
C++ & Object-Oriented Programming

David Vernon

C++ & Object-Oriented Programming

Based on

L. Ammeraal, *C++ For Programmers*, Wiley, 1994.

Abstract Data Types

ADTs are an old concept

Specify the complete set of values which a variable of this *type* may assume

Specify completely the set of all possible operations which can be applied to values of this *type*

ISET: An ADT specification of a set type

ISET

define a set type which can represent integer values

Sets can have more than one element

variables of type ISET can represent several integer values simultaneously

Sets have a number of basic operations, such as union, intersection, membership, ...

ISSET: An ADT specification of a set type

Let **S** denote all possible values of type ISSET (i.e. sets of *integers*)

Let **e** denote all possible values of type *integer*

Let **B** denote the set of Boolean values *true* and *false*

ISSET Operations

Syntax of ADT Operation:

Operation:

WhatYouPassIt → **WhatItReturns** :

ISSET Operations

Declare: $\rightarrow S$:

The function value of *Declare*(*S*) is an empty set

alternative syntax: *ISSET S*

ISSET Operations

Add: $e \times S \rightarrow S$:

The function value of $Add(e, S)$ is a set with the element e added

alternative syntax: $S2 = S1 + e$

ISSET Operations

Remove: $e \times S \rightarrow S$:

The function value of *Remove*(e, S) is a set with the element e removed

alternative syntax: $S2 = S1 - e$

ISSET Operations

Inclusion: $e \times S \rightarrow B$:

The function value of $In(e, S)$ is a Boolean value

True if e is an element of S

False if e is not an element of S

alternative syntax: $e ? S$

ISSET Operations

Assignment: $S \rightarrow S :$

The function value of $Assign(S1, S2)$ is a set with membership equal to S

alternative syntax: $S2 = S1$

ISSET Operations

Union: $S \times S \rightarrow S$:

The function value of $Union(S1, S2)$ is a set with members those elements which are either in $S1$ or $S2$, or both

alternative syntax: $S3 = S1 + S2$

ISSET Operations

Intersection: $S \times S \rightarrow S$:

The function value of *Intersection*(S1, S2) is a set with members those element which are in both S1 and S2

alternative syntax: $S3 = S1 . S2$

ISSET Operations

Cardinal: $S \rightarrow e$:

The function value of $Cardinal(S)$ is an integer value equal to the number of elements n set S

ADT Specification

The key idea is that we have not specified how the sets are to be implemented, merely their values and the operations to which they can be operands

This 'old' idea of data abstraction is one of the key features of object-oriented programming

C++ is a particular implementation of this object-oriented methodology

C++ Overview

Designed by B. Stroustrup (1986)

C++ and ANSI C (revised version of K&R C) are closely related

Hybrid language: OO and 'conventional' programming

More than just an OO version of C

Simple C++ Program

```
/* Example1: Compute the squares of both the sum and the
   difference of two given integers
*/
#include <iostream.h>
int main()
{
    cout << "Enter two integers: "; // Display
    int a, b;                       // request
    cin >> a >> b;                  // Reads a and b
    int sum = a + b, diff = a - b,
        u = sum * sum, v = diff * diff;
    cout << "Square of sum          : " << u << endl;
    cout << "Square of difference: " << v << endl;
    return 0;
}
```

Key Points

`/* */`

begin and end of a comment

`//`

beginning of a comment (ended by end of line)

`#include <iostream.h>`

Includes the file `iostream.h`, a header file for stream input and output, e.g. the `<<` and `>>` operators

To include means to replace the include statement with the contents of the file

must be on a line of its own

Key Points

In general, statements can be split over several lines
Every C++ program contains one or more functions,
one of which is called **main**

```
int main()          // no parameters here
{                  // beginning of body
    ...
}                  // end of body
```

A function comprises statements which are
terminated with a semi-colon

Key Points

Declaration

Unlike C, a declaration is a normal statement and can occur anywhere in the function

```
int      sum = a + b, diff = a - b,  
        u = sum * sum, v = diff * diff;
```

Declarations define variables and give them a type
Optionally, declarations initialize variables

Key Points

Output to the 'standard output stream'

<<

Input from the 'standard input stream'

>>

Output of the end of a line is effected using the `endl` keyword

Could also have used '`\n`' or "`\n`"

Identifiers

Sequence of characters in which only letters, digits, and underscore _ may occur

Case sensitive ... upper and lower case letters are different

Identifiers

Reserved identifiers (keywords):

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

Constants

Integer constants

123 (decimal)

0777 (octal)

0xFF3A (hexadecimal)

123L (decimal, long)

12U (decimal, unsigned)

Constants

Character constants

'A' enclosed in single quotes

Special characters (escape sequences)

'\n' newline, go to the beginning of the next line

'\r' carriage return, back to the beginning the current line

'\t' horizontal tab

'\v' vertical tab

'\b' backspace

'\f' form feed

'\a' audible alert

Constants

Character constants

<code>'\'</code>	backslash
<code>'\''</code>	single quote
<code>'\"'</code>	double quote
<code>'\?'</code>	question mark
<code>'\000'</code>	octal number
<code>'\xhh'</code>	hex number

Constants

Floating Constants

Type **double**

82.247

.63

83.

47e-4

1.25E7

61.e+4

Type **float**

82.247L

.63l

Constants

Floating Constants

Type	Number of Bytes
float	4
double	8
long double	10

Implementation dependent

Constants

String Constants

String literal

String

`"How many numbers?"`

`"a"`

`"a"` is not the same as `'a'`

A string is an array of characters terminated by the escape sequence `'\0'`

Other escape sequences can be used in string literals, e.g.

`"How many\nnumbers?"`

Constants

String Constants

Concatenation of string constants

`"How many numbers?"`

is equivalent to

`"How many"`
`" numbers?"`

This is new to C++ and ANSI C

Constants

String Constants

```
cout << "This is a string that is \  
regarded as being on one line";
```

is equivalent to

```
cout << "This is a string that is"  
      "regarded as being on one line";
```

Comments

```
/* text of comment */
```

```
// text of comment
```

Within a comment, the characters sequences `/*`, `*/`, and `//` have no meaning

So comments cannot be nested

Use

```
#if 0
```

```
code fragment to be commented out
```

```
...
```

```
#endif
```

Exercises

1. Write a program that prints your name and address. Compile and run this program
2. Write a program that prints what will be your age at the end of the year. The program should request you to enter both the current year and the year of your birth
3. Modify the program to print also your age at the end of the millenium

Exercises

4. Use the operator `<<` only once to print the following three lines:

One double quote: `"`

Two double quotes: `""`

Backslash: `\`

Exercises

5. Correct the errors in the following program

```
include <iostream.h>
int main();
{
    int i, j
    i = 'A';
    j = "B";
    i = 'C' + 1;
    cout >> "End of program";
    return 0
}
```

Expressions and Statements

Expressions

$a + b$

$x = p + q * r$

Statements

$a + b;$

$x = p + q * r;$

Operators

$+, *, =$

Operands

a, b, p, q, r, x

Arithmetic Operations

Unary operator: -, +

```
neg = -epsilon;
```

```
pos = +epsilon;
```

Arithmetic Operations

Binary operators: $+$, $-$, $*$, $/$, $\%$

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

Integer overflow is not detected

Results of division depends on the types of the operands

```
float fa = 1.0, fb = 3.0;
```

```
int a = 1, b = 3;
```

```
cout << fa/fb;
```

```
cout << a/b;
```

Arithmetic Operations

Remainder on integer division

$\%$

`39 % 5 // value of this expression?`

Arithmetic Operations

Assignment and addition

`x = x + a`

`x += a`

These are expressions and yield a value as well as performing an assignment

`y = 3 * (x += a) + 2; //!!!`

Arithmetic Operations

Other assignment operators

`x -= a`

`x *= a`

`x /= a`

`x %= a`

`++i` `// increment operator: i += 1`

`--i` `// decrement operator: i -= 1`

Arithmetic Operations

Other assignment operators

```
/* value of expression = new value of i */
```

```
++i      // increment operator: i += 1
```

```
--i      // decrement operator: i -= 1
```

```
/* value of expression = old value of i */
```

```
i++      // increment operator: i += 1
```

```
i--      // decrement operator: i -= 1
```

Types, Variables, and Assignments

Type	Number of Bytes
char	1
short (short int)	2
int	2
enum	2
long (long int)	4
float	4
double	8
long double	10

Types, Variables, and Assignments

Use `sizeof` to find the size of a type

e.g.

```
cout << sizeof (double)
```


Types, Variables, and Assignments

<< doesn't allow user-specified formatting of output; use (C library function) `printf`

```
char ch = 'A'; int i = 0;  
float f = 1.1; double ff = 3.14159;  
  
printf("ch = %c, i = %d\n", ch, i);  
printf("f = %10f, ff = %20.15f\n", f, ff);
```

Types, Variables, and Assignments

To use `printf` you must include `stdio.h`

```
#include <stdio.h>
```

syntax:

```
printf(<format string>, <list of variables>);
```

```
<format string>
```

String containing text to be printed and conversion specifications

Types, Variables, and Assignments

Conversion specifications

<code>%c</code>	characters
<code>%d</code>	decimals
<code>%f</code>	floats or doubles
<code>%s</code>	strings

can also include field width specifications:

<code>%m.kf</code>	m is the field width k is the number of digits after the decimal point
--------------------	--

Types, Variables, and Assignments

>> doesn't allow user-specification of input types; use (C library function) `scanf`

```
char ch = 'A'; int i = 0;  
float f = 1.1; double ff = 3.14159;  
  
scanf("%c %d %f %lf", &ch, &i, &f, &ff);
```

The ampersand `&` is essential

It takes the address of the variable that follows
`scanf` expects only variables

Types, Variables, and Assignments

Enumerated types enum

Used to define constant values whose names mean something but whose actual values are irrelevant

```
enum days
{ Sunday, Monday, Tuesday, Wednesday,
  Thursday, Friday, Saturday
} yesterday, today, tomorrow;
days the_day_after_tomorrow;
```

Sunday, ..., Saturday are symbolic integer constants, have values 0, .., 6, respectively and are the values of type days

```
scanf("%c %d %f %lf", &ch, &i, &f, &ff);
```

The ampersand & is essential

It takes the address of the variable that follows

Types, Variables, and Assignments

Enumerated types example

```
today = Monday;  
the_day_after_tomorrow = Tuesday;
```

C++ has no built-in logical or Boolean type

We can define one using enumerated types

```
enum Boolean {FALSE, TRUE};
```

Types, Variables, and Assignments

Register variables

access to data in registers is generally faster than access to data in memory

We can ask to compiler to put very frequently used variables in a register:

```
register int i;
```

Cannot take the address of a register variable

```
scanf("%d", &i); // illegal operation
```

Types, Variables, and Assignments

Use the type qualifier `const` to define constants

```
const int weeklength = 7;
```

The initialization of `weeklength` is essential since we cannot assign values to constants subsequently

```
weeklength = 7; // Error
```


Comparison and Logical Operators

Operator	Meaning
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to
&&	logical AND
	logical OR
!	logical NOT

Comparison and Logical Operators

<, >, <=, >= are relational operators

== and != are equality operators

relational operators have a higher precedence than equality operators

Expression formed with these operators yield one of two possible values

0 means false

1 means true

Both are of type int

Compound Statement

Statements describe actions

Expressions yield values

We use braces `{}` to build complex - compound - statement from simpler ones

Typically, we use compound statements in places where the syntax allows only one statement

```
{ x = a + b; y = a - b; }
```

Compound Statement

Compound statements are called blocks

A declaration in a block is valid from the point of declaration until the closing brace of the block

The portion of the program corresponding to this validity is called the scope of the variable which has been declared

Variables are only visible in their scope

Compound Statement

```
// SCOPE: Illustration of scope and visibility
#include <iostream.h>
int main()
{ float x = 3.4;
  { cout << "x = " << x << endl;
    // output: x = 3.4 (because float x is visible
    int x = 7;
    cout << "x = " << x << endl;
    // output x = 7 (because int x is visible
    // float x is still in scope but hidden
    char x = 'A';
    cout << "x = " << x << endl;
    // output x = A (because char x is visible
    // float x and int x are still in scope but hidden
  } // end of block
```

Compound Statement

```
cout << "x = " << x << endl;  
// output x = 3.4 (because char x is visible  
// int x and char x are out of scope  
return 0;  
} // end of main
```

Conditional Statements

Syntax

```
if (expression)  
    statement1  
else  
    statement2
```

The else clause is optional

Semantics

`statement1` is executed if the value of `expression` is non-zero

`statement2` is executed if the value of `expression` is zero

Conditional Statements

Where appropriate *statement1* and *statement2* can be compound statements

```
if (a >= b)
{
    x = 0;
    if (a >= b+1)
    {
        xx = 0;
        yy = -1;
    }
    else
    {
        xx = 100;
        yy = 200;
    }
}
```


Iteration Statements

while-statement syntax

```
while (expression)  
    statement
```

semantics

`statement` is executed (repeatedly) as long as `expression` is non-zero (true)

`expression` is evaluated before entry to the loop

Iteration Statements

```
// compute  $s = 1 + 2 + \dots + n$ 
```

```
s = 0;
```

```
i = 1;
```

```
while (i <= n)
```

```
{   s += i;
```

```
    i++;
```

```
}
```

Iteration Statements

do-statement syntax

do

statement

while (*expression*);

semantics

statement is executed (repeatedly) as long as *expression* is non-zero (true)

expression is evaluated after entry to the loop

Iteration Statements

```
// compute  $s = 1 + 2 + \dots + n$   
  
s = 0;  
i = 1;  
do                                // incorrect if  $n == 0$   
{  s += i;  
    i++;  
} while (i <= n)
```

Iteration Statements

for-statement

```
for (statement1 expression2; expression3)  
    statement2
```

semantics

statement1 is executed

statement2 is executed (repeatedly) as long as *expression2* is true (non-zero)

expression3 is executed after each iteration (i.e. after each execution of *statement2*)

expression2 is evaluated before entry to the loop

Iteration Statements

```
// compute  $s = 1 + 2 + \dots + n$ 
```

```
s = 0;
```

```
for (i = 1; i <= n; i++)
```

```
    s += i;
```

Iteration Statements

```
for (statement1 expression2; expression3)  
    statement2
```

We have *statement1* rather than *expression1* as it allows us to use an initialized declaration

```
int i=0;
```

Note that the for statement does not cause the beginning of a new block (and scope) so we can only declare a variable which has not already been declared in that scope.

The scope of the declaration ends at the next }

Iteration Statements

```
// compute  $s = 1 + 2 + \dots + n$ 
```

```
s = 0;
```

```
for (int i = 1; i <= n; i++)
```

```
    s += i;
```


Break and Continue

`break;`

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

Break and Continue

```
// example of the break statement
```

```
for (int i = 1; i <= n; i++)  
{  s += i;  
    if (s > max_int)  // terminate loop if  
        break;        // maximum sum reached  
}
```

```
/* Note:  there is a much better way */  
/* to write this code                  */
```

Break and Continue

`continue;`

the continue statement causes an immediate jump to the text for continuation of the (smallest enclosing) loop.

Break and Continue

```
// example of the continue statement
```

```
for (int i = 1; i <= n; i++)  
{  s += i;  
    if ((i % 10) != 0)    // print sum every  
        continue;        // tenth iteration  
    cout << s;  
}
```

```
/* Note:  there is a much better way */  
/* to write this code                  */
```

Switch

`switch (expression) statement`

the `switch` statement causes an immediate jump to the statement whose label matches the value of `expression`

`statement` is normally a compound statement with several statements and several labels

`expression` must be of type `int`, `char`, or `enum`

Switch

```
// example of the switch statement
```

```
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

```
// example of the switch statement
```

```
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;
}
```

Exercises

6. Write a program that reads 20 integers and counts how often a larger integer is immediately followed by a smaller one

Conditional Expressions

conditional expression syntax

expression1 ? *expression2* : *expression3*

semantics

if the value of *expression1* is true (non-zero)

then *expression2* is evaluated and this is the value of the entire conditional expression

otherwise *expression3* is evaluated and this is the value of the entire conditional expression

conditional expression

```
// example of the conditional expression
```

```
z = 3 * (a < b ? a + 1 : b - 1) + 2;
```

```
// alternative
```

```
if (a < b)
```

```
    z = 3 * (a + 1) + 2;
```

```
else
```

```
    z = 3 * (b - 1) + 2;
```

conditional expression

```
// example of the conditional expression
```

```
cout << "The greater of a and b is" <<  
      (a > b ? a : b);
```

```
// alternative
```

```
cout << "The greater of a and b is"  
if (a < b)  
    cout << a;  
else  
    cout << b;
```

The Comma-operator

comma-operator syntax

expression1 , *expression2*

semantics

expression1 and *expression2* are evaluated in turn and
the value of the entire (compound)
expression is equal to the value of *expression2*

The Comma-operator

```
// example of the comma operator  
// compute sum of input numbers
```

```
s = 0;  
while (cin >> i, i > 0)  
    s += i;
```

```
// or ...  
s = 0;  
while (scanf ("%d", &i), i > 0)  
    s += i;
```

The Comma-operator

```
// Note that here scanf() is used as an  
// expression and yields a value ... the  
// number of successfully-read arguments
```

```
s = 0;  
while (scanf ("%d", &i) == 1) // terminate on  
    s += i;                  // non-integer  
                             // input
```

Bit Manipulation

The following bit manipulation operators can be applied to integer operands:

&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
~	Inversion of all bits
<<	Shift left
>>	Shift right

Note, in C++, the meaning of an operator depends on the nature of its operands (cf &, <<, >>)

Simple Arrays

The array declaration

```
int a[100]
```

enables us to use the following variables:

```
a[0], a[1], ... a[99]
```

each element being of type `int`

Simple Arrays

subscripts can be an integer expression with value less than the array size (e.g. 100)

In the declaration, the dimension must be a constant expression

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

Simple Arrays

Alternatively

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

Simple Arrays

```
// LIFO: This program reads 30 integers and
// prints them out in reverse order: Last In, First Out
#include <iostream.h>
#include <iomanip.h>

int main()
{ const int LENGTH = 30;
  int i, a[LENGTH];
  cout << "Enter " << LENGTH << " integers:\n";
  for (i=0; i<LENGTH; i++) cin >> a[i];
  cout << "\nThe same integers in reverse order:\n";
  for (i=0; i<LENGTH; i++)
    cout << setw(6) << a[LENGTH - i - 1]
        << (i % 10 == 9 ? '\n' : ' ');
  return 0;
}
```

Simple Arrays

Initializing an array

```
const int LENGTH = 4;
...
int a[LENGTH] = {34, 22, 11, 10};

int b[LENGTH] = {34, 22}; // element 2, 3 = 0

int c[20] = "Tim";
        // same as = { 'T', 'i', 'm', '\0' }
```

Associativity

Most operators are left-associative

$a - b * c$ $//$ $((a - b) * c)$
 $//$ or $(a - (b * c))$

Right-associative operators

all unary operators

the operator `?:`, used in expressions

the assignment operators

`=`, `+=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`

Associativity

example

`-n++` `//` value for `n=1`? `-2` or `0`

Precedence of Operators

operators in order of decreasing precedence (same precedence for same line)

```
() [] . -> ::  
! ~ ++ + - (type) * & sizeof new delete // all unary  
. * -> *  
* / %  
+ -  
<< >>  
< <= > >=  
== !=
```

Precedence of Operators

&

^

|

& &

| |

? :

= += -= *= /= %= &= |= ^= <<= >>=

,

Precedence of Operators

Operator	Meaning
::	scope resolution
()	function calls
[]	subscripting
.	selecting a component of a structure
->	selecting a component of a structure by means of a pointer
.*	pointers to class members
->*	pointers to class members
!	NOT, unary operator
~	inversion of all bits, unary operator

Precedence of Operators

Operator	Meaning
++	increment, unary operator
--	decrement, unary operator
+	plus, unary operator
+	addition, binary operator
-	minus, unary operator
-	minus, binary operator
(type)	cast, unary operator
new	create (allocate memory)
delete	delete (free memory)
*	'contents of address', unary operator
*	multiplication, binary operator

Precedence of Operators

Operator	Meaning
&	bitwise AND, binary operator
&	'address of', unary operator
sizeof	number of bytes in memory, unary operator
/	division, either floating point or integer
%	remainder with integer division
<<	shift left; stream output
>>	shift right; stream input
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Precedence of Operators

Operator	Meaning
==	equal to
!=	not equal to
^	bitwise exclusive OR (XOR)
	bitwise OR
&&	logical AND
	logical OR
?:	conditional expression
=	assignment
+=	addition combined with assignment (other operators can also be combined with assignment)

Arithmetic Conversions

Every arithmetic expression has a type

This type can be derived from those of its operands

first, integral promotion may take place: operands of type `char`, `short`, and `enum` are ‘promoted’ to `int` if this type can represent all the values of the original type; otherwise the original type is converted to `unsigned int`

type conversion is now applied, as follows

Arithmetic Conversions

One of the following 7 rules is applied
(considering each in strict order)

If either operand is `long double`, the other is converted to this type

If either operand is `double`, the other is converted to this type

If either operand is `float`, the other is converted to this type

If either operand is `unsigned long`, the other is converted to this type

Arithmetic Conversions

One of the following 7 rules is applied
(considering each in strict order)

If either operand is `long` and the other is `unsigned`, the other is converted to `long`, provided that `long` can represent all the values of `unsigned`. If not, both operands are converted to `unsigned long`

If either operand is a `long`, the other is converted to this type

If either operand is a `unsigned`, the other is converted to this type

The cast-operator

Forced type conversion

casting

coercion

`(float)n // cast n as a float (C and C++)`

`float(n) // cast n as a float (C++)`

Example

```
int i=14, j=3;
```

```
float x, y;
```

```
x = i/j; // x = 4.0
```

```
y = float(i)/float(j); // y = 4.666..
```

```
y = float(i/j); // y = 4.0
```


The cast-operator

Example

```
int i;  
float x = -6.9;  
i = x;           // i = -6  
i = int(x);      // i = -6, but it is clear  
                 // that conversion takes  
                 // place
```

Lvalues

Consider an assignment expression of the form:

$$E1 = E2$$

Normally $E1$ will be a variable, but it can also be an expression

An expression that can occur on the left-hand side of an assignment operator is called a *modifiable lvalue*

Lvalues

Not lvalues:

```
3 * 5  
i + 1  
printf("&d", a)
```

lvalues (given `int i, j, a[100], b[100];`)

```
i  
a[3 * i + j]  
(i)
```

Lvalues

Array names are not lvalues:

```
a = b; // error
```

lvalues

```
(i < j ? i : j) = 0; // assign 0 to  
                    // smaller of i and j
```

Since `?:` has higher precedence than `=`, we can write as:

```
i < j ? i : j = 0; // !!!
```

Lvalues

The conditional expression `E1 ? E2 : E3` is an lvalue only if `E2` and `E3` are of the same type and are both lvalues

NB: this is a C++ feature; conditional expressions cannot be lvalues in C

The results of a cast is not an lvalue (except in the case of reference types, yet to come)

```
float(x) = 3.14; // error
```

Functions

In C++ there is no distinction between functions, procedure, and subroutine; we use the term function for all

Consider a function `fun`

with four parameters `x`, `y`, `i`, and `j`, of type `float`, `float`, `int`, and `int`, respectively

which computes and returns

$(x-y)/(i-j)$, $i \neq j$

10^{20} , $i = j$ and $x \neq y$ (with sign of $(x-y)$)

0, $i = j$ and $x = y$

Functions

```
// FDEM01: Demonstration program with a function
#include <iostream.h>

int main()
{
    float fun(float x, float y, int i, int j);
    float xx, yy;
    int ii, jj;
    cout << "Enter two real numbers followed by two integers:
\n";
    cin >> xx >> yy >> ii >> jj;
    cout << "Value returned by function: "
        << fu(xx, yy, ii, jj) << endl;
    return 0;
}
```

Functions

```
float fun(float x, float y, int i, int j)
{  float a = x - y;
   int   b = i - j;
   return b != 0 ? a/b :
           a > 0  ? +1e20 :
           a < 0  ? -1e20 : 0.0;
}
```


Functions

```
float fun(float x, float y, int i, int j) //ALTERNATIVE
{ float a, result;                      //FORMULATION
  int b;
  a = x - y;  b = i - j;
  if (b != 0)
    result = a/b;                      // non-zero denominator
  else
    if (a > 0)
      result = +1e20;                  // +ve numerator
    else
      if (a < 0)
        result = -1e20; // -ve numerator
      else
        result = 0.0;                  // zero numerator
  return result;
}
```

Functions

Key points

`fun(xx, yy, ii, jj)` is a function call

`xx, yy, ii, jj` are called arguments

the parameters `x, y, i, j` are used as local variables within the function

the initial values of `x, y, i, j` correspond to the passed arguments `xx, yy, ii, jj`

the `return expression;` statement assigns the value of `expression` to the function and then returns to the calling function (`main`, in this case)

Functions

Key points

The type of each parameter must be specified:

```
float fun(float x, float y, int i, int j) // correct
```

```
float fun(float x, y, int i, j) // incorrect
```

We can omit the parameter names in the function *declaration* (but not the *definition*) but it's not good practice

```
float fun(float, float, int, int) // declaration in  
                                // main()
```

A function may be declared either inside a function that contains a call to it or before it at the global level

Functions

Key points

A function may be declared either inside a function that contains a call to it or before it at the global level

```
float fun(float x, float y, int i, int j);  
...  
main()  
{  
}
```

Global declarations are valid until the end of the file

Can have many declarations

Can have only one definition (which must not occur inside another function)

Functions

Key points

If the definition occurs before the first usage of the function, there is no need to declare it (as a definition is also a declaration)

Function arguments can be expressions

```
float result = fun(xx+1, 2*yy, ii+2, jj-ii);
```

NOTE: the order in which the arguments are evaluated is undefined

```
/* ill defined function call */  
float result = fun(xx, yy, ++ii, ii+3);
```

The `void` Keyword

Some functions do not return a value

Similar to procedures and subroutines

The functions are given the type `void`

```
void max(int x, int y, int z)
{
    if (y > x) x = y;
    if (z > x) x = z;
    cout << "the maximum is " << x << endl;
}
// Poor programming style; why?
```

The `void` Keyword

Functions with no parameters are declared (and defined) with parameters of type `void`

```
/* read a real number */
double readreal(void)
{ double x; char ch;
  while (scanf("%lf", &x) != 1)
  { // skip rest of incorrect line
    do ch = getchar(); while (ch != '\n');
    printf("\nIncorrect. Enter a number:\n");
  }
  return x;
}
```

The `void` Keyword

In C omission of `void` would have implied that the function could have had any number of parameters

In C++ omission of information about parameters is not allowed

no parameters means NO parameters

```
double readreal(void) = double readreal()
```


The `void` Keyword

In C one normally writes

```
main()
```

This is equivalent to:

```
int main()
```

not `void main()` and implies `return 0;` at the end of the main function.

The `void` Keyword

It makes sense to adopt the `int` formulation (with the required `return statement`) since the operating system often picks up the main return value as a run-time error code

Global Variables

Local variables

- defined within a function

- visible only within that function (local scope)

Global variables

- defined outside functions

- visible from the point of definition to end of the file (global scope)

- modification of a global variable by a function is called a side-effect ... to be avoided

Global Variables

Scope Rules

If two variables have the same name and are both in scope, then the one with local scope is used

Scope resolution operator ::

C++ enables us to explicitly over-ride the local scope rule

Indicate that the global variable is meant by writing the scope-resolution-operator in front of the variable name

Scope Resolution

```
#include <iostream.h>
int i = 1;

int main()
{   int i=2;
    cout << ::i << endl; // Output: 1 (global variable)
    cout << i    << endl; // Output: 2 (local variable)
    return 0;
}

// more on :: in the section on structures and classes
```

Functions

Altering Variables via Parameters

C++ allows reference parameters

```
void swap1(int &x, int &y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

The arguments of `swap1` must be lvalues

Functions

Altering Variables via Parameters

C does not allow reference parameters

Instead of passing parameters by reference

we pass the address of the argument

and access the parameter indirectly in the function

& (unary operator)

address of the object given by the operand

* (unary operator)

object that has the address given by the operand

Functions

Altering Variables via Parameters

```
void swap1(int *p, int *q)
{
    int temp;
    temp = *p;
    *p = *q;
    *q = temp;
}
```

```
swap(&i, &j); // function call
```


Functions

Altering Variables via Parameters

Pointers

`*p` has type `int`

But `p` is the parameter, not `*p`

Variables that have addresses as their values are called *pointers*

`p` is a pointer

The type of `p` is *pointer-to-int*

Functions

Types of Arguments and Return Values

Argument Types

Function arguments are automatically converted to the required parameter types, if possible

If it is not, an (compile) error message is given

Thus it is valid to pass an integer argument to a function with a float parameter

The same rules for type conversion apply as did in assignment statements

Functions

Types of Arguments and Return Values

Types of return values

The conversion rules also apply to return-statements

```
int g(double x, double y)
{   return x * x - y * y + 1;
}
```

Since the definition says that `g` has type `int`, the value returned is `int` and truncation take place

Functions

Types of Arguments and Return Values

Types of return values

It would be better to explicitly acknowledge this with a cast

```
int g(double x, double y)
{   return int (x * x - y * y + 1);
}
```

Functions

Initialization

Variables can be initialized when they are declared.
However:

- Variables can be initialized only when memory locations are assigned to them

- In the absence of explicit initialization, the initial value 0 is assigned to all variables that are `global` or `static`

Functions

Initialization

`global`

- variables that are declared outside functions
- have a permanent memory location
- can only be initialized with a constant expression

`static`

- a keyword used in the declaration to enforce the allocation of a permanent memory location
- static variables local to a function are initialized **ONLY** the first time the function is called
- can only be initialized with a constant expression

Functions

Initialization

`auto`

a keyword used in the declaration to enforce the (default) allocation of memory from the stack

such variables are called automatic

memory associated with automatic variables is released when the functions in which they are declared are exited

unless initialized, you should not assume automatic variables have an initial value of 0 (or any value)

can be initialized with any valid expression (not necessarily a constant expression)

Functions

Initialization

```
#include <iostream.h>

void f()
{   static int i=1;
    cout << i++ << endl;
}

int main()
{   f();
    f();
    return 0;
}
```


Functions

Initialization

```
#include <iostream.h>

void f()
{   static int i=1;
    cout << i++ << endl;
}

int main()
{   f();           // prints 1
    f();           // prints 2
    return 0;
}
```

Functions

Initialization

Uses of local static variables

For example, as a flag which can indicate the first time a function is called

```
void f()
{
    static int first_time = 1;
    if (first_time)
    {
        cout <<
            "f called for the first time\n";
        first_time = 0; // false
    }
    cout << "f called (every time)\n";
}
```

Functions

Initialization

Initialization of arrays

write the initial values in braces

There must not be more initial values than there are array elements

There can be fewer (but at least one!)

Trailing elements are initialized to 0

```
float a[100] = {23, 41.5};  
// a[0]=23; a[1]=41.5; a[2]= ... = a[99]= 0
```

```
char str[16] = "Charles Handy";
```

Functions

Initialization

Default arguments

C++ allows a function to be called with fewer arguments than there are parameters

Must supply the parameters with default argument values (i.e. initialized parameters)

Once a parameter is initialized, all subsequent parameters must also be initialized

```
void f(int i, float x=0, char ch='A')  
{  
    ...  
}
```

Functions

Initialization

```
void f(int i, float x=0, char ch='A')
{
    ...
}

...
f(5, 1.23, 'E');
f(5, 1.23);    // equivalent to f(5,1.23,'A');
f(5);          // equivalent to f(5,0,'A');
```

Functions

Initialization

Default arguments

functions which are both defined and declared can also have default argument

Default value may only be specified once, either in the declaration or in the definition

```
// declaration
void f(int i, float, char ch);

// definition
void f(int i, float x=0; char ch='A' )
{
    ...
}
```

Functions

Separate Compilation

Large programs can be split into modules, compiled separately, and subsequently linked

Functions

Separate Compilation

Functions

Separate Compilation

Functions

Separate Compilation

Key points

- n is used in both modules

 - defined in module 2

 - declared (to be extern) in module 1

- $f()$ and $g()$ are used in module 1

 - defined in module 2

 - declared in module 1

A variable may only be used after it has been declared

Only definitions reserve memory (and hence can only be used with initializations)

Functions

Separate Compilation

Key points

- we defined a variable only once

- we can declare it many times

- a variable declaration at the global level (outside a function) is valid from that declaration until the end of the file (*global scope*)

- a declaration inside a function is valid only in that function (*local scope*)

- we don't need to use the keyword `extern` with functions

Functions

Separate Compilation

Key points

```
static int m=7;
```

`m` is already global and so its memory location is permanent

Thus, the keyword `static` might seem unnecessary;

However, `static` global variables are the private to the module in which they occur

cannot write

```
extern int m; // error
```

Functions

Separate Compilation

Key points

- Static can also be used with functions

- This makes them private to the module in which they are defined

- The keyword `static`, both for global variables and for functions, is very for

- avoiding name-space pollution

- restricting scope and usage to instances where usage is intended

- Avoid global variables (and make them static if you must use them)

- make functions static if they are private to your code

Functions

Standard Mathematical Functions

Declare standard maths functions by

```
#include <math.h>
```

```
double cos(double x);
```

```
double sin(double x);
```

```
double tan(double x);
```

```
double exp(double x);
```

```
double ln(double x);
```

```
double log10(double x);
```

```
double pow(double x, double y); // x to the y
```

```
double sqrt(double x);
```

```
double floor(double x); // truncate
```

```
double ceil(double x); // round up
```

```
double fabs(double x); // |x|
```

Functions

Standard Mathematical Functions

```
double acos(double x);  
double asin(double x);  
double atan(double x); // -pi/2 .. +pi/2  
double atan2(double y, double x);  
double cosh(double x);  
double sinh(double x);  
double tanh(double x);
```

abs and labs are defined in stdlib.h but return integer values

Functions

Function Overloading

C++ allows the definition of two or more functions with the same name

This is known as *Function Overloading*

number or types of parameters must differ

```
void writenum(int i)    // function 1
{   printf("%10d", i);
}
```

```
void writenum(float x) // function 2
{   printf("%10.4f", x);
}
```


Functions

Function Overloading

```
writenum(expression) ;
```

function 1 is called if expression is type int

function 2 is called if expression is type float

The functions are distinguished by their parameter types and parameter numbers

Functions

Function Overloading

Allowable or not?

```
int g(int n)
{ ...
}
```

```
float g(int n)
{ ...
}
```

Functions

Function Overloading

Allowable or not?

```
int g(int n)
{ ...
}
```

```
float g(int n)
{ ...
}
```

Not! parameters don't differ.

Functions

Function Overloading

Type-safe linkage

Differentiation between functions is facilitated by name mangling

- coded information about the parameters is appended to the function name

- all this information is used by the linker

Functions

Function Overloading

Type-safe linkage

Of use even if not using function overloading:

```
void f(float n) // definition in
{ ...          // module 1
}              // only one defn. of f

void f(int i);  // declaration in
}              // module 2
```

C++ compilers will catch this; C compilers won't

Functions

References as Return Values

The return value can also be a reference (just as parameters can be reference parameters)

Functions

References as Return Values

Functions

References as Return Values

Key points about the assignment

```
smaller(a, b) = 0;
```

Function `smaller` returns the argument itself (i.e. either `a` or `b`)

This gets assigned the value 0

The arguments must be variables

The `&` must be used with the parameters

The returned value must exist outside the scope of the function

Functions

Inline Functions and Macros

A call to a function causes

- a jump to a separate and unique code segment

- the passing and returning of arguments and function values

This trades off time efficiency in favour of space efficiency

Inline functions cause

- no jump or parameter passing

- duplication of the code segment in place of the function call

Functions

References as Return Values

```
inline int sum(int n)
{   return n*(n+1)/2; // 1+2+ ... n
}
```

Should only use for time-critical code
and for short functions

Inline functions are available only in C++,
not C

Functions

Inline Functions and Macros

In C we would have used a macro to achieve the effect of inline functions

- define a macro

- macro expansion occurs every time the compiler preprocessor meets the macro reference

- for example

```
#define sum(n) ((n) * ((n)+1)); // note ()
```

Functions

Inline Functions and Macros

The following macro call

```
y = 1.0 / sum(k+1) / 2;
```

expands to

```
y = 1.0 / ((k+1) * ((k+1)+1) / 2);
```

Functions

Inline Functions and Macros

If we had defined the macro without full use of parentheses

```
#define sum(n) n* (n+1) /2;
```

the expansion would have been

```
y = 1.0 / k+1 * (k+1+1) /2;
```

which is seriously wrong ... why?

Functions

Inline Functions and Macros

Some macros have no parameters

```
#define LENGTH 100
```

```
#define ID_number(i) array_id[i];
```

Since macro expansion occurs at preprocessor stage, compilation errors refer to the expanded text and make no reference to the macro definition per se

Functions

Inline Functions and Macros

```
#define f(x) ((x) * (x) + (x) + 1) ;
```

```
...
```

```
y = f(a) * f(b) ;
```

produces the syntactically incorrect code
(and a possibly confusing “invalid indirection”
error)

```
y = ((a) * (a) + (a) + 1) ; * ((b) * (b) + (b) + 1) ; ;
```

Functions

Inline Functions and Macros

Previously defined macros can be used in the definition of a macro.

Macros cannot call themselves

if, in a macro definition, its own name is used then this name is not expanded

```
#define cos(x) cos((x) * PI/180)
//cos (a+b) expands to cos((a+b))*PI/180)
```


Functions

Inline Functions and Macros

A macro can be defined more than once
The replacement text **MUST** be identical

```
#define LENGTH 100  
...  
#define LENGTH 1000 // not allowed
```

Consequently, the same macro can now be defined in more than one header file

And it is valid to include several such header files in the same program file

Functions

Inline Functions and Macros

The string generating character

In macro definitions, parameters immediately preceded by a # are surrounded by double quotes in the macro expansion

```
#define print_value(x) printf(#x " = %f\n", x)
```

...

```
print_value(temperature);  
// expands to printf("temperature" " = %f\n",  
temperature);
```

Consequently, the same macro can now be defined in more than one header file

And it is valid to include several such header files in the same program

Functions

Other Preprocessor Facilities

Header files

The preprocessor also expands `#include` lines

```
#include <stdio.h>
#include "myfile.h"
```

The two lines are logically replaced by the contents of these header files

<code><...></code>	search for the header file only in the general include directories
<code>"..."</code>	search in the current directory first, then search in the general include direct.

Functions

Other Preprocessor Facilities

Header files

- normally used to declare functions and to define macros
- included in several module files
- header files can also include files
- function definition should NOT be written in header files (except, perhaps, inline functions)

Functions

Other Preprocessor Facilities

Conditional compilation

compile a program fragment (A) only if a certain condition is met

```
#if constant expression
    program fragment A
#else
    program fragment B
#endif
```

The `#else` clause is optional

Functions

Other Preprocessor Facilities

Conditional compilation

a useful way to 'comment out' large sections of text which comprises statements and comments
(remember, we can't nest comments)

```
#if SKIP
    /* lots of statements */
    a = PI;
    ...
#endif
```

Functions

Other Preprocessor Facilities

Tests about names being known

```
#if !defined(PI)
#define PI 3.14159265358979
#endif
```

`defined()` can be used with the logical operators `!`, `||`, and `&&`

Older forms:

`#ifdef name` is equivalent to `#if defined (name)`
`#ifndef name` is equivalent to `#if !defined (name)`

Functions

Other Preprocessor Facilities

Tests about names being known

```
#undef PI
```

undefines a name (even if it hasn't been defined)

Functions

Other Preprocessor Facilities

Making the compiler print error messages

```
#include "myfile.h"
#if !(defined(V_B)
#error You should use Ver. B of myfile.h
#endif
```

Compilation terminates after printing the error message

Functions

Other Preprocessor Facilities

Predefined names

can be used in constant expressions

<code>__LINE__</code>	integer: current line number
<code>__FILE__</code>	string: current file being compiled
<code>__DATE__</code>	string: date in the form <i>Mmm dd yyyy</i> (<i>date of compilation</i>)
<code>__TIME__</code>	string: date in the form <i>Mmm dd yyyy</i> (<i>time of compilation</i>)
<code>__cplusplus</code>	a constant defined only if we are using a C++ compiler

Functions

Exercises

7. Write and test a function `sort4`, which has four parameters. If the integer variables `a`, `b`, `c`, and `d` are available and have been assigned values, we wish to write:

```
sort4(&a, &b, &c, &d);
```

to sort these four variables, so that, after this call, we have $a \leq b \leq c \leq d$

8. Write and test a function `sort4_2` which uses reference parameters

Functions

Exercises

9. Investigate (on paper and then with a computer) the effect of the following recursive function and calling program with values $k=0,1,2,\dots,5$

```
sort4(&a, &b, &c, &d);
```

Functions

Exercises

```
#include <iostream.h>

void f(int n)
{   if (n > 0)
    {   f(n-2); cout << n << " "; f(n-1);
    }
}

int main()
{   int k;
    cout << "Enter k: "; cin >> k;
    cout << "Output:\n";
    f(k);
    return 0;
}
```

Functions

Exercises

10. Write and test a (recursive) function `gcd(x, y)` which computes the greatest common divisor of the integers `x` and `y`. These two integers are non-negative and not both equal to zero. Use Euclid's algorithm:

$$\text{gcd}(x, y) = \begin{cases} x & \text{if } y = 0 \\ \text{gcd}(y, x \% y) & \text{if } y \neq 0 \end{cases}$$

Arrays, Pointers, and Strings

Address Arithmetic

Address of operator `&`

The value of an expression `&x` is an address

Other expressions yield addresses

the name of an array, written without brackets

the address is that of the first element of the array

```
char s[50];
```

`s` is equivalent to `&(s[0])`

we can combine the name of an array with integers

`s` is equivalent to `&(s[0])`

`s+i` is equivalent to `&(s[i])`

Arrays, Pointers, and Strings

Address Arithmetic

Such expressions are valid even when the array elements are not 1 byte in size

In general address arithmetic takes into account the size of the element

```
int a[10];
```

`a+i` is equivalent to `&(a[i])`

Such a capability leads some people to write:

```
for (i=0; i<10; i++) scanf("%d", a+i);
```

rather than

```
for (i=0; i<10; i++) scanf("%d", &a[i]);
```


Arrays, Pointers, and Strings

Address Arithmetic

Indirection operator $*$

The value of an expression such as $*a$ is the object to which the address a refers

$*a$ is equivalent to $a[0]$

$*(a+i)$ is equivalent to $a[i]$

Arrays, Pointers, and Strings

Function Arguments and Arrays

In C and C++ there is no need for special parameter-passing for arrays

We pass the address of the first element of the array

Which is the array name!

We automatically have access to all other elements in the array

Functions

Function Arguments and Arrays

```
// MINIMUM: finding the smallest element of an
// integer array
```

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

```
int main()
```

```
{  int table[10], minimum(int *a, int n);
    cout << "Enter 10 integers: \n";
    for (int i=0; i<10; i++) cin >> table[i];
    cout << "\nThe minimum of these values is "
         << minimum(table, 10) << endl;
    return 0;
}
```

Functions

Function Arguments and Arrays

```
// definition of minimum, version A
```

```
int minimum(int *a, int n)
{
    int small = *a;
    for (int i=1; i<n; i++)
        if (*(a+i) < small)
            small = *(a+i);
    return small;
}
```

Functions

Function Arguments and Arrays

```
// definition of minimum, version B (for Better!)
```

```
int minimum(int a[], int n)
{   int small = a[0];
    for (int i=1; i<n; i++)
        if (a[i] < small)
            small = a[i];
    return small;
}
```

Arrays, Pointers, and Strings

Pointers

In the following `p` is a pointer variable

```
int *p, n=5, k;
```

Pointers store addresses

```
p = &n;  
k = *p // k is now equal to???
```

`*` is sometimes known as a dereferencing operator and accessing the object to which the pointer points is known as dereferencing

```
p = &n
```

Arrays, Pointers, and Strings

Pointers

It is essential to assign value to pointers

after declaring `p` we must not use `*p` before assigning a value to `p`.

```
int main()
{
    char *p, ch;
    *p = 'A'; // Serious error!
    return 0;
}
```

Arrays, Pointers, and Strings

Pointers

It is essential to assign value to pointers

after declaring `p` we must not use `*p` before assigning a value to `p`.

```
int main()
{
    char *p, ch;
    p = &ch;
    *p = 'A';
    return 0;
}
```


Arrays, Pointers, and Strings

Pointers

Pointer conversion and void-pointers

```
int i;  
char *p_char;  
  
p_char = &i; // error: incompatible types  
             // pointer_to_char and  
             // pointer_to_int  
  
p_char = (char *)&i;  
          // OK: casting pointer_to_int  
          // as pointer_to_char
```

Arrays, Pointers, and Strings

Pointers

In C++ we have generic pointer types:
void_pointers

```
int i;
char *p_char;
void *p_void;

p_void = &i; // pointer_to_int to pointer_to_void
p_char = (char *)p_void;
           // cast needed in C++ (but not ANSI C)
           // for pointer_to_void to
           // pointer_to_int
```

Arrays, Pointers, and Strings

Pointers

void_pointers can be used in comparisons

```
int  *p_int;  
char *p_char;  
void *p_void;
```

```
if (p_char == p_int) ... // Error  
if (p_void == p_int) ... // OK
```

Address arithmetic must not be applied to void_pointers. Why?

Arrays, Pointers, and Strings

Pointers

Typedef declarations

used to introduce a new identifier denote an (arbitrarily complex) type

```
typedef double real;  
typedef int *ptr;  
...  
real x,y;    // double  
ptr p;       // pointer_to_int
```

Arrays, Pointers, and Strings

Pointers

Initialization of pointers

```
int i, a[10];  
int *p = &i;  // initial value of p is &i  
int *q = a;   // initial value of q is the  
              // address of the first element  
              // of array a
```

Arrays, Pointers, and Strings

Recap: addresses can appear in the following three forms

- expression beginning with the & operator

- the name of an array

- pointer

Another, fourth, important form which yields an address

- A string (string constant or string literal)

- “ABC”

Arrays, Pointers, and Strings

“ABC”

effectively an array with four char elements:

‘A’, ‘B’, ‘C’, and ‘\0’

The value of this string is the address of its first character and its type is `pointer_to_char`

<code>*"ABC"</code>	is equal to	<code>'A'</code>
<code>*("ABC" + 1)</code>	is equal to	<code>'B'</code>
<code>*("ABC" + 2)</code>	is equal to	<code>'C'</code>
<code>*("ABC" + 3)</code>	is equal to	<code>'\0'</code>

Arrays, Pointers, and Strings

“ABC”

effectively an array with four char elements:

‘A’, ‘B’, ‘C’, and ‘\0’

The value of this string is the address of its first character and its type is `pointer_to_char`

<code>"ABC"[0]</code>	is equal to	<code>'A'</code>
<code>"ABC"[1]</code>	is equal to	<code>'B'</code>
<code>"ABC"[2]</code>	is equal to	<code>'C'</code>
<code>"ABC"[3]</code>	is equal to	<code>'\0'</code>

Arrays, Pointers, and Strings

Assigning the address of a string literal to a pointer variable can be useful:

```
// POINTER
#include <stdio.h>
int main()
{   char *name = "main";
    printf(name);
    return 0;
}
```

Arrays, Pointers, and Strings

Strings Operations

Many string handling operations are declared in `string.h`

```
#include <string.h>
```

```
char s[4];
```

```
s = "ABC"; // Error: can't do this in C; Why?  
strcpy(s, "ABC"); // string copy
```

Arrays, Pointers, and Strings

Strings Operations

Many string handling operations are declared in `string.h`

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

int main()
{   char s[100]="Program something.", t[100];
    strcpy(t, s);
    strcpy(t+8, "in C++.");
    cout << s << endl << t << endl;
    return 0;
}   // what is the output?
```

Arrays, Pointers, and Strings

Strings Operations

Many string handling operations are declared in `string.h`

```
strlen(string);  
    // returns the length of the string
```

E.g.

```
int length;  
char s[100]="ABC";  
length = strlen(s); // returns 3
```

Arrays, Pointers, and Strings

Strings Operations

Many string handling operations are declared in `string.h`

```
strcat(destination, source);  
    // concatenate source to destination  
  
strncat(destination, source, n);  
    // concatenate n characters of source  
    // to destination  
    // programmer is responsible for making  
    // sure there is enough room
```

Arrays, Pointers, and Strings

Strings Operations

Many string handling operations are declared in `string.h`

```
strcmp(string1, string2);  
    // returns 0 in the case of equality  
    // returns <0 if string1 < string2  
    // returns >0 if string1 > string2  
  
strncmp(string1, string2, n);  
    // same as strcmp except only n characters  
    // considered in the test
```

Arrays, Pointers, and Strings

Dynamic Memory Allocation

Array declarations

require a constant length specification

cannot declare variable length arrays

However, in C++ we can create an array whose length is defined at run-time

```
int n;  
char *s;  
...  
cin >> n;  
s = new char[n];
```

Arrays, Pointers, and Strings

Dynamic Memory Allocation

If memory allocation fails

would have expected `new` to return a value `NULL`

however, in C++ the proposed standard is that instead a *new-handler* is called

we can (usually) force `new` to return `NULL` by calling

```
set_new_handler(0);
```

before the first use of `new`

This has been adopted in the Borland C++ compiler

Arrays, Pointers, and Strings

Dynamic Memory Allocation

```
// TESTMEM: test how much memory is available
#include <iostream.h>
#include <new.h> required for set_new_handler

int main()
{   char *p;
    set_new_handler(0); // required with Borland C++
    for (int i=1;;i++) // horrible style
    {   p = new char[10000];
        if (p == 0) break;
        cout << "Allocated: " << 10 * i << "kB\n";
    }
    return 0;
} // rewrite this in a better style!
```

Arrays, Pointers, and Strings

Dynamic Memory Allocation

Memory is deallocated with `delete()`

```
p = new int // deallocate with:  
delete p;
```

```
p = new int[m] // deallocate with:  
delete[] p;
```

`delete` is only available in C++

Arrays, Pointers, and Strings

Dynamic Memory Allocation

`malloc()`

standard C memory allocation function

declared in `stdlib.h`

its argument defines the number of bytes to be allocated

```
#include <stdlib.h>
int n;
char *s;
...
cin > n;
s = (char *) malloc (n);
```

Arrays, Pointers, and Strings

Dynamic Memory Allocation

`malloc()`

but to allocate an array of floats:

```
#include <stdlib.h>
int n;
float *f;
...
cin > n;
s = (float *) malloc (n * sizeof(float));
```

`malloc()` **returns NULL** if allocation fails

Arrays, Pointers, and Strings

Dynamic Memory Allocation

`malloc()`

```
s = (float *) malloc (n * sizeof(float));
if (s == NULL)
{   cout << "Not enough memory.\n";
    exit(1); // terminates execution of program
}           // argument 1: abnormal termination
```

Arrays, Pointers, and Strings

Dynamic Memory Allocation

`calloc()`

Takes two arguments

number of elements

size of each element in bytes

all values are initialized to zero

`calloc()` returns `NULL` if allocation fails

Arrays, Pointers, and Strings

Dynamic Memory Allocation

Memory is deallocated with `free()`

```
free(s);
```

Arrays, Pointers, and Strings

Input and Output of Strings

Input

```
char[40] s;  
...  
scanf("%s", s); // skips whitespace and terminates on  
                // whitespace  
cin >> s;       // same as scanf  
gets(s);        // reads an entire line  
  
// problems if more than 40 chars are typed:  
// ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz  
// requires a string of 53 elements
```


Arrays, Pointers, and Strings

Input and Output of Strings

Input

```
char[40] s;  
...  
scanf("%39s", s);    //reads at most 39 characters  
cin >> setw(40) >> s; // same as scanf  
fgets(s, 40, stdin); // reads a line of at most 39  
                      // characters, including \n  
cin.getline(s, 40);  // reads a line of at most 39  
                      // characters, including \n  
                      // but doesn't put \n in s
```

Arrays, Pointers, and Strings

Input and Output of Strings

Output

```
char[40] s;  
...  
printf(s);           // Display just the contents of s  
printf("%s", s);     // same  
cout << s;           // same  
printf("%s\n", s);   // Display s, followed by newline  
puts(s);             // same
```

Arrays, Pointers, and Strings

Input and Output of Strings

Output

```
// ALIGN1: strings in a table, based on standard I/O
#include <stdio.h>

int main()
{
    char *p[3] = {"Charles", "Tim", "Peter"};
    int age[3] = {21, 5, 12}, i;
    for (i=0; i<3; i++)
        printf("%-12s%3d\n", p[i], age[i]); // left align
    return 0;
}
```

Arrays, Pointers, and Strings

Input and Output of Strings

Output

```
// ALIGN2: strings in a table, based on stream I/O
#include <iostream.h>
#include <iomanip.h>
int main()
{   char *p[3] = {"Charles", "Tim", "Peter"};
    int age[3] = {21, 5, 12}, i;
    for (i=0; i<3; i++)
        cout << setw(12) << setiosflags(ios::left) << p[i]
            << setw(3) < resetiosflags(ios::left)
            << age[i];
    return 0;
}
```

Arrays, Pointers, and Strings

Multi-Dimensional Arrays

A table or matrix

can be regarded as an array whose elements are also arrays

```
float table[20][5]
```

```
int a[2][3] = {{60,30,50}, {20,80,40}};
```

```
int b[2][3] = {60,30,50,20,80,40};
```

```
char namelist[3][30]
```

```
= {"Johnson", "Peterson", "Jacobson"};
```

```
...
```

```
for (i=0; i<3; i++)
```

```
    cout << namelist[i] << endl;
```

Arrays, Pointers, and Strings

Multi-Dimensional Arrays

Pointers to 2-D arrays:

```
int i, j;  
int a[2][3] = {{60,30,50}, {20,80,40}};  
int (*p)[3]; // p is a pointer to a 1-D array  
              // of three int elements
```

...

```
p = a; // p points to first row of a  
a[i][j] = 0;           // all four statements  
(* (a+i)) [j] = 0;     // are equivalent  
p[i][j] = 0;           // remember [] has higher  
(* (p+i)) [j] = 0;     // priority than *
```

Arrays, Pointers, and Strings

Multi-Dimensional Arrays

Function Parameters

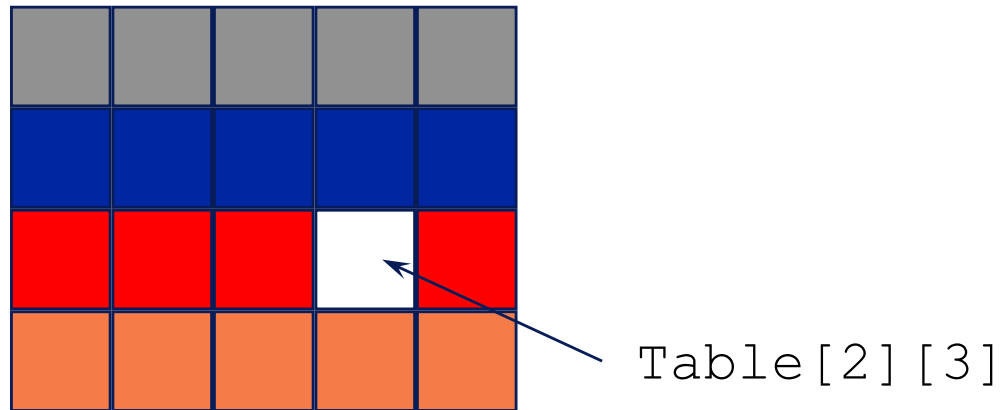
```
int main()
{   float table[4][5];
    int f(float t[][5]);

    f(table);
    return 0;
}

int f(float t[][5]) // may omit the first dimension
{                  // but all other dimensions must
}                  // be declared since it must be
                   // possible to compute the
                   // address of each element. How?
```

Arrays, Pointers, and Strings

Multi-Dimensional Arrays



The address of `Table[i][j]` is computed by the mapping function $5*i + j$ (e.g. $5*2+3 = 13$)

Arrays, Pointers, and Strings

Multi-Dimensional Arrays

Arrays of Pointers

we can create 2-D 'arrays' in a slightly different (and more efficient) way using

- an array of pointers to 1-D arrays

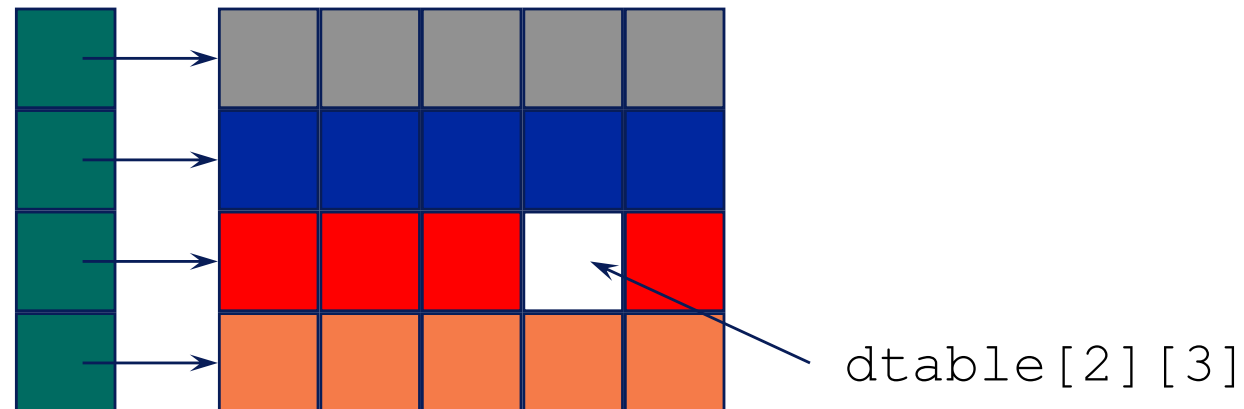
- a sequence of 1-D arrays

```
float *dtable[4]; // array of 4 pointers to floats
set_new_handler(0);
for (i=0; i<20; i++)
{
    dtable[i] = new float[5];
    if (dtable[i] == NULL)
    {
        cout << " Not enough memory"; exit(i);
    }
}
```

Arrays, Pointers, and Strings

Multi-Dimensional Arrays

dtable



`dtable[i][j]` is equivalent to `(*(dtable+i))[j]` ...
there is no multiplication in the computation
of the address, just indirection.

Arrays, Pointers, and Strings

Program Parameters

The `main()` function of a program can have parameters
called program parameters

an arbitrary number of arguments can be supplied

represented as a sequence of character strings

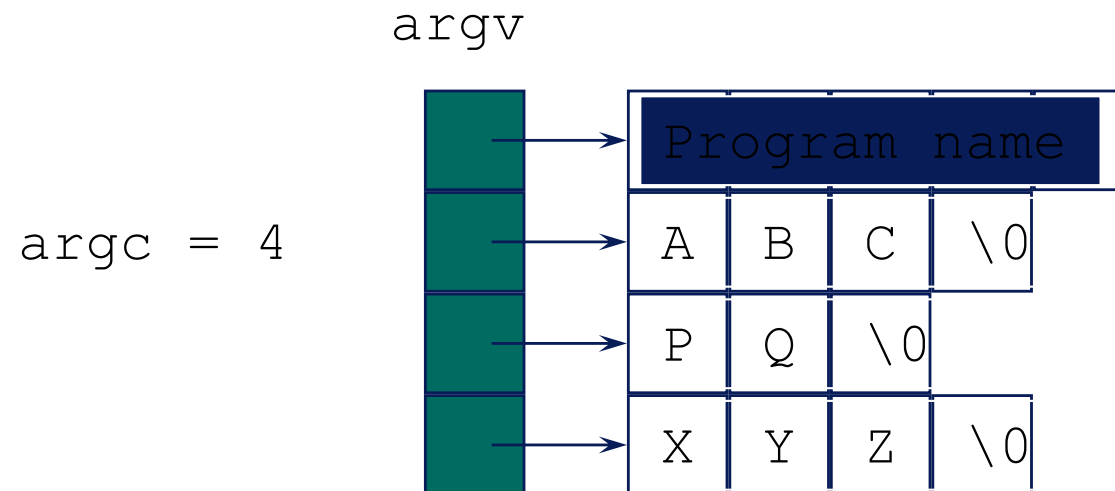
two parameters

`argc` ... the number of parameters (argument count)

`argv` ... an array of pointers to strings (argument vector)

Arrays, Pointers, and Strings

Program Parameters



Program parameters for an invocation of the form
program ABC PQ XYZ

Arrays, Pointers, and Strings

Program Parameters

```
// PROGPARAM: Demonstration of using program parameters
#include <iostream.h>

int main(int argc, char *argv[])
{   cout << "argc  = " << argc << endl;
    for (int i=1; i<argc; i++)
        cout << "argv[" << i << "]= " << argv[i] << endl;
    return 0;
}
```

Arrays, Pointers, and Strings

In-Memory Format Conversion

`sscanf()`

scans a string and converts to the designated type

```
#include <stdio.h>
```

```
...
```

```
char s[50]="123    456 \n98.756";
```

```
int i, j;
```

```
double x;
```

```
sscanf(s, "%d %d %lf", &i, &j, &x);
```

`sscanf` returns the number of value successfully scanned

Arrays, Pointers, and Strings

In-Memory Format Conversion

`sscanf()`

fills a string with the characters representing the passed arguments

```
#include <stdio.h>
```

```
...
```

```
char s[50]="123    456 \n98.756";
```

```
sprintf(s, "Sum: %6.3f Difference: %6.3f",  
        45 + 2.89, 45 - 2.89);
```

Arrays, Pointers, and Strings

Pointers to Functions

In C and C++ we can assign the start address of functions to pointers

```
// function definition
float example (int i, int j)
{   return 3.14159 * i + j;
}
```

```
float (*p)(int i, int j); // declaration
...
p = example;
```


Arrays, Pointers, and Strings

Pointers to Functions

And we can now invoke the function as follows

```
(*p) (12, 34); // same as example(12,34);
```

We can omit the `*` and the `()` to get:

```
p(12, 34);    // !!
```

Pointers to function also allow us to pass functions as arguments to other functions

Arrays, Pointers, and Strings

Exercise

11. Write and test a function to
read a string representing a WWW URL
(e.g. `http://www.cs.may.ie`)
replace the `//` with `\\`
write the string back out again

Arrays, Pointers, and Strings

Exercises

12. Write a interactive user interface which allows a user to exercise all of the set operators for three pre-defined sets A, B, and C

Commands should be simple single keywords with zero, one, or two operands, as appropriate

add A 10

union C A B

list A

intersection C A B

remove 1 B

Classes and Objects

Classes and Structures

Classes and structures - ways of grouping variables of different types

- similar to records in other languages

- a C++ class is a generalization of a structure in C

C++ has both classes and structures

Classes and Objects

Classes and Structures

Differences between Classes and C structures

Encapsulation

classes (and C++ structures) can have functions as their members
these functions operate on the data members

Data hiding

classes provide form member-access control
for each component in a class or structure we can indicate
whether or not data hiding is to apply

defaults:

class: full data-hiding

structure: no data-hiding

Classes and Objects

Classes and Structures

A class is a type

a variable of such a type is called an object

(more specifically, a class object)

Members of objects are accessed via the member names

Classes and Objects

Classes and Structures

Example of Structure Declaration

```
struct article {  
    int code;  
    char name[20];  
    float weight, length;  
};
```

this structure is a class with only public members

Classes and Objects

Classes and Structures

Equivalent example of Class Declaration

```
class article {  
public:  
    int code;  
    char name[20];  
    float weight, length;  
};
```

this structure is a class with only public members

Classes and Objects

Classes and Structures

Declaration (and definition) of a class variable (i.e. object)

```
article s, t;    // in C we would have  
                // had to write  
                // struct article s, t;
```

This form is identical to the conventional variable definitions

Classes and Objects

Classes and Structures

Since class declarations will probably be used by many program modules, it is good practice to put them in a header file which can be included as required

```
#include "my_classes.h"
```

Classes and Objects

Classes and Structures

Access to members of class variables

```
s.code = 123;
strcpy(s.name, "Pencil");
s.weight = 12.3;
s.length = 150.7;
t = s;    // !! object assignment possible
          // even though a member is an
          // array (and array assignment
          // is not possible)
```

Classes and Objects

Classes and Structures

Composite variables such as classes, structures, arrays are called aggregates

Members of aggregates which are in turn aggregates are called subaggregates

Classes and Objects

Classes and Structures

Pointers to class objects

```
article *p;  
...  
(*p).code = 123; // normal dereferencing  
p->code = 123;    // more usual shorthand
```

Classes and Objects

Classes and Structures

Initialization of class objects

```
class article {
public:
    int code;
    char name[20;
    float weight, length;
};

int main()
{ static article s = (246, "Pen", 20.6, 147.0},
  t;  // is t initialized?
  ...
}
```

Classes and Objects

Classes and Structures

Initialization of class objects with object of identical type

```
...  
article s = (246, "Pen", 20.6, 147.0};  
...  
  
void f(void)  
{ article t=s, u=t;  
    ...  
}
```

Classes and Objects

Classes and Structures

Subaggregates

access to array members

```
s.name[3] = 'A'; // i.e. (s.name)[3]
```

arrays of class objects

```
article table[2];  
...  
table[i].length = 0;  
table[i].name[j] = ' ';
```


Classes and Objects

Classes as Arguments and Return Values

Three possibilities

- By 'value'; entire class object as argument/return value

- By address

- By reference (effectively the same as address)

The following three examples will use a show how a function can create a new object with values based on an object passed to it

Classes and Objects

Classes as Arguments and Return Values

Assume the following class declaration

```
// ARTICLE.H: header file for 3 demos
class article {
public:
    int code;
    char name[20];
    float weight, length;
};
```

Classes and Objects

Classes as Arguments and Return Values

Pass by value: copy class objects

```
// ENTOBJ: passing an entire class object
#include <iostream.h>
#include "article.h"

article largeobj(article x)    // functional spec
{
    x.code++;                  // increment code
    x.weight *=2 ;             // double weight
    x.length *=2;              // double length
    return x;                  // return new obj.
}

// main to follow
```

Classes and Objects

Classes as Arguments and Return Values

Pass by value: copy class objects

```
int main()
{
    article s = (246, "Pen", 20.6, 147.0), t;
    t = largeobj(s);
    cout << t.code << endl;
    return 0;
}
```

Classes and Objects

Classes as Arguments and Return Values

Pass address of class object

```
// PTROBJ: pointer parameter & return value
#include <iostream.h>
#include <string.h>
#include "article.h"
article *plargeobj(article *px) // functional spec
{
    article *p;                // pointer
    p = new article;            // new article
    p->codex++;                  // increment code
    strcpy(p->name, px->name);   // copy name
    p->weight = 2 * px->weight;  // double weight
    p->length = 2 * px->length;  // double length
    return p;                   // return new obj.
}

// main to follow
```

Classes and Objects

Classes as Arguments and Return Values

Pass address of class object

```
int main()
{
    article s = (246, "Pen", 20.6, 147.0);
    article *pt;
    pt = plargeobj(&s);
    cout << pt->code << endl;
    return 0;
}
```

Classes and Objects

Classes as Arguments and Return Values

Pass address of class object; V.2

```
// PTROBJ2: pointer parameter
#include <iostream.h>
#include <string.h>
#include "article.h"
void penlargeobj(article *p)           // functional spec
{   p->codex++;                        // modify values
    p->weight *= 2;                    // of passed
    p->length *= 2;                    // object
}
// main to follow
...
penlargeobj(&s);
```

Classes and Objects

Classes as Arguments and Return Values

Pass address of class object - common error

```
// PTROBJ: pointer parameter & return value
#include <iostream.h>
#include <string.h>
#include "article.h"
article *plargeobj(article *px) // functional spec
{
    article obj;                // local object
    obj.codex = px->code +1;     // increment code
    strcpy(obj.name, px->name);  // copy name
    obj.weight = 2 * px->weight; // double weight
    obj.length = 2 * px->length; // double length
    return &obj;                // ERROR; WHY?
}
```


Classes and Objects

Classes as Arguments and Return Values

Pass reference to class object

```
// REFOBJ: reference parameter & return value
#include <iostream.h>
#include <string.h>
#include "article.h"
article &rlargeobj(article &x) // functional spec
{
    article *p;                // pointer
    p = new article;           // new article
    p->codex = x.code + 1;      // increment code
    strcpy(p->name, x.name);    // copy name
    p->weight = 2 * x.weight;   // double weight
    p->length = 2 * x.length;   // double length
    return *p;                 // return new obj.
}

// main to follow
```

Classes and Objects

Classes as Arguments and Return Values

Pass reference to class object

```
int main()
{
    article s = (246, "Pen", 20.6, 147.0);
    article *pt;
    pt = &rlargeobj(s);
    cout << pt->code << endl;
    return 0;
}
```

Classes and Objects

Classes as Arguments and Return Values

Dynamic data structures

Class member can have any type

A member could be a pointer *p* pointing to another object of the same type (as the one of which *p* is a member)

```
struct element {int num; element *p;};
```

Such types, together with dynamic memory allocation, allow the creation of objects dynamically and the creation of dynamic data structures (e.g. linked lists and binary trees)

Classes and Objects

Unions

Union is a special case of a class

so far, all members of class objects exist simultaneously
however, if we know that certain members are mutually exclusive we can save space (knowing they can never occur at the same time)

Unions allow class objects to share memory space

but it is the responsibility of the programmer to keep track of which members have been used.

Typically, we do this with a tag field

Classes and Objects

Unions

Union with a tag field

```
enum choice{intflag, floatflag};
struct either_or {
    choice flag;
    union {
        int i;
        float x;
    } num;
}
...
either_or a[100];
a[k].num.i = 0;
a[k].flag = intflag;  // etc.
```

Classes and Objects

Member Functions and Encapsulation

Member functions and encapsulation are features of C++

With data abstraction (and abstract data types) we identify

- the set of values a variable of a particular type can assume
- the set of functions which can operate on variables of a particular type

Classes and Objects

Member Functions and Encapsulation

C++ allows us to localise these definitions in one logical entity: the class

- by allowing functions to be members of classes (i.e. through encapsulation)

- by appropriate data hiding

Classes and Objects

Member Functions and Encapsulation

```
// VEC1: A class in which two functions are defined
// (inside the class, therefore they act as inline fns)
#include <iostream.h>

class vector {
public:
    float x, y;
    void setvec (float xx, float yy)
    {   x = xx;
        y = yy;
    }
    void printvec() const    // does not alter members
    {   cout << x << ' ' << y << endl;
    }
};
```


Classes and Objects

Member Functions and Encapsulation

```
int main()
{   vector u, v;
    u.setvec(1.0, 2.0);    // note form of function call
    u.printvec();
    v.setvec(3.0, 4.0);
    v.printvec();
    return 0;
}
```

Classes and Objects

Member Functions and Encapsulation

```
// VEC2: A class in which two functions are declared
// (but defined outside the class)
#include <iostream.h>

class vector {
public:
    float x, y;
    void setvec (float xx, float yy);
    void printvec() const;
};
```

Classes and Objects

Member Functions and Encapsulation

```
int main()
{
    vector u, v;
    u.setvec(1.0, 2.0); // note form of function call
    u.printvec();
    v.setvec(3.0, 4.0);
    v.printvec();
    return 0;
}

void vector::setvec (float xx, float yy) // note ::
{
    x = xx;
    y = yy;
}

void vector::printvec() const
{
    cout << x << ' ' << y << endl;
}
```

Classes and Objects

Member Functions and Encapsulation

Note the use of `vector::`

necessary to indicate that the functions are members of the class `vector`

as a consequence, we can use the member identifiers (i.e. `x` and `y`)

could also have used `this->x` and `this->y` to signify more clearly that `x` and `y` are members of class objects.

`this` is a C++ keyword

It is always available as a pointer to the object specified in the call to that function

Classes and Objects

Member Access Control

In both previous examples, the scope of the members `x` and `y` was global to the function in which `vector` was declared, i.e. `main()`

`x` and `y` could have been accessed by `main()`

this situation may not always be desired

We would like to distinguish between class members belonging to:

the interface ... those that are public

the implementation ... those that are private (accessible only to the encapsulated functions)

Classes and Objects

Member Access Control

Private class members are introduced by the keyword `private`

Public class members are introduced by the keyword `public`

The default for `structs` (i.e. no keyword provided) is `public`

The default for `classes` is `private`

Classes and Objects

Member Access Control

```
// VEC3: A class with private members x and y
```

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

```
class vector {
```

```
public:
```

```
    void setvec (float xx, float yy);
```

```
    void printvec() const;
```

```
private:
```

```
    float x, y;
```

```
};
```

Classes and Objects

Member Access Control

```
int main()
{
    vector u, v;
    u.setvec(1.0, 2.0); // note form of function call
    u.printvec();
    v.setvec(3.0, 4.0);
    v.printvec();
    return 0;
}

void vector::setvec (float xx, float yy) // note ::
{
    x = xx;
    y = yy;
}

void vector::printvec() const
{
    cout << x << ' ' << y << endl;
}
```


Classes and Objects

Member Access Control

```
// VEC3: A class with private members x and y
// Alternative (but not recommended) declaration
#include <iostream.h>
```

```
class vector {
    float x, y; // defaults to private
public:
    void setvec (float xx, float yy);
    void printvec() const;
};
```

Classes and Objects

Member Access Control

Using `::` for functions that return pointers

If we are defining a class member function outside the class

e.g. `void vector::setvec()`

And if that function returns a pointer...

Then the expected `*` goes before the class name

e.g. `char *vector::newvec`

and NOT (as might have been anticipated) after the scope resolution operator `::` and before the function name

e.g. `char vector::*newvec`

Classes and Objects

Constructors and Destructors

Often, we wish an action to be performed every time a class object is created

C++ provides an elegant way to do this:

Constructor ... action to be taken on creation

Destructor ... action to be taken on deletion

Constructor

Class member function with *same name as class*

implicitly called whenever a class object is created (defined)

Classes and Objects

Constructors and Destructors

Constructor

- Class member function with *same name as class*
- implicitly called whenever a class object is created (defined)
- no type associated with the function (void, int, etc.)
- must not contain return statement

Classes and Objects

Constructors and Destructors

Destructor

Class member function with *same name as class preceded by a tilde ~*

implicitly called whenever a class object is deleted (e.g. on returning from a function where the class object is an automatic variable)

no type associated with the function (void, int, etc.)

must not contain return statement

Classes and Objects

Constructors and Destructors

```
// CONSTR: Demo of a constructor and destructor

#include <iostream.h>
class row {
public:
    row(int n=3)    // constructor with default param = 3
    {   len = n;   ptr = new int[n];
        for (int i=0; i<n; i++)
            ptr[i] = 10 * i;
    }
    ~row()          // destructor
    {   delete ptr;
    }
    void printrow(char *str) const;
private:
    int *ptr, len;
};
```

Classes and Objects

Constructors and Destructors

```
void row::printrow(char *str) const
{   cout << str;
    for (int i=0; i<len; i++)
        cout << ptr[i] << ' ';
    cout << endl;
}
void tworows();
{   row r, s(5);    // two instantiations of row,
                    // one which used default param 3
                    // one which uses 5 as the parameter
                    // Note: can't write r();
    r.printrow("r: ");
    s.printrow("s: ");
} // destructor ~row() implicitly called on exit

int main()
{   tworows();
    return 0;
}
```

Classes and Objects

Constructors and Destructors

Default Constructor

Instead of providing the row() constructor with default argument:

- define one constructor with parameters

- define another constructor with no parameters, i.e. the default constructor

If the constructor takes a parameter, then we must provide either a default constructor or a default argument

Classes and Objects

Constructors and Destructors

```
// CONSTR: Demo of a constructor and destructor

#include <iostream.h>
class row {
public:
    row(int n)    // constructor with parameters
    {   len = n;   ptr = new int[n];
        for (int i=0; i<n; i++)
            ptr[i] = 10 * i;
    }
    row()        // default constructor
    {   len = 3;   ptr = new int[3];
        for (int i=0; i<3; i++)
            ptr[i] = 10 * i;
    }
    ~row()       // destructor
    {   delete ptr;
    }
    .....
}
```

Classes and Objects

Constructors and Destructors

Constructor Initializer

The `row()` constructor initializes the (private) class object members `len` and `ptr`

We can also do this another way using a constructor initializer

```
row(int n=3):len(n), ptr(new int[n])
{
    for (int i=0; i<n; i++)
        ptr[i] = 10 * i;
}
```

Classes and Objects

Constructors and Destructors

Constructors and dynamically-created objects
when defining class objects

```
row r, s(5);
```

the constructor `row()` is invoked in the creation

we can also create pointers to class objects

```
row *p;
```

but since this is only a pointer to `row`, the constructor is not called

However, if we create the `row` pointed to by `p`

```
p = new row;
```

the constructor is then called.

Classes and Objects

Constructors and Destructors

Note that the constructor `row()` is not invoked if we use `malloc()` ;

Note also that the destructor `~row()` is called when we delete the row object

```
delete p;
```

but it is NOT invoked if we use `free()` ;

We can also specify arguments in the creation:


```
p = new row(5); // 5 element row
```

Classes and Objects

Constructors and Destructors

Constructors and arrays of class objects

If we define an array of class objects, the constructor is called for every array element (i.e. for every class object)



```
int main()
{   row a[2], b[6]={5, 1, 2}; //how many rows?
    cout << "Array a (two elements) \n";
    for (int i=0; i<2; i++) {
        cout << i;
        a[i].printrow(": ");
    }
    cout << "\nArray b (six elements)\n";
    for (int j=0; j<6; j++) {
        cout << j;
        b[j].printrow(": ");
    }
}
```

Classes and Objects

Constructors and Destructors

Output:

Array a (two elements)

0: 0 10 20

1: 0 10 20

Array b (six elements)

1: 0 10 20 30 40

2: 0

3: 0 10

4: 0 10 20

5: 0 10 20

5: 0 10 20

Classes and Objects

Operator Overloading and Friend Functions

We have already seen that we can overload functions
must **not** have same number and type of parameters

We can also overload operators

`new delete`

<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>	<code>^</code>	<code>&</code>	<code> </code>	<code>~</code>
<code>!</code>	<code>=</code>	<code><</code>	<code>></code>	<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>	<code>%=</code>
<code>^=</code>	<code>&=</code>	<code> =</code>	<code><<</code>	<code>>></code>	<code>>>=</code>	<code><<=</code>	<code>==</code>	<code>!=</code>
<code><=</code>	<code>>=</code>	<code>&&</code>	<code> </code>	<code>++</code>	<code>--</code>	<code>,</code>	<code>->*</code>	<code>-></code>
<code>()</code>	<code>[]</code>							

Note that the precedence of the operator cannot be changed

Classes and Objects

Operator Overloading and Friend Functions

Example: vector addition

let $\mathbf{u} = (x_u, y_u)$ and $\mathbf{v} = (x_v, y_v)$

the vector sum $\mathbf{s} = (x_s, y_s) = \mathbf{u} + \mathbf{v}$ is given by

$$x_s = x_u + x_v$$

$$y_s = y_u + y_v$$

We will overload the addition operator $+$ for vectors so that we can write $\mathbf{s} = \mathbf{u} + \mathbf{v};$

Classes and Objects

Operator Overloading and Friend Functions

```
// OPERATOR: an operator function for vector addition
```

```
#include <iostream.h>
class vector {
public:
    vector(float xx=0, float yy=0)
    {   x = xx; y = yy;
    }
    void printvec()const;
    void getvec(float &xx, float &yy)const
    {   xx = x; yy = y;
    }
private:
    float x, y;
};
```

Classes and Objects

Operator Overloading and Friend Functions

```
void vector::printvec()const
{   cout << x << ' ' << y << endl;
}
```

```
vector operator+(vector &a, vector &b) //why ref params
{   float xa, ya, xb, yb;
    a.getvec(xa, ya);
    b.getvec(xb, yb);
    return vector(xa + xb, ya + yb); // can't write a.x
                                     // and a.y ... why?
}
```

```
int main()
{   vector u(3, 1), v(1,2), s;
    s = u + v;      // sum of two vectors
    s.printvec();  // what's the output?
    return 0;
}
```

Classes and Objects

Operator Overloading and Friend Functions

Friend Functions

recall we couldn't write `a.x` and `a.y` in `operator+` because the members `x` and `y` are private to the class object (and `operator+` is not a class member)

consequently we had to have the class member function `getvec()`

we can allow `operator+` (and other functions) access to the private members

- by defining it as a `friend` function (next)

- by having it as a class member function (second next)

Classes and Objects

Operator Overloading and Friend Functions

```
// FRIEND: the 'friend' keyword applied to an operator
//          function

#include <iostream.h>
class vector {
public:
    vector(float xx=0, float yy=0)
    {   x = xx; y = yy;
    }
    void printvec()const;
    friend vector operator+(vector &a, vector &b);
private:
    float x, y;
};
```

Classes and Objects

Operator Overloading and Friend Functions

```
void vector::printvec() const
{   cout << x << ' ' << y << endl;
}

vector operator+(vector &a, vector &b)
{
    return vector(a.x + b.x, a.y + b.y); //friend access
}

int main()
{   vector u(3, 1), v(1,2), s;
    s = u + v;           // sum of two vectors
    s.printvec();       // what's the output?
    return 0;
}

// NOTE: operator+ is a friend function but NOT a class
member
```

Classes and Objects

Operator Overloading and Friend Functions

```
// FRIEND: the 'friend' keyword applied to an operator
//          function
//          This time, define operator+ in the class

#include <iostream.h>
class vector {
public:
    vector(float xx=0, float yy=0)
    {   x = xx; y = yy;
    }
    void printvec()const;
    friend vector operator+(vector &a, vector &b)
    {
        return vector(a.x + b.x, a.y + b.y);
    };
private:
    float x, y;
};
```

Classes and Objects

Operator Overloading and Friend Functions

Operators as member functions

we can also allow `operator+` access to the private members

by defining it as a class member

however, the syntax is a little odd!

`operator+` is a binary operator but it is allowed have only one parameter (the second operand)

the first operand is accessed implicitly and directly

Classes and Objects

Operator Overloading and Friend Functions

```
// OPMEMBER: An operator function as a class member

#include <iostream.h>

class vector {
public:
    vector(float xx=0, float yy=0)
    {   x = xx; y = yy;
    }
    void printvec()const;
    vector operator+(vector &b);
private:
    float x, y;
};
```


Classes and Objects

Operator Overloading and Friend Functions

```
void vector::printvec()const
{   cout << x << ' ' << y << endl;
}

vector vector::operator+(vector &b)
{
    return vector(x + b.x, y + b.y); //first operand is
                                     //the vector for
                                     //which the function
int main()                          //is called
{   vector u(3, 1), v(1,2), s;
    s = u + v;      // sum of two vectors
    s.printvec();  // what's the output?
    return 0;
}
```

Classes and Objects

Operator Overloading and Friend Functions

in effect

```
s = u + v;
```

is equivalent to

```
s = u.operator+(v);
```

which is why there is only one operand for a binary operator!

Classes and Objects

Operator Overloading and Friend Functions

Note that we are not always free to choose between a member function and a friend function for operator overloading:

C++ requires that the following operators can only be overloaded using member functions (we cannot define friend functions for them)

`=`, `[]`, `()`, `->`

Classes and Objects

Operator Overloading and Friend Functions

Overloading applied to unary operators

Define the minus sign as the unary operator for vectors:

```
vector u, v;  
...  
v = -u;
```

and, from which, we can then proceed to define a binary minus operator since:

$$a - b = a + (-b)$$

Classes and Objects

Operator Overloading and Friend Functions

// UNARY: An unary operator, along with two binary ones

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

```
class vector {
public:
    vector(float xx=0, float yy=0)
    {   x = xx; y = yy;
    }
    void printvec() const;
    vector operator+(vector &b); // binary plus
    vector operator-();         // unary minus
    vector operator-(vector &b); // binary minus

private:
    float x, y;
};
```

Classes and Objects

Operator Overloading and Friend Functions

```
void vector::printvec()const
{   cout << x << ' ' << y << endl;
}
```

```
vector vector::operator+(vector &b) // Binary plus
{
    return vector(x + b.x, y + b.y);
}
```

```
vector vector::operator-() // Unary minus
{
    return vector(-x, -y);
}
```

```
vector vector::operator-(vector &b) // Binary minus
{
    return *this + -b; // recall 'this' is a pointer to
                       // the current object
}
```

Classes and Objects

Operator Overloading and Friend Functions

```
int main()
{   vector u(3, 1), v(1,2), sum, neg, diff;
    sum = u + v;
    sum.printvec();    // what's the output?
    neg = -sum;
    neg.printvec();    // what's the output?
    diff = u - v;
    diff.printvec();   // what's the output?
    return 0;
}
```

Classes and Objects

Copying a Class Object

A class object that contains a pointer to dynamically allocated memory can be copied in two ways:

Shallow copy

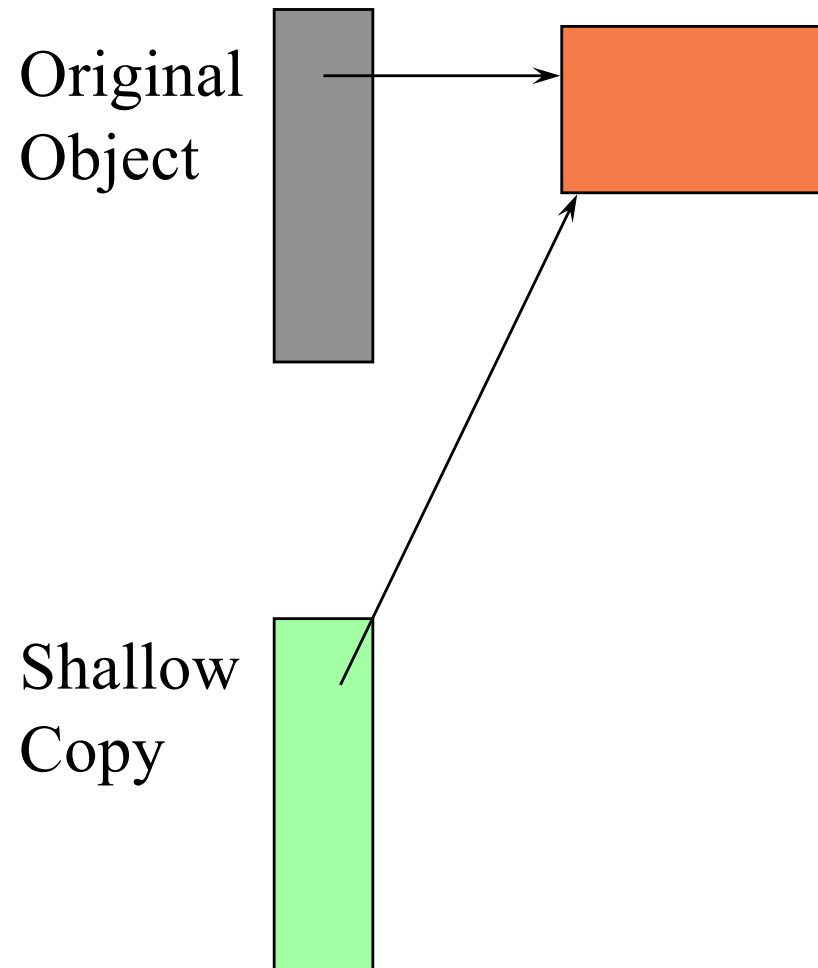
where the class contains only member functions and 'simple' data members (which are not classes)

copying is by default done 'bitwise'

all members, including the pointers, are copied literally

Classes and Objects

Copying a Class Object



Classes and Objects

Copying a Class Object

A class object that contains a pointer to dynamically allocated memory can be copied in two ways:

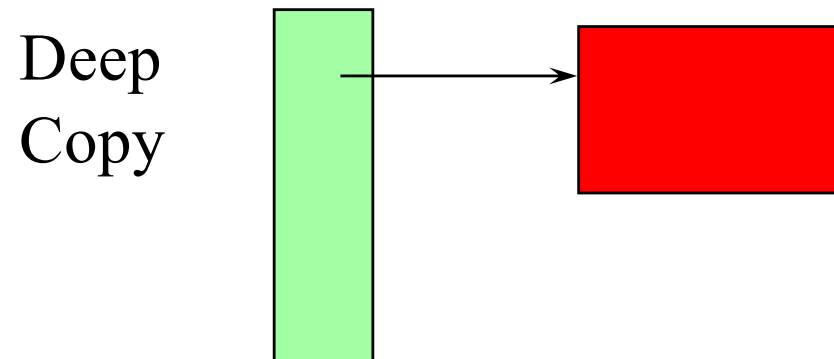
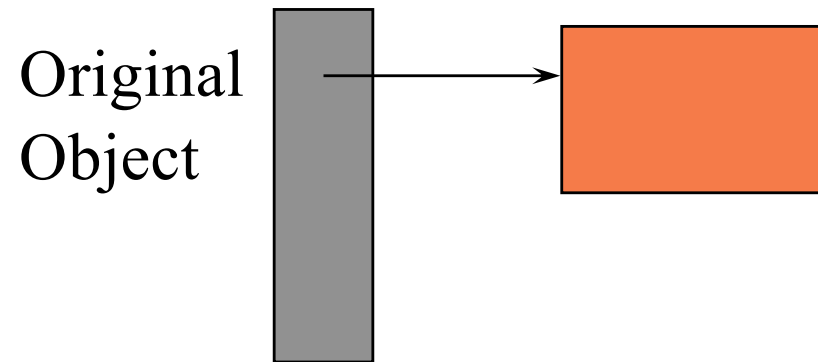
- Deep copy

 - all members are copied

 - but the pointer and the data to which it points are replicated

Classes and Objects

Copying a Class Object



Classes and Objects

Copying a Class Object

The difference between deep and shallow copies is important when the referenced memory is allocated by a constructor and deleted by a destructor

why?

because the shallow copies will also be effectively deleted by the destructor (and, anyway, attempting to delete the same thing twice is dangerous and illegal)

Classes and Objects

Copying a Class Object

If we require deep copies, then we must take care to define the constructors and assignments appropriately

Typically, we will define a **copy constructor** as a (function) member of a class for copying operations other than by assignment

This copy constructor will be declared in the following way:

```
class_name(const class_name &class_object);
```

parameter is always a reference parameter

Classes and Objects

Copying a Class Object

For copying by assignment, we must define an assignment operator to prevent shallow copying

```
class_name operator=(class_name class_object);
```

Object-Oriented Programming

Object-oriented programming - many definitions

exploitation of class objects, with private data members and associated access functions (cf. concept of an abstract data type)

However, Ellis and Stroustrup give a more limited meaning:

‘The use of derived classes and virtual functions is often called object-oriented programming’

So, we need some more C++!

Object-Oriented Programming

Interface, Implementation, and Application Files

Preferred practice for programs dealing with classes: 3 files

Interface

- between implementation and application

- Header File that declares the class type

- Functions are declared, not defined (except inline functions)

Implementation

- `#includes` the interface file

- contains the function definitions

Application ...

Object-Oriented Programming

Interface, Implementation, and Application Files

Preferred practice for programs dealing with classes: 3 files

Interface

Implementation

Application

`#includes` the interface file

contains other (application) functions, including the `main` function

Object-Oriented Programming

Interface, Implementation, and Application Files

When writing an application, we are class users

don't want to know about the implementation of the class (c.f. ADTs)

Thus, the interface must furnish all the necessary information to use the class

Also, the implementation should be quite general (cf. reusability)

Object-Oriented Programming

A Class for Sets

Required data type: sets of integers

Required functions:

declaration, e.g.

```
iset S, T=1000, U=T, V(1000);
```

S should be empty

T, U, and V should contain just one element (1000)

adding an element, e.g.

```
S += x
```

removing an element, e.g.

```
S -= x
```

must be valid even if x is not an element of S

Object-Oriented Programming

A Class for Sets

Required data type: sets of integers

Required functions:

Test if an element is included in the set, e.g.

```
if (S(x)) ... // is x in set X
```

Display all elements in increasing order, e.g.

```
S.print();
```

Set assignment, e.g.

```
T = S
```

Object-Oriented Programming

A Class for Sets

Required data type: sets of integers

Required functions:

Inquiring about the number of elements by converting to type int, e.g.

```
cout << "S contains " << int(S) << "elements";
```

Inquiring if the number of elements in the set is zero, e.g.

```
if (S==0) ....
```

```
if (!S) ....
```

Object-Oriented Programming

A Class for Sets

Note that we have NOT specified how the set is to be represented or implemented
(again, cf. ADTs)

Object-Oriented Programming

A Class for Sets - Application

```
// SETAPPL: Demonstration program for set operations
//           (application; file name: setappl.cpp)

#include "iset.h"

int main()
{   iset S=1000, T, U=S;
    if (!T) cout << "T is empty.\n";
    if int(U) cout << "U is not empty.\n";
    S += 100;   S += 10000;
    (((s += 10) +=1) += 20) += 200;
    cout << "There are " << int(S) << "elements in S\n";
    T += 50;   T += 50;
    cout << "S: "; S.print();
    S -= 1000; cout <<"1000 removed from S\n";
    if (S(1000))
        cout << "1000 belongs to S (error)\n";
    else
        cout << "1000 is no longer in S\n");
```

Object-Oriented Programming

A Class for Sets - Application

```
    if (S(100))
        cout << "100 still belongs to s\n";
    cout << "S: "; S.print();
    cout << "T: "; T.print();
    cout << "U: "; U.print();
    T = S;
    cout << "After assigning S to T, we have T: ";
    T.print();
    return 0;
}
```


Object-Oriented Programming

A Class for Sets - Application

Expected output

```
T is empty
U is not empty
There are 7 elements in S
S: 1 10 20 100 200 1000 10000
1000 removed from S
1000 is no longer in S
100 still belongs to S
S: 1 10 20 100 200 10000
T: 50
U: 1000
After assigning S to T, we have T:1 10 20 100 200 10000
```

Object-Oriented Programming

A Class for Sets - Application

```
// ISET.H:  Header file for set operations
//          (interface; file name: iset.h)

#include "iostream.h"

class iset {
public:
    iset()                // constructor to begin
    {   a = NULL;        // with empty set
        n = 0;
    }
    iset(int x)           // constructor to begin
    {   a = NULL;        // with one element x
        *this += x;
        n = 1;
    }
    ~iset()              // destructor
    {   delete[] a;
    }
```

Object-Oriented Programming

A Class for Sets - Application

```
    iset &operator+=(int x); // adds x to the set
    iset &operator-=(int x); // removes x from the set
    int operator()(int x) const; // is x in the set?
    void print() const;          // prints all elements of
                                // the set on one line

    iset &operator=(iset S); // assignment operator
    iset (const iset &S);    // copy constructor NB
    operator int()           // convert iset to int
    { return n;
    }
private:
    int n, *a;
}
```

Object-Oriented Programming

A Class for Sets - Application

```
// ISET:  Implementation file for set operations;  
//        (implmentation; file name: iset.cpp)  
// Note:  
// using a relatively inefficient array representation  
  
#include "iset.h"  
  
const int blocksize=5;  // may be replaced with larger  
                        // value
```

Object-Oriented Programming

A Class for Sets - Application

```
static int *memoryspace(int *p0, int n0, int n1)

/* if p0 == NULL, allocate an area for n1 integers */
/* if p0 != NULL, increase or decrease old sequence */
/* p0[0], ... , p[n0-1] */
/* in either case, the resulting new sequence is */
/* p1[0], ..., p1[n1-1], and p1 is returned */

{   int *p1 = new int[n1];
    if (p0 != NULL) // copy from p0 to p1:
    {   for (int i=(n0<n1?n0:n1)-1; i>=0; i--)
        p1[i] = p0[i];
        delete p0;
    }
    return p1;
}
```

Object-Oriented Programming

A Class for Sets - Application

```
int binsearch(int x, int *a, int n)

/* The array a[0], ... , a[n-1] is searched for x
   Return value:
       0 if n == 0 or x <= a[0]
       n if x > a[n-1]
       i if a[i-1] < x <= a[i]
*/

{   int m, l, r;
    if (n == 0 || x <= a[0]) return 0;
    if (x > a[n-1]) return n;
    l = 0; r = n-1;
    while (r - l > 1)
    {   m = (l + r)/2;
        (x <= a[m] ? r : l) = m;    // ouch! real C!
    }
    return r;
}
```

Object-Oriented Programming

A Class for Sets - Application

```
iset &iset::operator+=(int x)
{  int i=binsearch(x, a, n), j;  // !!
   if (i >= n || x != a[i]) // x is not yet in set?
   {  if (n % blocksize == 0)
       a = memspace(a, n, n+ blocksize);
       for (j=n; j>i; j--)
           a[j] = a[j-1];
       n++;
       a[i] = x;
   }
   return *this;
}
```

Object-Oriented Programming

A Class for Sets - Application

```
iset &iset::operator-=(int x)
{  int i=binsearch(x, a, n), j;  // !!
    if (i < n && x == a[i])
    {  n--;
        for (j=i; j<n; j++)
            a[j] = a[j-1];
        if (n % blocksize == 0)
            a = memspace(a, n+1, n);  // release one
                                        // block
    }
    return *this;
}
```


Object-Oriented Programming

A Class for Sets - Application

```
void iset::print() const
{   int i;
    for (i=0; i<n; i++)
        cout << a[i] << " ";
}
```

```
void iset::operator()(int x) const
{   int i=binsearch(x, a, n);
    return i < n && x == a[i];
}
```

```
static int *newcopy(int n, int *a)
// copy a[0], ..., a[n-1] to a newly allocate area
// and return the new start address
{   int *p = new int[n];
    for (int i=0; i<n; i++)
        p[i] = a[i];
    return p;
}
```

Object-Oriented Programming

A Class for Sets - Application

```
iset &iset::operator=(iset S) // assignment operator
{
    delete a;
    n = S.n;
    a = newcopy(n, S.a);
    return *this;
}
```

```
iset::iset(const iset &S) // copy constructor
{
    n = S.n;
    a = newcopy(n, S.a);
}
```

Object-Oriented Programming Exercises

13. Implement and test `iset` class as defined

Object-Oriented Programming

Derived Classes and Inheritance

Given a class B, we can derive a new one D
comprising all the members of B
and some new ones beside

we simply refer to B in the declaration of D

B is called the base class

D is called the derived class

the derived class inherits the members of the base
class

Object-Oriented Programming

A Class for Sets - Application

```
/* Consider the new class object geom_obj */

#include <iostream.h>

class geom_obj {
public:
    geom_obj(float x=0, float y=0): xC(x), yC(y){}
    void printcentre() const
    {   cout << xC << " " << yC << endl;
    }
protected:    // new keyword
    float xC, yC;
};

/* not a lot we can do with this class as it stands */
/* we wish to extend it to deal with circles and */
/* squares */
```

Object-Oriented Programming

A Class for Sets - Application

```
/* define derived class objects circle and square */

class circle: public geom_obj { // base class
public:                          // inherit all public
    circle(float x_C, float y_C, float r)
        : geom_obj(x_C, y_C)
    { radius = r;
    }
    float area() const
    { return PI * radius * radius;
    }
private:
    float radius;
};
```

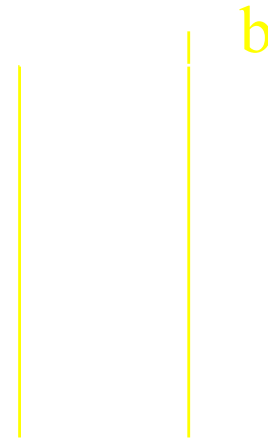
Object-Oriented Programming

A Class for Sets - Application

```
/* define derived class objects circle and square */
/* square defined by its centre and vertex          */
class square: public geom_obj { // base class
public: // inherit all public
    square(float x_C, float y_C, float x, float y)
        : geom_obj(x_C, y_C)
    {
        x1 = x;
        y1 = y;
    }
    float area() const
    {
        float a, b;
        a = x1 - x_C; b = y1 - y_C;
        return 2 * (a * a + b * b);
    }
private:
    float x1, y1;
};
```

Object-Oriented Programming

Derived Classes and Inheritance



Object-Oriented Programming

Derived Classes and Inheritance

`circle` and `square` are extensions of their base class `geom_obj`

the keyword `public` used in the declaration specifies that all public members of the base class `geom_obj` are also regarded as public members of the derived class

`class square` is publicly derived from `geom_obj`

```
square S(3, 3.5, 4.37, 3.85);
```

```
S.printcentre();
```

`printcentre()` is a public class member function of `geom_obj`

Object-Oriented Programming

Derived Classes and Inheritance

`xC` and `yC` are protected members of the base class
`geom_obj`

but they are used in the `area` member function of
`square`

protected members are similar to private ones

- except that a derived class has access to protected members of its base class

- a derived class does not have access to private members of its base class

Object-Oriented Programming

Derived Classes and Inheritance

User-defined assignment operators

if an assignment operator is defined as a member function of a base class, it is not inherited by any derived class

Object-Oriented Programming

Derived Classes and Inheritance

Constructors and destructors of derived and base classes

When an object of (derived) class is created

the constructor of the base class is called first

the constructor of the derived class is called next

When an object of a (derived) class is destroyed

the destructor of the derived class is called first

the destructor of the base class is called next

Object-Oriented Programming

Derived Classes and Inheritance

Constructors and destructors of derived and base classes

We can pass arguments from the constructor in a derived class to the constructor of its base class

normally do this to initialize the data members of the base class

» **write a constructor initializer**

```
class square: public geom_obj {  
public:  
    square(float x_C, float y_C, float x, float y)  
        : geom_obj(x_C, y_C)    // con. init.
```

Object-Oriented Programming

Derived Classes and Inheritance

since the `geom_obj` constructor has default arguments, this initializer is not obligatory here

if we omit it, the constructor of `geom_obj` is called with its default argument values of 0

however, the initializer would really have been required if there had been no default arguments, i.e. if we had omitted the `=0` from the constructor:

```
class geom_obj {  
public:  
    geom_obj(float x=0, float y=0): xC(x), yC(y) {}  
}
```

Object-Oriented Programming

Derived Classes and Inheritance

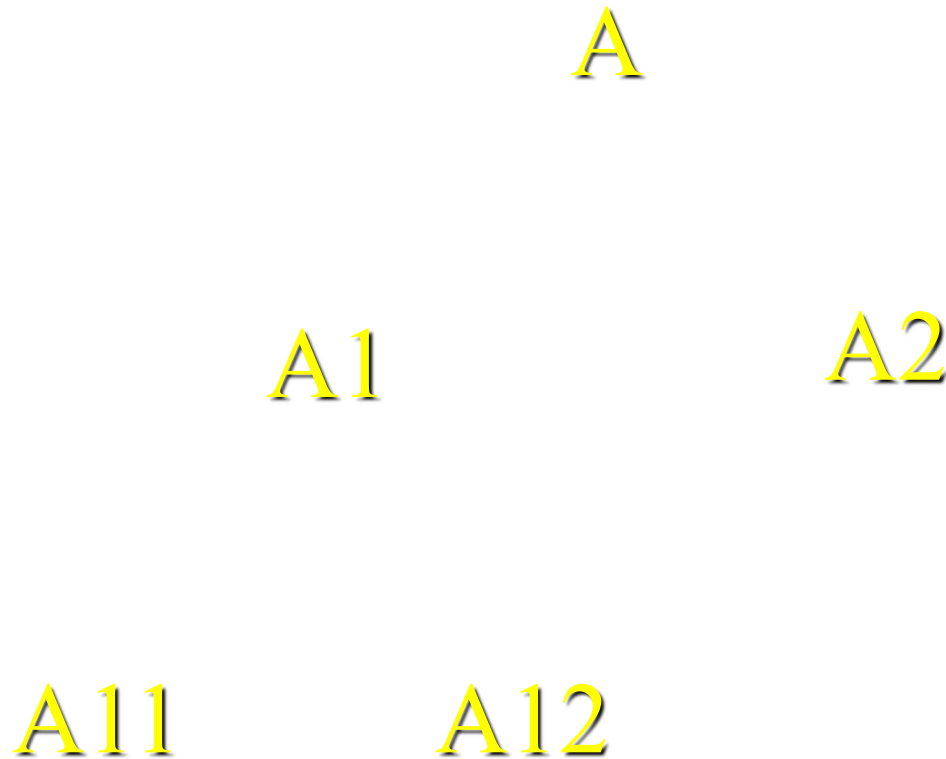
```
int main()
{   circle C(2, 2.5, 2);
    square S(3, 3.5, 4.37, 3.85);
    cout << "Centre of circle: "; C.printcentre();
    cout << "Centre of square: "; S.printcentre();
    cout << "Area of circle:    " << C.area() << endl;
    cout << "Area of square:    " << S.area() << endl;
    return 0;
}
```

```
/* output */
```

```
Centre of circle: 2 2.5
Centre of square: 3 3.5
Area of circle:    12.5664
Area of square:    3.9988
```

Object-Oriented Programming

Derived Classes and Inheritance



A tree of (derived) classes

Object-Oriented Programming

Derived Classes and Inheritance

Conversion from derived to base class

allowed: conversion from derived to base class

NOT allowed: conversion from base to derived

Same applies to corresponding pointer types

why? derived class objects may contain members that do not belong to the base class

Object-Oriented Programming

Derived Classes and Inheritance

```
/* code fragments to illustrate legal and illegal */
/* class conversions */

class B {...};           // base class B
class D: public B{ ...}  // derived class D
...
B b, *pb;
D d, *pd;
...
b = d;           // from derived to base: OK
pb = pd;         // corresponding pointer types: OK
d = b;           // from base to derived: error
d = (D)b;        // even with a cast: error
pd = pb;         // corresponding pointer types: error
pd = (D*)b;      // with cast: technically OK but suspicious
```

Object-Oriented Programming

Derived Classes and Inheritance

Multiple inheritance

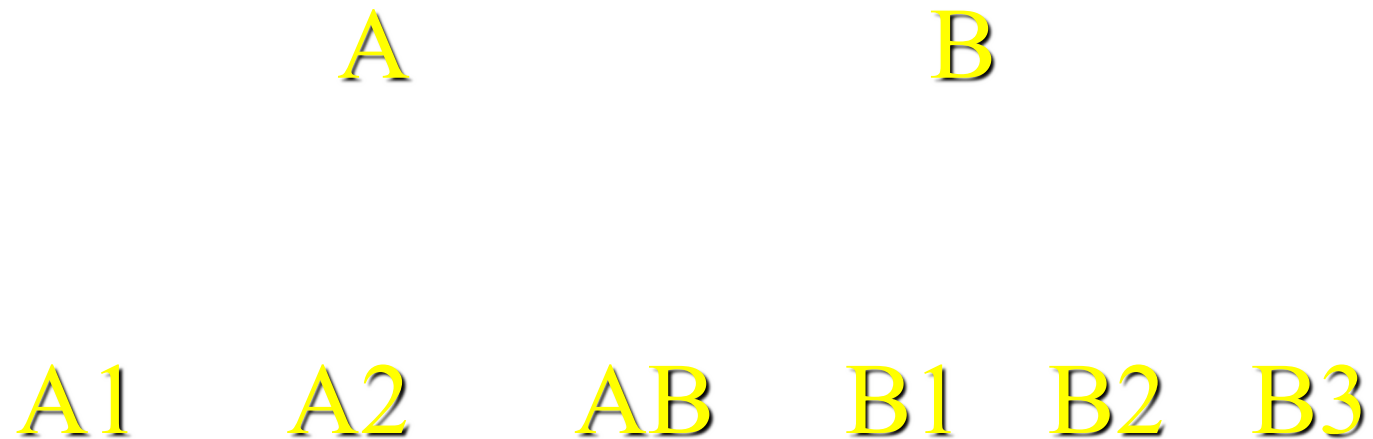
a class can be derived from more than one base class

C++ Release 2

```
class A {...}; // base class A
class B {...}; // base class B
class AB: public A, public B {
...
}
```

Object-Oriented Programming

Derived Classes and Inheritance



Multiple Inheritance

Object-Oriented Programming

Derived Classes and Inheritance

Multiple inheritance

If AB has a constructor with parameters, we can pass these to each base class:

```
class AB {  
public:  
    AB(int n=0, float x=0, char ch='A')  
        :A(n, ch), B(n, x)  
{ ...  
}
```

The creation of an object of class AB causes the three constructors for A, B, and AB, in that order, to be called.

Object-Oriented Programming

Virtual Functions and Late Binding

Suppose we have declared class `ctype` as follows:

```
class ctype {  
public:  
    virtual void f()  
    { ...  
    }  
    ...  
};
```

the keyword `virtual` is important if

`ctype` has derived classes, e.g. `ctype1` and `ctype2`, (i.e. it is a base class for derived classes `ctype1` and `ctype2`)

 and we are using pointers to class objects

Object-Oriented Programming

Virtual Functions and Late Binding

If we define only a pointer to the class and create the class object dynamically

```
ctype *p;  
...  
p = new ctype;
```

the `ctype` object `*p` created in this way is only loosely related to `p`

`p` can also point to the other `ctype` objects

Object-Oriented Programming

Virtual Functions and Late Binding

Let's declare two derived classes

```
class ctype1: public ctype {  
public:  
    void f() {...}  
    ...  
};
```

```
class ctype2: public ctype {  
public:  
    void f() {...};  
};
```

```
ctype *p
```


Object-Oriented Programming

Virtual Functions and Late Binding

since conversion of pointer to derived class to pointer to base class is allowed we can have:

```
p = new ctype1
```

```
/* or */
```

```
p = new ctype2
```

the three class types `ctype`, `ctype1`, and `ctype2` have member functions with the same name (`f`)

`f` is a virtual function

Object-Oriented Programming

Virtual Functions and Late Binding

$f()$ is a virtual function

given the function call

$p \rightarrow f()$

the decision as to which if the three possible functions

$c_{type} :: f$

$c_{type1} :: f$

$c_{type2} :: f$

is made at run-time on the basis of type of object pointed to by

p

Object-Oriented Programming

Virtual Functions and Late Binding

This run-time establishment of the link between the function f and the pointer p is called

late binding

dynamic binding

Object-Oriented Programming

Virtual Functions and Late Binding

If the keyword `virtual` had been omitted from the definition of `f` in declaration of `c_type`

only the type of `p` would have been used to decide which function to call, *i.e.*,

`c_type::f` would have been called

This establishment of the link between the function `f` and the pointer `p` is made at compile time and is called

early binding

static binding

Object-Oriented Programming

Virtual Functions and Late Binding

```
// VIRTUAL: A virtual function in action
```

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

```
class animal {  
public:  
    virtual void print() const  
    {   cout << "Unknown animal type\n";  
    }  
protected:  
    int nlegs;  
};
```

Object-Oriented Programming

Virtual Functions and Late Binding

```
class fish: public animal {
public:
    fish(int n)
    { nlegs = n;
    }
    void print() const
    { cout << "A fish has " << nlegs << " legs\n";
    }
};
```

Object-Oriented Programming

Virtual Functions and Late Binding

```
class bird: public animal {
public:
    bird(int n)
    { nlegs = n;
    }
    void print() const
    { cout << "A bird has " << nlegs << " legs\n";
    }
};
```

Object-Oriented Programming

Virtual Functions and Late Binding

```
class mammal: public animal {
public:
    mammal(int n)
    { nlegs = n;
    }
    void print() const
    { cout << "A mammal has " << nlegs << " legs\n";
    }
};
```


Object-Oriented Programming

Virtual Functions and Late Binding

```
int main()
{
    animal *p[4];
    p[0] = new fish(0);
    p[1] = new bird(2);
    p[2] = new mammal(4);
    p[3] = new animal;
    for (int i=0; i<4; i++) // key statement
        p[i]->print();      // which print is called?
    return 0;               //   fish::print
                            //   bird::print
                            //   mammal::print
                            //   animal::print
                            // the choice is made at
                            // run-time
}
```

Object-Oriented Programming

Virtual Functions and Late Binding

```
/* output */
```

```
A fish has 0 legs
```

```
A bird has 2 legs
```

```
A mammal has 4 legs
```

```
Unknown animal type
```

If print had not been defined as virtual, the binding would have been early (static ... at compile time) and the fourth output line would have been printed four times

Object-Oriented Programming

Virtual Functions and Late Binding

- **Object-Oriented Programming**

the use of virtual functions and derived classes

The style of programming is also called **Polymorphism**

objects of different (derived) types

are accessed in the same way

Member functions are sometimes called **methods**

calling an object's member function is referred to as **sending a message to an object**

Object-Oriented Programming

Virtual Functions and Late Binding

- **Suppressing the virtual mechanism**

in the following function call

```
p[1]->print();
```

the function calls the print function for the derived class bird

we can over-ride this with the scope resolution operator

```
p[1]->animal::print();
```

in which case the function calls the print function for the derived class animal

Object-Oriented Programming

Virtual Functions and Late Binding

- **Pure virtual functions and abstract classes**

We can declare the function `print` in the base class as a **pure virtual function** by writing

```
virtual void print() const = 0;
```

This has some important consequences

- the base class now becomes an abstract class

- an abstract class cannot be used to create objects (or this type)

- they can only be used for the declaration of derived classes

Object-Oriented Programming

Virtual Functions and Late Binding

- **Pure virtual functions and abstract classes**

So, for example, the following are illegal:

```
p[3] = new animal; // error  
animal A;           // error
```

Making the base class `animal` abstract makes it impossible to omit any of the print functions in the derived classes

Object-Oriented Programming

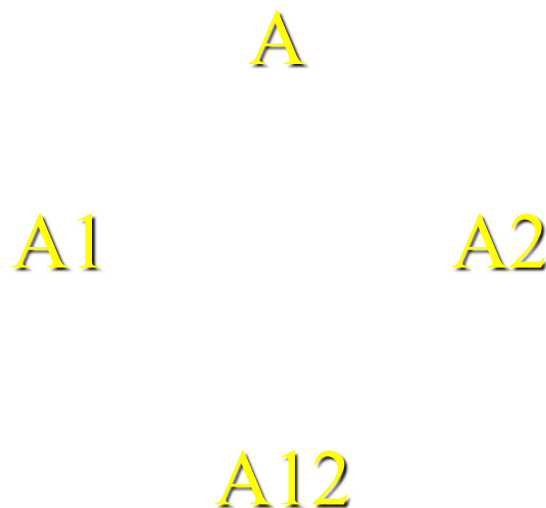
Virtual Functions and Late Binding

- **Virtual base classes**

assume we have a base class A

and derived classes A1 and A2

and a further derived class A12 with multiple inheritance



Object-Oriented Programming

Virtual Functions and Late Binding

- **Virtual base classes**

If base class A had a member a, then

Derived class A12 will inherit two members called a (one through A1 and one through A2)

Such duplication of indirectly inherited members can be suppressed by using the keyword `virtual` in the declarations of A1 and A2

```
class A { ... };  
class A1: virtual public A { ... };  
class A2: virtual public A { ... };  
class A12: public A1, public A2 { ... };
```


Object-Oriented Programming

Virtual Functions and Late Binding

- **Virtual base classes**

A derived class cannot **directly** inherit the members of a base class more than once

```
class A12: public A, public A2 { ... };  
// error
```

Object-Oriented Programming

Static Class Members

- **Static data members**

Normally, a data member of a class type is stored in every object of that type

If we use the keyword `static` for a class data member, however, there will be only one such member for the class, regardless of how many objects there are

a static class member belongs to its class type rather than to the individual objects of that type

Object-Oriented Programming

Static Class Members

- **Static member functions**

cannot use any data members that are specific for objects
the this pointer is not available in static member functions

Object-Oriented Programming

Static Class Members

```
/* STATMEM: Using a static class member to count
           how many times the constructor person()
           is called */
```

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

```
#include <string.h>
```

```
class person {
```

```
public:
```

```
    person (char *str)
```

```
    { strcpy(name, str);
```

```
      count++;    // increment the static counter
```

```
    }
```

```
    void print() const
```

```
    { cout << name << endl;
```

```
    }
```

Object-Oriented Programming

Static Class Members

```
static void printcount() // static function member
{   cout << "There are " << count
    << " persons." << endl;
}

private:
    char name[20];
    static int count; // static data member
};

int person::count=0; // must define (instantiate) the
                    // count member

int main()
{   person A("Mary"), B("Ciana"), C("Georgina"), *p;
    p = new person("Brighid");
    A.print(); B.print(); C.print(); p->print();
    person::printcount();
    return 0;
}
```

Object-Oriented Programming

Static Class Members

the output is as follows:

Mary

Ciana

Georgina

Brighid

There are 4 persons.

Object-Oriented Programming

Static Class Members

- **Static member functions - key points**

must define a static class member outside the class definition
(to instantiate the member)

```
int person::count=0
```

since the static class member is associated with the class and not the object, we must reference the member with the class name and not an object name

```
person::count
```

```
person::printcount()
```

Object-Oriented Programming

Pointers to Members

- **To use pointers to class member functions**
 - use the class name
 - followed by ::
 - as a prefix to the declaration of the pointer
 - which will point to the required class member function

Object-Oriented Programming

Static Class Members

```
class example {
public:
    example (int ii, int jj):i(ii), j(jj) {}
    int ivalue(){return i;}
    int jvalue(){return j;}
private:
    int i, j;
};

int (example::*p) (); // pointer to a member function in
                      // class example (no parameters
                      // and returning int

example u(1,2);
...

p = example::ivalue;
cout << (u.*p) (); // call *p for u ... output is 1
```

Object-Oriented Programming

Polymorphism and Reusability

- Example to demonstrate the power and efficiency of object-oriented programming

Heterogeneous linked-list of geometric shapes

Begin with circles and lines

Extend to triangles

Use virtual function `print`

in an abstract class `element` from which we will derive the appropriate class for each geometry

Lines: defined by two end points

Circles: defined by centre and radius

Object-Oriented Programming

Polymorphism and Reusability

```
/* FIGURES.H: interface file to build linked-list    */
/*           for lines and circles                    */

#include <iostream.h>
#include <stdio.h>

class point {
public:
    float x, y;
    point(float xx=0, float yy=0):x(xx), y(yy) {}
}

class element {                                     // abstract class
public:
    element *next;
    virtual void print() const=0; // pure virtual fn.
}
```

Object-Oriented Programming

Polymorphism and Reusability

```
class line: public element {
public:
    line (point &P, point &Q, element *ptr);
    void print() const;
private:
    point A, B;
};

class circle: public element {
public:
    circle (point &center, float radius, element *ptr);
    void print() const;
private:
    point C;
    float r;
};

void pr(const point &P, const char *str=" ", " ");
```

Object-Oriented Programming

Polymorphism and Reusability

```
// FIGURES: Implementation file (figures.cpp) for
//          linked lists of circles and lines

#include "figures.h"

line::line(point &P, point &Q, element *ptr)
{   A = P;
    B = Q;
    next = ptr;
}

void line::print() const
{   cout << "Line: ";
    pr(A); pr(B, "\n");
}
```

Object-Oriented Programming

Polymorphism and Reusability

```
circle::circle(point &center, float radius,
               element *ptr);
{   C = center;
    r = radius;
    next = ptr;
}

void circle::print() const
{   cout << "Circle: ";
    pr(C);
    cout << r << endl;
}

void pr(const point &P, const char *str)
{   cout << "(" << P.x << ", " << P.y << ")" << str;
}
```

Object-Oriented Programming

Polymorphism and Reusability

```
// FIGURES: sample application file

#include "figures.h"

int main()
{
    element *start=NULL;
    start = new line(point(3,2), point(5,5)), start);
    start = new circle(point(4,4), 2, start);
}
```

Object-Oriented Programming

Polymorphism and Reusability

- Some time later, we may wish to add a triangle type to our systems
- We then write a new interface file
- and a new implementation

Object-Oriented Programming

Polymorphism and Reusability

```
/* TRIANGLE.H: adding a triangle (interface file) */
/*      Class triangle is derived from class element */

class triangle: public element {
public:
    triangle(point &P1, point &P2, point &P3,
            element *ptr);
    void print() const;
private:
    point A, B, C;
}
```

Object-Oriented Programming

Polymorphism and Reusability

```
// TRIANGLE: adding a triangle class
//                (implementation file for triangle.cpp)

#include "figures.h"
#include "triangle.h"

triangle::triangle(point &P1, point &P2, point &P3,
                  element *ptr)
{
    A = P1;
    B = P2;
    C = P3;
    next = ptr;
}

void triangle::print() const
{
    cout << "Triangle: ";
    pr(A); pr(B); pr(C, "\n");
}
```

Object-Oriented Programming

Polymorphism and Reusability

- Later again, we may wish to add the ability to distinguish between lines of different thickness
- Instead of deriving a new class from the base class `element`, we can derive one from the class `line` for example `fatline`
- We then write a new interface file
- and a new implementation

Object-Oriented Programming

Polymorphism and Reusability

```
/* FATLINE.H: additional interface file for fat lines*/

class fatline: public line {
public:
    fatline(point &P, point &Q, float thickness,
            element *ptr);
    void print() const;
private:
    float w;
}
```

Object-Oriented Programming

Polymorphism and Reusability

```
// FATLINE: Implementation file (fatline.cpp) for
//          for thick lines

#include "figures.h"
#include "fatline.h"

fatline::fatline(point &P, point &Q, float thickness,
                element *ptr): line(P, Q, ptr)
    // note the constructor initializer
{
    w = thickness;
}

void fatline::print() const
{
    this->line::print();
    cout << "    Thickness: " << w << endl;
}
```

Object-Oriented Programming

Polymorphism and Reusability

```
// DEMO: This program builds a heterogeneous linked
//      list in which data about a line, a circle,
// a triangle, and a 'fat' line are stored
// To be linked with FIGURES, TRIANGLE, and FATLINE

#include "figures.h"
#include "triangle.h"
#include "fatline.h"

int main()
{   element *start=NULL, *p;
// Build a heterogeneous linked list
start = new line(point(3, 2), point(5, 5), start);
start = new circle(point(4, 4), 2, start);
start = new triangle(point(1, 1), point(6, 1),
                    point(3, 6), start);
start = new fatline(point(2, 2), point(3, 3), 0.2, start);
```

Object-Oriented Programming

Polymorphism and Reusability

```
// DEMO: This program builds a heterogeneous linked
//      list in which data about a line, a circle,
// a triangle, and a 'fat' line are stored
// To be linked with FIGURES, TRIANGLE, and FATLINE

#include "figures.h"
#include "triangle.h"
#include "fatline.h"

int main()
{   element *start=NULL, *p;
    // Build a heterogeneous linked list
    start = new line(point(3, 2), point(5, 5), start);
    start = new circle(point(4, 4), 2, start);
    start = new triangle(point(1, 1), point(6, 1),
                        point(3, 6), start);
    start = new fatline(point(2, 2), point(3, 3), 0.2, start);
```

Object-Oriented Programming

Polymorphism and Reusability

```
for (p=start; p!=NULL; p = p->next)
    p->print(); // polymorphic data handling and
                // late binding!
}
```


Object-Oriented Programming Exercises

14. Modify (and test) the `iset` class as follows
- replace the element addition `+=` with the addition operator `+`
`S = S + x;`
 - replace the element removal `-=` with the assignment operator `-`
`S = S - x;`
 - replace the null set check with the function `isempty()`;
 - add a set union operator `+`
 - add a set intersection operation `*`

Object-Oriented Programming Exercises

15. Modify (and test) the `iset` class as follows
- add a function `null` which removes all elements from the set and returns an empty set
 - replace the inclusion operator `S (x)` with the function `contains()` returning `TRUE` or `FALSE`

Object-Oriented Programming Exercises

16. Modify (and test) the `iset` class
- represent the set with a linear linked list and do insertions with an insertion sort
 - represent the set with a binary tree

Object-Oriented Programming Exercises

17. Create a new set class for character strings
`string_set`