Advanced Object-Oriented Programming Using C++

Module 2: C vs. C++ (Review), OOA / D / P with Rose

By

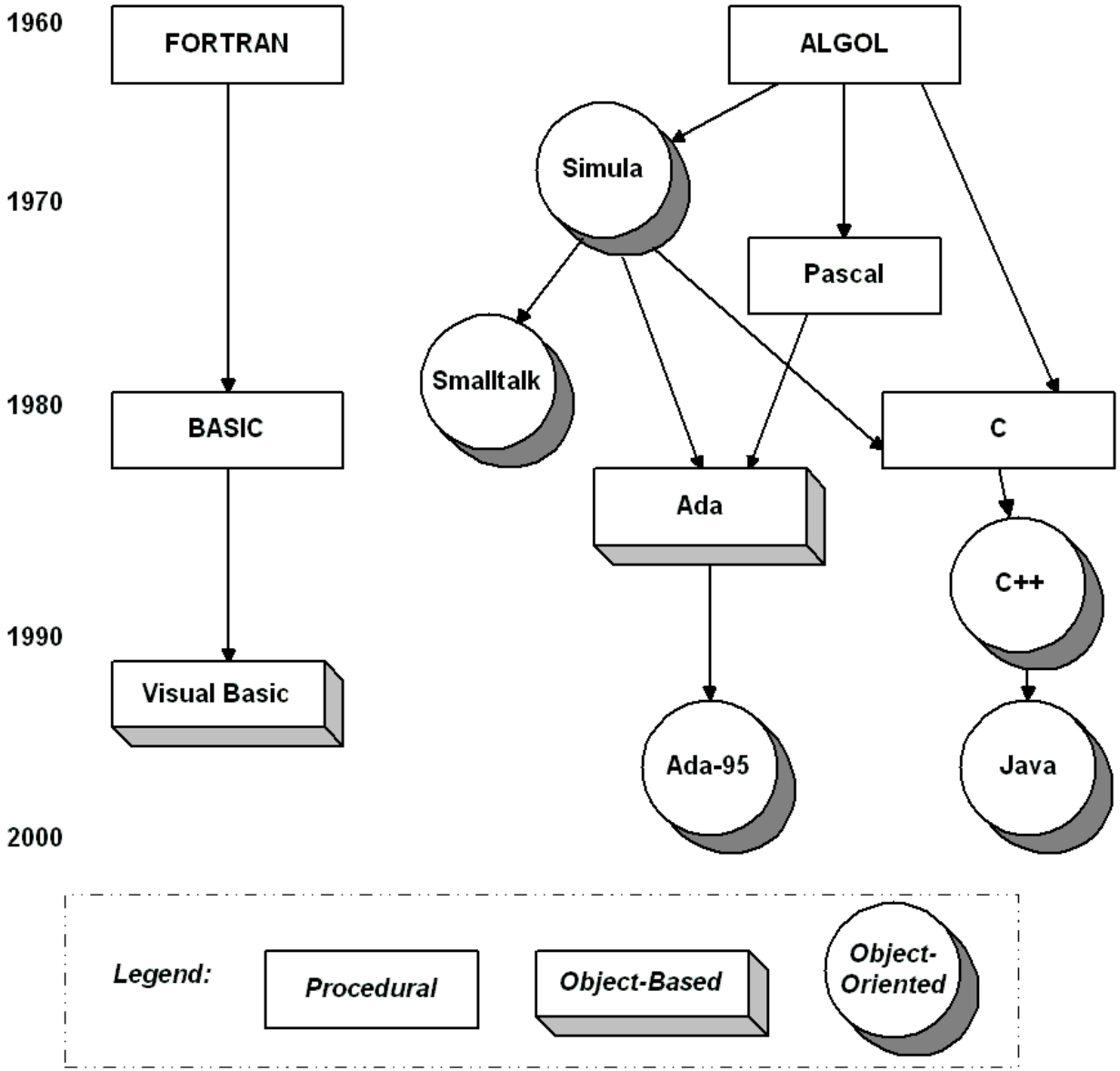
Nicholas Leuci

email: nick@noeticode.com

C / C++ Review, Rational 'Rose'

- **A Very Brief History of OOP Languages**
- **C / C++ Keywords**
- **C / C++ Types**
- **C / C++ Operator Summaries**
- **C / C++ Precedence, Evaluation, and Associativity**
- **C / C++ Basic Grammar and Syntax**
- **C / C++ Constants and Character Sets**
- **Overview of Advanced C++**
- **Introduction to OOA, OOD, OOP with Rational *Rose***

A Very Brief History of OOP Languages



C vs. C++: C Reserved (Key) Words

C is a very compact, small language compared to C++. Compare this set of keywords with those on the next slide:

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

C vs. C++: C++ Reserved (Key) Words

and	**continue**	**goto**	**public**	**try**
and_eq	**default**	**if**	**register**	**typedef**
asm	**delete**	**inline**	**reinterpret_cast**	**typeid**
auto	**do**	**int**	**return**	**typename**
bitand	**double**	**long**	**short**	**union**
bitor	**dynamic_cast**	**mutable**	**signed**	**unsigned**
bool	**else**	**namespace**	**sizeof**	**using**
break	**enum**	**new**	**static**	**virtual**
case	**explicit**	**not**	**static_cast**	**void**
catch	**export**	**not_eq**	**struct**	**volatile**
char	**extern**	**operator**	**switch**	**wchar_t**
class	**false**	**or**	**template**	**while**
compl	**float**	**or_eq**	**this**	**xor**
const	**for**	**private**	**throw**	**xor_eq**
const_cast	**friend**	**protected**	**true**	

C vs. C++: Some Type Bit-Resolutions

Remember that the *Bit-Resolution* for base types is directly related to the underlying hardware architecture. Here are the sizes of fundamental types for MS DOS / WIN32 (*Intel x86*):

Type	*Size*
char, unsigned char, signed char	1 byte
short, unsigned short	2 bytes
int, unsigned int	4 bytes
long, unsigned long	4 bytes
float	4 bytes
double	8 bytes
long double	8 bytes

C vs. C++: Built-in (Standard) Types

```
String:     "\nThis is a character array"

Integer:    1520    -16    (decimal)
            1520L   -16L   (long)
            987654321      (long)
            0221           (octal)
            0221L          (long octal)
            0x2BAD         (hexadecimal)
            0x2BADL        (long hexadecimal)

Enumerated: {beans, pumpkins, squash, oranges}

Floating:   14.     14.1     14.1e0     14.1E+0
            .3      .3e-22   .3E-22     .3e+22
            12e-16  4e+12
```

Fundamental Data Types

```
Integer: signed char, unsigned char,  char
         signed int,  unsigned [int], int
         signed short [int], unsigned short [int], short [int]
         signed long [int],  unsigned long [int],  long [int]

Floating: float, double, long double


Object specification: [sscl] data-type object-name;
```

C vs. C++: Built-in (Standard) Types

Category	*Type*	*Semantics*
Integral	**char**	Type **char** is an integral type that usually contains members of the execution character set — in Microsoft C++, this is ASCII. The C++ compiler treats variables of type **char**, **signed char**, and **unsigned char** as having different types. Variables of type **char** are promoted to **int** as if they are type **signed char** by default.
	short	Type **short int** (or simply **short**) is an integral type that is larger than or equal to the size of type **char**, and shorter than or equal to the size of type **int**. Objects of type **short** can be declared as **signed short** or **unsigned short**. **Signed short** is a synonym for **short**.
	int	Type **int** is an integral type that is larger than or equal to the size of type **short int**, and shorter than or equal to the size of type **long**. Objects of type **int** can be declared as **signed int** or **unsigned int**. **Signed int** is a synonym for **int**.
	long	Type **long** (or **long int**) is an integral type that is larger than or equal to the size of type **int**. Objects of type **long** can be declared as **signed long** or **unsigned long**. **Signed long** is a synonym for **long**.
Floating	**float**	Type **float** is the smallest floating type.
	double	Type **double** is a floating type that is larger than or equal to type **float**, but shorter than or equal to the size of type **long double**.
	long double	Type **long double** is a floating type that is equal to type **double**.

C vs. C++: Unary and Binary Operators

C includes the following unary operators:

Symbol	*Name*
– ~ !	Negation and complement operators
* &	Indirection and address-of operators
sizeof	Size operator
+	Unary plus operator
++ --	Unary increment and decrement operators

Binary operators associate from left to right. C provides the following binary operators:

Symbol	*Name*
* / %	Multiplicative operators
+ –	Additive operators
<< >>	Shift operators
< > <= >= == !=	Relational operators
& | ^	Bitwise operators
&& ||	Logical operators
,	Sequential-evaluation operator

The conditional-expression operator has lower precedence than binary expressions and differs from them in being right associative.

C vs. C++: Operator Precedence

comment	operator	direction	precedence	
Primary Expression	() [] -> . defined	→	1	HIGH
Unary	Post ++ --	→	2	
	Pre ++ -- ! ~ - * + & (type) sizeof	←		
Multiplicative	* / %	→	3	
Additive	+ -	→	4	
Bit Shift	<< >>	→	5	
Relational	< <= > >=	→	6	
Equality	== !=	→	7	
AND	&	→	8	
exclusive OR	^	→	9	
inclusive OR	|	→	10	
logical AND	&&	→	11	
logical OR	||	→	12	
conditional	? :	←	13	
Assignment	= %= += -= *= /= >>= <<= &= ^= |=	←	14	
comma	,	→	15	LOW

C and C++: Complete Operator Set

Scope Resolution:
Scope resolution: ::
Postfix:
Array element: [ ]
Function call: ( )
Type cast: (type)
Member selection: . or –>
Postfix increment: ++
Postfix decrement: —
Unary:
Indirection: *
Address of: &
Logical-NOT: !
One's complement: ~
Prefix increment: ++
Prefix decrement: —
sizeof
new
delete
Comma:
Comma: ,

Multiplicative:
Multiplication: *
Division: /
Modulus: %
Additive:
Addition: +
Subtraction: –
Shift:
Left shift: <<
Right shift: >>
Relational & Equality:
Less than: <
Less than or equal to: <=
Greater than: >
Greater than or equal to: >=
Equal: ==
Not equal: !=
Bitwise:
Bitwise-AND: &
Bitwise-exclusive-OR: ^
Bitwise-inclusive-OR: |

Logical:
Logical-AND: &&
Logical-OR: ||
Assignment:
Assignment: =
Addition: +=
Subtraction: –=
Multiplication: *=
Division: /=
Modulus: %=
Left shift assignment: <<=
Right shift assignment: >>=
Bitwise-AND: &=
Bitwise-exclusive-OR: ^=
Bitwise-inclusive-OR: |=
Conditional:
Conditional: ? :
Pointer to Member:
Pointer to member: .* or –>
Reference:
Reference: &

C vs. C++: Integer Operations

Operators for Type Integer

Arithmetic operators

+	unary plus, addition
−	unary minus, subtraction
*	multiplication
/	division
%	remainder
x=	modify and replace, where x can be +, −, *, /, or %
++	increment
−−	decrement

Logical operators

&&	and
||	or
!	negate
==	equal
!=	not equal
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal

Bitwise operators

&	and
|	or
^	exclusive or
~	negation
>>	right shift
<<	left shift
x=	modify and replace, where x can be &, |, ^, >>, or <<

C vs. C++: Float Operations

Operators for Type Float

Arithmetic operators

+	unary plus, addition
−	unary minus, subtraction
*	multiplication
/	division
x=	modify and replace, where x can be +, −, *, /, or %
++	increment
−−	decrement

Logical operators

&&	and
||	or
!	negate
==	equal
!=	not equal
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal

C vs. C++: Rational Number Operations

Operations Defined for Rational Number Class

Operator	Used to:
assign	Assign a given numerator and denominator to a rational number.
=	Assign one rational number to another.
x=	Perform the arithmetic operation, x on the rational number with the value given by the right side of the expression. x can be either +, −, *, or /.
==	Compare two rational numbers for equality.
!=	Compare two rational numbers for inequality.
+	Add two rational numbers.
−	Subtract two rational numbers.
*	Multiply two rational numbers.
/	Divide two rational numbers.
real	Convert a rational number to a float format.
<<	Overload the output stream operator.

C vs. C++: C++ Operator Precedence

C++ Operator Precedence Hierarchy

[]	Array subscripting
( )	Function invocation
.	Structure field selection
->	Structure field selection using indirection
++, --	Postfix/prefix increment and decrement. Postfix has higher precedence if both occur in the same expression
sizeof	Size of a variable or type (in bytes)
(cast)	Cast to a type
~	Bitwise negation
!	Logical NOT
-	Unary minus
&	Address of
*	Dereference operator (indirection)
*, /, %	Multiply, divide, remainder. Equal precedence
+, -	Addition, subtraction. Equal precedence
<<, >>	Left shift, right shift. Equal precedence
<, >, <=, >=	Inequality testing. Equal precedence
==, !=	Test for equality, inequality. Equal precedence
&	Bitwise AND
^	Bitwise exclusive OR
|	Bitwise OR
&&	Logical AND
||	Logical OR
? :	Conditional operator
=, +=, -=, *=, /=, <<=, >>=, &=, ^=, |=	Assign and replace. Equal precedence
,	Comma operator for sequential evaluation of expressions

C vs. C++: C++ Operator Associativity

C++ Operator Precedence and Associativity

Operator	Name or Meaning	Associativity
::	Scope resolution	None
::	Global	None
[ ]	Array subscript	Left to right
( )	Function call	Left to right
( )	Conversion	None
.	Member selection (object)	Left to right
–>	Member selection (pointer)	Left to right
++	Postfix increment	None
--	Postfix decrement	None
new	Allocate object	None
delete	Deallocate object	None
delete[ ]	Deallocate object	None
++	Prefix increment	None
--	Prefix decrement	None
*	Dereference	None
&	Address-of	None
+	Unary plus	None
–	Arithmetic negation (unary)	None
!	Logical NOT	None
~	Bitwise complement	None
sizeof	Size of object	None
sizeof ( )	Size of type	None
typeid( )	type name	None
(*type*)	Type cast (conversion)	Right to left
const_cast	Type cast (conversion)	None
dynamic_cast	Type cast (conversion)	None
reinterpret_cast	Type cast (conversion)	None

C vs. C++: Statement Types

Statements	Comment/Format
;	(null statement)
break;	(ends do, for, switch, and while)
continue;	(used with do, for, and while)
do	do statement while (expression);
for	for(init,cont,bottom) statement
goto	goto label; label:

if	if (expression)
	statement
	
	if (expression)
	statement
	else if
	statement
	else
	statement

return	return return expression;

switch	switch (integer-expression) {
	case (integer1) : statement
	break;
	case (integer2) : statement
	statement
	break;
	default : statement
	}

while	while (expression)
	statement
	
	while ()

C vs. C++: Statement Types (2)

Statements	Comment/Format
;	(null statement)
break;	(ends do, for, switch, and while)
continue;	(used with do, for, and while)
do	do statement while (expression);
for	for(init,cont,bottom) statement
goto	goto label; label:
if	if (expression) statement

	if (expression)
	  statement
	else if
	  statement
	else
	  statement

C vs. C++: Statement Types (3)

```
return      return                  return expression;

switch      switch (integer-expression)  {
                case (integer1) : statement
                                  break;
                case (integer2) : statement
                                  statement
                                  break;
                default : statement
            }

while       while (expression)      while ()
                statement
```

C vs. C++: Std. C String Functions

Sample Run-Time Library Functions

String Functions

```
char    *strcat((char *) string1, (char *) string2);
char    *strchr((char *) string1, (int) c);
int     *strcmp((char *) string1, (char *) string2);
int     *strcmpi((char *) string1, (char *) string2);
char    *strcpy((char *) string1, (char *) string2);
int     *strcspn((char *) string1, (char *) string2);
char    *strdup((char *) string);
int      strlen((char *) string);
char    *strlwr((char *) string);

char    *strncat((char *) string1, (char *) string2,\
                 (unsigned) n);

int     *strncmp((char *) string1, (char *) string2,\
                 (unsigned) n);

char    *strncpy((char *) string1, (char *) string2,\
                 (unsigned) n);

char    *strnset((char *) string1, (int) c,\
                 (unsigned) n);
```

C vs. C++: Std. C "Unbuffered" File I/O

Low-Level File I/O

```
int     close((int) handle);
int     creat((char *) pathname, (int) pmode);

long    lseek((int) handle, (long) offset,\
              (int) origin);

int     open((char *) pathname, (int) oflag\
           [,(int) pmode]);


int     read((int) handle, (char *) buffer,\
             (unsigned int));

long    tell((int) handle);

int     write((int) handle, (char *) buffer,\
              (unsigned) count );
```

C vs. C++: Std. C "Buffered" File I/O

High-Level File I/O

```
void     clearerr((FILE *) stream);
int      fclose ((FILE *) stream);
int      feof((FILE *) stream);
int      ferror((FILE *) stream);
int      fflush((FILE *) stream);
int      fgetc((FILE  *) stream);

char    *fgets((char *) string, (int) n,\
         (FILE *) stream);

FILE    *fopen((char *) pathname, (char *) type);

int      fprintf((FILE *) stream,
                 (char *) format-string [, arg0, ..., argn]);

int      fputc((int) c, (FILE *) stream);
int      fputs((char *) string, (FILE *) stream);
int      fread((char *) buffer, (int) size,\
               (int) count, (FILE *) stream);

FILE    *freopen((char *) pathname,\
                 (char *) type, (FILE *) stream);

int      fscanf((FILE *) stream,\
                (char *) format-string\
                [, (type *) arg0, ...]);
```

C vs. C++: Std. C "Buffered" File I/O (2)

```
int     fseek((FILE *) stream, (long) offset,\
              (int) origin);

long    ftell((FILE *) stream);

int     fwrite((char *) buffer, (int) size,\
               (int) count, (FILE *) stream);

int     getc((FILE *) stream);
int     getchar();
char   *gets((char *) buffer);
int     printf((char *) format-string [,arg0, ...]);
int     putc((int) c, (FILE *) stream);
int     putchar((int) c);
int     puts((char *) string);

int     scanf((char *) format-string,\
              [, (type *) arg0, ...]);

void    setbuf((FILE *) stream, (char *) ptr);

int     sprintf((char *) buffer, (char *) format-string\
                [,arg0, ...]);

int     sscanf((char *) buffer, (char *) format-string
               [, (type *) arg0, ...]);

int     ungetc((int) c, (FILE *) stream);
int     unlink((char *) pathname);
```

C vs. C++: Std. C Memory Management

Memory Allocation/Deal Location

```
char    *calloc((unsigned) n, (unsigned) size);
void     free((char *) pointer);
char    *malloc((unsigned) size);
char    *realloc((char *) ptr, (unsigned) size);

.fo on
.pa
.fo off
```

C vs. C++: Std. C "printf" Conversions

printf Conversion Format

```
%[flag(s)][width][.precision][l,L]conversion-character
```

flag value	meaning
-	left justify within field
+	display sign of value
blank	if no sign, prepend a blank
#	octal prepend a zero
	hex prepend 0X or 0x
	real display decimal point
	integer no effect

width	minimum size of display field in characters
*	specified as next argument

type of conversion	precision meaning
integer	number of digits
real	digits after decimal point
string	number of characters

C vs. C++: C “printf” Conversions (2)

Conv. Char(s)	Valid Object Type
%c	char
%s	string
%d	int
%o	int
%u	unsigned
%x	int
%X	int
%ld	long
%lo	long
%lu	unsigned long
%lx	long
%a	address

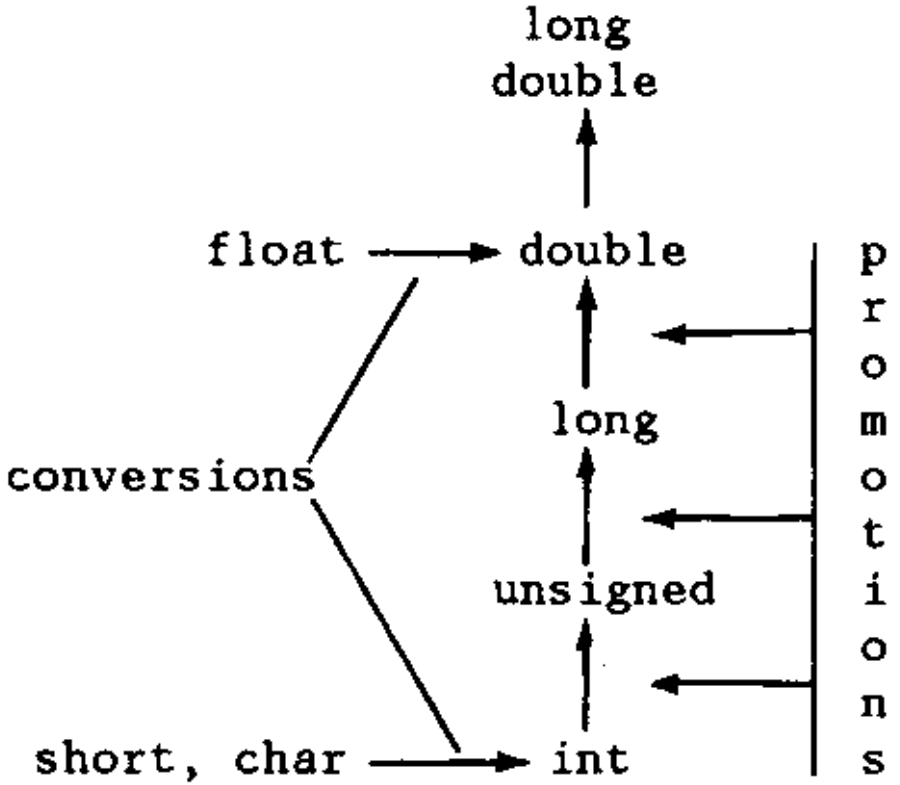
Conv. Char(s)	Valid Object Type
%lX	long
%f	float/double
%e	float/double
%E	float/double
%g	float/double
%G	float/double
%Lf	long double
%Le	long double
%LE	long double
%Lg	long double
%LG	long double

C vs. C++: C “printf” Conversions (3)

Target Object	Conversion Character(s)
char	%c
short	%hd, %ho, %hx
int	%d, %o, %x
long	%D, %ld, %lD, %O, %lo, %lO, %X, %lx, %lX
float	%e, %f
double	%le, %lf, %E, %F
long double	%Le, %Lf,
char array	%nc
string	%s, [...], [^...]

C vs. C++: Implicit Type Conversions

Object Promotion and Conversion



C vs. C++: "Storage Class" Comparison

Storage Class, Initialization, and Visibility

Storage Class	Declared	Initialization	Visibility
auto	in fun.	any expression	local/block
register	in fun.	any expression	local/block
static	in fun.	constant expr.	local/block
	outside fun.	constant expr.	file
extern	in fun.	illegal	local/block
	outside fun.	illegal	file
not declared	in fun.	any expression	local/block
	outside fun.	constant expr.	program

C vs. C++: C struct Syntax

Structures

```
struct sexamp { int  ivar;
                long lvar;
                int  bvar : 5; /* bit field */
                char *msg;
              } instance1 = {2, 8L, 4, "Hi"};

struct sexamp   instance2, sarray[4], *sexamp_ptr[8];

   instance2.ivar = instance1.ivar;
   sarray[2].ivar = instance1.ivar;
  *sexamp_ptr[2] = &instance1;
 (*sexamp_ptr[2]).lvar = 5L;
   sexamp_ptr[2]->lvar = 5L;
```

C vs. C++: C union Syntax

Unions

```
union uexamp { int   ivar;
               long  lvar;
               float fvar;
             } uinstance1; /* cannot init. */

union uexamp   uinstance2, uarray[4], *uexamp_ptr[8];

   uinstance2.ivar = uinstance1.ivar;
   uarray[2].ivar = uinstance1.ivar;
  *uexamp_ptr[2] = &uinstance1;
 (*uexamp_ptr[2]).lvar = 5L;
   uexamp_ptr[2]->lvar = 5L;
```

C vs. C++: C 'Pointer' Syntax

Declaration of Pointers, Arrays, and Functions

Declaration	Meaning
`long L;`	L is a long variable
`long *L;`	L is a pointer to a long variable
`long L[];`	L is an array of longs
`long L();`	L is function returning a long value
`long *L[];`	L is an array of pointers to longs
`long *L();`	L is a function returning a pointer to a long value
`long (*L)();`	L is a pointer to a function returning a long value
`long (*L)[];`	L is a pointer to an array of longs

C vs. C++: Arrays == Pointers

Arrays and Pointers

```
int  i;
char a[20], *a_ptr;
char b[20][5];

 i = *(&i);
```

```
      a  <-->  &a[0]
   a[i]  <-->  (*(a+i))
b[i][j]  <-->  (*(*(b+i)+j))

*a_ptr--  <-->  *(a_ptr--)
```

C vs. C++: C Preprocessor Directives

Preprocessor Directives

```
#define - defines character substitution strings
          and macros

defined - used in the #if to test for definitions

#elif   - used as an else-if with the #if

#else   - used as an else with the #if

#endif  - indicates end of a #if, #ifdef, #ifndef

#if     - tests constant values

#ifdef  - tests for defined values

#ifndef - tests for undefined values

#include - includes source files

#line   - sets line numbers for symbolic debuggers

#undef  - undefines a #define or command line
          definition
```

C vs. C++: Built-in C Constants

Constants

```
Character : 'a'   '\0'   '0'   '\0227'   '\x0D'
```

Escape sequences	
\a	sound alert
\b	backspace (BS)
\f	form feed (FF)
\n	newline (NL -or- LF)
\t	horizontal tab (HT)
\v	vertical tab
\r	carriage return (CR)
\(NL)	line continuation
\"	double quote (")
\'	single quote (')
\\	backslash (\)
\ddd	octal constant
\xddd	hexadecimal constant
\Xddd	hexadecimal constant

C vs. C++: Built-In ASCII Character Code

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	SIX
^C	3	03	♥	EIX
^D	4	04	♦	EOI
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HI
^J	10	0A	◙	LF
^K	11	0B	♂	VI
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	SLE
^Q	17	11	◄	CS1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	EIB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SIB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Char
32	20	sp
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂†

† ASCII code 127 has the code DEL. Under MS-DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL+BKSP key.

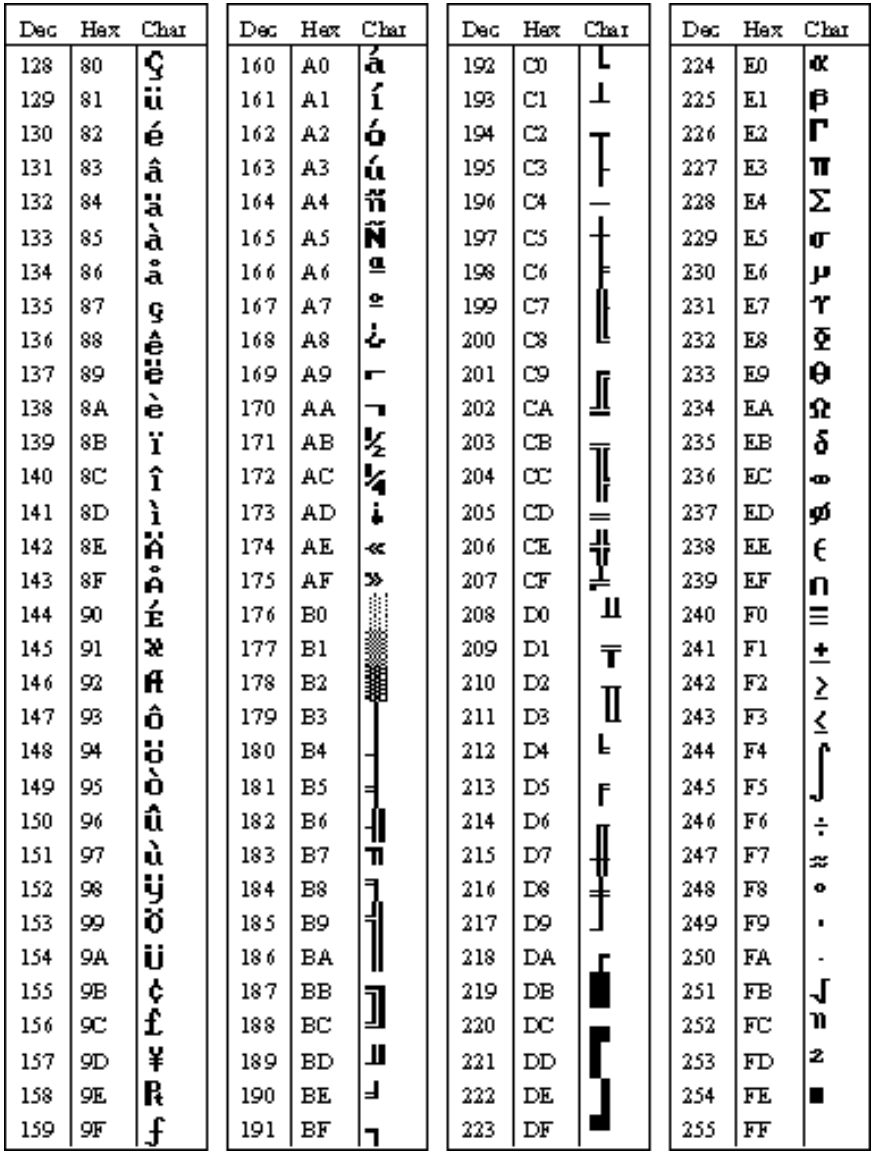
C vs. C++: Built-in ASCII Characters (2)

char	key	dec.	octal	hex	char	dec.	octal	hex
NUL	^@	0	\000	\x00	' '	32	\040	\x20
SOH	^A	1	\001	\x01	'!'	33	\041	\x21
STX	^B	2	\002	\x02	'"'	34	\042	\x22
ETX	^C	3	\003	\x03	'#'	35	\043	\x23
EOT	^D	4	\004	\x04	'$'	36	\044	\x24
ENQ	^E	5	\005	\x05	'%'	37	\045	\x25
ACK	^F	6	\006	\x06	'&'	38	\046	\x26
BEL	^G	7	\007	\x07	'''	39	\047	\x27
BS	^H	8	\010	\x08	'('	40	\050	\x28
TAB	^I	9	\011	\x09	')'	41	\051	\x29
LF	^J	10	\012	\x0A	'*'	42	\052	\x2A
VT	^K	11	\013	\x0B	'+'	43	\053	\x2B
FF	^L	12	\014	\x0C	','	44	\054	\x2C
CR	^M	13	\015	\x0D	'-'	45	\055	\x2D
SO	^N	14	\016	\x0E	'.'	46	\056	\x2E
SI	^O	15	\017	\x0F	'/'	47	\057	\x2F
DLE	^P	16	\020	\x10	'0'	48	\060	\x30
DC1	^Q	17	\021	\x11	'1'	49	\061	\x31
DC2	^R	18	\022	\x12	'2'	50	\062	\x32
DC3	^S	19	\023	\x13	'3'	51	\063	\x33
DC4	^T	20	\024	\x14	'4'	52	\064	\x34
NAK	^U	21	\025	\x15	'5'	53	\065	\x35
SYN	^V	22	\026	\x16	'6'	54	\066	\x36
ETB	^W	23	\027	\x17	'7'	55	\067	\x37
CAN	^X	24	\030	\x18	'8'	56	\070	\x38
EM	^Y	25	\031	\x19	'9'	57	\071	\x39
SUB	^Z	26	\032	\x1A	':'	58	\072	\x3A
ESC	^[	27	\033	\x1B	';'	59	\073	\x3B
FS	^\	28	\034	\x1C	'<'	60	\074	\x3C
GS	^]	29	\035	\x1D	'='	61	\075	\x3D
RS	^^	30	\036	\x1E	'>'	62	\076	\x3E
US	^_	31	\037	\x1F	'?'	63	\077	\x3F

C vs. C++: Built-in ASCII Characters (3)

char	dec.	octal	hex	char	dec.	octal	hex
'@'	64	\100	\x40	' '	96	\140	\x60
'A'	65	\101	\x41	'a'	97	\141	\x61
'B'	66	\102	\x42	'b'	98	\142	\x62
'C'	67	\103	\x43	'c'	99	\143	\x63
'D'	68	\104	\x44	'd'	100	\144	\x64
'E'	69	\105	\x45	'e'	101	\145	\x65
'F'	70	\106	\x46	'f'	102	\146	\x66
'G'	71	\107	\x47	'g'	103	\147	\x67
'H'	72	\110	\x48	'h'	104	\150	\x68
'I'	73	\111	\x49	'i'	105	\151	\x69
'J'	74	\112	\x4A	'j'	106	\152	\x6A
'K'	75	\113	\x4B	'k'	107	\153	\x6B
'L'	76	\114	\x4C	'l'	108	\154	\x6C
'M'	77	\115	\x4D	'm'	109	\155	\x6D
'N'	78	\116	\x4E	'n'	110	\156	\x6E
'O'	79	\117	\x4F	'o'	111	\157	\x6F
'P'	80	\120	\x50	'p'	112	\160	\x70
'Q'	81	\121	\x51	'q'	113	\161	\x71
'R'	82	\122	\x52	'r'	114	\162	\x72
'S'	83	\123	\x53	's'	115	\163	\x73
'T'	84	\124	\x54	't'	116	\164	\x74
'U'	85	\125	\x55	'u'	117	\165	\x75
'V'	86	\126	\x56	'v'	118	\166	\x76
'W'	87	\127	\x57	'w'	119	\167	\x77
'X'	88	\130	\x58	'x'	120	\170	\x78
'Y'	89	\131	\x59	'y'	121	\171	\x79
'Z'	90	\132	\x5A	'z'	122	\172	\x7A
'['	91	\133	\x5B	'{'	123	\173	\x7B
'\'	92	\134	\x5C	'|'	124	\174	\x7C
']'	93	\135	\x5D	'}'	125	\175	\x7D
'^'	94	\136	\x5E	'~'	126	\176	\x7E
'_'	95	\137	\x5F	del	127	\177	\x7F

C vs. C++: Built-in ASCII Characters (4)



Dec	Hex	Char
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	₧
159	9F	ƒ

Dec	Hex	Char
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	▒
177	B1	▒
178	B2	▒
179	B3	│
180	B4	┤
181	B5	╡
182	B6	╢
183	B7	╖
184	B8	╕
185	B9	╣
186	BA	║
187	BB	╗
188	BC	╝
189	BD	╜
190	BE	╛
191	BF	┐

Dec	Hex	Char
192	C0	└
193	C1	┴
194	C2	┬
195	C3	├
196	C4	─
197	C5	┼
198	C6	╞
199	C7	╟
200	C8	╚
201	C9	╔
202	CA	╩
203	CB	╦
204	CC	╠
205	CD	═
206	CE	╬
207	CF	╧
208	D0	╨
209	D1	╤
210	D2	╥
211	D3	╙
212	D4	╘
213	D5	╒
214	D6	╓
215	D7	╫
216	D8	╪
217	D9	┘
218	DA	┌
219	DB	█
220	DC	▄
221	DD	▌
222	DE	▐
223	DF	▀

Dec	Hex	Char
224	E0	α
225	E1	ß
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
231	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	ø
238	EE	ε
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	∙
250	FA	·
251	FB	√
252	FC	ⁿ
253	FD	²
254	FE	■
255	FF	

Overview of Advanced C++

How does C++ differ from C?

C++ is very, very large.

C++ fully contains C.

C is easy to learn.

C is very small.

C is hard to master.

C++ is easy to master.

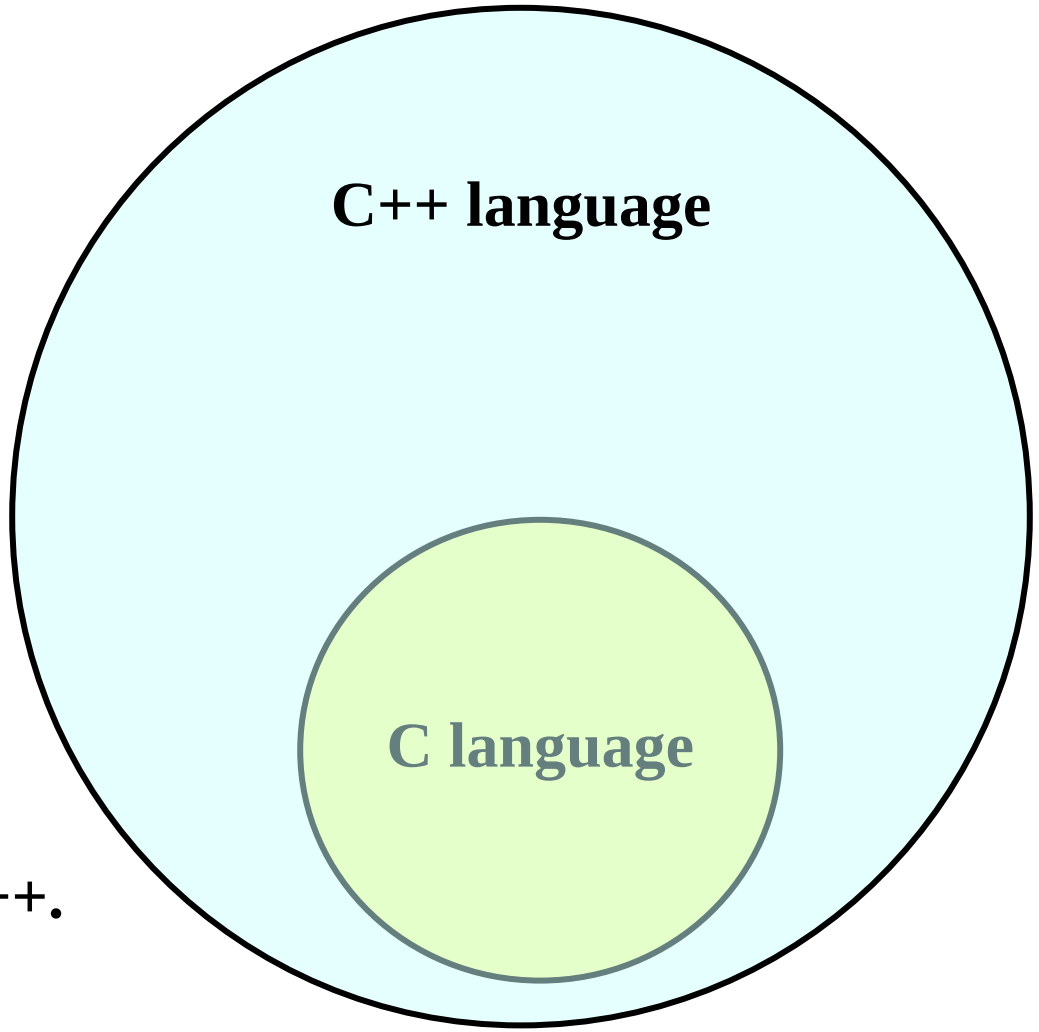
C++ is harder to learn.

C is very weakly typed.

C++ is more strongly typed.

C and C++ are polar opposites.

C is always accessible inside C++.



Overview of Advanced C++

Essential Elements of C++ not in C:

1. **Function prototypes (in ANSI C)**
2. **Function prototypes with unspecified parameters (in ANSI C)**
3. **Pointers to void and functions that return void (in ANSI C)**
4. **Comment delimiter to end of line**
5. **Name of a struct, enum, union, or class is a type name**
6. **Declaration within a block is a statement**
7. **Scope resolution operator**
8. **Const specifier (in ANSI C)**
9. **Anonymous unions**
10. **Explicit type conversions**
11. **Overloading of function names**
12. **Default values for function parameters**
13. **Reference pointers**
14. **Inline specifier**

Overview of Advanced C++

Essential Elements of C++ not in C (continued):

15. New and delete operators
16. Class keyword and Encapsulation and Abstraction
17. Struct as a special case of Class
18. Constructors and Destructors
19. Private, Protected, Public access controls
20. Objects and messages
21. Friends
22. Overloading of operators and functions in classes
23. Inheritance (derived classes)
24. Polymorphism and virtual functions
25. Stream I/O
26. Exceptions
27. Templates, Standard Template Library
28. Logical Scope

Overview of Advanced C++

C++ is based on ANSI C (*NOT* 'K&R' C). The main features of ANSI C now standard in C++ include:

- **Function prototypes -**

C++ requires full argument types for ALL function parameters and return types, which are ALWAYS checked:

```
double vector_sum(float vector[ ], int size) { … return sums; }
```

- **Function prototypes with unspecified parameters -**

```
int count(int rows[ ],...); /*0, 1, or more args OK after first arg*/
```

- **Pointers to void and functions that return void -**

```
void incr(void);  /*  no return value, no parameter */
int * pi = &X;      char * pc = "Hello World";
void * pvi = pi; void * pvc = pc;  printf("%s  %d ", *pvc, *pvi);
/*  all standard pointers can be reliably converted to void * pointers and
    back, without losing data or meaning.  */
```

Overview of Advanced C++

- **Comment delimiter to end of line -**

In addition to the C block-oriented comment delimiters (/* .. */), C++ also has a new comment delimiter, which has effect from the point encountered in the source until the end of that current line:

```
float rate  =  0.0825;  //  this is a comment to the end of line
```

- **Name of a struct, enum, union, or class is a type name -**

In C, a typedef statement is needed to convert any user defined element into a type. This mechanism still exists in C++. However, ANY definition of a user-defined type is AUTOMATICALLY a type definition, not requiring an additional typedef statement.

```
struct Rate_Adj  {  float rate;  int term;  /* etc. */ …};
Rate_Adj  rates[MAX_R];  //  declare an array of the new type
```

Overview of Advanced C++

- **Declaration within a block is a statement -**

C++ allows a declaration of any kind of data anywhere: after code statements, inside of ANY kind of block structure, and even at the head (initialization statement) of a loop.

…

```
x = y;  float pi =  3.1415;  {  struct lesson { … } new_lesson;}
for (int j = 0;  j < MAX_V;  j++) { … }
```

…

- **Scope resolution operator (‘::’) -**

C++ has an operator that allows a variable or function to be ‘fully qualified’ to its proper defintion or outer scope, to resolve naming conflicts and other object identity issues.

```
int x =  3;   //  This has an outer scope
{  int x = 9;  printf(“local x=%d, outer x=%d”,  x,  ::x);  }
//  ‘::x’  fully qualifies ‘x’ to its outer, not local scope
```

Overview of Advanced C++

- **const specifier – (also in ANSI C)**

C++ has a mechanism for creating constants: the 'const' qualifier. This allows any declaration to be made constant. Data declarations qualified as const become read-only. Functions qualified as const allow no 'side-effects'. Function parameters prototyped as const can't be changed inside the function.

```
const float pi = 3.1415;     const int mile = 5280;
const char * greatest_state = "Hawaii";
int query_account(const int account_num) const;
```

- **Anonymous unions –**

Unions without a name can be defined anywhere a variable or field can be defined.

Overview of Advanced C++

- **Explicit type conversions –**

Any valid type name can be used as a function to convert any argument type to the base type name.

```
int points = (int) (5.0 * 100);  //  in C:  points == 500
int points = int(5.0 * 100);   //  in C++ points == 500
account new_acc = account((PRIME + 2.0) * RateFactor);
```

- **Overloading of function names –**

The same function name have many different expressions, via a mechanism called 'name mangling'. Each overloaded function must have a unique prototype signature.

```
compute_sum (int x, int y);
compute_sum (int x, float y);
compute_sum (float x, float y);
```

Overview of Advanced C++

- **Default values for function parameters –**

Any function can have default values specified in its parameter list; when called with a value, the default is ignored, and when called without a value, the default is used.

```
some_func(int x, float varflo=1.17){;}  // 2nd arg has a default
some_func (some_int);              //  this call uses the default
some_func (some_int,  3.1415);  //  this doesn't use default
```

- **Reference pointers –**

C++ has a new pointer type, the 'reference pointer'. Simply stated, a reference pointer must always be bound to an 'L-value', and grammatically, it is coded as though it was an ALIAS of the actual value it is bound to.

```
int x=0;  int & ref = x;   //  ref is a reference bound to int x
ref++;   printf("%d", x);  // this prints '1', not '0'
```

Overview of Advanced C++

- **inline specifier –**

C++ has the means to substitute a code body in place of a function call, allowing for optimized performance. Unlike a macro, both the prototype and function body are seen by the compiler.

```
inline  int  max_xy(int x,  int y)
{  if x > y  ?  return x:  return y;  }
```

- **new and delete operators –**

C++ has new, built-in operators for dynamic memory management, in addition to the standard C malloc library:

new - **allocates an object dynamically from the heap**

delete - **releases an object back to the heap**

delete[ ] - **releases arrays of objects, including strings**

Overview of Advanced C++

- **class keyword and Encapsulation and Abstraction –**

  C++ has the keyword 'class' to allow creating user-defined types that have all the properties of objects (Modularity, Encapsulation, Abstraction, Inheritance). Functions can be defined in a class.

```
class  my_new_class  {  float rate;  int term();  …};
```

- **struct as a special case of class –**

  C++ was designed specifically to be backwards compatible with C. It was essential that C code compile and run in C++ with the same behavior. In C++, structs have their old C meaning, plus they have an EQUIVALENT representation of a class.

```
struct  my_new_struct  {  float rate;  int term();  …};
```

Overview of Advanced C++

- **Constructors and Destructors –**

A class created in C++ always has special functions inside of it dedicated to instantiating and removing class objects from memory.

```
class  my_new_class  {  …
    my_new_class() {;}  //  this is a class constructor
    ~ my_new_class () {;}  //  this is a class destructor
    …};
```

- **private, protected, public access controls –**

A class created in C++ always has access controls to regulate how the class object is accessed at runtime.

```
class  my_new_class  {  …
    private:  float  rate;  //  private means no external access
    public:  float  get_rate();  //  public allows external access
```

Overview of Advanced C++

- **Objects and messages –**

The basic element of C++ is the object. An object responds to messages. A message is a public interface to an object, a method that is used to query and update. All 'public' functions defined in a class are methods that allow objects to 'hear' messages.

- **Friends –**

A class itself or a class member can be declared as a 'friend' to another class, allowing internal access into that other class.

- **Overloading of operators and functions in classes –**

Basic arithmetic, logical and comparative operators can be overloaded as functions for classes, allowing customization for a specific application.

```
int  operator+ (int x, my_new_class * myc)  {
//  a new meaning of '+' for the class my_new_class
```

Overview of Advanced C++

- **Inheritance (derived classes) –**

A new class can be created by deriving it from an existing class. This mechanism is called 'inheritance', and it is a powerful engine of Reusability in all true Object-Oriented Programming Languages.

```
my_child_class  :  my_parent_class { … };
//  creates a new child class from an existing parent class
```

- **Polymorphism and virtual Functions –**

Inheritance can be used to create 'Polymorphism', whereby the same interface in a parent class is bound to many different behaviors in child classes. This is done via 'virtual functions'.

```
class  my_parent_shape_class  { …
virtual int draw() = 0; // this virtual function can have many
                        //  differing child class implementations
```

Overview of Advanced C++

- **Stream I/O –**

C++ has a completely new means for accomplishing input and output. Stream I/O uses dedicated objects for input, output, and input/output. These objects have an intuitive use, hide all hardware details, apply to all base (built-in) types, and are readily extensible to user-defined types.

```
#include <iostream.h>
cout  <<  "Hello, world.";
```

- **Exceptions –**

C++ has new keywords and mechanisms for Active Exception Handling, and other new features for Memory Management.

```
try { … throw(Error_Object);  }
catch (Error_Object)  {
    invoke_error_handler();  }
```

Overview of Advanced C++

- **Templates, Standard Template Library –**

C++ allows creating classes and functions that can have any type instantiated into them. Templates allow defining type-independent sorting, searching, filtering, and manipulating objects and methods. The Standard Template Library (STL) is a huge C++ extension with a vast assortment of reusable templates.

- **Logical Scope –**

C++ allows creating multiple, independent logical scopes eliminating name collisions and promoting large scale development.

```
namespace my_new_scope {
int x;   fn();   Object_t Ob; }
using my_new_scope;
cout << x << "\n" << fn() << "\n" << Ob << "\n
```

OOA / D / P with Rational 'Rose'

We have previously described in some detail the Booch Object Model. Grady Booch's pioneering work in developing a rigorous Object Model was followed by establishing a sound methodology and process for a complete life-cycle approach to Object-Oriented (OO) software development. Included with this is the Booch Notation, the famous (or infamous) 'Cloud Diagrams'. This body of work is documented in the second text for this class.

Other people besides Booch were developing OO notations and methodologies. In the mid 90s a polyglot of notational, methodology, and tool inconsistencies and incompatibilities occurred. Grady Booch was the first at Rational Corp., where he is Chief Scientist, to call for a *Unified* approach to OO development. Booch worked with many others to establish a new uniform OO development standard – UML.

What is UML?

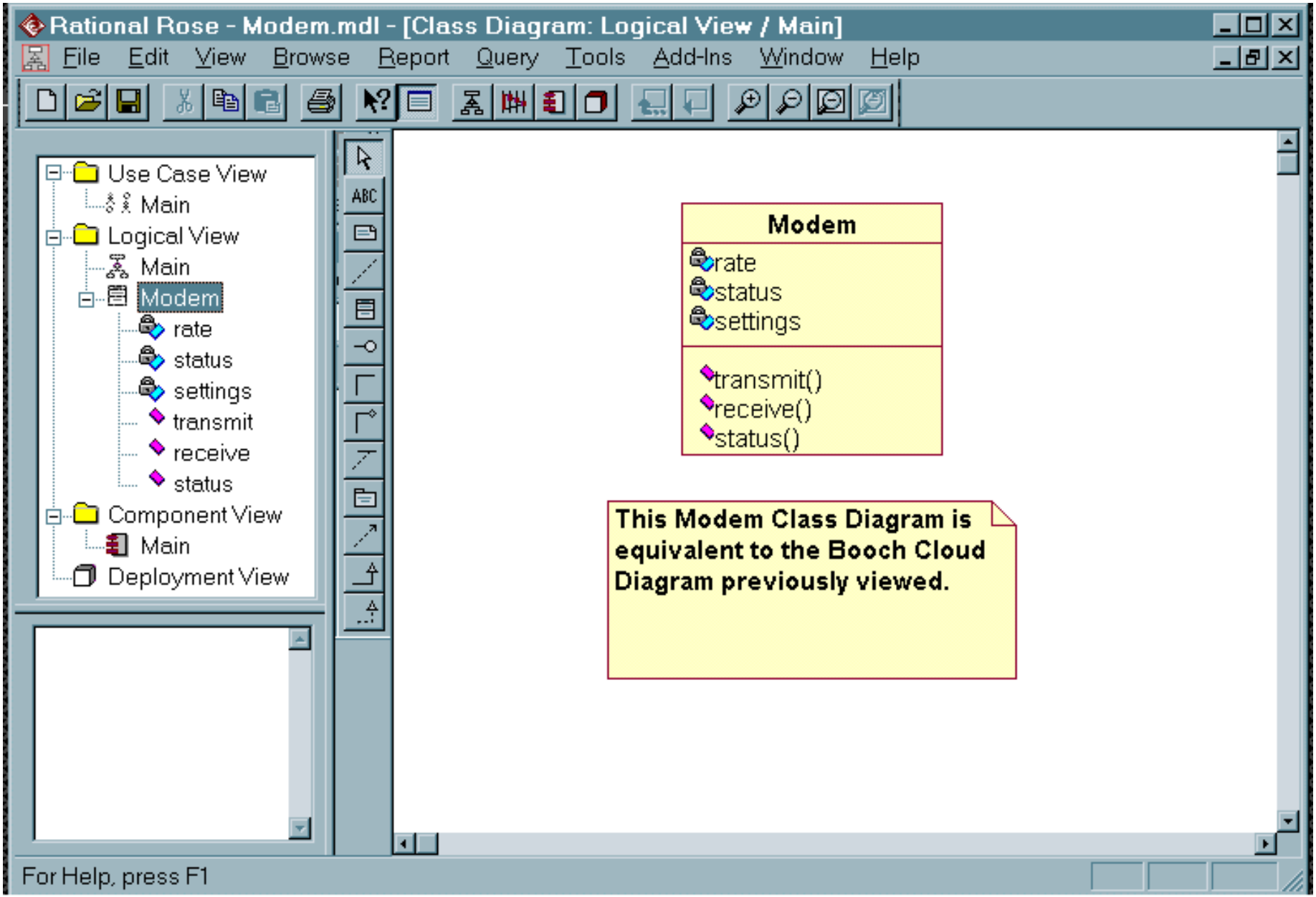
Booch not only provided a rigorous notion of an Object and Object Model, but he also added useful features for real time embedded development. Ivar Jacobson's Use Cases became widely used as Analysis for capturing and modeling requirements for OOD. James Rumbaugh's OMT added data modeling capability, and became widely used by corporations and institutions managing large sets of data (banks, insurance companies, etc.).

Rumbaugh and Jacobson all joined Booch at Rational Corp., and these 'Three Amigos' co-developed a new unified methodology and notation: Unified Modeling Language (UML).

Unified Modeling Language

Rational put UML in the public domain, and as of this writing, it has gone through four major iterations: 1.0, 1.1, 1.2, and 1.3. These feature changes are summarized in Fowler's 'UML Distilled, 2e'. By placing UML in the public domain, major OO tool vendors readily adopted it, and it has gained very rapid acceptance.

The advance is basically eliminating the polyglot confusion and getting OO stakeholders on the same page, with consistent terminology and notation. UML as a methodology is a synthesis of Use Cases, OMT, and Booch. UML as a notation is very similar to OMT, with some features from Booch. Rational has productized UML in its 'Rose' CASE tool.


Rational Rose - Modem.mdl - [Class Diagram: Logical View / Main]
File Edit View Browse Report Query Tools Add-Ins Window Help
Use Case View
Main
Logical View
Main
Modem
rate
status
settings
transmit
receive
status
Component View
Main
Deployment View
Modem
rate
status
settings
transmit()
receive()
status()
This Modem Class Diagram is equivalent to the Booch Cloud Diagram previously viewed.
For Help, press F1

Unified Modeling Language

A Computer-Aided Software Engineering (‘CASE’) tool is meant to automate the design and production of SOURCE CODE. Thus, Rational Rose is a tool that allows Object Modeling using a standard OO notation. It is in one sense a *Drawing* tool: Rose provides palette of controls for visualizing Objects in a variety of hierarchies and associations, with full attributes, including state changes, group interactions, messaging, etc. These drawings are captured in a ‘Repository’, the Rose ‘.mdl’ file.

The Repository is a proprietary database containing OO metadata; it is data about the data in your Object Model. The real power of Rose is in capturing the metadata in your Object Model. *Rose* as a Repository CASE tool is language and notation independent, as it allows an Object Model to be visualized in many notations besides UML, and it can generate classes for many OOP languages, besides C++.

UML and Booch in *Rose*

Rose is also a *Forward and Reverse Engineering* tool (FRE).

'Forward Engineering' means to build an Object Model in the Repository, refining the model until it is suitable, and then generating the source code: the set of C++ classes necessary as a nucleus to grow an application. Rose does not build an application; it generates the source code to create the objects in your Object Model, then you compile them in C++.

'Reverse Engineering' means to start with a set of legacy C++ classes and input them into Rose, which then builds an Object Model and saves it into a Repository, for later refinement.

This full cycle of Forward and Reverse Engineering is also called '*Round Trip Engineering*.' Rose supports both Booch and UML diagrams interchangeably, as well as OMT. In this course, the examples presented will be in the Booch notation.

UML and Booch in *Rose*

Because UML is in the public domain, there are many tools besides *Rose* which use UML as both a methodology and a notation. A comparison of these tools is interesting, but beyond the scope of the current course.

Here we will focus on Booch as productized by Rational in the *'Rose Demo Edition'*. In the next weeks we will examine the notation and methodology rigorously, using Rose to generate the various drawings, capture the Object Model in the Repository, and generate the C++ classes.

Rose is part of a family of tools offered by Rational, including tools to manage requirements, configuration, distribution, software defects, etc. These other tools are also outside of the scope of the current course.

Booch Notation in *Rose*

UML was created by the 'three Amigos': Grady Booch *('Booch')*, James Rumbaugh (*'OMT'*), and Ivar Jacobson (*'Use Cases'*).

Rumbaugh's Object Modeling Technique (OMT) is a direct parent of UML, and the notations are very close. Where OMT differs is in its emphasis on data modeling, a subordinate feature of UML.

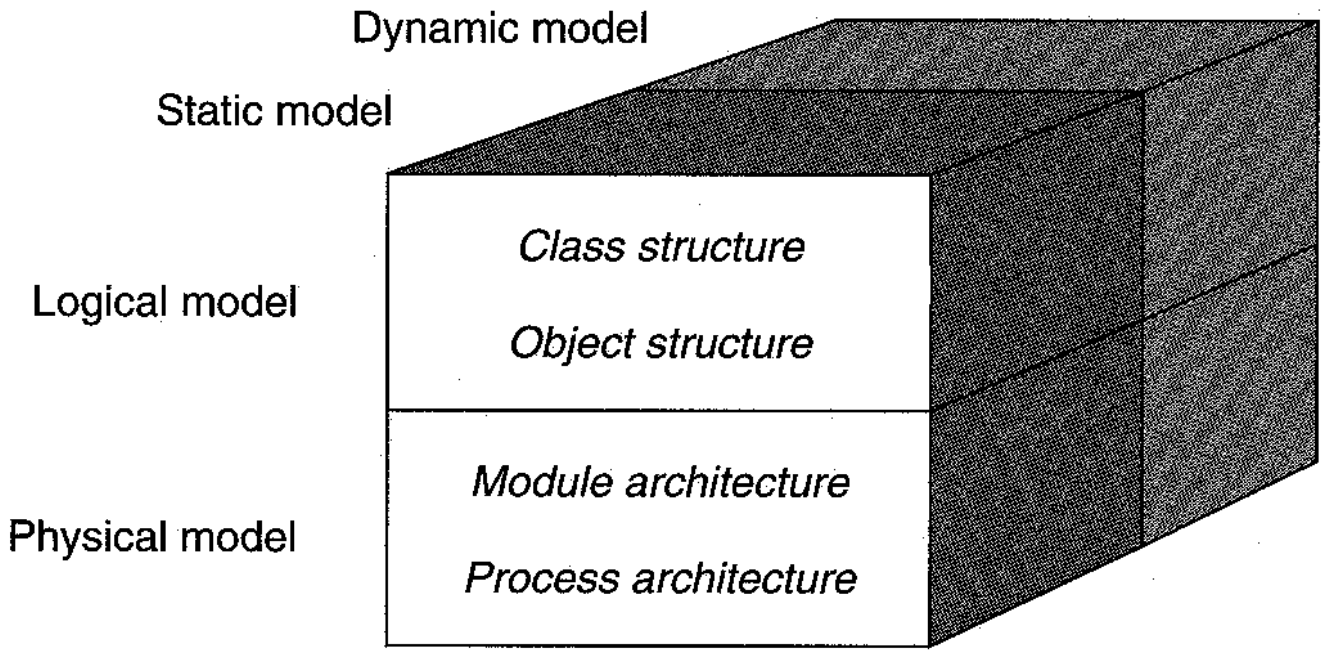
Booch's notation added real time and object concurrency features, which were missing in OMT, plus structural and process views of Object-Oriented Software Development.

Rose allows an Object Model to be drawn in any of these notations, and the user interface for the tool is similar; only the drawing palette some modeling language features change when the model is shown in another notation.

In some ways, UML is a simplification of Booch, which is very large notation and methodology.

Booch Method and Notation

Booch formulates two views of a system under development: the Static Model, which shows architecture and has both a Logical and Physical subview, and the Dynamic Model, which shows behavior.



Booch Process

Booch formulated two development processes for Object-Oriented Analysis and Design, Macro and Micro Processes.

Macro Process

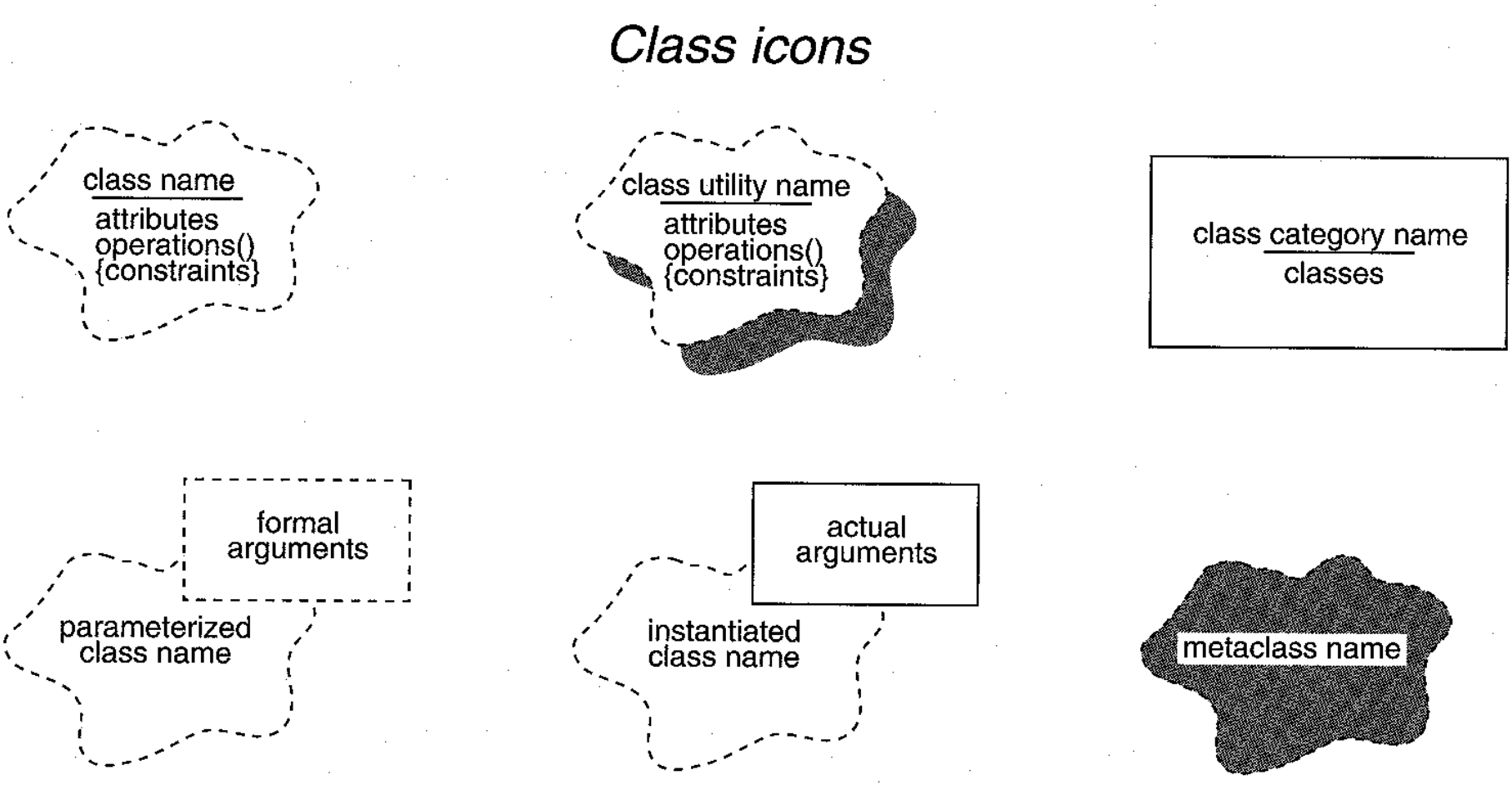
- **Establish core requirements (conceptualization).**
- **Develop a model of the desired behavior (analysis).**
- **Create an architecture (design).**
- **Evolve the implementation (evolution)**
- **Manage postdelivery implementation (maintenance)**

Micro Process

- **Identify classes and objects at a given level of abstraction.**
- **Identify the semantics among these classes and objects.**
- **Identify the relationships of these classes and objects.**
- **Specify the interface and then the implementation of these classes and objects.**

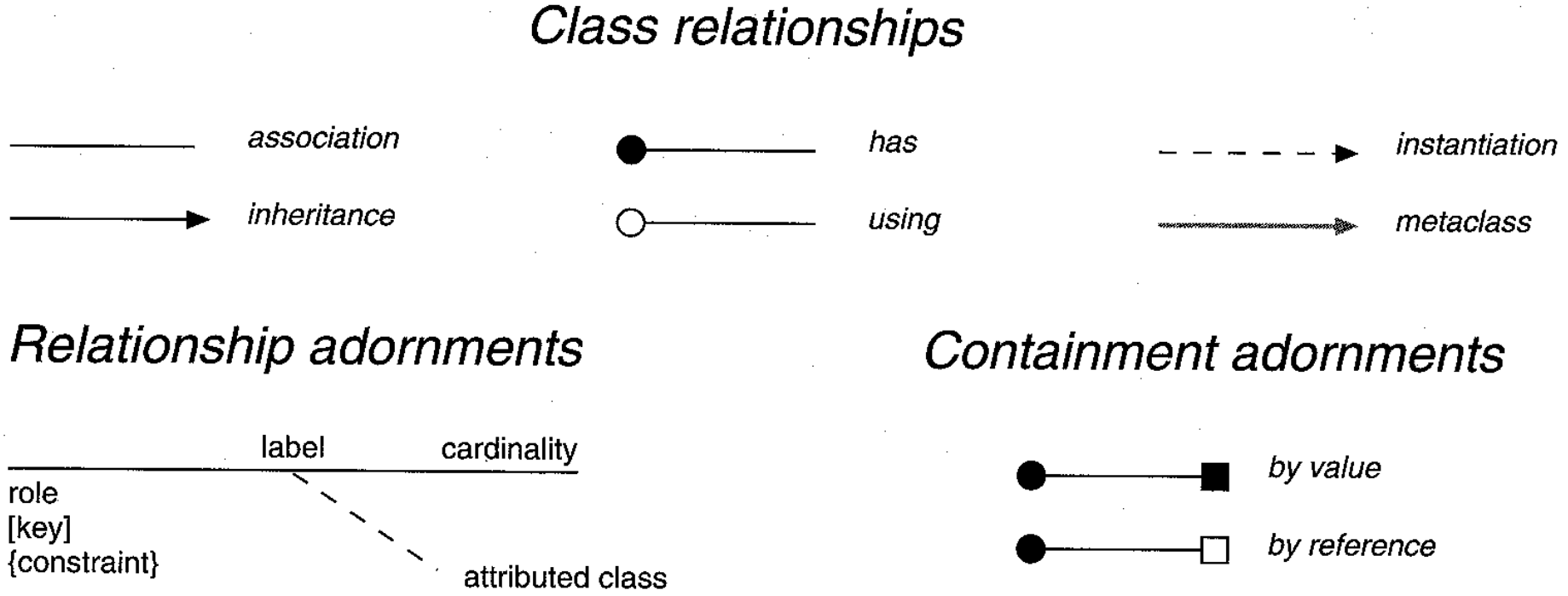
Booch Notation: Class Diagrams

Booch Class Diagrams show the existence of Classes and their relationships in the logical view of the system.



Booch Notation: Class Relationships

Relationships are shown as lines between 'clouds'.
'Adornments' provide extra detail for the diagrammed relationship.



Booch Notation: Class Adornments

Booch is unique in providing visual icons for important properties that are represented textually in other notations.

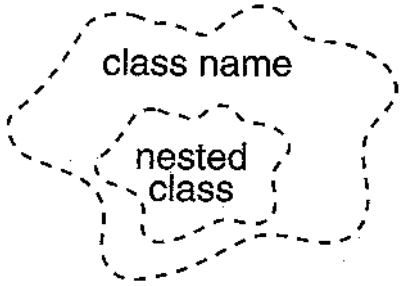
Properties

- A — *abstract class*
- S — *static*
- F — *friend*
- V — *virtual*

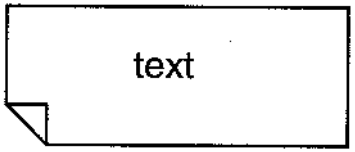
Export control

- *public*
- *protected*
- *private*
- *implementation*

Nesting

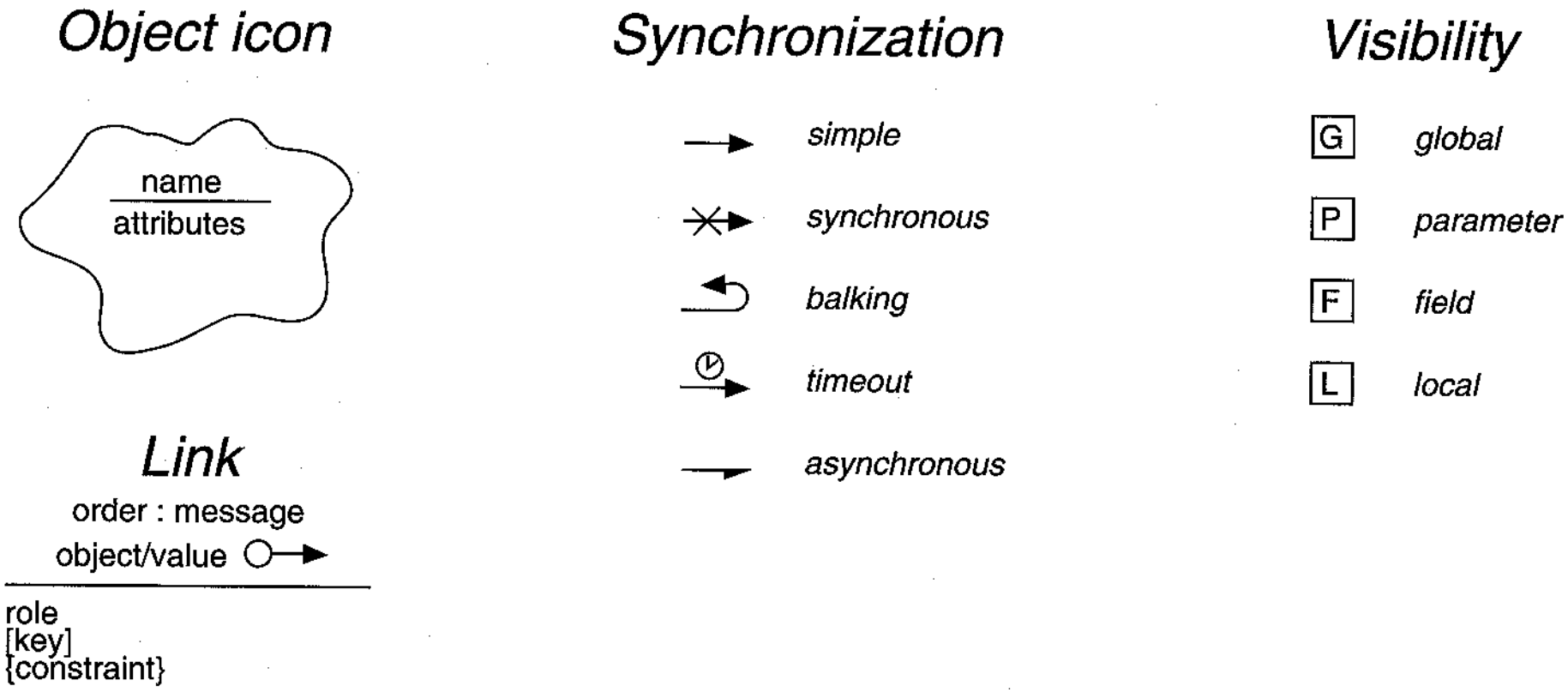


Notes



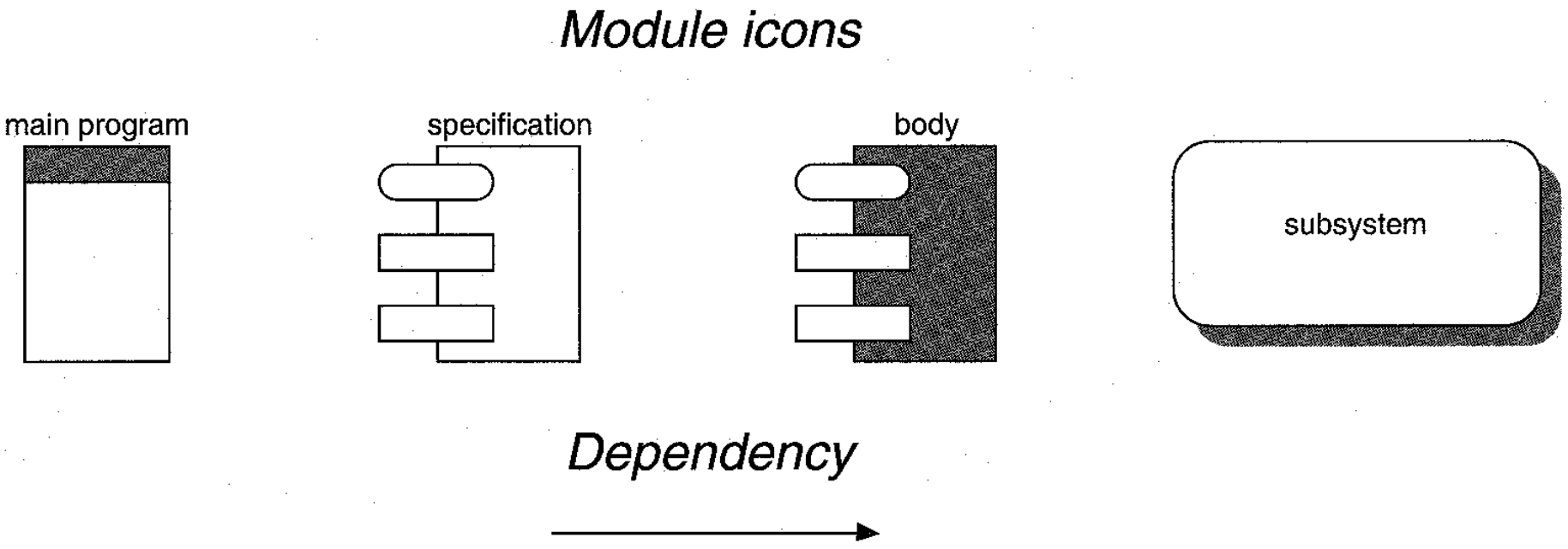
Booch Notation: Object Diagrams

Booch Object Diagrams show the existence of Objects and their relationships in the logical view of the system. Note the fine-grained Synchronization (concurrency) adornments.



Booch Notation: Module Diagrams

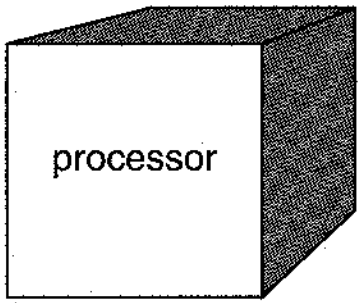
Booch Module diagrams show the allocation of classes and objects to modules in the physical view of the system.



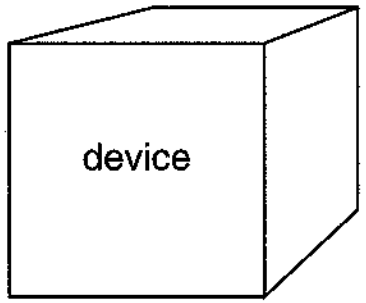
Booch Notation: Process Diagrams

Booch Process diagrams show the allocation of processes to processors in the physical view of a system. This is similar to a system 'riser' and a precursor to the deployment diagram in UML.

Icons



process 1
process 2
...
process n

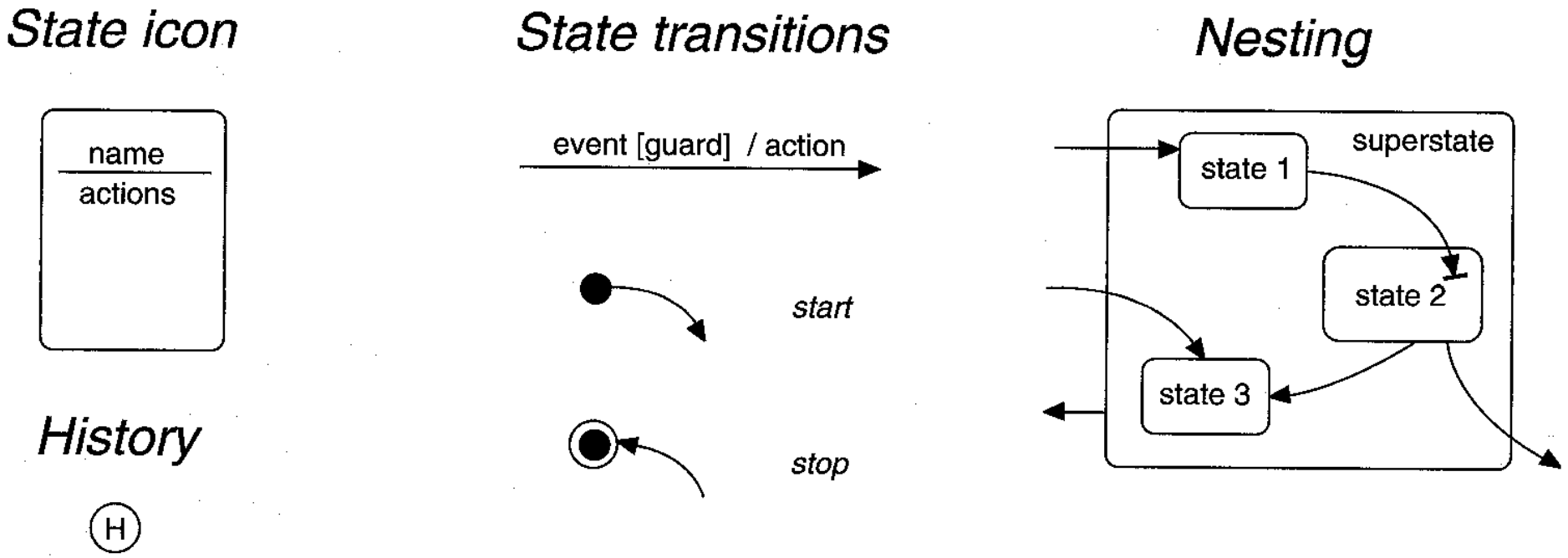


Connection

label

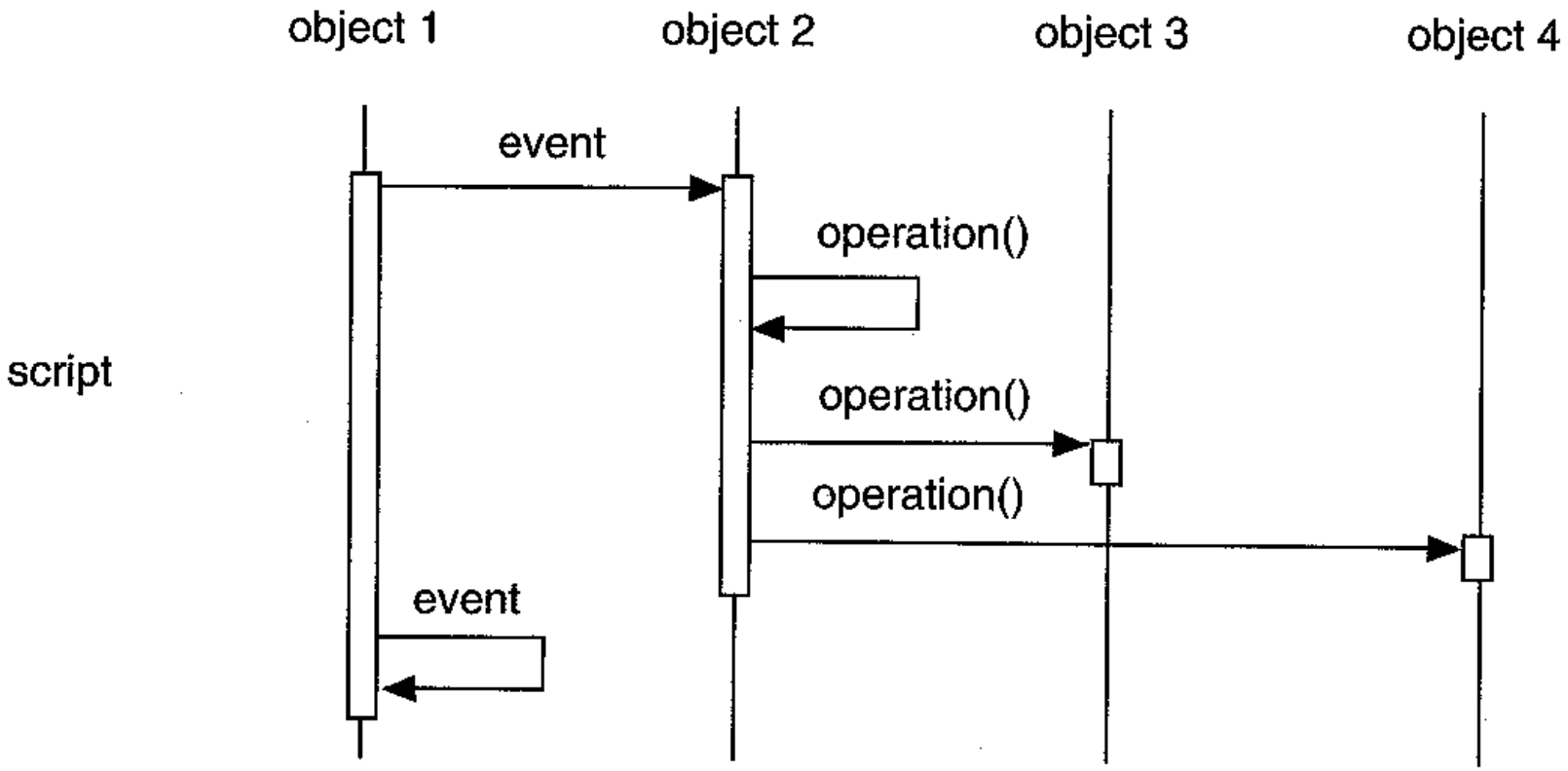
Booch: State Transition Diagrams

Booch State Transition Diagrams are a key element of the Dynamic Model, and were incorporated directly from 'Harel' Diagrams.



Booch: Interaction Diagrams

Booch Interaction diagrams trace the evolution of a 'scenario', a sequence of interactions among objects, and they also specify a 'contract of responsibilities' among objects.



Rose and C++

Rose as a product has modules for many languages, including C.

- **The *Rose Enterprise* tool comes with C++, Java, Visual Basic and Oracle language modules; additional modules can be added, and there is a large 3rd party market for them. The *Rose Demo Edition* used in this class only supports C++ code generation in a Win32 (Windows NT, Windows 95, Windows 98) environment.**

Once an Object Model is built in Rational Rose, it is a simple matter to generate the C++ source code. Many programmers are surprised by code that Rose generates. In general, code generation is weakest part of CASE technology, and this is true for Rose as well. There are large R&D projects ongoing to improve CASE code generation. In this course, we will not concern ourselves with this issue. We will show C++ coding strategies diagrammed in Booch, and nothing more.