01

Object-Oriented Programming

Lecturer: Prof Ronan Reilly
email: Ronan.Reilly@nuim.ie

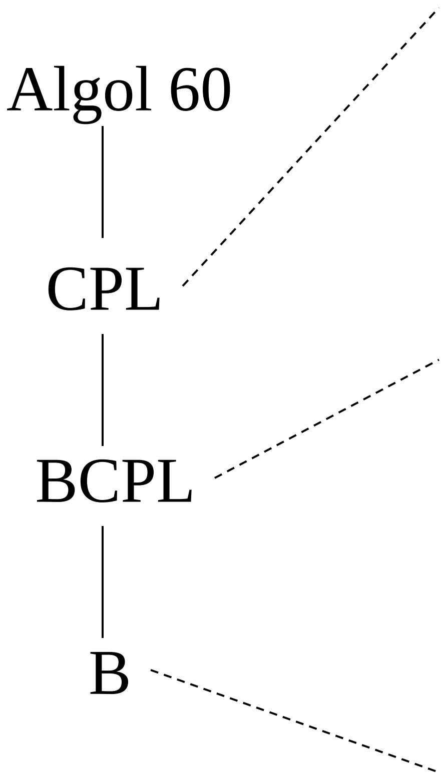
Thanks to previous lecturers on this course

Dr Rosemary Monahan, NUIM

Prof Brian A. Malloy, Clemson University

Dr Paul Gibson, formerly NUIM

A Long Lineage



An even larger language than Algol 60, adding features for business data processing.
Christopher Strachey et. al., 1962-1966

"Basic CPL." Vastly simplified. Typeless: manipulates untyped machine words. Introduced the C-family array idea: **`A[I]`***, written in BCPL as* **`A!I`***, is the same as a reference to the word at address* **`A+I`***.*
Martin Richards (a student of Strachey), 1967

*An even simpler language, developed for systems programming for the first Unix systems at Bell Labs. Included compound assignments (***`a+=b`***), borrowed from Algol 68.*
Ken Thompson, 1969

A Long Lineage, contd.

$$\begin{tikzcd}
B \arrow[d, dash] \\
C \arrow[d, dash] \\
C{+}{+} \arrow[d, dash] \\
Java \arrow[d, dash] \\
C\#
\end{tikzcd}$$

Extension of B (originally, "NB") to take advantage of more hardware (PDP-11). Type system, macro preprocessor, I/O library, etc. Used to reimplement the Unix kernel, and spread widely with Unix.
Dennis Ritchie et. al., 1971-1973

Originally a C preprocessor adding object-oriented features to C: "C with Classes". Added dynamic dispatch, overloaded operators and function names, multiple inheritance, templates, exception handling. Became and remains one of the most widely used languages.
Bjarne Stroustrup, 1984

Motivation for OO

- C++/Java/C# most widely used object-oriented languages,
- OO is the way of the present,
- familiarity increases marketability,
- viable for many years,
- excellent expressivity, and
- excellent speed.
- easy to learn another language

The 1st OO Language: Simula

- Kristen Nygaard and Ole-Johan Dahl, Norwegian Computing Center, 1961
- Simula I: a special-purpose Algol extension for programming simulations: airplanes at an airport, customers at a bank, etc.
- Simula 67: a general-purpose language with classes, objects, inheritance
- Co-routines rather than methods

Smalltalk

- Alan Kay, Xerox PARC, 1972
- Inspired by Simula, Sketchpad, Logo, cellular biology, etc.
- Smalltalk is more object-oriented than most of its more popular descendants
- Everything is an object: variables, constants, activation records, classes, etc.
- All computation is performed by objects sending and receiving messages

Smalltalk's Influence

- The Simula languages and Smalltalk inspired a generation of object-oriented languages
- Smalltalk still has a small but active user community
- Most later OO languages concentrate more on runtime efficiency:
  - Most use static typing (Smalltalk uses dynamic)
  - Most include non-object primitive types as well as objects

C++

- There is an intimidating amount of material in C++
- It was designed to be a <u>powerful tool</u> for <u>professional programmers</u> solving <u>real problems</u> in <u>diverse domains</u>
- It was NOT designed for academia
- It was NOT designed to be a nice, “pure” language, good for teaching students how to program

C# and Java …

Below is a list of features C# and Java share, which are intended to improve on C++
See (http://genamics.com/developer/csharp_comparative.htm)

- Compiles into machine-independent language-independent code which runs in a managed execution environment.
- Garbage Collection coupled with the elimination of pointers (in C# restricted use is permitted within code marked unsafe)
- Powerful reflection capabilities
- No header files, all code scoped to packages or assemblies, no problems declaring one class before another with circular dependencies
- Classes all descend from object and must be allocated on the heap with the **new** keyword
- Thread support by putting a lock on objects when entering code marked as locked/synchronized

C# and Java ctd.

- Interfaces, with multiple-inheritance of interfaces, single inheritance of implementations
- Inner classes
- No concept of inheriting a class with a specified access level
- No global functions or constants, everything belongs to a class
- Arrays and strings with lengths built-in and bounds checking
- The '**.**' operator is always used, no more **->**, **::** operators
- **null** and **boolean/bool** are keywords
- All values are initialized before use
- Can't use integers to govern if statements
- Try Blocks can have a finally clause

Our Focus

The object oriented programming paradigm

- Theoretical background required to understand object oriented principles and issues
- Technical components of object orientation
  - Classes, objects and the associated run time model, memory management, genericity and typing, exceptions, inheritance, polymorphism, dynamic binding
- Programming mostly in C++ and Java

02

Programming Basics

Ronan Reilly

CS613

Introduction to Basic Procedural Programming (with C++)

Try to simulate **compilation** and **execution** in your head

Try different variations on a theme - simply edit the examples

Easiest way to learn programming is to experiment

The computer is a laboratory

If you don’t know something then write a program to give you the answer … if you can

Otherwise, try looking on the web

You can always ask someone (doesn’t have to be me)

```
// Example 1
/* Ronan Reilly
   CS613 C++ Code
*/

main ()// does nothing!!
{}
```

One line comment

Multiple line comment

End of line comment

main program

```
> g++ -o example1 example1.cpp
> example

>
```

Compilation

Execution

Input/Output

Command Line Window

Example1 Variations - what will happen in each case?

```
/* Example 1A
   CS613 C++ Code
*/

int main ()// does nothing!!
{return 0;}
```

```
/* Example 1B
   CS613 C++ Code
*/

int main ()
{}
```

```
/* Example 1C
   CS613 C++ Code
*/

main ()// does nothing!!
{return 0;}
```

```
// Example 1D
int main () {return 7}
```

```
// Example 1E
int main (){return 0.7;;}
```

```
// Example 1F
float main (){return 0.7;}
```

Example Variations - some notes

Try to learn something new from each variation - see 1B

Try not to change more than one thing at once - see 1C

Try to introduce different compiler errors - syntax and semantics - see 1D

Try to distinguish errors from warnings - see 1E

Try to remember that the compiler can be sneaky - see 1F

Example 2: output to the screen

```
// Example 2

#include <iostream.h>
int main (){cout<<"Hello World!";}
```

Writes (outputs) -
Hello World!
To the screen

Question - what sort of variations might be useful ?

- remove the `include`
- change the output `Hello World!`
- add in a `return`

Variation 2A: remove the include

```
// Example 2

//#include <iostream.h>
int main (){cout<<"Hello World!";}
```

Writes (outputs) -

```
Example2A.cc: In function `int main()':
Example2A.cc:4: `cout' undeclared (first use this function)
Example2A.cc:4: (Each undeclared identifier is reported only once
Example2A.cc:4: for each function it appears in.)
```

To the screen

Note: this is a semantic error reported by the compiler

Variation 2B: change the output

```
// Example 2B

#include <iostream.h>
int main (){
cout<<"Hello Universe, ";
cout <<"how are"<<" you"<<"?";}
```

Program contains 2 instructions

You can **stream strings** together

Writes (outputs) -
Hello Universe, how are you?

Note: this is not an error it is the result of the code being executed

Variation 2C: change the output

```
// Example 2C

#include <iostream.h>
int main (){
cout<<"Hello Universe, \n";
cout <<"how are"<<endl<<" you"<<'?';}
```

String, in double quotes, contains special character `\n`

A `?` character between single quotes

Note the use of `endl` - a stream manipulator object!

Writes (outputs) -

```
Hello Universe,
how are
 you?
```

Note: the format - spaces and newlines

Variation 2D: change the output

```
// Example 2D

#include <iostream.h>
int main (){
cout<<"7 + 8 =  \n";
cout <<7+8;}
```

String, in double quotes, contains special character `\n`

An **integer expression** which needs to be evaluated

Writes (outputs) -
7 + 8 =
15

Note: the addition is correct (in base 10)

Variation 2E: add in a return

```
// Example 2E

#include <iostream.h>
int main (){return 0;
cout<<"7 + 8 =  \n";
cout <<7+8;}
```

Return at the beginning of the main body

Writes (outputs) absolutely nothing to the screen

Note: if we put the return in as the last line then we get the same behaviour as before

Example 3 - introducing variables

```
// Example 3

#include <iostream.h>
int main (){
int x = 7+8; cout <<x;
}
```

The variable `x` is identified by the sequence of characters "x". It has type **`int`** (an integer)

`x` is **declared** to be an `int` and **initialised** to value 15 in the same statement.

This outputs -

15

QUESTION: what variations to try?

Variation 3A: What if the variable is not initialised?

```
// Example 3A

#include <iostream.h>
int main (){
int x; cout <<x;
}
```

Output:
686796

Variation 3A: What if the variable is changed?

```
// Example 3B

#include <iostream.h>
int main (){
int x; cout <<x; x = 0;
cout <<x;
}
```

Output:
6867960

Variation 3C: What if the variable is initialised twice?

```
// Example 3C

#include <iostream.h>
int main (){
int x; int x; cout <<x;
}
```

Output:

```
Example3C.cc: In function `int main()':
Example3C.cc:5: redeclaration of `int x'
Example3C.cc:5: `int x' previously declared here
```

Variation 3D: What if the variable is changed?

```
// Example 3D

#include <iostream.h>
int main (){
int x = 1;
x = x+x;cout<<x*x;
}
```

We can change the value of `x` using `x`

This does not change the value of `x`

Output:

```
warning: 'main' : function should return a value;
```

Enumerated types

```
// Example 4

#include <iostream.h>

int main (){
enum days {Sunday, Monday, Tuesday,Wednesday,
           Thursday, Friday, Saturday} yesterday, today;
days tomorrow;
cout << Tuesday<<endl;
cout<< yesterday<<endl<<today<<endl<<tomorrow;
}
```

Outputs:

```
2
686796
143896
117208
```

Boolean type

C++ has a built-in logical or Boolean type

```
// Example 5

#include <iostream.h>

int main (){
bool flag = true;
cout << flag<< endl<<!flag}
```

This outputs:

```
1

0
```

Variation 5a - booleans are really integers?

```
// Example 5a

#include <iostream.h>

int main (){
bool flag1 = 100, flag2 = false, flag3;
cout << flag1<< endl<<flag2<<endl<<flag3;}
```

`true` is any non-zero `int`

`false` is zero

This outputs:

```
1
0
122
```

`true` is output as 1
`false` is output as 0

This was not initialised

What is a Named Constant?

- A **named constant** is a location in memory which we can refer to by an identifier, and in which a **data value that cannot be changed** is stored.

VALID CONSTANT DECLARATIONS

- `const    char     STAR  =   ‘*’  ;`
- `const    float    PI    =  3.14159 ;`
- `const    int      MAX_SIZE  =  50 ;`

Note: all caps

keywords: words reserved to C++

- bool, true, false,
- char, float, int, unsigned, double, long
- if, else, for, while, switch, case, break,
- class, public, private, protected, new,
  delete, template, this, virtual,
- try, catch, throw.

frequently used keywords

Note: there are many more … you may see them in the examples that follow

Example 6: prefix and postfix

```
// Example 6

#include <iostream.h>

int main (){
int x =3, y=3;
cout << ++x <<endl;
cout << y++ <<endl;}
```

++ (prefix) is a **unary** operator

<< is a **binary** operator

++ (postfix) is a **unary** operator

Output:

```
4
3
```

Example 7: a C+ ternary operator

```
// Example 7

#include <iostream.h>

int main (){
int x = 3, y = 4;
cout <<"The max. of x and y is: "
     << (x>y?x:y);
}
```

Output:

The max. of x and y is: 4

`expression1?expression2:expression3`

Means: *if expression1 then expression2 else expression3*

Program with Three Functions

```
// Example 8
#include <iostream.h>
// declares these 2 functions
int  Square ( int );
int  Cube ( int ) ;
int  main ( void ){
cout  <<  "The square of 27 is "
<<   Square (27) << endl ;// function call
cout  <<  "The cube of 27 is "
<<  Cube (27) << endl ;// function call
     return 0 ;
}
int Square ( int  n ){return   n * n ;}
int Cube ( int  n ){return  n * n * n ;}
```

Output:

```
The square of  27 is 729
The cube of 27 is 19683
```

Precedence of Some C++ Operators

Precedence	Operator	Description
Higher ↓	( )	Function call
	+	Positive
	-	Negative
	*	Multiplication
	/	Division
	%	Modulus (remainder)
	+	Addition
	-	Subtraction
Lower	=	Assignment

NOTE: Write some programs to test your understanding of precedence

Type Casting is Explicit Conversion of Type

```
// Example 9
#include <iostream.h>

int  main ( void ){
cout<<"int(4.8) ="<<int(4.8)<<endl;
cout<<"float(5) ="<<float(5)<<endl;
cout<<"float(2/4)="<<float(2/4)<<endl;
cout<<"float(2)/float(4)="<<float(2)/float(4)<<endl;
}
```

Output:

```
int(4.8) =4
float(5) =5
float(2/4)=0
float(2)/float(4)=0.5
```

Output of `float` may look like an `int`

Another way to read char data

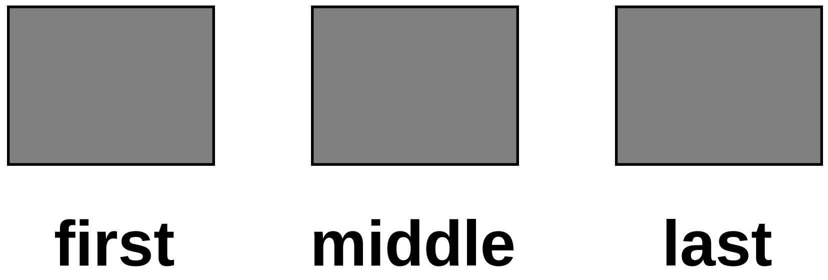
The **get( )** function can be used to read a single character.

It obtains the very next character from the input stream without skipping any leading white space characters.

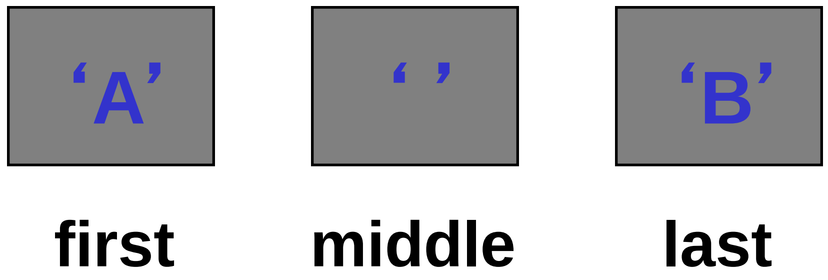
At keyboard you type:

`A[space]B[space]C[Enter]`

```
char   first ;
char   middle ;
char   last ;
```



```
cin.get ( first ) ;
cin.get ( middle ) ;
cin.get ( last ) ;
```



NOTE: The file reading marker is left pointing to the space after the 'B' in the input stream.

C++ control structures

- **Selection**

  if

  if . . . else

  switch

- **Repetition**

  for loop

  while loop

  do . . . while loop

Operator	Meaning	Associativity
!	NOT	Right
*, / , %	Multiplication, Division, Modulus	Left
+ , -	Addition, Subtraction	Left
<	Less than	Left
<=	Less than or equal to	Left
>	Greater than	Left
>=	Greater than or equal to	Left
==	Is equal to	Left
!=	Is not equal to	Left
&&	AND	Left
||	OR	Left
=	Assignment	Right

“SHORT-CIRCUIT” EVALUATION

- **C++ uses short circuit evaluation of logical expressions**
- **this means that evaluation stops as soon as the final truth value can be determined**

Short-Circuit Example

int Age, Height;

Age = 25;

Height = 70;

EXPRESSION

$$\underbrace{(\textbf{Age} > 50)}_{\textbf{false}} \quad \&\& \quad (\textbf{Height} > 60)$$

Evaluation can stop now

Better example

int Number;

float X;

(Number != 0) && (X < 1 / Number)

Beware of *dangling else*

what's the output?

???

```
//Example 12
#include  <iostream.h>
int  main ( void) {
int x = 7, y = 8;
if (x == 0)
   if (y == 0) cout << "yes"<< endl;
else   cout << "no"<<  endl;
cout << "end of output" << endl;
}
```

Beware of *dangling else*

what's the output?

end of output

```
//Example 12
#include  <iostream.h>
int  main ( void) {
int x = 7, y = 8;
if (x == 0)
   if (y == 0) cout << "yes"<< endl;
else   cout << "no"<<  endl;
cout << "end of output" << endl;
}
```

Iteration statements

```
// compute sum = 1 + 2 + ... + n
// using a while loop
int i;
int sum = 0;
i = 1;
while (i <= n) {
    sum += i;
    i++;
}
```

init of the lcv

loop termination condition.

body of the loop

incr of the lcv

Iteration statements

```
// compute sum = 1 + 2 + ... + n
// using for loop

int sum = 0;
for (int i = 1; i <= n; ++i) {
   sum += i;
}
```

i doesn't exist here!

Break

- `break`;
  - the execution of a loop or switch terminates immediately if, in its inner part, the `break`; statement is executed.

Combining *break* and *for*

```
char ch;
int count = 0;

for ( ;   ; )   {
    cin >> ch;
    if (ch == ‘\n’ ) break;
    ++count;
}
```

Switch

```
switch (letter) {
    case ‘N’: cout < “New York\n”;
              break;
    case ‘L’: cout < “London\n”;
              break;
    case ‘A’: cout < “Amsterdam\n”;
              break;
    default:  cout < “Somewhere else\n”;
              break;
}
```

Switch

```
switch (letter) {
    case ‘N’: case ‘n’: cout < “New York\n”;
                           break;
    case ‘L’: case ‘l’: cout < “London\n”;
                           break;
    case ‘A’: case ‘a’: cout < “Amsterdam\n”;
                           break;
    default:  cout < “Somewhere else\n”;
              break;
}
```

What’s the output?

```
int i = 0;
switch ( i ) {
    case 0 :   i+= 5;
    case 1 :   ++i;
    case 2 :   --i;
    case 3 :   i;
    default    ++i;
}
cout << “i is: “ << i << endl;
```

Simple arrays

- subscripts can be an integer expression
- In the declaration, the dimension must be a constant expression

```
const int LENGTH = 100;
...
int a[LENGTH]
...
for (int i=0; i<LENGTH; i++)
   a[i] = 0;  // initialize array
```

known at compile time!

Functions:
3 parameter transmission modes

- pass by *value (default)* **local copy**
- pass by *reference (&)* **pass the address**
- pass by *const reference (const &)*

good for big structures

Functions: example of pass by value

```
int sqr(int x) {




}
```

a local copy is made

The Swap Function

```
void swap(int x, int y)
{
  // Create a temporary variable
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

The swap doesn’t happen.

Why?

Passing values by reference

- C/C++ passes parameters by value, i.e. a copy of the variable is passed to the function, not the actual value itself.
- C++ can pass the actual variables themselves
  - known as *passing parameters by reference*.
  - To do this we place an & between the parameters type name and the parameter tag.

The New Swap Function

```
void swap(int &x, int &y)
{
  // Create a temporary variable
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

What about functions and arrays / structures?

Functions: example of pass by reference

```
void swap(int &x, int &y) {




}
```

Address

Functions: pass by *const reference*

- Makes sense with large structures or objects

We'll use it when
we make objects.

- const &

Arrays are passed by reference

```
const int MAX = 100;
void init(int a[], int x) {
   for (int i = 0, i < MAX; ++i)
   a[i] = rand() % 100; // remainder
   x = 99;
}

main() {
    int a[MAX], x = 7;
    init(a, x);
    cout << a[0]  << ‘\t’ << x << endl;
}
```

More examples: See AddArray.cc and AddArray.java

Pointers

What are they for ?

- Accessing array elements
- Passing arguments to a function when the function needs to modify the original argument.
- Passing arrays and strings to functions
- Obtaining memory from the system
- Creating data structures from linked lists

Pointers

- It <u>is</u> possible to do without pointers:
  - arrays can be accessed with array notation rather than pointer notation
  - a function can modify arguments passed, by reference, as well as those passed by pointers.
- However, in order to obtain the most from the language it is essential to use pointers.

The Address Operator &

It is possible to find out the address occupied by a variable by using the address of the operator &.

Addresses

The actual addresses occupied by variables in a program depend on many factors, such as

- the computer the program is running on,
- the size of the operating system,
- and whether any other programs are currently in memory.

- For these reasons no two computers will give the same answer to the above program.

Creating Pointers

- For creation of a pointer variable that can hold the address of a data type *int*, an asterisk is used to write the type definition of *ptr* such as

```
int  *ptr;
```

- As a result of this declaration, room for an address is allocated to *ptr*.

The NULL Pointer

- No address has been placed in *ptr*, so its value is undefined. With *ptr* being undefined, a comparison involving *p* would be an error, although most C++ compilers would not flag the mistake.
- It is possible to assign the value constant NULL to indicate that *ptr* does not point to a memory allocation (*ptr* is no longer undefined).

Printing out variables that hold address values are useful.

```
void main()
{
    int var1 = 11;
    int *ptr;       //defines a pointer
    cout <<endl <<&var1;
    ptr = &var1;
    cout <<endl <<ptr;
}
```

The * means *pointer to*. **ptr** is a pointer to an **int** (it can hold the address of integer variables).

Defining pointer variables

- defining pointer variables

```
char      *cptr;
int       *iptr;
float     *fptr;
```

- defining multiple pointer variables

```
char      *ptr1, * ptr2, * ptr3;
```

Putting values in pointers

- Before a pointer is used a specific address must be placed in it :

```
ptr = &var1;
```

- A pointer can hold the address of any variable of the correct type;
- **But…**
  - it must be given some value, otherwise it will point to an arbitrary address (because it has to point to something).

Accessing the variable pointed to

```
void main(){
        int  x = 22;
        int *ptr;
        ptr = &x;
        cout <<endl << *ptr;
   }
```

int *ptr declares ptr as a pointer to an integer

ptr holds the address of x

***ptr** holds the contents of x

Using Pointers to Modify Variables

```
void main(){
  int var1, var2;
  int *ptr;

  ptr = &var1;
  *ptr = 37;
  var2 = *ptr;

  cout <<endl<< var2;
}
```

Indirect Addressing

The * is used in declaration is different to the
 * used in assignment.

Using the * to access the value stored in an addesss is called *indirect addressing*, or sometimes *dereferencing* the pointer.

Pointers/References

```
//Pointers and references
//pointersF.cc

#include<iostream>
using namespace std;

int main()
{

int n = 10;
int *pn = &n;
int **ppn = &pn;

cout<<"ppn="<<ppn<<endl;
cout<<"*ppn="<<*ppn<<endl;
cout<<"**ppn="<< **ppn<<endl;
}
```

Output

```
ppn=0xa88b8
*ppn=0xa88bc
**ppn=10
```

In Summary ...

`int v;`	defines variable v of type int
`int *p;`	defines p as a pointer to int
`p = &v;`	assigns the address of variable v to pointer p
`v = 3;`	assigns 3 to v
`*p = 3;`	assigns 3 to v using indirect addressing, referring to the same variable v using its address at p

Void pointer

Note : by declaring a pointer as

```
void* ptr;
```

is declaring a general purpose pointer that can point to any data type irrespective of its type. It is possible to assign the address of an *int*, *float etc.*, to the same pointer variable, this eliminates the need to declare a separate pointer variable for each data type.

Pointers to Arrays

How could we write the following code using pointers...

```
void main(){
  int j;
  int intarray[5] = {31,54,77,52,93};
  for (j=0; j < 5; j++)
     cout <<endl <<intarray[j];
}
```

Pointer Constants

The expression intarray is the address where the system has decided to place the array, and it will stay at this address until the program terminates. intarray is a constant.

Using Pointer Notation

- The expression *(intarray+3) means accessing the fourth element in the array i.e. intarray[3]
- The name of an array is its address.
- The expression intarray + j is thus the address with something added to it. Hence we can write:

```
for (j=0; j < 5; j++)
    cout <<endl << *(intarray+j);
```

03

Object Oriented Programming

Classes

Principles of Modularity

1. Decomposibility: decompose complex problems into subproblems
2. Composibility: production of software elements that may be freely combined with each other to produce new software.
3. Continuity: small changes in specifications yield small changes in architecture.

Principles of Modularity

4. Information Hiding

- How do we “advertise” the capabilities of a module?
- Every module should be known to the outside world through an official, “public” interface.
- The rest of the module’s properties comprises its “secrets”. It should be impossible to access the secrets from the outside.
- Justifications:
  - Continuity
  - Decomposability

Principles of Modularity

5. The Open-Closed Principle : the **open/closed principle** states "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*".
[Bertrand Meyer, 1988: Obj' Oriented Software Construction, 1st ed, Prentice Hall]

The rationales are complementary:

- Closed : closed for changes (in use for example).
- Open: for extension (enhancement/changeable) but without effecting clients

Principles of Modularity

6. The Single Choice Principle: Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list.

- Editor: set of commands (insert, delete etc.)
- Graphics system: set of figure types (rectangle, circle etc.)
- Compiler: set of language constructs (instruction, loop, expression etc.)

Encapsulation languages

- Basic idea: gather a group of routines serving a related oo-purpose, such as *has*, *insert*, *remove* etc., together with the appropriate data structure descriptions.
- Advantages:
  – For the supplier: Get everything under one roof. Simplifies configuration management, change of implementation, addition of new primitives.
  – For the client: Find everything at one place. Simplifies search for existing routines, requests for extensions.

Why use objects?

- Reusability: Need to reuse whole data structures, not just operations
- Extendibility, Continuity: Objects remain more stable over time.

Object technology: A first definition

- Object-oriented software construction is the approach to system structuring that bases the architecture of software systems on the types of objects they manipulate — not on “the” function they achieve.

Issues of object-oriented design

- How to find the object types.
- How to describe the object types.
- How to describe the relations and commonalities between object types.
- How to use object types to structure programs.

Description of objects

- Consider not a single object but a type of objects with similar properties.
- Define each type of objects not by the objects' physical representation but by their behavior: the services (FEATURES) they offer to the rest of the world.

- External, not internal view i.e. ABSTRACT DATA TYPES

The theoretical basis

- The main issue: How to describe program objects (data structures):
  - Completely
  - Unambiguously
  - Without overspecifying?
    - (Remember information hiding)

Stack: An abstract data type

- Types:
  - *STACK* [*G*]

    -- *G*: Formal generic parameter
- Functions (Operations):
  - *put*: *STACK* [*G*] $\times$ *G* $\rightarrow$ *STACK* [*G*]
  - *remove*: *STACK* [*G*] $\rightarrow$ *STACK* [*G*]
  - *item*: *STACK* [*G*] $\rightarrow$ *G* /
  - *empty*: *STACK* [*G*] $\nrightarrow$ *BOOLEAN*
  - *new*: *STACK* [*G*]

Object technology: More precise definition

- Object-oriented software construction is the construction of software systems as structured collections of (possibly partial) abstract data type implementations.

Classes: The fundamental structure

- Merging of the notions of module and type:
  - Module = Unit of decomposition: set of services
  - Type = Description of a set of run-time objects ("instances" of the type)

- The connection:
  - The services offered by the class, viewed as a module, are the operations available on the instances of the class, viewed as a type.

Classes: The fundamental structure

- From the module viewpoint:
  - Set of available services ("features").
  - Information hiding.
  - Classes may be clients of each other.

- From the type viewpoint:
  - Describes a set of run-time objects (the instances of the class).
  - Used to declare entities (≈ variables), e.g.

    C x;

  - Possible type checking.
  - Notion of subtype.

- The run-time structures, some of them corresponding to “objects” of the modeled system, are objects.

- The software modules, each built around a *type* of objects, are classes.

- A system does not contain any “objects” (although its execution will create objects).

From procedural to object-oriented languages

In C and other procedural programming languages:

- programming is action oriented
- unit of programming is the function (or procedure)
- data exists to support the actions that the functions perform
- dynamically, functions are called

In C++, Java, C# and other OO languages:

- programming is object oriented
- unit of programming is the class (a user-defined type)
- data and functions are contained within a class
- dynamically, objects are created (and used through their methods)

Abstract Data Types

- We need to define
  - The **abstract** 'thing' we are trying to represent in our programs
  - The **data representing the state** of that 'thing'
  - The **behaviour** of that 'thing'
- The 'thing' we are trying to define is not the object, but a class of objects.
- The class defines the common attributes and methods for all objects of the class.

From Structures to Objects

Classes in C++ are a natural evolution of the C notion of **struct**

A typical structure example:

```
struct Time {
int hour;  // 0-23
int minute; // 0-59
int second; // 0-59
};
```

By exposing weaknesses of this struct example (which are typical of all structures) we will motivate the evolution towards classes

Using Structures with a function… a problem?

We can now write the following:

```
Time t1;
t.hour = 66; t.minute = 78;  t.second = 100;
print (t1);
Time t2;
print (t2);
```

Here, t1 is initialised incorrectly and t2 is not initialised at all.

Also, if the implementation of the structure is changed (for example, if we just count seconds since midnight and calculate hour and minute values) then the print function (and all other functions acting on Time) must be changed accordingly… in fact, all programs using Time must be changed!

Using Structures … more (C) problems

In C (and C++), structures are treated member-by-member:

- •structures cannot be printed as a unit
- •structures cannot be compared in their entirety
- •structures cannot have operators overloaded (like =, <, +, <<)

Moving from C structures to C++ classes is easy:

- •classes and structures can be used almost identically in C++
- •difference between the two is in accessibility to members

C++ /C#/Java Syntax for classes:

- **class** defines an ADT
- **private** defines the hidden part
- **public** defines the visible interface of the class

Attributes: the elements which make up an objects state. In the class definition we are only concerned with an attributes name and type.

Methods: the functions to change the **attributes** values/state. In the class definition we are concerned with the methods name, type and function definition

Its all about contracts

- **Objects**
  - **Association of data with it’s operations**
  - An integer has a set of associated operations
- **The Black Box**
  - No need to care about it’s internal mechanics
  - **Hiding an entities constituent data**
  - The data is irrelevant to the outside
- **Communication via contract**
  - A specification of what it does

Encapsulation

- Animating data
- Hiding Data from the client
- Moving away from manipulation of data by the user
- Interaction through an interface
- Communication between objects via messages
- Client Server approach to software components

Inheritance

- Re-use of existing objects
- Sharing similar behaviour
- New entities using facilities provided by existing ones
- Hierarchical approach

Polymorphism

Polymorphism

- Lets leave this one until later …

Objects

- Extension to user-defined types
- Binds two familiar concepts
- Organising data into capsules
- Manipulation of data with functions
- Fusion being the difference

Classes as an ADT implementation

Classes let the programmer model objects which have:

- attributes (represented as data members)
- behaviour (operations represented as member functions/methods)

Types containing data members and member functions are defined using the keyword **class** e.g. in C++

Consider **Time** as a class:

```
class Time {
private:
int hour;  // 0 -23
int minute; // 0-59
int second; // 0 -59
public:
Time ();
void setTime(int, int, int);
void print();
}
```

Class Time continued ...

```
class Time {
public:
Time ();
void setTime(int, int, int);
void print()
private:
int hour;  // 0 -23
int minute; // 0-59
int second; // 0 -59
};
```

•**public:** and **private:** are member access specifiers (ignore protected, for now)

•**public** --- member is accessible wherever the program has a **Time** object

•**private** --- accessible only to methods of the **Time** class

•the class definition contains *prototypes* for the 3 public member functions.

•the function with the same name as the class is a *constructor*

Constructors

Constructors are fundamental to all OO programming:

- A special function used to initialise data members
- Called automatically when an object of the class is created
- Common to have several constructors for a class (overloading)
- Most OO languages have default constructors
- Things get complicated when:
    - member objects of a class are themselves objects!
    - we inherit behaviour …
    - we have recursive data structures in classes

Time Constructor

Constructors can guarantee *valid* initialisation:

```
Time::Time() {hour = minute = second =0;};
```

Methods permit state changes:

```
void Time::setTime (int h, int m, int s){
hour = h;
minute = m;
second = s;
};
```

NOTE: member functions can be defined inside a class, but it is good practice to do it outside the class definition

NOTE: The C++ scope resolution operator (**::**)

Time Constructor

Constructors can guarantee *valid* initialisation:

```
Time::Time() {hour = minute = second =0;)
```

Methods can guarantee *valid* state changes:

```
void Time::setTime (int h, int m, int s){
hour = (h>=0 && h<24) ?h:0;
minute = (m>=0 && m<60) ? m:0;
second = (s>=0 && s<60) ? s:0; }
```

Maintaining these types of *invariant property* is a good programming style

Can we do better with our setTime definition?

How can I use a class?

In C++ we define the class C in a header file C.h, define the methods belonging to that class and write a main function which creates instances of the C class.

Objects are instances of classes

To instantiate an object of the C class

```
void main(){
   C object_c;
}
```

What do we do in Java and C#?

Using Pointers with Objects

- As with structs pointers may be used to reference dynamic objects in C++

```
void main()
{
 Time *ptime = new Time;
}
```

Accessing class members

- **By default class members cannot be accessed**
- For the above Time class the following would be illegal

```
void main()
{
 Time Time1;
 Time1.hour = 0;
}
```

- The rules of encapsulation do not allow member variables to be accessed directly from outside an object
- It is possible though to allow members to be accessed from outside an object.
- Access modifiers: Private, Public, Protected

Private v Public

- The private access modifier states that the members following it are visible only within that class (This is the default)

- The public access modifier states that the members following it are visible both inside and outside that class

- Making attributes visible breaks encapsulation

A Better Solution

```
class Time
{
private:
 int hour;
public:
 void setTime (int h, int m, int s);
}
```

Methods

- The Time example contains what appears to be a function prototype called setTime
- This is what is known as a Method or a member function
- A method cannot be used alone because it is associated with a class
- A method is referenced similarly to a member variable using ‘`.`’ or ‘`->`’ (Depending on whether an instance or pointer is used)

Calling Methods without Pointers

```
void main()
{   Time T1;
    T1.setTime(3,4,5);
}
```

Calling Methods with Pointers

```
void main()
{   Time *ptime = new Time;


    ptime->setTime(3,4,5);
    ptime->print();
}
```

Methods /Functions

- By referencing a method in this manner we are said to have invoked it
- A method is invoked on an Object
- Methods are like functions in many respects
  - Can return values
  - Can take parameters
- Methods differ from functions as the attributes of the class are accessible directly from the class – they don’t have to be passed into the function or declared within the function.

- Scope resolution operator i.e. **::** is used to link together classes and definitions of their methods.
- Attributes should be put into the private part of the class, where they can only be accessed via methods.

- Methods themselves appear in the public part of the class so that they may be accessed externally and provide an interface to the class.
- It is possible to put attributes into the public part of the class **but** this breaks the rules of encapsulation.
- It is also possible to put methods in the private part of the class. This is useful for methods which are used by other methods of the same class, but not appropriate as part of the external interface.

- Inline methods are declared and defined within the body of the class => duplicated for every object of the class.
- Usually methods are declared within a class and defined outside the class

Implementing Methods

- As with functions the implementation for methods is kept in a source file
- A method implementation declaration includes the class name and the method name separated by '::'

```
void Time::setTime (int h, int m, int s){
hour = h;
minute = m;
second = s; }
```

- All other class members are accessible directly from within a method
- The public methods are usually the interface to a class. They set up the contract between the class and its user.

An Example.

```
class Number
{
 private:
   int i;


   public:
          void setvalue (int value);
          int get_value();
};   // note ;
```

The Method Definitions

```
void Number :: setvalue(int value)
{
   i= value;
}  // note absence of ;

int Number::get_value()
{
   return i;
} // note absence of ;
```

- The public methods are the only things which can be used with an object of a class. There is no direct access to the private elements.

- Calling a method is known as *sending a message*.

- Where we define a class we store it in a header file e.g. Number.h.
- This file is included in program by

  #include “Number.h”
- Note < Number.h> = > look for file in the usual library directory.
- ***A class defines the types of data appropriate to the class (the attributes) and its set of allowed behaviours (the methods).***

Classes and Objects

- Class = “Object Factory”
  - i.e. classes allow us to create new objects of the class, each one of which follows an identical pattern of attributes and methods.

- These objects are declared within main.

Given the declaration of a class called Bank_Account stored in the file Bank.h, the following code allows us to use the class Bank_Account to

- create an object MyAccount
- credit the bank account with 20 pounds, using the method credit from the Bank_Account class.

```
#include ”Bank.h”
void main()
{ Bank_Account MyAccount;
 Bank_Account.credit(20);
}
```

Class v Object

- Classes:
  - Unique name, Attributes, Methods
- Object:
  - Identity, State, Behaviour
- Classes exist all the time when a program is running whereas Objects may be created and destroyed at run time.
- Any number of objects may be instantiated for a given class.

Initialising An Object

We have seen before that it is possible to initialise variables of an intrinsic type using assignment `int x = 0;`

- Because an object may be quite complex a mere assignment may not be possible
- To allow objects to be initialised a special method called a constructor is used

Constructors

- Used to allocate memory for an object. Constructors are automatically called when an object is created.
- An object constructor
    - takes the same name as the class
    - may have arguments
    - cannot return a value.

- Constructors are executed every time an Object of a particular class is created
- This method takes the same name as the class and does not return a value

```
class Time {
private:
int hour;  // 0 -23
int minute; // 0-59
int second; // 0 -59
public:
Time ();
void setTime(int, int, int);
void print();
}
```

The 3 types of Constructor

- The default constructor – takes no parameters, performs no processing other that the reservation of memory . The compiler always calls it if no user-defined constructor is available.
- User defined constructors – used to initialize an object when it is created. Over loading is possible.
- The copy constructor – a way of using assignment when creating new objects.
  - New_object object2 = object1
  - This instantiates object 2 as object 1 has been instantiated.

User Defined Constructor

For user defined constructor definitions

look as follows:

```
Bank_Account:: Bank_Account()
{   current_balance = 0
    // Initialize attributes here.}
```

In main an object will be set up as follows:

Bank_Account MyAccount();

giving a bank account with the current balance of 0.

User Defined Constructor

- It is also possible to pass parameters to a constructor.

```
Bank_Account:: Bank_Account(int x)
{
    current_balance = x
    // Initialize attributes here.
}
```

In main an object will be set up as follows:

```
Bank_Account MyAccount(100);
```

giving a bank account with the current balance of 100.

Class Declaration for the Time Class

In a C++ header file called Time.h:

```
class Time {
private:
int hour;  // 0 -23
int minute; // 0-59
int second; // 0 -59
public:
Time ();
void setTime(int, int, int);
void print();
}
```

In a File called Time.cpp define:

```
Time::Time() {hour = minute = second =0;};

void Time::setTime (int h, int m, int s){

hour = h;

minute = m;

second = s;

};

void Time::print (){

cout << “The time is” << h  << “:” << m << “:” << s <<endl;

};
```

Using the Time Class

```
#include “Time.h“
#include <iostream.h>
void main()
 {
    Time T;
    T.setTime(3,4,5); T.print();

   Time * PT1 = new Time;
   PT1->setTime(6,7,8);
   PT1 -> print();
 }
```

Destructors

A function with the same name as the class but preceded with a tilde character (~) is called the destructor

When not explicitly given (as in **Time**), the compiler provides a default

Destructors cannot take arguments … so are never overloaded

Destructors do termination housekeeping on each object before the memory is reclaimed by the system

These are very important when using dynamic data structures.

ADT and software architecture

- Abstract data types provide an ideal basis for modularizing software.
- Identify every module with an implementation of an abstract data type, i.e. the description of a set of objects with a common interface.
- The interface is defined by a set of operations (implementing the functions of the ADT) constrained by abstract properties (the axioms and preconditions).
- The module consists of a representation for the abstract data type and an implementation for each of the operations. Auxiliary operations may also be included.

Terminology

- A class is an implementation of an abstract data type.
  - Instances of the class may be created at run-time; they are objects.
  - Every object is an instance of a class. (In a pure O-O language such as Eiffel and Smalltalk this is true even of basic objects such as integers etc. Not true in C++ or Java where such values have special status.)
  - A class is characterized by features. Features comprise attributes (representing data fields of instances of the class) and routines (operations on instances).
  - Routines are subdivided into procedures (effect on the instance, no result) and functions (result, normally no effect).
  - Every operation (routine or attribute call) is relative to a distinguished object, the current instance of the class.

Extra Terminology

- Attributes are also called instance variables or data member.
- Routines are also called methods, subprograms, or subroutines.
- Feature call — applying a certain feature of a class to an instance of that class — is also called passing a message to that object.
- The notion of feature is provides a single term to cover both attributes and routines.

C++/Java

- User1.cc
- User2.cc
- User3.cc
- User.java

04

More on Object Destruction

C++ /Java

Object Construction

C++

```
class User { ... }

User u (...);
// u is a User object

User *p = new User(...);
/* p can only hold a memory
   address to a User object*/
```

Java

```
class User { ... }
User q  = new User(...);
/* q can only hold a reference
   to a User object (a reference
   is like a disguised memory
   address – disguised in that
   you can’t deference it like
   you can dereference a C++
   pointer.*/
```

What is the difference between object references and pointers?

References

If the JVM decides to move the User object to a different place in memory for reasons of memory management, the object reference held by q would still be able to find the object.

Pointers

If the object pointed to by p was moved to some other memory location then, the value of p would need to be explicitly changed under program control.

When do objects go out of Scope?

- In C++ they are automatically destroyed by the invocation of their destructors
- Destructor methods
  - Have the same name as the class preceded by ~
  - Cannot take arguments
  - Cannot return a value
- If a destructor is not supplied then the default meaning of the destructor is invoked for fields of the class. This simply means that the memory that they occupy becomes free.

Example: C++

```
class GameScore {
private:
    int homeTeamScore;
    int otherTeamScore;
public:
    GameScore(int score1, int score2)
    {       homeTeamScore = score1;
            otherTeamScore = score2;
    }
    ~ GameScore()
    {       cout << "GameScore object destroyed " <<       homeTeamScore << "
    vs. " << otherTeamScore                <<endl;
    }
};
```

Example:

```
class GameScore {
private:
    int homeTeamScore;
    int otherTeamScore;
public:
    GameScore(int score1, int score2)
    {       homeTeamScore = score1;
            otherTeamScore = score2;
    }
    ~GameScore()
    {       cout << "GameScore object destroyed " <<       homeTeamScore << "
    vs. " << otherTeamScore
            << endl;
    }
};
```

```
int main() {
 GameScore gs1(28,3);
 GameScore gs2(35,7);
 return 0;
}
```

Example:

```
GameScore object destroyed 35 vs. 7

GameScore object destroyed 28 vs. 3
```

```
int main() {
 GameScore gs1(28,3);
 GameScore gs2(35,7);
 return 0;
}
```

```
        ~GameScore()
        {       cout << "GameScore object destroyed " <<       homeTeamScore << " vs. " << otherTeamScore
                << endl;
        }
    };
```

Example:

```
class GameScore {
private:
	int homeTeamScore;
	int otherTeamScore;
public:
	GameScore(int score1, int score2)
	{	homeTeamScore = score1;
		otherTeamScore = score2;
	}
	~GameScore()
	{	cout << "GameScore object	destroyed " <<		homeTeamScore << " vs. "
	<< otherTeamScore
		<< endl;
	}
};
```

```
int main() {
 GameScore gs1(28,3);
 GameScore gs2(35,7);
 GameScore*p;
 p = new GameScore(29,0);
 return 0;
}
```

Example:

```
GameScore object destroyed 35 vs. 7

GameScore object destroyed 28 vs. 3
```

```
int main() {
 GameScore gs1(28,3);
 GameScore gs2(35,7);
 GameScore*p;
 p = new GameScore(29,0);
 return 0;
}
```

```
}          cout << "GameScore object          destroyed " <<          homeTeamScore << " vs. " <<
```

The memory where gs1, gs2 and p is stored is freed at }
What is stored at p is just the address of the 3rd **GameScore** object
This object is not destroyed – its destructor is not automatically called.
We need to use delete.

Example:

```
GameScore object destroyed 29 vs. 0

GameScore object destroyed 35 vs. 7


GameScore object destroyed 28 vs. 3
```

```
int main() {
 GameScore gs1(28,3);
 GameScore gs2(35,7);
 GameScore*p;
 p = new GameScore(29,0);
 delete p;
 return 0;
}
```

```
        }         cout << "GameScore object          destroyed " <<              homeTeamScore << " vs. " <<
```

The memory where gs1, gs2 and p is stored is freed at }
What is stored at p is just the address of the 3rd **GameScore** object
This object is not destroyed – its destructor is not automatically called.
We need to use delete.

Example: C++ Destructor needed

```
class X
{
  private:
              int* ptrToArray;
              int size;

   public:
              X( int* ptr, int sz ):size(sz)
              {
                          ptrToArray = new int[size];
                          for ( int i = 0; i < size; i++ )
                                      ptrToArray[i] = ptr[i];
               }
               ~X()
               {
                          cout << "hello from the destructor"
                          <<          endl;
                          delete [] ptrToArray;
               }
};
```

Example: C++ Destructor needed

```
class X
{
  private:
		int* ptrToArray;
		int size;

  public:
		X( int* ptr, int sz ):size(sz)
		{
			ptrToArray = new int[size];
			for ( int i = 0; i < size; i++ )
				ptrToArray[i] = ptr[i];
		}
		~X()
		{
			cout << "hello from the destructor"
			<<		endl;
			delete [] ptrToArray;
		}
};
```

Acquire memory for the array

Copy the argument ptr into the newly acquired memory

Release the memory that was acquired by the constructor

Example: C++ Destructor needed

```
int main()
{
    int freshData[100] = {0};
    X xobj( freshData, 100 );
    X* p = new X(freshData, 100 );
    delete p;
    return 0;
}
```

What’s the output?

Example: C++ Destructor needed

```
int main()
{
    int data[100] = {0};
    X xobj(data, 100 );
    X* p = new X(data, 100 );
    delete p;
    return 0;
}
```

```
Hello from the destructor
Hello from the destructor
```

X destroyed and the pointer p destroyed

When do objects go out of Scope?

- In Java, we cannot destroy a particular object at a particular time – we can request that unreferenced objects are destroyed and their memory reclaimed.
- A garbage collector (`gc`) runs in the background, but before destroying an unreferenced object the `gc` calls the classes finalize method.
- This method in every class automatically and can be redefined (overridden) for our own purposes e.g. closing open files I/O connections before an object is permanently destroyed.
- The `gc` carries no guarantees about the order of finalizing objects or reclaiming memory

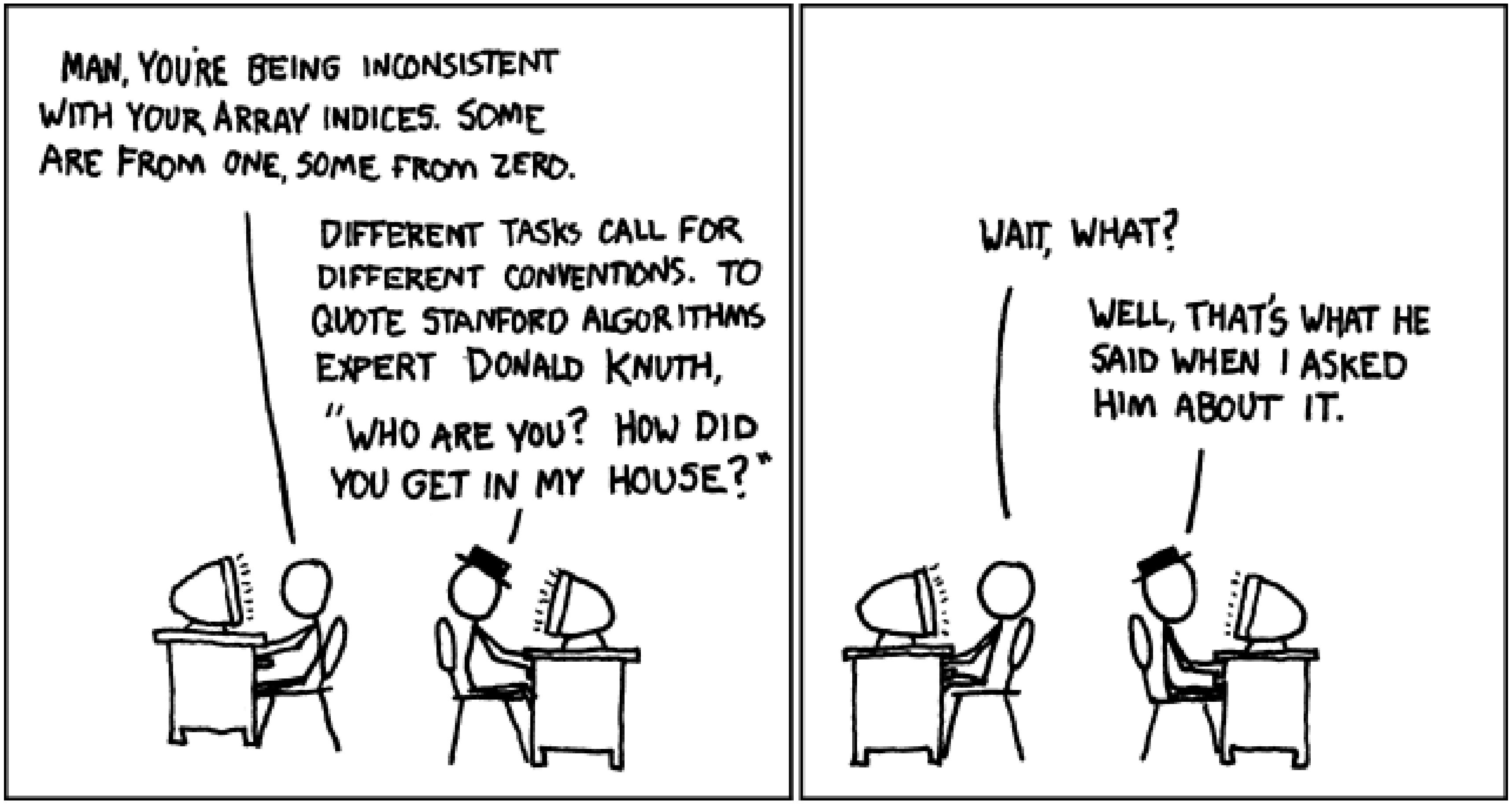
GC.java

- Objects of type x are given an id: the first object having the id 1 (for example), the second having the id 2 and so on.
- In main() we create an array of 10,000 objects of type X with id’s 1 … 10,000.
- Xarray = null sets the array reference to null suddenly causing the array to be unreferenced. It also causes the 10,000 objects to become unreferenced and hence a target for GC.
- System.gc(); explicitly calls the gc to reclaim memory occupied by these objects
- When control returns from System.gc() this means the JVM has reclaimed what it can.

Executing this program

- Open GC.java using JCreator or a similar tool
- Execution of a java file will generate output which differs from run to run as the unreferenced objects may be finalised and discarded in any order
- (Note that in our example gc.java just prints the id if objects involved have an id that is a multiple of 1000).
- The JVM may quit before the **gc** finalizes and collects all the objects

GC.Java


MAN, YOU'RE BEING INCONSISTENT WITH YOUR ARRAY INDICES. SOME ARE FROM ONE, SOME FROM ZERO.
DIFFERENT TASKS CALL FOR DIFFERENT CONVENTIONS. TO QUOTE STANFORD ALGORITHMS EXPERT DONALD KNUTH, "WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"
WAIT, WHAT?
WELL, THAT'S WHAT HE SAID WHEN I ASKED HIM ABOUT IT.

05

Object Lifetimes

Types of Object

External (Global) Objects

- Exist through the program lifetime
- Have file scope

Automatic (Local) Objects

- Exist throughout the local scope in which they are created

Static Objects

- Exist through the program lifetime
- Only visible within local scope

Dynamic Objects

- Lifetime controlled within a particular scope

External (Global) Objects

In C++…

```
BankAccount acc;
void main()
{…
}
```

The declaration of objects is all that we can do outside the scope of {}

We cannot call methods of acc outside {}

External Linkage

In a system containing several program modules, such objects may be declared as extern in other modules if their visibility needs to extend beyond the file scope in which they are defined.

Prog1.ccp	Prog2.ccp	Prog3.ccp
Manager Joe;	extern Manager Joe;	extern Manager Joe;
void main()	…	…
{ …		
}		

Automatic (Local) Objects

In C++…

- Declared inside the body of main.
- Visible anywhere inside main and persists until main finishes.

Example:

```
void main()
{
   BankAccount account1;
   …
}
```

Static Objects

A static object is declared, like an automatic object, within a local scope. However it is only initialised once, regardless of the number of times it falls in and out of scope.

In C++

```
void  function()
{ BankAccount account1;
  static BankAccount account2;
}
```

Automatic vs. Static

Both objects (account1 & account2) come into scope when the function is called but

- the automatic object will be created and destroyed each time
- the static object will be instantiated once and will persist to the end of the program.

Dynamic Objects

- If objects are unpredictable they must be represented dynamically.
- We cannot give dynamic objects unique names so we must reference them using pointers.
- In C++ a pointer can be directed to an area of dynamically allocated memory at run time in order to reference a newly created object. In this way the constructor is called via a pointer.

Constructors & Destructors

Constructor: called when an object is created, the default constructor is called if the user doesn't define one.

Destructor: used to destroy the object by freeing up the memory used by the object (clean up).

With dynamic objects there is no default destructor – the user must call it explicitly.

Destructors in C++

Destructor methods

- Have the same name as the class preceded by ~
- Cannot take arguments
- Cannot return a value

Creating dynamic objects

```
classname *pclassname = new classname;
e.g. counter *pcounter = new counter;
```

pcounter is not the name of an object but the name of a pointer to an object of type counter.

Dynamic object do not have names – they simply occupy a memory space which may be referenced by a pointer.

Calling Methods of Dynamic Objects

```
pclassname -> method(parameters);
e.g. pcounter -> increment();
```

In order to call a method of a dynamic object, the pointer must be dereferenced before the objects methods may be accessed.

The '-> ' operator must be used instead of '.'

Clean up

delete pclassname;

delete pcounter;

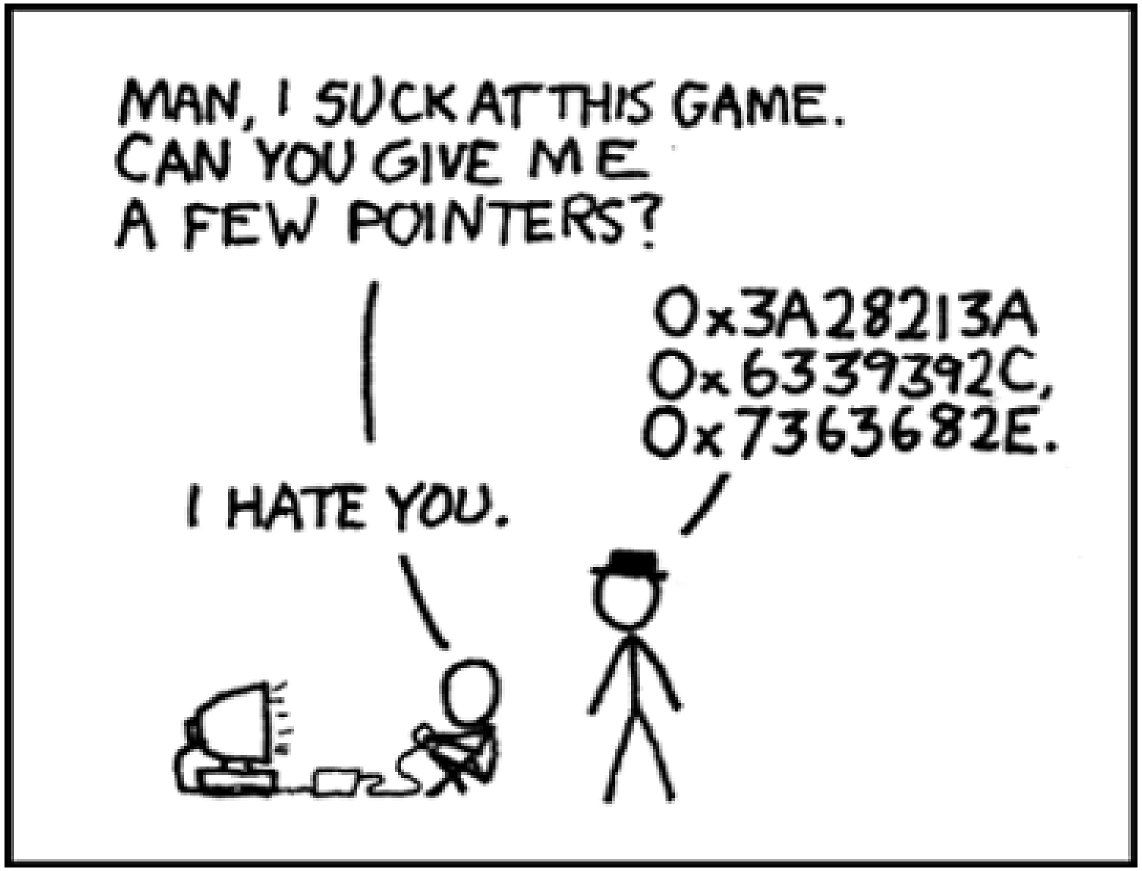
- delete cleans up memory by explicitly calling the destructor which destroys the object currently referenced by the pointer.
- delete destroys the object NOT the pointer.
- The pointer is still available to point to other objects of the class.

Null Pointers

A pointer which is not pointing to an object may be pointing to any random area of memory. These pointers should be directed to ‘**NULL**’.

NULL is a constant which is declared in standard header files stddef.h, stdlib.h.

Pointers


MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

Objects in C++
vs.
Objects in Java

C++ Objects

```
class User{
private:
  string name;
  int age;

public:
  User(string nam, int a) {
    name = nam;
    age = a;
   }
};
```

```
int main()
{  User u1(“Ann”, 8);
   User *u2 = new User(“Jane”, 12);
   User &u3 = u1;
   const User &u4 = User(“Amy”, 13);
   cout <<  u1.name << endl;   //Ann
   cout <<  u2->name << endl;  //Jane
   cout <<  u3.name << endl;   //Ann
   cout <<  u4.name << endl;   //Amy

   User *p = &u1; User *q = &u3;
   cout <<  p->name << endl;   //Amy
   cout <<  q->name << endl;   //Amy
}
```

Java Objects

```
class User {
    private String name;
    private int age;

    public User( String nam; int a )
    {
     name = nam; age = a; }
    }
}
```

- Declaring `User u = new User(“Marie, 10);` means that `u` holds a reference to an Object of type User.

Memory alloc/dealloc in C++

```
string *str = new string(buffer);
delete str;  str = null;


string *str = new string(“Hello”);
delete str; str = null;

int * p = new int[100]
delete [] p;      //to delete an array!
```

Memory alloc/dealloc in Java

```
User  u = new User ("Tom", 7);
u = null;
//marks the object created above for collection
//  by the garbage collector

User [] UserArray = new User[100];
UserArray = new User[20];
//marks the object created above for automatic
//collection by the garbage collector allowing
//the memory occupied by them to be freed up.
```

The Metaclass

The MetaClass

- The class holds the attributes and methods which apply to objects of the class – it is the class of the objects. A class acts as a kind of blueprint or skeleton for objects of the class.
- The metaclass holds the attributes and methods which apply to the class itself – it is therefore the class of the class.
- Every class has one (and only one) metaclass, and the metaclass contains those parts of a class which are not appropriate to be duplicated for every object.

Class Attributes

Sometimes we may want to have a physical representation of something about a whole class of objects rather that about one object in particular – suppose we want to know how many objects of a class exist at one particular time – where do we store this information?

Store it in the class?

If we store this attribute in each object it gets duplicated for every object that we create. To keep the value of that attribute correct we would have to change its value for **every** object each time we create/destroy an object – this would be unnecessary duplication of the same data which could lead to inconsistencies.

The Metaclass

- To solve this problem we need a single attribute for the whole class.
- We could have a simple counting variable outside the class itself but this would break the rules of encapsulation by making the attribute a global variable accessible from outside the class.
- We cannot put the attribute in the class as this would make it an attribute for every instance of that class.
- The answer is to put the attribute in the metaclass.

A Metaclass is …?

The metaclass is a repository for class attributes which only have one instance regardless of how many objects of the class exist. Class attributes tell us about the class as a whole as opposed to objects of the class.

Class attributes are just as visible as object attributes to an object. However, the value of object attributes (an objects state) remain invisible to the class.

Class Methods

- Class methods also exist in the metaclass.
- These methods are used to access the class attributes. Like attributes, class methods are just as visible as object methods to an object.

Therefore, it is possible to access the value of a class attribute using an object’s method.

Class Methods

- What if there are no objects of the class instantiated and we want to return the value of the class attributes?
- If an object is not created we cannot use it to access the class attributes. Hence, we need another way of accessing the class attributes – class methods.
- Class methods may be called by the class directly as well as by an object.

(Note :

Default constructors and destructors belong to the metaclass)

```
#include <string.h>
class BankAccount
{
private:
// object attributes
  int account_number;
  char account_holder[20];
  float current_balance;
// a class attribute (static) to count
// the number of all bank accounts
  static int account_total;
public:
  BankAccount(float start_balance = 0.00);
  int getAccountNumber();
  char* getAccountHolder();
  float getCurrentBalance();
  void setAccountNumber(int number_in);
  void setAccountHolder(char* holder_in);
  void deposit(float amount);
  void withdrawal(float amount);
// a class method (static) to return the
// number of all bank accounts
  static int getAccountTotal();
};
```

```
BankAccount::BankAccount(float
  start_balance)
{
  current_balance = start_balance;
// increment the class attribute
//when an object is constructed
  account_total++;
}
```

```
// the definition of the class method
int BankAccount::getAccountTotal()
{
  return account_total;
}

// reserve storage for the class
// attribute
int BankAccount :: account_total = 0;
```

Using Metaclass Attributes

```
#include <iostream.h>
#include ”BankAccount.h"
void main()
{
// create a number of bank accounts
  BankAccount acc1, acc2, acc3, acc4, acc5;
// test the class method
  cout << "Total number of accounts is: " <<
  BankAccount::getAccountTotal() << endl;
}
```

06

Inheritance

What is Inheritance?

- Inheritance enables classes to inherit attributes and methods from other classes. It allows the sharing of common elements between classes without having to repeat definitions for each class.

Inheritance occurs between

***classes (not** Objects!)*

Derived & Base Classes (1)

- A class does not contain state values – it only defines the methods and attributes which may be used in a class. State values are contained in individual objects.
- A **derived** class inherits the attributes and methods of a **base** class.
- We may build on the derived class by adding attributes and methods.

Derived & Base Classes (2)

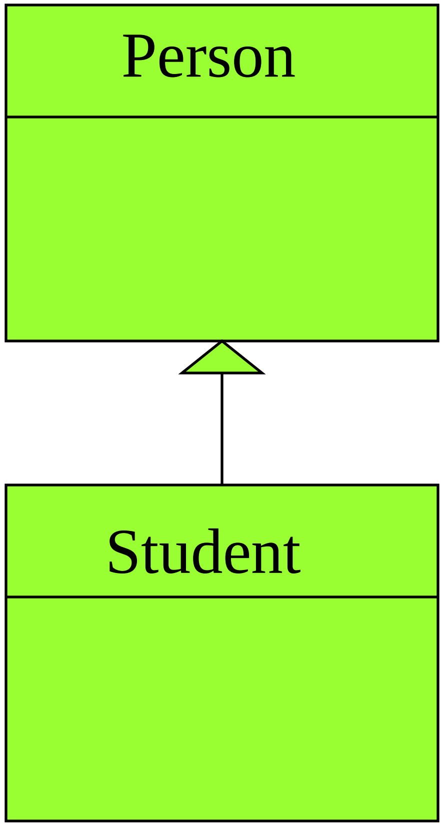
- **Base class:** a class form which another class inherits.

  Base classes don’t have to represent anything concrete which could be instantiated as an object of its own.

- **Derived class objects** do not inherit any state values from base class objects.

  The derived class may be extended without effecting the original class.

public inheritance



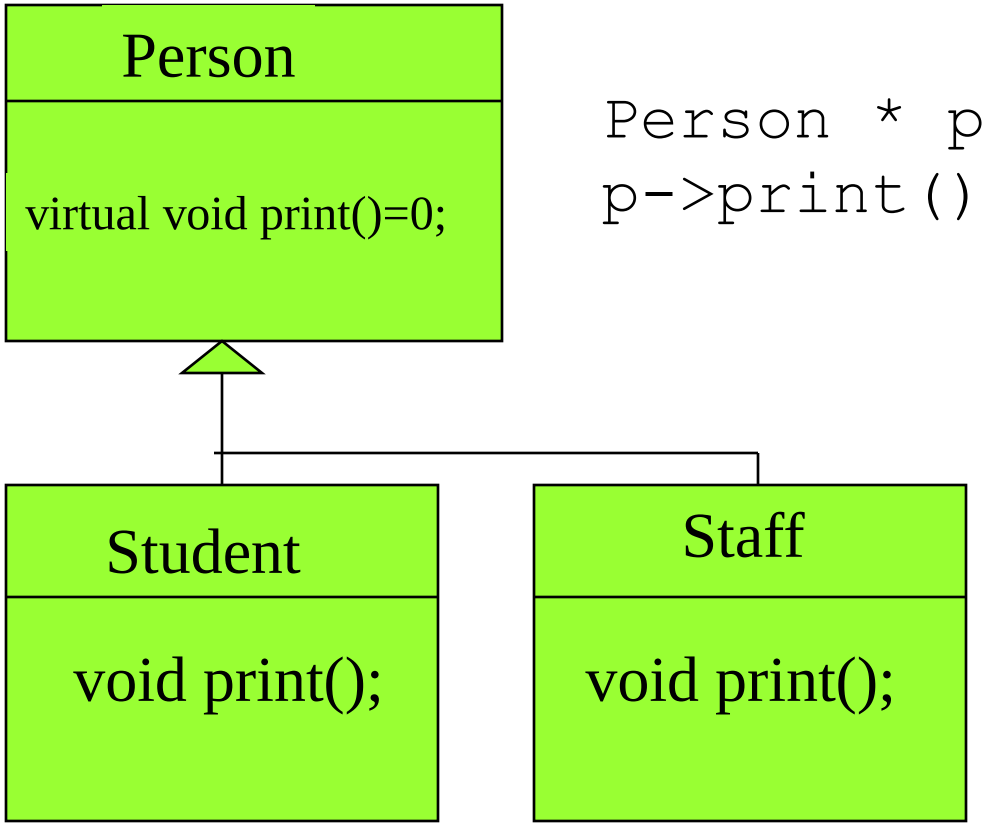
Every student is a person, but not every person is a student.

C++ compiler enforces this!

“IS-A”

```
class Person {
   void play();
};

class Student : public Person {
   void study();
};

Person p;
Student s;
p.play();           // ok
s.play();           // ok
s.study();          // ok
p.study();          // error!
```

Typical design



```
Person * p = new Student("Hank
p->print();
```

Passing parameters to base class constructor

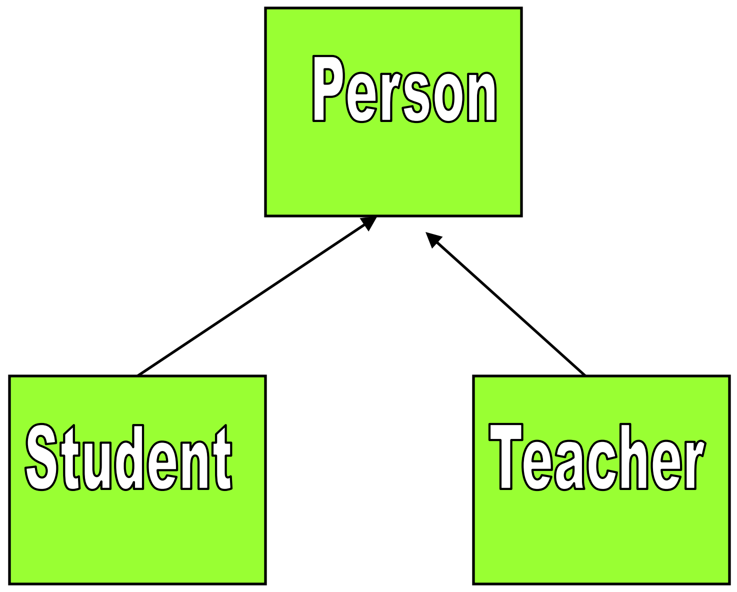
```
class Person {
public:
   Person(const string & n) : name(n) {}
private:
   string name;
};

class Student {
public:
   Student(const string & n, float g) : Person(n),
                                        gpa(g) {}
private:
   float gpa;
};
```

Typical uses of inheritance

- Single inheritance – one base class
- attributes that describe all classes go in the base class
- attributes that are “special” go in derived class

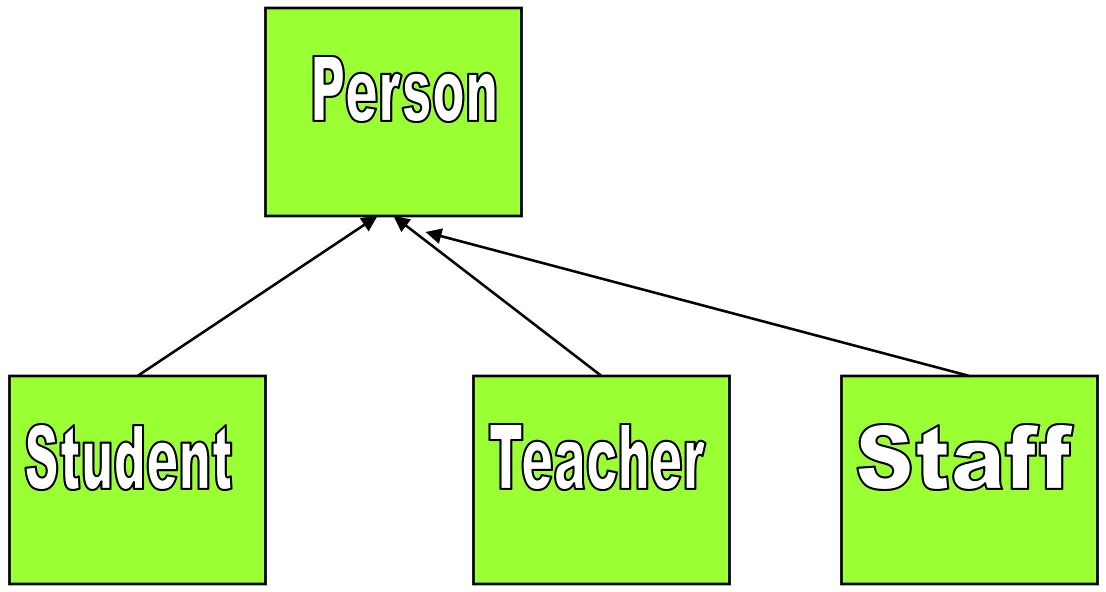
Software Reuse

Derived classes reuse attributes in the base class.



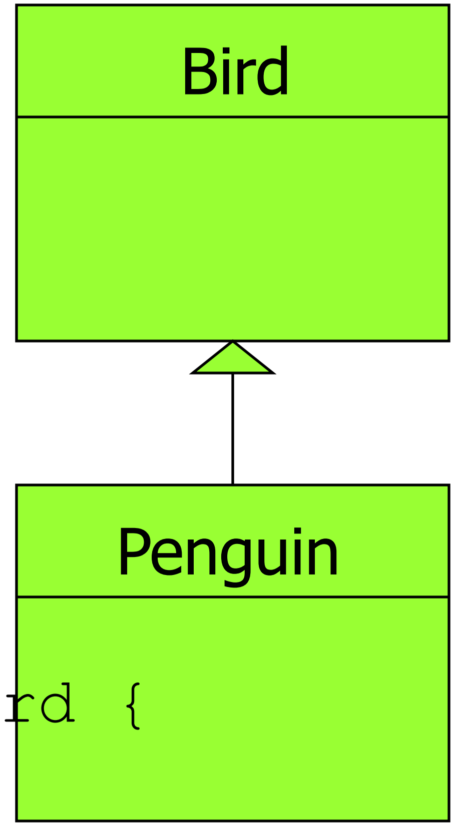
Easy to extend

Can add a new derived class, extending the framework, without changing any existing code: extensibility



“IS-A” doesn’t always fit!

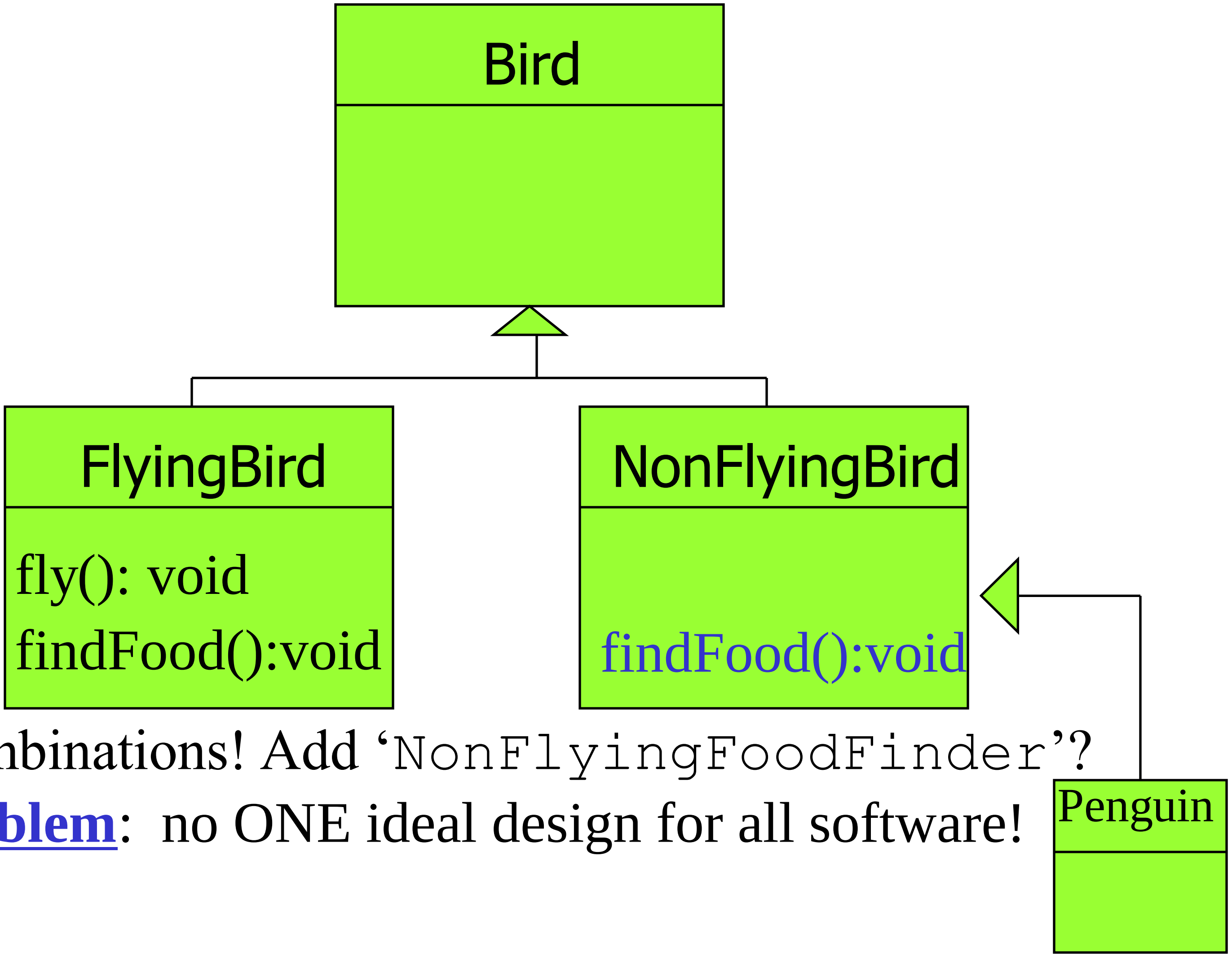
- fact: birds can fly
- fact: a penguin is a bird



```
class Bird {
    virtual void fly();
};

class Penguin : public Bird {
};
```

What’s wrong? (“IS-LIKE-A”)

Inheritance Hierarchies: Problem!



Combinations! Add ‘`NonFlyingFoodFinder`’?

Problem: no ONE ideal design for all software!

Second Approach

Redefine "fly" so that it generates a runtime error:

```
void error(const string & msg);

class Penguin : public Bird {
public:
   virtual void fly() { error("Penguins can't fly!");

};
```

Better to detect errors at compile time, rather than at runtime.

An Introduction to Inheritance

- Inheritance: extend classes by adding methods and fields (*a generalization-specialization relationship*)
- Example:

Savings account = bank account with interest

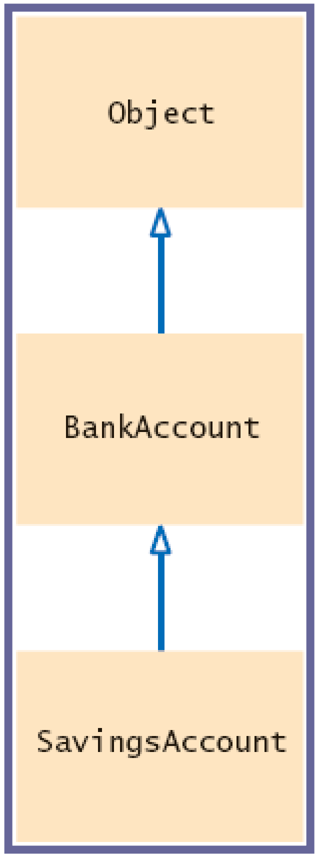
```
class SavingsAccount:public BankAccount
{
   new methods
   new instance fields
}
```

An Introduction to Inheritance

- Inheriting from class ≠ implementing interface: subclass inherits behavior and state
- One advantage of inheritance is code reuse

An Inheritance Diagram

- Every class extends the Object class either directly or indirectly



An Inheritance Diagram

C++ Syntax for Inheritance

- In C++ ‘:’ denotes inheritance (also called specialisation or derivation or extension)
- public derivations and private derivations are possible.
- The protected keyword allows derived classes to access inherited attributes.

Public Inheritance in C++

```
class IntObj{
private:
  int val;
public:
  IntObj(){val =0;}
  int getint() const {return val;}
  void setint(int x){val = x;}}
  void print(){cout <<val<<endl;}
};
class SubintObj: public intObj{
//...
}
int main(){
  IntObj obj; obj.print();
  SubintObj Sobj;
  Sobj.print();
}
```

The semantics:

`SubintObj` is derived from (or a subclass of) `IntObj`

It is public meaning that the public members of the base class become public members of the derived class.

Question: what can we do with the derived class?

Public or Private Derivations?

- **Public derivations:**
  - Allow objects of a derived class access to the public section of the base class.

- **Private derivations:**
  - Allow a derived object to use the methods defined in the derived class, not those inherited from the base class.

Public Inheritance in C++

```
class IntObj{
private:
  int val;
public:
  IntObj(){val =0;}
  int getint() const {return val;}
  void setint(int x){val = x;}}
  void print(){cout <<val<<endl;}
};
class SubintObj: public intObj{
//...
}
int main(){
  IntObj obj; obj.print();
  SubintObj Sobj;
  Sobj.print();
}
```

Question: How can we make the derived class different from the base class?

Put something in its class definition.

Checking Account

- Given a `BankAccount` class with
  - Attribute: `float balance`
  - Constructor: `BankAccount()`
  - Methods: `deposit`, `withdraw`, `getBalance`

- `CheckingAccount` object inherits the `balance` instance field from `BankAccount`, and we can add an additional instance field: `transactionCount` and one method `deductFees()`

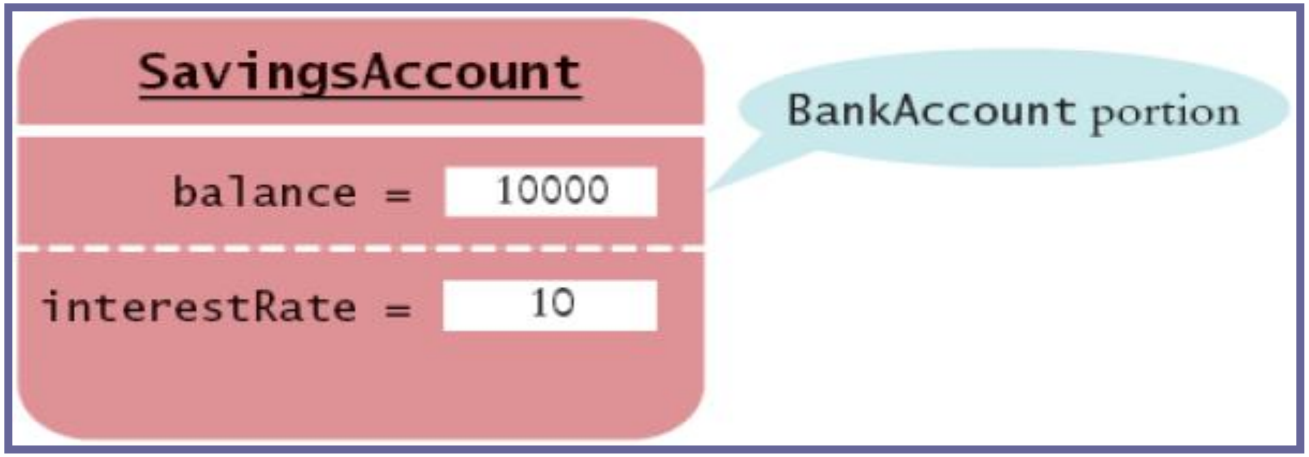
Savings Account

- Given a `BankAccount` class with
  - Attribute: `float balance`
  - Constructor: `BankAccount()`
  - Methods: `deposit`, `withdraw`, `getBalance`

- `SavingsAccount` object inherits the `balance` instance field from `BankAccount`, and gains one additional instance field: `interestRate` and one method `addInterest()`

Layout of a Subclass Object

- `SavingsAccount` object inherits the `balance` instance field from `BankAccount`, and gains one additional instance field: `interestRate`:



Layout of a Subclass Object

“Derived” (or “Sub”) class

- In the subclass, we specify added instance fields, added methods, and changed or overridden methods

Instance Fields & Methods

An object of class `SavingsAccount` has two instance fields: `balance` and `interestRate`.

There are four methods that you can apply to `SavingsAccount` objects: `deposit`, `withdraw`, `getBalance`, and `addInterest`.

Inheritance Hierarchies

- Sets of classes can form complex inheritance hierarchies
- Example:

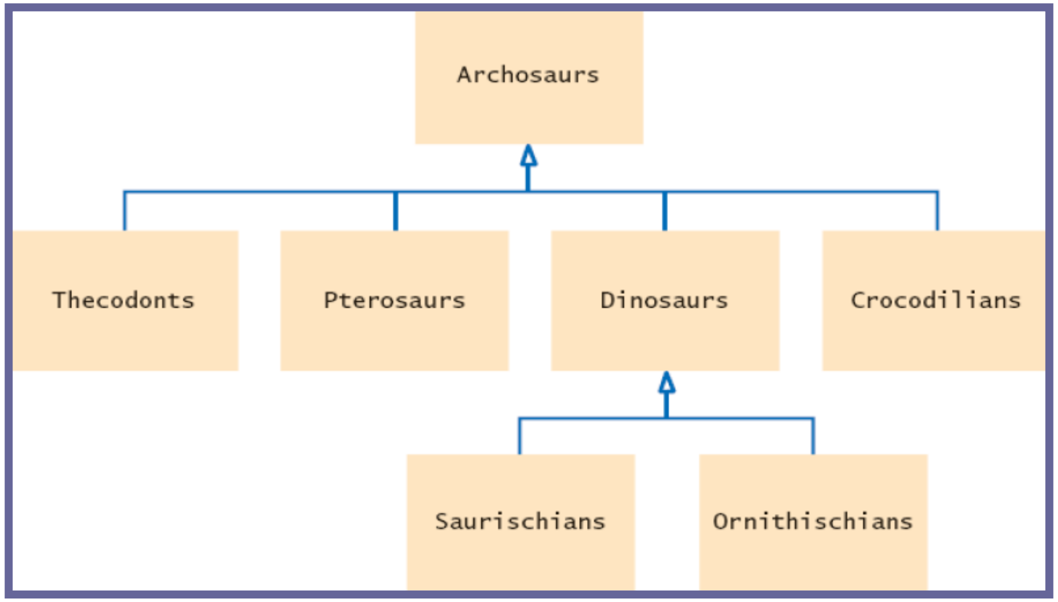


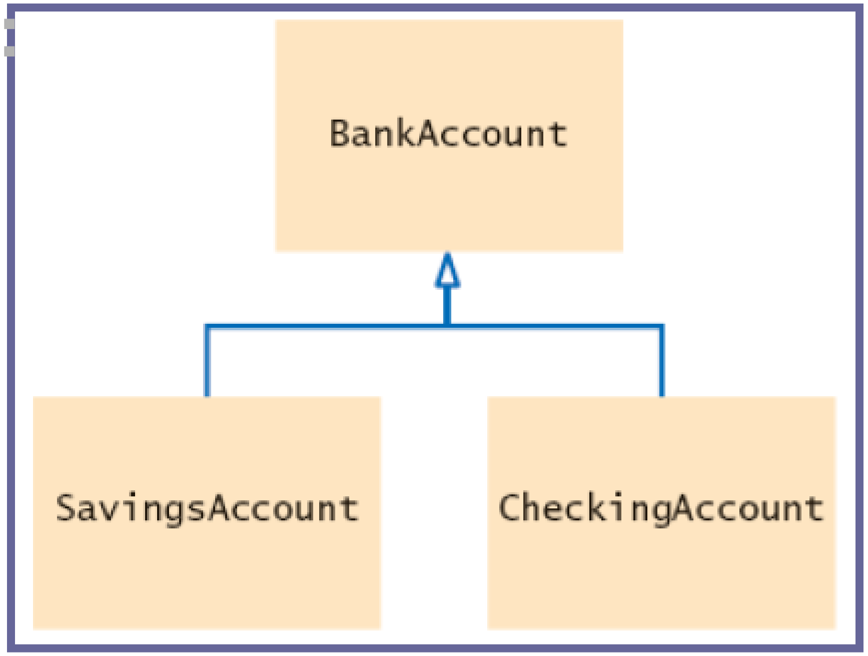
Figure 3:
A Part of the Hierarchy of Ancient Reptiles

A Simpler Hierarchy: Hierarchy of Bank Accounts

- Consider a bank that offers its customers the following account types:
  1. Checking account: no interest; small number of free transactions per month, additional transactions are charged a small fee
  2. Savings account: earns interest

A Simpler Hierarchy: Hierarchy of Bank Accounts

- Inheritance hierarchy:



Inheritance Hierarchy for Bank Account Classes

A Simpler Hierarchy: Hierarchy of Bank Accounts

- All bank accounts support the `getBalance` method
- All bank accounts support the `deposit` and `withdraw` methods, but the implementations differ
- Checking account needs a method `deductFees`; savings account needs a method `addInterest`

Inheriting Methods

- Override method:
  - Supply a different implementation of a method that exists in the superclass
  - Must have same signature (same name and same parameter types)
  - If method is applied to an object of the subclass type, the overriding method is executed
- Inherit method:
  - Don't supply a new implementation of a method that exists in superclass

Inheriting Methods

- Add a method:
  - Supply a new method that doesn't exist in the superclass
  - New method can be applied only to subclass objects

Inheriting Instance Fields

- Can't override fields
- Inherit a field: All fields from the superclass are automatically inherited
- Add a field: Supply a new field that doesn't exist in the superclass

Inheriting Instance Fields

- What if you define a new field with the same name as a superclass field?
  - Each object would have two instance fields of the same name
  - Fields can hold different values
  - Legal but extremely undesirable

Implementing the `CheckingAccount` Class

- Overrides deposit and withdraw to increment the transaction count:

```
class CheckingAccount:public BankAccount
{
   public:    void deposit(double amount) {. . .}
              void withdraw(double amount) {. . .}
              void deductFees() {. . .} // new method
   private:   int transactionCount; // new instance field
}
```

Implementing the `CheckingAccount` Class

- Each `CheckingAccount` object has two instance fields:
  - `balance` (inherited from `BankAccount`)
  - `transactionCount` (new to `CheckingAccount`)

Continued…

Implementing the `CheckingAccount` Class

- You can apply four methods to `CheckingAccount` objects:
  - `getBalance()` (inherited from `BankAccount`)
  - `deposit(double amount)` (overrides `BankAccount` method)
  - `withdraw(double amount)` (overrides `BankAccount` method)
  - `deductFees()` (new to CheckingAccount)

Inherited Fields Are Private

- Consider `deposit` method of `CheckingAccount`

```
void deposit(double amount)
{
    transactionCount++;
    // now add amount to balance
    . . .
}
```

- Can't just add `amount` to `balance`
- `balance` is a *private* field of the superclass

Inherited Fields Are Private

- A subclass has no access to private fields of its superclass
- Subclass must use public interface

Public Inheritance in C++:add a method

```
class intObj{
private:int val;
public:
intObj(){val =0;}
int getint() const {return val;}
void setint(int x){val = x;}
void print(){cout <<val<<endl;}
};

class SubintObj: public intObj{
public:
void inc(){val++;}
};

int main(){
SubintObj Sobj; Sobj.print();
Sobj.inc();Sobj.print();
}
```

Put another method in:

increment the internal value of the integer

But, the compiler will return an error:

```
intObj.cc: In method
`void SubintObj::inc()':
intObj.cc:20: member
`val' is private
```

Introducing protected data members

The SubintObj class has a problem --

- it cannot access the private val data member of its base class (intObj)
- it *needs* to access it to implement the inc method … (Question: does it?)

By saying that the val is a *protected* member (in the base class) this will allow all subclasses (derived classes) access to it.

When we do this (see over) the code compiles and works as required.

Public Inheritance in C++: add a method

```
class intObj{
protected: int val;
public:
intObj(){val =0;}
int getint() const {return val;}
void setint(int x){val = x;}
void print(){cout <<val<<endl;}
};

class SubintObj: public intObj{
public:
void inc(){val++;}
};

int main(){
SubintObj Sobj; Sobj.print();
Sobj.inc();Sobj.print();
}
```

protected : to allow access

This will print out

```
0

1
```

Public Inheritance in C++: add a data member

```
class intObj{
private:int val;
public:
intObj() {val =0;}
int getint() const  {return val;}
void setint(int x) {val = x;}
void print() {cout
<<val<<endl;}
};

class SubintObj: public intObj{
public:
int count;
};

int main(){
SubintObj Sobj; Sobj.print();
cout << Sobj.count;
}
```

Put another data member in:

the number of times it has been printed, for example

This will output (typically):

```
0
2285548
```

where the second number shows that the count has not been initialised

Public Inheritance in C++: add a data member

```
class intObj{
private:int val;
public:
intObj(){val =0;}
int getint() const {return val;}
void setint(int x){val = x;}
void print(){cout <<val<<endl;}
};

class SubintObj: public intObj{
public: int count;
SubintObj(){count =0;}
};

int main(){
SubintObj Sobj; Sobj.print();
cout << Sobj.count;
}
```

Question: how can we make sure that the added data type gets initialised correctly when an object of the derived class is constructed?

Answer: we write a constructor for the derived class (in this case a default constructor with no parameters).

The output is now:

```
0

0
```

Question: how did val get initialised on the previous slide??

Parent constructors and destructors

When an object of a derived class (a subclass) is declared, then the parent's **default** constructor is called before the required constructor of the derived class

If the parent has a parent then the grandparent's **default** is called followed by the parent's **default** followed by the derived class constructor

In effect, for an arbitrary number of parents, the parent **default** constructors are called in a top-down fashion.

The same thing happens with the destructors except it is done bottom-up (with the derived child class calling its own destructor first)

NOTE: This can get very complicated (esp. with *multiple inheritance*)

Parameters?

- A derived class will always inherit the constructor of a base class, as well as having its own. The base class constructor is always called first, followed by the constructor of the derived class, and so on down the tree.
- If the base constructor takes no parameters inheritance is implicit.
- If it takes parameters these must be stated explicitly in each derived class.

An Example:

```
class customer
{
private:
        char name[30];
  public:
         {   customer(char * name_in)
              strncpy(name,name_in,29);
              name[29] = '/0';
         }
}
```

http://stackoverflow.com/questions/1258550/why-should-you-use-strncpy-instead-of-strcpy

Inheriting the constructor.

```
class accountcustomer:public customer
{ private:
      int account_number;
  public:
      accountcustomer(char* name_in);
};
accountcustomer::accountcustomer(char*
  name_in):customer(name_in)
{ // constructor body
}
```

07

- Abstract classes & interfaces
- Multiple inheritance
- Copy constructors

Abstract Classes and Interfaces

- **Abstract Classes** are common to both C++ and Java
- Java also supports **Interfaces**. These are abstract classes that are not allowed to contain any implementation code for any member
- An interface is not allowed to have any data fields that can be given values for an object (i.e. they can contain static “metaclass” fields)

C++ Abstract Classes

- A class is abstract in C++ if one or more of the methods of a class is declared to be **pure virtual** e.g.

```
class Shape{
public: virtual double area() = 0;
//…
}
```

- This means that we cannot create objects of that class
- Subclasses may implement the pure virtual methods

C++ Abstract Classes

```
class Shape {
public:
    virtual double area() = 0;
    virtual double circumference() = 0;
};

class Polygon : public Shape {
protected:
    int numVertices;
    bool starShaped;
};

class CurvedShape : public Shape {
public:
    virtual void polygonalApprox() = 0;
};
```

C++ Abstract Classes

```
class Circle : public CurvedShape {
protected:
    double r;
    static double PI;
public:
    Circle() { r = 1.0; }
    Circle( double r ) { this->r = r; }
    double area() { return PI*r*r; }
    double circumference() { return 2 * PI * r; }

    double getRadius() {return r;}

    void polygonalApprox() {
        cout << "polygonal approximation code goes
      here" << endl;
    }
};
double Circle::PI = 3.14159265358979323846;
```

C++ Abstract Classes

```
class Rectangle : public Polygon {
private: double w, h;
public:
    Rectangle() { w=0.0; h = 0.0; numVertices = 0;
             starShaped = true; }

    Rectangle( double w1, double h1 ) {
        w = w1;
        h = h1;
        numVertices = 4;
        starShaped = true;
    }
    double area() { return w * h; }
    double circumference() { return 2 * (w + h); }
    double getWidth() { return w; }
    double getHeight() { return h; }
};
```

C++ Abstract Classes

```
int main()
{
    Shape* shapes[ 3 ];
    shapes[0] = new Circle( 2.0 );
    shapes[1] = new Rectangle( 1.0, 3.0 
);
    shapes[2] = new Rectangle( 4.0, 2.0 
);

    double total_area = 0;
    for (int i=0; i < 3; i++ )
      total_area += shapes[i]->area();
    cout << "Total area = " << 
total_area << endl;
    return 0;
}
```

- Shape example exercise
  - Using shapes.cpp in Day3.zip implement a triangle class and include its area calculation in main

Java Abstract Classes

- In Java we explicitly annotate a class with abstract to write an abstract class e.g.

```
abstract class Shape{
 abstract public double area();
 //…
 }
```

- The class must start with the keyword abstract and, if we don’t intend to provide an implementation for a method we must also annotate that method with abstract.

Java Abstract Classes

```
abstract class Shape {
    abstract protected double area();
    abstract protected double circumference();
}

abstract class Polygon extends Shape {
    protected int numVertices;
    protected boolean starShaped;
}

abstract class CurvedShape extends Shape {
  abstract public void polygonalApprox();
}
```

Java Abstract Classes

```
class Circle extends CurvedShape {
    protected double r;
    protected static double PI = 3.14159;

    public Circle() { r = 1.0; }
    public Circle( double r ) { this.r = r; }

    public double area() { return PI*r*r; }
    public double circumference() {
            return 2 * PI * r;
    }

    public double getRadius() {return r;}

    public void polygonalApprox() {
        System.out.println("code goes here");
    }
}
```

Java Abstract Classes

```
class Rectangle extends Polygon {
    double w, h;
    public Rectangle() {
        w=0.0; h = 0.0; numVertices = 0;
        starShaped = true;
    }
    public Rectangle( double w, double h ) {
        this.w = w;
        this.h = h;
        numVertices = 4;
        starShaped = true;
    }
    public double area() { return w * h; }
    public double circumference() {
                  return 2 * (w + h);
    }
    public double getWidth() { return w; }
    public double getHeight() { return h; }
}
```

Java Abstract Classes

```
class Test {
    public static void main( String[] args )
    {
        Shape[] shapes = new Shape[ 3 ];
        shapes[0] = new Circle( 2.0 );
        shapes[1] = new Rectangle( 1.0, 3.0 );
        shapes[2] = new Rectangle( 4.0, 2.0 );

        double total_area = 0;
        for (int i=0; i < shapes.length; i++ )
          total_area += shapes[i].area();
        System.out.println("Total area = " +
                                    total_area);
    }
```

Why use abstract classes?

- They help organise classes in the class hierarchy
- They can represent generalized behaviour, which when inherited, can be added-to to give the behaviour of an derived object that we might want to create
- They can help when building up an implementation incrementally
- They can be used to take advantage of inheritance – if we place abstract methods in the base class, all non-abstract derived classes are forced to implement them before an object can be created.

Java Interfaces

- In Java, a class is only allowed to inherit implementations from one direct superclass
- In C++, a class is allowed to inherit implementations from many direct superclasses
- Supporting multiple inheritance can cause programming problems (see later)
- Not supporting multiple inheritance can cause design problems. Java attempts to solve this by supporting Interfaces
- In Java a class can be an implementer of many interfaces

Java Interfaces – the syntax

```
interface Collection {
    public boolean add(Object o);
    public boolean remove(Object o);
    //…
}

class MyCollection implements Collection{
    //…
    // include code for methods declared in
    Collection
    //…
}
```

Java Interfaces

```
interface Drawable {
    public void setColor( Color c );
    public void setPosition( double x, double y );
    public void draw( DrawWindow dw );
}
```

Java Interfaces

```
class DrawableRectangle extends Rectangle implements
   Drawable
{
     private Color c;
     private double x, y;
     public DrawableRectangle( double w, double h )
   { super( w, h ); }

     public void setColor( Color c ) { this.c = c; }
     public void setPosition( double x, double y ){
          this.x = x; this.y = y;
     }
     public void draw( DrawWindow dw ) {
               dw.drawRect( x, y, w, h, c );
     }
}
```

Java Interfaces

```
class DrawableRectangle extends Rectangle implements Drawable
```

Implementations must be supplied for the methods in the interface

```
   public void setPosition( double x, double y )
  {
       this.x = x; this.y = y;
   }
   public void draw( DrawWindow dw )
 { dw.drawRect( x, y, w, h, c ); }
}
```

Multiple interfaces in Java

```
interface Scalable
{
   public Shape scaleTransform();
}

class DrawableScalableRectangle extends Rectangle implements
    Drawable, Scalable
{
   // We must implement the methods of
    // Drawable and Scalable here
}
```

Multiple Inheritance

C++

Example:Syntax

```
class TeachingAssistant:public Student, public Employee
{ …
}
```

Teaching Assistant maintains the attributes and the methods associated with both base classes Student, and Employee

Polymorphic Assignment

```
TeachingAssistant * mary = new
TeachingAssistant();

Employee * e = mary;
//Legal as a Teaching Assistant is-an Employee

Student * s = mary;
//Legal as a Teaching Assistant is-a Student
```

Common Misuse of Multiple Inheritance

A common misuse is to use Multiple Inheritance as a tool to achieve composition rather than specialization (*is-a*):

E.g. The following is a **poor design choice**:

```
class Car : public Engine, public Transmission
{
}
```

Problems with Multiple Inheritance:

- Name Ambiguity:
  - Similar names can be used for different operations in the base classes e.g. an employee might have an id_number and a student might also have an id_number field.

Problems with Multiple Inheritance:

- Name Ambiguity:
  - Both base classes might have a similar get_id() method.
  - The C++ compiler cannot determine which version to use in the TeachingAssistant class: the get_id() in the Employee class or the get_id () in the Student class.

Problems with Multiple Inheritance: Name Ambiguity

Not Advised work-around 1:

Use a fully qualified function name:

```
TeachingAssistant * jo;
jo = new   TeachingAssistant();
cout << “ The TeachingAssistant is” <<
           jo -> Employee::get_id() << “\n”;
```

Problems with Multiple Inheritance: Name Ambiguity

Not Advised work-around 2:

Redefine the ambiguous function in the new class and hide the qualified name in a method body:

```
class TeachingAssistant:public Student, public
  Employee
{
   public: string get_id();
   public: string student_id();
}
```

```
string TeachingAssistant::get_id()
{
    return Employee::get_id();
}

string TeachingAssistant::student_id()
{
    return Student::get_id();
}
```

08

Associations

How do objects communicate?

- In any OO system we would expect many objects of many different classes to communicate with each other.
- In order to do this we must provide links between object to allow them communicate.

- At the level of class design these links are known as associations and may be
  - 1:1 one object of a class has a link with one object of another class
  - 1:N one object of a class has a link with many objects of another class
  - N:M many objects of a class has a link with many objects of another class

- Associations are generally assumed to be bi-directional.
- Associations more frequently occur between objects of different classes, but also occur between objects of the same class.
- The term “actor” (Booch) is used to describe an object which acts upon other objects, “server” for an object that is acted upon and agent for an object which does both.

Aggregation

- Associations form hierarchies which are different from inheritance. These hierarchies are known as aggregations.
- These special types of association is described by terms which include:
  - Aggregation, Composition, Part-Whole,
  - A-Part-Of, Has-a, Containment

A Part Of

- In this type of hierarchy, classes do not inherit from other classes but are composed of other classes.
- Aggregation is **a part of** relationship in which objects representing the components of something are associated with an object representing an entire assembly:

  Example:

  – A sentence is **a part of** a paragraph which is **a part of** a document.

- **Composition**
  - A composition hierarchy defines how an object is composed of other objects in a fixed relationship. The aggregate object cannot exist without its components, which are usually a fixed and stable number.

- **Container**
  - A container is an object of a container class which is able to contain other objects. The existence of the container is independent of whether it actually contains any objects at a particular time. The contained objects are usually dynamic and may be of many different classes.

- **Composition**
  - internal objects are not seen from the outside e.g. Clock is composed of Hand, Face and Works
- **Aggregation**
  - internal objects can be directly accessed from outside objects have a separate existence e.g. Clock Battery

- Elements of a composition hierarchy must be instantiated at run time in the process of instantiating the composite object.

- **E.g** we must instantiate the objects face, hand and works before we use them for a “clock” object.

```
class Wheel
{
private:
  int size;

public:
  Wheel(int wheel_size){
    size = wheel_size;
   }
  int getSize(){ return size; }
};
```

```
class Car{
private:
  Wheel rightfront_wheel, rightback_wheel,
  leftfront_wheel, leftback_wheel;
  Engine engine;
  int passengers;

public:
//will define constructor as Composite
  Car(int wheel_size, int cc_in, int
  passengers_in);
  void showcar();
  int getFrontWheelSize();
  int getBackWheelSize();
};
```

The Constructor (composition)

```
Car::Car(int wheel_size, int
  cc_in, int passengers_in):
      rightfront_wheel(wheel_size),
      rightback_wheel(wheel_size),
      leftfront_wheel(wheel_size),
      leftback_wheel(wheel_size),
      engine(cc_in)
{
  passengers= passengers_in;
}
```

The Method

```
void car::showcar()
{
  cout<<”Wheel Size”
         << rightfront_wheel.getSize()<<endl;
  cout<<”EngineCapacity:”
                 << engine.getCC()<<endl;
  cout<<”No. of Passengers”
                 << passengers<<endl;
}
```

Main

```
void main()
{

  car mycar(24,500,2);
  mycar.showcar();
)
```

Implementing
Associations & Aggregations

- There are two basic ways in which associations and aggregations are implemented
    - Objects contain objects (fixed aggregation)
    - Objects contain pointers to objects (variable aggregations and simple associations)

Fixed aggregations:

- Fixed aggregations are implemented by using a class type as an attribute type in the aggregate class.

```
#include “Chapter.h”
class book
{     private:
        Chapter c1,c2,c3,c4;
      public:
        book(int pages);
        int get_book_pages();
}
```

Class chapter

The class chapter must be defined in a header file and imported in to the file which defines book.

```
class Chapter
{      private:
          int no_of _pages;
       public:
          chapter(int pages);
          int get_pages();
}
```

```
Chapter::Chapter(int pages)
  {
   no_of_pages = pages;
  }
int Chapter::get_pages()
  {
   return no_of_pages;
  }
```

Constructor

The constructor for book calls the chapter constructor:

```
Book(int pages):
c1(pages),c2(pages),c3(pages),c4(pages)
{
}
```

The body of the constructor is empty as the book has no other attributes to instantiate.

Note that this generates a book where each chapter has the same number of pages.

Aggregation Vs Inheritance

- In inheritance a base class constructor is called from a derived class as follows:

```
Derived_class_name::
  Derived_class_constructor_name(types and
  parameters):
  base_class_constructor_name(parameters)
{ set parameters which are not set by the
  base class constructor;
}
```

Aggregation Vs Inheritance

In aggregation a component class constructor is called from an aggregate class as follows:

```
Aggregate_class_name::
  Aggregate_class_construcor_name(type
  s and parameters):
  Component_class_instance(parameters)
{
  set parameters which are not set by
  the component class constructor;
}
```

Variable Aggregations

A component may not always be present at runtime. Therefore, we may represent it using a pointer to an object rather than by an object itself.

Example:

A lecturer may be assigned a course to teach. That lecturer may be removed from that course at anytime. The relationship is one that may change. Hence a fixed aggregation is unsuitable for representing this relationship. Variable aggregation is more suitable.

Variable Aggregation

```
#include “Lecturer.h”
class Course
{ private:
    Lecturer * lec
  public:
    Course();
    void addLecturer();
    void removeLecturer();
}
```

The class lecturer is defined in the header file lecturer.h.

- The Course constructor should initialise the pointer `lec` to NULL as no lecturer is associated with the class when it is set up.

- The method addLecturer() should put the pointer `lec` pointing at a new object of type lecturer:

```
void Course::addlecturer()
{
    lec = new Lecturer;
}
```

- The method removeLecturer should delete the object `lec` is pointing at and set `lec` pointing to NULL:

```
void course::removelecturer()
{
        delete lec;
        lec = NULL;
}
```

If we don’t want to create / delete a new object (lecturer) each time we add/remove we could define the above methods as follows:

```
     void course::addLecturer(Lecturer *teacher)
     {
        lec = teacher;
     }
   void course::removeLecturer()
     {
        delete lec;
        lec = NULL;
     }
```

Implementing 1:1 Associations in one direction.

1:1 associations in one direction are implemented as above with a pointer to a class.

Example:

In the class "Button" the association Button changes Light is modelled by providing a pointer to a light object as an attribute of Button.

The Light Class

```
const int OFF = 0;
const int ON = 1;

class Light
{ private:
  int light_state;
   public:
  Light();
  void change_state();
  void show_state();
}
```

```
Light::Light()
{ light_state = OFF;
}
void Light::change_state()
{
  if(light_state==OFF)
  {     light_state=ON;
  }
  else  {    light_state==OFF;
        }
}
```

```
void Light::show_state()
{
 if(light_state==OFF)
 {
  cout<<”Light is off” << endl;
 }
 else
 {
  cout<<”Light is on” << endl;
 }
}
```

The Button class with association

```
class Button
{ private:
  Light* light_bulb;  // 1:1 association
   public:
  Button(Light* bulb);
  void press();
};
  Button::Button(Light* bulb)
   {
    light_bulb = bulb;
   }
```

```
void Button::press()
  {
   light_bulb -> change_state();

  }
```

Implementing 1:1 Bi-directional Associations

In 1:1 bi-directional associations both classes must include a reference(via a pointer).

```
class Person{
  private:
    string name;
    Person * partner;

   public:
    Person(string name_in) {name = name_in);
    string getname() {return name;}
    void marry(Person *spouse);
    void divorce();
    void showpartner();
  };
```

```
void Person::marry(Person *spouse)
{
  partner = spouse;
  /*We need  to change the method marry so that
  when a partner marries a spouse that spouse
  is also married to that partner - watch for
  recursion!!!*/
}
void Person::divorce()
{
  partner = NULL;
   /*We need  to change the divorce method so
  that when a partner divorces a spouse the
  spouse is also divorced*/
}
```

```
void Person::showpartner()
{
  if (partner != NULL)
  {
   cout << name << “is married
     to”<<partner->getname() << endl;
  }
   else
  { cout << name << “is single.” <<
  endl;
  }
}
```

```
void main ()
{Person * Ross = new Person (“Ross”);
 Person * Rachel = new Person (“Rachel”);

  Ross -> showpartner();
  Rachel -> showpartner();
  Ross -> marry(Rachel);
  Rachel -> marry(Ross );

  Ross -> showpartner();
  Rachel -> showpartner();
  Ross -> divorce ();
  Rachel -> divorce();
}
```

```
void main ()
{Person * Ross = new Person (“Ross”);
 Person * Rachel = new Person (“Rachel”);

  Ross -> showpartner();
  Rachel -> showpartner();
  Ross -> marry(Rachel);
  Rachel -> marry(Ross );

  Ross -> showpartner();
  Rachel -> showpartner();
  Ross -> divorce ();
  Rachel -> divorce();
}
```

Problem: User has to update both ends of the relationship.

The *this* operator

- Problem: User has to update both ends of the relationship.
- In C++ this may be avoided if a link is created in one direction and modified in the opposite direction using the operator ***this***.

- Using the class course defined earlier we can model a 1:1 association between a lecturer and course as follows:

A lecturer teaches one course and a course is taught by one lecturer.

```
class Course
{private:
        Lecturer *lec // L is a pointer to
              //an object of type lecturer
 public:
        Course(int code);
        void addLecturer();
        void removeLecturer();
}
```

```
class Lecturer
  { private:
        Course *c
             // c is a pointer to an
             //   object of type course
    public:
        Lecturer(int number):
        void addCourse();
        void removeCourse();
    }
```

```
void Course::addLecturer(Lecturer
 *teacher)
{
 lec = teacher;
 lec -> addCourse(this);
}

void Lecturer::addCourse(Course
 *module)
{
 c = module;
}
```

```
void main()
{
  Course *cs613 = new Course(1);
  Lecturer *Tom = new Lecturer(12);

  cs613->addLecturer(Tom);
  // this operator => the association
  //is updated for both objects.
}
```

- *Exercise: Fix the problem with the marriage example.*

- *Note: N:M associations are implemented using arrays of pointers from one class to another.*

09

Operator Overloading

Java: String str = “Over” + “load”;

C++: string str = “Over” + “load”;

Operator Overloading C++/Java

- Many operators are overloaded in Java and in C++
- You are not allowed to create your own operator overloading in Java
- You can in C++ as …

  when the argument supplied to an operator like

  `+, = , <<` are of class type then the compiler

  invokes an operator function associated with the

  operator token e.g. for the token **+** the operator

  function ***operator+*** is invoked.

Operator Overloading

- The compilers built in operator symbols work with classes defined by the programmer.
- The meaning of an operator may be overloaded (in OO languages), so that its behaviour may be polymorphic i.e. implemented differently for different classes.

Overloading operators allow us to give all classes in a system a common interface, allowing us to perform similar operations on a range of different objects.

Assignment

- Assignment (=) is already overloaded. It has a default behaviour for all objects. It can be overridden for individual classes and not automatically for their descendant classes.
- The user defined behaviour of any other operator may be inherited by descendant classes.

Some things to remember…

from “The C++ Programming Language”, Stroustrup

- When an operator is overloaded in C++, at least one of its operands must be of class type or enumeration type
- At least one of the operands in an operator overload definition must not be a built in type
- The predefined precedence of an operator cannot be altered by an overload definition
- An overload definition cannot alter the predefined arity of an operator
- Arguments can be passed to the parameters of an overloaded operator by value, by reference but not by pointer
- Default arguments for overloaded operators are illegal, except for operator ‘()’
- The following operators cannot be overloaded

  :: .* . ?:

C++ Syntax

```
Return_type operator symbol(parameter list)
```

- Operators which may be overloaded include arithmetic and relational operators.
- The return type for the boolean operators is a boolean value represented by 1 or 0.
- Arithmetic operators return type is an object of the same class as other objects in the expression.

C++ Syntax

- The parameter list usually consists of an object of the class, **passed by reference** rather that by value, to avoid making an unnecessary copy.

- The **const** prefix is frequently used to indicate that, although the object has been passed by reference it should not be modified by the method.

- Overload definitions **can be global or can be member functions for a class** – the overloaded operator works the same with either implementation.

Overloading the ‘+’ operator

```
class StudentGrade
{private:
      int mathsgrade;
      int englishgrade;
public:
      StudentGrade();
      int getmathsgrade();
      int getenglishgrade();
      void setmathsgrade(int grade_in);
      void setenglishgrade(int grade_in);
      studentgrade operator +(const
      studentgrade& grade_in);
};
```

Adding student grades :

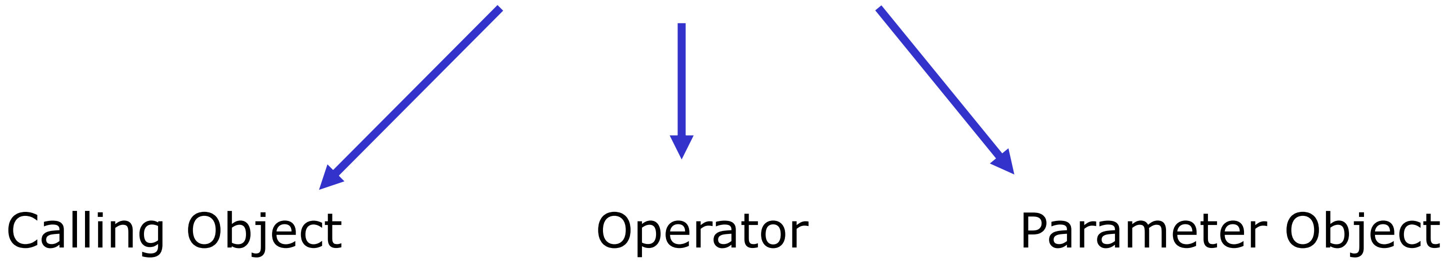
Grade 3 = Grade 1 + Grade 2

Defining the operator + as

```
StudentGrade operator +(const StudentGrade& grade_in);
```

lets us add 2 studentgrade objects together as follows:

```
void main()
{
      StudentGrade grade_total, student1, student2;
      grade_total = student1 + student2;
}
```



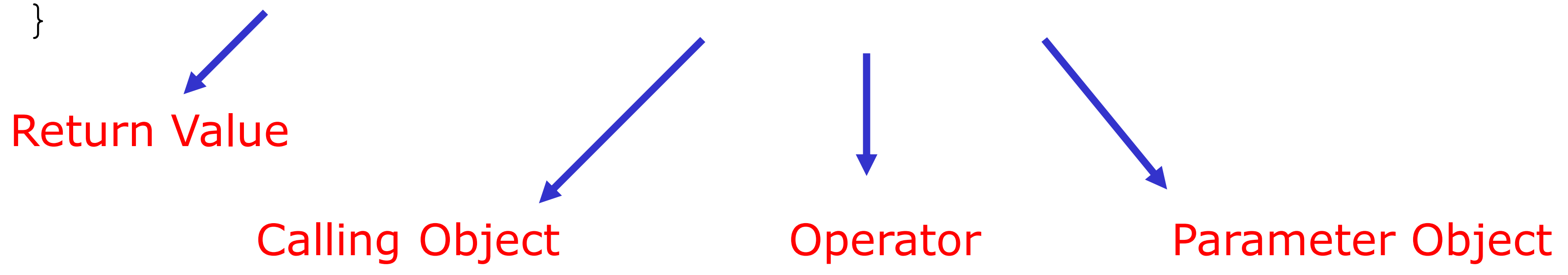
The operator definition

```
StudentGrade StudentGrade::operator + (const
  StudentGrade& grade_in)
{
      StudentGrade result;
      result.mathsgrade =
            mathsgrade + grade_in.mathsgrade;

      result.englishgrade =
            englishgrade + grade_in.englishgrade;

      return result;
}
```

Adding student grades :

In main we can reference the components of the object grade_total as `grade_total.getmathsgrade()` …

```
StudentGrade operator +(const StudentGrade& grade_in);
void main()
{
      StudentGrade grade_total, student1, student2;
      grade_total = student1 + student2;
}
```



Stream Classes

- A stream is a general name given to a flow of data e.g. cin, cout. In C++ a stream is represented by an object of a particular class.
- There are no formatting characters in streams, since each object knows how to display itself (due to operator overloading).
- Existing operators and functions (e.g. insertion << and extraction >>) may be overloaded to work with classes that you create.

Istream & ostream

- The istream class performs specific input activities or extractions
- The ostream class handles output or insertion activities
- To overload the ostream and istream operators we use the following syntax:

```
ostream& operator << (ostream&, class_type&);
istream& operator >> (istream&, class_type&);
```

where class_type is a parameter of some user defined class, passed by reference. The method returns a reference to an ostream / istream object, and also takes one as parameter.

```
class person{
private:
  char name[20];
  int age;
public:
  char* getname();
  int getage();
  void setname(char * name_in);
  void setage(int age_in);
};
```

The Methods

```
char* person::getname()
{ return name; }

int person::getage()
{ return age; }

void person::setname(char* name_in)
{ strncpy(name, name_in, 19);
   name[19] = '\0';}

void person:: setage(int age_in)
{ age = age_in; }
```

Overload >>

```
istream&operator >> (istream& in, person& person)
{   char temp[20];
    int age;
    cout << “Enter Age”;
    in >> age;
    person.setname(name);
    person.setage(age);
    return in;
}
```

Overloading <<

```
ostream& operator <<
            (ostream& out, person& person)
{ return out << “Name”<<
  person.getname() <<endl
  <<”Age” << person.getage() <<endl;
}
```

Main

In main a person is input / output using cin and cout

```
void main()
{ person p;
  cin >> p;
  cout << p;
}
```

Example: Complex Numbers

```
class MyComplex {
private:    double re, im;
public:
    MyComplex( double r, double i ) : re(r), im(i)
   {}
    double getReal() const { return re; }
    double getImag() const { return im; }
};
```

`MyComplex operator+( const MyComplex arg1,`

```
MyComplex operator+( const MyComplex arg1,
                               const MyComplex arg2
   )
{
  double d1 = arg1.getReal() + arg2.getReal();
   double d2 = arg1.getImag() + arg2.getImag();
      return MyComplex( d1, d2 );
}
```

Example: Complex Numbers

```
class MyComplex {
private:    double re, im;
public:
    MyComplex( double r, double i ) : re(r), im(i) {}
    double getReal() const { return re; }
    double getImag() const { return im; }
};

MyComplex operator+( const MyComplex arg1,
                               const MyComplex arg2 )
{
   double d1 = arg1.getReal() + arg2.getReal();
   double d2 = arg1.getImag() + arg2.getImag();
       return MyComplex( d1, d2 );
}
```

Global Definition Example: Complex Numbers

```
class MyComplex {
private:    double re, im;
public:
    MyComplex( double r, double i ) : re(r), im(i) {}
    double getReal() const { return re; }
    double getImag() const { return im; }
};

MyComplex operator+( const MyComplex arg1, const MyComplex
   arg2 )
{
       double d1 = arg1.getReal() + arg2.getReal();
   double d2 = arg1.getImag() + arg2.getImag();
       return MyComplex( d1, d2 );
}
```

operator

Object argument 1

Object argument 1

```
MyComplex operator-( MyComplex arg1, MyComplex arg2 )
{
  double d1 = arg1.getReal() - arg2.getReal();
    double d2 = arg1.getImag() - arg2.getImag();
    return MyComplex( d1, d2 );
}

ostream& operator<< (ostream& os, const MyComplex& arg )
{
      double d1 = arg.getReal();
      double d2 = arg.getImag();
      os << "(" << d1 << ", " << d2 << ")" << endl;
            return os;
}
```

Using the operator overloading

```
int main()
{
    MyComplex first(3, 4);
    MyComplex second(2, 9);

    cout << first;                          // (3, 4)
    cout << second;                         // (2, 9)
    cout << first + second;                 // (5, 13)
    cout << first - second;                 // (1, -5)
    return 0;
}
```

Member function overload

```
class MyComplex {
    double re, im;
public:
    MyComplex( double r, double i ) : re(r),
  im(i) {}
    MyComplex operator+( MyComplex) const;
    MyComplex operator-( MyComplex) const;
    //  ostream& operator<< ( const MyComplex&
  );              // WRONG
    friend ostream& operator<< ( ostream&,
  const MyComplex& );
};
```

```
MyComplex MyComplex::operator+( const MyComplex arg ) const{
      double d1 = re + arg.re;
     double d2 = im + arg.im;
     return MyComplex( d1, d2 );
}
```

operator

Parameter object

A promise not to modify any data fields of the calling object

```
MyComplex MyComplex::operator+( const MyComplex arg ) const
   {
    double d1 = re + arg.re;
   double d2 = im + arg.im;
   return MyComplex( d1, d2 );
}

MyComplex MyComplex::operator-( const MyComplex arg ) const
   {
   double d1 = re - arg.re;
        double d2 = im - arg.im;
        return MyComplex( d1, d2 );
}
```

```
MyComplex MyComplex::operator+( const MyComplex arg ) const
   {
        double d1 = re + arg.re;
   double d2 = im + arg.im;
   return MyComplex( d1, d2 );
}

MyComplex MyComplex::operator-( const MyComplex arg ) const
   {
   double d1 = re - arg.re;
        double d2 = im - arg.im;
        return MyComplex( d1, d2 );
}

ostream& operator<< ( ostream& os, const MyComplex& c ) {

   os << "(" << c.re << ", " << c.im << ")" << endl;
        return os;
}
```

```
int main()
{
    MyComplex first(3, 4);
    MyComplex second(2, 9);

    cout << first;                          // (3, 4)
    cout << second;                         // (2, 9)
    cout << first + second;                 // (5, 13)
    cout << first - second;                 // (1, -5)
    return 0;
}
```

Global vs Member Definition

Global:

```
MyComplex operator+( const MyComplex arg1,
                     const MyComplex arg2 )
```

first + second is translated by the compiler as

operator+ (first, second)

Member:

```
MyComplex MyComplex::operator+( const MyComplex arg ) const
```

first + second is translated by the compiler as

first.operator+ (second)

Global vs Member Definition

This is why writing a member definition for <<
would be wrong:
i.e. ostream& operator<< (const MyComplex c);
would mean that cout<< expression; would be
interpreted as cout.operator<<(expression).

We are not allowed to define new member functions for
the output stream class (of which cout is an object) so
we cannot do this.

.

Global vs Member Definition

This is why writing a member definition for <<
would be wrong:
i.e. ostream& operator<< (const MyComplex c);
would mean that cout<< expression; would be
interpreted as cout.operator<<(expression).

We are not allowed to define new member functions for
the output stream class (of which cout is an object) so
we cannot do this.

We must use a global overload definition for the
output operator.
ostream& operator<< (ostream& os, const MyComplex& c)
which is interpreted as << (cout, expression)

Unary operators: global definition

```
class MyComplex {
    double re, im;
public:
    MyComplex( double r, double i ) : re(r), im(i) {}
    double getReal() const { return re; }
    double getImag() const { return im; }
};

// global overload definition for "-" as a unary operator
MyComplex operator-( const MyComplex arg ) {
    return MyComplex( -arg.getReal(), -arg.getImag() );
}
```

Unary operators: global definition

```
// global overload definition for "<<" as a binary operator
ostream& operator<< ( ostream& os, const MyComplex& arg ) {
    double d1 = arg.getReal();
    double d2 = arg.getImag();
    os << "(" << arg.getReal() << ", " << arg.getImag() << ")" << endl;
    return os;
}

int main()
{
    MyComplex c(3, 4);
    cout << c;              // (3, 4)
    cout << -c;             // (-3, -4)
    return 0;
}
```

Unary operators: member definition

```
class MyComplex {
   double re, im;
public:
   MyComplex( double r, double i ) : re(r), im(i) {}
   MyComplex operator-() const;
   friend ostream& operator<< ( ostream&, const MyComplex& );
};

//Member-function overload definition for "-"
MyComplex MyComplex::operator-() const {
   return MyComplex( -re, -im );
}
```

Unary operators: member definition

```
//This overload definition has to stay global
ostream& operator<< ( ostream& os, const MyComplex& c ) {
    os << "(" << c.re << ", " << c.im << ")" << endl;
    return os;
}

int main()
{
    MyComplex c(3, 4);
    MyComplex z = -c;
    cout << z << endl;                    // (-3, -4)
    return 0;
}
```

Unary operators: member definition

```
//This overload definition has to stay global
ostream& operator<< ( ostream& os, const MyComplex& c ) {
    os << "(" << c.re << ", " << c.im << ")" << endl;
    return os;
}

int main()
{
    MyComplex c(3, 4);
    MyComplex z = -c;
    cout << z << endl;          // (-3, -4)
    return 0;
}
```

Translated by the compiler to
MyComplex z = c.operator-()

Friends

```
class MyComplex {
    double re, im;
public:
    MyComplex( double r, double i ) : re(r), im(i) {}
    MyComplex operator+( MyComplex) const;
    MyComplex operator-( MyComplex) const;
     friend ostream& operator<< ( ostream&, const MyComplex& );
};
```

Friends

- Friend declarations, can be in the private, public or protected sections of a class.
- They allow the global overload definition of << to directly access the private data members of the MyComplex argument object.

Friends

- **Friend Functions** – Functions/ methods/ classes which are not actually members of a given class, but nevertheless have direct access to its attributes, just as if it were a member.
- They overcome the following problems with overloaded operators:
  - Without friend functions
    - parameter objects may only be accessed by their external interface (methods)
    - there is no evidence in the class body that the operator has been overloaded for objects of that class

Friends

- A friend function can act like a bridge between two classes
- If you want a method to operate on two different classes which are unrelated, the method concerned should be declared in both classes with the keyword friend before it. This gives the method access to the private data of both classes (exception to the rule that only member functions can access private data from outside the class)
- Friend functions should be used sparingly

Friends

Friend functions should be used sparingly!!!

Examples with the class distance

- nofri.cpp : + operator is overloaded but it is limited
- frengl.cpp: Friend overloaded + operator overcomes these limitations
- misq.cpp: has a square function for distances
- frisq.cpp: Rewrites square as a functional operator

Rational Numbers Exercise

- See Rational.cpp and Rational.h files
  - Overload +, -, <=, >=, !=
  - The operators *, / , <, >. == are already overloaded for you in Rational .cpp

I/O streams

- In C++ and Java, data I/O is carried out with streams.
- You open a stream to the destination to which you want to write data.
- You open the stream to the source from where you want to bring in the information.

C++ Streams

- Most programs need to save data to disk files and read it back. Working with disk files require the following:
  - ifstream for input
  - fstream for input and output
  - ofstream for output

- Objects of these classes can be associated with disk files, and we can use their member functions to read and write to files.
- These classes are declared in **fstream.h** which includes the **iostream.h** file

Input-Output Ops for Char Streams

```
ofstream out (”invoice.dat”);
if (!out)
cout << “cannot open output file” << endl;
int x = 1;
out << x << “ “ << 4.5 <<“ “ << “Hello”<< endl;
```

Input-Output Ops for Char Streams

```
ofstream out (”invoice.dat”);
if (!out)
cout << “cannot open output file” << endl;
int x = 1;
out << x << “ “ << 4.5 <<“ “ << “Hello”<< endl;
```

Open the file invoice.dat for writing text to
- previous contents will be over-written

System provided overloaded operator <<

This program causes the character string
1 4.5 Hello
To be written into the file invoice.dat

Input-Output Ops for Char Streams

```
ofstream out (”invoice.dat”, ios.app);
if (!out)
cout << “cannot open output file” << endl;
int x = 1;
out << x << “ “ << 4.5 <<“ “ << “Hello”<< endl;
```

Open the file invoice.dat for writing text to
Using ios.app requests that the new items are appended to the existing contents

This program causes the character string
1   4.5 Hello
To be written into the file invoice.dat

Input-Output Ops for Char Streams

```
int x; double y; string s;
ifstream in (”invoice.dat”);
if (!in)
cout << “cannot open input file” << endl;
in >> x;
in  >> y;
in  >> s;

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

This program causes x to be given the value 1, y the value 4.5 and z the value Hello
The values are read from the file invoice.dat

Input-Output Ops for Char Streams

```
fstream inout (”invoice.dat”, ios::in | ios::out);
if (!inout)
cout << “cannot open file” << endl;
```

Open in read and write mode

A stream that can be used for reading and writing

Input-Output Ops for Char Streams

```
fstream inout (”invoice.dat”, ios::in | ios::out);
if (!inout)
cout << “cannot open file” << endl;
```

A file that has been opened for both reading and writing has two positions of the file pointer associated with it :

1. The current file position for reading a byte (the get position)
2. The current file position for writing a byte (the put position)

Input-Output Ops for Char Streams

```
fstream inout (”invoice.dat”, ios::in | ios::out);
if (!inout)
cout << “cannot open file” << endl;
```

A file that has been opened for both reading and writing has two positions of the file pointer associated with it :

1. The current file position for reading a byte (the get position)
2. The current file position for writing a byte (the put position)

seekp (for the put position) and seekg (for the get position) may be used to control these positions.

tellp (for the put position) and tellg (for the get position) may be used to get the current positions.

Opening modes:

- ios::in opens for input (default for ifstream)
- ios::out opens for output (default for ofstream)
- ios.ate open and seek end of file
- ios::app append all output
- ios::trunc destroys contents of existing file
- ios::nocreate open fails if file doesn't exist
- ios::noreplace open fails if the file exists
- e.g. *ofstream outfilename("fdata.txt" , ios::app | ios::nocreate);*

Closing a file

– *filename.close();* closes the file filename

Detecting EOF

– *while (!infilename.eof())* // while the end of file has not been encountered

Detecting Errors

– *while (infilename.good())* // while an error has not been encountered

I/O of single characters

– put() and get() may be used for I/O for single characters

- o*utfilename(put(A[i]);*
- *infilename(get(A[i]);*

```
 int main() {

fstream inout( "invoice.dat", ios::in | ios::out );
//write to the file:
inout << 1234 << "    "  << 56.78 << "   " << "apples" << '\n';

 //current get position:
cout << inout.tellg() << endl;                  // 21
 //current put position:
cout << inout.tellp() << endl;                  // 21

 //reset get position to the beginning of the file:
inout.seekg( ios::beg ); //could use inout.seekg(0);
```

```
     cout << inout.tellg() << endl;              // 0
     cout << inout.tellp() << endl;              // 0
     int x; double y;          string z;

    //read from file:
     inout >> x >> y;
     inout >> z;

    cout << x << " " << y << " " << z << endl;        // 1234 56.78 apples
    return 0;
}
```

If we want to skip some data e.g. skip over 56.78 we would write …

```
int x = 0;
    double y = 0.0;
    string z = "";

    //read first item from file:
    inout >> x;

    //move the stream 8 positions to the right of the
    //current get position: (this will cause the stream
    //to skip over the number 56.78)
    inout.seekg( 8, ios::cur );

    //read next item from file:
    inout >> z;
    cout << x << " " << y << " " << z << endl;
                                // 1234  0  apples
```

Searching a file

- Read from the file
- Use strcmp to compare the file contents to the item you are searching for
  - E.g. The following code reads an object called person. The class that this object belongs to has a method called getname. This method returns a string which is compared to the string (str) which is been searched for.

```
    person.read((char*)&person, sizeof person);
    if ((strcmp person.getname(), str) == 0)
...
```

- **Writing to a file**

  cout << “\n Enter Person Data”;

  pers.getdata();

  file.write((char) &person,sizeof(person))*

- **Reading from the start of a file**

  file.seekg(o)

  file.read((char) &person, sizeof(person))*

```
// saves person object to disk
#include <fstream.h>        //for file streams

class person                // class of persons
  { private:
      char name[40];          // person's name
      int age;                  // person's age
   public:
     void getData()  // get person's data
      void showData()  // display person's data
   };
```

```
// get person's data
void Person ::getData() {
    cout << "Enter name: “<<endl ; cin >> name;
    cout << "Enter age: “ <<endl; cin >> age;
}
// display person's data
void Person::showData()
{
    cout << "\n   Name: " << name <<endl;
  cout << "\n   Age: " << age <<endl;
}
```

```
void writing()
 {  person pers;     // create a person
          pers.getData();   // get data for person

   // create ofstream object
   ofstream outfile("PERSON.DAT",         ios::binary);
   // write to it
   outfile.write((char*)&pers,sizeof(pers));

   }
```

```
void reading()
{   person pers; // create person variable

            // create stream
            ifstream infile("PERSON.DAT", ios::binary);

            // read stream
            infile.read((char*)&pers, sizeof(pers) );

            // display person
            pers.showData();
}
```

I/O Streams for Java

- In C++ all the input/output functionality is packed into a few stream classes
- In Java there is a separate stream class for most situations
- The files WriteIntToFile.java, ReadIntFromFile.java WriteStringToFile.java and ReadStringFromFile.java illustrate some of the alternatives…
- ObjectIO.java shows how to read and write objects from/to a file in Java

I/O Streams for Java

- Classes that implement serializable must implement special methods with these exact signatures:
  - private void writeObject(java.io.ObjectOutputStream out) throws IOException
  - private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException;

I/O Exercise

- Write a main method that writes/reads rational number objects to/from files …
- See examples on Moodle: Files and Objects folder for mulpers.cpp as an example for reading and writing multiple objects.
- See the Distance class handout with file DIST.DAT

10

Templates

Templates

- In C++ genericity is achieved by the use of templates. A template will operate on any data type for which the internal implementation is appropriate
  - e.g. a template function which compares two items using > may be used on any data type on which the > symbol is appropriate
- Generic functions require that all parameters passed to them are able to respond to the internal processes e.g. > , < , This means that objects passed as parameters to generic functions may have to have overloaded operations.

- Because overloading requires a different implementation to be provided for all possible parameter types it is not easy to extend to handle new types
- A generic function only has one implementation which may be used for all parameter types, even those which are not yet defined.
- Genericity gives a common interface to all objects

C++ Templates Syntax

```
enum bool {false, true}
template <class T> bool isGreater(T x, T y)
{ bool isGreater;
  if ( x > y )
  { is_greater = true;
  }else{
       is_greater = false;
  }

  return is_greater;
}
```

Main

```
void main()
{ int x,y; bool z;
  cout << “Enter two integers:” << endl;
  cin >> x; cin >> y;
  z = isgreater(x,y);
}
```

With the studentgrade class with overloaded > we could write:

```
  Studentgrade student1,student2;
  z = isgreater (student1,student2);
```

Generic Classes and C++ Templates

A generic class can be thought of as an abstract specification of a set of classes

It is a recipe for producing classes

An instance of a generic class is a class

(An instance of a class is an object)

In C++, generic classes are defined by templates

Templates can be used to produce functions and classes

C++’s template facility is very powerful (surprise surprise!) but it can become very complicated when combined with other language features

Generic Classes and C++ Templates … continued...

We will start by looking at function templates

A good example is the swap function

This is often used in sorting algorithms

How would we write a generic algorithm to sort elements of any class?

A good first step is to provide a swap for elements of any class.

Note: we will look at the importance of any in later examples

Function Templates … a swap...

```
template <class T>
void swap (T& x, T& y){
T temp = x; x = y; y = temp;}


int main (){
int a = 1, b =2;
cout << "a is "<< a<<", b is "<<b<<endl;
swap (a,b);
cout << "a is "<< a<<", b is "<<b<<endl;
}
```

This outputs:

```
a is 1, b is 2
a is 2, b is 1
```

This example can now be extended to show the power of the generic swap function

Function Templates … a swap...

```
template <class T>
void swap (T& x, T& y){
T temp = x; x = y; y = temp;}
int main (){
int a = 1, b =2;
cout << "a is "<< a<<", b is"<<b<<endl;
swap (a,b);
cout << "a is "<< a<<", b is"<<b<<endl;
char c = 'c', d = 'd';
cout << "c is "<< c<<", d is"<<d<<endl;
swap (c,d);
cout << "c is "<< c<<", d is"<<d<<endl;
}
```

This outputs:

```
a is 1, b is 2
a is 2, b is 1
c is c, d is d
c is d, d is c
```

Function Templates … an <u>invalid</u> swap...

```
template <class T>
void swap (T& x, T& y){
T temp = x; x = y; y = temp;}

int main (){
int a = 1, b =2;
char c = 'c', d = 'd';
swap (a,d); // will not compile
cout << "a is "<< a<<", d is
"<<d<<endl;}
```

The compiler reports the following error:

```
generic.cc: In function
`int
main()':generic.cc:20:

 no matching function for
call to
`swap (int &, char &)'
```

Function Templates … a bubble sort ...

```
template <class T>
void swap (T& x, T& y){
T temp = x; x = y; y = temp;}

template <class S>
void sort (S* v, int n){
S temp;
for (int i =1; i< n; i++)
  for (int j = 0; j < n-i; j++)
     if (v[j] > v[j+1]) swap (v[j],
v[j+1]);}

int main (){int a[6] = {3, 1, 6, 8,
2, 4};
sort (a,6);
for (int i =0; i<6; i++) cout <<
a[i]<<endl;
}
```

This will output:

```
1
2
3
4
6
8
```

Question: how to improve things?

Function Templates … a better bubble sort ...

```
template <class T>
void swap (T& x, T& y){
T temp = x; x = y; y = temp;}

template <class S>
void sort (S* v, int n){
S temp;
for (int i =1; i< n; i++)
   for (int j = 0; j < n-i; j++)
      if (v[j] > v[j+1]) swap
 (v[j], v[j+1]);}

template <class R>
void print (R* v, int n){
for (int i =0; i<n; i++) cout << "
"<< v[i]; cout <<endl;}
```

```
int main (){
int a[6] = {3, 1, 6,
8, 2, 4};
char b[4] =
{'c','x','a','b'};
sort (a,6);print(a,6);
sort (b,4);print(b,4);
}
```

Output:

```
1 2 3 4 6 8
a b c x
```

Function Templates … some ‘problems’ ...

Leave the definitions of the generic functions swap, sort and print as before. Now, try to test them on a user defined class:

```
class intObj{
public:
intObj(int x){val =x;}
int val;};
int main(){
intObj obj1(1), obj2(2);
swap(obj1,obj2);
cout << "obj1 has value "<<
obj1.val;
cout << ", obj2 has value
"<<obj2.val<<endl;}
```

Output:

```
obj1 has value 2,
obj2 has value 1
```

This seems to work as required!

Function Templates … some 'problems' ...

Leave the definitions of the generic functions swap, sort and print as before. Now, try to test them on a user defined class:

```
class intObj{
public:
intObj(int x){val =x;}
int val;};

int main (){

intObj obj1(1), obj2(2), obj3(3);
intObj objlist[3] ={obj1, obj2,
obj3};
print(objlist,3); // will not
compile
}
```

Question: why do we get a load of compiler errors?

Question: what sort of errors are they and how can we fix them?

Function Templates … some ‘problems’ ...

```
class intObj{
public:
intObj(int x){val =x;}
int val;
void print(){cout <<" "<<val;}
};
template <class R>
void print (R* v, int n){
for (int i =0; i<n; i++) v[i].print();
 cout <<endl;}

int main (){
intObj obj1(1), obj2(2), obj3(3);
intObj objlist[3] ={obj1, obj2, obj3};
print(objlist,3);
}
```

The following changes fix the problem:

add a print method to the intObj class

use this method in the print template for printing arrays

This gives the output:

```
 1 2 3
```

Question: why is this not a good approach?

Question: what is a better approach?

Function Templates … some ‘problems’ ...

```
class intObj{
public:
intObj(int x){val =x;}
int val;
void print(){cout <<" "<<val;}
};
template <class R>
void print (R* v, int n){
for (int i =0; i<n; i++) v[i].print();
 cout <<endl;}

int main (){
int a[6] = {3, 1, 6, 8, 2, 4};
print(a,6);// Will not compile
}
```

We now try to use the new (improved?) code to print an array of integers.

Question: why do we get the following compiler error -

```
generic.cc: In function `void
print<int>(int *, int)':
generic.cc:27: request for member
`print' in `*(v + +(i * 4))',
which is of non-
aggregate type `int'
```

Function Templates … some ‘problems’ …

```
class intObj{
public:
intObj(int x){val =x;}
int val;
void print(){cout <<" "<<val;}
};
template <class R>
void print (R* v, int n){
for (int i =0; i<n; i++)
v[i].print();
 cout <<endl;}

int main (){
int a[6] = {3, 1, 6, 8, 2, 4};
sort(a,6);
}
```

But we can still sort them.

Question: why do we not get a compiler error when we call sort instead of print?

Note: this all seems very complicated. It is because of the way C++ mixes classes and types!

Solution: overload the standard operators in classes e.g. cout <<

11

Copy Constructors and Copy Assignments

C++ / Java

C++ Example: No Pointers

```
class X {
    private: int n;
    public: X(…) {…}
};
void f()
{   X x1;
    X x2 = x1;
    X x3;
    x3 = x2;
}
```

C++ Example: No Pointers

```
class X {
    private: int n;
    public: X(…) {…}
};
void f()
{   X x1;
    X x2 = x1;
    X x3;
    x3 = x2;
}
```

Copy byte by byte the data fields of object x1 into the memory locations reserved for the data members of object x2.

Copy byte by byte the data fields of object x2 into the memory locations reserved for the data members of object x3.

C++ Example: No Pointers

```
class X {
    private: int n;
    public: X(…) {…}
};
void f()
{   X x1;
    X x2 = x1;
    X x3;
    x3 = x2;
}
```

Copy byte by byte the data fields of object x1 into the memory locations reserved for the data members of object x2.

Copy byte by byte the data fields of object x2 into the memory locations reserved for the data members of object x3.

Three objects delete – all copies of the same object but stored in 3 locations

C++ Example: Pointers

```
class X {
    private:        int *p; int size;
    public:         X(…) {…}
                    ~X(){delete [] p;}
};
void f()
{   X x1;
    X x2 = x1;
    X x3;
    x3 = x2;
}
```

C++ Example: Pointers

```
class X {
    private:        int *p; int size;
    public:         X(…) {…}
                    ~X(){delete [] p;}
};
void f()
{   X x1;
    X x2 = x1;
    X x3;
    x3 = x2;
}
```

If we assume the same meaning of = as before *(i.e. a field by field copy of the object on the right into the memory locations for the members of the object on the left of =)* then x1.p, x2.p and x3.p will all end up pointing at the same chunk of memory

What happens here?

C++ Example: Pointers

```
class X {
    private:        int *p; int size;
    public:         X(…) {…}
                    ~X(){delete [] p;}
};
void f()
{   X x1;
    X x2 = x1;
    X x3;
    x3 = x2;
}
```

If we assume the same meaning of = as before *(i.e. a field by field copy of the object on the right into the memory locations for the members of the object on the left of =)*
then x1.p, x2.p and x3.p will all end up pointing at the same chunk of memory

The destructor for the X object is called three times for the same object – this will cause a crash or at least some unpredictable behaviour.

C++ Assignment Operators

This problem can be avoided by defining what it means to copy an object for the purpose of initialisation (copy constructor) and for the purpose of assignment (copy assignment operator)

For type T, the prototype of the copy constructor is:
T (const T&)

For type T, the prototype of the copy assignment operator is:
T& operator =(const T&)

The C++ Copy Constructor

T (const T&)

construct an object of type T whose members are set according to the composition of the argument passed to the constructor e.g.

```
class X
{
        private:
        int *ptr; int size;

        public:

        X(...) {...}

        ~X()
        {delete [] ptr;}
};
```

```
X(const X& xobj)
{
   size = xobj.size;
   ptr = new int[size];
   for(int i = 0; i < size; i ++)
  {          ptr[i] = xobj.ptr[i];
   }
}
```

The C++ Copy Constructor

T (const T&)

construct an object of type T whose members are set according to the composition of the argument passed to the constructor e.g.

```
class X
{
        private:
           int *ptr;
           int size;

        public:
           X(...) {...}

           ~X()
           {delete [] ptr;}
};
```

```
X(const X& xobj)
{
   size = xobj.size;
   ptr = new int[size];
   for(int i = 0; i < size; i ++)
  {         ptr[i] = xobj.ptr[i];
   }
}
```

Copy contents of xobj

Acquire new memory

C++ Assignment Operators

copy assignment operator prototype: **T& operator = (const T&)**
copy an object of type T into another object of type T e.g.

```
class X
{
        private:
          int *ptr;
          int size;

        public:
          X(…) {…}

          ~X()
          {delete [] ptr;}
};
```

```
X& operator=(const X& xobj)
{
   if (this !=&xobj)
   {
          delete [] ptr;
          size = xobj.size;
          ptr = new int[size];
          for(int i = 0; i < size; i ++)
          {          ptr[i] = xobj.ptr[i];
          }
          return *this;
   }
}
```

C++ Assignment Operators

copy assignment operator prototype: **T& operator = (const T&)**
copy an object of type T into another object of type T e.g.

```
class X
{
        private:
          int *ptr;
          int size;

        public:
          X(…) {…}

          ~X()
          {delete [] ptr;}
};
```

```
X& operator=(const X& xobj)
{
    if (this !=&xobj)
   {
          delete [] ptr;
          size = xobj.size;
          ptr = new int[size];
          for(int i = 0; i < size; i ++)
          {          ptr[i] = xobj.ptr[i];
          }
          return *this;
   }
}
```

C++ Assignment Operators

copy assignment operator prototype: T& operator = (const T&)

copy the object of type T)whose members are set according to the composition of the argument passed as parameter) into another object of type T e.g.

```
X& operator=(const X& xobj){
  if (this !=&xobj){
              delete [] ptr;
              size = xobj.size;
              ptr = new int[size];
              for(int i = 0; i < size; i ++)
              {           ptr[i] = xobj.ptr[i];
              }
  return *this;
}
```

Delete the block of memory Pointed to by the ptr member of this

copy

```
X x1;
X x2;
x2= x1;
```

Java: Assignment Operator

- There are no pointers in Java (not in the same sense as C++)
- There is no need to specify copy assignment operators
- What does = between objects mean in Java?

```
Point p1 = new Point(3,4);
Point p2 = p1; //???
```

Example:

```
class User {
    public String name;
    public int age;
    public User( String str, int n ) { name = str; age = n; }
}

class Test {
    public static void main( String[] args )
    {
        User u1 = new User( "ariel", 112 );
        System.out.println( u1.name );        // ariel
        User u2 = u1;
        u2.name = "muriel";
        System.out.println( u1.name );        // muriel
    }
```

Copy the object reference in u1 into u2 so that u1 and u2 have references to the same object (i.e. the same memory)

mple:

```
class User {
    public String name;
    public int age;
    public User( String str, int n ) { name = str; age =
}

class Test {
    public static void main( String[] args )
    {
        User u1 = new User( "ariel", 112 );
        System.out.println( u1.name );        // ariel
        User u2 = u1;
        u2.name = "muriel";
        System.out.println( u1.name );        // muriel
    }
```

Changes to u2 are reflected in u1

Copy the object reference in u1 into u2 so that u1 and u2 have references to the same object (i.e. the same memory)

mple:

```
class User {
    public String name;
    public int age;
    public User( String str, int n ) { name = str; age =
}

class Test {
    public static void main( String[] args )
    {
        User u1 = new User( "ariel", 112 );
        System.out.println( u1.name );       // ariel
        User u2 = u1;
        u2.name = "muriel";
        System.out.println( u1.name );       // muriel
    }
```

Changes to u2 are reflected in u1

How could we have made u2 refer to a copy of the object u1 holds a reference to?

Overview
Standard C++ Library

Concepts
&
Examples

Question: Why should I write ***Stack***, ***List***, ***Set***, ***Sort***, ***Search, etc***?

Answer: don’t!!!

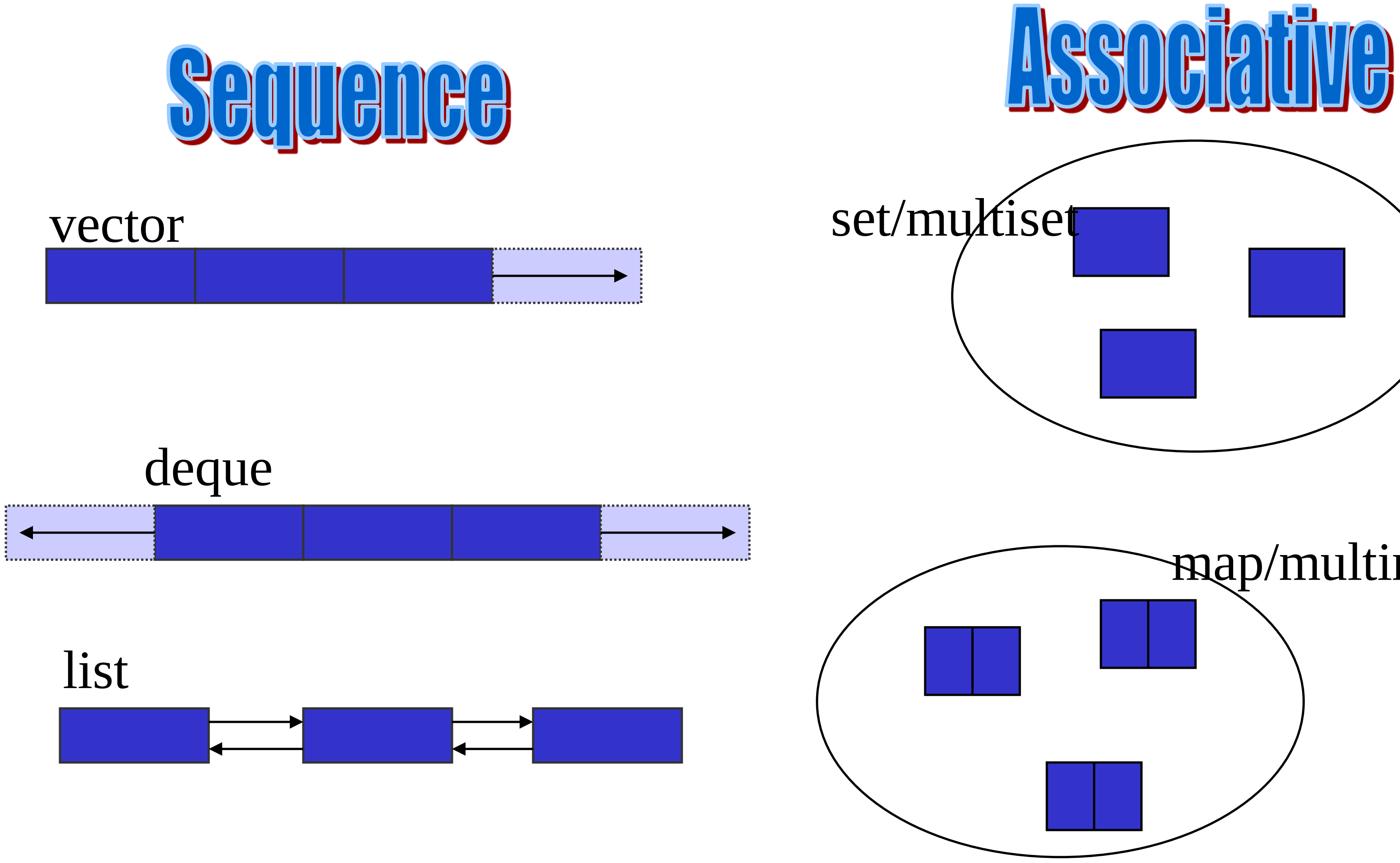
use the C++ Standard Library.

Heart of the library: STL

The standard C++ library has:

- ***Containers***
- ***Iterators*** to “run through” the containers
- ***Algorithms*** that work on containers: **find, sort, etc.**
- container ***adapters***: stack, queue, priority_queue
- iterator adapters
- strings

STL Containers



STL Containers

string

adapters:

stack

queue

priority_queue

containers include

- ***sequential*****:** position depends on time and place of insertion but not on the value of item -- vector, deque, list
- ***associative:*** position depends on its value -- set/multiset, map/multipmap
- ***adapters:*** stack, queue, priority_queue
- ***string***

containers

- grow as needed -- good to manage data, as used in typical programs:

  programs start with an empty container, read or compute data, store it, access it, store some more…
- built in arrays don't grow
- all 3 sequential containers have ***push_back***

Some functions for sequential containers:

- push_back(), pop_back()
- push_front(), pop_front()
- size()
- begin() -- points to first item in vector
- end() -- points <u>one past</u> end of vector
- iterator: pointer to an item in the vector

C++ strings are containers: better than C strings

- Memory managed for you:

  C++ ==> ***string s("dog")***

  C ==> ***s = new char[4]; strcpy(s, "dog")***

- easier to use

  C++ ==> ***s <= t***

  C ==> ***strcmp(s, t) <= 0***

- less error prone than C strings

Iterators

- step through a container, independent of internal structure
- small but common interface for any container
- every container has its own iterator
- interface for iterators almost the same as pointers: *, ++, --

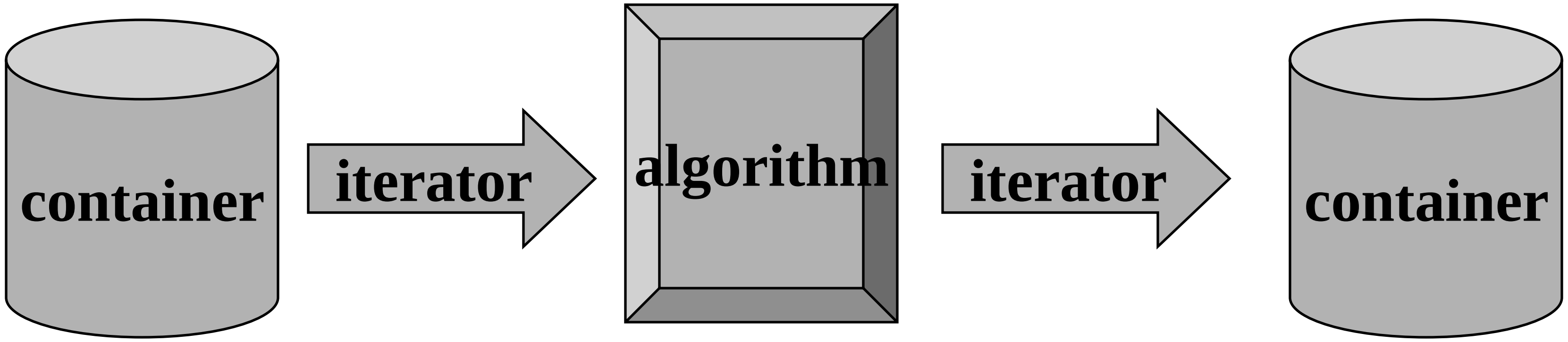
Operations

- operator *
- operator++ and operator--
- operators == and !=
- operator =
- begin()
- end() (past the end iterator)
- rbegin()
- rend()

const and non const iterators:

- ***const_iterator*** - items in the container may not be modified and must only be used with constant member functions
- ***iterator*** - permits modification of the items in the container
- ***reverse_iterator*** – backward traversal
- ***const_reverse_iterator***

Algorithms



Algorithms process the elements of a container
Algorithms use iterators

Separation of data and operations:
an algorithm can be written once and work with arbitrary containers because iterators have a common interface.

STL and OO

- OO says: encapsulate data and the operations on that data
- The STL contradicts OO philosophy
- reason: in principle, you can combine every kind of container with every kind of algorithm
- result: small, flexible framework that is type independent

Ranges

- all algorithms process a range
- the range might be the whole container, but not necessarily
- pass beginning and end of range as two separate arguments
- programmer must ensure that the range is valid
- passing an invalid range is undefined

open-ended range

- every algorithm processes a half-open range
- a range includes the beginning position but excludes the end position
- [begin, end)
- advantages:
  - simple end criterion for loops
  - avoids special handling for empty ranges

Some algorithms & ranges

```
list<int> mylist, secondlist;
vector<int> vec;
for (int I = 0; I < 100; ++I) mylist.push_back(I);

list<int>::const_iterator ptr =
      find(mylist.begin(), mylist.end(), 99);

if ( equal(mylist.begin(), mylist.end(), secondlist.begin()) ) {
   …
}
copy( mylist.begin, mylist.end(),       // source
      vec.begin() );                       // destination
```

More efficiency: value vs const/ref

```
int my_locate(const vector<int> & vec) {
    for (int i = 0; i < vec.size(); ++i) {
        if (vec[i] == 99 ) return vec[i];
    }
    return 0;
}

int your_locate(vector<int> vec) {
    for (int i = 0; i < vec.size(); ++i) {
        if (vec[i] == 99 ) return vec[i];
    }
    return 0;
}
```

Efficiency

- Hard to beat STL built in algorithms, like *find*, *copy*, iterator adapters
- Passing large structures by value is costly, difference would be more dramatic if vector contained large structure, or strings

Iterator adapters

- Iterators are pure abstractions
- STL provides several predefined iterators:
  - insert iterators
  - stream iterators
  - reverse iterators

Insert iterator

- Inserters allow algorithms to operate in ***insert*** mode, rather than ***overwrite*** mode
- solve a problem: allow algorithms to write to a destination that doesn’t have enough room – grow the destination
- three kinds that insert – at front, at end, or at a given location

```
int main() {
   list<int>        mylist;
   vector<int>      vec;
   deque<int> deck;
   set<int>         myset;

   for (int I = 1; I <= 10; ++I) {
      mylist.push_back(I);
   }
   copy(mylist.begin(), bylist.end(),   // source
             back_inserter(vec) );      // destination
   copy(mylist.begin(), mylist.end(),   // source
       front_inserter(deck);            // destination
   copy(mylist.begin(), mylist.end(),   // source
       inserter(myset, myset.begin() )) // destination
}
```

what does vec, deck & myset look like?

Stream iterators

- read from and write to a stream
- a stream is an object that represents I/O channels
- lets input from keyboard behave as a collection, from which you can read
- can redirect output to a file or screen

What does this code do?

```
int main() {
   vector<string> vec;
   copy ( istream_iterator<string>(cin),// start of source
          istream_iterator<string>(),    // end of source
          back_inserter(vec) );          // destination

   sort( vec.begin(), vec.end() );

   unique_copy(vec.begin(), vec.end(),  // source
       ostream_iterator<string>(cout, "\n") );  // destination
}
```

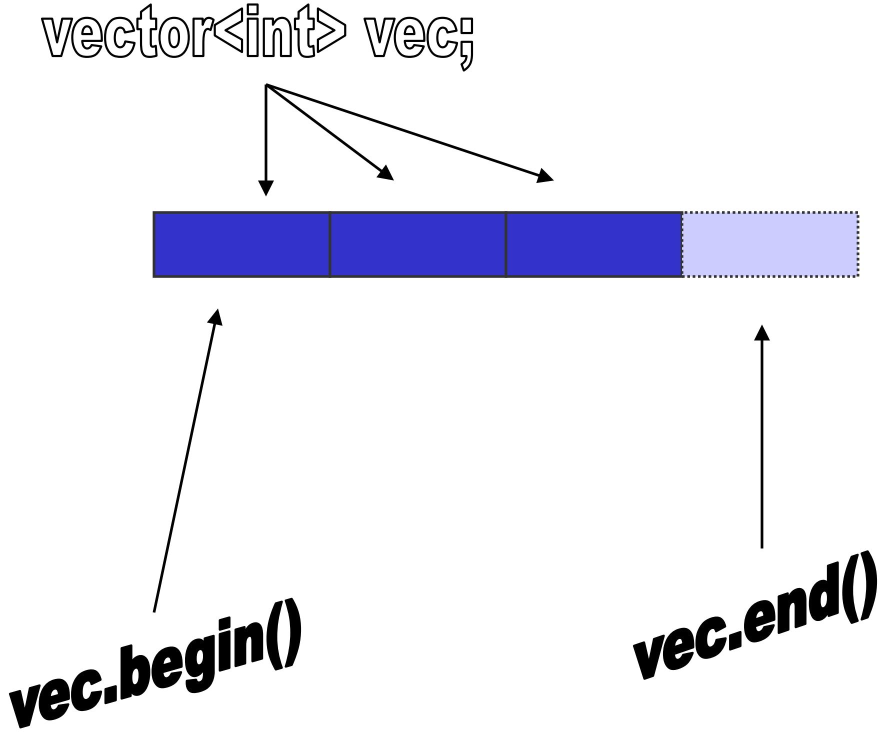
inserting & erasing vector/list elements

- insert(pos, elem) – inserts a copy of elem at position pos, returns position of new element
- insert(pos, n, elem) – inserts n copies of elem at pos (returns nothing)
- insert(pos, beg, end) – inserts, starting at pos, copy of elements [beg, end) – (returns nothing)
- erase(pos) -- removes the element at pos, returning position of next element
- erase(beg, end) – removes elements from [beg, end) & returns position of next element

An ordered collection: vector

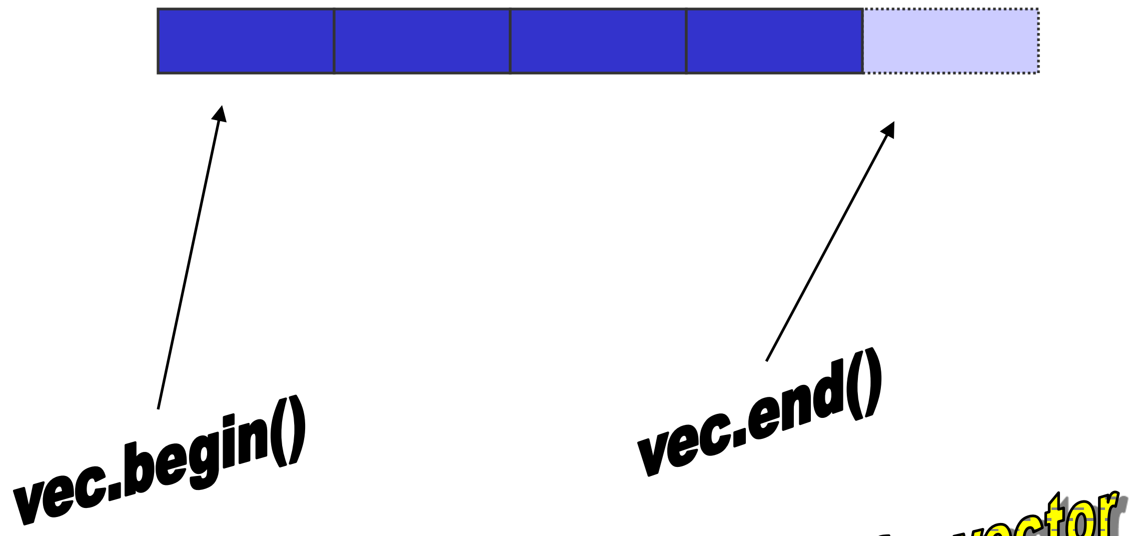
a random-access container
(not necessarily sorted)

A vector is a block of data with sub-blocks of size *type* for vector<*type*>:



push_back() adds a sub-block to the end of the block:

`vector<int> vec;`



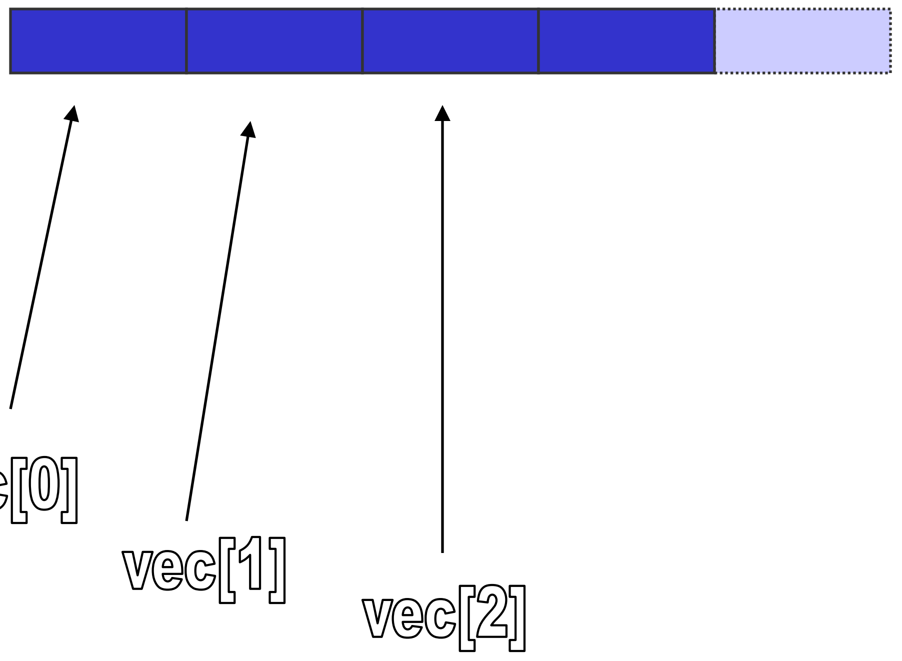
we can think of a vector as having infinite size

Vector efficiency:

- good performance for random access
- good performance when append/delete at end
- push_back() & pop_back() on average: O(1)
- push_front() and pop_front() are not supported
- insert(begin()) and delete(begin()) are O(n)
- if inserting or deleting from front or middle, switch to different container

random access of items: use array notation:

```
vector<int> vec;
```



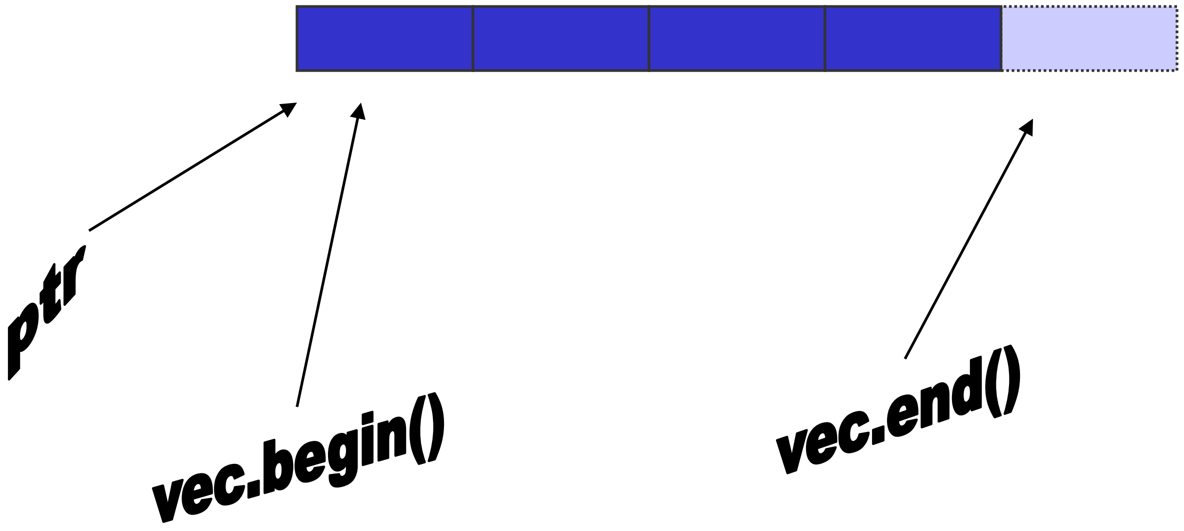
array notation with vectors:

- familiar
- only way to access non-sequentially
- but, library algorithms use iterators
- iterators can be written to work on any container – can then switch if necessary

An iterator is a pointer to a subblock:

```
vector<int> vec;
```

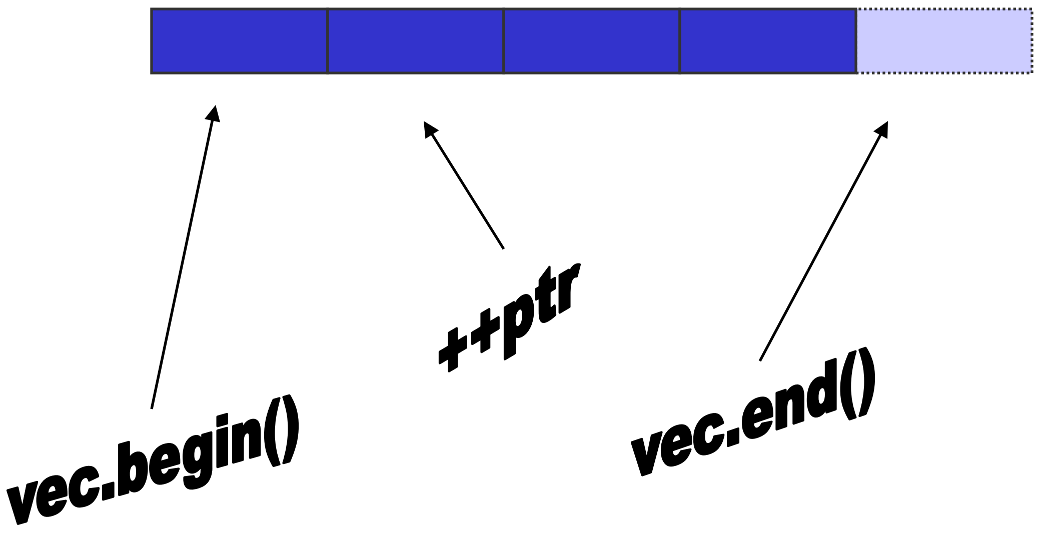
```
vector<int>::const_iterator  ptr = vec.begin();
```



We can move the iterator:

```
vector<int> vec;
```

```
vector<int>::const_iterator  ptr = vec.begin();
```



size/capacity

- size() – returns actual number in container
- capacity() – returns the current “amount of room”
- max_size() – max number of elements a container might contain; implementation defined
- empty()

13

Containers in Java

Based on a hierarchy of data types:

Collection, List, Set, Sorted Set

Map, Sorted Map

Arrays

- Array: Sequence of values of the same type
- Construct array:

```
new double[10]
```

- Store in variable of type `double[]`

```
double[] data = new double[10];
```

- When array is created, all values are initialized depending on array type:
  - Numbers: `0`
  - Boolean: *`false`*
  - Object References: *`null`*

Arrays

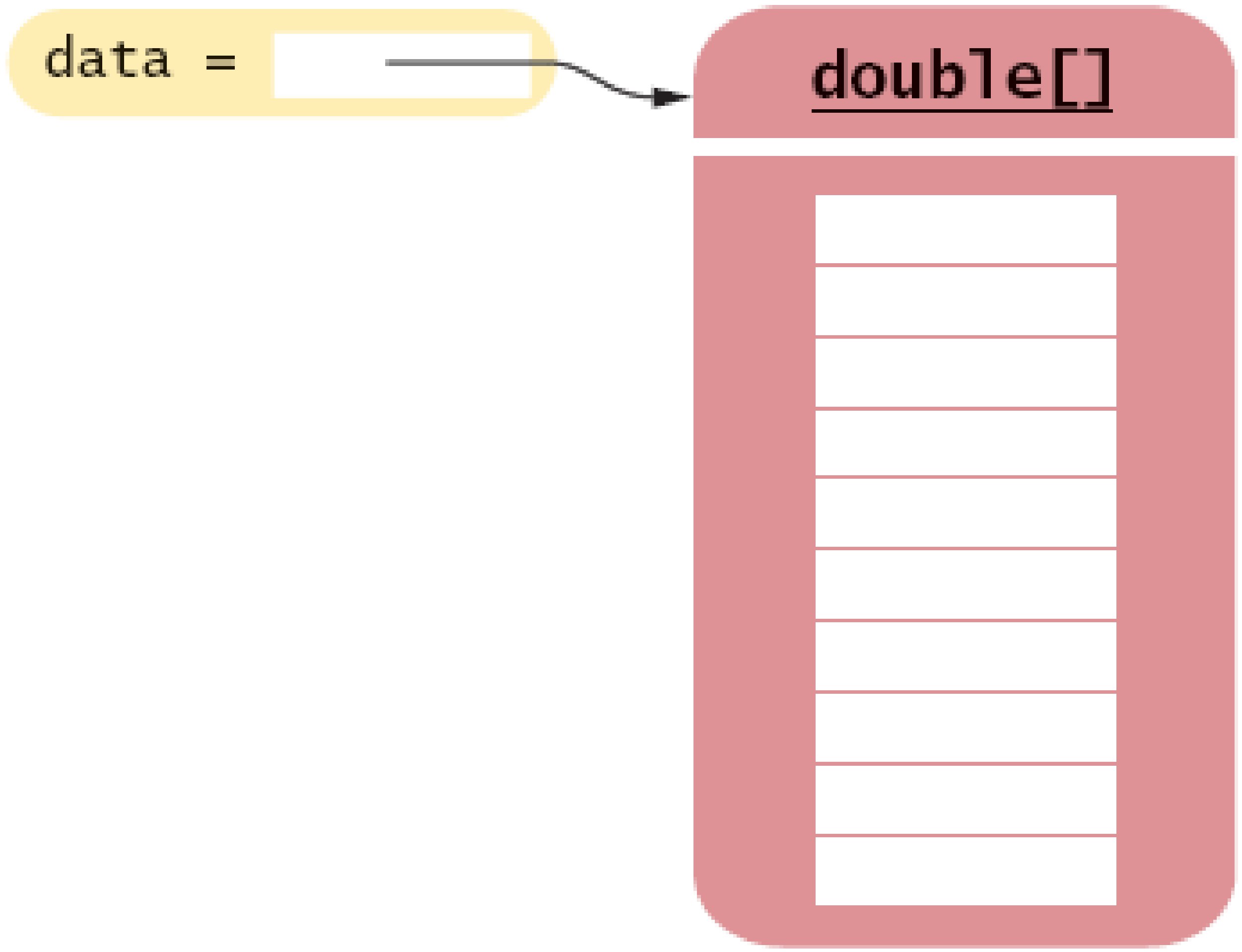


Figure 1 An Array Reference and an Array

Arrays

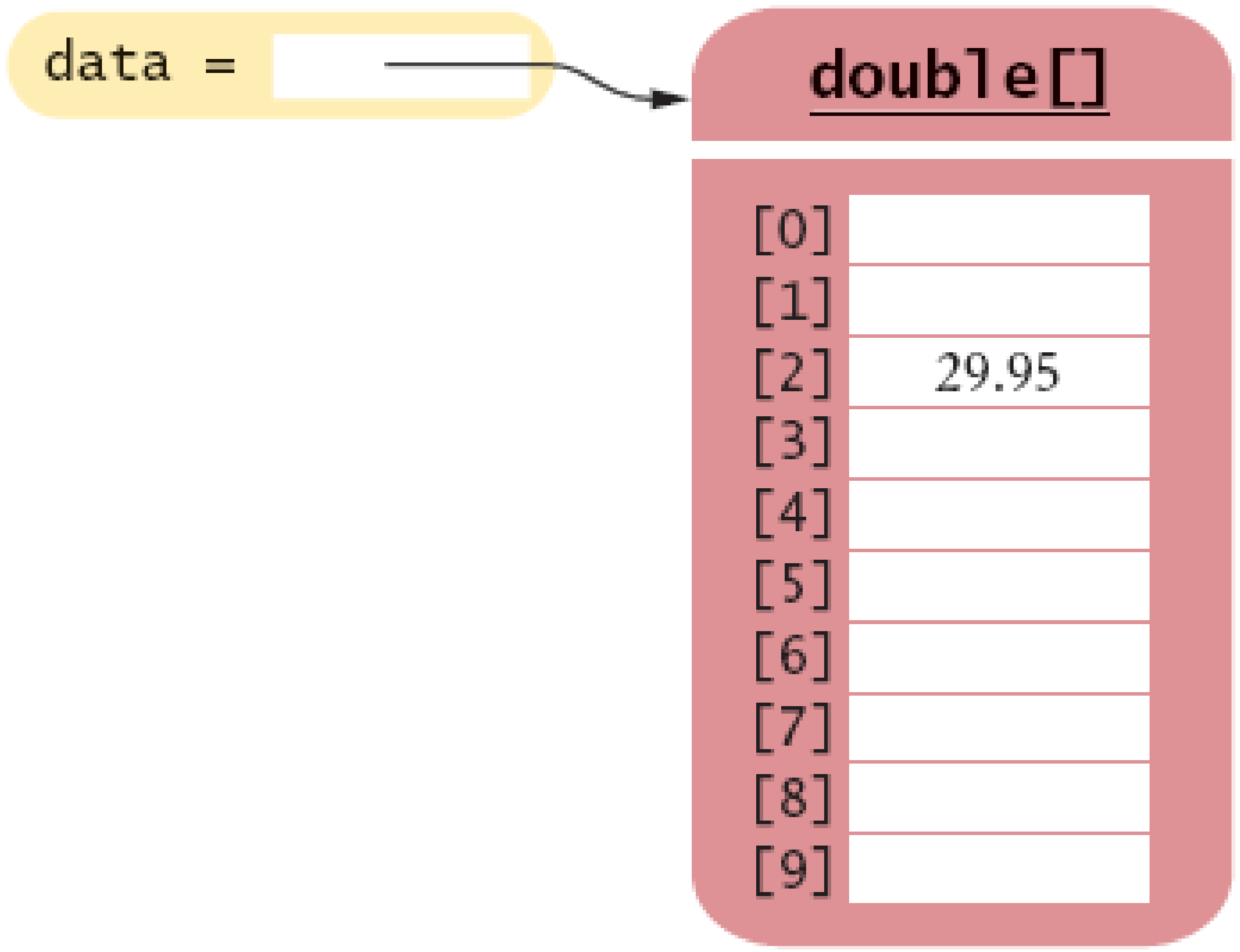


Figure 2 Storing a Value in an Array

Arrays

- Using the value stored:

```
System.out.println("The value of this data item is "
      + data[4]);
```

- Get array length as `data.length` (Not a method!)
- Index values range from `0` to `length - 1`
- Accessing a nonexistent element results in a bounds error

```
double[] data = new double[10];
data[10] = 29.95; // ERROR
```

- Limitation: Arrays have fixed length

Array Lists

- The `ArrayList` class manages a sequence of objects
- Can grow and shrink as needed
- `ArrayList` class supplies methods for many common tasks, such as inserting and removing elements
- The `ArrayList` class is a generic class: `ArrayList<T>` collects objects of type `T`:

```
ArrayList<BankAccount> accounts = new
       ArrayList<BankAccount>();
accounts.add(new BankAccount(1001));
accounts.add(new BankAccount(1015));
accounts.add(new BankAccount(1022));
```

- `size` method yields number of elements

Retrieving Array List Elements

- Use `get` method
- Index starts at 0
- `BankAccount anAccount = accounts.get(2); // gets the third element of the array list`
- Bounds error if index is out of range
- Most common bounds error:

```
int i = accounts.size();
anAccount = accounts.get(i); // Error
//legal index values are 0. . .i-1
```

Adding Elements

- `set` overwrites an existing value

```
BankAccount anAccount = new BankAccount(1729);
accounts.set(2, anAccount);
```

- `add` adds a new value before the index

```
accounts.add(i, a)
```

Continued

The Generalized `for` Loop

- Traverses all elements of a collection:

```
double[] data = . . .;
double sum = 0;
for (double e : data) // You should read this loop as
          "for each e in data"
{
   sum = sum + e;
}
```

- Traditional alternative:

```
double[] data = . . .;
double sum = 0;
for (int i = 0; i < data.length; i++)
{
   double e = data[i];
   sum = sum + e;
}
```

The Generalized `for` Loop

- Works for `ArrayLists` too:

```
ArrayList<BankAccount> accounts = . . . ;
double sum = 0;
for (BankAccount a : accounts)
{
   sum = sum + a.getBalance();
}
```

- Equivalent to the following ordinary `for` loop:

```
double sum = 0;
for (int i = 0; i < accounts.size(); i++)
{
   BankAccount a = accounts.get(i);
   sum = sum + a.getBalance();
}
```

Syntax 7.3 The "for each" Loop

```
for (Type variable : collection)
   statement
```

Example:

```
for (double e : data)
   sum = sum + e;
```

Purpose:

To execute a loop for each element in the collection. In each iteration, the variable is assigned the next element of the collection. Then the statement is executed.

ch07/bank/Bank.java

```java
import java.util.ArrayList;

/**
   This bank contains a collection of bank accounts.
*/
public class Bank
{
   /**
      Constructs a bank with no bank accounts.
   */
   public Bank()
   {
      accounts = new ArrayList<BankAccount>();
   }

   /**
      Adds an account to this bank.
      @param a the account to add
   */
   public void addAccount(BankAccount a)
   {
      accounts.add(a);
   }
```

Continued

ch07/bank/Bank.java (cont.)

```java

    /**
       Gets the sum of the balances of all accounts in this bank.
       @return the sum of the balances
    */
    public double getTotalBalance()
    {
       double total = 0;
       for (BankAccount a : accounts)
       {
          total = total + a.getBalance();
       }
       return total;
    }

    /**
       Counts the number of bank accounts whose balance is at
       least a given value.
       @param atLeast the balance required to count an account
       @return the number of accounts having least the given balance
    */
    public int count(double atLeast)
    {
```

Continued

ch07/bank/Bank.java (cont.)

```
      int matches = 0;
      for (BankAccount a : accounts)
      {
         if (a.getBalance() >= atLeast) matches++; // Found a match
      }
      return matches;
   }

   /**
      Finds a bank account with a given number.
      @param accountNumber the number to find
      @return the account with the given number, or null if there
      is no such account
   */
   public BankAccount find(int accountNumber)
   {
      for (BankAccount a : accounts)
      {
         if (a.getAccountNumber() == accountNumber) // Found a match
            return a;
      }
      return null; // No match in the entire array list
   }

```

Continued

ch07/bank/Bank.java (cont.)

```java
   /**
      Gets the bank account with the largest balance.
      @return the account with the largest balance, or null if the
      bank has no accounts
   */
   public BankAccount getMaximum()
   {
      if (accounts.size() == 0) return null;
      BankAccount largestYet = accounts.get(0);
      for (int i = 1; i < accounts.size(); i++)
      {
         BankAccount a = accounts.get(i);
         if (a.getBalance() > largestYet.getBalance())
            largestYet = a;
      }
      return largestYet;
   }

   private ArrayList<BankAccount> accounts;
}
```

ch07/bankBankTester.java

```
/**
   This program tests the Bank class.
*/
public class BankTester
{
   public static void main(String[] args)
   {
      Bank firstBankOfJava = new Bank();
      firstBankOfJava.addAccount(new BankAccount(1001, 20000));
      firstBankOfJava.addAccount(new BankAccount(1015, 10000));
      firstBankOfJava.addAccount(new BankAccount(1729, 15000));

      double threshold = 15000;
      int c = firstBankOfJava.count(threshold);
      System.out.println("Count: " + c);
      System.out.println("Expected: 2");

      int accountNumber = 1015;
      BankAccount a = firstBankOfJava.find(accountNumber);
      if (a == null)
```

Continued

ch07/bankBankTester.java (cont.)

```java
            System.out.println("No matching account");
         else
            System.out.println("Balance of matching account: " +
                  a.getBalance());
         System.out.println("Expected: 10000");

         BankAccount max = firstBankOfJava.getMaximum();
         System.out.println("Account with largest balance: "
               + max.getAccountNumber());
         System.out.println("Expected: 1001");
      }
   }
```

Output:

```
Count: 2
Expected: 2
Balance of matching account: 10000.0
Expected: 10000
Account with largest balance: 1001
Expected: 1001
```

Java Containers

- Map is a base class which stores <key, value> pairs with no duplicate keys allowed.
- Sorted Map Inherits from Map. SortedMap objects are map objects that store the pairs in key-sorted order
- Collection considers a container to be a group of objects, without any assumptions about the uniqueness of objects in the container. It is a base class declaring methods common to types List and Set. These include add, clear, IsEmpty, iterator, remove, removeAll, toArray (which constructs an array version of any container).
- Set inherits form Collection – all Set elements must be unique.
- SortedSet inherits from Set and stores elements in sorted order.

Note: These are all Java Interfaces, so they only contain method declarations

Example: Lists

```
import java.util.*;

class ListOps {
    public static void main( String[] args )
    {
        List animals = new ArrayList();
        animals.add( "cheetah" );
        animals.add( "lion" );
        animals.add( "cat" );
        animals.add( "fox" );
        animals.add( "cat" );          //duplicate cat
        System.out.println( animals );  //cheetah, lion, cat, fox, cat

        animals.remove( "lion" );
        System.out.println( animals );  //cheetah, cat, fox, cat

        animals.add( 0, "lion" );
        System.out.println( animals );  //lion, cheetah, cat, fox, cat
animals.add( 3, "racoon" );
System.out.println( animals );  //lion, cheetah, cat, racoon, fox, cat

animals.remove(3);
System.out.println( animals );  //lion, cheetah, cat, fox, cat

Collections.sort( animals );
System.out.println( animals );  //cat, cat, cheetah,fox, lion

List pets = new LinkedList();
pets.add( "cat" ); pets.add( "dog" ); pets.add( "bird" );
System.out.println( pets );     //cat, dog, bird

animals.addAll( 3, pets );
System.out.println( animals );  //cat, cat, cheetah, cat, dog, bird, fox, lion

ListIterator iter = animals.listIterator(); /* ListIterators can move in 2 directions whereas
      iterators only move in one direction*/
while ( iter.hasNext() ) {
          System.out.println( iter.next()  );
       }
    }
}
```

Using Linked Lists

- A linked list consists of a number of nodes, each of which has a reference to the next node
- Adding and removing elements in the middle of a linked list is efficient
- Visiting the elements of a linked list in sequential order is efficient
- Random access is not efficient

Inserting an Element into a Linked List

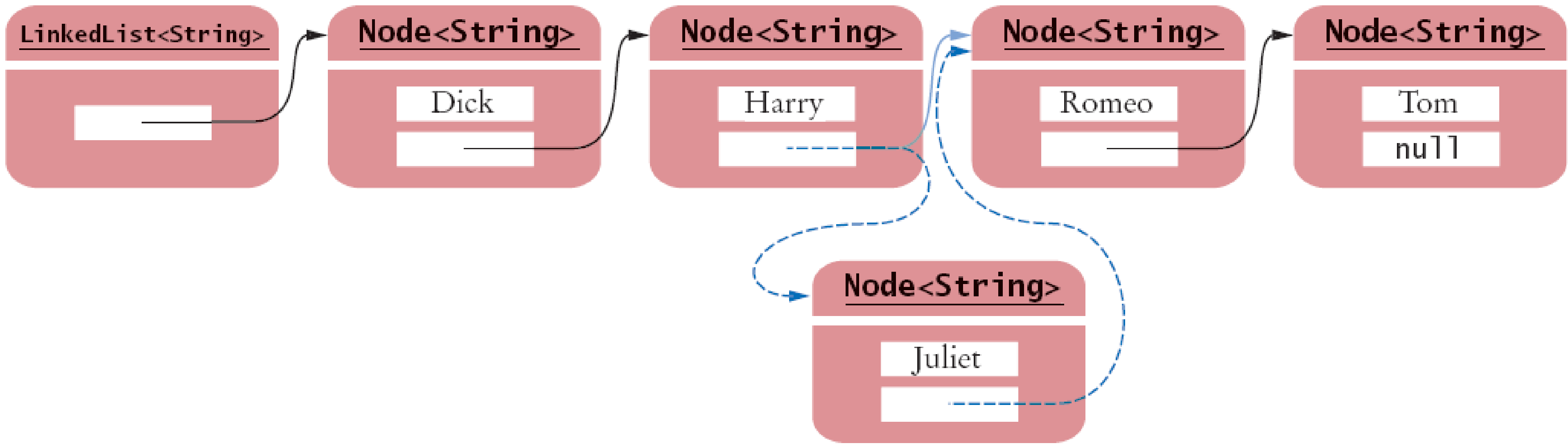


Figure 1 Inserting an Element into a Linked List

Java's `LinkedList` class

- Generic class
  - *Specify type of elements in angle brackets:* `LinkedList<Product>`
- Package: `java.util`
- Easy access to first and last elements with methods
  void addFirst(E obj)
  void addLast(E obj)
  E getFirst()
  E getLast()
  E removeFirst()
  E removeLast()

List Iterator

- `ListIterator` type
- Gives access to elements inside a linked list
- Encapsulates a position anywhere inside the linked list
- Protects the linked list while giving access

A List Iterator

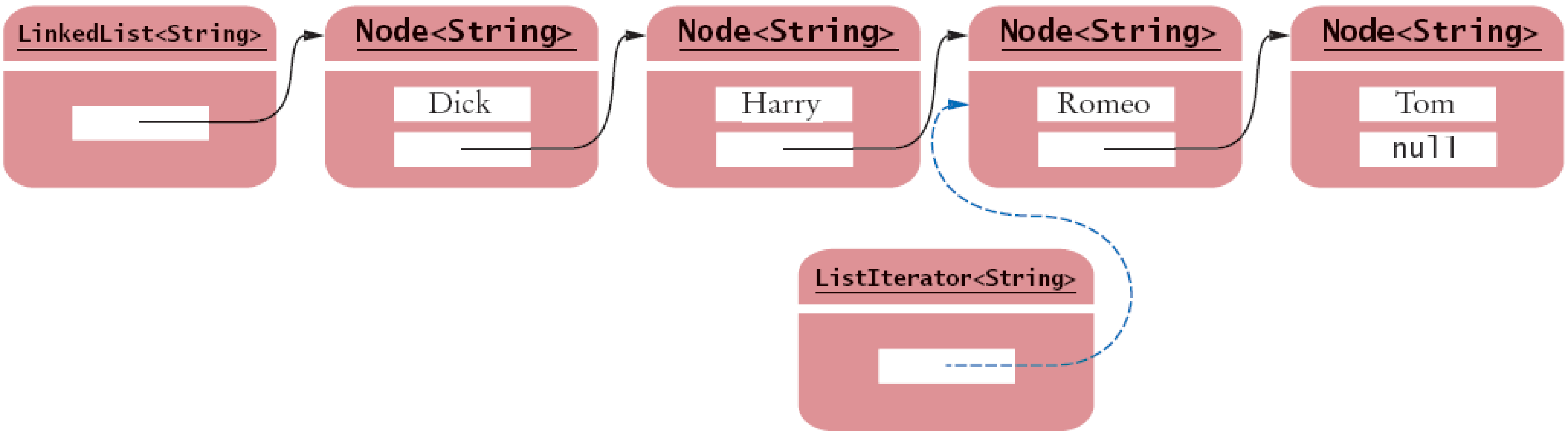


Figure 2 A List Iterator

A Conceptual View of the List Iterator

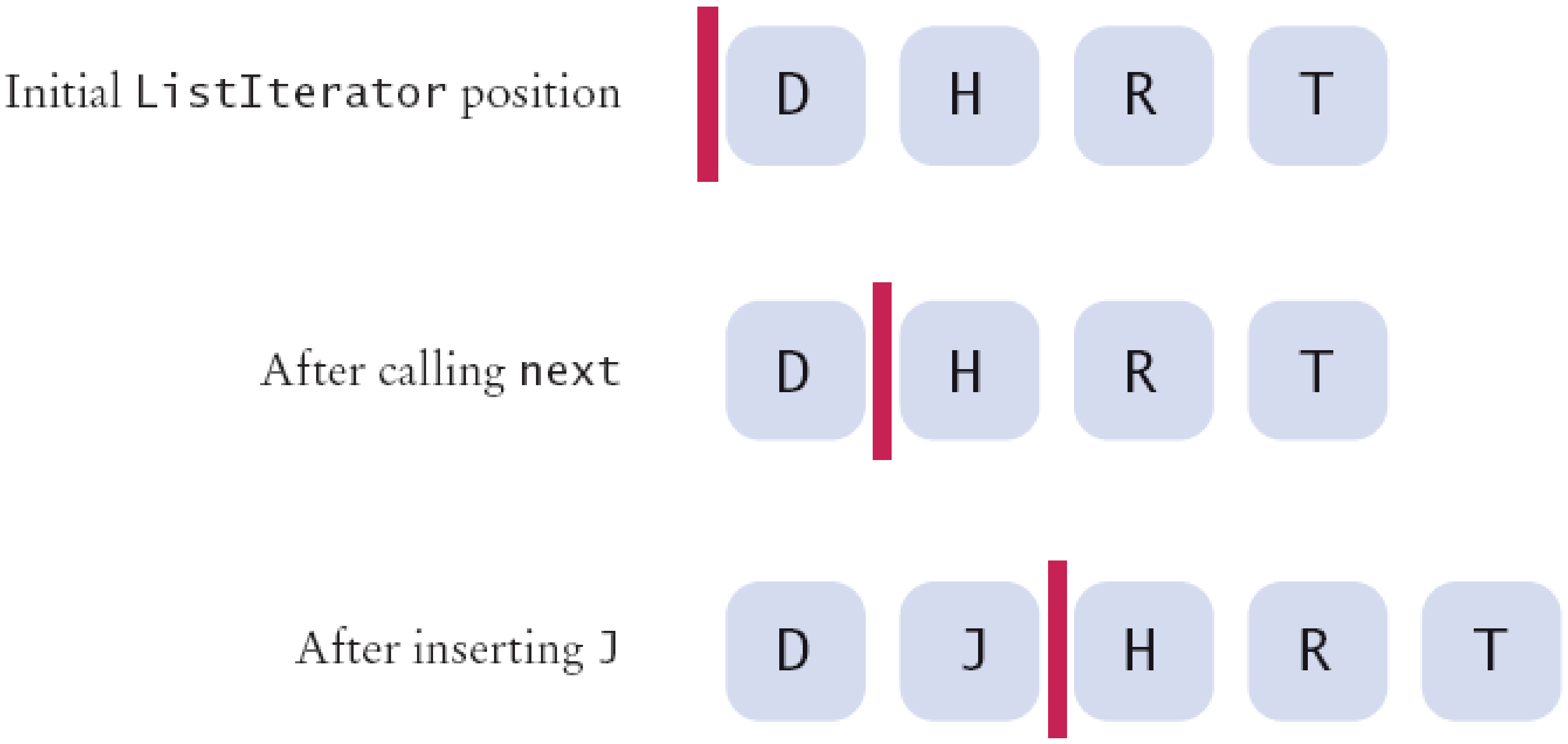


Figure 3 A Conceptual View of the List Iterator

List Iterator

- Think of an iterator as pointing between two elements
  - *Analogy: like the cursor in a word processor points between two characters*
- The `listIterator` method of the `LinkedList` class gets a list iterator

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iterator =
      employeeNames.listIterator();
```

List Iterator

- Initially, the iterator points before the first element
- The `next` method moves the iterator

```
iterator.next();
```

- `next` throws a `NoSuchElementException` if you are already past the end of the list
- `hasNext` returns true if there is a next element

```
if (iterator.hasNext())
iterator.next();
```

List Iterator

- The `next` method returns the element that the iterator is passing

```
while iterator.hasNext()
{
   String name = iterator.next();
   Do something with name
}
```

- Shorthand:

```
for (String name : employeeNames)
{
   Do something with name
}
```

Behind the scenes, the for loop uses an iterator to visit all list elements

List Iterator

- `LinkedList` is a *doubly linked list*
  - *Class stores two links:*
    - One to the next element, and
    - One to the previous element
- To move the list position backwards, use:
  - *`hasPrevious`*
  - *`previous`*

Adding and Removing from a LinkedList

- The `add` method:
  - *Adds an object after the iterator*
  - *Moves the iterator position past the new element*

```
iterator.add("Juliet");
```

Adding and Removing from a LinkedList

- The `remove` method
  - *Removes and*
  - *Returns the object that was returned by the last call to `next` or `previous`*

```
 //Remove all names that fulfill a certain condition
while (iterator.hasNext())
{
String name = iterator.next();
if (name fulfills condition)
   iterator.remove(); }
```

- Be careful when calling `remove`:
  - *It can be called only once after calling `next` or `previous`*
  - *You cannot call it immediately after a call to `add`*
  - *If you call it improperly, it throws an `IllegalStateException`*

Sample Program

- `ListTester` is a sample program that
    - *Inserts strings into a list*
    - *Iterates through the list, adding and removing elements*
    - *Prints the list*

ch15/uselist/ListTester.java

```java
import java.util.LinkedList;
import java.util.ListIterator;

/**
   A program that tests the LinkedList class
*/
public class ListTester
{
   public static void main(String[] args)
   {
      LinkedList<String> staff = new LinkedList<String>();
      staff.addLast("Dick");
      staff.addLast("Harry");
      staff.addLast("Romeo");
      staff.addLast("Tom");

      // | in the comments indicates the iterator position

      ListIterator<String> iterator
            = staff.listIterator(); // |DHRT
      iterator.next(); // D|HRT
      iterator.next(); // DH|RT
```

Continued

ch15/uselist/ListTester.java (cont.)

```java

      // Add more elements after second element

      iterator.add("Juliet"); // DHJ|RT
      iterator.add("Nina"); // DHJN|RT

      iterator.next(); // DHJNR|T

      // Remove last traversed element

      iterator.remove(); // DHJN|T

      // Print all elements

      for (String name : staff)
         System.out.print(iterator.next() + " ");
      System.out.println();
      System.out.println("Expected: Dick Harry Juliet Nina Tom");
   }
}
```

ch15/uselist/ListTester.java (cont.)

Output:

```
Dick Harry Juliet Nina Tom
Expected: Dick Harry Juliet Nina Tom
```

Sets

- Set: unordered collection of distinct elements
- Elements can be added, located, and removed
- Sets don't have duplicates

Sets

- We could use a linked list to implement a set
    - *Adding, removing, and containment testing would be relatively slow*
- There are data structures that can handle these operations much more quickly
    - *Hash tables*
    - *Trees*
- Standard Java library provides set implementations based on both data structures
    - *`HashSet`*
    - *`TreeSet`*
- Both of these data structures implement the `Set` interface

Iterator

- Use an iterator to visit all elements in a set
- A set iterator does not visit the elements in the order in which they were inserted
- An element can not be added to a set at an iterator position
- A set element can be removed at an iterator position

Code for Creating and Using a Hash Set

- `//Creating a hash set`
  `Set<String> names = new HashSet<String>();`
- `//Adding an element`
  `names.add("Romeo");`
- `//Removing an element`
  `names.remove("Juliet");`
- `//Is element in set`
  `if (names.contains("Juliet") { . . .}`

Listing All Elements with an Iterator

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
   String name = iter.next();
   Do something with name
}

// Or, using the "for each" loop
for (String name : names)
{
   Do something with name
}
```

ch16/set/SetDemo.java

```java
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;


/**
   This program demonstrates a set of strings. The user
   can add and remove strings.
*/
public class SetDemo
{
   public static void main(String[] args)
   {
      Set<String> names = new HashSet<String>();
      Scanner in = new Scanner(System.in);

      boolean done = false;
      while (!done)
      {
         System.out.print("Add name, Q when done: ");
         String input = in.next();
```

Continued

ch16/set/SetDemo.java (cont.)

```
            if (input.equalsIgnoreCase("Q"))
               done = true;
            else
            {
               names.add(input);
               print(names);
            }
         }

         done = false;
         while (!done)
         {
            System.out.print("Remove name, Q when done: ");
            String input = in.next();
            if (input.equalsIgnoreCase("Q"))
               done = true;
            else
            {
               names.remove(input);
               print(names);
            }
         }
      }
```

Continued

ch16/set/SetDemo.java (cont.)

```java

   /**
      Prints the contents of a set of strings.
      @param s a set of strings
   */
   private static void print(Set<String> s)
   {
      System.out.print("{ ");
      for (String element : s)
      {
         System.out.print(element);
         System.out.print(" ");
      }
      System.out.println("}");
   }
}


```

Continued

ch16/set/SetDemo.java (cont.)

Output:

```
     Add name, Q when done: Dick
     { Dick }
     Add name, Q when done: Tom
     { Tom Dick }
     Add name, Q when done: Harry
     { Harry Tom Dick }
     Add name, Q when done: Tom
     { Harry Tom Dick }
     Add name, Q when done: Q
     Remove name, Q when done:
Tom
     { Harry Dick }
     Remove name, Q when done:
Jerry
     { Harry Dick }
```

Self Check 16.1

Arrays and lists remember the order in which you added elements; sets do not. Why would you want to use a set instead of an array or list?

Answer: Efficient set implementations can quickly test whether a given element is a member of the set.

Self Check 16.2

Why are set iterators different from list iterators?

Answer: Sets do not have an ordering, so it doesn't make sense to add an element at a particular iterator position, or to traverse a set backwards.

Maps

- A map keeps associations between key and value objects
- Mathematically speaking, a map is a function from one set, the *key set*, to another set, the *value set*
- Every key in a map has a unique value
- A value may be associated with several keys
- Classes that implement the `Map` interface
  - *`HashMap`*
  - *`TreeMap`*

An Example of a Map

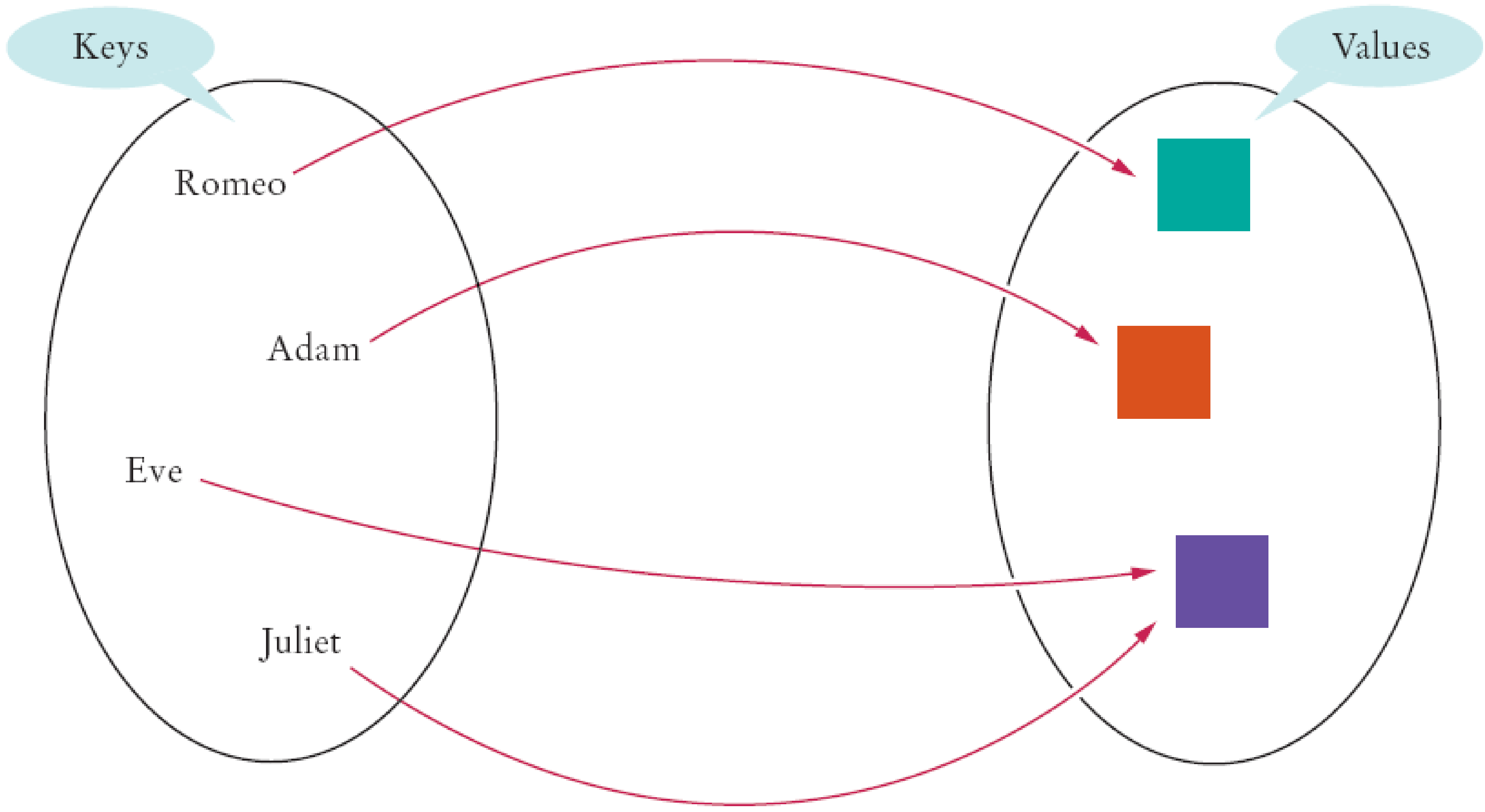


Figure 3 A Map

Code for Creating and Using a HashMap

-
  ```
  //Creating a HashMap
  Map<String, Color> favoriteColors = new HashMap<String,
        Color>();
  ```

-
  ```
  //Adding an association
  favoriteColors.put("Juliet", Color.PINK);
  ```

-
  ```
  //Changing an existing association
  favoriteColor.put("Juliet",Color.RED);
  ```

-
  ```
  //Getting the value associated with a key
  Color julietsFavoriteColor =
        favoriteColors.get("Juliet");
  ```

-
  ```
  //Removing a key and its associated value
  favoriteColors.remove("Juliet");
  ```

Printing Key/Value Pairs

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
   Color value = m.get(key);
   System.out.println(key + "->" + value);
}
```

ch16/map/MapDemo.java

```java
import java.awt.Color;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/**
   This program demonstrates a map that maps names to colors.
*/
public class MapDemo
{
   public static void main(String[] args)
   {
      Map<String, Color> favoriteColors
            = new HashMap<String, Color>();
      favoriteColors.put("Juliet", Color.PINK);
      favoriteColors.put("Romeo", Color.GREEN);
      favoriteColors.put("Adam", Color.BLUE);
      favoriteColors.put("Eve", Color.PINK);

```

***Continue
d***

ch16/map/MapDemo.java (cont.)

```
        Set<String> keySet = favoriteColors.keySet();
        for (String key : keySet)
        {
           Color value = favoriteColors.get(key);
           System.out.println(key + "->" + value);
        }
    }
}
```

Continued

ch16/map/MapDemo.java (cont.)

Output:

```
Romeo->java.awt.Color[r=0,g=255,b=0]
Eve->java.awt.Color[r=255,g=175,b=175]
Adam->java.awt.Color[r=0,g=0,b=255]
Juliet->java.awt.Color[r=255,g=175,b=175]
```

Java Containers:Vectors

- The Vector class dates back to early Java releases.
- It has now been retro-fitted to implement the List interface to enable older Java code to run on newer platforms.
- Vectors have all of the nmethods listed in the List interface …
  - Plus a few more e.g. addElement, size, elementAt(i)

- Examples in the folder called Java Container Examples

Java Containers Algorithms

- The java.util.Collections class provides some predefined algorithms for activities such as
  - sorting, searching, copying, filling, finding maximum values in a container, finding minimum values in a container, reversing the contents of a container, shuffling the container contents etc…

Java API at
http://java.sun.com/j2se/1.5.0/docs/api/