University of Waterloo

School of Computer Science

Course Notes

CS 246

Software Abstraction and Specification

http://www.student.cs.uwaterloo.ca/~cs246

Winter 2009

1 Administration

1.1 What CS 246 is about

- C/C++
- UNIX tools
- software design

1.2 Individual Course Work

- **All course assignments are done independently, except final project (pairs).**
- Students may:
  - study together
  - help each other in diagnosing compiler/runtime errors
  - help each other to use tools
- Students may not:
  - discuss assignment answers before due date
  - show work (in progress or completed) to each other
- Students must protect the confidentiality of source code that they develop.

2 C++

2.1 Program Structure

- A C++ program is composed of comments strictly for people, and statements for both people and the preprocessor/compiler.
- A source file contains a mixture of comments and statements.
- The C/C++ preprocessor/compiler only reads the statements and ignores the comments.

2.1.1 Comment

- Comments document what a program does and how it does it.
- A comment may be placed anywhere a whitespace (space, tab, newline) is allowed.
- There are two kinds of comments in C/C++ (same as Java):

	Java / C / C++
1	/* ... */
2	// remainder of line

© Peter A. Buhr

- First comment begins with the start symbol, /*, and ends with the terminator symbol, */, and hence, can extend over multiple lines.
- **Cannot be nested one within another**:

```
/*  ...  /*  ...  */  ...  */
                  ↑     ↑
        end comment     treated as statements
```

- Be extremely careful in using this comment to elide/comment-out code:

```
/* attempt to comment-out a number of statements
while ( ... ) {
    /* ... nested comment causes errors */
    if ( ... ) {
        /* ... nested comment causes errors */
    }
}
*/
```

- Second comment begins with the start symbol, //, and continues to the end of the line, i.e., only one line long.
- Can be nested one within another:

```
// ... // ... nested comment
```

so it can be used to comment-out code:

```
// while ( ... ) {
//     /* ... nested comment does not cause errors */
//     if ( ... ) {
//         // ... nested comment does not cause errors
//     }
// }
```

2.1.2 Statement

- C++ is actually composed of 4 languages:
  1. The preprocessor language (cpp) modifies (text-edits) the program ***before*** compilation .
  2. The template (generic) language adds new types and routines ***during*** compilation .
  3. The C programming language specifying basic declarations and control flow to be executed ***after*** compilation.
  4. The C++ programming language specifying advanced declarations and control flow to be executed ***after*** compilation.

- A programmer uses the four programming languages as follows:

  user edits $\rightarrow$ **preprocessor edits** $\rightarrow$ **templates expand** $\rightarrow$ **compilation**
  ($\rightarrow$ linking/loading $\rightarrow$ execution)

- C is composed of languages 1 & 3.
- A preprocessor statement is a **#** character, followed by a series of tokens separated by whitespace, which is usually a single line and not terminated by punctuation.
- The syntax for a C/C++ statement (both template and regular) is a series of tokens separated by whitespace and terminated by a semicolon. ({} is an exception)

2.2 First Program

- **Java**

```
import java.lang.*; // implicit
class hello {
  public static void main( String[] args ) {
    System.out.println("Hello World!");
    System.exit( 0 );
  }
}
```

- **C++**

```
#include <iostream>      // insert contents of file iostream
using namespace std; // direct naming of I/O facilities

int main() {                 // program starts here
    cout << "Hello World!" << endl;
    return 0;                // return 0 to shell
}
```

- **#include** <iostream> copies basic I/O descriptions (no equivalent in Java).

- **using namespace** std allows imported I/O names to be accessed directly, i.e., ***without*** qualification.
- **int** main() is the routine where execution starts.
- curly braces, { ... }, denote a block of code, i.e., routine body of main.
- cout << `"Hello World!"` << endl prints `"Hello World!"` to standard output, called cout (System.out in Java).
- endl start newline after `"Hello World!"` (println in Java).
- Optional **return** 0 returns zero to the shell indicating successful completion of the program; non-zero usually indicates an error.
- main magic! If no value is returned, 0 is implicitly returned.
- Routine exit (Java System.exit) stops a program at any location and returns a code to the shell, e.g., exit(0).

2.3 Declaration

- Declarations define new variables and types in a program.

2.3.1 Identifier

- name used to refer to a variable or type.
- syntax : (letter | '_') (letter | '_' | digit)*
- **case-sensitive**:

  VeryLongVariableName   Page1   Income_Tax   _75

- Some identifiers are reserved (e.g., **if**, **while**), and hence, **keyword**s.

2.3.2 Basic Types

Java	C / C++	
boolean	**bool** (C <stdbool.h>)	
char	**char** / **wchar_t**	
byte	**char** / **wchar_t**	integral types
int	**int**	
float	**float**	floating-point types
double	**double**	

- C/C++ treat **char** and **wchar_t** (unicode characters) as an integral type.
- Java types **short** and **long** are created using type qualifiers.

2.3.3 Variable Declaration

- Declaration in C/C++ same as Java: type followed by list of identifiers.

Java / C / C++
char a, b, c, d; **int** i, j, k; **double** x, y, z;

- Declarations can be intermixed among executable statements in a block.
- Variable names can be reused in nested blocks, i.e., hide (**override**) names in a containing block.

```
{   int i; . . .        // first i
    {   int i; . . .    // second i
```

- C/C++ do not check for uninitialized variables. (maybe)

```
int i;
if ( i < 3 ) . . .      // i has no value
```

- C/C++ declaration may have initializing assignment (except for fields in **struct**/**class**):

```
int i = 3;
```

2.3.4 Type Qualifier

- C/C++ provide only two basic integral types **char** and **int**.
- Other integral types are generated using type qualifiers.
- C/C++ provide signed (positive/negative) and unsigned (positive only) integral types:

integral types	range
signed char / **char**	at least -127 to 127
unsigned char	at least 0 to 255
signed short int / **short**	at least -32767 to 32767
unsigned short int / **unsigned short**	at least 0 to 65535
signed int / **int**	at least -32767 to 32767
unsigned int	at least 0 to 65535
signed long int / **long**	at least -2147483647 to 2147483647
unsigned long int / **unsigned long**	at least 0 to 4294967295

- Range of values for **int** is machine specific: 2 bytes for 16-bit computers and 4 bytes for 32/64-bit computers.
- **long** is 4 bytes for 16-bit computers and 8 bytes for 32/64-bit computers.

- C/C++ support write-once/read-only constant variables with type qualifier **const** (Java **final**), in any variable declaration context.

Java	**C/C++**
final short x = 3, y; y = x + 7; **final char** c = 'x';	**const short int** x = 3, y = x + 7; **disallowed** **const char** c = 'x';

- C/C++ **const** identifier ***must*** be assigned a value at declaration (or by a constructor's declaration); the value can be the result of an expression:
- A constant variable can appear in read-only contexts after it is initialized.

2.3.5 Constants

- Java and C/C++ share almost all the same constants for the basic types (except for unsigned).
- A **designated constant** indicates its type with suffixes: L/l for long, LL/ll for long long, U/u for unsigned, and F/f for float.
- Unlike Java, there is no D/d suffix for **double** constants.

- The type of an **undesignated integral constant** (octal/decimal/hexadecimal) is the smallest **int** type that holds the value, and the type of a floating-point constant is **double**.

boolean	**false**, **true**
decimal	123, -456L, 789u, 21UL
octal, prefix 0	0144, -045l, 0223U, 067ULL
hexadecimal, prefix 0X or 0x	0xfe, -0X1fL, 0x11eU, 0xffUL
floating-point	.1, 1., -1., -7.3E3, -6.6e-2F, E/e exponent
character, single character	'a', '\''
string, multi-character	"abc", "\"\""

- Use the right constant with types character or string:

```
char ch = "a";      // use 'a'
char *str = 'a';    // use "a"
```

- An escape sequence allows special characters to appear in a character or string constant and starts with a backslash, \.

```
"\\ \" \t \n \012 \xf3"
```

- The most common escape sequences are (see a C++ textbook for others):

`'\\'`	backslash
`'\''`, `"\""`	single and double quote
`'\t'`, `'\n'`	tab, newline
`'\0'`	zero, string termination character
`'\ooo'`	octal value, `ooo` up to 3 octal digits
`'\xhh'`	hexadecimal value, `hh` up to 2 hexadecimal digits (not in Java)

- C/C++ string constant implicitly terminated with character `'\0'`.
- E.g., `"abc"` is 4 characters composed of `'a'`, `'b'`, `'c'`, `'\0'`.

2.3.6 Type Constructor

- A **type constructor** is a declaration that builds a more complex type from the basic types.

constructor	**Java**	**C/C++**
enumeration	**enum** Colour { R, G, B }	**enum** Colour { R, G, B }
pointer		*any-type* *p;
reference	*class-type* r;	*any-type* &r; (C++ only)
structure	**class**	**struct** or **class**
array	**int** v[ ] = **new int**[10]; **int** m[ ][ ] = **new int**[10][10];	**int** v[10]; **int** m[10][10];

- Java/C/C++ use **name equivalence** to decide if two types are the same:

```
class T1 {                          class T2 {    // identical structure
    int i, j, k;                        int i, j, k;
    double x, y, z;                     double x, y, z;
}                                   }
T1 t1;
T2 t2 = t1;        // incompatible types
```

- Types T1 and T2 have identical structure but have different names so the initialization of variable t2 fails, even though technically it could work.
- An **alias** is a different name for same type, so alias types are equivalent.

2.3.6.1 Enumeration

- An **enumeration** is a type defining a set of named constants with only assignment, comparison and implicit cast to integer operations:

```
enum Day {Mon,Tue,Wed,Thu,Fri,Sat,Sun}; // type declaration, implicit numbering
Day day = Sat;                          // variable declaration, initialization
enum {Yes, No} vote = Yes;              // anonymous type and variable declaration
enum Colour {R=0x1, G=0x2, B=0x4} colour; // type/variable declaration, explicit numberi
colour = B;                             // assignment
day = colour;                           // fails C++, works C
```

- Names in an enumeration are called **enumerators**.
- Enumerators can be numbered explicitly.
- Enumeration in C++ denotes a new type; enumeration in C is alias for **int**.
- C/C++ enumeration only has underlying type **int**; Java enumeration can give names (and operations) to any value.
- Java enumerator names must always be qualified.
- C/C++ enumerator names are unqualified ⇒ unique in a lexical scope.
- In C, **enum** must always be specified for a declaration:

```
enum Days day = Sat;   // repeat "enum" on variable declaration
```

2.3.6.2 Pointer/Reference

- **pointer/reference** is an indirect mechanism to access a type instance.
- ***All*** variables have an address in memory, e.g., **int** x = 5, y = 7:

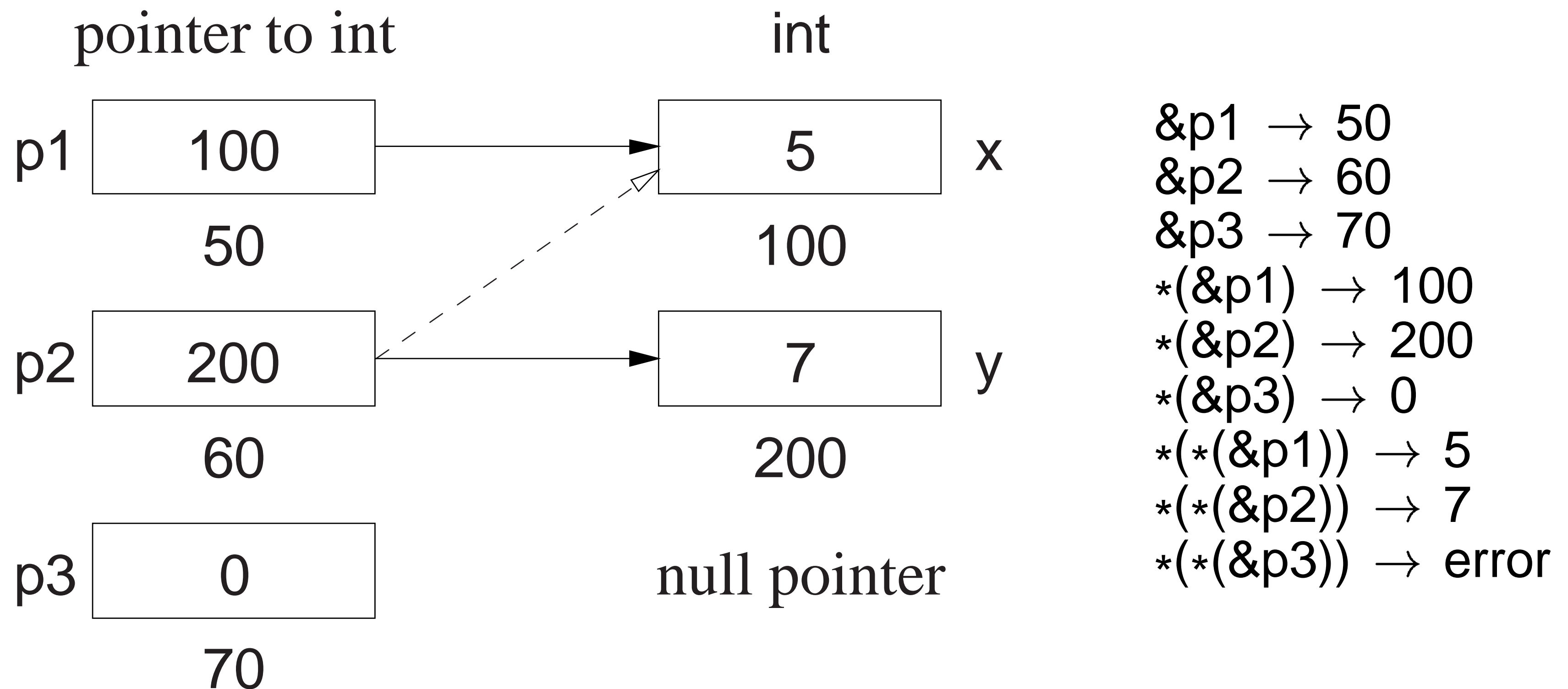
type		int		int
variable/value	x	5	y	7
address		100		200

- Value of a pointer/reference is the address of a variable.
- Accessing this address is different for a pointer or reference.
- Two basic pointer/reference operations:
  1. **referencing**: obtain address of a variable; unary operator & in C++:

     &x $\rightarrow$ 100
&y $\rightarrow$ 200
  2. **dereferencing**: retrieve value at an address; unary operator * in C++:

     *(&x) $\rightarrow$ *(100) $\rightarrow$ 5
*(&y) $\rightarrow$ *(200) $\rightarrow$ 7
- Compiler automatically does first dereference, so x is really *(&x).

- Unary and binary use of operators &/* for reference/deference and conjunction/multiplication.
- Special address no variable has (**null pointer**), null in Java, 0 in C++.
- Pointer/reference variable has: memory address of another variable (**indirection**), null pointer (or an undefined address if uninitialized).



- Because of implicit 1st dereference, p1 is 100 and *p1 is 5.
- A pointer/reference may point to the same memory address as another pointer/reference (dashed line).

- Dereferencing null pointer is an error as no variable at address 0.
- Explicit dereference is an operation usually associated with a pointer:

```
*p2 = *p1;        ≡   y = x;     // value assignment
*p1 = *p2 * 3;    ≡   x = y * 3;
```

- Address assignment does not require dereferencing:

```
p2 = p1;          // address assignment
```

- p2 is assigned the same memory address as p1, i.e., p2 points at x; values of x and y do not change.
- Having to perform explicit dereferencing can be tedious and error prone.

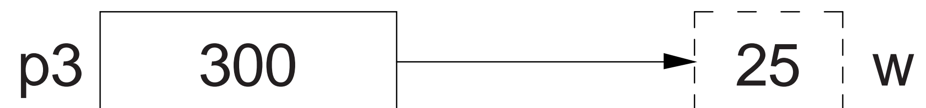
```
p1 = p2 * 3;      // implicit deference
```

  unreasonable as p1 is assigned address in p2 times 3.
- Reasonable if value pointed to by p1 is assigned value pointed to by p2 times 3.
- A pointer that provides implicit dereferencing is a **reference**.
- However, implicit dereferencing generates an ambiguous situation for:

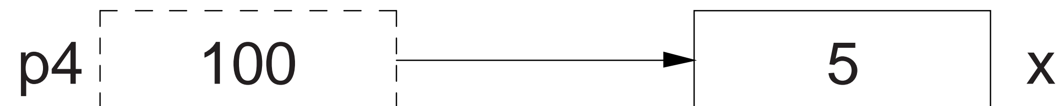
```
p2 = p1;
```

- Should this expression perform address or value assignment, and how are both cases specified?
- C provides only a pointer; C++ provides a pointer and a restricted reference; Java provides only a general reference.
- C/C++ pointer:
  - created using the * type-constructor,
  - may point to any type (i.e., basic or object type) in any storage location (i.e., global, stack or heap storage),
  - and no implicit referencing or dereferencing.
  - type qualifiers can be used to modify pointer types:

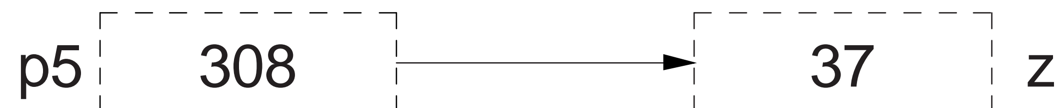
```
const short int w = 25;
const short int *p3 = &w;
```




```
int * const p4 = &x;
( int &p4 = x; )
```




```
const long int z = 37;
const long int * const p5 = &z;
```



- p3 may point at any **const short int** variable.
- Pointer can change to point at different variables, but the value of the variables cannot be changed through the pointer.
- p4 may only point at variable x.
- Pointer cannot change to point at a different variable, but the value of the variable can be changed through the pointer.
- p5 may only point at variable z.
- Pointer cannot change to point at a different variable, and the value of the variable z cannot be changed through the pointer.

- C++ reference
  - created using the & type-constructor,
  - may point to any type (i.e., basic or object type) in any storage location (i.e., global, stack or heap storage),
  - restricted to a constant pointer to user created (non-temporary/non-constant) storage,
  - and always has implicit dereferencing.
  - constant-pointer restriction of a C++ reference is equivalent to a Java **final** reference or * **const** pointer with implicit dereferencing.

- Java reference can vary what it points to, but it can only point to objects in heap storage.
- C++ constant-pointer restriction has two implications:
  1. A C++ reference must be initialized at the point of declaration.
     * initializing expression has implicit referencing because an address is ***always*** required;

```
int &r1 = &x; // error, unnecessary & before x
```

  2. No need for address assignment after a C++ reference declaration because the address cannot change.
     * Java interprets reference assignment r2 = r1 as address assignment and has no mechanism to perform value assignment between reference types.

- **Pointer/reference type-constructor is not distributed across the identifier list:**

`int * p1, p2;`	only p1 is a pointer, p2 is an integer, should be	`int *p1, *p2;`
`int & rx, ry;`	only rx is a reference, ry is an integer, should be	`int &rx, &ry;`

2.3.6.3 Aggregation (Structure/Array)

- Like Java, C++ is object-oriented, but it does not subscribe to the notion that everything is a basic type or an object.
- Instead, aggregation is performed by structures and arrays, and computation is performed by routines.
- An object type is the composition of a structure and routines.
- In C++, a routine can exist without being embedded in a **struct**/**class**.

Structure is a mechanism to group together heterogeneous values, including (nested) structures:

Java	C/C++

`class Foo {`
`    public int i = 3;`
`    . . . // more fields`
`}`
	`struct Foo {`
	`    int i; // no initialization`
	`    . . . // more members`
	`}; // semi-colon terminated`

- Components of a structure are called **members**[1] in C++.

[1]Java subdivides members into fields (data) and methods (routines).

- All members of a structure are accessible (public) by default (excluding Java package visibility).
- A structure member cannot be directly initialized (unlike Java) , and a structure is terminated with a semicolon.
- As for enumerations, a structure can be defined and instances declared in a single statement.

```
struct S { int i; } s; // definition and declaration
```

- In C, **struct** must always be specified for a declaration:

```
struct Complex a, b;    // repeat "struct" on variable declaration
```

- **Recursive types** (lists, trees) are defined using a pointer in a structure:

```
struct Node {
    . . .               // data members
    Node *link;         // pointer to another Node
};
```

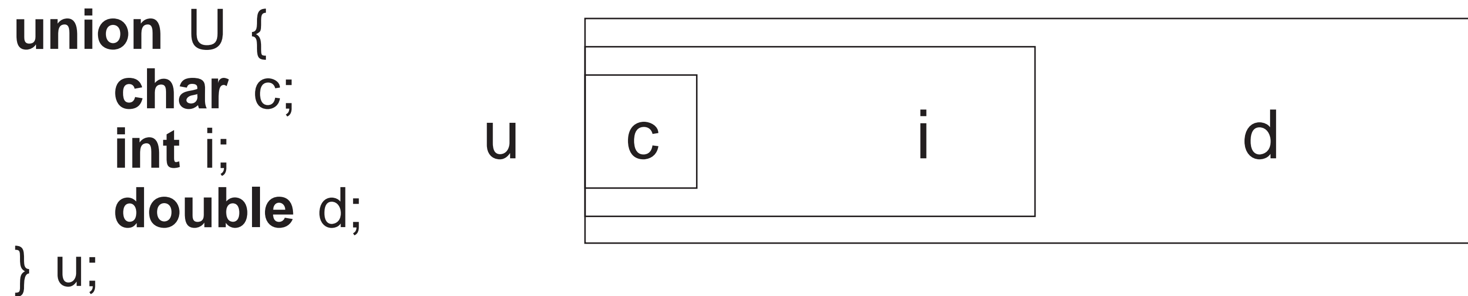
- A **bit field** allows direct access to individual bits of memory:

```
struct S {
    int i : 3;    // 3 bits
    int j : 7;    // 7 bits
    int k : 6;    // 6 bits
};
i = 2;  // 10
j = 5;  // 101
k = 9;  // 1001
```

- A bit field must be an integral type.
- Unfortunately, bit-fields are not portable.
- On little-endian architectures (e.g., like Intel/AMD x86), the compiler reverses the bit order.
- However, the compiler does not implicitly reverse the bit order.
- Hence, the bit-fields in variable s above must be reversed for little-endian architectures.
- While it is unfortunate C/C++ bit-fields lack portability, they are the highest-level mechanism to manipulate bit-specific information.

Union is a heterogeneous aggregation mechanism, where all members overlay the same storage:

```
union U {
    char c;
    int i;
    double d;
} u;
```



- Used to access internal representation or save storage by reusing it for different purposes at different times.

```
union U {
    float f;
    struct {
        unsigned int sign : 1;
        unsigned int exp : 8;
        unsigned int val : 23;
    } s;
    int i;
} u;
```

```
u.f = 3.5;        cout << hex << u.f << "\t" << u.i << endl;
u.i = 3;          cout << u.i << "\t" << u.f << endl;
u.f = 3.5e3;      cout << u.s.sign << "\t" << u.s.exp << "\t" << u.s.val
u.f = -3.5e-3;    cout << u.s.sign << "\t" << u.s.exp << "\t" << u.s.val
```

produces:

```
3.5 40600000
3   4.2039e-45
0   8a  5ac000
1   76  656042
```

- ***Reusing storage is dangerous and can usually be accomplished via other techniques.***

Array is a mechanism to group together homogeneous values.

- C/C++ array is simple because dimension information is not stored with an array object.
- No equivalent to Java's `length` member for arrays, ***no subscript checking***, and no array assignment.
- Array variables can have dimensions specified on a declaration and all the array elements are implicitly allocated:

```
int x[10];           // int x[ ] = new int[10]
int y[10][20];       // int y[ ][ ] = new int[10][20]
```

- Be careful not to write:

```
int b[10, 20];       // not int b[10][20]
```

- C++ only supports a compile-time dimension value; g++ allows a runtime expression.

```
int r, c;
cin >> r >> c;       // input dimensions
int array[r];        // dynamic dimension, g++ only
int matrix[r][c];    // dynamic dimension, g++ only
```

- Like Java, an array is subscripted starting at 0.

2.3.7 Type-Constructor Constant

enumeration	enumerators
pointer	0 or NULL indicates a null pointer
structure	**struct** { **double** r, i; } c = { 3.0, 2.1 };
array	**int** v[3] = { 1, 2, 3 };

- C/C++ use 0 to initialize pointers versus null in Java.
- System include-files define the preprocessor variable NULL as 0.
- Structure and array initialization can only occur as part of a declaration.

```
struct { int i; struct { double r, i; } s; } d = { 1, { 3.0, 2.1 } };  // nested structure
int m[2][3] = { {93, 67, 72}, {77, 81, 86} };  // multidimensional array
```

- Values in initialization list are placed into a variable starting at the beginning of the structure or array.
- Not all the members/elements must be initialized.
- A nested structure or multidimensional array is created using braces.
- String constants can be used as a shorthand array initializer value:

`char s[6] = "abcde";` rewritten as `char s[6] = { 'a', 'b', 'c', 'd', 'e', '\0' };`

- It is possible to leave out the first dimension, and its value is inferred from the number of constants in that dimension:

```
char s[] = "abcde";    // 1st dimension inferred as 6 (Why 6?)
int  v[] = { 0, 1, 2, 3, 4 } // 1st dimension inferred as 5
int  m[][3] = { {93, 67, 72}, {77, 81, 86} }; // 1st dimension inferred as 2
```

2.3.8 Type Aliasing

- C/C++ provides **typedef** to create a synonym for an existing type:

```
typedef short int shrint1;   // shrint1 => short int
typedef shrint1 shrint2;     // shrint2 => short int
typedef short int shrint3;   // shrint3 => short int
shrint1 s1;       // implicitly rewritten as: short int s1
shrint2 s2;       // implicitly rewritten as: short int s2
shrint3 s3;       // implicitly rewritten as: short int s3
```

- All combinations of assignments are allowed among s1, s2 and s3, because they have the same type name "**short int**".
- Java provides no mechanism to rename types.

2.4 Expression

	Java	C/C++	priority
unary	., (), [ ], call	., ->, (), [ ], call, **dynamic_cast**	high
	cast, +, -, !, ~ **new**	cast, +, -, !, ~, &, * **new**, **delete**, **sizeof**	
binary	*, /, %	*, /, %	
	+, -	+, -	
bit shift	<<, >>, >>>	<<, >>	
relational	<, <=, >, >=, instanceof	<, <=, >, >=	
equality	==, !=	==, !=	
bitwise	& and	&	
	^ exclusive-or	^	
	| or	|	
logical	&& short-circuit	&&	
	||	||	
conditional	?:	?:	
assignment	=, +=, -=, *=, /=, %= <<=, >>=, >>>=, &=, ^=, |=	=, +=, -=, *=, /=, %= <<=, >>=, &=, ^=, |=	
comma		,	low

- Like algebra, operators are prioritize and performed from high to low.
- Operators with same priority are done left to right, except for unary, ?, and assignment operators, which associate right to left.

```
int **a, **b, c, d, *w[10];
**a = **b > c ? ( *a = *b, d - 1 ) : (*w)[3] * 7 + 3;
```

- Order of evaluation of subexpressions and argument evaluation is unspecified (Java left to right).

```
( i + j ) * ( k + j );        // either + done first
( i = j ) + ( j = i );        // either = done first
g( i ) + f( k ) + h( j );     // g, f, or h called in any order
f( p++, p++, p++ );           // arguments evaluated in any order
```

- Referencing (address-of), &, and dereference, *, operators do not exist in Java because access to storage is restricted.
- Find address of any variable in any storage context, e.g., &x, &s.d, &v[5].
- Arrow operator, ->, is unique to C/C++ and is an anomaly among programming languages.
- Exists because the priority of selection operator "." is incorrectly higher than dereference operator "*", so *p.f executes as *(p.f) instead of (*p).f.

- -> operator performs a dereference and member selection in the correct order, i.e., p->f is implicitly rewritten as (*p).f.
- Assignment is an operator; useful for **cascade assignment** to initialize multiple variables of the same type:

```
a = b = c = 0; // cascade assignment
x = y = z + 4;
```

- **Other uses of assignment in an expression are discouraged!**; i.e., assignment only on left side.
- C/C++ allows any expression to appear as a statement:

```
3;    j + i;    ( i + j ) * ( k + j );    sin(x);
```

- Complex assignment operators, e.g., lhs += rhs, are implicitly rewritten:

```
temp = &(lhs);  *temp = *temp + rhs;
```

- Hence, the left-hand side, lhs, is evaluated only once:

```
v[ rand() % 5 ] += 1;    // only calls random once
v[ rand() % 5 ] = v[ rand() % 5 ] + 1;  // calls random twice
```

- Comma expression is a series of expressions separated by commas:

`a, f + g, k(3) / 2, m[ i ][ j ]` ← value returned

- Expressions evaluated left to right with the value of rightmost expression returned as result.
- Comma expression allows multiple expressions to be evaluated in a context where only a single expression is allowed.
- Dimension problem `m[10, 20]` actually means `m[20]` because `10, 20` is a comma expression not a dimension list.
- Subscripting problem `m[3, 4]` means `m[4]`, 4th row of matrix.
- Operators `++` / `--` are discouraged because subsumed by general `+=` / `-=`.

2.4.1 Conversion

- Conversion implicitly/explicitly transforms a value from one type to another.
- Two kinds of conversions:
  - **widening conversion**, no information is lost:

char	→	**short int**	→	**long int**	→	**double**
'\x7'		7		7		7.000000000000000

– **narrowing conversion**, information can be lost:

double	→	long int	→	short int	→	char
77777.77777777777		77777		12241		'\xd1'

- C/C++ support both implicit widening and narrowing conversions (Java only implicit widening).
- **Implicit narrowing conversions can cause problems:**

```
int i;   double r;
i = r = 3.5;          // r -> 3.5
r = i = 3.5;          // r -> 3.0 ???
```

- Better to perform narrowing conversions explicitly using **cast** operator.

```
int i;   double x, y;
i = (int) x;          // explicit narrowing conversion
i = (int) x / (int) y; // explicit narrowing conversions for integer division
i = (int)(x / y);     // alternative technique
```

- C/C++ supports casting among the basic types and user defined types.
- g++ has a cast extension allowing construction of structure and array constants in executable statements not just declarations:

```
void rtn( const int m[2][3] );
struct Complex { double r, i; } c;
rtn( (int [2][3]){ {93, 67, 72}, {77, 81, 86} } );        // g++ only
c = (Complex){ 2.1, 3.4 };                                // g++ only
```

- In both cases, a cast is used to indicate the meaning and structure of the constant.

2.5 Control Structures

	Java	C/C++
block	{ *intermixed decls/stmts* }	{ *intermixed decls/stmts* }
selection	**if** (*bool-expr1*) *stmt1* **else if** (*bool-expr2*) *stmt2* . . . **else** *stmtn*	**if** (*cond-expr1*) *stmt1* **else if** (*cond-expr2*) *stmt2* . . . **else** *stmtn*

	switch (*integral-expr*) {
	  case c1: *stmt1*; **break**;
	  . . .
	  case cn: *stmtn*; **break**;
	  default: *stmt0*;
	}
		switch (*integral-expr*) {
		  case c1: *stmt1*; **break**;
		  . . .
		  case cn: *stmtn*; **break**;
		  default: *stmt0*;
		}

looping	**while** (*bool-expr*) *stmt*	**while** (*cond-expr*) *stmt*
	do *stmt* **while** (*bool-expr*) ;	**do** *stmt* **while** (*cond-expr*) ;
	for (*init-expr*;*bool-expr*;*incr-expr*) *stmt*	**for** (*init-expr*;*cond-expr*;*incr-expr*) *stmt*
transfer	**break** *[label]*	**break**
	continue *[label]*	**continue**
		goto *label*
	return *[expr]*	**return** *[expr]*
label	*label* : *stmt*	*label* : *stmt*

2.5.1 Block

- **Block** is a series of statements bracketed by braces, {...}, which can be nested.
- Block serves two purposes: bracket several statements into a single statement and introduce local declarations.
- **When a statement is required, good practice is to always use a block to allow easy insertion and removal of statements to or from block.**
- Putting local declarations precisely where they are needed can help reduce declaration clutter at the beginning of an outer block.
- However, it can also make locating them more difficult.

2.5.2 Conditional

- C/C++ uses a **conditional expression** in control structures to cause conditional transfer (Java uses a boolean expression).
- A conditional expression is evaluated and implicitly tested for not equal to zero, i.e., *cond-expr* $\equiv$ *expr* != 0.
- Boolean expressions are converted to 0 for **false** and 1 for **true** before comparison to zero, e.g.:

```
if ( x > y )...    implicitly rewritten as    if ( (x > y) != 0 )...
```

- Hence, other expressions are allowed in a conditional (C/C++ idiom):

```
if ( x ) ...       implicitly rewritten as    if ( (x) != 0 )...
while ( x )...                               while ( (x) != 0 )...
```

- Watch for the common mistake in a conditional:

```
if ( x = y )...    implicitly rewritten as    if ( (x = y) != 0 )...
```

which assigns y to x and tests x != 0 (possible in Java for one type).

2.5.3 Selection

- C/C++ selection statements are **if** and **switch** (same as Java, except for boolean versus conditional expression).
- An **if** statement selectively executes one of two alternatives based on the result of a comparison, e.g.:

```
if ( x > y ) max = x;
else max = y;
```

- Java/C/C++ have the **dangling else** problem of associating an **else** clause with its matching **if** in nested **if** statements.

- E.g., reward WIDGET salesperson who sold more than $10,000 worth of WIDGETS and dock pay of those who sold less than $5,000.

Dangling Else	Fix Using Null Else	Fix Using Blocks

`if ( sales < 10000 )`
`    if ( sales < 5000 )`
`        income -= penalty;`
`else // incorrect match!!!`
`    income += bonus;`
	`if ( sales < 10000 )`
	`    if ( sales < 5000 )`
	`        income -= penalty;`
	`    else ;   // null statement`
	`else`
	`    income += bonus;`
		`if ( sales < 10000 ) {`
		`    if ( sales < 5000 ) {`
		`        income -= penalty;`
		`    }`
		`} else {`
		`    income += bonus;`
		`}`

- A **switch** statement selectively executes one of N alternatives based on matching an integral value with a series of case clauses, e.g.:

```
switch ( day ) {            // integral expression
  case MON: case TUE: case WED: case THU:  // case value list
    cout << "PROGRAM" << endl;
    break;                  // exit switch
  case FRI:
    wallet += pay;
    // fall through !!!!!
  case SAT:
    cout << "PARTY" << endl;
    wallet -= party;
    break;                  // exit switch
  case SUN:
    cout << "REST" << endl;
    break;                  // exit switch
  default:
    cerr << "ERROR" << endl;
    exit( -1 );             // terminate program
}
```

- Once a case clause is matched, its statements are executed, and control continues to the ***next*** statement.
- **break** statement is used at end of a case clause to exit **switch** statement.

- **It is a common error to forget the break.**
- If no case clause is matched and there is a **default** clause, its statements are executed, and control continues to the ***next*** statement.
- Otherwise, the **switch** statement does nothing.
- Only one label for each **case** clause but a list of **case** clauses is allowed.

2.5.4 Conditional Expression Evaluation

- **Conditional expression evaluation** performs partial evaluation of expressions.
- Control structures not operators because both operands may not be evaluated.

&&	only evaluates the right operand if the left operand is true
||	only evaluates the right operand if the left operand is false
?:	only evaluates one of two alternative parts of an expression

- && and || (**short-circuit**) are similar to logical & and | for bitwise operands, i.e., both produce a logical conjunctive or disjunctive result.
- However, conditional && and || evaluate operands lazily until a result is determined, short-circuiting the evaluation of other operands.

- Logical & and | evaluate operands eagerly, evaluating both operands.
- With boolean operands, corresponding operators are interchangeable.
- Conditional ?: evaluates one of two expressions, and returns the result of the evaluated expression.
- Acts like an **if** statement in an expression:

```
abs2 = ( a < 0 ? -a : a ) + 2    if ( a < 0 ) {
                                     abs2 = -a;
                                 } else {
                                     abs2 = a;
                                 }
                                 abs2 += 2;
```

2.5.5 Looping

- C/C++ looping statements are **while**, **do** and **for** (same as Java, except for boolean versus conditional expression).
- **while** statement executes its statement zero or more times.

- **Beware of accidental infinite loops.**

```
x = 0;
while (x < 5);   // extra semicolon!
    x = x + 1;
```

```
x = 0;
while (x < 5)   // missing block
    y = y + x;
    x = x + 1;
```

- **do** statement executes its statement one or more times.

```
do {
    . . .   // executed at least once
} while ( x < 5 );
```

- **for** statement is a specialized **while** statement for iterating with an index.

```
init-expr;
while ( cond-expr ) {
    stmt;
    incr-expr;
}
```

```
for ( init-expr; cond-expr; incr-expr ) {
    stmt;
}
```

- Many ways to use the **for** statement to construct iteration:

```
for ( i = 1; i <= 10; i += 1 ) {         // count up
    // loop 10 times
} // i has the value 11 on exit
```

```
for ( i = 10; 1 <= i; i -= 1 ) {             // count down
    // loop 10 times
} // i has the value 0 on exit
for ( p = l; p != NULL; p = p->link ) {  // pointer index
    // loop through list structure
} // p has the value NULL on exit
for ( i = 1, p = l; i <= 10 & p != NULL; i += 1, p = p->link ) {  // 2 indices
    // loop until 10th node or end of list encountered
}
```

- Comma expression is used to initialize and increment 2 indices in a context where normally only a single expression is allowed.
- Default **true** value inserted if no conditional is specified in **for** statement.

```
for ( ; ; )          // rewritten as: for ( ; true ; )
```

- Short-circuit expression evaluation is often used for a linear search of an array for a key, where the loop index indicates the position of the key in the array if the key is found, or the array size plus 1 if not found:

```
for ( i = 0; i < size && list[i] != key; i += 1 );    // no loop body
```

- Short-circuit && prevents evaluation of conditional second operand if the first operand is true to prevent subscript error when the key is not found.
- Logical & would be incorrect because it evaluates both operands.
- **continue**/**break** statements available in all iteration constructs to immediately advance to next loop iteration or terminate loop construct.

```
for ( i = 0; ; i += 1 ) {          // infinite loop, conditional is "true"
    if ( i == size ) break;        // exit if not found
    if ( list[i] == key ) break;   // exit if found
}
```

- C/C++ **goto** statement simulates Java labelled **break** and **continue**.

Java	C / C++

<pre>L1: {
    ... declarations ...
    L2: switch (...) {
        L3: for (...) {
            ... break L1; ... // exit block
            ... break L2; ... // exit switch
            ... break L3; ... // exit loop
        }
        ...
    }
    ...
}</pre>
	<pre>{
	    ... declarations ...
	    switch (...) {
	        for (...) {
	            ... goto L1; ...
	            ... goto L2; ...
	            ... goto L3; ...
	        } L1: ;
	        ...
	    } L2: ;
	    ...
	} L3: ;</pre>

- **Only use goto to simulate labelled break and continue.**

2.6 Preprocessor

- Preprocessor manipulates the text of the program ***before*** compilation.
- **Program you see is not what the compiler sees!**
- The three most commonly used preprocessor facilities are substitution,

file inclusion, and conditional inclusion.

2.6.1 Substitution

- **#define** statement declares a preprocessor variable, and its value is all the text after the name up to the end of line.

```
#define Integer int
#define begin {
#define end }
#define PI 3.14159
#define gets =
#define set
#define with =
Integer main() begin            // same as: int main() {
    Integer x gets 3, y;        // same as: int x = 3, y;
    x gets PI;                  // same as: x = 3.14159;
    set y with x;               // same as: y = x;
end                             // same as: }
```

- Preprocessor can transform the syntax of C/C++ program (**discouraged**).
- Predefined preprocessor-variables exist identifying hardware and software environment, e.g., mcpu is kind of CPU.

- Use **const** declarations (**final** in Java) rather that **#define**:

```
const double PI = 3.14159;
const int arraySize = 100;
```

- **#define** can declare macros with parameters, which expand during compilation, textually substituting arguments for parameters, e.g.:

```
#define MAX( a, b ) ((a > b) ? a : b)
z = MAX( x, y );     // implicitly rewritten as: z = ((x > y) ? x : y)
```

- Use **inline** routines in C/C++ rather that **#define** macros.

2.6.2 File Inclusion

- File inclusion copies text from a file into a C/C++ program.
- An included file may contain anything.
- An include file normally imports preprocessor and C/C++ templates/declarations for use in a program.
- All included text goes through every compilation step, i.e., preprocessor, compiler, etc.

- Java does implicit inclusion by matching class names with file names in CLASSPATH directories, then extracting and including necessary declarations.
- The **#include** statement specifies the file to be included.
- C convention uses suffix ".h" for include files containing C declarations.
- C++ convention drops suffix ".h" for its standard libraries and has special file names for equivalent C files, e.g., cstdio versus stdio.h.

```
#include <stdio.h>      // C style
#include <cstdio>       // C++ style
#include "user.h"
```

- A file name can be enclosed in <> or `""`.
- <> means preprocessor only looks in the system include directories.
- `""` means preprocessor starts looking for the file in the same directory as the file being compiled, then in the system include directories.
- System files limits.h and unistd.h contains many useful **#define**s, like the null pointer constant NULL (e.g., see /usr/include/limits.h).

2.6.3 Conditional Inclusion

- Preprocessor has an **if** statement, which may be nested, to conditionally add/remove code from a program.
- Conditional of **if** uses the same relational and logical operators as C/C++, but operands can only be integer or character values.

```
#define DEBUG 0        // declare and initialize preprocessor variable
. . .
#if DEBUG == 1         // level 1 debugging
#   include "debug1.h"
. . .
#elif DEBUG == 2       // level 2 debugging
#   include "debug2.h"
. . .
#else                  // non-debugging code
. . .
#endif
```

- By changing value of preprocessor variable DEBUG, different parts of the program are included for compilation.
- To exclude code (comment-out), use 0 conditional as 0 implies false.

```
#if 0
. . .                            // code commented out
#endif
```

Independent of language structure, can overlap definitions and routines.

- It is also possible to check if a preprocessor variable is defined or not defined by using **#ifdef** or **#ifndef**:

```
#ifndef __MYDEFS_H__ // if not defined
#define __MYDEFS_H__ 1 // make it so
. . .
#endif
```

- Used in an **#include** file to ensure its contents are only expanded once.
- Note difference between checking if a preprocessor variable is defined and checking the value of the variable.
- The former capability does not exist in most programming languages, i.e., checking if a variable is declared before trying to use it.

2.7 Input/Output

- Input/Output (I/O) is divided into two kinds:

1. **Formatted I/O** transfers data with implicit conversion of internal values to/from human-readable form.
   – Conversion is based on the type of variables and format codes.
2. **Unformatted I/O** transfers data without conversion, e.g., internal integer and floating-point values.

2.7.1 Formatted I/O

Java	C	C++
File, Scanner, PrintStream	FILE	ifstream, ofstream
Scanner in = **new** Scanner(**new** File(`"f"`))	in = fopen(`"f"`, `"r"`);	ifstream in(`"f"`);
PrintStream out = **new** PrintStream(`"g"`)	out = fopen(`"g"`, `"w"`)	ofstream out(`"g"`)
in.close()	close(in)	scope ends
out.close()	close(out)	scope ends
nextInt()	fscanf(in, `"%d"`, &i)	in >> T
nextFloat()	fscanf(in, `"%f"`, &f)	
nextByte()	fscanf(in, `"%c"`, &c)	
next()	fscanf(in, `"%s"`, &s)	
hasNext()	feof(in)	in.eof()
hasNextT()	fscanf return value	in.fail()
		in.clear()
skip(`"regexp"`)	fscanf(in, `"%*[regexp]"`)	in.ignore(n, c)
out.print(String)	fprintf(out, `"%d"`, i)	out << T
	fprintf(out, `"%f"`, f)	
	fprintf(out, `"%c"`, c)	
	fprintf(out, `"%s"`, s)	

- Formatted I/O occurs to/from a **stream file**.
- C++ has three implicit stream files: cin, cout and cerr, which are automatically declared and opened (Java has in, out and err).
- C has stdin, stdout and stderr, which are automatically declared and opened.
- Include iostream has all necessary declarations for cin, cout and cerr.
- cin reads input from the keyboard (unless redirected by shell).
- cout writes to the terminal screen (unless redirected by shell).
- cerr writes to the terminal screen even when cout output is redirected.
- ***Error and debugging messages should always be written to* cerr**:
  - normally not redirected by the shell,
  - unbuffered so output appears immediately.
- Stream files other than 3 implicit ones require declaring each file object:

```
#include <fstream> // required for stream-file declarations
ifstream infile( "myinfile" );      // input file
ofstream outfile( "myoutfile" );   // output file
```

- Type of the file, ifstream or ofstream, indicates whether the file can be read or written.
- Declaration **opens** a file making it accessible through the variable name, e.g., infile and outfile are used for file access.
- Check for successful opening of a file using the stream member fail, e.g., infile.fail(), which returns **true** if the open failed and **false** otherwise.
- Connection between the file name in the program and operating-system file is done at the declaration:
  - infile reads from file `myinfile`
  - outfile writes to file `myoutfile`

  where both files are located in the directory where the program is run.
- C++ I/O library overloads the bit-shift operators << and >> to perform I/O.
- C I/O library uses fscanf(outfile,. . .) and fprintf(infile,. . .), which have short forms scanf(. . .) and printf(. . .) for stdin and stdout.
- Parameters in C are always passed by value, so arguments to fscanf must be preceded with & (except arrays) so they can be changed.

- Both I/O libraries can cascade multiple I/O operations, i.e., input or output multiple values in a single expression.

2.7.1.1 Input

- Java formatted input uses an ***explicit*** Scanner attached to an input file to convert characters to basic types.
- C/C++ formatted input has ***implicit*** character conversion for all basic types and is extensible to user-defined types.

Java	C	C++

import java.io.*;
import java.util.Scanner;
Scanner in =
    new Scanner(**new** File("f"));
PrintStream out =
    new PrintStream("g");
int i, j;
while (in.hasNext()) {
   i = in.nextInt(); j = in.nextInt();
   out.println("i:"+i+" j:"+j);
}
in.close();
out.close();
	#include <stdio.h>
	FILE *in = fopen("f", "r");
	
	FILE *out = fopen("g", "w");
	
	int i, j;
	for (;;) {
	   fscanf(in, "%d%d", &i, &j);
	 if (feof(in)) **break**;
	   fprintf(out,"i:%d j:%d\n",i,j);
	}
	close(in);
	close(out);
		#include <fstream>
		ifstream in("f");
		
		ofstream out("g");
		
		int i, j;
		for (;;) {
		   in >> i >> j;
		 if (in.eof()) **break**;
		   out << "i:" << i
		       <<"j:"<<j<<endl;
		}
		// in/out closed implicitly

- Input values for a stream file are C/C++ constants: 3, 3.5e-1, etc., separated by whitespace.
- Except for characters and character strings, ***which are not in quotes***, so cannot read strings containing white spaces.
- Type of operand indicates the kind of constant expected in the stream file, e.g., an integer operand means an integer constant is expected.
- Stream cin starts reading where the last cin left off.

- After all input values on current line are read, cin proceeds to next line.
- Hence, the placement of input values on lines of a file is often arbitrary.
- Unlike Java, C/C++ must attempt to read ***before*** end-of-file is set and can be tested for.
- End of file can be detected in two ways:
- cin and fscanf return 0 and EOF when eof is reached.
- C++ member eof and the C routine feof return true when eof is reached.
- **end-of-file** is the detection of the physical end of a file; there is no end-of-file character.
- From a keyboard, <ctrl>-d (press the <ctrl> and d keys simultaneously) causes the shell to close the current input file marking its physical end.
- When bad data is read, stream must be reset and bad data cleared:

```
#include <iostream>
using namespace std;
int main() {
    int n;
    for ( ;; ) {
        cout << "Enter a number: ";
        cin >> n;
      if ( cin.eof() ) break;         // eof ?
        if ( ! cin.fail() ) {         // number ?
            cout << "n = " << n << endl;
        } else {
            cout << "Not a number. ";
            cin.clear();              // reset stream failure
            cin.ignore( numeric_limits<int>::max(), '\n' );  // skip until new
        }
    }
    cout << endl;
}
```

- After an unsuccessful read, clear() resets the stream.
- ignore skips either n characters, e.g., cin.ignore(5) or until a specified character.

2.7.1.2 Output

- Java output style converts values to strings, concatenates strings, and prints final long string:

```
System.out.println( i + " " + j );    // build a string and print it
```

- C/C++ output style supplies a list of formats and values, and output operation generates the strings:

```
cout << i << " " << j << endl;    // print each string when formed
```

- There is no implicit conversion from the basic types to string in C++ (but one can be constructed).
- **While it is possible to use the Java string-concatenation style in C++, it is incorrect style.**
- Input/output format is controlled via **manipulators** defined in include file iomanip:

```
#include <iostream>     // cin, cout, cerr
#include <iomanip>      // manipulators
using namespace std;
int i = 7; double r = 2.5; char c = 'z'; char *s = "abc";
cout << "i:" << setw(2) << i
     << " r:" << fixed << setw(7) << setprecision(2) << r
     << " c:" << c << " s:" << s << endl;
#include <stdio.h>
fprintf( stdout, "i:%2d r:%7.2f c:%c s:%s\n", i, r, c, s );
" i: 7 r:   2.50 c:z s:abc"
```

oct	values in octal
dec	values in decimal
hex	values in hexadecimal
left / right (default)	values with padding after / before values
boolapha / noboolapha (default)	bool values as false/true instead of 0/1
showbase / noshowbase (default)	values with / without prefix 0 for octal & 0x for hex
fixed (default) / scientific	float-point values without / with exponent
setprecision(N)	fraction of float-point values in maximum of N columns
setw(N)	NEXT VALUE ONLY in minimum of N columns
setfill('ch')	padding character before/after value (default blank)
endl	flush output buffer and start new line (**output only**)
skipws (default) / noskipws	skip whitespace characters (**input only**)

- manipulators applies to all constants/variables after it, even to the next I/O expression for a specific stream file.
- **Except manipulator setw, which only applies to the next value in the I/O expression.**
- endl is not the same as `'\n'`; only endl flushes for interactive output.

2.7.2 Unformatted I/O

- **Unformatted I/O** transfers data without conversion, e.g., internal integer and floating-point values.
- Uses same mechanisms as formatted I/O to connect program to file (open/close).
- Uses read and write routines to transfer bytes without conversion from/to a file.

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

int main() {
    ofstream outfile( "xxx" );          // open output file "xxx"
    if ( outfile.fail() ) {             // successful open ?
        cerr << "Error!" << endl;
        exit( -1 );
    } // if

    double d = 3.0;
    outfile.write( (char *)&d, sizeof( d ) );  // coercion
    outfile.close();                    // close file before attempting read
```

```
    ifstream infile( "xxx" );          // open input file "xxx"
    if ( infile.fail() ) {             // successful open ?
        cerr << "Error!" << endl;
        exit( -1 );
    } // if

    double e;
    infile.read( (char *)&e, sizeof( d ) );   // coercion
    cout << e << endl;
    infile.close();
}
```

- read and write take a **char** * pointer and length.

```
read( char *buffer, streamsize num );
write( char *buffer, streamsize num );
```

- To pass any kind of pointer for unformatted I/O requires a **coercion**, which is a cast ***without*** a conversion.
- ***Coercion breaks the type system; use it very sparingly*** (and would be unnecessary if buffer type was **void** *).

2.8 Dynamic Storage Management

- C++ operator **new** takes a type operand and return a pointer to new storage of that type allocated in an area called the **heap**.
- Unlike Java, C/C++ allow ***all*** types to be dynamically allocated not just object types, e.g., **new int**.
- C/C++ do not have **garbage collection** of dynamically allocated storage after a variable is no longer accessible.
- Therefore, an additional dynamic storage-management operation to free storage.
- C++ provides dynamic storage-management operations **new**/**delete** and C provides malloc/free.
- ***Do not mix the two forms in a C++ program.***

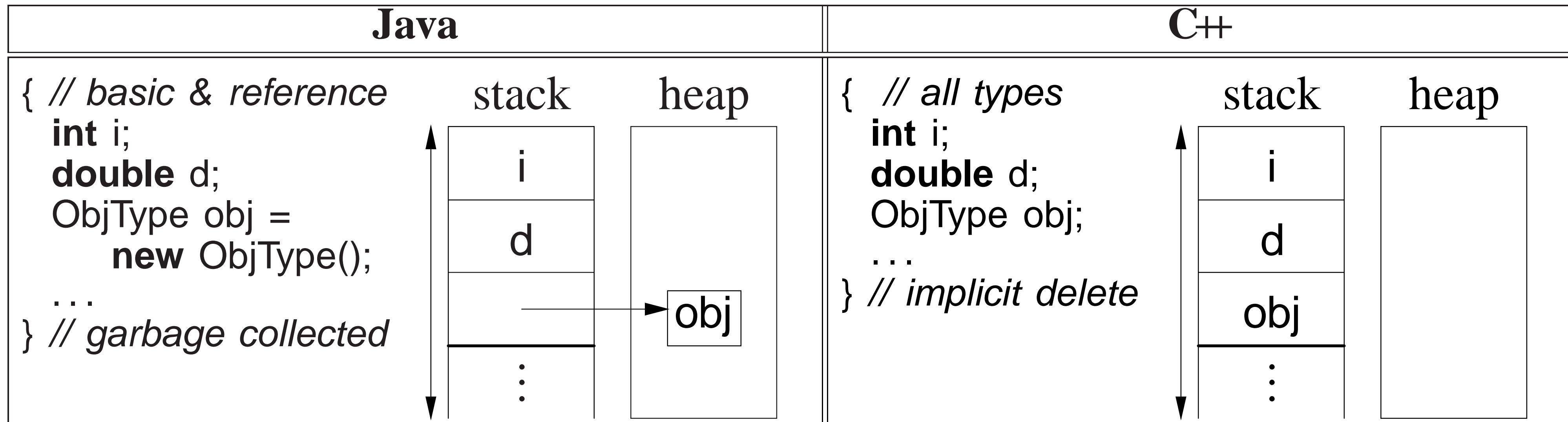
Java	C/C++

<pre>class Foo {
    char a, b, c;
}
class Test {
    public static void main(String[] args) {
        Foo f = new Foo();
        f.c = 'R';
    }
}</pre>
	<pre>struct Foo {
	    char a, b, c;
	};
	
	int main() {
	    Foo *f = new Foo(); // opt parenthesis
	    f->c = 'R';
	    delete f; // explicit free
	}</pre>

- Parenthesis after the type name in the **new** operation are optional.
- After storage is no longer needed it ***must*** be explicitly deleted.
- After storage is deleted, it ***must*** not be used:

```
delete f;
f->c = 'S';     // result of dereference is undefined
```

- Aggregate types can be allocated on the stack, i.e., local variables of a block:



- **Stack allocation is more efficient than heap allocation and does not require explicit storage management — use it whenever possible.**
- Dynamic allocation in C++ should be used only when:
  - a variable's storage must outlive the block in which it is allocated:

```
ObjType *rtn(. . .) {
    ObjType *obj = new ObjType();
    . . . // use obj
    return obj; // storage outlives block
} // obj deleted later
```

  - when each element of an array of objects needs initialization:

```
ObjType *v[10]; // array of object pointers
for ( int i = 0; i < 10; i += 1 ) {
    v[i] = new ObjType( i ); // each element has different initialization
}
```

- Declaration of a pointer to an array is complex in C/C++.
- Because no array-size information, the dimension value for an array pointer is often unspecified:

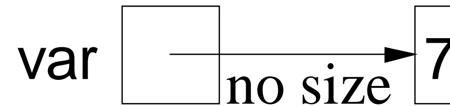
```
int *arr = new int[10];  // think arr[ ], pointer to array of 10 ints
```

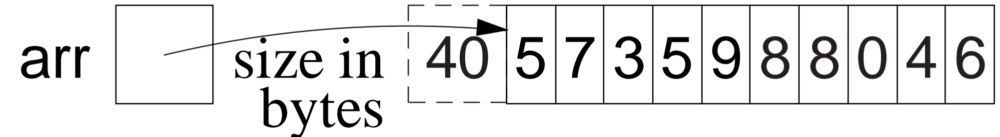
- Java notation:

```
int arr[ ] = new int[10];
```

  cannot be used because **int** arr[ ] is actually rewritten as **int** arr[N], where N is the size of the initializer value.
- Note, the lack of dimension information for an array means there is no subscript checking.
- As well, no dimension information results in the following ambiguity:

```
int *var = new int;
int *arr = new int[10];  // arr[ ]
```





- Variables var and arr have the same type but one is an array, which poses a problem when deleting a dynamically allocated array.
- To solve the problem, special syntax is used to distinguish these cases:

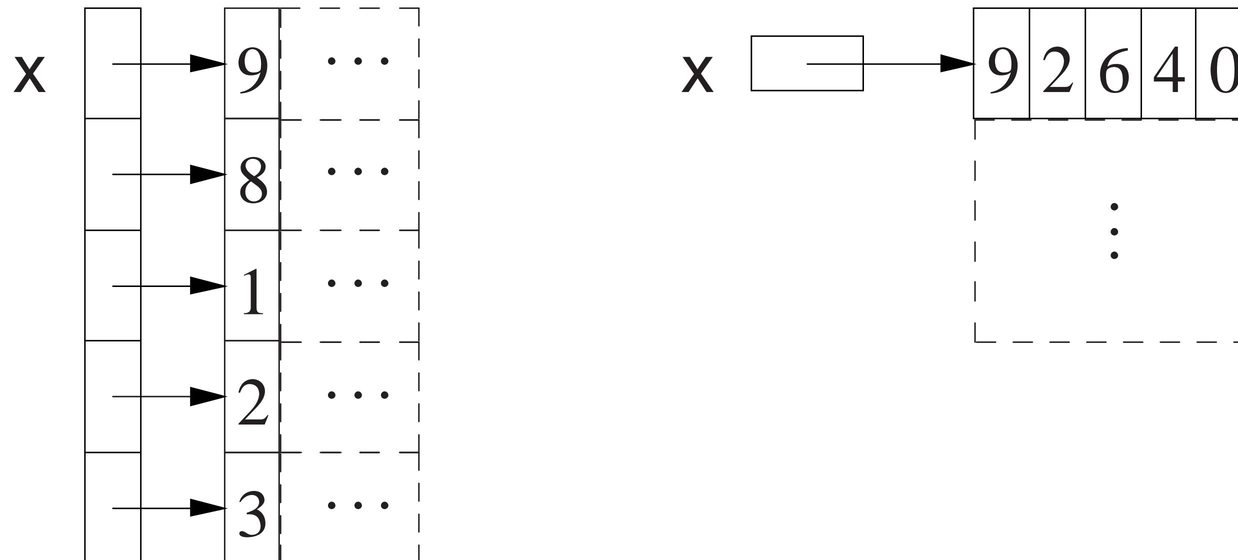
```
delete var;       // single element
delete [ ] arr;   // multiple elements
```

- [ ] indicates multiple elements (but unknown number and size of dimensions) and array-size is stored with the array.
- **Never do this:**

```
delete [ ] arr, var; // => (delete [ ] arr), var;
```

which is an incorrect use of a comma expression; var is not deleted.

- Declaration of a pointer to a matrix is complex in C/C++, e.g., **int** *x[5] could mean:



- Left: array of 5 pointers to an array of unknown number of integers.
- Right: pointer to matrix of unknown number of rows with 5 columns of integers.
- For * and [ ] which applied first?
- Dimension is higher priority (as subscript, see Section 2.4, p. 32), so declaration is interpreted as **int** (*(x[5])) (left).
- To read a declaration, parenthesize type qualifiers, read inside parenthesis outwards, start with variable name and end with type name on left.

- Only the left example (above) of declaring a matrix can be generalized to allow a dynamically-sized matrix.

```
int main() {
    int *m[5];                                  // 5 rows
    for ( int r = 0; r < 5; r += 1 ) {
        m[r] = new int[4];                      // 4 columns per row
        for ( int c = 0; c < 4; c += 1 ) { // initialize matrix
            m[r][c] = r + c;
        }
    }
    for ( int r = 0; r < 5; r += 1 ) {      // print matrix
        for ( int c = 0; c < 4; c += 1 ) {
            cout << m[r][c] << ", ";
        }
        cout << endl;
    }
    for ( int r = 0; r < 5; r += 1 ) {
        delete [] m[r];                         // delete each row
    }
}                                               // implicitly delete array "m"
```

2.9 Routine

C	C++

<pre>void p(or   T f(// parameters
      T1 a   // pass by value
 	
)
{   // routine body
      // intermixed decls/stmts
}</pre>
	<pre>void p(or T f(// parameters
	      T1 a, // pass by value
	      T2 &b, // pass by reference
	      T3 c = 3 // optional, default value
	      )
	{ // routine body
	      // intermixed decls/stmts
	}</pre>

- C++ routines are not part of aggregation (not combined in an object), e.g., routine main is not defined in a type.
- A routine is either a **procedure** or a **function** based on the return type.
- A procedure does return a value, indicated with return type of **void**:

```
void r( . . . ) { . . . }
```

- A routine with no parameters has parameter **void** in C and empty parameter list in C++:

```
... r( void ) { ... }        // C: no parameters
... r() { ... }              // C++: no parameters
```

- Routines cannot be nested in other routines, so all routine names are at the same scope level in a source file.
- Routine scope is between the global scope of the source file and a routine body:

```
int i = 1;              // global scope
int main() {
    int i = 2;          // local scope, hides previous variable i
}
```

- A procedure terminates when control runs off the end of routine body or a **return** statement is executed:

```
void proc() {
    ... return; ...
    ... // run off end
}
```

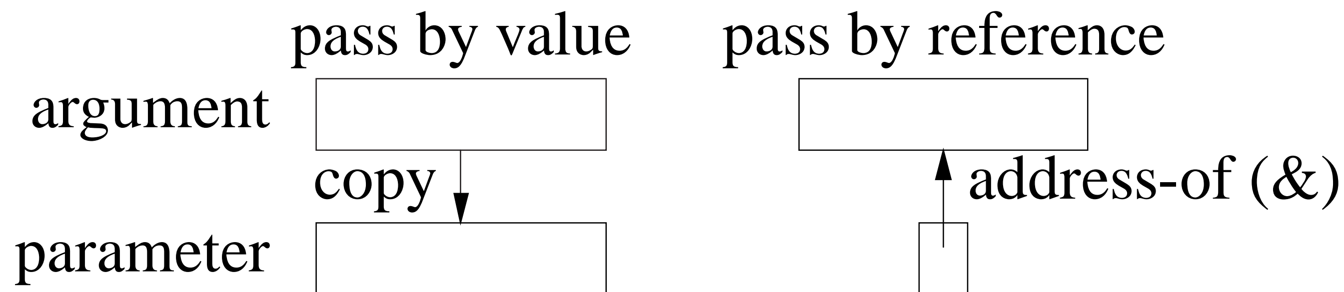
- A function ***must*** execute a **return** statement specifying a value:

```
int func() {
    . . . return 3; . . .
    return a + b;
}
```

- A **return** statement can appear anywhere in a routine body, and multiple return statements are possible.

2.9.1 Argument/Parameter Passing

- Arguments are passed to parameters by:
  - **value**: parameter is initialized by the argument (usually bit-wise copy).
  - **reference**: parameter is a reference to the argument and is initialized to the argument's address.



- Java/C, parameter passing is by value, i.e., basic types and object references are copied.
- C++, parameter passing is by value or reference depending on the type of the parameter.
- Argument expressions are evaluated ***in any order***.
- For value parameters, each argument-expression result is pushed on the stack to become the corresponding parameter, ***which may involve an implicit conversion***.
- For reference parameters, each argument-expression result is referenced (address of) and this address is pushed on the stack to become the corresponding reference parameter.

```
#include <iostream>
using namespace std;
struct Complex { double r, i; };
void r( int i, int &ri, Complex c, Complex &rc ) {
    ri = i = 3;
    rc = c = (Complex){ 3.0, 3.0 };
}
int main() {
    int i1 = 1, i2 = 2;
    Complex c1 = { 1.0, 1.0 }, c2 = { 2.0, 2.0 };
    r( i1, i2, c1, c2 );
}
```

- Which arguments change?
- What if routine call is changed to `r( i1, i1+i2, c1, c2 )`.
- Value passing is most efficient for basic and small structures because the values are accessed directly in the routine.
- Reference passing is most efficient for large structures and arrays because the values are not duplicated in the routine.
- Use type qualifiers to create read-only reference parameters so the corresponding argument is guaranteed not to change:

```
void r( const int &i, const Complex &c, const int v[5] ) {
    i = 3;          // assignments disallowed, read only!
    c.r = 3.0;
    v[0] = 3;
}
r( i + j, (Complex){ 1.0, 7.0 }, (int [5]){ 3, 2, 7, 9, 0 } );
```

- Provides efficiency of pass by reference for large variables, security of pass by value because argument cannot change, and allows temporary variables and constants as arguments.
- C++ parameter can have a **default value**, which is passed as the argument value if no argument is specified at the call site.

```
void r( int i, double g, char c = '*', double h = 3.5 ) { . . . }
r( 1, 2.0, 'b', 9.3 );          // maximum arguments
r( 1, 2.0, 'b' );               // h defaults to 3.5
r( 1, 2.0 );                    // c defaults to '*', h defaults to 3.5
```

- In a parameter list, once a parameter has a default value, all parameters to the right must have default values.
- In a call, once an argument is omitted for a parameter with a default value, no more arguments can be specified to the right of it.

2.9.2 Array Parameter

- Array copy is unsupported so arrays cannot be passed by value only by reference.
- Therefore, all array parameters are implicitly reference parameters, and hence, do not have a reference symbol.
- A formal parameter array declaration can specify the first dimension with a dimension value, [10] (which is ignored), an empty dimension list, [ ], or a pointer, *:

```
double sum( double v[5] );    double sum( double v[] );    double sum( double *v );
double sum( double *m[5] );   double sum( double *m[] );   double sum( double **m );
```

- Good programming practice uses the middle form because it clearly indicates the variable is going to be subscripted.
- An actual declaration cannot use [ ]; it must use *:

```
double sum( double v[] ) { // formal declaration
    double *cv;            // actual declaration, think cv[]
    cv = v;                // address assignment
```

- Routine to add up the elements of an arbitrary-sized array or matrix:

```
double sum( int cols, double v[] ) {
    int total = 0.0;
    for ( int c = 0; c < cols; c += 1 )
        total += v[c];
    return total;
}
```

```
double sum( int rows, int cols, double *m[] ) {
    int total = 0.0;
    for ( int r = 0; r < rows; r += 1 )
        for ( int c = 0; c < cols; c += 1 )
            total += m[r][c];
    return total;
}
```

2.9.3 Routine Pointer

- The flexibility and expressiveness of a routine comes from the argument/parameter mechanism, which generalizes a routine across any argument variables of matching type.
- However, the code within the routine is the same for all data in these variables.
- To generalize a routine further, it is necessary to pass code as an argument, which is executed within the routine body.
- Most programming languages allow a routine pointer (Java does not) for further generalization and reuse.
- As for data parameters, routine pointers are specified with a type (return

type, and number and types of parameters), and any routine matching this type can be passed as an argument, e.g.:

```
int f( int v, int (*p)( int ) ) { return p( v * 2 ) + 2; }
int g( int i ) { return i - 1; }
int h( int i ) { return i / 2; }
cout << f( 4, g ) << endl;   // pass routines g and h as arguments
cout << f( 4, h ) << endl;
```

- Routine f is generalized to accept any routine argument of the form: returns an **int** and takes an **int** parameter.
- Within the body of f, the parameter p is called with an appropriate **int** argument, and the result of calling p is further modified before it is returned.
- A routine pointer is passed as a constant reference in virtually all programming languages; in general, it makes no sense to change or copy routine code, like copying a data value.
- C/C++ require the programmer to explicitly specify the reference via a pointer, while other languages implicitly create a reference.
- Two common uses of routine parameters are fix-up and call-back

routines.

- A **fix-up routine** is passed to another routine and called if an unusual situation is encountered during a computation.
- E.g., when inverting a matrix, the matrix may not be invertible if its determinant is 0 (singular).
- Rather than halt the program for a singular matrix, invert routine calls a user supplied fix-up routine to possible recover and continue with a correction (e.g., modify the matrix):

```
int singularDefault( ... ) { return 0; }
int invert( int matrix[ ][10], int rows, int cols,
                int (*singular)( ... ) = singularDefault ) {
    ...
    if ( determinant( matrix, rows, cols ) == 0 ) {
        // compute correction to continue the computation
        correction = singular( matrix, rows, cols );
    }
    ...
}
```

- A fix-up parameter generalizes a routine as the corrective action is specified for each call, and the action can be tailored to a particular

usage.

- Giving fix-up parameter a default value, eliminates having to provide a fix-up argument.
- A **call-back routine** is used in event programming.
- When an event occurs, one or more call-back routines are called (triggered) and each one performs an action specific for that event.
- E.g., a graphical user interface has an assortment of interactive "widgets", such as buttons, sliders and scrollbars.
- When a user manipulates the widget, events are generated representing the new state of the widget, e.g., button down or up.
- A program registers interest in transitions for different widgets by supplying a call-back routine, and each widget calls its supplied call-back routine(s) when the widget changes state.
- Normally, a widget passes the new state of the widget to each call-back routine so it can perform an appropriate action, e.g.:

```
int callback( /* information about event */ ) {
    // examine event information and perform appropriate action
    // return status of callback action
}
. . .
registerCB( closeButton, callback );
```

- Call-back programming become difficult if it depending on the number of times it is called or previous argument values.

2.10 String

- Strings are supported in C by language and library facilities.
- Language facility ensures all string constants are terminated with a character value `'\0'`.
- E.g., the string constant `"abc"` is actually an array of the 4 characters: `'a'`, `'b'`, `'c'`, and `'\0'`, which occupies 4 bytes of storage.
- Zero value is a **sentinel** used by C string routines to locate the string end.
- Drawbacks:
  1. A string cannot contain a character with the value `'\0'`.

2. String operations needing the length of a string must linearly search for `'\0'`, which is expensive for long strings.
3. Management of variable-sized strings is the programmer's responsibility, with complex storage management problems.

- C++ solves these Drawbacks by providing a string type using a length member and managing all of the storage for the variable-sized strings.
- Unlike Java, instances of the C++ string type are not constant.
- Values can change so a companion type like StringBuffer in Java is unnecessary.
- ***Note, it is seldom necessary to iterate through the characters of a string variable!***

Java String methods	**C char** [] routines	**C++** string members
	strcpy, strncpy	=
+, concat	strcat, strncat	+
compareTo	strcmp, strncmp	==, !=, <, <=, >, >=
length	strlen	length
charAt	[]	[]
substring		substr
replace		replace
indexOf, lastIndexOf	strstr	find, rfind
	strcspn	find_first_of, find_last_of
	strspn	find_first_not_of, find_last_not_of

- All of the C++ string find members return string::npos if a search is unsuccessful.

```
string a, b, c;        // declare string variables
cin >> c;              // read white-space delimited sequence of characters
getline( cin, c, '\n' ); // read remaining characters until newline (newline is c
cout << c << endl;     // print string
a = "abc";             // set value, a is "abc"
b = a;                 // copy value, b is "abc"
c = a + b;             // concatenate strings, c is "abcabc"
if ( a == b )          // compare strings, lexigraphical ordering
string::size_type l = c.length(); // string length, l is 6
char ch = c[4];        // subscript, ch is 'b', zero origin
c[4] = 'x';            // subscript, c is "abcaxc", must be character constant
string d = c.substr( 2, 3 ); // extract starting at position 2 (zero origin) for len
c.replace( 2, 1, d);   // replace starting at position 2 for length 1 and insert d,
string::size_type p = c.find( "ax" ); // search for 1st occurrence of string "ax
p = c.rfind( "ax" );   // search for last occurrence of string "ax", p is 5
p = c.find_first_of( "aeiou" ); // search for first vowel, p is 0
p = c.find_first_not_of( "aeiou" ); // search for first consonant (not vowel),
p = c.find_last_of( "aeiou" ); // search for last vowel, p is 5
p = c.find_last_not_of( "aeiou" ); // search for last consonant (not vowel),
```

2.11 Shell Argument

- Routine main can be written without parameters, but it actually has two parameters, passed arguments from the invoking shell.

```
int main( int argc, char *argv[] )
```

- Shell takes the command line tokens and transforms them into C/C++ arguments.
- argc is the number of tokens in the shell command, including the name of executable file.
- Because the executable file-name is included, ***number of tokens is one greater than in Java***.
- argv is an array of pointers to the character strings that make up token arguments.

```
% ./a.out -option infile.C outfile.C

argc    = 4
argv[0] = "./a.out\0"      // not included in Java
argv[1] = "-option\0"
argv[2] = "infile.C\0"
argv[3] = "outfile.C\0"
argv[4] = 0                // mark end of variable length list
```

- Note, call of main by the shell is different because string tokens are passed not C/C++ values of or references to variables.
- A shell argument of `"32"` does not mean integer 32, and may have to converted.
- Routine main usually begins by checking argc for shell arguments.

<table>
<tr><th>Java</th><th>C/C++</th></tr>
<tr><td><pre>
class Prog {
    public static void main(String[] args) {
        switch (args.length) {
          case 0: . . .   // no args
            break;
          case 1: . . . args[0] . . . // 1 arg
            break;
          case . . .   // others args
            break;
          default: . . .   // usage message
            System.exit(-1);
        }
        . . .
</pre></td><td><pre>

int main(int argc, char *argv[]) {
    switch(argc) {
      case 1: . . .   // no args
        break;
      case 2: . . . args[1] . . . // 1 arg
        break;
      case . . .   // others args
        break;
      default: . . .   // usage message
        exit(-1);
    }
    . . .
</pre></td></tr>
</table>

- Arguments are processed in the range `argv[1]` through `argv[argc - 1]`, i.e., starting one greater than Java.

2.12 Object

- Object-oriented programming was developed in the mid-1960s by Dahl and Nygaard and first implemented in SIMULA67.
- Objects are structure based, used for organizing logically related data:

unorganized	organized

int people_age[30];
bool people_sex[30];
char people_name[30][50];
	struct Person {
	    int age;
	    bool sex;
	    char name[50];
	} people[30];

- Both approaches create an identical amount of information.
- Difference is solely in the information organization (and memory layout).
- Computer does not care as the information and its manipulation is largely the same.
- Structuring is an administrative tool for programmer understanding and convenience.
- Objects extend organizational capabilities of the structure by allowing routine members.

<table>
<tr><th>structure form</th><th>object form</th></tr>
<tr><td><pre>struct Complex {
    double re, im;
};
double abs(Complex &This) {
    return sqrt(This.re * This.im);
}
Complex x; // structure
abs(x);   // call abs</pre></td><td><pre>struct Complex {
    double re, im;
    double abs() {
        return sqrt(re * im);
    }
};
Complex x; // object
x.abs();   // call abs</pre></td></tr>
</table>

- ***Each object provides both data and the operations necessary to manipulate that data in one self-contained package.***
- Routine member is constant, and cannot be assigned (e.g., **const** member).
- What is the scope of a routine member?
- Structure creates a scope, and therefore, a routine member can access the structure members, e.g., abs member can refer to members re and im.
- Structure scope is implemented via a pointer-to-structure parameter, called **this**, implicitly passed to each routine member (like left example).

```
double abs() { return sqrt( this->re + this->im ); }
```

- Except for the syntactic differences, the two forms are identical.
- ***Like Java, the use of implicit parameter* this*, e.g.,* this->f*, is seldom necessary in C++.***
- Member routines are accessed like other members, using member selection, x.abs, and called with the same form, x.abs().
- No parameter needed because of implicit structure scoping via **this** parameter.
- Add arithmetic operations:

```
struct Complex {
    . . .
    Complex add( Complex c ) {
        Complex sum = { re + c.re, im + c.im };
        return sum;
    }
};
```

- To sum x and y, write x.add(y).

- Because addition is a binary operation, add needs a parameter as well as the implicit context in which it executes.

2.12.1 Operator Member

- It is possible to use operator symbols for routine names:

```
struct Complex {
    . . .
    Complex operator+( Complex c ) {
        Complex sum = { re + c.re, im + c.im };
        return sum;
    }
};
```

- Addition routine is called +, and x and y can be added by x.**operator**+(y) or y.**operator**+(x), which is only slightly better.
- For convenience, C++ implicit rewrites x + y as x.**operator**+(y).

```
Complex x = { 3.0, 5.2 }, y = { -9.1, 7.4 };
cout << "x:" << x.re << "+" << x.im << "i" << endl;
cout << "y:" << y.re << "+" << y.im << "i" << endl;
Complex sum = x + y;
cout << "sum:" << sum.re << "+" << sum.im << "i" << endl;
```

2.12.2 Type Nesting

- Type nesting is useful for controlling visibility for types:

```
struct Foo {
    enum Colour { R, G, B };        // nested type
    . . .
};
```

- Enumeration Colour is nested in Foo to control visibility.
- References outside the object must be qualified with type operator "::":

```
Foo::Colour colour = Foo::R;
```

- C++ selection operator ".", e.g., Foo.Colour, cannot be used because it requires an object not a type.
- Aggregate types may be nested, but ***nesting does not imply scoping***:

```
struct Foo {
    int g;
    int r(...) { ... }
    struct Bar {                            // nested object type
        int s(...) { g = 3; r(...); }       // references to g and r fail
    };
};
```

- In effect, structure nesting is flattened.
- As a result, the references in routine s to members g and r in Foo fail because there is no scope relationship between types Bar and Foo.

2.12.3 Constructor

- A **constructor** is a special member used to perform initialization after object allocation to ensure the object is valid before use.

```
struct Complex {
    double re, im;
    Complex() { re = 0.; im = 0.; } // default constructor
    ... // other members
};
```

- Constructor name is unusual because it is overloaded with the type name of the structure in which it is defined.
- Constructor without parameters is the **default constructor** and is implicitly called after storage allocation:

`Complex x;`
`Complex *y = new Complex;`
	implicitly rewritten as	`Complex x; x.Complex();`
		`Complex *y = new Complex;`
		`y->Complex();`

- Unlike Java, C++ does not initialize all object members to default values.
- When a C++ constructor executes, the constructor is responsible for initializing members not initialized via other constructors.
- Because a constructor is a routine, arbitrary execution can be performed (e.g., loops, routine calls, etc.) to perform initialization.
- A constructor may have parameters but no return type (not even **void**).
- ***When declaring a local object in C++, never put parenthesis to invoke the default constructor:***

```
Complex x(); // routine with no parameters and returning a complex
```

- Once a constructor is specified, structure initialization is disallowed:

```
Complex x = { 3.2 };              // disallowed
Complex y = { 3.2, 4.5 };         // disallowed
```

- Replaced using overloaded constructors with parameters:

```
struct Complex {
    double re, im;
    Complex() { re = 0.; im = 0.; }
    Complex( double r ) { re = r; im = 0.; }
    Complex( double r, double i ) { re = r; im = i; }
    . . .
};
```

- Unlike Java, constructor argument(s) can be specified *after* a variable for local declarations:

```
Complex x, y(1.0), z(6.1, 7.2);
```

implicitly rewritten as

```
Complex x; x.Complex();
Complex y; y.Complex(1.0);
Complex z; z.Complex(6.1, 7.2);
```

- Dynamic allocation is same as Java:

```
Complex *x = new Complex(); // parenthesis optional
Complex *y = new Complex(1.0);
Complex *z = new Complex(6.1, 7.2);
```

- Unlike Java, constructor cannot be called explicitly at start of another constructor, so constructor reuse done through a separate member:

Java	C++

`class Foo {`
`    int i, j;`
 	
`    Foo() { this( 2 ); } // explicit call`
`    Foo( int p ) { i = p; j = 1; }`
`}`
	`struct Foo {`
	`    int i, j;`
	`    void common( int p ) { i = p; j = 1; }`
	`    Foo() { common( 2 ); }`
	`    Foo( int p ) { common( p ); }`
	`};`

2.12.3.1 Constant

- Constructors can be used to create object constants, like g++ type-constructor constants:

```
Complex x, y, z;
x = Complex( 3.2 );          // complex constant with value 3.2+0.0i
y = x + Complex(1.3, 7.2); // complex constant with value 1.3+7.2i
z = Complex( 2 ); // 2 widened to 2.0, complex constant with value 2.0+0.
```

- Previous operator + for Complex is changed because type-constructor constants are disallowed for a type with constructors:

```
Complex operator+( Complex c ) {
    return Complex( re + c.re, im + c.im ); // create new complex value
}
```

2.12.3.2 Conversion

- Constructors are implicitly used for conversions:

```
int i;
double d;
Complex x, y;
```

`x = 3.2;`		`x = Complex( 3.2 );`
`y = x + 1.3;`	implicitly	`y = x.operator+( Complex(1.3) );`
`y = x + i;`	rewritten as	`y = x.operator+( Complex( (double)i ) );`
`y = x + d;`		`y = x.operator+( Complex( d ) );`

- Allows built-in constants and types to interact seamlessly with user-defined types.
- Note, two implicit conversions are performed on variable i in x + i: **int** to **double** and then **double** to Complex.
- Implicit constructor conversion is turned off with qualifier **explicit**:

```
struct Complex {
    . . .
    explicit Complex( double r ) { re = r; im = 0.; } // turn off
    . . .                                            // implict conversion
};
```

- However, this capability fails for commutative binary operators.
- 1.3 + x, fails because it is rewritten as (1.3).**operator+**(x), but member **double operator+**(Complex) does not exist in built-in type **double**.
- Solution, move operator + out of the object type and made into a routine, which can also be called in infixed form:

```
struct Complex { . . . }; // same as before, except operator + removed
Complex operator+( Complex a, Complex b ) { // 2 parameters
    return Complex( a.re + b.re, a.im + b.im );
}
```

x + y;	implicitly rewritten as	+(x, y)
1.3 + x;		+(Complex(1.3), x)
x + 1.3;		+(x, Complex(1.3))

- Compiler first checks for an appropriate operator in object type, and if found, applies conversions only on the second operand.
- If no appropriate operator in object type, the compiler checks for an appropriate routine (it is ambiguous to have both), and if found, applies applicable conversions to *both* operands.
- In general, commutative binary operators should be written as routines to allow implicit conversion on both operands.

2.12.3.3 Copy

- Constructor with a **const** reference parameter is the **copy constructor**:

```
Complex( const Complex &c ) { . . . }
```

- Used in two important initialization contexts: declarations and parameters.
- Declaration initialization:

  `Complex y = x` implicitly rewritten as `Complex y; y.Complex( x );`

- Operator "=" is misleading because it calls copy constructor not assignment operator.
- Value on the right-hand side of assignment is argument to copy constructor.
- Parameter initialization:

```
Complex foo( Complex a, Complex b );
Complex x, y;
foo( x, y )
```

- Call `foo( x, y )` performing the following implicit action in `foo`:

```
Complex foo( Complex a, Complex b ) {
    a.Complex( x ); b.Complex( y ); // initialize parameters with arguments
```

- If a copy constructor is not specified, an implicit one is generated that does a bit-wise copy.

- Why does C++ differentiate between copy and assignment?
- For copy situation (and constructors in general), after allocation, an object's members contain undefined values (unless a member has a constructor) and a constructor initializes appropriate members.
- For assignment, `lhs = rhs`, the left-hand variable may contain values and assignment only needs to copy a subset of values from the right-hand variable.
- For example, if an object type has a member variable to count the number of assignments, the counter is set to zero on initialization and incremented on assignment.
- Hence, changing a variable in C++ can be redefined to selectively modify its members.

2.12.3.4 const/Object Member

- Unlike Java, a C/C++ **const** member of a structure must be initialized at the declaration:

```
struct Foo {
    const int i; ...
} x = { 3 }; // must be initialized as it is write-once/read-only
```

- However, this form of initialization is disallowed for objects, and must be replaced with a constructor:

```
struct Foo {
    const int i; ...
    Foo() { i = 3; } // attempt to initialize const member
};
```

- However, this fails because it is assignment not initialization, and a **const** variable can only be initialized to ensure a read does not occur before the initial write.
- Therefore, a special syntax is used for initializing **const** members of an object *before* the constructor is executed:

<table>
<tr><th>Java</th><th>C++</th></tr>
<tr><td>

```
class Bar {}
class Foo {
    final int i;

    final Bar rp;
    Foo ( Bar b ) {
        i = 3;

        rp = b;
        ...
    }
}
```

</td><td>

```
class Bar {};
class Foo {
    const int i;
    Bar * const p;  // explicit const pointer
    Bar &rp;        // implicit const reference
    Foo ( Bar b ) : // initializing const members
        i( 3 ),
        p( &b ),    // explicit referencing
        rp( b ) {   // implicit referencing
        ...
    }
};
```

</td></tr>
</table>

- In the example, member i is initialized to 3, and p and r are initialized to point at argument b, for the object's lifetime.
- This syntax is also used for local objects with constructors, and can be used to initialize non-**const** members:

```
struct Bar {
    Bar( int i ) {...}
};
struct Foo {
    Bar b( 3 );      // fails
    int i;
    Foo() : b( 3 ), i( 3 ) {...} // b initialized here
};
```

2.12.4 Destructor

- A **destructor** (finalize in Java) is a special member used to perform uninitialization at object deallocation:

Java	C++

`class Foo {`
`    . . .`
`    finalize() { . . . }`
`}`
	`struct Foo {`
	`    . . .`
	`    ~Foo() { . . . } // destructor`
	`};`

- An object type has one destructor; its name is the character "~" followed by the type name (like a constructor).

- A destructor has no parameters nor return type (not even **void**):
- ***A destructor is only necessary if an object changes its environment***, e.g., closing files, freeing dynamically allocated storage, etc.
- A self-contained object, like a Complex object, requires no destructor.
- A destructor is invoked immediately *before* an object is deallocated, either implicitly at the end of a block or explicitly by a **delete**:

```
{
    Foo x, y;
    Foo *z = new Foo;
    . . .
    delete z;
    . . .
}
```

implicitly rewritten as

```
{ // allocate local storage
    Foo x; x.Foo(); y.Foo();
    Foo *z = new Foo; z->Foo();
    . . .
    z->~Foo(); delete z;
    . . .
    y.~Foo(); x.~Foo();
} // deallocate local storage
```

- For local variables in a block, destructors are called in ***reverse*** order to constructors (independent of explicit **delete**).
- A destructor is more common in C++ than a finalize in Java due to the lack of garbage collection in C++.

- ***If an object type performs dynamic storage allocation, it needs a destructor to free the storage:***

```
struct Foo {
    int *i;    // think int i[]
    Foo( int size ) { i = new int[size]; } // dynamic allocation
    ~Foo() { delete [] i; }   // must deallocate storage
    . . .
};
```

- Also, a C++ destructor is invoked at a deterministic time (block termination or **delete**), ensuring prompt cleanup of the execution environment.
- A Java finalize is invoked at a non-deterministic time during garbage collection or *not at all*, so cleanup of the execution environment is unknown.

2.13 Forward Declaration

- C/C++ have **Declaration Before Use** (DBU), e.g., a variable declaration must appear before its usage in a block:

```
{
    i += 1;      // no prior declaration of i
    int i;       // declaration after usage
}
```

- A compiler can handle some DBU situations, but there are ambiguous cases:

```
int i;
{
    i += 1;      // now which i should be used?
    int i;       // declaration after usage
}
```

- C always requires DBU.
- C++ requires DBU in a block and among types but not within a type.
- Java only requires DBU in a block, but not for declarations in or among classes.
- DBU has a fundamental problem specifying **mutually recursive** references:

```
void f() {    // f calls g
    g();      // g is not defined and being used
}
void g() {    // g calls f
    f();      // f is defined and can be used
}
```

- Cannot type-check the call to g in f to ensure matching number and type of arguments and the return value is used correctly.
- Clearly, interchanging the two routines does not solve the problem.
- A **forward declaration** introduces a routine's type before its actual declaration:

```
int f( int i, double ); // routine prototype: parameter names optional
...                     // and no routine body
int f( int i, double d ) { // type repeated and checked with prototype
    ...
}
```

- Prototype parameter names are optional (good documentation).
- Actual routine declaration repeats routine type, which must match prototype.

- Routine prototypes also useful for organizing routines in a source file.

```
void g( int );        // forward declarations without parameter names
void f( int );
int main() {          // appears first rather than last
    f( 5 );           // actual declarations later
    g( 4 );
}
void g( int i ) { ... } // actual declarations
void f( int i ) { ... }
```

- E.g., allowing main routine to appear first, and for separate compilation.
- Like Java, C++ does not require DBU for mutually-recursive routines within a type:

```
struct T {
    void f( int i ) { ... g(...); ... }  // g is not defined but it works!
    void g( int i ) { ... f(...); ... }
};
```

- Unlike Java, C++ requires a forward declaration for mutually-recursive declarations among types:

Java	C++

<pre>class T1 {
    final T2 t2;
    T1(final T2 t2) { this.t2 = t2; }
    void g(int i) { . . . t2.f(. . .) . . . }
}
class T2 {
    final T1 t1
        = new T1(this);
    void f(int i) { . . . t1.g(. . .) . . . }
}</pre>
	<pre>struct T2; // forward declaration, no body
	struct T1 { // T1 referencing T2
	    T2 &t2; // know about T2 from forward
	    T1(T2 &t2) : t2(t2) {} // constructor initialize
	    void g(int i) { . . . t2.f(. . .); . . . } // FAILS!!!
	};
	struct T2 { // T2 referencing T1
	    T1 &t1;
	    T2() : t1(*this) {} // constructor initialize
	    void f(int i) { . . . t1.g(. . .); . . . }
	};</pre>

- The forward declaration of T2 allows the declaration of variable T1::t2.
- Note, a forward declaration only introduces the name of a type.
- Given just a type name, only pointer/reference declarations to the type are possible, which allocate storage for an address versus an object.
- An object declaration and usage requires the object’s size and members so storage can be allocated, initialized, and usages type-checked.
- As a result, the C++ usage t2.f in T1::g fails because the information about type T2’s members is defined later.

- Is it possible to change the declaration of T2::t1 from T1 &t1 to T1 t1, i.e., from a reference to an actual object?
- Java's solution to this problem is to find the definition of T2 to obtain needed information (not DBU).
- C++'s solution involves forward declarations and a syntactic trick (DBU).
- First, a member containing the non-DBU reference is replaced by a forward declaration:

```
struct T1 {                 // T1 referencing T2
    . . .                   // as above
    void g( int i );        // forward
};
```

- Second, a syntactic trick allows the actual member definition to be placed ***after both types are defined***:

```
void T1::g( int i ) { . . . t2.f(. . .); . . . }
```

- Now the compiler knows all the information about the types to verify usage in T1::g.

- Note, the trick use of qualified names T1::g to specify this is actually a member logically declared in T1 but physically located after the types.

2.14 Overloading

- **Overloading** occurs when a name has multiple meanings in the same context.
- Most languages have some overloading.
- E.g., most built-in operators are overloaded on both integral and floating-point operands, i.e., the + operator is different for 1 + 2 than for 1.0 + 2.0.
- Overloading requires the compiler to disambiguate among identical names based on some criteria.
- The normal criterion is type information.
- In general, overloading is done on operations not variables:

```
int i;                  // variable overloading disallowed
double i;
void r( int ) {}        // routine overloading allowed
void r( double ) {}
```

- Power of overloading occurs when the type of a variable changes: operations on the variable are implicitly reselected to the variable's new type.
- E.g., after changing a variable's type from **int** to **double**, all operations implicitly change from integral to floating-point.
- Like Java, C++ overloads the built-in operators for the basic types and allows users to overload members in a type.
- C++ also allows routines to be overloaded including operators.
- Number and types of the parameters ***but not the return type*** are used to select among a name's different meanings:

```
int r(int i, int j) { . . . }      // overload name r three different ways
int r(double x, double y) { . . . }
int r(int k) { . . . }
r( 1, 2 );          // invoke 1st r based on integer arguments
r( 1.0, 2.0 );      // invoke 2nd r based on double arguments
r( 3 );             // invoke 3rd r based on number of arguments
```

- Implicit conversions between arguments and parameters can cause problems:

```
r( 1, 2.0 ); // ambiguous, convert either argument to integer or double
```

- Use explicit cast to disambiguate:

```
r( 1, (int)2.0 )       // 1st r
r( (double)1, 2.0 )  // 2nd r
```

- Overlap between overloading and default arguments for parameters with same type:

Overloading	Default Argument

`int r( int i, int j ) { . . . }`
`int r( int i ) { int j = 2; . . . }`
`r( 3 ); // 2nd r`
	`int r( int i, int j = 2 ) { . . . }`
	
	`r( 3 ); // default argument of 2`

- If the overloaded routine bodies are essentially the same, use a default argument, otherwise use overloaded routines.
- I/O operators << and >> often overloaded for user types:

```
ostream &operator<<( ostream &os, Complex c ) {
    return os << c.re << "+" << c.im << "i";
}
cout << "x:" << x; // rewritten as: <<( cout.operator<<("x:"), x )
```

- Standard C++ convention for I/O operators to take and return a stream reference to allow cascading stream operations.
- << operator in object cout is used to first print string value, then overloaded routine << to print the complex variable x.
- Why write as a routine versus a member?

2.15 Inheritance

- object-oriented languages usually provide **inheritance** for writing general, reusable program components.

Java	C++
class Base { . . . } **class** Derived extends Base { . . . }	**struct** Base { . . . } **struct** Derived : **public** Base { . . . };

- Inheritance has two orthogonal sharing concepts: implementation and type, each is discussed separately.

2.15.1 Implementation Inheritance

- Implementation inheritance reuses declarations in one object to build another object.
- One way to understand this technique is to model it via explicit inclusion, e.g.:

Inclusion	Inheritance

struct Base {
    int i;
    int r(. . .) { . . . }
    Base() { . . . }
};
struct Derived {
    Base b; *// explicit inclusion*
    int s(. . .) { b.i = 3; b.r(. . .); . . . }
    Derived() { . . . }
} d;
d.b.i = 3; *// inclusion reference*
d.b.r(. . .); *// inclusion reference*
d.s(. . .); *// direct reference*
	struct Base {
	    int i;
	    int r(. . .) { . . . }
	    Base() { . . . }
	};
	struct Derived : **public** Base { *// implicit ir*
	
	    int s(. . .) { i = 3; r(. . .); . . . }
	    Derived() { . . . }
	} d;
	d.i = 3; *// direct reference*
	d.r(. . .); *// direct reference*
	d.s(. . .); *// direct reference*

- Inclusion implies explicitly creating an object member, b, to aid in the implementation.
- Object type Derived inherits from Base type via "**public** Base" clause.
- Inheritance implicitly:
  - creates an anonymous object member
  - ***opens*** the scope of anonymous member so its members are accessible without qualification, both inside and outside the inheriting object type.
- A Derived declaration must first implicitly create an invisible Base object in the Derived object, like inclusion, for the implicit references to Base::i and Base::r in Derived::s.
- As well, constructors and destructors must be invoked for all implicitly declared objects in the inheritance hierarchy as done for an explicit member in the inclusion.

```
                            Base b;  b.Base(); // implicit, hidden declaration
Derived d;   implicitly     Derived d;  d.Derived();
...          rewritten as   ...
                            d.~Derived();  b.~Base(); // reverse order of constru
```

- If included object type has members with the same name as including type, it works like nested blocks: a name in the inner scope hides (overrides) a name at the outer scope.
- It is always possible to access these members with "::" qualification to specify the particular nesting level.

Java	C++

<pre>class Base1 {
    int i;
}
class Base2 extends Base1 {
    int i;
}
class Derived extends Base2 {
    int i;
    void s() {
        int i = 3;
        this.i = 3;
        ((Base1)this).i = 3; // super.i
        ((Base2)this).i = 3;
    }
}</pre>
	<pre>struct Base1 {
	    int i;
	};
	struct Base2 : public Base1 {
	    int i; // hides Base1::i
	};
	struct Derived : public Base2 {
	    int i; // hides Base2::i
	    void r() {
	        int i = 3; // hides Derived::i
	        Derived::i = 3; // this.i
	        Base2::i = 3;
	        Base2::Base1::i = 3;
	    }
	};</pre>

- Implementation inheritance reuses program components by composing a new object's implementation from an existing object, taking advantage of previously written and tested code.
- Substantially reduces the time to compose and debug a new object type.

- Unfortunately, having to inherit all of the members is not always desirable; some members may be inappropriate for the new type (e.g, large array).
- As a result, both the inherited and inheriting object must be very similar to have so much common code.
- In general, routines provide smaller units for reuse than entire objects.

2.15.2 Type Inheritance

- Type inheritance extends name equivalence to allow routines to handle multiple types, called **polymorphism**, e.g.:

```
struct Foo {                    struct Bar {
    int i;                          int i;
    double d;                       double d;
} f;                            } b;
void r( Foo f ) { . . . }
r( f );   // valid call
r( b );   // should also work
```

- Since types Foo and Bar are identical, instances of either type should work as arguments to routine r.

- Even if type Bar has more members at the end, routine r only accesses the common ones at the beginning as its parameter is type Foo.
- However, name equivalence precludes the call r(b) even though b is structurally identical to f.
- Type inheritance relaxes name equivalence by aliasing the derived name with its base-type names:

```
struct Foo {                  struct Bar : public Foo { // inheritance
    int i;                        // no members
    double d;
} f;                          } b;
void r( Foo f ) { . . . }
r( f );   // valid call, derived name matches
r( m );  // valid call because of inheritance, base name matches
```

- E.g., create a new type Mycomplex that counts the number of times abs is called for each Mycomplex object.
- Use both implementation and type inheritance to simplify building type Mycomplex:

```
struct Mycomplex : public Complex {
    int cntCalls;                              // add
    Mycomplex() : cntCalls(0) {}               // add
    double abs() {   // override, reuse complex's abs routine
        cntCalls += 1;
        return Complex::abs();
    }
    int calls() { return cntCalls; }           // add
};
```

- Derived type Mycomplex uses the implementation of the base type Complex, adds new members, and overrides abs to count each call.
- Allows reuse of Complex's addition and output operation for Mycomplex values, because of the relaxed name equivalence provided by type inheritance between argument and parameter.
- Why is the qualification Complex:: necessary in Mycomplex::abs?
- Now variables of type Complex are redeclared to Mycomplex, and member calls returns the current number of calls to abs for any Mycomplex object.
- Implementation inheritance provides reuse ***inside*** an object type; type

inheritance provides reuse ***outside*** the object type by allowing existing code to access the base type.

- I.e, any routine that manipulates the base type also manipulates the derived type.
- Two significant problems with type inheritance.

1. – Complex routine **operator+** is used to add the Mycomplex values because of the relaxed name equivalence provided by type inheritance:

```
int main() {
    Mycomplex x;
    x = x + x;
}
```

   – However, the result type from **operator+** is Complex, not Mycomplex.
   – Assignment of a complex (base type) to Mycomplex (derived type) fails because the Complex value is missing the cntCalls member!
   – Hence, a Mycomplex can mimic a Complex but not vice versa.
   – This fundamental problem of type inheritance is called **contra-variance**.
   – C++ provides various solutions, all of which have problems and are

beyond this course.

2.
```
void r( Complex &c ) { c.abs(); }
int main() {
    Mycomplex x;
    x.abs();      // direct call of abs
    r( x );       // indirect call of abs
    cout << "x:" << x.calls() << endl;
}
```

– While there are two calls to abs on object x, only one is counted!

2.15.3 Virtual Routine

- When a member is called, it is usually obvious which one is invoked even with overriding:

```
struct Base {
    void r() { ... }
};
struct Derived : public Base {
    void r() { ... }        // override Base::r
};
Base b;
b.r();    // call Base::r
Derived d;
d.r();    // call Derived::r
```

- However, it is not obvious for arguments/parameters and pointers/references:

```
void s( Base &b ) { b.r(); }
s( d );            // inheritance allows call: Base::r or Derived::r ?
Base &bp = d;  // assignment allowed because of inheritance
bp.r();            // Base::r or Derived::r ?
```

- Inheritance masks the actual type of the object, but both calls should invoke Derived::r because argument b and reference bp point at an object of type Derived.
- If variable d is replaced with b, the calls should invoke Base::r.

- Programmer may want to access members in Base even if the actual object is of type Derived, which is possible because Derived *contains* a Base.
- C++ provides mechanism to override the default at the call site.
- To invoke the routine defined in the referenced object, qualify the member routine with **virtual**.
- To invoke the routine defined by the type of the pointer/reference, do not qualify the member routine with **virtual**.
- C++ uses non-virtual as the default because it is more efficient.
- Java ***always*** uses virtual for all calls to objects.
- Once a base type qualifies a member as virtual, ***it is virtual in all derived types regardless of the derived type's qualification for that member***.

Java	C++

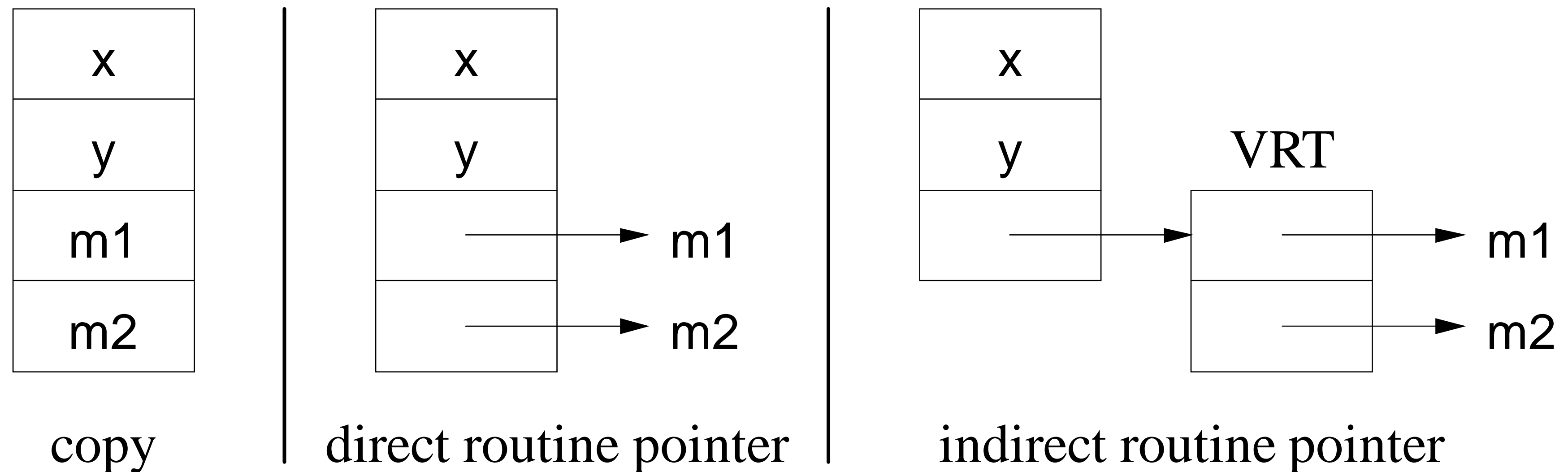
<pre>**class** Base {
    public void f() {} *// virtual*
    public void g() {} *// virtual*
    public void h() {} *// virtual*
}
class Derived extends Base {
    public void g() {} *// virtual*
    public void h() {} *// virtual*
}
final Base bp = **new** Derived();
bp.f();   *// Base.f*
((Base)bp).g();   *// Derived.g*
bp.g();   *// Derived.g*
((Base)bp).h();   *// Derived.h*
bp.h();   *// Derived.h*</pre>
	<pre>**struct** Base {
	    void f() {} *// non-virtual*
	    void g() {} *// non-virtual*
	    virtual void h() {} *// virtual*
	};
	struct Derived : **public** Base {
	    void g() {}; *// non-virtual*
	    void h() {}; *// virtual*
	};
	Base &bp = *****new** Derived(); *// polymorphic assi*
	bp.f(); *// Base::f, pointer type*
	bp.g(); *// Base::g, pointer type*
	((Derived &)bp).g(); *// Derived::g, pointer type*
	bp.Base::h(); *// Base::h, explicit selection*
	bp.h(); *// Derived::h, object type*</pre>

- Java casting does not provide access to base-type's member routines.
- ***Virtual members are only necessary to access derived members through a base type reference or pointer.***

- If a type is not involved in inheritance (final class in Java), virtual members are unnecessary so use more efficient call to its members.
- For inheritance, C++ virtual members are qualification in the base type as opposed to the derived type.
- Hence, C++ requires the base-type definer to presuppose how derived definers might want the call default to work.
- ***Good programming practice for inheritable object types is to make all routine members virtual.***
- Any type with virtual members and a destructor needs to make the destructor virtual so the most derived destructor is called through a base-type pointer/reference.
- Virtual routines are implemented by routine pointers.

```
class Base {
    int x, y;                   // data members
    virtual void m1(...);       // routine members
    virtual void m2(...);
};
```

- May be implemented in a number of ways:



2.15.4 Down Cast

- Type inheritance can mask the actual type of an object through a pointer/reference.
- Like Java, C++ provides a mechanism to dynamically determine the actual type of a pointer/reference.
- The Java operator instanceof and the C++ operator **dynamic_cast** perform a dynamic check of the object addressed by a pointer/reference:

Java	**C++**
Base bp = **new** Derived(); **if** (bp instanceof Derived) ((Derived)bp).rtn();	Base *bp = **new** Derived(); **if** (**dynamic_cast**<Derived *>(bp) != 0) ((Derived *)bp)->rtn();

- ***To use* dynamic_cast *on a type, the type must have at least one virtual member*.**

2.15.5 Constructor/Destructor

- Constructors are *implicitly* executed top-down, from base to most derived type.
- Mandated by scope rules, which allow a derived-type constructor to use a base type's variables so the base type must be initialized first.
- Destructors are *implicitly* executed bottom-up, from most derived to base type.
- Order is mandated by the scope rules, which allow a derived-type destructor to use a base type's variables so the base type must be uninitialized last.
- Java finalize must be *explicitly* called from derived to base type.

- Unlike Java, C++ disallows calls to other constructors at the start of a constructor.
- To pass arguments to other constructors, use the same syntax as for initializing **const** members.

Java	C++

class Base {
    Base(**int** i) { ... }
};
class Derived extends Base {
    Derived() { super(3); ... }
    Derived(**int** i) { super(i); ... }
};
	struct Base {
	    Base(**int** i) { ... }
	};
	struct Derived : **public** Base {
	    Derived() : Base(3) { ... }
	    Derived(**int** i) : Base(i) {...}
	};

2.15.6 Abstract Interface

- Create an abstract interface from which actual types are defined:

Java	C++

interface Shape {
    void move(**int** x, **int** y);
};
class Circle **implements** Shape {
    public void move(**int** x, **int** y) {}
};
	struct Shape {
	    virtual void move(**int** x, **int** y) **= 0**;
	};
	struct Circle : **public** Shape {
	    void move(**int** x, **int** y) {}
	};

- Note strange initialization of member Shape::move to 0, which means this member *must* be defined by any derived type of Shape.
- ***Cannot instantiate objects from an abstract interface.***
- C++ allows the abstract interface to contain actual members, which results in a combination of implementation inheritance and abstract description.

2.16 Template

- Inheritance provides reuse for types organized into a hierarchy that extends name equivalence.

- Alternate kind of reuse where no type hierarchy and types are not equivalent.
- E.g., overloading, where there is identical code but different types:

```
int abs( int val ) { return val >= 0 ? val : -val; }
double abs( double val ) { return val >= 0 ? val : -val; }
```

- Template routine eliminates duplicate code by using types as compile-time parameters:

```
template<typename T>  T abs( T val ) { return val >= 0 ? val : -val; }
```

- **template** introduces type parameter T used to declare return and parameter types.
- At a call, compiler infers type T from argument(s), and constructs a specialized routine with inferred type(s):

```
cout << abs( 1 ) << " " << abs( -1 ) << endl; // T-> int
cout << abs( 1.1 ) << " " << abs( -1.1 ) << endl; // T -> double
```

- Template type prevents duplicating code that manipulates different types.

- E.g., collection data-structures (e.g., stack), have common code to manipulate data structure, but type stored in collection varies:

```
template<typename T, int N = 10> struct Stack {
    T elems[N]; // maximum N elements
    int size;
    Stack() { size = 0; }
    void push( T e ) { elems[size] = e; size += 1; }
    T pop() { size -= 1; return elems[size]; }
};
```

- Type parameter, T, declares the element type of array elems, and return and parameter types of the member routines.
- Integer parameter, N, denotes the maximum stack size.
- For template types, the compiler cannot infer the type parameter, so it must be explicitly specified:

```
Stack<int, 20> si;            // stack of int
Stack<double> sd;             // stack of double
Stack< Stack<int> > ssi;      // stack of stack of int
si.push(3);
sd.push(3.0);
ssi.push( si );
int i = si.pop();
double d = sd.pop();
si = ssi.pop();
```

- ***There must be a space between the two ending chevrons or >> is parsed as operator>>.***
- C++ Standard Template Library (STL) provides different kinds of containers: vector, stack, queue, list, deque, set, map.
- STL vector container is an alternative to C/C++ arrays.

```
#include <vector>
int i, size;
cin >> size;
vector<int> vals(size);   // think int vals[size]
for ( i = 0; i < vals.size(); i += 1 ) {
    cin >> vals.at(i);    // think vals[i]
}
vector<int> v;            // think: int v[ ]
v = vals;                 // array assignment
for ( i = v.size() - 1; 0 <= i; i -= 1 ) {
    cout << v.at(i) << "  ";
}
cout << endl;
```

- vector is dynamically sized, length is accessible size, has subscript checking at, and supports assignment.
- Vector declaration ***may*** specify an initial size, e.g., vector<**int**> vals(size), like a dimension.
- While the size of a vector may increase (or decrease) dynamically, it is more efficient to dimension, when the size is known.
- Matrix declaration is a vector of vectors, e.g.,

vector< vector<**int**> > m(5), which specifies 5 rows.

```
#include <vector>
vector< vector<int> > m( 5 ); // 5 rows
for ( int r = 0; r < m.size(); r += 1 ) {
    m[r].resize( 4 );       // 4 columns per row
    for ( int c = 0; c < m[r].size(); c += 1 ) {
        m[r][c] = r+c;      // or m.at(r).at(c)
    }
}
for ( int r = 0; r < m.size(); r += 1 ) {
    for ( int c = 0; c < m[r].size(); c += 1 ) {
        cout << m[r][c] << ", ";
    }
    cout << endl;
}
```

- Before values can be assigned into a row, each row is dimensioned to the specific size, m[r].resize(4).
- All loop bounds are controlled using dynamic size of the row or column.
- If indexed (direct) access is not required, use more efficient STL list container(s):

```
#include <list>
struct Node {
    char c;  int i;  double d;
    Node( char c, int i, double d ) : c(c), i(i), d(d) {}
};
list<Node> top;                              // doubly linked list
for ( int i = 0; i < 10; i += 1 ) {          // create list nodes
    Node n( 'a'+i, i, i+0.5 );               // node to be added
    top.push_back( n );                      // copy node at end of list
}
list<Node>::iterator ni;                     // iterator for doubly linked list
for ( ni = top.begin(); ni != top.end(); ++ni ) { // traverse list
    cout << "c:" << ni->c << " i:" << ni->i << " d:" << ni->d << endl
}
cout << endl;
while ( 0 < top.size() ) {                   // destroy list nodes
    Node n = top.front();                    // copy node at front of list
    top.erase( top.begin() );                // remove first node
    cout << "c:" << n.c << " i:" << n.i << " d:" << n.d << endl;
}
```

- First loop creates and initializes a node, and calls push_back to copy

node at end (back) of list.

- push_back is also used with vector to extend a vector's size.
- Containers either copy nodes into the list or point to the nodes outside the list.
- Copying implies node type must have default and/or copy constructor so instances can be created without having to know constructor arguments.
- STL containers use copying and requires node type to have a default constructor.
- Containers use an **iterator** to traverse nodes so knowledge about container implemented is hidden.
- Iterator capabilities depend on container, e.g., a singly linked list only allows unidirectional traversal while doubly linked list allows bidirectional traversal.
- STL containers provides iterator(s) as a nested object type, e.g., list<Node> has list<Node>::iterator.
- Second loop traverses list using iterator index, ni, from start of the list, stepping through nodes until ni is ***past*** the list end().
- Note, iterator ni points to a node in the list so field access is with ->.

- As well, the operator “++” advances ni to the next node.
- Third loop destroys list by repeatedly erasing the first node until the number of nodes is zero.
- For bidirectional iterators, operator “--” moves in the reverse direction to “++”.
- STL template-routine for_each provides an alternate mechanism to iterate through a container, applying an action to each node:

```
#include <iostream>
#include <list>
#include <vector>
using namespace std;
void print( int i ) { cout << i << "  "; }        // print node
int main() {
    list< int > int_list;
    vector< int > int_vec;
    for ( int i = 0; i < 10; i += 1 ) {          // create lists
        int_list.push_back( i );
        int_vec.push_back( i );
    }
    for_each( int_list.begin(), int_list.end(), print ); // print each node
    for_each( int_vec.begin(), int_vec.end(), print );
}
```

- An action routine to `for_each` is called for each node in the container passing the node to the routine for processing.
- In general, the type of the action routine is `void rtn( T )`, where `T` is the type of the container node.
- E.g., `print` has an `int` parameter matching the container node type.

- More complex actions are possible by constructing a "function object", called a **functor**, using the routine-call operator.
- E.g., an action to print on a specified stream must store the stream and have an **operator**() allowing the object to behave like a function:

```
struct Print {
    ostream &stream;             // stream used for output
    Print( ostream &stream ) : stream( stream ) {}
    void operator()( int i ) { stream << i << " "; }
};
int main() {
    list< int > int_list;
    vector< int > int_vec;
    for_each( int_list.begin(), int_list.end(), Print(cout) );
    for_each( int_vec.begin(), int_vec.end(), Print(cerr) );
}
```

- Expression Print(cout) creates a constant Print object, and for_each calls **operator**()(Node) in the object.

2.17 Namespace

- C++ has a mechanism to organize complex programs and libraries composed of multiple types and declarations.
- E.g., namespace std contains all the I/O declarations and container types.
- Names in a namespace form a declaration region, like the scope of block.
- Unlike Java, C++ allows multiple namespaces to be defined in a file.
- Types and declarations do not have to be added consecutively.

Java source file	C++ source file

package Foo; *// one package / file*
// types / declarations
	namespace Foo {
	    // types / declarations
	};
	namespace Bar {
	    // types / declarations
	};
	namespace Foo {
	    // more types / declarations
	};

- Contents of a namespace can be accessed using full-qualified names:

Java	C++
Foo.T t = **new** Foo.T();	Foo::T *t = **new** Foo::T();

- Or by importing individual items or all of the namespace content.

Java	C++
import Foo.T; import Foo.*;	**using** Foo::T; *// import individual* **using namespace** Foo; *// import all*

2.18 Abstraction/Encapsulation

- **Abstraction** is the separation of interface and implementation allowing an object's implementation to change without affecting usage, which is essential for reuse and maintenance.
- E.g., a user of type Complex should not have or need direct access its implementation to perform operations:

```
struct Complex {
    double re, im;  // implementation data
    ... // interface routine members
};
```

- Possible to change from Cartesian to polar coordinates and user interface remains constant.
- ***Developing good interfaces for objects is important.***
- **Encapsulation** is hiding the implementation for security or financial reasons (**access control**).
- ***Abstraction and encapsulation are neither essential nor required to develop software.***
- Uses follow a convention of not directly accessing the implementation.
- However, relying on users to follow conventions is dangerous.
- Encapsulation is provided by a combination of C and C++ features.
- C features work largely among source files, and are indirectly tied into separate compilation.
- C++ features work both within and among source files.
- Like Java, C++ provides 3 levels of visibility control for object types:

Java	**C++**

class Foo {
    private . . .
    . . .
    protected . . .
    . . .
    public . . .
    . . .
};
	struct Foo {
	  private: *// within and friends*
	    // private members
	  protected: *// within, friends, inherited*
	    // protected members
	  public: *// within, friends, inherited, users*
	    // public members
	};

- Java requires encapsulation specification for each member.
- C++ groups members with the same encapsulation, i.e., all members after a label, **private**, **protected** or **public**, have that visibility.
- Visibility labels can occur in any order and multiple times in an object type.
- Only the object type can access the private members, ***so implementation members are normally private***.
- Inherited object types can access and modify public and protected members allowing access to some of an object's implementation.

- Public members define an object type's **interface**, i.e., what a user can access.
- While a user can see private and protected members, they cannot be accessed, preventing code from violating abstraction.
- **struct** has an implicit **public** inserted at the beginning, i.e., all members are public.
- **class** is the same as **struct** except it has an implicit **private** at the beginning, i.e., all members are private:

```
class Base {
  private:
    int x;
  protected:
    int y;
  public:
    int z;
};
class Derived : public Base {
  public:
    Derived() { x; y; z; };
};
int main() {
    Derived d;
    d.x; d.y; d.z;
}
```

- Encapsulation introduces a new problem for routines outside of an object used to implement binary operations for an object.
- An outside routine may need to access an object's implementation, but it cannot access private members.
- C++ provides a mechanism to state that an outside routine is allowed

access to its implementation, called *friendship* (similar to package visibility in Java).

```
class Complex {
    friend Complex operator+(Complex a, Complex b);
    . . .
};
Complex operator+(Complex a, Complex b) { . . . }
```

- The **friend** prototype indicates a routine with the specified name and type may access this object's implementation:

```
class Complex {
    friend Complex operator+(Complex a, Complex b);
    friend ostream &operator<<( ostream &os, Complex c );
    double re, im;
  public:
    double abs() { return sqrt( re * re + im * im ); }
    Complex() { re = 0.; im = 0.; }
    Complex(double r) { re = r; im = 0.; }
    Complex(double r, double i) { re = r; im = i; }
};
Complex operator+( Complex a, Complex b ) { . . . }
ostream &operator<<( ostream &os, Complex c ) { . . . }
```

2.19 Separate Compilation

- Like Java's package access, a C/C++ **source file** provides another mechanism for encapsulation.
- By default, all global variables and routines in a source file are exported outside the file (package).
- To encapsulate declarations in a source file, the declaration must be qualified with **static**.

```
// file.C
int i;                      // public (exported)
void f(. . .) {}            // public (exported)
static int j;               // private
static void g(. . .) {}     // private
```

- Like Java, a type is encapsulated in a source file, unless explicitly denoted as public.
- Java has automatic access to public contents of a source file.
- First, C/C++ require the use of the preprocessor and forward declarations to access public contents.
- Declarations are divided into its interface and implementation in two (or more) files.
- Interface declarations are usually composed of the prototype declaration(s) (but possibly some implementation).
- Implementation declarations are composed of the actual declarations and code.
- Second, interface is entered into one or more include files (.h files), and the implementation is entered into one or more source files (.C files).

- Encapsulation is provided by giving a user access to only the include file(s) and the compiled source file(s), but not the implementation in the source file(s).
- Most software supplied from software vendors comes this way.
- Include files contain prototypes for exported variables and routines, which are qualified with **extern** (not types):

```
// file.h
extern int i;          // public, implementation elsewhere
extern void f(...);    // public, implementation elsewhere (extern optional for
```

- Complex prototype information is placed into file complex.h, which users include in their programs.

```
#ifndef __COMPLEX_H__
#define __COMPLEX_H__          // protect against multiple inclusion
#include <iostream>            // access: ostream
using std::ostream;
extern void complexStats();
class Complex {
    friend Complex operator+( Complex a, Complex b );
    friend ostream &operator<<( ostream &os, Complex c );
    double re, im;             // exposed implementation
  public:
    Complex();
    Complex( double r );
    Complex( double r, double i );
    double abs();
};
extern Complex operator+( Complex a, Complex b );
extern ostream &operator<<( ostream &os, Complex c );
#endif // __COMPLEX_H__
```

- Complex implementation information is placed in file complex.C.

```
#include "complex.h"
#include <cmath>                              // access: sqrt
using namespace std;
// private declarations
static int cplxObjCnt = 0;                    // must be initialized
// interface declarations
void complexStats() { cout << cplxObjCnt << endl; }
complex::complex() { re = 0.; im = 0.; cplxObjCnt += 1; }
complex::complex( double r ) { re = r; im = 0.; cplxObjCnt += 1; }
complex::complex(double r, double i) { re = r; im = i; cplxObjCnt += 1; }
double complex::abs() { return sqrt( re * re + im * im ); }
complex operator+( complex a, complex b ) {
     return complex( a.re + b.re, a.im + b.im );
}
ostream &operator<<( ostream &os, complex c ) {
    return os << c.re << "+" << c.im << "i";
}
```

- .C file normally includes the .h file so that there is only one copy of the constants, declarations, and prototype information.
- cplxObjCnt is qualified with **static** to make it a private variable to this source file.

- No user can access it, but each constructor implementation can increment it when a Complex object is created.
- ***All static variables, whether in a class or file, must be explicitly initialized in the .C file***, e.g., cplxObjCnt is set to 0.
- Users call complexStats to print the number of Complex objects created so far in a program.
- Notice, all the member routines of Complex are separated into a forward declaration and an implementation after the object type, allowing the implementation to be placed in the .C file.
- Note, by reading .h, it may be possible to determine the implementation technique used, so there is only partial encapsulation.
- To provide complete encapsulation requires abstract type and (more expensive) references:

```
#ifndef __COMPLEX_H__
#define __COMPLEX_H__            // protect against multiple inclusion
#include <iostream>              // access: ostream
using std::ostream;
extern void complexStats();
class Complex {
    friend Complex operator+( Complex a, Complex b );
    friend ostream &operator<<( ostream &os, Complex c );
    struct ComplexImpl;          // hidden implementation, nested class
    ComplexImpl &impl;           // indirection to implementation
  public:
    Complex();
    Complex( double r );
    Complex( double r, double i );
    ~Complex();
    Complex( const Complex &c ); // copy constructor
    Complex &operator=( const Complex &c ); // assignment operator
    double abs();
};
extern Complex operator+( Complex a, Complex b );
extern ostream &operator<<( ostream &os, Complex c );
#endif // __COMPLEX_H__
```

- ***Compiler requires a template definition for each usage so both the interface and implementation of a template must be in a `.h` file***, precluding some forms of encapsulation.

```
#include "complex.h"
#include <cmath>                        // access: sqrt
using namespace std;
// private declarations
static int cplxObjCnt = 0;
struct Complex::ComplexImpl {           // actual implementation, nested class
    double re, im;
};
// interface declarations
void complexStats() { cout << cplxObjCnt << endl; }
Complex::Complex() : impl(*new ComplexImpl) {
    impl.re = 0.; impl.im = 0.; cplxObjCnt += 1;
}
Complex::Complex( double r ) : impl(*new ComplexImpl) {
    impl.re = r; impl.im = 0.; cplxObjCnt += 1;
}
Complex::Complex( double r, double i ) : impl(*new ComplexImpl) {
    impl.re = r; impl.im = i; cplxObjCnt += 1;
}
Complex::~Complex() { delete &impl; }
Complex::Complex(const Complex &c) : impl(*new ComplexImpl) {
    impl.re = c.impl.re; impl.im = c.impl.im; cplxObjCnt += 1;
}
```

```
Complex &Complex::operator=(const Complex &c) {
    impl.re = c.impl.re; impl.im = c.impl.im; return *this;
}
double Complex::abs() {
    return sqrt( impl.re * impl.re + impl.im * impl.im );
}
Complex operator+( Complex a, Complex b ) {
    return Complex( a.impl.re + b.impl.re, a.impl.im + b.impl.im );
}
ostream &operator<<( ostream &os, Complex c ) {
    return os << c.impl.re << "+" << c.impl.im << "i";
}
```

- A copy constructor and assignment operator must be used because complex objects now contain a reference pointer to the implementation.
- A reference pointer cannot be copied on initialization or assignment without generating storage management problems.
- E.g., copying the reference pointer can result in two complex objects pointing at the same complex value and both may eventually attempt to delete it.
- As well, overwriting a reference pointer may lose the only pointer to the

storage so it can never be freed.

- An encapsulated object is compiled using the -c compilation flag and subsequently linked with other compiled source files to form a program:

```
g++ -c complex.C
```

- Creates file complex.o containing a compiled version of the source code.
- To use an encapsulated object, a program specifies the necessary include file(s) to access the object's interface:

```
#include "complex.h"
#include <iostream>
using namespace std;
int main() {
    Complex x, y, z;
    x = Complex( 3.2 );
    y = x + Complex( 1.3, 7.2 );
    z = Complex( 2 );
    cout << "x:" << x << " y:" << y << " z:" << z << endl;
}
```

- Then links with any necessary executables:

```
g++ usecomplex.C complex.o
```

- Notice, iostream is included twice, once in this program and once in complex.h, which is why each include file needs to prevent multiple inclusions.

3 Software Tools

3.1 Shell

- After signing onto a computer, there must exist a way to display information and perform operations.
- The two main approaches are graphical and command-line.
- A **graphical interface**:
  - uses icons to represent programs (actions),
  - clicking on an icon launches (starts) a program,
  - the program may pop up a dialog box to obtain arguments to specify its execution.
- A **command-line interface**:
  - uses text strings (names) to represent programs (commands),
  - a command is typed after a prompt in an interactive area to start it,
  - arguments follow the command to specify its execution.
- Graphical interfaces can be convenient for people, but seldom generalizes to a programming environment.

© Peter A. Buhr

- Command-line interfaces are slightly more work for people, but generalizes to a programming environment.
- A **shell** is a program that reads commands and shell statements, and interprets them.
- Shell statements often form a complete programming language with *string* variables and executable statements.
- Unix shell falls into two basic camps, sh and csh, each with different syntax and semantics.
- sh has variants: ksh, bash
- csh has variants: tcsh
- In UNIX, the area (window) in which a shell runs is called an **xterm**.
- Each shell line begins with a prompt denoted by the % sign (the prompt can be customized).
- A command is typed after the prompt.
- A command is *not* executed until <Return> is pressed:

```
% date<Return>
Sun Oct 19 19:27:30 EDT 2008
% uname
SunOS
```

- Most commands have options, specified by a minus followed by one or more characters, which change how the command operates.

```
% uname -a
SunOS services16.student.cs 5.8 Generic_117350-56 sun4u sparc SUNW
```

- Unfortunately, there is no standardization for option syntax and names.
- Most shells terminate with command exit.
- If an xterm's shell terminates, the xterm terminates, and if the xterm is the login window, you are signed off of the computer.

3.1.1 File System

- Files are containers for data stored on secondary storage (e.g., usually disk).
- Each file has a unique name.

- UNIX organizes file names in a hierarchy: directories are the vertices and files are the leaves.

```
/           root of the local file system
    usr
        bin         more UNIX commands
        lib         system libraries
        include             system include files, .h files
    bin         basic UNIX commands
    lib         system libraries
    tmp         system temporary files
    u1          user files
    u2          user files
        . . .
        jfdoe           student home directory
            .cshrc, .emacs, .login, . . .          student's files
            cs246
                assign1
                    q1, q2, q3
        . . .
    u           magic directory combining what is under u1-u5
```

- usr, etc, bin, lib – most of the UNIX commands, and system include and

library files.

- tmp – location of temporary files created by commands.
- u1, u2, u3, u4, u5 – user files are distributed across these directories. The directory for a particular user is called the user's **home directory**.
- u – *magic* directory that contains all of the users under the individual user directories.
- A file is referenced in one of two ways: **absolute pathname** or **relative pathname**.
- An absolute pathname is a list of all the directories from the root to the file separated by the character "/".
- E.g., the absolute pathname /u/jfdoe/cs246/assign1/q1 denotes the file q1.
- The name /u2/jfdoe/cs246/assign1/q1 denotes the same file.
- A relative pathname requires a starting location other than the root, called the **current directory**.
- When you sign on, the current directory is set to your home directory.
- Any file name not starting with "/" is automatically prefixed with the current directory.

- E.g., if user jfdoe signs on and specifies the file name cs246/assign1/q1, then the actual file used is /u/jfdoe/cs246/assign1/q1.
- There are special directory names, "." (dot), ". ." (dot dot), and "~" (tilde).
- "." is the name of the current directory, so ./cs246/assign1/q1 is the same as /u/jfdoe/cs246/assign1/q1.
- ". ." is the name of the directory above the current directory, i.e., the parent directory, so . ./jfdoe/cs246/assign1/q1 is the same as /u/jfdoe/cs246/assign1/q1.
- "~" is a user's home directory, so ~/cs246/assign1/q1 is the same as /u/jfdoe/cs246/assign1/q1.

3.1.2 Pattern Matching

- Shell's support pattern matching of file names (globbing) to reduce typing lists of file names.
- Pattern matching is provided through special characters, *, ?, [, {, denoting different wildcards.

- (Different shells and commands support slightly different forms and syntax for patterns.)
- A file name containing a special character is enclosed in quotes "".
- * matches 0 or more characters, e.g., if the current directory is /u/jfdoe/cs246/assign1, file name q*, matches file names q1, q2, q3.
- ?, matches 1 characters, e.g., file name q?, matches file names q1, q2, q3.
- [...], matches any characters in the set, e.g., filename q[123] matches file names q1, q2, q3
- ranges are possible using the hyphen, [0-3] matches characters 0,1,2,3, [a-zA-Z] matches a lower or upper case letter, [^a-zA-Z] matches any character not a letter.
- range can be modified with * to be any number of characters in the set, [a-zA-Z]* matches any number of lower or upper case letters.
- {...}, matches any alternative in the set, f.{cc,cpp,C}, matches f.cc, f.cpp, f.C.
- Patterns can be composed, e.g., q[0-9]*a*.c matches file names that start with q followed by 0 or more digits, followed by a, followed by 0 or more characters, and terminating with the two characters ".c".

3.2 Commands

3.2.1 Shell Commands

- Commands executed directly by the shell because they usually update its state.
- cd change the current directory.

  cd *[new-directory-path]*

  - argument must be a directory pathname and not a file pathname
  - cd .. moves you up one directory level
  - no directory pathname means moves to home directory (same as cd ~)
  - cd ~/bin moves to the bin directory contained in your home directory
  - If the specified path does not exist, cd fails and the current directory is not changed.
- time execute a command and print a time summary
- history print a numbered history of last N commands entered.
  - re-run command N, type "!N";
  - "!!" re-runs the last command.

- re-run last command starting with the string "xyz", use the command "!xyz".

- alias define string substitutions for command names.

```
alias [ command-name [=] value ]
```

  - without arguments, print all currently defined alias names and values.
  - value is substituted for command command-name (= may be required)
  - provide nickname for frequently use or variation of a command:

```
% alias d date
% d
Mon Oct 27 12:56:36 EDT 2008
```

  - aliases are composable:

```
% alias now d
% now
Mon Oct 27 12:56:37 EDT 2008
```

  - useful for setting command options for particular commands, as in:

```
% alias cp cp -i
% alias mv mv -i
% alias rm rm -i
```

which always uses the -i option on commands cp, mv and rm.

– A sequence of commands can be specified separated by semi-colons in quotation marks:

```
% alias off "clear; logout"
```

which clears the screen before logging off.

– An alias can be overridden by putting quoting the command name:

```
% "rm" -r xyz
```

which does not add the -i option

– An alias entered on a command line only takes effect for the current shell session.

– There are two options for making aliases permanent for across login session:

1. insert the alias commands in your .{*shell*}rc file
2. place a list of alias commands in a file called .aliases in your home directory and execute that file from your .{*shell*}rc file.

3.2.2 System Commands

- Commands executed by UNIX.

- pwd print the current directory.
- ls lists the directories and files in the specified directory.

  ls [-al] [*directory-name*]

  - -a lists *all* files, including those that begin with a dot.
  - -l generates a *long* listing for each file: mode, number of links, owner, size in bytes, last modification time and file name.
  - If no directory is given, the current directory is assumed.
- mkdir creates a new directory in the current directory.

  mkdir *directory-name-list*

- cp copies files, and with the -r option, copies directories.

  cp [-i] *source-file target-file*
  cp [-i] -r *source-file/directory-list target-directory*

  - -i prompt for verification if a target file is being replaced.
  - -r recursively copy the contents of a source directory to the target directory.
- mv moves files and/or directories to another location in the file hierarchy.

mv [-i] *source-file* *target-file*
mv [-i] *source-file* *target-directory*
mv [-i] *source-directory* *target-directory*

- if the target-file does not exist, the source-file is renamed; otherwise the target-file is replaced.
- -i prompt for verification if a target file is being replaced.

• rm removes (deletes) files, and with the -r option, removes directories.

rm [-ir] *directory-list*

- rmdir command is the same as rm -r.
- -i prompts for verification for each file/directory being removed.
- -r recursively delete the contents of a directory.
- UNIX does not give you a second chance to recover deleted files, so you must be very careful when using rm.

• more/less/cat list a file's content to standard out.

- more/less paginate the contents one screen at a time
- cat shows the contents in one continuous stream.

• lpr/lpq/lprm add, query and remove files from the printer queues.

```
lpr [-P printer-name] options file-list
lpq -P printer-name
lprm -P printer-name job-number
```

- if no printer is specified, the queue is a default printer.
- each job on a printer's queue has a unique number.
- use this number to remove a job from a print queue.

3.2.3 File Permissions

- UNIX file structure supports 3 levels of security on each file or directory:
  - user : owner of the file,
  - group : arbitrary name associated with a number of userids,
  - other : any other userid.
- At each level, a directory or file can have the following permissions: read, write, and execute (or search).
- Readable and writable allow any of the specified users to read or write/change a file/directory.
- Executable for files means the file can be executed as a command, e.g., file contains a program or shell script.

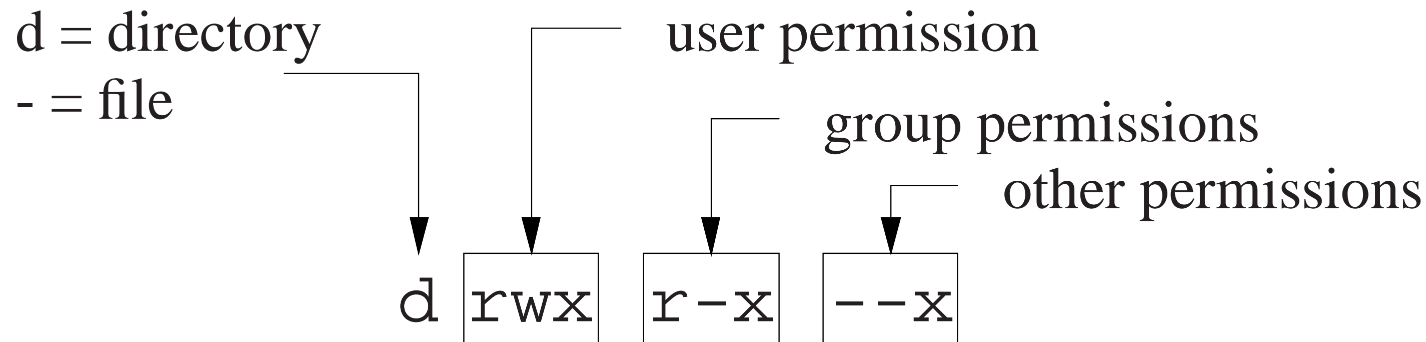
- Executable for directories means the directory can be searched by certain system operations but not read in general.
- Use ls -l to see file permission information in the current directory:

```
drwx------ 7 cs246 cs246  4096 Oct 20 13:07 ./
drwxr-x--- 5 cs246 cs246  4096 Oct 15 08:07 ../
drwx------ 2 cs246 cs246  4096 Oct 19 18:19 C++/
drwx------ 2 cs246 cs246  4096 Oct 21 08:51 Tools/
-rw------- 1 cs246 cs246 22714 Oct 21 08:50 notes.a
-rw------- 1 cs246 cs246 63332 Oct 21 08:50 notes.d
```

- Columns are permissions, #-files-in-directory, owner, group, file size, change date, file name.
- chgrp command changes the group associated with the file:

  chgrp *group-name file-list*

- Permission information is complex:



- E.g., drwxr-x---, indicates
  - directory in which the user has read, write and execute permissions,
  - group has only read and execute permissions,
  - others have no permissions at all.
- ***In general, you should not allow other users to read or write your files.***
- Default permissions on a file are rw-r----- (usually), which means owner has read/write permission, and group has only read permission.
- Default permissions on a directory are rwx------, which means owner has read/write/execute.
- chmod command allows adding or removing from any of the 3 security levels.

  chmod *mode-list file-list*

- *mode-list* has the form *security-level operator permission*.
- security levels are denoted by u for you user, g for group, o for other.
- Operator + adds permission, - removes permission.
- Permissions are denoted by r for readable, w for writable and x for executable.
- The elements of the *mode-list* are separated by commas.
- E.g., to remove read and write permissions from security levels group and other for file xyz, in the long and short forms:

```
chmod g-r,o-r,g-w,o-w xyz
chmod go-rw xyz
```

3.2.4 Input/Output Re-direction

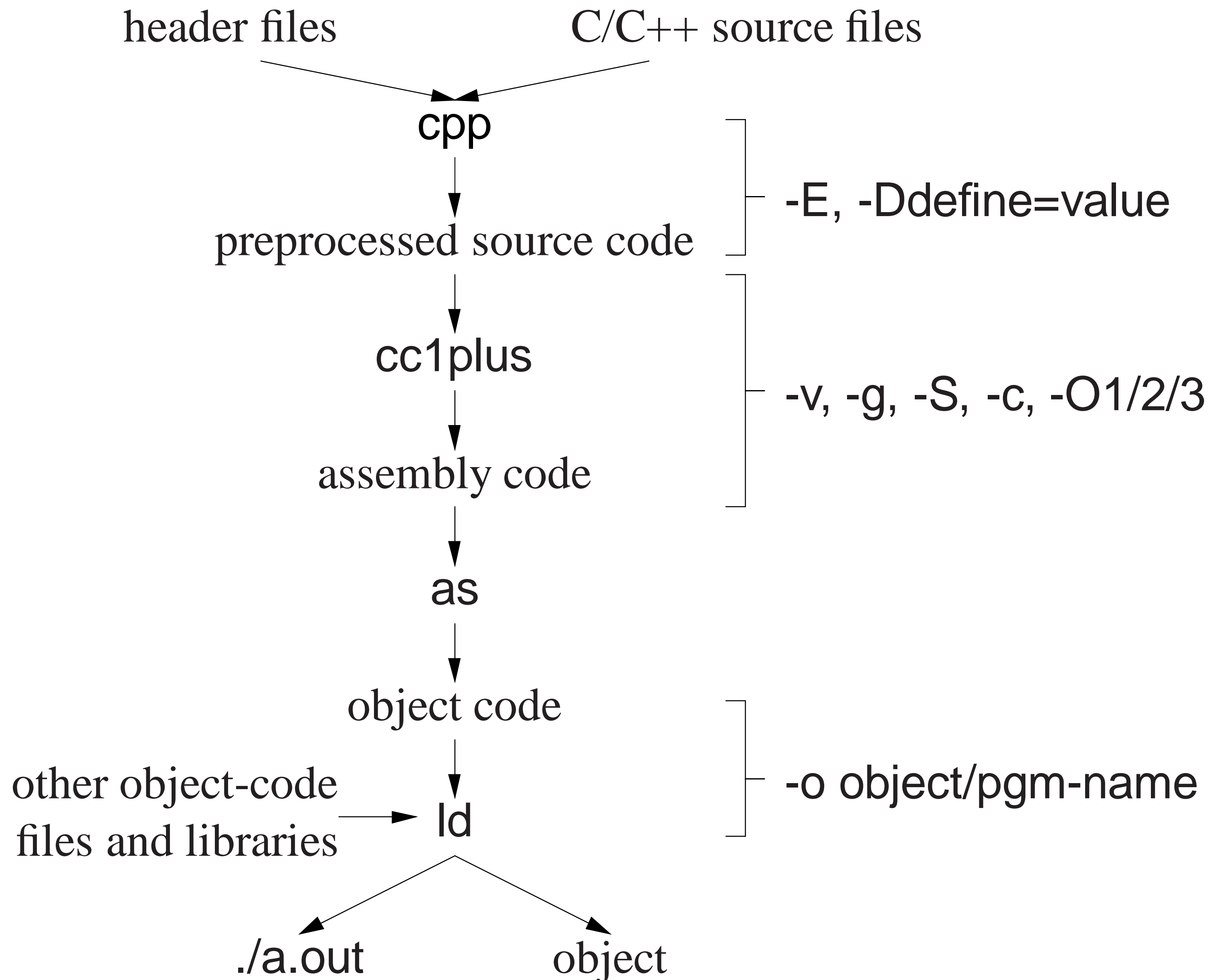
- Input or output of commands can be redirect by the shell to/from sources other than the keyboard (standard in) and screen (standard out/error).
- Shell provides the redirection operators < for redirecting standard input and > for redirecting standard output.
- A command is unaware of the redirection.

- ls -l > xxx put output of ls into file xxx
- more < xxx get input from file xxx and print on standard output
- more < xxx > yyy get input from file xxx and print to file yyy
- Normally, standard error (e.g., error messages) are not redirected because of their importance.
- To redirect *all* output, use the redirection operator >& in the csh and 2>&1 in sh.

```
(csh) % a.out >& xxx   # put standard out and error into file xxx
(sh)  % a.out 2>&1 xxx # put standard out and error into file xxx
```

- Shell pipe operator | takes the output of one command and makes it the input to the next command, without having to create an intermediate file.
- ls -al | more
- Output from the ls command is “piped” into the more command as input.

3.3 Compilation



- **Compilation** is the process of translating a program from human to machine readable form.
- The translation is performed by a tool called a **compiler**.
- Compilation is subdivided into multiple steps, using a number of tools.
- Often a number of options to control the behaviour of each step.
- Option are presented for g++, but other compilers have similar options.
- General format:

  g++ *option-list* *infiles* -o *outfile*

  where *infiles* is C/C++ source (.C) and object files (.o).

3.3.1 Preprocessor

- Preprocessor (cpp) takes a C++ source file, removes comments, and expands **#include**, **#define**, and **#if** directives.
- Options:
  - -E run only the preprocessor step and writes the preprocessor output to standard out.

```
% g++ -E source-files
... much output from the preprocessor
```

– -D define and optionally initialize preprocessor variables from the compilation command:

```
% g++ -DDEBUG=2 -DASSN . . . source-files
```

same as putting the following **#define**s in a program without changing the program:

```
#define DEBUG 2
#define ASSN
```

3.3.2 Compiler (cc1plus)

- Compiler (cc1plus) takes a preprocessed file and converts the C++ language into assembly language for the target machine.
- Options:
  – -v shows each step of the compilation and information about what each step is doing:

```
% g++ -v source-files
... much output from each compilation step
```

Look for these system include files:

```
#include <...> search starts here:
/usr/include/c++/3.3
/usr/include/c++/3.3/i486-linux
/usr/include/c++/3.3/backward
/usr/local/include
/usr/lib/gcc-lib/i486-linux/3.3.5/include
/usr/include
```

where cpp looks for system includes.

– -g add symbol-table information to object file for debugger
– -S stop after translation and write assemble code to file *source-file*.s
– -O1/2/3 optimize translation to different levels, where each level takes more compilation time and possibly more space in executable

3.3.3 Assembler

- Assembler (as) takes an assembly language file and converts it to object code (machine language).

3.3.4 Linker

- Linker (ld) takes the implicit .o file from translated source and any explicit .o files from the command line, and combines them into a new object or executable file.
- Linking options:
  - -o gives the file name where the combined object/ executable is placed.
  - If no name is specified, default name a.out is used.

3.4 Debugging

- Debugging is the process of determining why a program does have an intended behaviour.
- Often debugging is associated with fixing a program after a failure.
- However, debugging can be applied to fixing other kinds of problems, like poor performance.
- Before using debugger tools it is important to understand what you are looking for and if you need them.

3.4.1 Debug Print Statements

- An excellent way to debug a program is to *start* by inserting debug print statements (i.e., as the program is written).
- It takes more time, but the alternative is wasting hours trying to figure out what the program is doing.
- The two aspects of a program that you need to know are: where the program is executing and what values it is calculating.
- Debug print statements show the flow of control through a program and print out intermediate values.
- E.g., every routine should have a debug print statement at the beginning and end, as in:

```
int p( . . . ) {
    // declarations
    cerr << "Enter p " << parameter variables << endl;
    . . .
    cerr << "Exit p " << return value(s) << endl;
    return r;
}
```

- Result is a high-level audit trail of where the program is executing and what values are being passed around.
- Finer resolution requires more debug print statements in important control structures:

```
if ( a > b ) {
    cerr << "a > b" << endl ;                          // debug print
    for ( . . . ) {
        cerr << "x=" << x << ", y=" << y << endl; // debug print
        . . .
    }
} else {
    cerr << "a <= b" << endl;                          // debug print
    . . .
}
```

- By examining the control paths taken and intermediate values generated, it is possible to determine if the program is executing correctly.
- Unfortunately, debug print statements can generate enormous amounts of output.

It is of the highest importance in the art of detection to be able to recognize out of a number of facts which are incidental and which

vital. (Sherlock Holmes, The Reigate Squires)

- Gradually comment out (**#if**) debug statements as parts of the program begin to work to remove clutter from the output, but do not delete them until the program works completely.
- When you go for help, either from your instructor or an advisor, you should have debug print statements in your program.
- In general, debug print statements never appear in the program you hand in for marking.

3.4.2 Assertions

- **Assertions** enforce pre-conditions, post-conditions, and invariants, which document program assumptions:

```
#include <cassert>
int main( int argc, char *argv[] ) {
    assert( argc == 2 );      // must have 1 argument
}
```

- When run without an argument, this produces:

```
% ./a.out
Assertion failed: argc == 2, file test.cc, line 3
Abort (core dumped)
```

- Codify program assumptions with assertions:

```
int main( int argc, char *argv[] ) {
    vector<int> a(10), b(10);
    // read values into a, b
    assert( a.size() == b.size() );    // must be the same size
    for ( i = 0; a[i] == b[i]; i += 1 ) {
        assert( i < a.size() );        // must have an unequal element
    }
    cout << i << endl;
}
```

- Assertions can significantly increase a program's cost.
- Compiling a program with preprocessor variable NDEBUG defined removes all asserts.

```
% g++ -DNDEBUG ... # all asserts removed
```

3.4.3 Errors

- Debug print statements do not prevent errors, they simply aid in finding errors.
- What you do about an error depends on the kind of error.
- Errors fall into two basic categories: syntax and semantic.
- **Syntax error** is in the arrangement of the tokens in the programming language.
- These errors correspond to spelling or punctuation errors when writing in a human language.
- Fixing syntax errors is usually straight forward especially if the compiler generates a meaningful error message.
- Always ***read*** the error message carefully and ***check*** the statement in error.

  You see (Watson), but do not observe. (Sherlock Holmes, A Scandal in Bohemia)

- Difficult syntax errors are:

- Forgetting a closing `"` or `*/`, as the remainder of the program is *swallowed* as part of the character string or comment.
- Missing a { or }, especially if the program is properly indented (editors can help here)

- **Semantic error** is incorrect behaviour or logic in the program.
- These errors correspond to incorrect meaning when writing in a human language.
- Semantic errors are harder to find and fix than syntax errors.
- A semantic or execution error message only tells why the program stopped not what caused the error.
- Must work backwards from the error to determine the cause of the problem.

> *In solving a problem of this sort, the grand thing is to able to reason backwards. This is very useful accomplishment, and a very easy one, but people do not practise it much. In the everyday affairs of life it is more useful to reason forward, and so the other comes to be neglected. (Sherlock Holmes, A Study in Scarlet)*

- E.g., an infinite loop with nothing wrong with the loop; the initialization is wrong.

```
i = 10;
while ( i != 5 ) {
    . . .
    i += 2;
}
```

- In general, when a program stops with a semantic error, the statement that caused the error is not usually the one that must be fixed.
- Difficult semantic errors are:
  - Forgetting to assign a value to a variable before using it in an expression.
  - Using an invalid subscript or pointer value.
- Finally, if a statement appears not to be working properly, but looks correct, check the syntax.

```
if ( a = b ) {
    cerr << "a == b" << endl;
}
```

When you have eliminated the impossible whatever remains, however improbable must be the truth. (Sherlock Holmes, Sign of Four)

3.5 Debugger

- An interactive, symbolic **debugger** effectively allows debug print statements to be added and removed to/from a program dynamically.
- You should not rely solely on a debugger to debug a program.
- You may work on a system without a debugger or the debugger may not work for certain kinds of problems.
- A good programmer uses a combination of debug print statements and a debugger when debugging a complex program.
- A debugger does not debug your program for you, it merely helps in the debugging process.
- Therefore, you must have some idea about what is wrong with a program before starting to look or you will simply waste your time.

3.5.1 GDB

- The two most common UNIX debuggers are: dbx and gdb.
- File test.cc contains:

```
1  void r( int a[] ) {
2      int i = 100000000;
3      a[i] += 1;   // really bad subscript error
4  }
5  int main() {
6      int a[10] = { 0, 1 };
7      r( a );
8  }
```

- Compile program using the -g flag to include names of variables and routines for symbolic debugging:

```
% g++ -g test.cc
```

- Start gdb:

```
% gdb ./a.out
. . . gdb disclaimer
(gdb) ← gdb prompt
```

- Like a shell, gdb uses a command line to accept debugging commands.
- **r**un command begins execution of the program:

```
(gdb) run
Starting program: /u1/cs246/u/pabuhr/teaching/notes/Tools/a.out
Program received signal SIGSEGV, Segmentation fault.
0x000106f8 in r (a=0xffbefa20) at test.cc:3
3         a[i] += 1;     // really bad subscript error
```

  – If there are no errors in a program, running in GDB is the same as running in a shell.
  – If there is an error, control returns to gdb to allow examination.
- **ba**cktrace command prints a stack trace of called **routine activation**s.

```
(gdb) backtrace
#0  0x000106f8 in r (a=0xffbefa08) at test.cc:3
#1  0x00010764 in main () at test.cc:7
```

- **f**rame [n] command moves the **current stack frame** to the nth routine activation on the stack.

```
(gdb) f 0
#0  0x000106f8 in r (a=0xffbefa08) at test.cc:3
3           a[i] += 1;    // really bad subscript error
(gdb) f 1
#1  0x00010764 in main () at test.cc:7
7           r( a );
```

- If n is not present, prints the current frame
- Once moved to a new frame, it becomes the current frame.
- All subsequent commands apply to the current frame.

- **p**rint command prints variables accessible in the current routine, object, or external area.

```
(gdb) f 0
#0  0x000106f8 in r (a=0xffbefa08) at test.cc:3
3           a[i] += 1;    // really bad subscript error
(gdb) print i
$1 = 100000000
```

- $1 is the name of a history variable (like history variables in a shell).
- Name $\$N$ can be used in subsequent commands to access previous values of i.

- Can print any C++ expression:

```
(gdb) print a
$2 = (int *) 0xffbefa20
(gdb) p *a
$3 = 0
(gdb) p a[1]
$4 = 1
(gdb) p a[1]+1
$5 = 2
(gdb) p $3
$6 = 0
```

- **set var**iable command changes the value of a variable in the current routine, object or external area.

```
(gdb) set variable i = 7
(gdb) p i
$7 = 7
(gdb) set var a[0] = 3
(gdb) p a[0]
$8 = 3
(gdb) p $3
$9 = 0
```

- Change the values of variables while debugging to:
  - investigate how the program behaves with new values without recompile and restarting the program,
  - to make local corrections and then continue execution.
- To trace program execution, **breakpoint**s are required.
- **break** command establishes a point in the program where execution suspends and control returns to the debugger.

```
(gdb) break main
Breakpoint 1 at 0x10710: file test.cc, line 6.
(gdb) break test.cc:3
Breakpoint 2 at 0x106d8: file test.cc, line 3.
```

- Set breakpoint using routine name or source-file:line-number.
- If program is not compiled with -g flag, only the location is given.
- Command info breakpoints prints breakpoints currently set.

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x00010710 in main at test.cc:6
2   breakpoint     keep y   0x000106d8 in r(int*) at test.cc:3
```

- Breakpoints numbered consecutively from 1 and can be disabled, enabled or deleted at any time using commands:

(gdb) disable 1	*temporarily disable breakpoint 1*
(gdb) enable 1	*re-enable disabled breakpoint 1*
(gdb) delete 1	*remove breakpoint completely 1*

  (Pretend none of these commands are entered.)

- Run program again to get to the breakpoint:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /u1/cs246/u/pabuhr/teaching/notes/Tools/a.out
Breakpoint 1, main () at test.cc:6
6         int a[10] = { 0, 1 };
(gdb) p a[7]
$10 = 0
```

- Once a breakpoint is reached, execution of the program can be continued in several ways.
- `step [n]` command executes the next `n` lines of the program and stop.

```
(gdb) step
7         r( a );
(gdb) s
r (a=0xffbefa20) at test.cc:2
2         int i = 100000000;
(gdb) s
Breakpoint 2, r (a=0xffbefa20) at test.cc:3
3         a[i] += 1;    // really bad subscript error
(gdb)
Program received signal SIGSEGV, Segmentation fault.
0x000106f8 in r (a=0xffbefa20) at test.cc:3
3         a[i] += 1;    // really bad subscript error
(gdb) s
Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
```

- If n is not present, 1 is assumed.
- **<Return>** ***without a command repeats the last command.***
- If the next line is a routine call, control enters the routine and stops at the first line.

- **next** [n] command, like step, but routine calls are treated as a single statement, so control stops at the statement after the routine call instead

of the first statement of the called routine.

```
(gdb) run
...
Breakpoint 1, main () at test.cc:6
6       int a[10] = { 0, 1 };
(gdb) next
7       r( a );
(gdb) n
Breakpoint 2, r (a=0xffbefa20) at test.cc:3
3       a[i] += 1;   // really bad subscript error
(gdb) n
Program received signal SIGSEGV, Segmentation fault.
0x000106f8 in r (a=0xffbefa20) at test.cc:3
3       a[i] += 1;   // really bad subscript error
```

- **c**ontinue command continues execution until the next breakpoint is reached.

```
(gdb) run
...
Breakpoint 1, main () at test.cc:6
6         int a[10] = { 0, 1 };
(gdb) s
7         r( a );
(gdb) s
r (a=0xffbefa20) at test.cc:2
2         int i = 100000000;
(gdb) s
Breakpoint 2, r (a=0xffbefa20) at test.cc:3
3         a[i] += 1;    // really bad subscript error
(gdb) p i
$4 = 100000000
(gdb) set var i = 3
(gdb) c
Continuing.
Program exited normally.
```

- **fin**ish command finishes execution of the current routine and stops at the statement after the routine call.

```
(gdb) run
...
Breakpoint 1, main () at test.cc:6
6         int a[10] = { 0, 1 };
(gdb) c
Continuing.
Breakpoint 2, r (a=0xffbefa20) at test.cc:3
3         a[i] += 1;    // really bad subscript error
(gdb) set var i = 3
(gdb) fin
Run till exit from #0  r (a=0xffbefa20) at test.cc:3
main () at test.cc:8
8    }
(gdb) c
Continuing.
Program exited normally.
```

  – Print the value returned by the finished routine, if any.

- During debugging, it is useful to print variables each time the program stops at a breakpoint.
- Normally, requires typing a print commands each time the program stop.

- **displ**ay command is like the print command, with the addition of printing each time the program stops.

```
(gdb) run
...
Breakpoint 1, main () at test.cc:6
6         int a[10] = { 0, 1 };
(gdb) display a[0]
1: a[0] = 67568
(gdb) s
7         r( a );
1: a[0] = 0
(gdb) s
r (a=0xffbefa20) at test.cc:2
2         int i = 100000000;
```

  - Each displayed variable is numbered, in this case, a is numbered 1.
  - Use number to stop displaying a variable via undisplay *n* command.
  - If a variable goes out of scope, the display stops printing.
- **l**ist command lists source code.

```
(gdb) list
2       int i = 100000000;
3       a[i] += 1;
4   }
5   int main() {
6       int a[10];
7       r( a );
8   }
(gdb) list 3
1   void r( int a[ ] ) {
2       int i = 100000000;
3       a[i] += 1;
4   }
5   int main() {
6       int a[10];
7       r( a );
8   }
```

- with no argument, list code around current execution location
- with argument line number, list code around line number

- **q**uit command terminate gdb.

```
(gdb) run
...
Breakpoint 1, main () at test.cc:6
6         int a[10] = { 0, 1 };
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

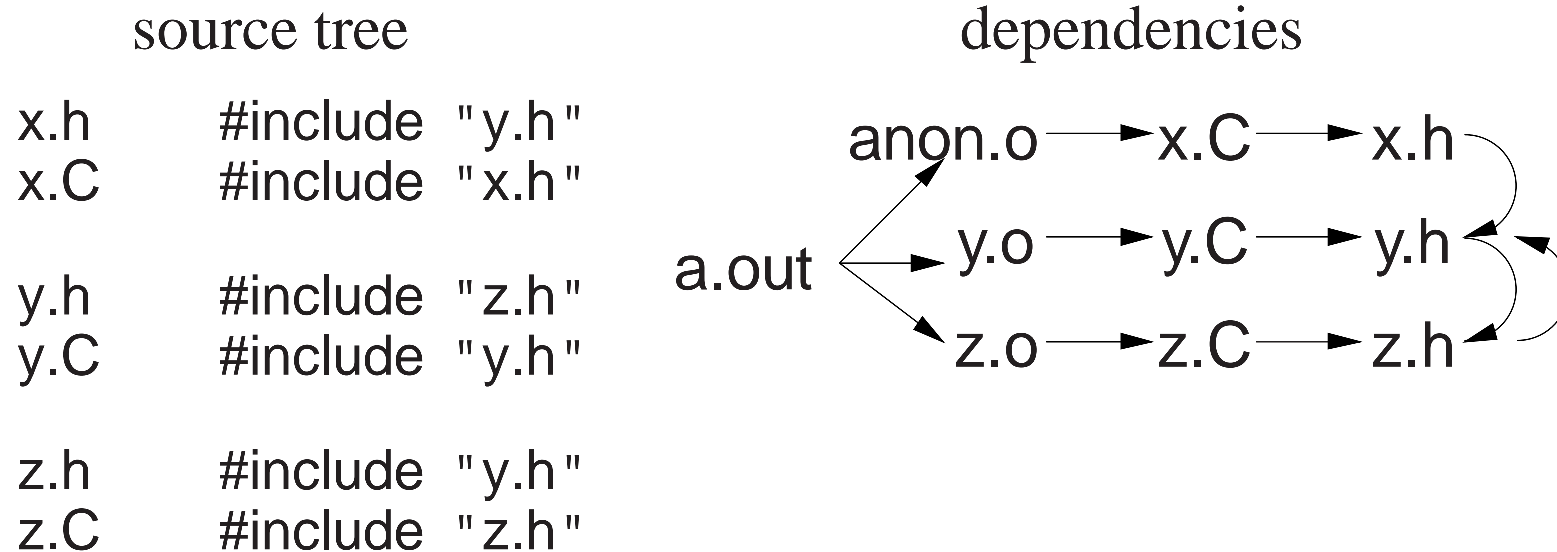
3.6 Compiling Complex Programs

- Separate compilation has an advantage and disadvantage.
- Advantage: saves significant amounts of computer and people time by recompiling only the portions of a program that are changed.
- In theory, if an expression is changed, only that expression needs to be recompiled.
- In practice, the unit of compilation is much coarser: the **translation unit** (TU), which is a file in C/C++.
- In theory, each line of code (expression) could be put in a separate file, but impractical (and doesn't work).
- So a TU should not be too big and not be too small.

- Disadvantage: TUs must depend on each other because a program shares many forms of information, especially types.
- Not a problem when all the code is in a single TU (except for DBU).
- As a program grows, the number of TUs grow, so does the dependencies among TUs.
- Now, when one TU is changed, it may require other TUs to change that depend on some or all of the shared information.
- For a large numbers of TUs, the dependencies turn into a nightmare with respect to recompiled.

3.6.1 Dependences

- Dependences in C/C++ normally occur as follows:
  - executable depends on .o files
  - .o files depend on .C files
  - .C files depend on .h files



- The hierarchical **source tree** is compiled as follows:

```
% g++ -c z.C          # generates z.o
% g++ -c y.C          # generates y.o
% g++ x.C y.o z.o     # generates a.out
# alternative
% g++ -c x.C          # generates x.o
% g++ x.o y.o z.o     # generates a.out
```

- If a change is made to y.h, which files need to be recompiled?
- Does ***any*** change to y.h require these recompilations?
- There is no mechanism to know the kind of change made within a file,

e.g., changing a comment, type, variable.

- So dependence is coarse grain, based on ***any*** change to a file.
- One way to denote file changes is with **time stamp**s.
- UNIX stores in the directory the time a subfile was last changed, with second precision.
- Establishing dependencies means establishing a temporal ordering in the dependence graph so the root has the newest (or equal) time and the leafs the oldest (or equal) time.

3.6.2 Make

- **make** is a UNIX command that takes a dependence graph and uses file change-times to trigger rules that bring the dependence graph up to date.
- A make dependence graph expresses a relationship between a product and a set of sources.
- Make does not express a relationship among sources, one that exists at the source code level and is important.
- E.g., source x.C depends on source x.h but x.C is not a product of x.h like x.o is a product of x.C and x.h.

- The two most common UNIX makes are: make and gmake (on Linux, make is gmake).
- Like shells, there is minimal syntax and semantics for make, which is mostly portable across systems.
- The most common non-portable features are specifying dependencies and implicit rules.
- A basic makefile consists of string variables with initialization and a list of targets and rules.
- This file can have any name, but make implicitly looks for a file called makefile or Makefile if no file is specified.
- Each target has a list of dependencies, and possibly a set of commands specifying how to re-establish the target.

```
variable = value
target : dependency1 dependency2 ...
        command1
        command2
        ...
```

- make is invoked with a target, which is a subnode or root of a dependence hierarchy.

- make builds the dependency graph and decorates the edges with time stamps for the specified files.
- If any of the dependency files (leafs) are newer than the target file (root), or if the target file does not exist, the commands are executed by the shell to update the target (generate a new product).
- Makefile for previous dependencies:

```
a.out : x.o y.o z.o
	g++ x.o y.o z.o -o a.out

x.o : x.C x.h y.h z.h
	g++ -g -Wall -c x.C
y.o : y.C y.h z.h
	g++ -g -Wall -c y.C
z.o : z.C z.h y.h
	g++ -g -Wall -c z.C
```

- Update dependencies by:

```
% gmake -n -f Makefile a.out
g++ -g -Wall -c x.C
g++ -g -Wall -c y.C
g++ -g -Wall -c z.C
g++ x.o y.o z.o -o a.out
```

- -n only checks the dependencies and shows rules to be triggered (leave off to trigger rules)
- -f Makefile is the dependency file (leave off if named [M|m]akefile)
- a.out target name to be updated (leave off if first target)

- Eliminate duplication using variables:

```
CXX = g++                        # variables
CXXFLAGS = -g -Wall -c
OBJECTS = x.o y.o z.o
EXEC = a.out

${EXEC} : ${OBJECTS}
	${CXX} ${OBJECTS} -o ${EXEC}

x.o : x.C x.h y.h z.h
	${CXX} ${CXXFLAGS} x.C
y.o : y.C y.h z.h
	${CXX} ${CXXFLAGS} y.C
z.o : z.C z.h y.h
	${CXX} ${CXXFLAGS} z.C
```

- Eliminate common rules:

```
CXX = g++                        # variables and initialization
CXXFLAGS = -g -Wall              # can remove -c
OBJECTS = x.o y.o z.o
EXEC = a.out

${EXEC} : ${OBJECTS}
    ${CXX} ${OBJECTS} -o ${EXEC}

x.o : x.C x.h y.h z.h
y.o : y.C y.h z.h
z.o : z.C z.h y.h

clean :
    rm -rf ${OBJECTS} ${EXEC}
```

- gmake *knows* how to construct simple rules when files have specific suffixes and when special variable names are used.
- Target clean removes files that can be rebuilt to save space.

```
gmake clean
```

- Eliminate dependencies:

```
CXX = g++                       # variables and initialization
CXXFLAGS = -g -Wall -MMD        # build dependency graph in .d files
OBJECTS = x.o y.o z.o
DEPENDS = ${OBJECTS:.o=.d} # substitute ".o" with ".d"
EXEC = a.out

${EXEC} : ${OBJECTS}
	${CXX} ${OBJECTS} -o ${EXEC}

clean :
	rm -rf ${DEPENDS} ${OBJECTS} ${EXEC}

-include ${DEPENDS}
```

– g++ flag -MMD generates a dependency graph for only user source file.

```
x.d
    x.o: x.C x.h y.h z.h
y.d
    y.o: y.C y.h z.h
z.d
    z.o: z.C z.h y.h
```

– g++ flag -MD generates a dependency graph for user and system

source file.

– -**include** reads the .d files and runs dependencies again.

3.7 Source Code Management

- UNIX files are used for TUs.
- These files only support the ***current*** version of the program.
- As a program develops/matures, it changes in many ways.
- UNIX files do not support this temporal notion of a program, i.e., history of program over time.
- A history allows access to older versions of the program, supporting operations like backing out of changes because of design changes or problems.
- Another issue is sharing program files among multiple developers each making independent changes.
- Current sharing allows damaging the contents of the files for simultaneous writes.
- Approaches:

- Make copies of some or all of the project files before making changes.
  Wastes storage for unchanged files and burden of managing copied files.
- Share files using group file permissions.
  Simultaneous access is unsafe and developers cannot test changes in isolation.
- Giving each developer a separate copy of the code base.
  Merging in changes from different developers is tricky and time consuming.

• To solve these problems, a **source control system** is used to manage cooperative work.

3.7.1 CVS

• **Concurrent Versions System** (CVS) is a source control system with the following features:
  - Master copy of all project files is kept in a **repository**.
  - Multiple versions of files are automatically stored in the repository.
  - Developers can check out a complete copy of the project.

– Helpful integrated back into the repository using text merging. Programmers still have to deal with conflicts.

3.7.2 Repository

- A shared repository must be created at some accessible location in the file system:

```
% cd cs246
% mkdir cvsroot                  # make repository directory
% chgrp cs246_75 cvsroot # set group on directory
% chmod g+rwx cvsroot       # allow group members access
% mkdir cvsroot/CVSROOT # required (administration)
% mkdir cvsroot/assn6        # specific project
```

- Must have administration CVSROOT directory at the root of repository.
- Other directories at root represent projects (can have any name).
- Group members must add this line to their shell startup file:

```
sh:
    % set CVSROOT /u/userid/cs246/cvsroot
    % export CVSROOT
csh:
    % setenv CVSROOT /u/userid/cs246/cvsroot
```

3.7.3 Checking Out

- checkout command creates a working copy of the project:

```
% cvs checkout assn6          # checkout project
cvs checkout assn6
cvs checkout: Updating assn6
U assn6/y.C
U assn6/y.h
U assn6/z.C
U assn6/z.h
% cd assn6                    # move into project directory
```

- Copies the entire project directory to the current directory.
- A checked out copy can be modify in any way without other developers seeing these changes until committed.

3.7.4 Adding/Editting/Removing

- add command tell CVS to add new files (in current directory) to the repository:

```
% ...                              # add files x.h, x.C
% cvs add x.*
cvs add: scheduling file ‘x.C’ for addition
cvs add: scheduling file ‘x.h’ for addition
cvs add: use 'cvs commit' to add these files permanently
```

- Schedules files for addition, which occurs on cvs commit.
- ***Forgetting cvs add is a common mistake.***
- Edited files (in current directory) do not require any CVS command:

```
% ...                              # edit files y.h y.C
```

- Implicitly schedules files for update, which occurs on cvs commit.
- remove command tell CVS to remove existing files from the repository:

```
% ...                              # remove files z.h z.C
% cvs remove z.h z.C
cvs remove: scheduling 'z.h' for removal
cvs remove: scheduling 'z.C' for removal
cvs remove: use 'cvs commit' to remove these files permanently
```

- Schedules files for removal, which occurs on cvs commit.
- In fact, any removed file can always be retrieved from old versions.

3.7.5 Checking In

- commit updates the repository with the changes made in checkout directory.

```
% cvs commit -m "... description of changes ..."
cvs commit: Examining .
RCS file: /u/userid/cs246/cvsroot/assn6/x.C,v
done
Checking in x.C;
/u/userid/cs246/cvsroot/assn6/x.C,v  <--  x.C
initial revision: 1.1
done
RCS file: /u/userid/cs246/cvsroot/assn6/x.h,v
done
Checking in x.h;
/u/userid/cs246/cvsroot/assn6/x.h,v  <--  x.h
initial revision: 1.1
done
Checking in y.C;
/u/userid/cs246/cvsroot/assn6/y.C,v  <--  y.C
new revision: 1.2; previous revision: 1.1
done
Checking in y.h;
/u/userid/cs246/cvsroot/assn6/y.h,v  <--  y.h
new revision: 1.2; previous revision: 1.1
done
```

```
Removing z.C;
/u/userid/cs246/cvsroot/assn6/z.C,v  <--  z.C
new revision: delete; previous revision: 1.1
done
Removing z.h;
/u/userid/cs246/cvsroot/assn6/z.h,v  <--  z.h
new revision: delete; previous revision: 1.1
done
```

- Provides a record of the changes that have been made, which are available using cvs log.
- If -m flag not used, cvs prompts for a change description using an editor.
- Always make sure that your code compiles and runs before committing.
- It is unfair to pollute the source base with bugs.

3.7.6 Update

- Cannot commit changes if other developers have checked in changes during a checkout.
- Changes must now be merged and then committed.
- update command merges changes into repository.

- Causes merged file in current directory to be updated.
- Merge algorithm is generally very good if changes do not overlap.
- Overlapping changes result in a conflict, which must be resolved manually.

```
% cvs commit
cvs commit: Examining .
cvs commit: Up-to-date check failed for ‘Makefile’
cvs [commit aborted]: correct above errors first!
% cvs update
cvs update: Updating .
RCS file: /u/userid/cvsroot/assn6/Makefile,v
retrieving revision 1.2
retrieving revision 1.3
Merging differences between 1.2 and 1.3 into Makefile
```

- Conflict is marked in Makefile:

```
CXX = g++                        # variables and initialization
<<<<<<< Makefile
CXXFLAGS = -g -MMD
=======
CXXFLAGS = -g -Wall
>>>>>>> 1.3
```

3.7.7 Versions

- Each time a file is committed, it receives a new version number.
- Version number is displayed during commit, and at other times.
- cvs status prints version information.
- Old versions are accessible using:

  ```
  cvs update -p -r 1.2 Makefile   # -p prints to standard output
  ```

  which prints version 1.2 of Makefile to standard output.
- Differences between versions can be generated:

  ```
  cvs diff -r 1.2 -r 1.1 Makefile
  ```

  which shows the differences between version 1.2 and version 1.1.

3.7.8 Tagging

- Version numbers are nondescript and often too low level (i.e., little changes here and there).
- It is possible to give a meaningful, symbolic name to a version, often at a stable point or before big changes.
- tag command adds a synbolic name to the current version of every file checked out:

```
cvs tag debug1          # name current version "debug1"
```

- Use symbolic name like version number:

```
cvs update -p -r debug1
```

- To compare named versions:

```
cvs diff -r debug1 -r debug2
```

4 Software Engineering

- **Software Engineering** (SE) is the social process of designing, writing, and maintaining computer programs.
- SE attempts to find good ways to help people understand and develop software.
- However, what is good for people is not necessarily good for the computer.
- Many SE approaches are counter productive in the development of high-performance software.
- E.g.: The computer does not execute the documentation!
- Documentation is unnecessary to the computer, and significant amounts of time are spent building it so can be ignored (program comments).
- Remember, the ***truth*** is always in the code.
- However, without documentation, developers have difficulty designing and understanding software.
- E.g., designing by anthropomorphizing the computer is seldom a good approach (desktops/graphical interfaces).

© Peter A. Buhr

- What works for people does not necessarily work for the computer.
- Software tools spend significant amounts of time undoing SE design and coding approaches to generate efficient programs.
- It is important to know these differences to achieve a balance between programs that are good for people and good for the computer.

4.1 Software Crisis

- Large software systems ($>$ 100,000 lines of code) require many people and months to develop.
- These projects normally emerge late, over budget, and do not work well.
- Today, hardware costs are nil, and manpower cost is great.
- While commodity software is available, someone still has to write it.
- Since people produce software $\Rightarrow$ software cost is great.
- Coupled with a shortage of software personnel $\Rightarrow$ problems.
- Unfortunately, software is complex and precise, which requires time and patience.

4.2 Software Development

- Techniques for program development for small, medium, and large systems.
- Objectives:
  - to plan and schedule software projects
  - to produce reliable, flexible, efficient programs
  - to produce programs that are easily maintained
  - to reduce the cost of software
  - to reduce program failure
- E.g., a typical software project:
  - estimate 12 months of work
  - hire 3 people for 4 months
  - make up milestones for the end of each month
- However, first milestone is reached after 2 months instead of 1.
- To finish on time, hire 2 more people, but:
  - new people require training
  - work must be redivided

This takes at least 1 month.

- Now 2 months behind with 9 months of work to be done in 1 month by 5 people.
- To get the project done:
  - must reschedule
  - trim project goals
- In general, adding manpower to a late software project makes it later.
- Illustrates the need for a methodology to aid in the development of software projects.

4.2.1 Programming Methodology

- System Analysis (next year)
  - Study the problem, the existing systems, the requirements, the feasibility.
  - Analysis is a set of requirements describing the system inputs, outputs, processing, and constraints.
- System Design

  - Breakdown of requirements into modules, with their relationships and data flows.
  - Results in a description of the various modules required, and the data interrelating these.
- Implementation
  - writing the program
- Testing & Debugging
  - get it working
- Operation & Review
  - was it what the customer wanted and worth the effort?
- Feedback
  - If possible, go back to the above steps and augment the project as needed.

4.2.2 System Design

- In designing a system of any size it must be modularized.

- **Modularization** is the division of the system into smaller parts on some systematic basis.
- Modularization is necessary to:
  - make it easier to design and implement
  - make it easier to read
  - make it easier to maintain and modify
  - abstract the data structures
  - abstract the algorithms
- Two basic strategies exist to systematically modularize a system:
  - top-down or functional decomposition
  - bottom-up
- Both techniques have much in common and so examine only one.

4.2.3 Top-Down Design

- Start at highest level of abstraction and break down problem into cohesive units.
- Then refine each unit further generating more detail at each division.

- This recursive process is called **stepwise refinement**.
- Each subunit is divided until a level is reached where the parts are comprehensible, and can be coded directly.
- Unit are independent of a programming language, but ultimately must be mapped into constructs like:
  - generics (templates)
  - modules
  - classes
  - routines
- Details look at data and control flow within and among units.
- Implementation programming language is often chosen only after the system analysis/design process.

4.2.4 Factoring

- **Factoring** is the modularization of code in one module into multiple modules.
- Stop factoring when:

- cannot find a well defined function to factor out
- interface to the module would be as complicated as the module itself

- Factoring is done to:
  - reduce module size : $\approx$ 30-60 lines of code, i.e., 1-2 screens with documentation
  - make system easier to understand
  - eliminate duplicate code
  - localize modifications
- Avoid having the same function performed in more than one module (create useful general purpose modules)
- Separate work from management:
  - Higher-level modules only make decisions (management) and call other routines to do the work.
  - Lower-level modules become increasingly detailed and specific, performing finer grain operations.
- In general:
  - do not worry about little inefficiencies unless the code is executed a LARGE number of times

– put thought into readability of program
– avoid high levels of nesting (3-5 levels is fine)

4.3 Structured Programming

- **Structured programming** is about managing (restricting) control flow using a fixed set of well defined control structures.
- A small set of control structures used with a particular programming style make programs easier to write and understand, as well as maintain.
- All programmers adopt this approach so there is universal (common) approach to managing control flow (e.g., like traffic rules).
- Developed during the 1970's to overcome the indiscriminant use of the GOTO statement.
- GOTO leads to convoluted logic in programs (i.e., does NOT support a methodical thought process).
- Arbitrary transfer of control results in programs that are difficult to understand and maintain.
- Fixed control structures fix the points where the flow of control can be altered, and therefore, are easy to follow.

- There are 3 levels of structured programming:

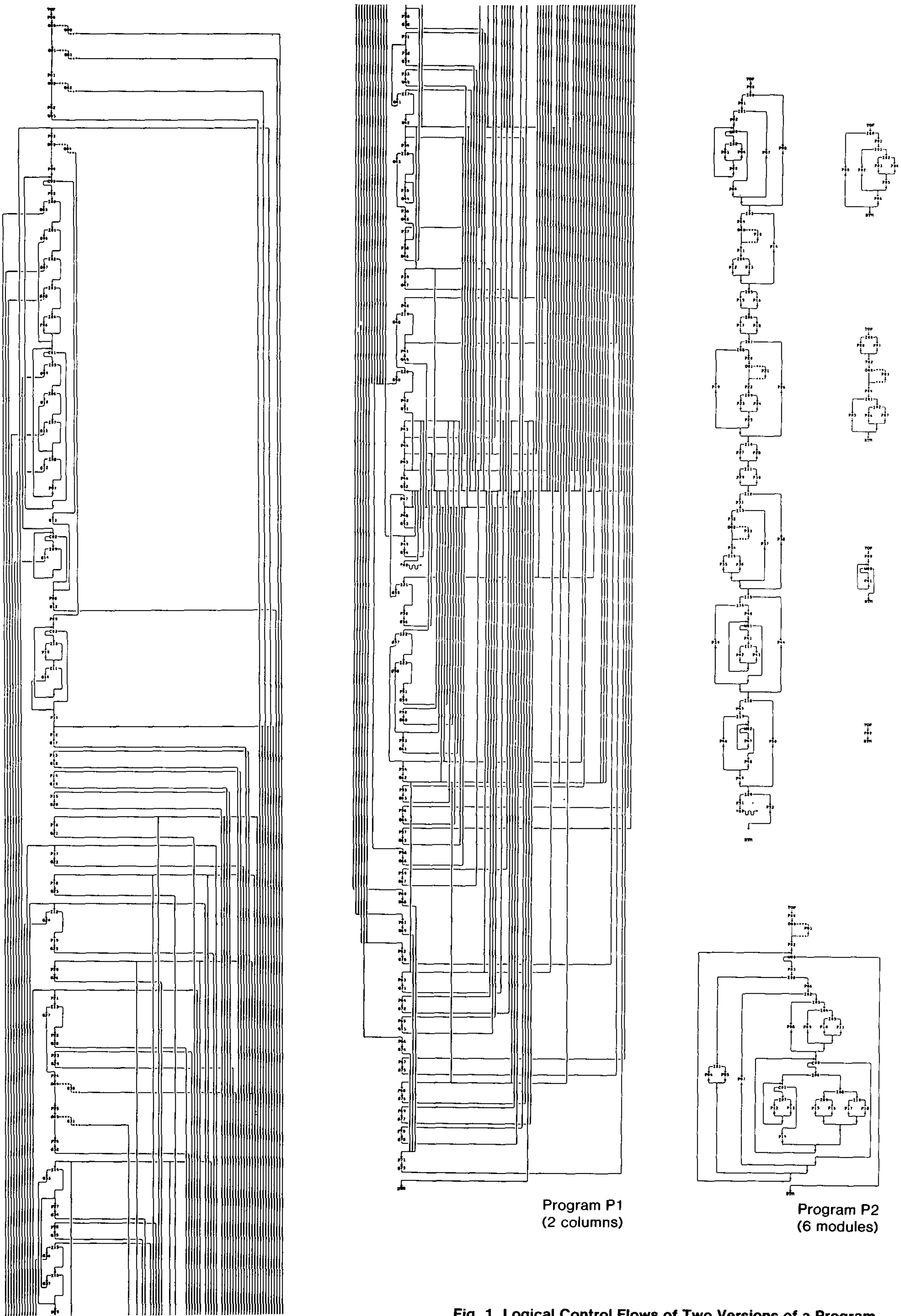


Fig. 1. Logical Control Flows of Two Versions of a Program.

classical

– sequence: series of statements
– if-then-else: conditional structure for making decisions
– while: structure for loops with test at top

Can write any program (actually only need while).

extended

– classical control structures
– repeat/do-while: structure for loops with test at bottom
– case/switch: conditional structure for making decisions

modified

– extended control structures
– one or more exits from arbitrary points in a loop
– exits from multiple nested control structures
– exits from multiple nested routine calls

Eliminates the need for flag variables.

4.3.1 Multi-Exit Loops

- A **multi-exit loop** (or mid-test loop) is a loop with one or more exit locations occurring *within* the body of the loop.

```
loop
    . . .
  exit when i >= 10;
    . . .
end loop
```

- Or allow multiple exit conditions:

```
loop
    . . .
  exit when i >= 10;
    . . .
  exit when j >= 10;
    . . .
end loop
```

- Eliminates priming (copied) code necessary with **while**:

```
read( input, d );               loop
while ! eof( input ) do             read( input, d );
    . . .                         exit when eof( input );
    read( input, d );               . . .
end while                       end loop
```

- C/C++ idioms for this situation are:

C	C++
while ((d = getc(stdin)) != EOF)	**while** (cin >> d)

- Results in expression side-effects and precludes analysis of d without code duplication.
- The loop exit is always outdented (like **else**) or clearly commented (or both) so it can be found without having to search the entire loop body.
- A multi-exit loop can be written in C/C++ in the following ways:

```
for ( ;; ) {                while ( true ) {            do {
    . . .                       . . .                       . . .
  if ( i >= 10 ) break;       if ( i >= 10 ) break;       if ( i >= 10 ) break;
    . . .                       . . .                       . . .
  if ( j >= 10 ) break;       if ( j >= 10 ) break;       if ( j >= 10 ) break;
    . . .                       . . .                       . . .
}                           }                           } while( true );
```

- The **for** version is more general as it can be easily modified to have a loop index or a while condition.

```
for ( int i = 0; i < 10; i += 1 ) { // loop index
for ( ; x < y; ) { // while condition
```

- In general, the programming language and code-typing style should allow insertion of new code without having to change existing code.
- E.g., write linear search such that:
  - no invalid subscript for unsuccessful search
  - index points at the location of the key for successful search.
- Some languages have a special **short-circuit** version of logical *and* and *or* with minimum evaluation.

```
for ( i = 0; i < size && key != list[i]; i += 1 ){};
        // rewrite: if ( i < size ) if ( key != list[i] )
if ( i < size ) { . . .         // found
} else { . . .                  // not found
}
```

- Short-circuit logical operators are control structures in the middle of an expression because `e1 && e2` $\not\equiv$ `&&( e1, e2 )` (unless lazy evaluation).
- Alternatively, using multi-exit loop.

```
for ( i = 0; ; i += 1 ) { // or  for ( i = 0; i < size; i += 1 )
  if ( i >= size ) break;
  if ( key == list[i] ) break;
}
if ( i < size ) { . . .        // found
} else { . . .                 // not found
}
```

- The extra test after the loop can be eliminated by introducing it into the loop body.

```
for ( i = 0; ; i += 1 ) {
  if ( i >= size ) { . . .              // not found
        break;
     } // exit
  if ( key == list[i] ) { . . .         // found
        break;
     } // exit
} // for
```

- E.g., an element is looked up in a list of items, if it is not in the list, it is added to the end of the list, if it exists in the list its associated list counter is incremented.

```
for ( i = 0; ; i += 1 ) {
  if ( i >= size ) {
        list[size].count = 1;
        list[size].data = key;
        size += 1;
    break;
    } // exit
  if ( key == list[i].data ) {
        list[i].count += 1;
    break;
    } // exit
} // for
```

4.3.2 Static Multi-Level Exit

- **Static multi-level exit** exits multiple control structures where exit points are *known* at compile time.
- Labelled exit (break) (or continue) often provides this capability:

```
L1: {
    ... declarations ...
    L2: switch ( ... ) {
      case ...:
        L3: for ( ... ) {
              ... break L1; ... // exit compound statement
              ... break L2; ... // exit switch
              ... break L3; ... // exit loop
        }
        break;
        ... // more case clauses
    }
    ...
}
```

- Labelled **break** transfers control out of the control structure with the corresponding label, terminating any block that it passes through.
- Commonly used with nested loops:

```
A: for ( ;; ) {    // while ( flag1 && . . . )
    B: for ( ;; ) {    // while ( flag2 && . . . )
        C: for ( ;; ) {    // while ( flag3 && . . . )
            . . .
            if ( . . . ) break A; // exit 3 levels
            . . .
            if ( . . . ) break B; // exit 2 levels
            . . .
            if ( . . . ) break C; // or break, exit 1 level
            . . .
        }
    }
}
```

A:, B: and C: are labels.

- Labelled **break** transfers control out of the **for** with the corresponding label, terminating any block that it passes through.
- Eliminates flag variables, which are the variable equivalent to a goto.
- Normal and labelled **break** are a **goto** with restrictions:
  - Cannot be used to create a loop (i.e., cause a backward branch in the program); hence, all situations that result in repeated execution of

statements in a program are clearly delineated.

– Cannot be used to branch *into* a control structure.

- The simple case (exit 1 level) of multi-level exit is a multi-exit loop.
- Why is good practice to label all exits?
- A static multi-level exits is written in C/C++ in the following way:

```
for ( ;; ) {
    for ( ;; ) {
        for ( ;; ) {
            . . .
  if ( . . . ) goto A;
            . . .
      if ( . . . ) goto B;
            . . .
          if ( . . . ) goto C;       // or break
            . . .
        } C: ;
    } B: ;
} A: ;
```

- **return** statements in a routine can generate multi-exit loop and multi-level exit.

- Static multi-level exits appear infrequently, but are extremely concise and execution-time efficient.

4.3.3 Dynamic Multi-Level Exit

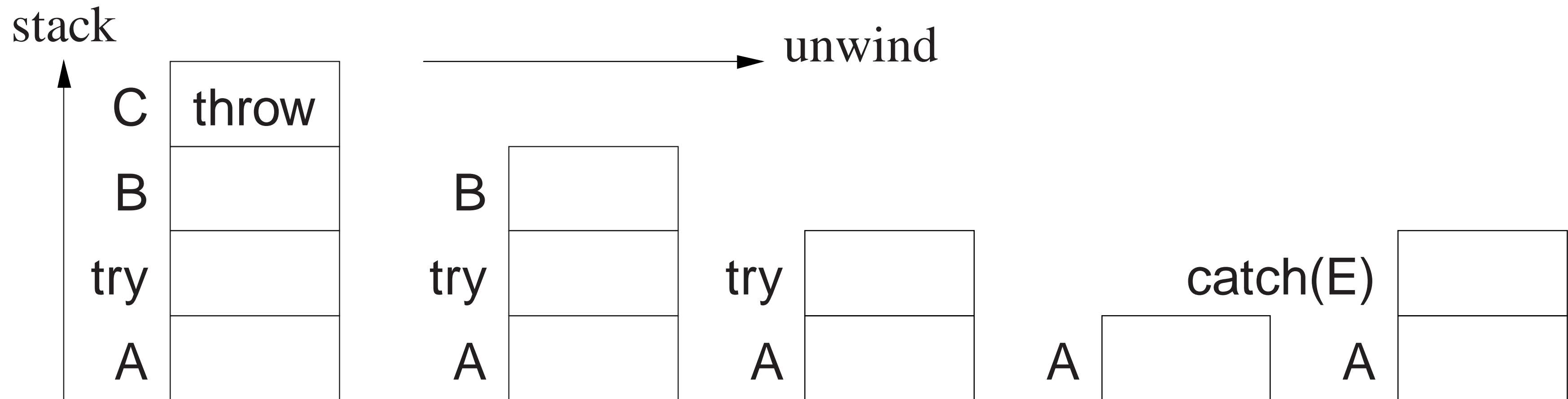
- Basic and advanced control structures allow virtually any control flow ***within*** a routine.
- However, control flow *among* routines is rigidly controlled by the call/return mechanism.
  - given A calls B calls C, it is impossible to transfer directly from C back to A, terminating B in the transfer.
- Dynamic multi-level exit extend call/return semantics to transfer in the ***reverse*** direction to normal routine calls.
- This complex control-flow among routines is often called **exception handling**.
- Exception handling is more than error handling.
- An **exceptional event** is an event that is (usually) known to exist but which is *ancillary* to an algorithm.
  - an exceptional event usually occurs with low frequency

 – e.g., division by zero, I/O failure, end of file, pop empty stack

- Exceptions in Java/C++ provide a *limited* mechanism to transfer to blocks on the call stack:

```
struct E {};       // label
void C(...) throw(E) {
    ... throw E(); // raise (goto)
    // control never returns here
}
void B(...) throw(E) {
    ... C() ...
}
void A() {
    try {
        ... B(...); ...
    } catch( E ) {...}  // handler 1

    try {
        ... B(...); ...
    } catch( E ) {...}  // handler 2
}
```



- Stack is unwound from the raise to the handler.
- Destructors are invoked for objects contained in unwound blocks.
- Handler is called like a routine from A.
- Handler continues after **try** block not after **throw**.
- Do not know statically where **throw** E() is caught (handler1 or handler2).

4.4 System Design

- **System design** involves modelling a complex system in an abstract way to provide a specific description of how the system works.
- The design grows from nothing to become a model of sufficient detail to be transformed into a functioning system.

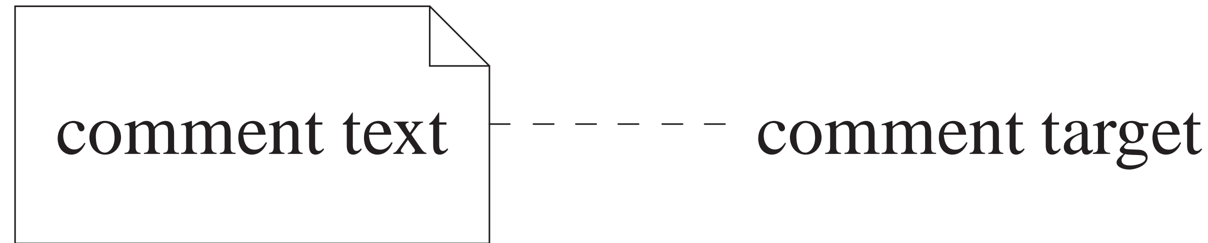
- After which, the design provides high-level documentation of the system, for understanding (education) and for making changes in a systematic manner.
- Top-down successive refinement (TDSR) is a foundational mechanism used in all system design.
- System modelling has multiple viewpoints:
  - **class model** : describes static kinds and structure of system objects
  - **state model** : describes dynamic (temporal) behaviour of system objects
  - **interaction model** : describes the kinds of interactions among objects
- Multiple design tools (past and present) for supporting system design, most are graphical and all are programming language independent:
  - flowcharts (1920-1970)
  - pseudo-code
  - Warnier-Orr Diagrams
  - Hierarchy Input Process Output (HIPO)
  - UML
- Design tools can be used in various ways:

- to *sketch* out high-level design or complex parts of a system,
- to *blueprint* the entire system abstractly with high accuracy,
- to *generate* interfaces directly.

- Key advantage of design tool is the generic, abstract model of the system, which can be transformed into any format.
- Key disadvantage is the design tool is seldom linked to the implementation mechanism, so the two often differ **(implementation = truth)**.
- As with design strategies, design tools have much in common and so only one is studied.

4.4.1 UML

- **Unified Modelling Language** (UML) is a graphical notation for describing and designing software systems, with emphasis on the object-oriented style.
- UML can handle class, state and interaction modelling. (focus on class modelling)
- Note/comment



- **Class diagram** collection of class templates and associated relationships.
- Class specifies a template for objects : name, attributes, operations.
- **attribute** : value description (field)
  [visibility] name [":" [type] ["[" multiplicity "]"] ["=" default]
  ["{" property "}"]]
  - visibility : access of attribute information by other classes
    $+ \Rightarrow$ public, $- \Rightarrow$ private
  - name : required identifier for attribute (like field name in structure)
  - type : restriction on kind of objects associated with attribute
  - multiplicity : restriction on number of objects associated with attribute
    - range 0..($N|*$), from 0 to N or unlimited, N short for $N..N$, $*$ short for 0..$*$
  - default : value of newly created object

  - property : additional aspects of attribute, e.g., { readonly }
  - class attributes (static) are underlined

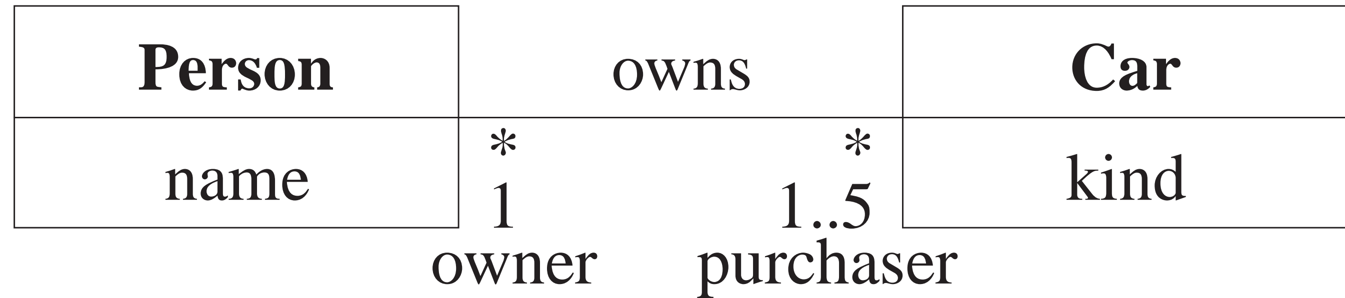
- **operation** : action changing or returning object state (method)
  [visibility] name ["(" { direction parameter-*attribute* }* ")"]
  [":" return-*attribute*] ["{" property "}"]
  - name : required identifier for operation (like method name in structure)
  - visibility : access of attribute information by other classes, $+ \Rightarrow$ public, $- \Rightarrow$ private
  - direction : $\left[\right.$ in $\mid$ out $\mid$ inout $\left.\right]$ indicates direction of parameter data flow
  - parameters : input/output values for operation
  - return-type : output from operation
  - property : additional aspects of operation, e.g., { readonly }

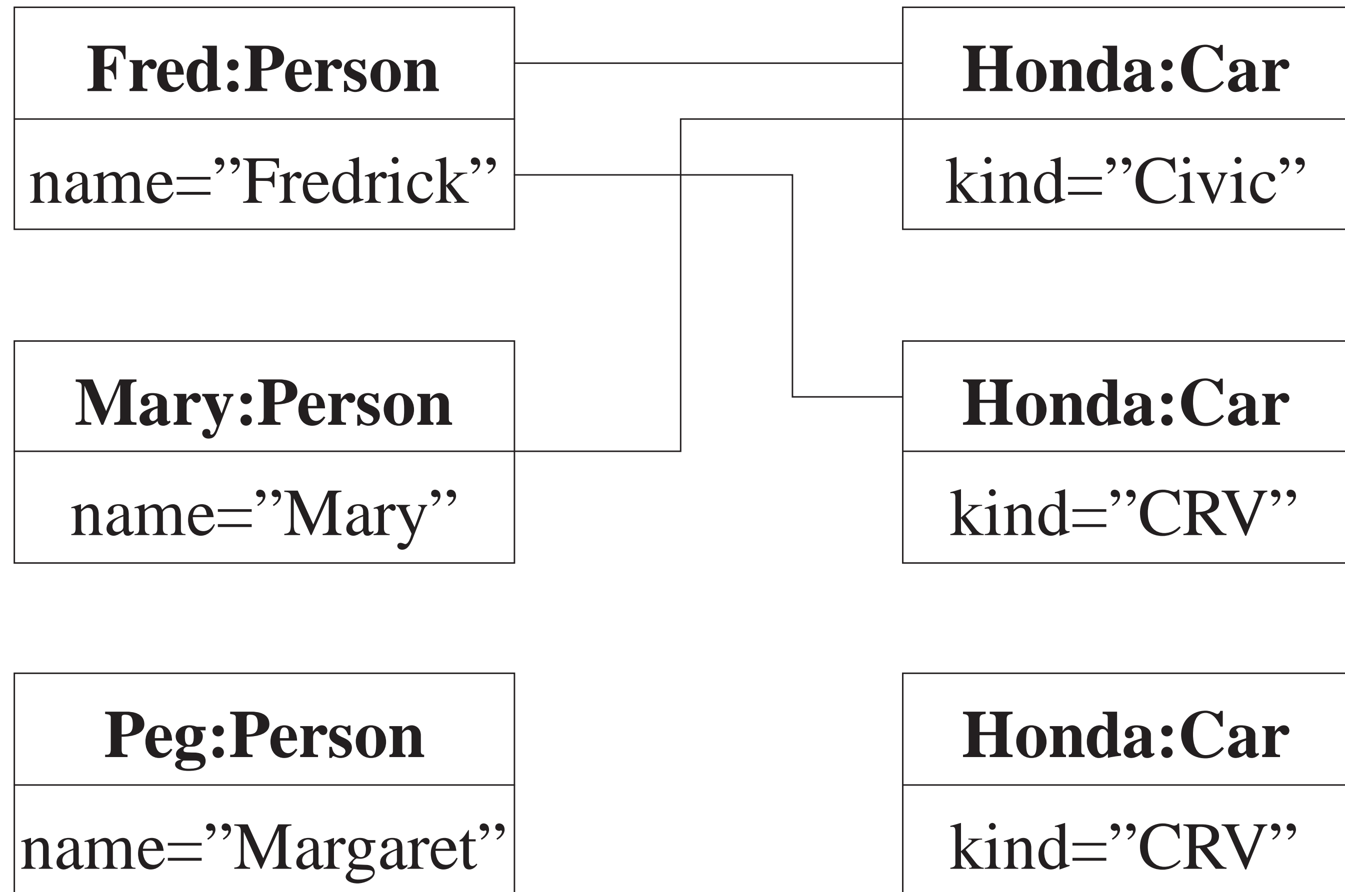
	Vending
attributes	- printer : Printer - nameServer : NameServer - Id : Integer - sodaCost : Integer - maxStockPerFlavour : Integer - stock : Integer [*]
operations	+ buy(in flavour : Flavours, inout card : WATCard) : Boolean + inventory : Integer [*] + restocked + cost : Integer + getId : Integer

- **Object diagram** : instance of a class.

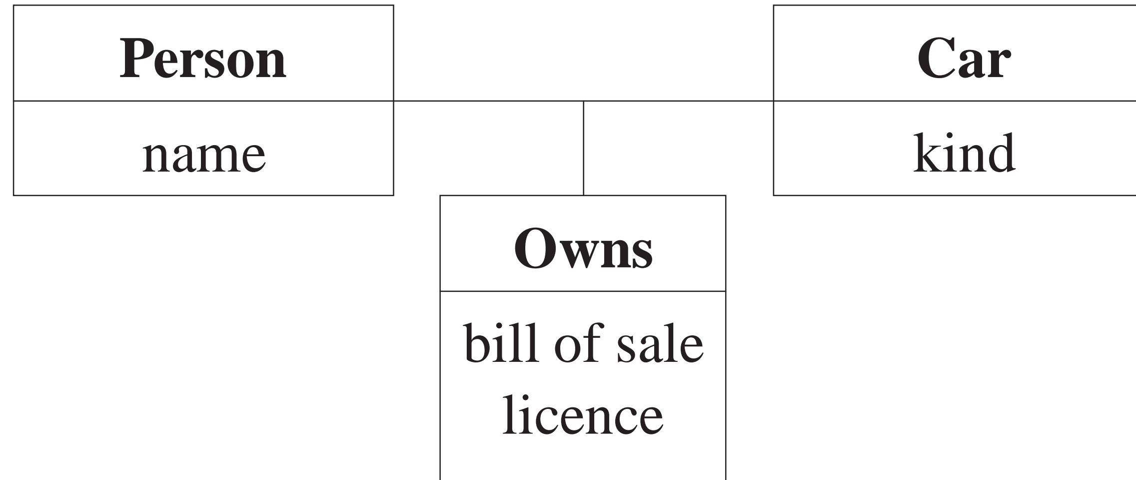
vm1:Vending	**vm2:Vending**	**vm3:Vending**

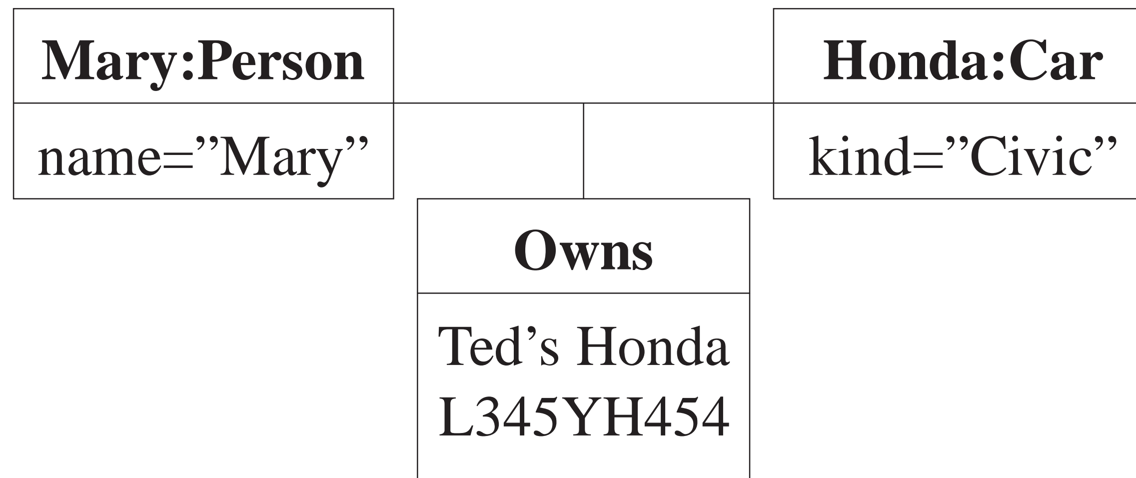
- **Association** : a named conceptual/physical connection among objects.


class diagram
Person
name
owns
Car
kind
*
1
owner
*
1..5
purchaser


object diagram
Fred:Person
name="Fredrick"
Honda:Car
kind="Civic"
Mary:Person
name="Mary"
Honda:Car
kind="CRV"
Peg:Person
name="Margaret"
Honda:Car
kind="CRV"

- name depicts connection : employee, hasGame, ownsHouse
- Association is inherently bidirectional even if name implies a specific direction.
- end names depict specific bidirectional aspect
  employer | worksFor | employee
- association is *ownership* (owns)
  - person can *own* 0 or more cars (*)
    person can *own* 1 to 5 cars
  - car can be *owned* by 0 or more people (*)
    car can be *owned* by 1 person
- Association may be implemented in a number of ways:
  - pointer from one object to another
  - related elements in arrays
- **Association Class** : association that is also a class

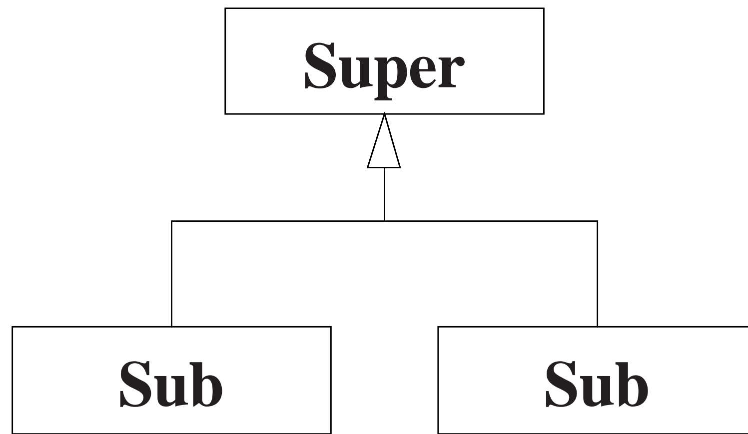




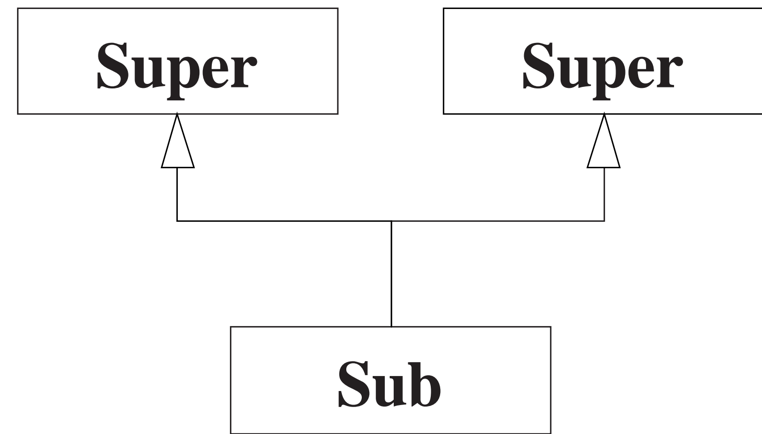
– people without cars do not need “owns” fields
  cars without owners do not need “owns” fields

– not real class because it cannot logically exist without association

- **Generalization** : reuse through form of inheritance.



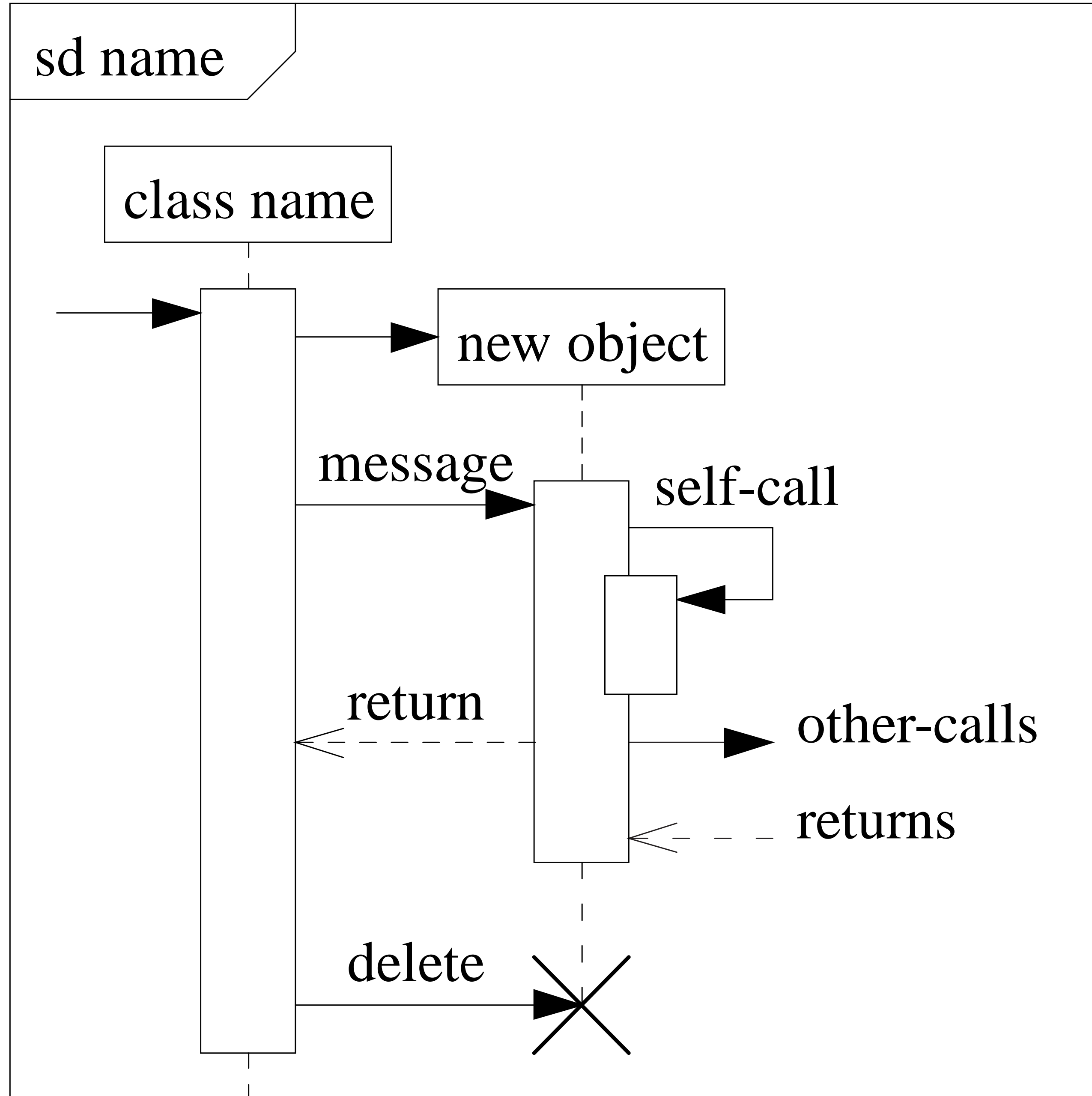
Inheritance



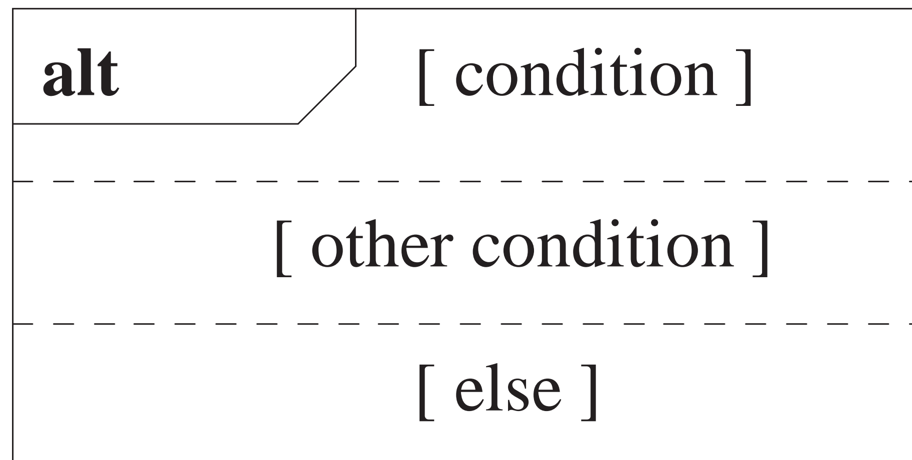
multiple inheritance

  – Inheritance establishes "is-a" relationship on type, and reuse of attributes and operations.
  – Association class can be implemented with forms of multiple inheritance (mixin).

- **Sequence diagram** : describes control-flow among objects with respect to particular scenario.
  – show static frame of program animation (call sequence).


sd name
class name
new object
message
self-call
return
other-calls
returns
delete

– show control flow



– complex and specific
– more concise to use pseudo-code (or actual code if it exists)
– use to show important/complex control flow sequences

- UML is significantly more general, supporting very complex

descriptions of relationships among entities.

- However, it is a VERY large visual mechanisms, with several confusing graphical representations.
- **Code = truth**

4.5 Programming Language Selection

- imperative, functional, logic
  - imperative : prescribes a sequence of actions directed by the state of variables, which are allowed to have multiple values (i.e., vary)
  - functional : like imperative, but variables are restricted to only one value (i.e., constant)
  - logic : series of logical expressions that are proven correct or incorrect through unification
- scripting : specialized languages (often only string type) for specific purpose (shell, GUI, awk, Perl)
- interactive/interpreted : not compiled, can be typed and executed immediately (shell language)

- managed language : hide aspects of implementation to simplify programming, e.g., hide memory management via garbage collection, execution via virtual machine
- static/dynamic type-system : variable types are fixed at compile time or allowed to vary at runtime.
- reification : manipulate program symbol-table and code at runtime, possibly with dynamic compilation.
- Useful language properties for SE:
  - abstraction/encapsulation : separate implementation from interface, and hide implementation
  - module/package : high-level bundling of types/variables/code with global initialization, e.g., container library
    - requires transitive closure of modules over program for initialization (cycles?)
  - class : aggregate data and code into single type
  - coroutines : retain control flow knowledge across routine call
  - concurrency : multiple simultaneous threads of execution (inherently difficult and complex)

- polymorphism : generalization data/code across multiple types with similar structure and behaviour
- libraries : error-free, efficient, reusable abstractions:
  * data structures
  * math
  * GUI
  * distributed/web
- compilation/runtime errors : specific, comprehensible error messages
- efficiency : after it works, after its good code, then make sure it is efficient
  * efficiency should never be an afterthought; it comes from good programming practice
  * nevertheless, programs have execution hot-spots that require extra attention
- security : subscript checking, type checking, virtual machine, dynamic checking, etc.

• Java : imperative, managed, static typing (inconsistent builtin & object types), reification, abstraction/encapsulation, packages, class (strongly object-oriented), concurrency, medium polymorphism, large libraries,

good error reporting, average to poor efficiency

- C++: imperative, not managed, static typing (consistent builtin & object types), abstraction/encapsulation, weak packages, class, routines, no concurrency, strong polymorphism, average libraries, poor error reporting, average to excellent efficiency
- Ada : imperative, many good features, but not used much anymore
- Python/Ruby/Tcl : scripting
- Haskell, Scheme : functional

4.6 Development Processes

- There are different conceptual approaches for developing software, e.g.:

  waterfall : break down project based on activity and divide activities across a timeline
  - activities : (cycle of) requirements, analysis, design, coding, testing, debugging
  - timeline : assign time to accomplish each activity up to project completion time

iterative/spiral : break down project based on functionality and divide functions across a timeline
  - functions : (cycle of) acquire/verify data, process data, generate data reports
  - timeline : assign time to perform software cycle on each function up to project completion time

staged delivery : combination of waterfall and iterative
  - start with waterfall for analysis/design, and finish with iterative for coding/testing

agile/extreme : short, intense iterations focused largely on code (versus documentation)
  - often analysis and design are done dynamically
  - often coding/testing done in pairs

- Pure waterfall is problematic because all coding/testing comes at end $\Rightarrow$ major problems can appear near project deadline.
- Pure agile can leave a project with little or no documentation.
- Selecting a process depends on:
  - kind/size of system

  - quality of system (mission critical?)
  - hardware/software technology used
  - kind/size of programming team
  - working style of teams
  - nature of completion risk
  - consequences of failure
  - culture of company
- Meta-processes specifying the effectiveness of processes:
  - Capability Maturity Model Integration (CMMI)
  - International Organization for Standardization (ISO) 9000
- Requirements
  - procedures cover key aspects of processes
  - monitoring mechanisms
  - adequate records
  - checking for defects, with appropriate and corrective action
  - regularly reviewing processes and its quality
  - facilitating continual improvement

4.7 Design Patterns

- **Design patterns** have existed since people/trades developed formal approaches.
- E.g., parent's raising children, mason's building pyramid/cathedral.
- **Pattern** is a common/repeated issue; it can be a problem or a solution.
- Name and codify common patterns for educational and communication purposes.
- Software pattern are solutions to problems:
  - name : descriptive name
  - problem : kind of issues pattern can solve
  - solution : general elements composing the design, and their relationships, responsibilities, and collaborations
  - consequences : results and trade-offs of applying the pattern (alternative/implementation issues)

4.7.1 Pattern Catalog

	creational	structural	behavioural
class	**factory method**	**adapter**	interpreter **template**
object	abstract factory builder prototype **singleton**	**adapter** bridge composite **decorator** facade flyweight **proxy**	responsibility chain command **iterator** mediator memento **observer** state strategy **visitor**

- Scope : applies to classes or objects
- Purpose : class/object creation issues, structural form, and behavioural interaction

- Class

  factory method/abstract : abstract class/template defining structure (and possibly some implementation) for creating other classes

```
struct F {                    // factory/abstract-class
    virtual void m1() = 0;
    virtual void m2() = 0;
};
struct P1 : public F {  // products
    void m1();
    void m2();
};
struct P2 : public F {
    void m1();
    void m2();
};
```

adapter/wrapper : convert interface into another

```
struct T1 {                          struct T2 {
    virtual void x(...);                 virtual void x(...);
    virtual void y(...);                 virtual void z(...);
};                                   };
struct T2toT1 : public T1, private T2 {  // adapter/wrapper
    void x(...) { T2::x(...); }
    void y(...) { ... z(...); ... }
};
void p( T1 t1 ) { ... }
T2toT1 t;
p( t );
```

template method : provide pre/post actions for subclass methods

```
class TM {
    virtual void doAction() = 0;
  protected:
    virtual void action() {
        pre-code   doAction();   post-code
    }
};
class AM : public TM {
    void doAction() {...}
  public:
    void action() { TM::action(); }
};
```

- Object

  adapter : convert interface into another

```
struct T1 {                          struct T2 {
    virtual void x(...);                 virtual void x(...);
    virtual void y(...);                 virtual void z(...);
};                                   };
struct T2toT1 : public T1 {  // adapter/wrapper
    T2toT1 &t2;
    T2toT1( T2 &t2 ) : t2( t2 ) {}
    void x(...) { t2.x(...); }
    void y(...) { ... t2.z(...); ... }
};
void p( T1 t1 ) { ... }
T2 t2;
T2toT1 t( t2 );        // any T2
p( t );
```

iterator : abstract mechanism to traverse container

```
list<Node>::iterator ni;
for ( ni = top.begin(); ni != top.end(); ++ni ) { // traverse list
    cout << "c:" << ni->c << " i:" << ni->i << endl;
}
```

singleton : single instance of class

```
class Singleton {
    struct SingletonImpl { int x, y; ... };
    static SingletonImpl *impl;  // one for all objects
  public:
    void m(...) { impl->x; ... impl->y; ... }
};
Singleton::SingletonImpl *Singleton::impl = new Singleton::SingletonImpl;
```

proxy : frontend for another object to control access

```
struct T {
    void m1(...);
    void m2(...);
};
struct SProxyT : public T {                // static
    void m1(...) { ... T:m1(...); ... }
    void m2(...) { ... T:m2(...); ... }
};
struct DProxyT : public T {                // dynamic
    T *t;
    DProxyT() { t = NULL; }
    void m1(...) { if ( t == NULL ) t = new T; t->m1(...); ... }
    void m2(...) { ... don't need t ... }
};
```

decorator : attach additional responsibilities to an object dynamically

```
struct Abstract {                          struct Concrete : public Abstract {
    virtual void m1(...) = 0;                  void m1(...);
    virtual void m2(...) = 0;                  void m2(...);
};                                         };
struct Decorator : public Abstract {    // generalize
    Abstract *parent;
    Decorator( Abstract &parent ) : parent( &parent ) {}
    void m1(...) { parent->m1(...); }    // forward
    void m2(...) { parent->m1(...); }    // forward
};
struct Decoratee : public Decorator {  // specialize
    ...
    Decoratee( Abstract &parent, ... ) : Decorator( parent ), ... {}
    void m1(...) { decorate  Decorator::m1(...);  decorate }
    void m2(...) { decorate  Decorator::m2(...);  decorate }
};
Concrete c;
Decoratee d( c );
d.m1(...);
```

observer : 1 to many dependency ⇒ change updates dependencies

```
struct Observee {                    // generalize
    Observer &oer;
    Observee( Observer &oer ) : oer( oer ) {}
    virtual void update() = 0;
};
struct Observer {
    list<Observee *> oees;           // list of observees
    static void perform( Observee *oee ) { oee->update(); }
    void attach( Observee &oee ) { oees.push_back( &oee ); }
    void deattach( Observee &oee ) { oees.remove( &oee ); }
    void notify() { for_each( oees.begin(), oees.end(), perform ); }
};
struct Oee : private Observee { // specialize
    Oee( Observer &oer ) : Observee( oer ) { oer.attach( *this ); }
    ~Oee() { oer.deattach( *this ); }
    void update() { perform update action }
};
Observer oer;
Oee oee1( oer ), oee2( oer );   // register
oer.notify();                    // trigger updates
```

visitor : perform operation on elements of heterogeneous container

```
struct Visitor {
    void visit( N1 &n ) { perform action on node }
    void visit( N2 &n ) { perform action on node }
};
struct Node {
    virtual void action( Visitor &v ) = 0;
};
struct N1 : public Node {
    void action( Visitor &v ) { v.visit( *this ); } // overload
};
struct N2 : public Node {
    void action( Visitor &v ) { v.visit( *this ); } // overload
};
Visitor v;
list<Node *> l;
for ( int i = 0; i < 10; i += 1 ) {
    l.push_back( i % 2 == 0 ? (Node *)new N1 : (Node *)new N2 );
}
for ( list<Node *>::iterator it = l.begin(); it != l.end(); ++it ) {
    (*it)->action( v );
}
```

4.8 Testing and Debugging

- A major phase in program development is testing and debugging.
- This phase often requires more time and effort than design and coding phases combined.
- Testing and debugging are not one and the same.
- **Testing** is the process of "executing" a program with the intent of discovering errors.
  - Good test is one with a high probability of finding an error.
  - Successful test is one that finds a new error.
- **Debugging** is the process of determining the cause of an error discovered by testing and correcting it.

4.8.1 Techniques

- **Human testing** is the process by which people attempt to discover errors in a program by reading its source code.
  - This is normally performed after the program has been coded but before it has been run.

  - Studies have shown that 30–70% of logic design and coding errors can be detected in this manner.

- **Code inspection** a team of people check the program for a list of common errors such as the following:
  - data reference errors: undefined variables, bad subscripts, incorrect data types
  - data declaration errors: undeclared variables, improperly initialized variables
  - computation errors: mixed mode, overflow, zero divide, etc.
  - comparison errors: incorrect relational operators (== instead of !=)
  - control errors: loop termination and initialization, off-by-one errors, boundary values
  - interface errors: arguments/parameters not matched in number or type (especially for external programs)
  - I/O errors: incorrect formats, end of file, titles, etc.
- **Walkthrough** a team of people examining the logic of a program, executing the program by hand (“play computer”)
- **Desk checking** a single person “plays computer”

- **Machine Testing** is the process of running the program using test data which has been designed to discover errors in the code.
  - Machine testing should be attempted only after human testing has been performed.
  - Test-case design, for machine testing, involves determining what subset of all possible test cases has the highest probability of detecting the greatest number of errors.
  - There are two major methods of doing this:
    * **Black-Box Testing** : program's design and internal logic are unknown when the test cases are drawn up (i.e., program is treated as a black box)
    * **White-Box Testing** : knowledge of the program's design and internal logic are used to develop the test cases
  - In generating test cases it is usually best to start with the black-box approach and then supplement these test cases with white-box tests.
  - Black-Box Testing
    * **equivalence partitioning**
      · partition all possible input cases into equivalence classes
      · select only one representative from each class for testing

- E.g., payroll program with input HOURS

```
        HOURS <= 40
   40 < HOURS <= 45   (time and a half)
   45 < HOURS         (double time)
```

- 3 equivalence classes, plus invalid hours
- Since there are many types of invalid data, invalid hours can also be partitioned into equivalence classes

∗ **boundary value testing**

- test cases which are on, above, and below boundary cases

```
39, 40, 41   (hours)
44, 45, 46      "
-1,  0,  1      "
```

∗ **cause-effect graphing**

- used to generate test cases representing combinations of conditions
- construct boolean logic-graphs, which are converted to decision tables (describing test inputs and expected outputs)

∗ **error guessing**

- surmise, through intuition and experience, what the likely errors are and then test for them

- White-Box (logic coverage) Testing
  * develop test cases which attempt to cover (exercise) every possible logic path through the program
  * test every decision alternative at least once
  * test all combinations of decisions that may affect execution
  * often impossible because of the number of tests involved
  * E.g., consider a program which contains 32 independent decisions:

```
32 independent decisions
   => 4,294,967,296 logic paths (test cases)
assume 10 test cases can be run per second
   => over 5 yr. CPU time to run
assume 1 line of output per test case
   => 70,000,000 pages long
   => stretches half way around the world
   => would fill 1600 disc dives
assume a person can read 1 line per second
   => would take 120 yr. to read output
```

  * Clearly in this case it is impossible to perform a complete test.

4.8.2 Testing Mechanics

- Module testing involves the testing of a module separately before it is integrated into, and tested with, the entire program.
- There are two major approaches to integration:
  - Non-incremental (big bang approach) : after testing (or not testing) all individual modules, of which there may be hundreds in a large system, put all the modules together and test the entire system at once
  - Incremental : as each module is tested, integrate it with modules which have already been tested and integrated
- Incremental vs Non-incremental
  - non-incremental requires the construction of test drivers to call the module and pass it different test values
  - non-incremental also requires the construction of stub modules to simulate the modules called by the module being tested
  - non-incremental testing allows a greater number of tests to be carried out in parallel
  - incremental requires fewer drivers and stubs since some of the real modules already exist and will be used in testing

- incremental detects interfacing problems earlier
- incremental debugging should be simplified because of early detection of interface problems
- incremental testing is usually more thorough
- incremental testing is considered superior to non-incremental testing.

- Stub and driver:

```
// Stub module used in testing a higher-level routine
// which calls a table insert procedure.
void tab_insert( Rectype rec ) {
    cout << "TAB_INSERT INVOKED" << endl
         << "INSERTING RECORD : " << rec << endl
         << "TAB_INSERT RETURNING" << endl;
}
```

```
// Driver module to test search routine.
// Passes multiple search KEYS and prints result of each search.
void search_driver() {
    cout << "BEGINNING SEARCH TESTS" << endl;
    for ( ;; ) {
        cin >> key;
      if ( cin.eof() ) break;
        search( key, posn, found );
        cout << "KEY:" << key << " POSN:" << posn << endl;
        cout << (found ? "" : "NOT") << "FOUND" << endl;
    } // for
    cout << "TERMINATING SEARCH TESTS" << endl;
} // search_driver
```

4.8.3 Top-down and Bottom-up Incremental Testing

- Top-down incremental testing is performed by writing the high level (control) module first, and testing it with stubs.
- Subordinate modules are written, tested, and integrated into the system, until the lowest level (worker) modules have been added.
- Bottom-up incremental testing is performed by writing the low level

(worker) modules first, and testing them with driver modules.

- Repeated by writing, testing, and integrating successively higher-level modules into the system, until the top-level module is reached.
- Top-down
  - disadvantages
    - ∗ need to write stub modules
    - ∗ some stub modules may be complicated since they must simulate the actions of the lower level modules
    - ∗ some complex tests are hard to perform because many lower level modules are missing and can't be simulated adequately by stubs
    - ∗ until the I/O modules are present, it is difficult to read test data and print results (I/O modules are typically low level)
    - ∗ encourages implementation to proceed in parallel with design (if both are top-down) which will inhibit high level design changes
    - ∗ encourages deferred testing since one is tempted to wait for the real subordinate modules to be written rather than writing stubs
  - advantages
    - ∗ better at testing the high level control logic of the system

    * boosts morale since parts of the system are working earlier (i.e., overall system using stubs is running early)

- Bottom-up
  - disadvantages
    * need to write driver modules
    * program as an entity does not work until the last (top) module is produced
  - advantages
    * better for testing low level logic
    * test conditions are easier to create
    * observation of results is easier (since I/O routines are written early)
- In practice, a combination of top-down and bottom-up testing is usually used.

4.8.4 Higher-Order Testing

- Testing methods discussed so far only test the program from the tester's point of view.

- That is, the tests check that the program behaves as the tester believes it should.
- The end user, who is paying for the program, may have a different idea of what the program should do.
- To test the program from this point of view the following tests are performed:
  - **Functional testing** : test program against its specifications to determine if it actually performs the desired functions.
  - **System testing** : compare the program against the original objectives to test the specifications and determine if the program can be used to solve the original problem
  - **Performance testing** : test if the program lives up to its speed and throughput requirements.
  - **Volume testing** : test program with large volumes of test data, possibly over long period of time.
  - **Stress testing** : test program with extreme volumes of data over a short period of time, e.g., can air traffic control system handle 250 planes at same time?

- **Usability testing** : test whether users have the skill necessary to operate the system
- **Security testing** : test whether programs and data are secure, i.e., can unauthorized people gain access to programs, files, etc.

4.8.5 Tester

- A program should not be tested by its writer, but in practice this often occurs.
- Remember, the tester only tests what they thinks it should do.
- Any misunderstandings the writer had while coding the program are carried over into testing.
- Any system written for an end user must be tested by the end user to determine if it is acceptable.
- I.e., is the system what the user ordered?
- This process is known as **acceptance testing**.
- Points to the need for a written specification to protect both the end user and the supplier.

4.8.6 Debugging

- This is the process of first determining the cause of an error discovered by testing and then fixing the problem.
- While it is undesirable to test your own programs, it is generally more productive.
- Debugging can be very hard on the ego because you have to search out your own faults.
- It can be taxing mentally as some problems can be very difficult and time consuming to track down.
- (i.e., Is the error in the algorithm or is it in the coding of the algorithm.)

4.8.7 Techniques

- Brute Force
  - throw in random print statements to display execution behaviour
  - use debugger after program fails
  - By far the most common method of debugging and by far the least efficient.

- While it requires the least effort, provides the least focus on where and what the problem is.
- Using these basic techniques with more sophisticated techniques can be very useful.

- Induction
  - Involves reasoning from the particular (clues, symptoms of the error) to the general (the cause of the error).
    - ∗ locate all pertinent data : categorize output data as correct or incorrect
    - ∗ organize data: look for contradictions
    - ∗ devise a hypothesis for the cause of the problem
    - ∗ prove the hypothesis is consistent with both correct and incorrect data, and does it account for all errors
- Deductive
  - Involves starting with a set of theories and, using the process of elimination, working towards the cause of the error.
    - ∗ list all possible causes of the problem
    - ∗ use data to find contradictions to eliminate as many hypotheses as possible

    * refine any remaining hypotheses
    * prove the hypothesis

- Backtracking
  - Working backwards through the program logic (from the point of the incorrect result) to determine where the program went wrong.
- Debugging by Testing
  - Once a problem has been discovered, make up additional test cases to zero in on this particular error.
- Debugging Principles
  - THINK
  - If you reach an impasse, sleep on it or describe it to someone else.
  - Avoid blind experimentation; it is unproductive and often complicates the problem by introducing new errors or spurious information.
  - Use debugging tools only as aids, not as the primary technique.