yearBuilt;
    railroadCar () {}
    int age () {return current_year - yearBuilt;}
};
```

Define Classes with Inheritance

- ▶ Define the `box` class to inherit from `container` class:



```
class box : public container {
public:
    double height, width, length;
    box ( ) {}
    double volume () {return height * width * length;}
};
```

Define Classes with Inheritance

- ▶ Define the `cylinder` class to inherit from `container` class:

```
class cylinder : public container {
public:
    double radius, length;
    cylinder () {}
    double volume () {return pi*radius*radius*length;}
};
```

Define Classes with Inheritance

- ▶ With all the base class in place, we can now define:

```
class boxCar : public railroadCar, public box {
public:
    boxCar () {height = 10.5; width = 9.2; length =
    40.0;}
};
class tankCar : public railroadCar, public cylinder {
public:
    tankCar () {radius = 3.5; length = 40.0;}
};
class engine : public railroadCar {
public: engine () {}
};
class caboose : public railroadCar {
public: caboose () {}
};
```

Define Classes with Inheritance

- ▶ Constructors will call the default constructor of the direct superclass:

```
void main () {
    boxCar b;
    // . . .
}
```

result:

1. call `railroadCar` default constructor
2. call `container` default constructor
3. call `box` default constructor
4. call `boxCar` default constructor

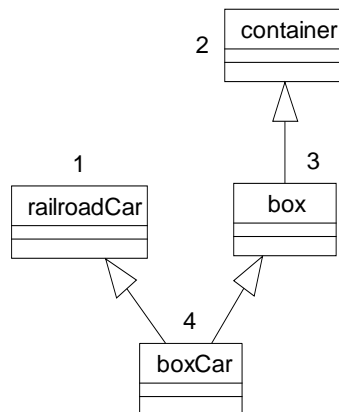


Box Car

Define Classes with Inheritance

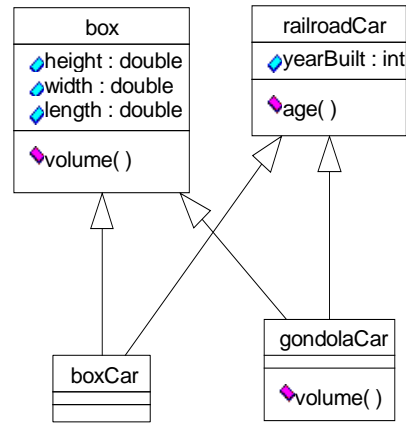
- ▶ Default constructor call sequence:

```
class boxCar : public railroadCar, public box { . . . };
class box : public container { . . . };
```



Box Car

Define Classes with Inheritance



Gondola Car

Define Classes with Inheritance

- ▶ `gondolaCar` class:

```

class gondolaCar : public railroadCar, public box {
public:
    gondolaCar() {height=6.0; width=9.2; length=40.0;}
    double volume() {return 1.2 * height*width*length;}
};
  
```

shadow box volume()

- ▶ `volume` member function in `box` is **shadowed** or **overridden** by the `gondolaCar` `volume` member function



Gondola Car

Define Classes with Inheritance

- ▶ Program to illustrate shadowing and explicit member function calls:

```
main ( ) {
    gondolaCar g;
    cout << "Viewed as a gondola, the car's volume is "
         << g.volume() << " units." << endl
         << "Viewed as a box, the car's volume is "
         << g.box::volume() << " units." << endl;
}
```

result

Viewed as a gondola, the car's volume is 2649.6 units.
Viewed as a box, the car's volume is 2208 units.

Define Classes with Inheritance

- ▶ **overloading** occurs when C++ can distinguish two procedures with the same name by examining the number or types of their parameters
- ▶ **shadowing** or **overriding** occurs when two procedures with the same name, the same number of parameters, and the same parameter types are defined in different classes, one of which is a superclass of the other

Design Classes and Class Hierarchies

- ▶ Explicit-representation principle
- ▶ Modularity principle
- ▶ No-duplication principle
- ▶ Look-it-up principle
- ▶ Need-to-know principle
- ▶ Is-a versus has-a principle

Perform Tests w/ Numerical Predicates

- ▶ Functions that return values `true` or `false` are called **predicates**.
- ▶ C++ uses the keyword `bool` to declare boolean type
- ▶ Any value other than 0 is consider true in C++

```
cout << (1!=2) << endl  
      << (1==2) << endl;
```

Result

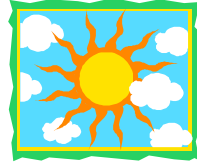
```
1  
0
```

Good C++ style - no space

One-Way and Two-Way Conditional

► One-way conditional statement:

```
#include <iostream.h>
main ( ) {
    int temperature;
    cin >> temperature;
    if (temperature<25) cout << "It is too cold!" << endl;
    if (temperature>50) cout << "It is too warm!" << endl;
}
```



One-Way and Two-Way Conditional

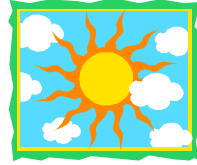
► Two-way conditional statement:

```
#include <iostream.h>
main ( ) {
    int temperature;
    cin >> temperature;
    if (temperature<25) cout << "It is too cold!" << endl;
    else
        if (temperature>50) cout << "It is too warm!" << endl;
}
```



One-Way and Two-Way Conditional

- ▶ Conditional operator (ternary operator):



```
#include <iostream.h>
main ( ) {
    int change;
    cin >> change;
    cout << "The temperature has changed by " << change
        << (change==1 ? " degree" : " degrees") << endl;
}
```

if

then

else

Combine Boolean Expressions

- ▶ The && and || operators return bool type



- ▶ Example:

```
if (25<temperature && temperature<50) {
    cout << "The temperature is normal." << endl;
}
```

and

How to Write While Iteration Statements

- ▶ While iterative loops:

```
int powerOf2 (int n) {  
    int result = 1;  
    while (n!=0) {  
        result = 2 * result;  
        n = n - 1;  
    }  
    return result;  
}
```

How to Write While Iteration Statements

- ▶ The following works for C/C++ but not for Java:

```
int powerOf2 (int n) {  
    int result = 1;  
    while (n) {  
        result = 2 * result;  
        n = n - 1;  
    }  
    return result;  
}
```

How to Write For Loops

- ▶ The same `powerOf2` method with a `for` loop:

```
int powerOf2 (int n) {  
    int counter, result = 1;  
    for (counter = n; counter!=0; counter = counter - 1) {  
        result = 2 * result;  
    }  
    return result;  
}
```

Augmented Assignment Operators

- ▶ Simplify assignment statements:

```
result = result * 2    ->    result *= 2  
result = result + 2    ->    result += 2
```

- ▶ Increment/decrement operators:

```
result = result + 1    ->    result++    or    ++result  
result = result - 1    ->    result--    or    --result
```

powerOf2

- ▶ powerOf2 using decrement operator:

```
int powerOf2 (int n) {  
    int counter, result = 1;  
    for (counter = n; counter--;) {  
        result *= 2;  
    }  
    return result;  
}
```

empty continuation expression

decrement operator

powerOf2

- ▶ Initialise variable inside for loop:

```
int powerOf2 (int n) {  
    for (int counter = n, result = 1; counter--;) {  
        result *= 2;  
    }  
    return result;  
}
```

initialise variable

powerOf2

- ▶ empty for loop statement:

```
int powerOf2 (int n) {
    for (int counter = n, result = 1;    // initialization
        counter--;                      // test
        result *= 2)                    // reassignment
    ;
    return result;
}
```

empty statement

How to Process Data in Files

- ▶ Using a while statement:

```
#include <iostream.h>
double boxVolume(double h, double w, double l) {return h*w*l;}
main ( ) {
    double height, width, depth;
    while (cin >> height >> width >> depth)
        cout << "The volume of a "
            << height << " by " << width << " by " << depth
            << " box car is " << boxVolume(height, width, depth)
            << endl;
    cout << "You appear to have finished." << endl;
}
```

stops if input evaluates to 0

How to Process Data in Files

▶ Keychord to input 0:

	Unix	Dos
To stop a program	control-c	control-c
To force input expression to 0	control-d	control-z

▶ Alternatively input can be from a file:

```
boxCar < test.data
```

Note: input expressions evaluate to 0 when end of file is encountered

How to Process Data in Files

▶ Using `for` statement:

```
#include <iostream.h>
#include <iostream.h>
double boxVolume(double h, double w, double l) {return h*w*l;}
main ( ) {
    double height, width, depth;
    int count;
    for (count = 0; cin >> height >> width >> depth; ++count)
        cout << "The volume of a "
            << height << " by " << width << " by " << depth
            << " box car is "
            << boxVolume(height, width, depth) << endl;
    cout << "You have computed the volumes of "
        << count << " box cars." << endl;
```

stops if input
evaluates to 0

How to Write Recursive Methods

- ▶ A recursive version of powerOf2:

```
int recursivePowerOf2 (int n) {  
    if (n==0) return 1;  
    else return 2 * recursivePowerOf2 (n - 1);  
}
```

How to Write Recursive Methods

- ▶ Female rabbits mature 1 month after birth. Once they mature, female rabbits have one female child each month. At the beginning of the first month, there is one immature female rabbit. Rabbits live forever. And there are always enough male on hand to mate with all the mature females.
- ▶ Compute number of rabbits after n months:

How to Write Recursive Methods

- ▶ The rabbit method:

```
#include <iostream.h>
int rabbits (int n) {
    if (n==0 || n==1) return 1;
    else return rabbits (n - 1) + rabbits (n - 2);
}
void main ( ) {
    cout << "End of month 5, there are " << rabbits (5) <<
    endl;
}
```

- ▶ Fibonacci function!

Solving Definition Ordering Problems

- ▶ The following will not compile:

```
int previous_month (int n) {return rabbits (n - 1);}
int penultimate_month (int n) {return rabbits (n - 2);}
int rabbits (int n) {
    if (n==0 || n==1) return 1;
    else return previous_month (n) + penultimate_month (n);
}
```

- ▶ C++ requires functions to be declared before use

Solving Definition Ordering Problems

- ▶ Add function prototype before use:

```
int rabbits (int);  
  
int previous_month (int n) {return rabbits (n - 1);}  
int penultimate_month (int n) {return rabbits (n - 2);}  
int rabbits (int n) {  
    if (n==0 || n==1) return 1;  
    else return previous_month (n) + penultimate_month (n);  
}
```

function prototype