University of Waterloo

School of Computer Science

CS 246
Object-Oriented Software Development

Course Notes* Spring 2015

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

June 25, 2015

Outline

Introduction to basic UNIX software development tools and object-oriented programming in C++ to facilitate designing, coding, debugging, testing, and documenting of medium-sized programs. Students learn to read a specification and design software to implement it. Important skills are selecting appropriate data structures and control structures, writing reusable code, reusing existing code, understanding basic performance issues, developing debugging skills, and learning to test a program.

Contents

1 Shell **1**
1.1 File System . . . 3
1.2 Pattern Matching . . . 4
1.3 Quoting . . . 6
1.4 Shell Commands . . . 7
1.5 System Commands . . . 10
1.6 File Permission . . . 16
1.7 Input/Output Redirection . . . 17
1.8 Script . . . 19
1.9 Shift . . . 21
1.10 Shell Variables . . . 21
1.11 Eval . . . 23
1.12 Arithmetic . . . 24
1.13 Routine . . . 25
1.14 Control Structures . . . 26
1.14.1 Test . . . 26
1.14.2 Selection . . . 27
1.14.3 Looping . . . 29
1.15 Examples . . . 31
1.15.1 Hierarchical Print Script . . . 32
1.15.2 Cleanup Script . . . 33
1.16 Source-Code Management . . . 34
1.17 Global Repository . . . 34
1.17.1 Project Member . . . 35
1.18 Local Repository . . . 36
1.18.1 Modifying . . . 38
1.18.2 Conflicts . . . 42
1.19 Testing . . . 43
1.19.1 Human Testing . . . 44
1.19.2 Machine Testing . . . 44
1.19.3 Testing Strategies . . . 45
1.19.4 Tester . . . 46

2 C++ **47**
2.1 C/C++ Composition . . . 47
2.2 First Program . . . 48
2.3 Comment . . . 49
2.4 Declaration . . . 49
2.4.1 Basic Types . . . 49
2.4.2 Variable Declaration . . . 49
2.4.3 Type Qualifier . . . 50
2.4.4 Literals . . . 51
2.4.5 String . . . 52
2.5 Input/Output . . . 55
2.5.1 Formatted I/O . . . 55
2.5.1.1 Formats . . . 57
2.5.1.2 Input . . . 57
2.5.1.3 Output . . . 60
2.5.2 Unformatted I/O . . . 60
2.6 Expression . . . 61
2.6.1 Conversion . . . 62
2.6.2 Coercion . . . 63
2.6.3 Math Operations . . . 64
2.7 Control Structures . . . 65
2.7.1 Block . . . 65
2.7.2 Selection . . . 66
2.8 Structured Programming . . . 68
2.8.1 Multi-Exit Loop . . . 68
2.8.2 Multi-Level Exit . . . 71
2.9 Command-line Arguments . . . 73
2.10 Type Constructor . . . 76
2.10.1 Enumeration . . . 76
2.10.2 Pointer/Reference . . . 77
2.10.3 Aggregates . . . 79
2.10.3.1 Array . . . 79
2.10.3.2 Structure . . . 81
2.10.3.3 Union . . . 83
2.11 Type Equivalence . . . 83
2.12 Type Nesting . . . 84
2.13 Type-Constructor Literal . . . 85
2.14 Routine . . . 86
2.14.1 Argument/Parameter Passing . . . 87
2.14.2 Array Parameter . . . 89
2.15 Overloading . . . 89
2.16 Dynamic Storage Management . . . 91
2.17 Preprocessor . . . 94
2.17.1 File Inclusion . . . 95
2.17.2 Variables/Substitution . . . 95

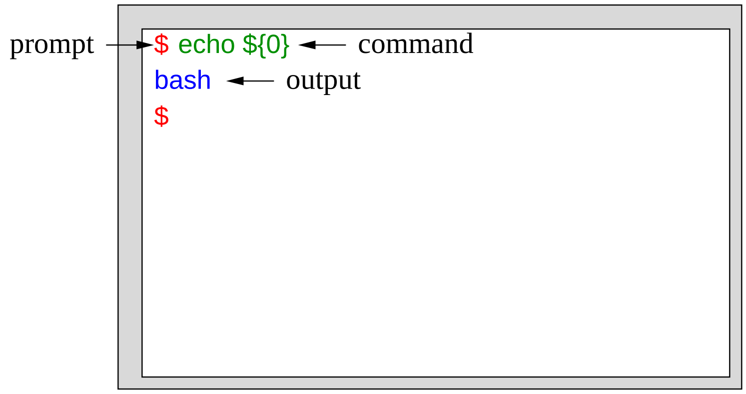
2.17.3 Conditional Inclusion . . . 96
2.18 Assertions . . . 97
2.19 Debugging . . . 99
2.19.1 Debug Print Statements . . . 99
2.20 Debugger . . . 100
2.20.1 GDB . . . 101
2.21 Object . . . 105
2.21.1 Object Member . . . 106
2.21.2 Operator Member . . . 107
2.21.3 Constructor . . . 108
2.21.3.1 Literal . . . 110
2.21.3.2 Conversion . . . 110
2.21.4 Destructor . . . 111
2.21.5 Copy Constructor / Assignment . . . 112
2.21.6 Initialize const / Object Member . . . 116
2.21.7 Static Member . . . 117
2.22 Random Numbers . . . 118
2.23 System Modelling . . . 120
2.23.1 UML . . . 121
2.24 Singleton Pattern . . . 126
2.25 Declaration Before Use . . . 127
2.26 Encapsulation . . . 130
2.27 Separate Compilation . . . 133
2.28 Design Quality . . . 139
2.28.1 Coupling . . . 139
2.28.2 Cohesion . . . 140
2.29 Inheritance . . . 142
2.29.1 Implementation Inheritance . . . 142
2.29.2 Type Inheritance . . . 144
2.29.3 UML inheritance . . . 146
2.29.4 Constructor/Destructor . . . 147
2.29.5 Copy Constructor / Assignment . . . 148
2.29.6 Overloading . . . 149
2.29.7 Virtual Routine . . . 149
2.29.8 Downcast . . . 152
2.29.9 Slicing . . . 152
2.29.10 Protected Members . . . 153
2.29.11 Abstract Class . . . 153
2.29.12 Multiple Inheritance . . . 155
2.30 Composition / Inheritance Design . . . 155
2.31 Observer Pattern . . . 157
2.32 Decorator Pattern . . . 158
2.33 Factory Pattern . . . 158
2.34 Template Method . . . 159
2.35 Compiling Complex Programs . . . 160

2.35.1 Dependencies 160
2.35.2 Make 161
2.36 Template 164
2.36.1 Standard Library 166
2.36.1.1 Vector 167
2.36.1.2 Map 170
2.36.1.3 List 171
2.36.1.4 for_each 172
2.37 Namespace 173
2.38 Decorator Pattern 175

Index **177**

1 Shell

- Computer interaction requires mechanisms to display information and perform operations.
- Two main approaches are graphical and command line.
- **Graphical user interface** (GUI) (desktop):
  - icons represent actions (programs) and data (files),
  - click on icon launches (starts) a program or displays data,
  - program may pop up a dialog box for arguments to affect its execution.
- **Command-line interface** (shell):
  - text strings access programs (commands) and data (file names),
  - command typed after a prompt in an interactive area to start it,
  - arguments follow the command to affect its execution.
- Graphical interface easy for simple tasks, but seldom programmable for complex operations.
- Command-line interface harder for simple tasks (more typing), but allows programming.
- Many systems provide both.
- **Shell** is a program that reads commands and interprets them.
- Provide a simple programming-language with *string* variables and few statements.
- Unix shells falls into two basic camps: **sh** (**ksh**, **bash**) and **csh** (**tcsh**), each with slightly different syntax and semantics.
- Focus on **bash** with some tcsh.
- **Terminal** or **xterm** area (window) is where shell runs.



- Command line begins with (customizable) **prompt**: $ (sh) or % (csh).
- Command typed after prompt ***not*** executed until **Enter**/Return key pressed.

```
$ dateEnter                      # print current date
Thu Aug 20 08:44:27 EDT 2009
$ whoamiEnter                    # print userid
jfdoe
$ echo Hi There!Enter            # print any string
Hi There!
```

- Command comment begins with hash (#) and continues to end of line.

```
$ # comment text . . . does nothing
$
```

- Multiple commands on command line separated by semi-colon.

```
$ date ; whoami ; echo Hi There!    # 3 commands
Sat Dec 19 07:36:17 EST 2009
jfdoe
Hi There!
```

- Commands can be editted on the command line (not sh):

```
$ data ; Whoami ;  cho Hi There!
```

  - position cursor, ▯, with ◁ and ▷ arrow keys,
  - type new characters before cursor,
  - remove characters before cursor with backspace/delete key,
  - press Enter at any point to execute modified command.

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

```
$ uname -m              # machine type
x86_64
$ uname -s              # operating system
Linux
$ uname -a              # all system information
Linux linux008.student.cs 2.6.31-21-server #59-Ubuntu SMP x86_64 GNU/Linux
```

- Options are normally processed left to right; one option may cancel another.
- ***No standardization for command option names and syntax.***
- Shell/terminal terminates with command **exit**.

```
$ exit                  # exit shell and terminal
```

  - when the login terminal/xterm terminates, sign off computer (logout).

1.1 File System

- Shell commands interact extensively with the **file system**.
- Files are containers for data stored on persistent storage (usually disk).
- File names organized in N-ary tree: directories are vertexes, files are leaves.
- Information is stored at specific locations in the hierarchy.

```
/                       root of local file system
    bin                 basic system commands
    lib                 system libraries
    usr
        bin             more system commands
        lib             more system libraries
        include         system include files, .h files
    tmp                 system temporary files
    home or u           user files
        jfdoe           home directory
            ., . .          current, parent directory
            .bashrc, .emacs, .login,. . .  hidden files
            cs246           course files
                a1      assignment 1 files
                        q1x.C, q2y.h, q2y.cc, q3z.cpp
        other users
```

- Directory named "/" is the root of the file system.
- bin, lib, usr, include : system commands, system library and include files.
- tmp : temporary files created by commands (***shared among all users***).
- home or u : user files are located in this directory.
- Directory for a particular user (**jfdoe**) is called their **home directory**.
- Shell special character "~" (tilde) expands to user's home directory.

```
~/cs246/a1/q1x.C        # => /u/jfdoe/cs246/a1/q1x.C
```

- ***Every*** directory contains 2 special directories:
  - "." points to current directory.

    ```
    ./cs246/a1/q1x.C        # => /u/jfdoe/cs246/a1/q1x.C
    ```

  - ". ." points to parent directory above the current directory.

    ```
    . ./. ./usr/include/limits.h  # => /usr/include/limits.h
    ```

- **Hidden files** contain administrative information and start with "." (dot).

- Each file has a unique path-name referenced with an absolute pathname.
- **Absolute pathname** is a list of all the directory names from the root to the file name separated by the backslash character "/".

```
/u/jfdoe/cs246/a1/q1x.C        # => file q1x.C
```

- Shell provides concept of **working directory** (**current directory**), which is the active location in the file hierarchy.
- E.g., after sign on, the working directory starts at user's home directory.
- File name not starting with "/" is prefixed with working directory to create necessary absolute pathname.
- **Relative pathname** is file name/path prefixed with working directory.
  - E.g., when user jfdoe signs on, home/working directories set to /u/jfdoe.

```
.bashrc                  # => /u/jfdoe/.bashrc
cs246/a1/q1x.C           # => /u/jfdoe/cs246/a1/q1x.C
```

1.2 Pattern Matching

- Shells provide pattern matching of file names, **globbing** (regular expressions), to reduce typing lists of file names.
- Different shells and commands support slightly different forms and syntax for patterns.
- Pattern matching is provided by characters, *, ?, {}, [], denoting different **wildcard**s (from card games, e.g., Joker is wild, i.e., can be any card).
- Patterns are composable: multiple wildcards joined into complex pattern (Aces, 2s and Jacks are wild).
- E.g., if the working directory is /u/jfdoe/cs246/a1 containing files:
  q1x.C, q2y.h, q2y.cc, q3z.cpp
  - * matches 0 or more characters

```
$ echo q*  # shell globs "q*" to match file names, which echo prints
q1x.C q2y.h q2y.cc q3z.cpp
```

  - ? matches 1 character

```
$ echo q*.??
q2y.cc
```

  - {. . .} matches any alternative in the set

```
$ echo *.{C,cc,cpp}
q1x.C q2y.cc q3z.cpp
```

- [. . .] matches 1 character in the set

```
$ echo q[12]*
q1x.C q2y.h q2y.cc
```

- [!. . .] (^ csh) matches 1 character ***not*** in the set

```
$ echo q[!1]*
q2y.h q2y.cc q3z.cpp
```

- Create ranges using hyphen (dash)

```
[0-3]        # => 0 1 2 3
[a-zA-Z]     # => lower or upper case letter
[!a-zA-Z]    # => any character not a letter
```

- Hyphen is escaped by putting it at start or end of set

```
[-?*]*       # => matches file names starting with -, ?, or *
```

- If globbing pattern does not match any files, the pattern is the file name (including wildcards).

```
$ echo q*.ww q[a-z].cc  # files do not exist so no expansion
q*.ww q[a-z].cc
```

csh prints: **echo**: No match.

- **Hidden files**, starting with "." (dot), are ignored by globbing patterns
  - ⇒ * does not match all file names in a directory.
- Pattern .* matches all hidden files:
  - match ".", then zero or more characters, e.g., .bashrc, .login, etc., ***and*** ".", ". ."
  - ***matching ".", ". ." can be dangerous***

```
$ rm .*       # remove hidden files, and current/parent directory!!!
```

- Pattern .[!.]* matches all single "." hidden files but ***not*** " ." and " . ." directories.
  - match ".", then any character NOT a " .", and zero or more characters
  - ⇒ there must be at least 2 characters, the 2nd character cannot be a dot
  - "." starts with dot but fails the 2nd pattern requiring another character
  - ". ." starts with dot but the second dot fails the 2nd pattern requiring non-dot character
- On the command line, pressing the tab key after typing several characters of a command/file name causes the shell to automatically complete the name.

```
$ ectab              # cause completion of command name to echo
$ echo q1tab         # cause completion of file name to q1x.C
```

- If the completion is ambiguous (i.e., more than one):
  - shell "beeps" (maybe),

- prints all completions if tab is pressed again,
- then you type more characters to uniquely identify the name.

```
$ datab              # beep
$ datab              # print completions
dash  date
$ dattab             # add "t" to complete command giving "date"
```

1.3 Quoting

- **Quoting** controls how shell interprets strings of characters.
- **Backslash** (\ ) : **escape** any character, including special characters.

```
$ echo .[!.]*          # globbing pattern
.bashrc  .emacs  .login  .vimrc
$ echo \.\[\!\.\]\*    # print globbing pattern
.[!.]*
```

- **Backquote** (`) : execute text as a command, and substitute with command output.

```
$ echo `whoami`     # $ whoami => jfdoe
jfdoe
```

- Globbing does NOT occur within a single/double quoted string.
- **Single quote** (') : protect everything (even newline) except single quote.
  - E.g., file name containing special characters (blank/wildcard/comment).

```
$ echo Book Report #2
Book Report
$ echo 'Book Report #2'
Book Report #2
$ echo '.[!.]*'          # no globbing
.[!.]*
$ echo '\.\[\!\.\]\*'# no escaping
\.\[\!\.\]\*
$ echo '`whoami`'        # no backquote
`whoami`
$ echo 'abc              # yes newline
> cdf'        # prompt ">" means current line is incomplete
abc           # yes newline
cdf
$ echo '\''              # no escape single quote
>
```

A single quote cannot appear inside a single quoted string.

- **Double quote** (") : protect everything except double quote, backquote, and dollar sign (variables, see Section 1.10), which can be escaped.

```
$ echo ".[!.]*"          # no globbing
.[!.]*
$ echo "\.\[\!\.\]\*" # no escaping
\.\[\!\.\]\*
$ echo "`whoami`"        # yes backquote
cs246
echo "abc                # yes newline
> cdf"
abc                 # yes newline
cdf
$ echo "\""              # yes escape double quote
"
```

- ***To stop prompting or output from any shell command***, type <ctrl>-c (C-c), i.e., press <ctrl> then c key, causing shell to interrupt current command.

```
$ echo "abc
> C-c
$
```

1.4 Shell Commands

- A command typed after the prompt is executed by the shell (shell command) or the shell calls a command program (system command, see Section 1.5, p. 10).
- Shell commands read/write shell information/state.
- **help** : display information about bash commands (not sh or csh).

  help *[command-name]*

  ```
  $ help cd
  cd: cd [-L|-P] [dir]
      Change the shell working directory. ...
  ```

  - without argument, lists all bash commands.
- **cd** : change the working directory (navigate file hierarchy).

  cd *[directory]*

  ```
  $ cd .           # change to current directory
  $ cd ..          # change to parent directory
  $ cd cs246       # change to subdirectory
  $ cd cs246/a1    # change to subsubdirectory
  $ cd ..          # where am I ?
  ```

  - argument must be a directory and not a file
  - **cd** : move to home directory, same as **cd** ~
  - **cd** - : move to previous working directory

- **cd** ~/cs246 : move to cs246 directory contained in jfdoe home directory
- **cd** /usr/include : move to /usr/include directory
- **cd** .. : move up one directory level
- If path does not exist, **cd** fails and working directory is unchanged.

- **pwd** : print absolute path for working directory (when you're lost).

```
$ pwd
/u/jfdoe/cs246
```

- **history** and "!" (bang!) : print a numbered history of most recent commands entered and access them.

$ **history**	$!2 *# rerun 2nd history command*
1 date	whoami
2 whoami	jfdoe
3 **echo** Hi There	$!! *# rerun last history command*
4 **help**	whoami
5 **cd** ..	jfdoe
6 **pwd**	$!ec *# rerun last history command starting with "ec"*
	echo Hi There
	Hi There

- !N rerun command N
- !! rerun last command
- !xyz rerun last command starting with the string "xyz"
- Arrow keys △/▽ move forward/backward through history commands on command line.

```
$ △ pwd
$ △ cd ..
$ △ help
```

- **alias** : substitution string for command name.

alias *[command-name=string]*

- No spaces before/after "=" (csh does not have "=").
- Provide *nickname* for frequently used or variations of a command.

```
$ alias d=date
$ d
Mon Oct 27 12:56:36 EDT 2008
$ alias off="clear; exit"  # why quotes?
$ off                      # clear screen before terminating shell
```

- ***Always use quotes to prevent problems.***
- Aliases are composable, i.e., one alias references another.

```
$ alias now="d"
$ now
Mon Oct 27 12:56:37 EDT 2008
```

- ◦ Without argument, print all currently defined alias names and strings.

```
$ alias
alias d='date'
alias now='d'
alias off='clear; exit'
```

- ◦ ***Alias CANNOT be command argument*** (see page 22).

```
$ alias a1=/u/jfdoe/cs246/a1
$ cd a1      # alias only expands for command
bash: cd: a1: No such file or directory
```

- ◦ Alias entered on command line disappears when shell terminates.
- ◦ Two options for making aliases persist across sessions:
  1. insert the **alias** commands in the appropriate (hidden) .*shell*rc file,
  2. place a list of **alias** commands in a file (often .aliases) and **source** (see page 26) that file from the .*shell*rc file.

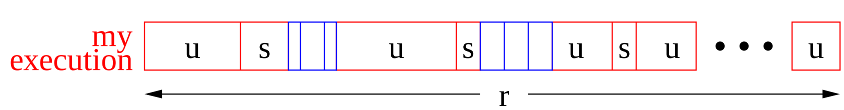
- **type** (csh **which**) : indicate how name is interpreted as command.

```
$ type now
now is aliased to 'd'
$ type d
d is aliased to 'date'
$ type date
date is hashed (/bin/date)     # hashed for faster lookup
$ type -p date                 # -p => only print command file-name
/bin/date
$ type fred                    # no "fred" command
bash: type: fred: not found
$ type -p fred                 # no output
```

- **echo** : write arguments, separated by spaces and terminated with newline.

```
$ echo  We  like  ice  cream    # 4 arguments
We like ice cream
$ echo " We  like  ice  cream " # 1 argument, notice spaces
 We  like  ice  cream
```

- **time** : execute a command and print a time summary.
  - ◦ program execution is composed of user and system time.
    - ∗ **user time** is the CPU time used during execution of a program.
    - ∗ **system time** is the CPU time used by the operating system to support execution of a program (e.g., file or network access).
  - ◦ program execution is also interleaved with other programs:



  - **real time** is from start to end including interleavings: user + system ≈ real-time
- different shells print these values differently.

$ **time** myprog	% **time** myprog
real 1.2	0.94u 0.22s 0:01.2
user 0.9	
sys 0.2	

- test if program modification produces change in execution performance
  - used to compare user (and possibly system) times before and after modification

- **exit** : terminates shell, with optional integer exit status (return code) N.

  exit *[N]*

  - *[N]* is in range 0-255; larger values are truncated (256 ⇒ 0, 257 ⇒ 1, etc.) , negative values (if allowed) become unsigned (-1 ⇒ 255).
  - exit status defaults to zero if unspecified (see pages 21 and 27 for status usage), which usually means success.

1.5 System Commands

- Command programs called by shell (versus executed by shell).
- sh / bash / csh / tcsh : start **subshell** to switching among shells.

```
$ ...          # bash commands
$ tcsh         # start tcsh in bash
% ...          # tcsh commands
% sh           # start sh in tcsh
$ ...          # sh commands
$ exit         # exit sh
% exit         # exit tcsh
$ exit         # exit original bash and terminal
```

- chsh : set login shell (bash, tcsh, etc.).

```
$ echo ${0}    # what shell am I using ?
/bin/tcsh
$ chsh         # change to different shell
Password: XXXXXX
Changing the login shell for jfdoe
Enter the new value, or press ENTER for the default
        Login Shell [/bin/tcsh]: /bin/bash
```

- man : print information about command, option names (see page 2) and function.

```
$ man bash
...            # information about "bash" command
$ man man
...            # information about "man" command
```

- cat/more/less : print files.

  cat *file-list*

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

```
$ cat q1.h
...          # print file q1.h completely
$ more q1.h
...          # print file q1.h one screen at a time
             # type "space" for next screen, "q" to stop
```

- mkdir : create a new directory at specified location in file hierarchy.

  mkdir *directory-name-list*

```
$ mkdir d d1 d2 d3 # create 4 directories in working directory
```

- ls : list the directories and files in the specified directory.

  ls *[* -al *] [file or directory name-list]*

  - -a lists *all* files, including hidden files (see page 3)
  - -l generates a *long* listing (details) for each file
  - no file/directory name implies working directory

```
$ ls .          # list working directory (non-hidden files)
q1x.C  q2y.h  q2y.cc  q3z.cpp
$ ls -a         # list working directory plus hidden files
. .. .bashrc  .emacs  .login  q1x.C  q2y.h  q2y.cc  q3z.cpp
```

- cp : copy files; with the -r option, copy directories.

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

  - -i prompt for verification if a target file is being replaced.
  - -r recursively copy contents of a source directory to a target directory.

```
$ cp f1 f2         # copy file f1 to f2
$ cp f1 f2 f3 d    # copy files f1, f2, f3 into directory d
$ cp -r d1 d2 d3 # copy directories d1, d2 recursively into directory d3
```

- mv : move files and/or directories to another location in the file hierarchy.

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

- rename source-file if target-file does not exist; otherwise replace target-file.
- -i prompt for verification if a target file is being replaced.

```
$ mv f1 foo       # rename file f1 to foo
$ mv f2 f3        # delete file f3 and rename file f2 to f3
$ mv f3 d1 d2 d3 # move file f3 and directories d1, d2 into directory d3
```

- rm : remove (delete) files; with the -r option, remove directories.

  rm *[* -ifr *] file-list/directory-list*

```
$ rm f1 f2 f2      # file list
$ rm -r d1 d2     # directory list, and all subfiles/directories
$ rm -r f1 d1 f2  # file and directory list
```

  - -i prompt for verification for each file/directory being removed.
  - -f (default) do not prompt for removal verification for each file/directory.
  - -r recursively delete the contents of a directory.
  - ***UNIX does not give a second chance to recover deleted files; be careful when using rm, especially with globbing, e.g., rm * or rm .****
  - UW has hidden directory .snapshot in every directory containing backups of all files in that directory
    - per hour for 8 hours, per night for 7 days, per week for 21 weeks

```
$ ls .snapshot                    # directories containing backup files
hourly.0   hourly.6   nightly.4    weekly.11  weekly.17  weekly.3   weekly.9
hourly.1   hourly.7   nightly.5    weekly.12  weekly.18  weekly.4
hourly.2   nightly.0  nightly.6    weekly.13  weekly.19  weekly.5
hourly.3   nightly.1  weekly.0     weekly.14  weekly.2   weekly.6
hourly.4   nightly.2  weekly.1     weekly.15  weekly.20  weekly.7
hourly.5   nightly.3  weekly.10    weekly.16  weekly.21  weekly.8
$ cp .snapshot/hourly.0/q1.h q1.h # restore file from previous hour
```

- **alias** : setting command options for particular commands.

```
$ alias cp="cp -i"
$ alias mv="mv -i"
$ alias rm="rm -i"
```

  which always uses the -i option (see page 11) on commands cp, mv and rm.

- Alias can be overridden by quoting or escaping the command name.

```
$ "rm" -r xyz
$ \rm -r xyz
```

  which does not add the -i option.

- lp/lpstat/lprm : add, query and remove files from the printer queues.

```
lp [ -d printer-name ] file-list
lpstat [ -d ] [ -p [ printer-name ] ]
lprm [ -P printer-name ] job-number
```

  - if no printer is specified, use default printer (ljp_3016 in MC3016).
  - lpstat : -d prints default printer, -p without printer-name lists all printers
  - each job on a printer's queue has a unique number.
  - use this number to remove a job from a print queue.

```
$ lp -d ljp_3016 uml.ps    # print file to printer ljp_3016
$ lpstat                   # check status, default printer ljp_3016
Spool queue: lp (ljp_3016)
Rank   Owner      Job Files          Total Size
1st    rggowner   308 tt22           10999276 bytes
2nd    jfdoe      403 uml.ps         41262 bytes
$ lprm 403                 # cancel printing
services203.math: cfA403services16.student.cs dequeued
$ lpstat                   # check if cancelled
Spool queue: lp (ljp_3016)
Rank   Owner      Job Files          Total Size
1st    rggowner   308 tt22           10999276 bytes
```

- cmp/diff : compare 2 files and print differences.

```
cmp file1 file2
diff  file1 file2
```

  - return 0 if files equal (no output) and non-zero otherwise (output difference)
  - cmp generates the first difference between the files.

	file x	file y
1	a\n	a\n
2	b\n	b\n
3	c\n	c\n
4	**d**\n	**e**\n
5	g\n	h\n
6	h\n	i\n
7		g\n

```
$ cmp x y
x y differ: char 7, line 4
```

newline is counted $\Rightarrow$ 2 characters per line in files

  - diff generates output describing how to change first file into second file.

```
$ diff x y
4,5c4    # replace lines 4 and 5 of 1st file
< d      #   with line 4 of 2nd file
< g
---
> e
6a6,7    # after line 6 of 1st file
> i      #   add lines 6 and 7 of 2nd file
> g
```

- Useful for checking output from previous program with current program.

- find : search for names in the file hierarchy.

  find *[file/directory-list] [expr]*

  - if *[file/directory-list]* omitted, search working directory, ".”
  - if *[expr]* omitted, match all file names, “-name "*"”
  - recursively find file/directory names starting in working directory matching pattern “t*”

    ```
    $ find -name "t*"        # why quotes ?
    ./test.cc
    ./testdata
    ```

  - -name *pattern* restrict file names to globbing pattern
  - -type f | d select files of type <u>f</u>ile or <u>d</u>irectory
  - -maxdepth *N* recursively descend at most *N* directory levels (0 ⇒ working directory)
  - logical *not*, *and* and *or* (precedence order)

    ```
    -not expr
    expr -a expr
    expr -o expr
    ```

    -a assumed if no operator, *expr expr* ⇒ *expr* -a *expr*
  - \\(*expr* \\) evaluation order
  - recursively find only file names starting in working directory matching pattern “t*”

    ```
    $ find . -type f -name "t*"  # same as -type f -a -name “t*”
    test.cc
    ```

  - recursively find only file names in file list (excluding hidden files) to a maximum depth of 3 matching patterns t* or *.C.

    ```
    $ find * -maxdepth 3 -a -type f -a \( -name "t*" -o -name "*.C" \)
    test.cc
    q1.C
    testdata/data.C
    ```

- egrep : (<u>e</u>xtended <u>g</u>lobal <u>r</u>egular <u>e</u>xpression <u>p</u>rint) search & print lines matching pattern in files (Google). (same as grep -E)

  egrep -irn *pattern-string file-list*

- list lines containing “main” in files with suffix “.cc”

```
$ egrep main *.cc          # why no quotes ?
q1.cc:int main() {
q2.cc:int main() {
```

- -i ignore case in both pattern and input files
- -r recursively examine files in directories.
- -n prefix each matching line with line number
- returns 0 if one or more lines match and non-zero otherwise (counter intuitive)
- list lines with line numbers containing “main” in files with suffix “.cc”

```
$ egrep -n main *.cc
q1.cc:33:int main() {
q2.cc:45:int main() {
```

- list lines containing “fred” in any case in file “names.tex”

```
$ egrep -i fred names.txt
names.txt:Fred Derf
names.txt:FRED HOLMES
names.txt:freddy mercury
```

- list lines that match start of line “^”, “#include”, 1 or more space or tab “[ ]+”, either “"” or “<”, 1 or more characters “.+”, either “"” or “>”, end of line “$” in files with suffix “.h” or “.cc”

```
$ egrep '^#include[  ]+["<].+[">]$' *.{h,cc}   # why quotes ?
egrep: *.h: No such file or directory
q1.cc:#include <iostream>
q1.cc:#include <iomanip>
q1.cc:#include "q1.h"
```

- **egrep pattern is different from globbing pattern** (see man egrep).
  Most important difference is “*” is a wildcard qualifier not a wildcard.

- ssh : (secure shell) safe, encrypted, remote-login between client/server hosts.

  ssh *[* -Y *] [* -l user *] [user@]* hostname

  - -Y allows remote computer (University) to create windows on local computer (home).
  - -l login user on the server machine.
  - To login from home to UW environment:

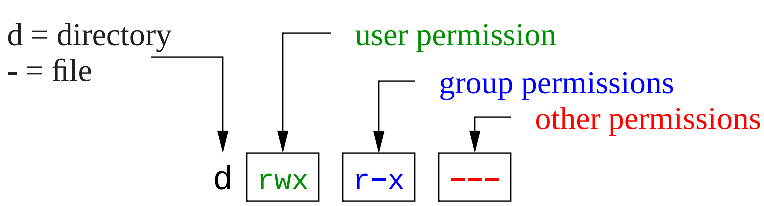
```
$ ssh -Y -l jfdoe linux.student.cs.uwaterloo.ca
...             # enter password, run commands (editor, programs)
$ ssh -Y jfdoe@linux.student.cs.uwaterloo.ca
```

1.6 File Permission

- UNIX supports security for each file or directory based on 3 kinds of users:
  - user : owner of the file,
  - group : arbitrary name associated with a set of userids,
  - other : any other user.
- File or directory has permissions, read, write, and execute/search for the 3 sets of users.
  - Read/write allow specified set of users to read/write a file/directory.
  - Executable/searchable:
    * file : execute as a command, e.g., file contains a program or shell script,
    * directory : search by certain system operations but not read in general.
- Use `ls -l` command to print file-permission information.

```
drwxr-x--- 2 jfdoe jfdoe  4096 Oct 19 18:19 cs246
drwxr-x--- 2 jfdoe jfdoe  4096 Oct 21 08:51 cs245
-rw------- 1 jfdoe jfdoe 22714 Oct 21 08:50 test.cc
-rw------- 1 jfdoe jfdoe 63332 Oct 21 08:50 notes.tex
```

- Columns are: permissions, #-of-directories (including "." and ".."), owner, group, file size, change date, file name.
- Permission information is:



- E.g., d `rwx` `r-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, never allow "other" users to read or write your files.***
- Default permissions (usually) on:
  - file: `rw-` `r--` `---`, owner read/write, group only read, other none.
  - directory: `rwx` `---` `---`, owner read/write/execute, group/other none.

- chgrp : change group-name associated with file.

  chgrp *[* -R *] group-name file/directory-list*

  - -R recursively modify the group of a directory.

  ```
  $ chgrp cs246_05 cs246   # course directory
  $ chgrp -R cs246_05 cs246/a5   # assignment directory/files
  ```

  Must associate group along entire pathname and files.

- ***Creating/deleting group-names is done by system administrator.***

- chmod : add or remove from any of the 3 security levels.

  chmod *[* -R *] mode-list file/directory-list*

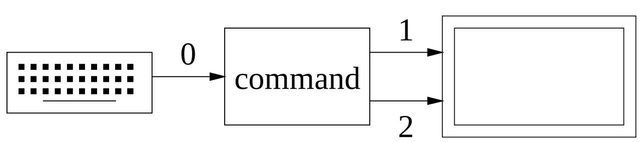
  - -R recursively modify the security of a directory.
  - *mode-list* has the form *security-level operator permission*.
  - Security levels are **u** for user, **g** for group, **o** for other, **a** for all (ugo).
  - Operator **+** adds, **-** removes, **=** sets specific permission.
  - Permissions are **r** for readable, **w** for writable and **x** for executable.
  - Elements of the *mode-list* are separated by commas.

  ```
  chmod g-r,o-r,g-w,o-w foo  # long form, remove read/write for group/others users
  chmod go-rw foo            # short form
  chmod g=rx cs246           # allow group users read/search
  chmod -R g+rw cs246/a5     # allow group users read/write, recursively
  ```

  To achieve desired access, must associate permission along entire pathname and files.

1.7 Input/Output Redirection

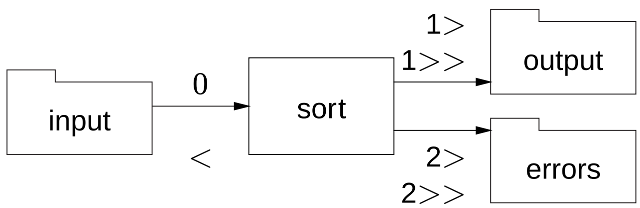
- Every command has three standard files: input (0), output (1) and error (2).
- By default, these are connected to the keyboard (input) and screen (output/error).




```
$ sort -n    # -n means numeric sort
7            sort reads unsorted values from keyboard
30
5
C-d          close input file
5            sort prints sorted values to screen
7
30
```

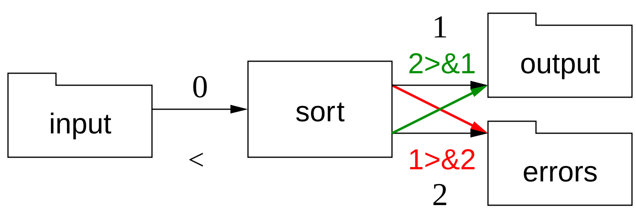
- ***To close an input file from the keyboard***, type <ctrl>-d (C-d), i.e., press <ctrl> then d key, causing the shell to close the keyboard input file.
- Redirection allows:
  - input from a file (faster than typing at keyboard),
  - saving output to a file for subsequent examination or processing.
- Redirection performed using operators < for input and > / >> for output to/from other sources.

```
$ sort -n  < input  1> output  2> errors
```



  - < means read input from file rather than keyboard.
  - > (same as 1>), 1>, 2> means (create if needed) file and write output/errors to file rather than screen (destructive).
  - >> (same as 1>>), 1>>, 2>> means (create if needed) file and append output/errors to file rather than screen.
- Command is (usually) unaware of redirection.
- Can tie standard error to output (and vice versa) using ">&" ⇒ both write to same place.

```
$ sort -n  < input  1> output  2>&1   # stderr (2) goes to stdout (1)
$ sort -n  < input  1> output  1>&2   # stdout (1) goes to stderr (2)
```



- Order of tying redirection files is important.

```
$ sort 2>&1 > output    # tie stderr to screen, redirect stdout to "output"
$ sort > output 2>&1    # redirect stdout to "output", tie stderr to "output"
```

- To ignore output, redirect to pseudo-file /dev/null.

```
$ sort data 2> /dev/null  # ignore error messages
```

- Redirection requires explicit creation of intermediate (temporary) files.

```
$ sort data > sortdata    # sort data and store in "sortdata"
$ egrep -v "abc" sortdata > temp  # print lines without "abc", store in "temp"
$ tr a b < temp > result  # translate a's to b's and store in "result"
$ rm sortdata temp        # remove intermediate files
```

- Shell pipe operator | makes standard output for a command the standard input for the next command, without creating intermediate file.

```
$ sort data | grep -v "abc" | tr a b > result
```

- Standard error is not piped unless redirected to standard output.

```
$ sort data 2>&1 | grep -v "abc" 2>&1 | tr a b > result 2>&1
```

  now both standard output and error go through pipe.

- Print file hierarchy using indentation (see page 3).

`$ find cs246` `cs246` `cs246/a1` `cs246/a1/q1x.C` `cs246/a1/q2y.h` `cs246/a1/q2y.cc` `cs246/a1/q3z.cpp`	`$ find cs246 | sed 's|[^/]*/|   |g'` `cs246` `   a1` `      q1x.C` `      q2y.h` `      q2y.cc` `      q3z.cpp`

- sed : inline editor
  - pattern changes all occurrences (g) of string [^/]*/ (zero or more characters not "/" and then "/", where "*" is a wildcard qualifier not a wildcard) to 3 spaces.

1.8 Script

- A **shell program** or **script** is a file (scriptfile) containing shell commands to be executed.

```
#!/bin/bash  [ -x ]
date                # shell and OS commands
whoami
echo Hi There
```

- First line begins with magic comment: "*#!*" (sha-bang) with shell pathname for executing the script.
- Forces specific shell to be used, which is run as a subshell.
- If "*#!*" is missing, the subshell is the same as the invoking shell for sh shells (bash) and sh is used for csh shells (tcsh).
- ***Optional -x is for debugging and prints trace of the script during execution.***

- Script can be invoked directly using a specific shell:

```
$ bash scriptfile          # direct invocation
Sat Dec 19 07:36:17 EST 2009
jfdoe
Hi There!
```

  or as a command if it has executable permissions.

```
$ chmod u+x scriptfile # Magic, make script file executable
$ ./scriptfile         # command execution
Sat Dec 19 07:36:17 EST 2009
jfdoe
Hi There!
```

- Script can have parameters.

```
#!/bin/bash  [ -x ]
date
whoami
echo ${1}              # parameter for 1st argument
```

- Arguments are passed on the command line:

```
$ ./scriptfile "Hello World"
Sat Dec 19 07:36:17 EST 2009
jfdoe
Hello World
$ ./scriptfile Hello World
Sat Dec 19 07:36:17 EST 2009
jfdoe
Hello
```

  Why no World?

- Special parameter variables to access arguments/result.
  - ${#} number of arguments, excluding script name
  - ${0} always name of shell script

    ```
    echo ${0}              # in scriptfile
    ```

    prints scriptfile.
  - ${1}, ${2}, ${3}, ... refers to arguments by position (not name), i.e., 1st, 2nd, 3rd, ... argument
  - ${*} and ${@} list all arguments, e.g., ${1} ${2} ..., excluding script name

    Difference occurs inside double quotes:
    - "${*}" arguments as a single string string, e.g., "${1} ${2} ..."
    - "${@}" arguments as separate strings, e.g., "${1}" "${2}" ...
  - ${$} process id of executing script.

◦ ${?} exit status of the last command executed; 0 often ⇒ exited normally.

```
$ cat scriptfile
#!/bin/bash
echo ${#}                # number of command-line arguments
echo ${0} ${1} ${2} ${3} ${4} # some arguments
echo "${*}"              # all arguments as a single string
echo "${@}"              # all arguments as separate strings
echo ${$}                # process id of executing subshell
exit 21                  # script exit status
```

```
$ ./scriptfile a1 a2 a3 a4 a5
5                        # number of arguments
scriptfile a1 a2 a3 a4 # script-name / args 1-4
a1 a2 a3 a4 a5           # args 1-5, 1 string
a1 a2 a3 a4 a5           # args 1-5, 5 strings
27028                    # process id of subshell
$ echo ${?}              # print script exit status
21
```

- Interactive shell session is a script reading commands from standard input.

```
$ echo ${0}         # shell you are using (not csh)
bash
```

1.9 Shift

- **shift** *[N]* : destructively shift parameters to the left N positions, i.e., ${1}=${N+1}, ${2}=${N+2}, etc., and ${#} is reduced by N.
  - If no N, 1 is assumed.
  - ***If N is 0 or greater than ${#}, there is no shift.***

$ cat scriptfile	$ ./scriptfile 1 2 3 4 5 6 7
#!/bin/bash	1
echo ${1}; **shift** 1	2
echo ${1}; **shift** 2	4
echo ${1}; **shift** 3	7
echo ${1}	

1.10 Shell Variables

- Each shell has a set of environment (global) and script (local/parameters) variables.
- variable-name syntax : [_a-zA-Z][_a-zA-Z0-9]*, “*” is wildcard qualifier
- **case-sensitive**:

  VeryLongVariableName   Page1   Income_Tax   _75

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

- Variable is declared ***dynamically*** by assigning a value with operator "=".

```
$ a1=/u/jfdoe/cs246/a1  # declare and assign
```

No spaces before or after "=".

- Variable can be removed.

```
$ unset var            # remove variable
$ echo var is ${var}
var is                 # no value for undefined variable "var"
```

- ***Variable ONLY holds string value (arbitrary length).***

```
$ i=3        # i has string value "3" not integer 3
```

- Variable's value is dereferenced using operators "$" or "${}".

```
$ echo $a1 ${a1}
/u/jfdoe/cs246/a1 /u/jfdoe/cs246/a1
$ cd $a1        # or ${a1}
```

- Dereferencing an undefined variable returns an empty string.

```
$ echo var is ${var}   # no value for undefined variable "var"
var is
$ var=Hello
$ echo var is ${var}
var is Hello
```

- ***Beware concatenation.***

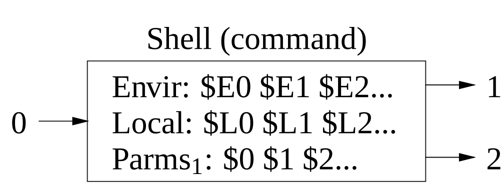
```
$ cd $a1data         # change to /u/jfdoe/cs246/a1data
```

Where does this move to?

- Always use braces to allow concatenation with other text.

```
$ cd ${a1}data       # cd /u/jfdoe/cs246/a1data
```

- Shell has 3 sets of variables: environment, local, routine parameters.



- New variable declare on the local list.

```
$ var=3              # new local variable
```

- A variable is moved to environment list if exported.

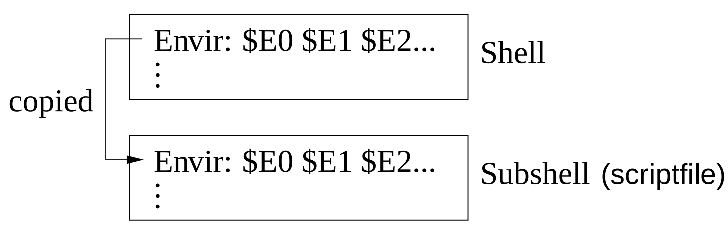
```
$ export var          # move from local to environment list
```

- Login shell starts with a number of useful environment variables, e.g.:

```
$ set                 # print variables/routines (and values)
HOME=/u/jfdoe         # home directory
HOSTNAME=linux006.student.cs   # host computer
PATH=...              # lookup directories for OS commands
SHELL=/bin/bash       # login shell
...
```

- A script executes in its own subshell with a ***copy*** of calling shell's environment variables (works across different shells), but not calling shell's locals or arguments.

```
$ ./scriptfile        # execute script in subshell
```



- When a (sub)shell ends, changes to its environment variables do not affect the containing shell (***environment variables only affect subshells***).

- ***Only put a variable in the environment list to make it accessible by subshells.***

1.11 Eval

- Unlike alias, variable can be command or argument (see page 9).

```
$ alias d=date
$ d                   # alias expands as command
Sun Apr 15 11:20:11 EDT 2012
$ d=date
$ $d                  # variable expands as command
Sun Apr 15 11:20:20 EDT 2012
$ echo d              # alias does not expand as argument
d
$ echo ${d}           # variable expands as argument
date
```

- ***Beware commands/arguments composed in variables.***

```
$ out=sortdata                # output file
$ dsls='ls | sort -r > ${out}' # store files names in descending (-r) order
$ ${dsls}                     # execute command
ls: cannot access |: No such file or directory
ls: cannot access sort: No such file or directory
ls: cannot access >: No such file or directory
ls: cannot access ${out}: No such file or directory
```

- Behaviour results because the shell tokenizes, substitutes variables, and then executes.
- Shell sees only one token, "${dsls}", so tokens ***within*** the variable are not marked correctly, e.g., "|" and ">" not marked as pipe/redirection tokens.
- Then variable substitution for "${dsls}", giving tokens
  `'ls' '|' 'sort' '-r' '>' '${out}'`
  so ls is the command and remaining tokens are file names.

  Why no "cannot access" message above for `-r`?
- Shell needs to tokenize and substitute a second time ***before*** execution.
- **eval** command causes its arguments to be processed by shell.

```
$ eval ${dsls}      # tokenize/substitute then tokenize/substitute
$ cat sortdata      # no errors, check results
...                 # list of file names in descending order
```

  - 1st tokenize/substitute gives
    eval `'ls' '|' 'sort' '-r' '>' '${out}'`
  - 2nd tokenize/substitute gives
    `'ls | sort -r > sortdata'`
    which shell executes

1.12 Arithmetic

- Arithmetic requires integers, `3 + 7`, not strings, `"3" + "17"`.

```
$ i=3+1
$ j=${i}+2
$ echo ${i} ${j}
3+1 3+1+2
```

- Arithmetic is performed by:
  - converting a string to an integer (if possible),
  - performing an integer operation,
  - and converting the integer result back to a string.

- bash performs these steps with shell-command operator $((*expression*)).

```
$ echo $((3 + 4 - 1))
6
$ echo $((3 + ${i} * 2))
9
$ echo $((3 + ${k}))            # k is unset
bash: 3 + : syntax error: operand expected (error token is " ")
```

- Basic integer operations, +, -, *, /, % (modulus), with usual precedence, and ().

- For shells without arithmetic shell-command (e.g., sh, csh), use system command expr.

```
$ `expr 3 + 4 - 1`      # for sh, csh
6
$ `expr 3 + ${i} \* 2`    # escape *
8
$ `expr 3 + ${k}`        # k is unset
expr: non-numeric argument
```

1.13 Routine

- Routine is a script in a script.

```
routine_name() {     # number of parameters depends on call
    # commands
}
```

- Invoke like a script.

```
routine_name [ args ... ]
```

- Variables/routines should be created before used.

- E.g., create a routine to print incorrect usage-message.

```
usage() {
    echo "Usage: ${0} -t -g -e input-file [ output-file ]"
    exit 1          # terminate script with non-zero exit code
}
usage               # call, no arguments
```

- Routine arguments are accessed the same as in a script.

```
$ cat scriptfile
#!/bin/bash
rtn() {
    echo ${#}        # number of command-line arguments
    echo ${0} ${1} ${2} ${3} ${4} # some arguments
    echo "${*}"      # all arguments as a single string
    echo "${@}"      # all arguments as separate strings
    echo ${$}        # process id of executing subshell
    return 17        # routine exit status
}
rtn a1 a2 a3 a4 a5  # invoke routine
echo ${?}            # print routine exit status
exit 21              # script exit status

$ ./scriptfile       # run script
5                    # number of arguments
scriptfile a1 a2 a3 a4 # script-name / args 1-5
a1 a2 a3 a4 a5       # args 1-5, 1 string
a1 a2 a3 a4 a5       # args 1-5, 5 strings
27028                # process id of subshell
17                   # routine exit status
$ echo ${?}          # print script exit status
21
```

- **source** filename : execute commands from a file in the current shell.
  - For convenience or code sharing, a script may be subdivided into multiple files.
  - E.g., put commonly used routines or set of commands into separate files.
  - No "*#!...*" at top, because not invoked directly like a script.
  - Sourcing file ***includes*** it into current shell script and ***evaluates*** lines.

    ```
    source ./aliases      # include/evaluate aliases into .shellrc file
    source ./usage.bash # include/evaluate usage routine into scriptfile
    ```

  - Created or modified variables/routines from sourced file immediately affect current shell.

1.14 Control Structures

- Shell provides control structures for conditional and iterative execution; syntax for bash is presented (csh is different).

1.14.1 Test

- **test** ([ ]) command compares strings, integers and queries files.
- **test** expression is constructed using the following:

test	operation	priority
! expr	not	high
\(expr \)	evaluation order (***must be escaped***)	
expr1 -a expr2	logical and (***not short-circuit***)	
expr1 -o expr2	logical or (***not short-circuit***)	low

- **test** comparison is performed using the following:

test	operation
string1 = string2	equal (***not ==***)
string1 != string2	not equal
integer1 -eq integer2	equal
integer1 -ne integer2	not equal
integer1 -ge integer2	greater or equal
integer1 -gt integer2	greater
integer1 -le integer2	less or equal
integer1 -lt integer2	less
-d file	exists and directory
-e file	exists
-f file	exists and regular file
-r file	exists with read permission
-w file	exists with write permission
-x file	exists with executable or searchable

- Logical operators -a (and) and -o (or) evaluate both operands.

- **test** returns 0 if expression is true and 1 otherwise (counter intuitive).

```
$ i=3
$ test 3 -lt 4              # integer test
$ echo ${?}                 # true
0
$ test `whoami` = jfdoe     # string test
$ echo ${?}                 # false
1
$ test 2 -lt ${i} -o `whoami` = jfdoe # compound test
$ echo ${?}                 # true
0
$ [ -e q1.cc ]              # file test, using brackets [] with spaces
$ echo ${?}                 # true
0
```

1.14.2 Selection

- An **if** statement provides conditional control-flow.

if test-command **then** commands **elif** test-command **then** commands ... **else** commands **fi**	**if** test-command **;** **then** commands **elif** test-command **;** **then** commands ... **else** commands **fi**

Semi-colon is necessary to separate test-command from keyword.

- test-command is evaluated; exit status of zero implies true, otherwise false.
- Check for different conditions:

```
if test "`whoami`" = "jfdoe" ; then
    echo "valid userid"
else
    echo "invalid userid"
fi

if diff file1 file2 > /dev/null ; then  # ignore diff output
    echo "same files"
else
    echo "different files"
fi

if [ -x /usr/bin/cat ] ; then          # alternate syntax for test
    echo "cat command available"
else
    echo "no cat command"
fi
```

- ***Beware unset variables or values with special characters (e.g., blanks).***

```
if [ ${var} = 'yes' ] ; then ...  # var unset => if [ = 'yes' ]
bash: [: =: unary operator expected
if [ ${var} = 'yes' ] ; then ...  # var="a b c" => if [ a b c = 'yes' ]
bash: [: too many arguments
if [ "${var}" = 'yes' ] ; then ...  # var unset => if [ "" = 'yes' ]
if [ "${var}" = 'yes' ] ; then ...  # var="a b c" => if [ "a b c" = 'yes' ]
```

When dereferencing, always quote variables, except for safe variables ${#}, ${$}, ${?}, which generate numbers.

- A **case** statement selectively executes one of N alternatives based on matching a string expression with a series of patterns (globbing), e.g.:

```
case expression in
    pattern | pattern | ... )  commands  ;;
    ...
    * )  commands ;;               # optional match anything
esac
```

- When a pattern is matched, the commands are executed up to ";;", and control exits the **case** statement.
- If no pattern is matched, the **case** statement does nothing.
- E.g., command with only one of these options:

  -h, --help, -v, --verbose, -f file, --file file

  use **case** statement to process option:

```
usage() { ... }                # print message and terminate script
verbose=no
case "${1}" in                 # process single option
  '-h' | '--help' ) usage ;;
  '-v' | '--verbose' ) verbose=yes ;;
  '-f' | '--file' )            # has additional argument
    shift 1                    # access argument
    file="${1}"
    ;;
  * ) usage ;;                 # default, has to be one argument
esac
if [ ${#} -ne 1 ] ; then usage ; fi # check only one argument remains
...                            # execute remainder of command
```

1.14.3 Looping

- **while** statement executes its commands zero or more times.

```
while test-command          while test-command ; do
  do
    commands                    commands
done                        done
```

- test-command is evaluated; exit status of zero implies true, otherwise false.
- Check for different conditions:

```
# search command-line parameters
while [ "${1}" != "" ] ; do # string compare
    echo "${1}"
    shift                      # destructive
done

# print parameters hard way, non-destructive
i=1
while [ ${i} -le ${#} ] ; do
    eval arg="\${${i}}" # 1st step ${1}, 2nd step argument 1
    echo "${arg}"              # process value
    i=$((${i} + 1))
done
```

```
# process files data1, data2, ...
i=1
file=data${i}
while [ -f "${file}" ] ; do # file regular and exists?
    ...                       # process file
    i=$((${i} + 1))           # advance to next file
    file=data${i}
done
```

- **for** statement is a specialized **while** statement for iterating with an index over list of strings.

```
for index [ in list ] ; do
    commands
done
for name in ric peter jo mike ; do
    echo ${name}
done
for arg in "${@}" ; do # process parameters, why quotes?
    echo ${arg}
done
```

Assumes **in** "${@}", if no **in** clause.

- Or over a set of values:

```
for (( init-expr; test-expr; incr-expr )) ; do  # double parenthesis
    commands
done
for (( i = 1; i <= ${#}; i += 1 )) ; do
    eval echo "\${${i}}"  # ${1-#}
done
```

- Use directly on command line (rename .ccp files to .cc):

```
$ for file in *.cpp ; do mv "${file}" "${file%cpp}"cc ; done
```

% removes matching suffix; # removes matching prefix

- A **while**/**for** loop may contain **break** and **continue** to terminate loop or advance to the next loop iteration.

```
i=1                           # process files data1, data2, ...
while [ 0 ] ; do              # while true, infinite loop
    file=data${i}             # create file name
  if [ ! -f "${file}" ] ; then break ; fi # file not exist, stop ?
    ...                       # process file
  if [ ${?} -ne 0 ] ; then continue ; fi   # bad return, next file
    ...                       # process file
    i=$((${i} + 1))           # advance to next file
done
```

1.15 Examples

- How many times does word $1 appear in file $2?

```
#!/bin/bash
cnt=0
for word in `cat $2` ; do
    if [ "${word}" = ${1} ] ; then
        cnt=$((${cnt} + 1))
    fi
done
echo ${cnt}
```

- Last Friday in any month is payday. What day is payday?

```
$ cal October 2013
    October 2013
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

  - cut selects at least 6th column `-f 6` of each line, where columns separated by blanks `-d ' '`

```
$ cal October 2013 | cut -f 6 -d ' '
2013
Fr

8
18
25
```

  - grep selects only the lines with numbers

```
$ cal October 2013 | cut -d ' ' -f 6 | grep "[0-9]"
2013
8
18
25
```

  - tail `-1` selects the last line

```
$ cal October 2013 | cut -d ' ' -f 6 | grep "[0-9]" | tail -1
25
```

  - Generalize to any month/year.

1.15.1 Hierarchical Print Script (see Section 1.7, p. 19)

```
#!/bin/bash
#
# List directories using indentation to show directory nesting
#
# Usage: hi [ -l | -d ] [ directory-name ]*
# Examples:
#   $ hi -d dir1 dir2
# Limitations
#   * does not handle file names with special characters

opt= ; files= ;
while [ ${#} -ne 0 ] ; do      # option and files in any order
    case "${1}" in
     -l) opt=l  ;;
     -d) opt=d  ;;
     -H | -help | --help | -*)
         echo 'Usage: hi [ -l | -d | -s ] directory-list . . .'
         exit 1  ;;
     *)  files="${files} ${1}"  ;;
    esac
    shift
done
case $opt in
    l) find ${files} -exec ls -ldh {} \; | sort -k9,9f | \
        sed 's|\./|  |' | sed 's|[^  /]*/|  |g' ;; # add tab then spaces
    d) du -ah ${files} | sort -k2,2f | sed 's|[^       /]*/|  |g'  ;; # replace tab
    *) find ${files} -print | sort -f | sed 's|[^/]*/|  |g'  ;;  # sort ignore case
esac
```

```
$ hi
.
   jfdoe
      cs246
         a1
            q1x.C
            q2y.h
            q2y.cc
            q3z.cpp
         a2
            q1p.C
            q2q.cc
            q2r.h
```

```
$ hi -d jfdoe
64K jfdoe
60K     cs246
28K        a1
4.0K          q1x.C
8.0K          q2y.cc
8.0K          q2y.h
4.0K          q3z.cpp
28K        a2
16K           q1p.C
4.0K          q2q.cc
4.0K          q2r.h
```

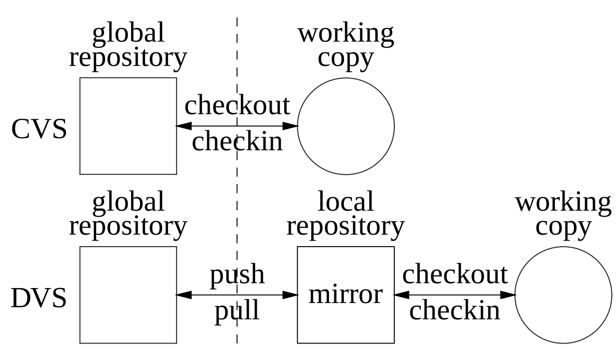
```
$ hi -l
drwx------ 3 jfdoe jfdoe 4.0K May  1 07:05 .
drwx------ 3 jfdoe jfdoe 4.0K May  1 07:05    jfdoe
drwx------ 4 jfdoe jfdoe 4.0K May  1 07:06       cs246
drwx------ 2 jfdoe jfdoe 4.0K May  1 07:32          a1
-rw------- 1 jfdoe jfdoe 3.2K May  1 07:32             q1x.C
-rw------- 1 jfdoe jfdoe 8.0K May  1 07:32             q2y.h
-rw------- 1 jfdoe jfdoe 4.1K May  1 07:32             q2y.cc
-rw------- 1 jfdoe jfdoe 160 May  1 07:32              q3z.cpp
drwx------ 2 jfdoe jfdoe 4.0K May  1 07:34          a2
-rw------- 1 jfdoe jfdoe 14K May  1 07:33              q1p.C
-rw------- 1 jfdoe jfdoe 800 May  1 07:33              q2q.cc
-rw------- 1 jfdoe jfdoe 160 May  1 07:33              q2r.h
```

1.15.2 Cleanup Script

```
#!/bin/bash
#
# List and remove unnecessary files in directories
#
# Usage: cleanup [ [ -r | R ] [ -i | f ] directory-name ]+
#   -r | -R clean specified directory and all subdirectories
#   -i | -f prompt or not prompt for each file removal
# Examples:
#   $ cleanup jfdoe
#   $ cleanup -R .
#   $ cleanup -r dir1 -i dir2 -r -f dir3
# Limitations:
#   * only removes files named: core, a.out, *.o, *.d
#   * does not handle file names with special characters

usage() {                               # print usage message & terminate
    echo "Usage: ${0} [ [ -r | -R ] [ -i | -f ] directory-name ]+"
    exit 1
}
defaults() {                            # defaults for each directory
    prompt="-i"                         # do not prompt for removal
    depth="-maxdepth 1"                 # not recursive
}
remove() {
    for file in `find "${1}" ${depth} -type f -a \( -name 'core' -o \
        -name 'a.out' -o -name '*.o' -o -name '*.d' \)`
    do
        echo "${file}"                  # print removed file
        rm "${prompt}" "${file}"
    done
}
if [ ${#} -eq 0 ] ; then usage ; fi     # no arguments ?
defaults                                # set defaults for directory
while [ ${#} -gt 0 ] ; do               # process command-line arguments
    case "${1}" in
      "-h") usage ;;                    # help ?
      "-r" | "-R") depth="" ;;          # recursive ?
      "-i" | "-f") prompt="${1}" ;; # prompt for deletion ?
      * )                               # directory name ?
        if [ ! -d "${1}" ] ; then       # directory exist ?
            echo "${1} does not exist or is not a directory";
        else
            remove "${1}"               # remove files in directory
            defaults                    # reset defaults for next directory
        fi
        ;;
    esac
    shift                               # remove argument
done
```

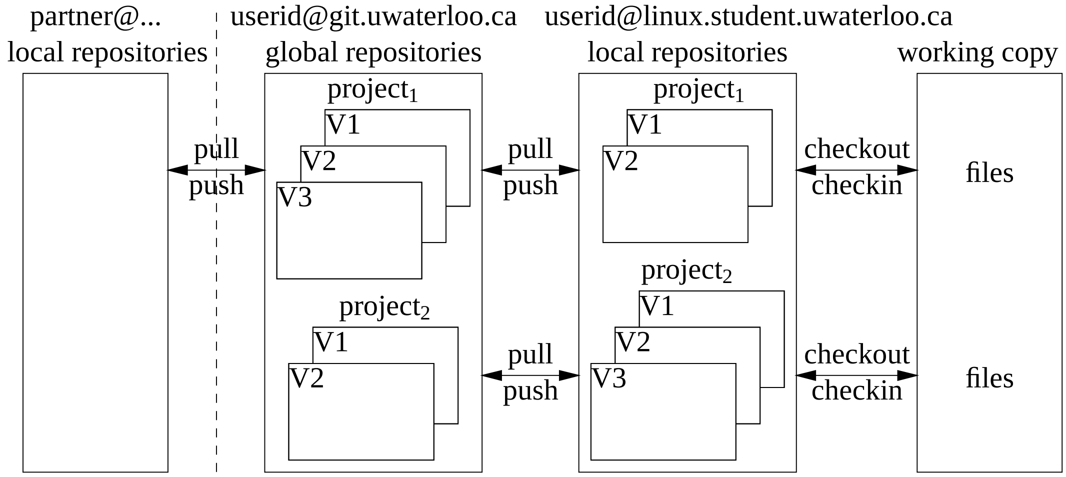
1.16 Source-Code Management

- As a program develops/matures, it changes in many ways.
  - UNIX files do not support the temporal development of a program (**version control**), i.e., history of program over time.
  - Access to older versions of a program is useful.
    - backing out of changes because of design problems
    - multiple development versions for different companies/countries/research
- Program development is often performed by multiple developers each making independent changes.
  - Sharing using files can damage file content for simultaneous writes.
  - Merging changes from different developers is tricky and time consuming.
- To solve these problems, a **source-code management-system** (SCMS) is used to provide versioning and control cooperative work.
- SCMS can provide centralized or distributed versioning (CVS)/(DVS)
  - CVS – global repository, checkout working copy
  - DVS – global repository, pull local mirror, checkout working copy



1.17 Global Repository

- **gitlab** is a global (public) repository for:
  - storing git repositories,
  - sharing repositories among students doing group project.



- Perform the following steps to setup your userid in the global repository.

  1. log into https://git.uwaterloo.ca (note https) with your WatIAm userid/password via LDAP login (top right).
  2. Click new project "+" (top right) in Dashboard to create a new project.
  3. Enter Project path, Description, and Visibilty for project.

Project path	project
Description	cs246 group project with Jane
Visibility level	⋆ Private

     ◦ Visibility level Private (default) is used for course work, and restricts access to you and any granted guests.
  4. Click Create project (at bottom).
  5. Click HTTPS (middle) to get project url. (SSH is default.)
  6. Instructions are provided under Git global setup to setup the local project-repository to check in/out versions from the global repository.

- Click logout "⇒" (top right) in Dashboard to logout.

1.17.1 Project Member

- To add a partner to your project in gitlab

  1. sign onto Gitlab
  2. Click on project name on left
  3. Click on edit (bottom icon on left side-bar menu) (more icons appear)
  4. Click on team (3rd icon)
  5. Click on New project member (top right)
  6. Click in People box and select your partner from drop-down list of gitlab users
  7. Click in Project Access and select Developer (allows push/pull to your project)
  8. Click on Add User

1.18 Local Repository

- **git** is a distributed source-code management-system using the **copy-modify-merge** model.
  - master copy of all **project** files kept in a **global repository**,
  - multiple versions of the project files managed in the repository,
  - developers **pull** a local copy (mirror) of the global repository for modification,
  - developers change working copy and commit changes to local repository,
  - developers **push** committed changes from local repository with integration using **text merging**.

  Git works on file content not file time-stamps.

- config : registering.

```
$ git config --global user.name "Jane F Doe"
$ git config --global user.email jfdoe@uwaterloo.ca
$ git config --list
Jane F Doe
jfdoe@uwaterloo.ca
...
```

  - creates hidden file .gitconfig in home directory

```
$ cat ~/.gitconfig
[user]
    name = Jane F Doe
    email = jfdoe@uwaterloo.ca
```

- create local repository directory and change to that directory.

```
$ mkdir project
$ cd project
```

- init : create and initialize a repository.

```
$ git init    # create empty git repository or reinitialize existing one
Initialized empty Git repository in /u/jfdoe/project/.git/
$ ls -aF
./  ../  .git/
```

  - creates hidden directory .git to store local repository information.

- remote : connect local with global repository

```
$ git remote add origin gitlab@git.uwaterloo.ca:jfdoe/project.git
```

  If mistake typing **origin** URL, remove and add again.

```
$ git remote rm origin
```

- status : compare working copy structure with local repository

```
$ git status
# On branch master
#
# Initial commit
nothing to commit (create/copy files and use "git add" to track)
```

- add : add file contents to index

```
$ emacs README.md    # project description
$ git add README.md
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#    new file:    README.md
```

  - By convention, file README.md contains short description of project.
  - If project directory exists, create README.md and add existing files.
  - ***Addition only occurs on next commit.***
  - Forgetting git add for new files is a common mistake.
  - After project starts, joining developers ***clone*** the existing project.

    ```
    $ git clone gitlab@git.uwaterloo.ca:jfdoe/project.git
    ```

  - ***Add only source files into repository***, e.g., *.o, *.d, a.out, do not need to be versioned.
  - Create hidden file .gitignore in projct directory and list working files not versioned.

    ```
    # ignore self
    .gitignore
    # build files
    *.[do]
    a.out
    ```

- commit : record changes to local repository

```
$ git commit -a -m "initial commit"
[master (root-commit) e025356] initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
$ git status
# On branch master
nothing to commit (working directory clean)
```

  - -a (all) automatically stage modified and deleted files

- -m (message) flag documents repository change.
- if no -m (message) flag specified, prompts for documentation (using an editor if shell environment variable EDITOR set).

- push : record changes to global repository

```
$ git push -u origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 234 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To gitlab@git.uwaterloo.ca:jfdoe/project.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

  - Options **-u origin master** only required for first push of newly created repository.
  - Subsequently, use **git push** with no options.
  - ***Always make sure your code compiles and runs before pushing;*** it is unfair to pollute a shared global-repository with bugs.

- pull : record changes from global repository

```
$ git pull
```

Developers must periodically ***pull*** the latest global version to local repository.

1.18.1 Modifying

- Editted files in working copy are implicitly ***scheduled*** for update on next commit.

```
$ emacs README.md # modify project description
```

- Add more files.

```
$ mkdir gizmo
$ cd gizmo
$ emacs Makefile x.h x.cc y.h y.cc   # add text into these files
$ ls -aF
./  ../  Makefile  x.cc  x.h  y.cc  y.h
$ git add Makefile  x.cc  x.h  y.cc  y.h
```

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    new file:   Makefile
#    new file:   x.cc
#    new file:   x.h
#    new file:   y.cc
#    new file:   y.h
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#    modified:   ../README.md
```

- Commit change/adds to local repository.

```
$ git commit -a -m "update README.md, add gizmo files"
[master 0f4162d] update README.md, add gizmo files
 6 files changed, 6 insertions(+)
 create mode 100644 gizmo/Makefile
 create mode 100644 gizmo/x.cc
 create mode 100644 gizmo/x.h
 create mode 100644 gizmo/y.cc
 create mode 100644 gizmo/y.h
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# nothing to commit (working directory clean)
```

- rm removes files from **BOTH local directory and repository**.

```
$ git rm x.*                    # globbing allowed
rm 'gizmo/x.cc'
rm 'gizmo/x.h'
$ ls -aF
./  ../  Makefile  y.cc  y.h
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    deleted:    x.cc
#    deleted:    x.h
```

- And update a file.

```
$ emacs y.cc        # modify y.cc
```

- Possible to revert state of working copy ***BEFORE*** commit.

```
$ git checkout HEAD x.cc x.h y.cc  # cannot use globbing
$ ls -aF
./  ../  Makefile  x.cc  x.h  y.cc  y.h
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# nothing to commit (working directory clean)
```

  - **HEAD** is symbolic name for last commit, titled "update README.md, add gizmo files".

- Possible to revert state ***AFTER*** commit by accessing files in previous commits.

```
$ git commit -a -m "remove x files, update y.cc"
[master ecfbac4] remove x files, update y.cc
 3 files changed, 1 insertion(+), 2 deletions(-)
 delete mode 100644 gizmo/x.cc
 delete mode 100644 gizmo/x.h
```

- Revert state from previous commit.

```
$ git checkout HEAD~1 x.cc x.h y.cc
$ ls -aF
./  ../  Makefile  x.cc  x.h  y.cc  y.h
```

  - **HEAD~1** means one before current commit (subtract 1), titled "update README.md, add gizmo files".

- Check log for revision number.

```
$ git log
commit ecfbac4b80a2bf5e8141bddfdd2eef2f2dcda799
Author: Jane F Doe <jfdoe@uwaterloo.ca>
Date:   Sat May 2 07:30:17 2015 -0400

    remove x files, update y.cc

commit 0f4162d3a95a2e0334964f95495a079341d4eaa4
Author: Jane F Doe <jfdoe@uwaterloo.ca>
Date:   Sat May 2 07:24:40 2015 -0400

    update README.md, add gizmo files

commit e025356c6d5eb2004314d54d373917a89afea1ab
Author: Jane F Doe <jfdoe@uwaterloo.ca>
Date:   Sat May 2 07:04:10 2015 -0400

    initial commit
```

- Count top to bottom for relative commit number, or use a commit name ecfbac4b80a2bf5e8141bddfdd2eef2f2dcda799.

- Commit restored files into local repository.

```
$ git commit -a -m "undo file changes"
[master 265d6f8] undo files changes
 3 files changed, 2 insertions(+), 1 deletion(-)
 create mode 100644 gizmo/x.cc
 create mode 100644 gizmo/x.h
```

- git mv renames file in BOTH local directory and repository.

```
$ git mv x.h w.h
$ git mv x.cc w.cc
$ ls -aF
./  ../  Makefile  w.cc  w.h  y.cc  y.h
```

- Copy files in the repository by copying working-copy files and add them.

```
$ cp y.h z.h
$ cp y.cc z.cc
$ git add z.*
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 3 commits.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    renamed:    x.cc -> w.cc
#    renamed:    x.h -> w.h
#    new file:   z.cc
#    new file:   z.h
```

- Commit changes in local repository.

```
$ git commit -a -m "renaming and copying"
[master 0c5f473] renaming and copying
 4 files changed, 2 insertions(+)
 rename gizmo/{x.cc => w.cc} (100%)
 rename gizmo/{x.h => w.h} (100%)
 create mode 100644 gizmo/z.cc
 create mode 100644 gizmo/z.h
```

- Push changes to global repository.

```
$ git push
Counting objects: 19, done.
Delta compression using up to 48 threads.
Compressing objects: 100% (10/10), done.
Writing objects: 100% (17/17), 1.34 KiB, done.
Total 17 (delta 1), reused 0 (delta 0)
To gitlab@git.uwaterloo.ca:jfdoe/project.git
   e025356..0c5f473  master -> master
Branch master set up to track remote branch master from origin.
```

1.18.2 Conflicts

- When multiple developers work on ***SAME*** files, source-code conflicts occur.

jfdoe	kdsmith
modify y.cc	modify y.cc
	remove y.h
	add t.cc

- Assume kdsmith commits and pushes changes.
- jfdoe commits change and attempts push.

```
$ git push
To gitlab@git.uwaterloo.ca:jfdoe/project.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'gitlab@git.uwaterloo.ca:jfdoe/project.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' section of 'git push --help' for details.
```

- Resolve differences between local and global repository by pulling.

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 5 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From git.uwaterloo.ca:jfdoe/project
   2a49710..991683a  master     -> origin/master
Removing gizmo/y.h
Auto-merging gizmo/y.cc
CONFLICT (content): Merge conflict in gizmo/y.cc
Automatic merge failed; fix conflicts and then commit the result.
```

- File y.h is deleted, t.cc added, but conflict reported in gizmo/y.cc

```
$ ls -aF
./  ../  Makefile  t.cc  w.cc  w.h  y.cc  z.cc  z.h
$ cat y.cc
<<<<<<< HEAD
I like file y.cc
=======
This is file y.cc
>>>>>>> 5d89df953499a8fdfd6bc92fa3a6be9c8358dbd1
```

- Chevrons "<" bracket conflicts throughout the file.
- jfdoe can resolve by reverting their change or getting kdsmith to revert their change.
- No further push allowed by jfdoe until conflict is resolved but can continue to work in local repository.
- jfdoe decides to revert to kdsmith version.

  ```
  $ cat y.cc
  This is file y.cc
  ```

  and commits the reversion so pull now indicates up-to-date.

  ```
  $ git commit -a -m revert y.cc
  [master 5497d17] revert y.cc
  $ git pull
  Already up-to-date.
  ```

- Conflict resolution tools exist to help with complex conflicts (unnecessary for this course).

1.19 Testing

- A major phase in program development is testing (> 50%).
- This phase often requires more time and effort than design and coding phases combined.
- Testing is not debugging.
- **Testing** is the process of "executing" a program with the intent of determining differences between the specification and actual results.
  - Good test is one with a high probability of finding a difference.
  - Successful test is one that finds a difference.
- Debugging is the process of determining why a program does not have an intended testing behaviour and correcting it.

1.19.1 Human Testing

- **Human Testing** : systematic examination of program to discover problems.
- Studies show 30–70% of logic design and coding errors can be detected in this manner.
- **Code inspection** team of 3-6 people led by moderator (team leader) looking for problems, often “grilling” the developer(s):
  - data errors: wrong types, mixed mode, overflow, zero divide, bad subscript, initialization problems, poor data-structure
  - logic errors: comparison problems (== / !=, < / <=), loop initialization / termination, off-by-one errors, boundary values, incorrect formula, end of file, incorrect output
  - interface errors: missing members or member parameters, encapsulation / abstraction issues
- **Walkthrough** : less formal examination of program, possibly only 2-3 developers.
- **Desk checking** : single person “plays computer”, executing program by hand.

1.19.2 Machine Testing

- **Machine Testing** : systematic running of program using test data designed to discover problems.
  - speed up testing, occur more frequently, improve testing coverage, greater consistency and reliability, use less people-time testing
- Commercial products are available.
- Should be done after human testing.
- Exhaustive testing is usually impractical (too many cases).
- **Test-case design** involves determining subset of all possible test cases with the highest probability of detecting the greatest number of errors.
- Two major approaches:
  - **Black-Box Testing** : program’s design / implementation is unknown when test cases are drawn up.
  - **White-Box Testing** : program’s design / implementation is used to develop the test cases.
  - **Gray-Box Testing** : only partial knowledge of program’s design / implementation know when test cases are drawn up.
- Start with the black-box approach and supplement with white-box tests.
- Black-Box Testing

- ◦ **equivalence partitioning** : completeness without redundancy
  - ∗ 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 below, on, and above boundary cases

```
39, 40, 41  (hours)     valid cases
44, 45, 46     "
 0,  1,  2     "
-2, -1,  0     "        invalid cases
59, 60, 61     "
```

- ◦ **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 to cover (exercise) important logic paths through program
  - ◦ try to test every decision alternative at least once
  - ◦ test all combinations of decisions (often impossible due to size)
  - ◦ test every routine and member for each type
  - ◦ cannot test all permutations and combinations of execution
- **Test Harness** : a collection of software and test data configured to run a program (unit) under varying conditions and monitor its outputs.

1.19.3 Testing Strategies

- **Unit Testing** : test each routine/class/module separately before integrated into, and tested with, entire program.
  - ◦ requires construction of drivers to call the unit and pass it test values
  - ◦ requires construction of stub units to simulate the units called during testing
  - ◦ allows a greater number of tests to be carried out in parallel
- **Integration Testing** : test if units work together as intended.
  - ◦ after each unit is tested, integrate it with tested system.

  - done top-down or bottom-up : higher-level code is drivers, lower-level code is stubs
  - In practice, a combination of top-down and bottom-up testing is usually used.
  - detects interfacing problems earlier

- Once system is integrated:
  - **Functional Testing** : test if performs function correctly.
  - **Regression Testing** : test if new changes produce different effects from previous version of the system (diff results of old / new versions).
  - **System Testing** : test if program complies with its specifications.
  - **Performance Testing** : test if program achieves speed and throughput requirements.
  - **Volume Testing** : test if program handles difference volumes of test data (small $\Leftrightarrow$ large), possibly over long period of time.
  - **Stress Testing** : test if program handles extreme volumes of data over a short period of time with fixed resources, 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.
  - **Acceptance Testing** : does the system satisfies what the client ordered.
- If a problem is discovered, make up additional test cases to zero in on the issue and ultimately add these tests to the test suite for regression testing.

1.19.4 Tester

- A program should not be tested by its writer, but in practice this often occurs.
- Remember, the tester only tests what *they* think it should do.
- Any misunderstandings the writer had while coding the program are carried over into testing.
- Ultimately, any system must be tested by the client to determine if it is acceptable.
- Points to the need for a clear specification to protect both the client and developer.

2 C++

2.1 C/C++ Composition

- C++ is composed of 4 languages:

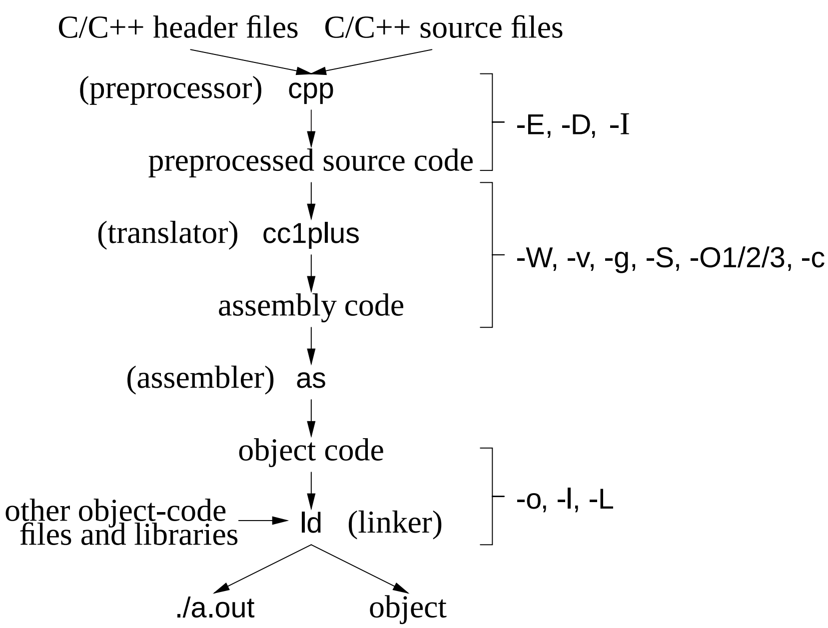
  1. preprocessor language (cpp) modifies (text-edits) the program ***before*** compilation (see Section 2.17, p. 94).
  2. template (generic) language adds new types and routines ***during*** compilation (see Section 2.36, p. 164).
  3. C programming language specifying basic declarations and control flow to be executed ***after*** compilation.
  4. 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 → **preprocessor edits** → **templates expand** → **compilation**
  (→ linking/loading → execution)

- C is composed of languages 1 & 3.

- The compiler interface controls all of these steps.



2.2 First Program

Java	C	C++
import java.lang.*; // *implicit* **class** Hello { **public static** **void** main(String[] args) { System.out.println("Hello!"); System.exit(0); } }	**#include <stdio.h>** **int** main() { **printf("Hello!\n");** **return** 0; }	**#include <iostream>** // *access to output* **using namespace std;** // *direct naming* **int** main() { // *program starts here* **cout << "Hello!" << endl;** **return** 0; // *return 0 to shell, optional* }

- **#include** <iostream> copies (imports) basic I/O descriptions (no equivalent in Java).
- **using namespace** std allows imported I/O names to be accessed directly (otherwise qualification is necessary, see Section 2.37, p. 173).
- cout << "Hello!" << endl prints "Hello!" to standard output, called cout (System.out in Java, stdout in C).
- endl starts a newline after "Hello!" (println in Java, '\n' in C).
- Routine exit (Java System.exit) terminates a program at any location and returns a code to the shell, e.g., exit(0) (**#include** <cstdlib>).
  - Literals EXIT_SUCCESS and EXIT_FAILURE indicate successful or unsuccessful termination status.
  - e.g., **return** EXIT_SUCCESS or exit(EXIT_FAILURE).
- C program-files use suffix .c; C++ program-files use suffixes .C / .cpp / **.cc**.
- Compile with g++ command:

```
$ g++ -Wall -g -std=c++11 firstprog.cc -o firstprog  # compile, create "a.out"
$ ./firstprog                                          # execute program
```

- -W***kind*** generate warning message for this "***kind***" of situation.
  - -W**all** print ALL warning messages.
  - -W**error** make warnings into errors so program does not compile.
- -g add symbol-table information to object file for debugger
- -std=**c++11** allow new C++11 extensions (requires gcc-4.8.0 or greater)
- -o file containing the executable (a.out default)
- set shell alias for g++ to alway use options **g++-4.9 -Wall -g -std=c++11**

2.3 Comment

- Comment may appear where whitespace (space, tab, newline) is allowed.

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

- **/*. . .*/ comment cannot be nested**:

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

- Be extremely careful in using this comment to elide/comment-out code. (page 97 presents another way to comment-out code.)

2.4 Declaration

- A declaration introduces names or redeclares names from previous declarations.

2.4.1 Basic Types

Java	C / C++	
boolean	**bool** (C <stdbool.h>)	
char	**char** / **wchar_t**	ASCII / unicode character
byte	**char** / **wchar_t**	integral types
int	**int**	
float	**float**	real-floating types
double	**double**	
		label type, implicit

- C/C++ treat **char** / **wchar_t** as character and integral type.
- Java types **short** and **long** are created using type qualifiers (see Section 2.4.3).

2.4.2 Variable Declaration

- C/C++ declaration: type followed by list of identifiers, except label with an implicit type (same in Java).

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

- Declarations may have an initializing assignment:

```
int i = 3;          int i = 3, j = i, k = f( j );
int j = 4 + i;
int k = f( j );
```

  C restricts initializer elements to be constant for external declarations.

2.4.3 Type Qualifier

- Other integral types are composed with type qualifiers modifying integral types **char** and **int**.
- C/C++ provide size (**short**, **long**) and signed-ness (**signed** ⇒ positive/negative, **unsigned** ⇒ positive only) qualifiers.
- **int** provides ***relative*** machine-specific types: usually **int** $\geq$ 4 bytes for 32/64-bit computer, **long** $\geq$ **int**, **long long** $\geq$ **long**.
- **#include** <climits> specifies names for lower/upper bounds of a type's range of values for a machine, e.g., a 32/64-bit computer:

integral types	range (lower/upper bound name)
char (**signed char**)	SCHAR_MIN to SCHAR_MAX, e.g., -128 to 127
unsigned char	0 to UCHAR_MAX, e.g. 0 to 255
short (**signed short int**)	SHRT_MIN to SHRT_MAX, e.g., -32768 to 32767
unsigned short (**unsigned short int**)	0 to USHRT_MAX, e.g., 0 to 65535
int (**signed int**)	INT_MIN to INT_MAX, e.g.,-2147483648 to 2147483647
unsigned int	0 to UINT_MAX, e.g., 0 to 4294967295
long (**signed long int**)	(LONG_MIN to LONG_MAX), e.g., -2147483648 to 2147483647
unsigned long (**unsigned long int**)	0 to (ULONG_MAX, e.g. 0 to 4294967295
long long (**signed long long int**)	LLONG_MIN to LLONG_MAX, e.g.,-9223372036854775808 to 9223372036854775807
unsigned long long (**unsigned long long int**)	0 to (ULLONG_MAX), e.g., 0 to 18446744073709551615

- C/C++ provide two basic real-floating types **float** and **double**, and one real-floating type generated with type qualifier.
- **#include** <cfloat> specifies names for precision and magnitude of real-floating values.

real-float types	range (precision, magnitude)
float	FLT_DIG precision, FLT_MIN_10_EXP to FLT_MAX_10_EXP, e.g,. 6+ digits over range 10^{-38} to 10^{38}, IEEE (4 bytes)
double	DBL_DIG precision, DBL_MIN_10_EXP to DBL_MAX_10_EXP, e.g., 15+ digits over range 10^{-308} to 10^{308}, IEEE (8 bytes)
long double	LDBL_DIG precision, LDBL_MIN_10_EXP to LDBL_MAX_10_EXP, e.g., 18-33+ digits over range 10^{-4932} to 10^{4932}, IEEE (12-16 bytes)

float : ±1.17549435e-38 to ±3.40282347e+38
double : ±2.2250738585072014e-308 to ±1.7976931348623157e+308
long double : ±3.36210314311209350626e-4932 to ±1.18973149535723176502e+4932

2.4.4 Literals

- Variables contain values; values have **constant** (C) or **literal** (C++) meaning.

```
3 = 7;    // disallowed
```

- C/C++ and Java share almost all the same literals for the basic types.

type	literals
boolean	**false**, **true**
character	'a', 'b', 'c'
string	"abc", "a b c"
integral	decimal : 123, -456, 123456789 octal, prefix 0 : 0144, -045, 04576132 hexadecimal, prefix 0X / 0x : 0xfe, -0X1f, 0xe89abc3d
real-floating	.1, 1., -1., 0.52, -7.3E3, -6.6e-2, E/e exponent

- Use the right literal for a variable's type:

```
bool b = true;              // not 1
int i = 1;                  // not 1.0
double d = 1.0              // not 1
char c = 'a';               // not 97
const char *cs = "a";       // not 'a'
```

- Juxtaposed string literals are concatenated.

```
"John" "Doe";  // divide literal for readability
"JohnDoe";
```

- Every string literal is implicitly terminated with a character '\0'.
  - "abc" is actually 4 characters: 'a', 'b', 'c', and '\0', which occupies 4 bytes.
- Zero value is a **sentinel** used by C-string routines to locate the string end.
- Drawbacks:
  - String cannot contain a character with the value '\0'.
  - Computing string length requires $O(N)$ search for '\0'.
- Escape sequence provides quoting of special characters in a character/string literal using a backslash, \.

`'\\'`	backslash
`'\''`	single quote
`'\"'`	double quote
`'\t'`, `'\n'`	(special names) 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 for **char**, up to 4 hexadecimal digits for **wchar_t** (not Java)

- Octal digits terminated by length (3) or first character not octal digit.
- Hex digits arbitrarily long, but value truncated to fit character type.
- How many characters? `"\0123\128\xaaa\xaw"`
- C/C++ provide user literals (write-once/read-only) with type qualifier **const** (Java **final**).

Java	C/C++
final char Initial = 'D'; **final short int** Size = 3, SupSize; SupSize = Size + 7; **final double** PI = 3.14159;	**const char** Initial = 'D'; **const short int** Size = 3, SupSize = Size + 7; **disallowed** **const double** PI = 3.14159;

- C/C++ **const** variable ***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 (only) appear in contexts where a literal can appear.

```
Size = 7;    // disallowed
```

- **Good practise is to name literals so all usages can be changed via its initialization value.** (see Section 2.10.1, p. 76)

```
const short int Mon=0,Tue=1,Wed=2,Thu=3,Fri=4,Sat=5,Sun=6;
```

2.4.5 String

- **string** (**#include** <string>) is a sequence of characters with powerful operations performing actions on groups of characters.
- C provided strings by an array of **char**, string literals, and library facilities.

```
char s[10];        // string of at most 10 characters
```

- Because C-string variable is fixed-sized array:
  - management of variable-sized strings is the programmer's responsibility,
  - requiring complex storage management.

- C++ solves these problems by providing a "string" type:
  - maintaining string length versus sentinel character '\0',
  - managing storage for variable-sized string.

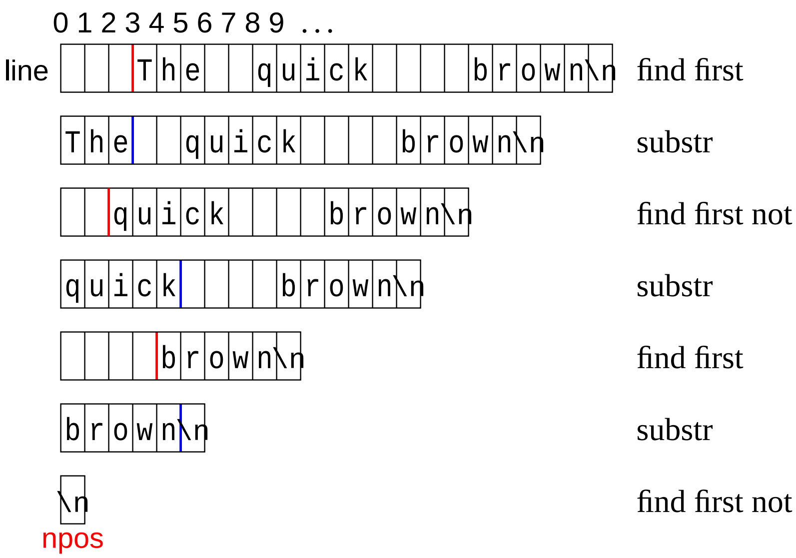
Java String	**C char** [ ]	**C++** string
	strcpy, strncpy	=
+, concat	strcat, strncat	+
equal, compareTo	strcmp, strncmp	==, !=, <, <=, >, >=
length	strlen	length
charAt	[ ]	[ ]
substring		substr
replace		replace
indexOf, lastIndexOf	strstr	find, rfind
	strcspn	find_first_of, find_last_of
	strspn	find_first_not_of,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

- find routines return value string::npos of type string::size_type, if unsuccessful search.
- c_str converts a string to a **char** * pointer ('\0' terminated).

```
string a, b, c;         // declare string variables
cin >> c;               // read white-space delimited sequence of characters
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 literal
string d = c.substr(2,3); // extract starting at position 2 (zero origin) for length 3, d is "cax"
c.replace(2,1,d); // replace starting at position 2 for length 1 and insert d, c is "abcaxaxc"
string::size_type p = c.find( "ax" ); // search for 1st occurrence of string "ax", p is 3
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 is 1
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), p is 7
```

- **Note different call syntax c.substr(2, 3) versus substr(c, 2, 3)** (see Section 2.21, p. 105).
- Count and print words in string line containing words separated by whitespace.

```
unsigned int count = 0;
string line, alpha = "abcdefghijklmnopqrstuvwxyz"
                     "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
... // line is initialized with text
line += "\n";                          // add newline as sentinel
for ( ;; ) {                           // scan words off line
    // find position of 1st alphabetic character
    string::size_type posn = line.find_first_of( alpha );
  if ( posn == string::npos ) break; // any characters left ?
    line = line.substr( posn );        // remove leading whitespace
    // find position of 1st non-alphabetic character
    posn = line.find_first_not_of( alpha );
    // extract word from start of line
    cout << line.substr( 0, posn ) << endl; // print word
    count += 1;                        // count words
    line = line.substr( posn );        // delete word from line
} // for
```



- Contrast C and C++ style strings (note, management of string storage):

```
#include <string>          // C++ string routines
using namespace std;
#include <string.h>        // C string routines
int main() {
    // C++ string
    const string X = "abc", Y = "def", Z = "ghi";
    string S = X + Y + Z;
    // C string
    const char *x = "abc", *y = "def", *z = "ghi";
    char s[strlen(x)+strlen(y)+strlen(z)+1]; // pre-compute size
    strcpy( s, "" ); // initialize to null string
    strcat( strcat( strcat( s, x ), y ), z );
}
```

Why “+1” for dimension of s?

- **Good practice is NOT to iterate through the characters of a string variable!**

2.5 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.
  2. **Unformatted I/O** transfers data without conversion, e.g., internal integer and real-floating values.

2.5.1 Formatted I/O

Java	C	C++
import java.io.*; import java.util.Scanner;	**#include** <stdio.h>	**#include** <iostream>
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, in.close()
out.close()	close(out)	scope ends, out.close()
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.fail()
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**, and values are conversed based on the type of variables and format codes.

- C++ has three implicit stream files: cin, cout and cerr, which are implicitly declared and opened (Java has in, out and err).

- C has stdin, stdout and stderr, which are implicitly declared and opened.

- **#include** <iostream> imports all necessary declarations to access 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
```

- File types, ifstream/ofstream, indicate whether the file can be read or written.
- File-name type, "myinfile"/"myoutfile", is **char** * (**not string**, see page 59).
- Declaration **open**s an operating-system file making it accessible through the variable name:
  - infile reads from file myinfile
  - outfile writes to file myoutfile

  where both files are located in the directory where the program is run.
- 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.

```
if ( infile.fail() ) ... // open failed, print message and exit
if ( outfile.fail() ) ... // open failed, print message and exit
```

- C++ I/O library overloads (see Section 2.15, p. 89) 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.
- Both I/O libraries can cascade multiple I/O operations, i.e., input or output multiple values in a single expression.

2.5.1.1 Formats

- Format of input/output values is controlled via **manipulators** defined in **#include** <iomanip>.

oct (%o)	integral values in octal
dec (%d)	integral values in decimal
hex (%x)	integral values in hexadecimal
left / right (default)	values with padding after / before values
boolalpha / noboolalpha (default)	bool values as false/true instead of 0/1
showbase / noshowbase (default)	values with / without prefix 0 for octal & 0x for hex
showpoint / noshowpoint (default)	print decimal point if no fraction
fixed (default) / scientific	float-point values without / with exponent
setfill('ch')	padding character before/after value (default blank)
setw(W) (%Wd)	NEXT VALUE ONLY in minimum of W columns
setprecision(P) (%.Pf)	fraction of float-point values in maximum of P columns
endl (\n)	flush output buffer and start new line (**output only**)
skipws (default) / noskipws	skip whitespace characters (**input only**)

- **Manipulators are not variables for input/output**, but control I/O formatting for all literals/variables after it, continuing 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'`, as `'\n'` does not flush buffered data.
- For input, skipsw/noskipws toggle between ignoring whitespace between input tokens and reading whitespace (i.e., tokenize versus raw input).

2.5.1.2 Input

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

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

- Input values for a stream file are C/C++ literals: `3`, `3.5e-1`, etc., separated by whitespace.
- Except for characters and character strings, ***which are not in quotes***.
- Type of operand indicates the kind of literal expected in the stream
  - e.g., an integer operand means an integer literal is expected.
- Input starts reading where the last input left off, and scans lines to obtain necessary number of literals.
  - Hence, placement of input values on lines of a file is often arbitrary.
- C/C++ must attempt to read ***before*** end-of-file is set and can be tested.
- **End of file** is the detection of the physical end of a file; **there is no end-of-file character**.
- In shell, typing <ctrl>-d (C-d), i.e., press <ctrl> and d keys simultaneously, causes shell to close current input file marking its physical end.
- In C++, end of file can be explicitly detected in two ways:
  - stream member eof returns **true** if the end of file is reached and **false** otherwise.
  - stream member fail returns **true** for invalid literal OR no literal if end of file is reached, and **false** otherwise.
- Safer to check fail and then check eof.

```
for ( ;; ) {
    cin >> i;
  if ( cin.eof() ) break;    // should use "fail()"
    cout << i << endl;
}
```

- If `"abc"` is entered (invalid integer literal), fail becomes **true** but eof is **false**.
- Generates infinite loop as invalid data is not skipped for subsequent reads.
- Stream has coercion to **void** $*$: if fail(), null pointer; otherwise non-null.

```
cout << cin;              // print fail() status of stream cin
while ( cin >> i ) . . .  // read and check pointer to != 0
```

  - results in side-effects in the expression (changing variable i)
  - precludes analysis of the input stream (cin) without code duplication

```
while ( cin >> i ) {
    cout << cin.good() << endl;

    . . .
}
cout << cin.good() << endl;
```

```
for ( ;; ) {
    cin >> i;
    cout << cin.good() << endl;
  if ( cin.fail() ) break;
    . . .
}
```

- After an unsuccessful read, clear() resets the stream.

```
#include <iostream>
#include <limits>                  // numeric_limits
using namespace std;
int main() {
    int n;
    cout << showbase;              // prefix hex with 0x
    cin >> hex;                    // input hex literals
    for ( ;; ) {
        cout << "Enter hexadecimal number: ";
        cin >> n;
        if ( cin.fail() ) {        // problem ?
      if ( cin.eof() ) break;      // eof ?
            cout << "Invalid hexadecimal number" << endl;
            cin.clear();           // reset stream failure
            cin.ignore( numeric_limits<int>::max(),'\n'); // skip until newline
        } else {
            cout << hex << "hex:" << n << dec << " dec:" << n << endl;
        }
    }
    cout << endl;
}
```

- ignore skips *n* characters, e.g., cin.ignore(5) or until a specified character.

- getline(stream, string, **char**) reads strings with white spaces allowing different delimiting characters (no buffer overflow):

```
getline( cin, c, ' ' ); // read characters until ' ' => cin >> c
getline( cin, c, '@' ); // read characters until '@'
getline( cin, c, '\n' ); // read characters until newline (default)
```

- Read in file-names, which may contain spaces, and process each file:

```
#include <fstream>
using namespace std;
int main() {
    ifstream fileNames( "fileNames" );  // requires char * argument
    string fileName;
    for ( ;; ) {                        // process each file
        getline( fileNames, fileName ); // may contain spaces
      if ( fileNames.fail() ) break;    // handle no terminating newline
        ifstream file( fileName.c_str() ); // access char *
        // read and process file
    }
}
```

- In C, routine feof returns **true** when eof is reached and fscanf returns EOF.

- Parameters in C are always passed by value (see Section 2.14.1, p. 87), so arguments to fscanf must be preceded with & (except arrays) so they can be changed.

2.5.1.3 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 has a list of formats and values, and output operation generates strings:

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

- 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.**

- Use manipulators to generate specific output formats:

```
#include <iostream>     // cin, cout, cerr
#include <iomanip>      // manipulators
using namespace std;
int i = 7; double r = 2.5; char c = 'z'; const 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
```

2.5.2 Unformatted I/O

- Expensive to convert from internal (computer) to external (human) forms (bits ⇔ characters).

- When data does not have to be seen by a human, use efficient unformatted I/O so no conversions.

- Uses same mechanisms as formatted I/O to connect variable to file (open/close).

- read and write routines directly transfer bytes from/to a file, where each takes a pointer to the data and number of bytes of data.

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

- Read/write of types other than characters requires a coercion cast (see Section 2.6.2, p. 63) or C++ **reinterpret_cast**.

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

int main() {
    ofstream outfile( "myfile" );     // open output file "myfile"
    if ( outfile.fail() ) . . .       // unsuccessful open ?
    double d = 3.0;
    outfile.write( (char *)&d, sizeof( d ) );  // coercion
    outfile.close();                  // close file before attempting read

    ifstream infile( "myfile" );      // open input file "myfile"
    if ( infile.fail() ) . . .        // unsuccessful open ?
    double e;
    infile.read( reinterpret_cast<char *>(&e), sizeof( e ) );  // coercion
    if ( d != e ) . . .               // problem
    infile.close();
}
```

- Coercion is unnecessary if buffer type was **void** *.

2.6 Expression

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

- 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
```

- **Beware of overflow.**

```
unsigned int a = 4294967295, b = 4294967295, c = 4294967295;
(a + b) / c;      // => 0 as a+b overflows leaving zero
a / c + b / c;    // => 2
```

Perform divides *before* multiplies (if possible) to keep numbers small.

- C++ relational/equality return **false**/**true**; C return 0/1.
- Referencing (address-of), &, and dereference, *, operators (see Section 2.10.2, p. 77) do not exist in Java because access to storage is restricted.
- General assignment operators only evaluate left-hand side (lhs) once:

```
v[ f(3) ] += 1;              // only calls f once
v[ f(3) ] = v[ f(3) ] + 1;   // calls f twice
```

- Bit-shift operators, << (left), and >> (right) shift bits in integral variables left and right.
  - left shift is multiplying by 2, modulus variable's size;
  - right shift is dividing by 2 if unsigned or positive (like Java >>>); otherwise undefined.

```
int x, y, z;
x = y = z = 1;
cout << (x << 1) << ' ' << (y << 2) << ' ' << (z << 3) << endl;
x = y = z = 16;
cout << (x >> 1) << ' ' << (y >> 2) << ' ' << (z >> 3) << endl;
2 4 8
8 4 2
```

Why are parenthesis necessary?

2.6.1 Conversion

- **Conversion** transforms a value to another type by changing the value to the new type's representation (see Section 2.21.3.2, p. 110).
- Conversions occur implicitly by compiler or explicitly by programmer using **cast** operator or C++ **static_cast** operator.

```
int i;   double d;
d = i;              // implicit (compiler)
d = (double) i;     // explicit with cast (programmer)
d = static_cast<double>( i ); // C++
```

- Two kinds of conversions:
  - **widening/promotion** conversion, no information is lost:

bool	→	char	→	short int	→	long int	→	double
true		1		1		1		1.000000000000000

where **false** → 0; **true** → 1

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

double	→	long int	→	short int	→	char	→	bool
77777.77777777777		77777		12241		209		**true**

where 0 → **false**; non-zero → **true**

- C/C++ have implicit widening and narrowing conversions (Java only implicit widening).
- **Beware of implicit narrowing conversions:**

```
int i;    double d;
i = d = 3.5;  // d -> 3.5
d = i = 3.5;  // d -> 3.0 truncation
```

- Good practice is to perform narrowing conversions explicitly with cast as documentation.

```
int i;    double d1 = 7.2, d2 = 3.5;
i = (int) d1;             // explicit narrowing conversion
i = (int) d1 / (int) d2; // explicit narrowing conversions for integer division
i = static_cast<int>(d1 / d2); // alternative technique after integer division
```

- C++ supports casting among user defined types (see Section 2.21, p. 105).

2.6.2 Coercion

- **Coercion** reinterprets a value to another type but the result is may not be meaningful in the new type's representation.
- Some narrowing conversions are considered coercions.
  - E.g., when a value is truncated or converting non-zero to **true**, the result is nonsense in the new type's representation.
- Also, having type **char** represent ASCII characters ***and*** integral (byte) values allows:

```
char ch = 'z' - 'a';    // character arithmetic!
```

  which is often unreasonable as it can generate an invalid character.

- But the most common coercion is through pointers (see Section 2.10.2, p. 77):

```
int i, *ip = &i;          // ip is a pointer to an integer
double d, *dp = &d; // dp is a pointer to a double
dp = (double *) ip;  // lie, say dp points at double but really an integer
dp = reinterpret_cast<double *>( ip );
```

  Using explicit cast, programmer has lied to compiler about type of ip.

- **Good practice is to limit narrowing conversions and NEVER lie about a variable's type.**

2.6.3 Math Operations

- **#include** <cmath> provides overloaded real-float mathematical-routines for types **float**, **double** and **long double**:

operation	routine	operation	routine
$\|x\|$	abs(x)	$x \bmod y$	fmod(x, y)
$\arccos x$	acos(x)	$\ln x$	log(x)
$\arcsin x$	asin(x)	$\log x$	log10(x)
$\arctan x$	atan(x)	x^y	pow(x, y)
$\lceil x \rceil$	ceil(x)	$\sin x$	sin(x)
$\cos x$	cos(x)	$\sinh x$	sinh(x)
$\cosh x$	cosh(x)	$\sqrt{x}$	sqrt(x)
e^x	exp(x)	$\tan x$	tan(x)
$\lfloor x \rfloor$	floor(x)	$\tanh x$	tanh(x)

and math literals:

```
M_E          2.7182818284590452354   // e
M_LOG2E      1.4426950408889634074   // log_2 e
M_LOG10E     0.43429448190325182765  // log_10 e
M_LN2        0.69314718055994530942  // log_e 2
M_LN10       2.30258509299404568402  // log_e 10
M_PI         3.14159265358979323846  // pi
M_PI_2       1.57079632679489661923  // pi/2
M_PI_4       0.78539816339744830962  // pi/4
M_1_PI       0.31830988618379067154  // 1/pi
M_2_PI       0.63661977236758134308  // 2/pi
M_2_SQRTPI   1.12837916709551257390  // 2/sqrt(pi)
M_SQRT2      1.41421356237309504880  // sqrt(2)
M_SQRT1_2    0.70710678118654752440  // 1/sqrt(2)
```

- Some systems also provide **long double** math literals.

- pow(x,y) (x^y) is computed using logarithms, $10^{y \log x}$ (versus repeated multiplication), when y is non-integral value $\Rightarrow y \geq 0$

  pow(-2.0, 3.0); $-2^3 = -2 \times -2 \times -2 = -8$
  pow(-2.0, 3.1); $-2^{3.1} = 10^{3.1 \times log -2.0} =$ nan (**n**ot **a** **n**umber)

  nan is generated because $\log -2$ is undefined.

- Quadratic roots of $ax^2 + bx + c$ are $r = -b \pm \sqrt{b^2 - 4ac}/2a$

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

int main() {
    double a = 3.5, b = 2.1, c = -1.2;
    double dis = b * b - 4.0 * a * c, dem = 2.0 * a;
    cout << "root1: " << ( -b + sqrt( dis ) ) / dem << endl;
    cout << "root2: " << ( -b - sqrt( dis ) ) / dem << endl;
}
```

- Must explicitly link in the math library:

```
$ g++ roots.cc -lm  # link math library
```

2.7 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** (*bool-expr1*) *stmt1* **else if** (*bool-expr2*) *stmt2* . . . **else** *stmtN*
	switch (*integral-expr*) { **case** c1: *stmts1*; **break**; . . . **case** cN: *stmtsN*; **break**; **default**: *stmts0*; }	**switch** (*integral-expr*) { **case** c1: *stmts1*; **break**; . . . **case** cN: *stmtsN*; **break**; **default**: *stmts0*; }
looping	**while** (*bool-expr*) *stmt*	**while** (*bool-expr*) *stmt*
	do *stmt* **while** (*bool-expr*) ;	**do** *stmt* **while** (*bool-expr*) ;
	for (*init-expr*;*bool-expr*;*incr-expr*) *stmt*	**for** (*init-expr*;*bool-expr*;*incr-expr*) *stmt*
transfer	**break** *[label]*	**break**
	continue *[label]*	**continue**
		goto *label*
	return *[expr]*	**return** *[expr]*
	throw *[expr]*	**throw** *[expr]*
label	*label* : *stmt*	*label* : *stmt*

2.7.1 Block

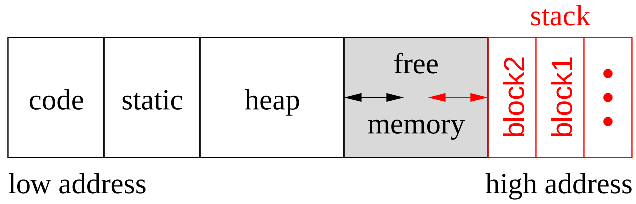
- Compound statement serves two purposes:
  - bracket several statements into a single statement
  - introduce local declarations.
- **Good practice is to always use a block versus single statement to allow adding statements.**

no block	block
`if ( x > y )` `    x = 0;`	`if ( x > y ) {` `    x = 0;` `}`

Does the shell have this problem?

- Nested block variables are allocated first-in first-out (FIFO) from the **stack** memory area.

```
{ // block1
    // variables
    { // block2
        // variables
    }
}
```



- Nested block declarations reduces declaration clutter at start of block.

```
int i, j, k; // global          int i;
... // use i, j, k               ... // use i
                                 {   int j; // local
                                     ... // use i, j
                                     {   int k; // local
                                         ... // use i, j, k
```

However, can also make locating declarations more difficult.

- Variable names can be reused in different blocks, i.e., possibly **overriding** (hiding) prior variables.

```
int i = 1; ...          // first i
{   int k = i, i = 2, j = i; ... // k = first i, second i overrides first
    {   int i = 3;... // third i (overrides second)
```

2.7.2 Selection

- C/C++ selection statements are **if** and **switch** (same as Java).
- For nested **if** statements, **else** matches closest **if**, which results in the **dangling else** problem.
- E.g., reward WIDGET salesperson who sold $10,000 or more worth of WIDGETS and dock pay of those who sold less than $5,000.

Dangling Else	Fix Using Null Else	Fix Using Block
`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;`

- Unnecessary equality for boolean as value is already **true** or **false**.

```
bool b;
if ( b == true )                  if ( b )
```

- Redundant **if** statement.

```
if ( a < b ) return true;         return a < b;
else return false;
```

- Conversion causes problems.

```
if ( -0.5 <= x <= 0.5 )...       // looks right and compiles
if ( ( ( -0.5 <= x ) <= 0.5 ) )... // what does this do?
```

- Assign in expressions causes problems (possible in Java for one type).

```
if ( x = y )... // what does this do?
```

- A **switch** statement selectively executes one of *N* alternatives based on matching an integral value with a series of case clauses (see Section 2.9, p. 73).

```
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:                        // optional
    cerr << "ERROR: bad day" << endl;
    exit( EXIT_FAILURE );         // TERMINATE PROGRAM
}
```

- Only one label for each **case** clause but a list of **case** clauses is allowed.
- Once case label matches, the clauses statements are executed, and control continues to the ***next*** statement. **(comment each fall through)**
- If no case clause is matched and there is a **default** clause, its statements are executed, and control continues to the ***next*** statement.
- Unless there is a **break** statement to prematurely exit the **switch** statement.
- **It is a common error to forget the break in a case clause.**
- Otherwise, the **switch** statement does nothing.

2.8 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.
- Most programmers adopt this approach so there is a universal (common) approach to managing control flow (e.g., like traffic rules).

2.8.1 Multi-Exit Loop

- **Multi-exit loop** (or mid-test loop) has one or more exit locations *within* the loop body.
- While-loop has 1 exit located at the top (Ada):

```
while i < 10 {          loop                    -- infinite loop
                          exit when i >= 10;  -- loop exit
    . . .                   . . .           ↑ reverse condition
}                       end loop
```

- Repeat-loop has 1 exit located at the bottom:

```
do {                    loop                    -- infinite loop
    . . .                   . . .
                          exit when i >= 10;  -- loop exit
} while ( i < 10 )      end loop        ↑ reverse condition
```

- Exit should not be restricted to only top and bottom, i.e., can appear in the loop body:

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

- Loop exit has ability to change kind of loop solely by moving the exit line.
- In general, your coding style should allow changes and insertion of new code with minimal changes to existing code.
- Eliminate priming (duplicated) 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
```

- **Good practice is to reduce or eliminate duplicate code.** Why?
- Loop exit is outdented or commented or both (**Eye Candy**) ⇒ easy to find without searching entire loop body.

- Same indentation rule as for the **else** of if-then-else (outdent **else**):

```
if . . . then          if . . . then
    XXX                    XXX
    else                 else      // outdent
    XXX                    XXX
end if                 end if
```

- 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;
    . . .                        . . .                        . . .
}                            }                            } while( true );
```

- The **for** version is more general as easily modified to have a loop index.

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

- Eliminate **else** on loop exits:

BAD	GOOD	BAD	GOOD
for (;;) { S1 if (C1) { S2 } else { break; } S3 }	for (;;) { S1 if (! C1) break; S2 S3 }	for (;;) { S1 if (C1) { break; } else { S2 } S3 }	for (;;) { S1 if (C1) break; S2 S3 }

S2 is logically part of loop body ***not*** part of an **if**.

- Allow multiple exit conditions:

```
                               bool flag1 = false, flag2 = false;
for ( ;; ) {                   while ( ! flag1 & ! flag2 ) {
    S1                             S1
  if ( i >= 10 ) break;            if ( C1 ) flag1 = true;
                                   } else {
    S2                                 S2
  if ( j >= 10 ) break;                if ( C2 ) flag2 = true;
                                       } else {
    S3                                     S3
}                                      }
                                   }
                               }
```

- Eliminate flag variables necessary with **while**.
  - **flag variable** is used solely to affect control flow, i.e., does not contain data associated with a computation.

- Case study: examine linear search such that:
  - no invalid subscript for unsuccessful search
  - index points at the location of the key for successful search.
- First approach: use only control-flow constructs **if** and **while**:

```
int i = -1;  bool found = false;
while ( i < size - 1 & ! found ) { // rewrite: &(i<size-1, !found)
      i += 1;
      found = key == list[i];
}
if ( found ) { ...       // found
} else { ...             // not found
}
```

  Why must the program be written this way?

- Second approach: allow short-circuit control structures.

```
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
}
```

- How does && prevent subscript error?
- Short-circuit && does not exist in all programming languages, and requires knowledge of Boolean algebra (false and anything is?).
- Third approach: use multi-exit loop (especially if no && exits).

```
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
}
```

- When loop ends, it is known if the key is found or not found.
- Why is it necessary to re-determine this fact after the loop?
- Can it always be re-determined?
- Extra test after loop can be eliminated by moving it into 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, increment its associated list counter.

```
for ( int i = 0; ; i += 1 ) {
  if ( i >= size ) {
        list[size].count = 1;  // add element to list
        list[size].data = key;
        size += 1;  // needs check for array overflow
        break;
    } // exit
  if ( key == list[i].data ) {
        list[i].count += 1;   // increment counter
        break;
    } // exit
} // for
```

- None of these approaches is best in all cases; select the approach that best fits the problem.

2.8.2 Multi-Level Exit

- **Multi-level exit** transfers out of multiple control structures where exit points are *known* at compile time.
- Labelled exit (**break**/**continue**) provides this capability (Java):

```
L1: {
    ... declarations ...
    L2: switch ( ... ) {
        L3: for ( ... ) {
            ... break L1; ... // exit block
            ... break L2; ... // exit switch
            ... break L3; ... // exit loop
        }
        ...
    }
    ...
}
```

- Labelled **break**/**continue** transfer control out of the control structure with the corresponding label, terminating any block that it passes through.

- C/C++ do not have labelled **break**/**continue** ⇒ simulate with **goto**.
- **goto** *label* allows arbitrary transfer of control ***within*** a routine.
- **Label variable** declared by prefixing "identifier**:**" to a statement.

```
L1: i += 1;            // associated with expression
L2: if ( ... ) ...;    // associated with if statement
L3: ;                  // associated with empty statement
```

- Labels have ***routine scope*** (see Section 2.4.2, p. 49), i.e., cannot be overridden in local blocks.

```
int L1;                // identifier L1
L2: ;                  // identifier L2
{
    double L1;         // can override variable identifier
    double L2;         // cannot override label identifier
}
```

- **goto** transfers control backwards/forwards to labelled statement.

```
L1: ;
...
goto L1;               // transfer backwards, up
goto L2;               // transfer forward, down
...
L2: ;
```

- Transforming labelled **break** to **goto**:

```
{
    ... declarations ...
    switch ( ... ) {
        for ( ... ) {
            ... goto L1; ... // exit block
            ... goto L2; ... // exit switch
            ... goto L3; ... // exit loop
        }
        L3: ; // empty statement
        ...
    }
    L2: ;
    ...
}
L1: ;
```

- Why are labels at the end of control structures not as good as at start?
- Why is it good practice to associate a label with an empty statement?

- Multi-level exits are commonly used with nested loops:

```
int i, j;

for ( i = 0; i < 10; i += 1 ) {

    for ( j = 0; j < 10; j += 1 ) {

        . . .
      if ( . . . ) goto B2;  // outdent

        . . . // rest of loop
  if ( . . . ) goto B1;  // outdent

        . . . // rest of loop


    } B2: ;

    . . . // rest of loop

} B1: ;
```

```
int i, j;
bool flag1 = false;
for ( i = 0; i < 10 && ! flag1; i += 1 ) {
    bool flag2 = false;
    for ( j = 0; j < 10 &&
         ! flag1 && ! flag2; j += 1 ) {
        . . .
        if ( . . . ) flag2 = true;
        else {
            . . . // rest of loop
            if ( . . . ) flag1 = true;
            else {
                . . . // rest of loop
            } // if
        } // if
    } // for
    if ( ! flag1 ) {
        . . . // rest of loop
    } // if
} // for
```

- Indentation matches with control-structure terminated.
- Eliminate all flag variables with multi-level exit!
  - ***Flag variables are the variable equivalent to a goto*** because they can be set/reset/tested at arbitrary locations in a program.
- Simple case (exit 1 level) of multi-level exit is a multi-exit loop.
- Why is it good practice to label all exits?
- Normal and labelled **break** are a **goto** with restrictions:
  - Cannot be used to create a loop (i.e., cause a backward branch); hence, all repeated execution is clearly delineated by loop constructs.
  - Cannot be used to branch ***into*** a control structure.
- **Only use goto to perform static multi-level exit, e.g., simulate labelled break and continue.**
- **return** statements can simulate multi-exit loop and multi-level exit.
- Multi-level exits appear infrequently, but are extremely concise and execution-time efficient.

2.9 Command-line Arguments

- Starting routine main has two overloaded prototypes.

```
int main(); // C: int main( void );
int main( int argc, char *argv[] ); // parameter names may be different
```

- Second form is used to receive command-line arguments from the shell, where the command-line string-tokens are transformed into C/C++ parameters.
- argc is the number of string-tokens on the command line, including the command name.
- ***Java does not include command name, so number of tokens is one less***.
- argv is an array of pointers to C character strings that make up token arguments.

```
% ./a.out -option infile.cc outfile.cc
        0      1         2          3
argc    = 4              // number of command-line tokens
argv[0] = ./a.out\0      // not included in Java
argv[1] = -option\0
argv[2] = infile.cc\0
argv[3] = outfile.cc\0
argv[4] = 0              // mark end of variable length list
```

- Because shell only has string variables, a shell argument of "32" does not mean integer 32, and may have to converted.
- Routine main usually begins by checking argc for command-line arguments.

Java	C/C++
<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>	<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(EXIT_FAILURE); } . . .</pre>

- Arguments are processed in the range argv[1] through argv[argc - 1] (one greater than Java).
- Process following arguments from shell command line for command:

```
cmd [ size (> 0) [ code (> 0) [ input-file [ output-file ] ] ] ]
```

- Note, **new**/**delete** versus malloc/free, stringstream (atoi does not indicate errors), and no duplicate code.

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <cstdlib>                       // exit
using namespace std;                     // direct access to std

bool convert( int &val, char *buffer ) { // convert C string to integer
    stringstream ss( buffer );           // connect stream and buffer
    ss >> dec >> val;                    // convert integer from buffer
    return ! ss.fail() &&                // conversion successful ?
        // characters after conversion all blank ?
        string( buffer ).find_first_not_of( " ", ss.tellg() ) == string::npos;
} // convert

enum { sizeDeflt = 20, codeDeflt = 5 };  // global defaults

void usage( char *argv[] ) {
    cerr << "Usage: " << argv[0] << " [ size (>= 0 : " << sizeDeflt <<
     ") [ code (>= 0 : " << codeDeflt << ") [ input-file [ output-file ] ] ] ]"
     << endl;
    exit( EXIT_FAILURE );                // TERMINATE PROGRAM
} // usage

int main( int argc, char *argv[] ) {
    int size = sizeDeflt, code = codeDeflt; // default value
    istream *infile = &cin;              // default value

    ostream *outfile = &cout;            // default value
    switch ( argc ) {
      case 5:
        outfile = new ofstream( argv[4] );
        if ( outfile->fail() ) usage( argv ); // open failed ?
        // FALL THROUGH
      case 4:
        infile = new ifstream(  argv[3] );
        if ( infile->fail() ) usage( argv );  // open failed ?
        // FALL THROUGH
      case 3:
        if ( ! convert( code, argv[2] ) || code < 0 ) usage( argv ) ; // invalid ?
        // FALL THROUGH
      case 2:
        if ( ! convert( size, argv[1] ) || size < 0 ) usage( argv ); // invalid ?
        // FALL THROUGH
      case 1:                            // all defaults
        break;
      default:                           // wrong number of options
        usage( argv );
    }
    // program body
    if ( infile != &cin ) delete infile;       // close file, do not delete cin!
    if ( outfile != &cout ) delete outfile;    // close file, do not delete cout!
} // main
```

2.10 Type Constructor

- **Type constructor** declaration builds more complex type from basic types.

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

2.10.1 Enumeration

- An **enumeration** is a type defining a set of named literals with only assignment, comparison, and conversion to integer:

```
enum Days {Mon,Tue,Wed,Thu,Fri,Sat,Sun}; // type declaration, implicit numbering
Days 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 numbering
colour = B;                              // assignment
```

- Identifiers in an enumeration are called **enumerator**s.
- First enumerator is implicitly numbered 0; thereafter, each enumerator is implicitly numbered +1 the previous enumerator.
- Enumerators can be explicitly numbered.

```
enum { A = 3, B, C = A - 5, D = 3, E }; // 3 4 -2 3 4
enum { Red = 'R', Green = 'G', Blue = 'B' }; // 82, 71, 66
```

- Enumeration in C++ denotes a new type; enumeration in C is alias for **int**.

```
day = Sat;          // enumerator must match enumeration
day = 42;           // disallowed C++, allowed C
day = R;            // disallowed C++, allowed C
day = colour;       // disallowed C++, allowed C
```

- Alternative mechanism to create literals is **const** declaration (see page 52).

```
const short int Mon=0,Tue=1,Wed=2,Thu=3,Fri=4,Sat=5,Sun=6;
short int day = Sat;
days = 42;          // assignment allowed
```

- C/C++ enumerators must be unique in block.

```
enum CarColour { Red, Green, Blue, Black };
enum PhoneColour { Red, Orange, Yellow, Black };
```

Enumerators Red and Black conflict. (Java enumerators are always qualified).

- In C, "**enum**" must also be specified for a declaration:

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

- Trick to count enumerators (if no explicit numbering):

```
enum Colour { Red, Green, Yellow, Blue, Black, No_Of_Colours };
```

No_Of_Colours is 5, which is the number of enumerators.

- Iterating over enumerators:

```
for ( Colour c = Red; c < No_Of_Colours; c = (Colour)(c + 1) ) {
    cout << c << endl;
}
```

Why is the cast, (Colour), necessary? Is it a conversion or coercion?

2.10.2 Pointer/Reference

- **pointer**/**reference** is a memory address.

```
int x, y;
int *p1 = &x, *p2 = &y, *p3 = 0; // or p3 is uninitialized
```

- Used to access the value stored in the memory location at pointer address.
- Can a pointer variable point to itself?
- Pointer to a string literal must be **const**. Why?

```
const char *cs = "abc";
```

- Pointer variables have two forms of assignment:
  - pointer assignment

    ```
    p1 = &x;        // pointer assignment
    p2 = p1;        // pointer assignment
    ```

    no dereferencing to access values.
  - value assignment

    ```
    *p2 = *p1;      // value assignment, y = x
    ```

    dereferencing to access values.
- Value used more often than pointer.

```
*p2 = ((*p1 + *p2) * (*p2 - *p1)) / (*p1 - *p2);
```

- Less tedious and error prone to write:

```
p2 = ((p1 + p2) * (p2 - p1)) / (p1 - p2);
```

- C++ reference pointer provides extra implicit dereference to access target value:

```
int &r1 = x, &r2 = y;
r2 = ((r1 + r2) * (r2 - r1)) / (r1 - r2);
```

- **Hence, difference between plain and reference pointer is an extra implicit dereference.**
  - I.e., do you want to write the "*", or let the compiler write the "*"?
- However, implicit dereference generates problem for pointer assignment.

```
r2 = r1;    // not pointer assignment
```

- C++ solves the missing pointer assignment by making reference pointer a literal (**const**), like a plain variable.
  - Hence, a reference pointer cannot be assigned after its declaration, **so pointer assignment is impossible**.
  - As a literal, initialization must occur at declaration, but initializing expression has implicit referencing because address is ***always*** required.

```
int &r1 = &x; // error, should not have & before x
```

- Java solves this problem by only using reference pointers, only having pointer assignment, and using a different mechanism for value assignment (clone).
- Is there one more solution?
- Since reference means its target's value, address of a reference means its target's address.

```
int i;
int &r = i;
&r;            ⇒ &i not &r
```

- Hence, cannot initialize reference to reference or pointer to reference.

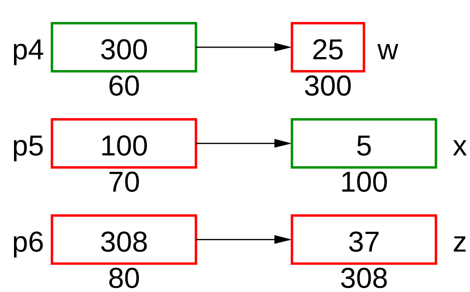
```
int & &rr = r;     // reference to reference, rewritten &r
int &*pr = &r;     // pointer to reference
```

- As well, an array of reference is disallowed (reason unknown).

```
int &ra[3] = { i, i, i };    // array of reference
```

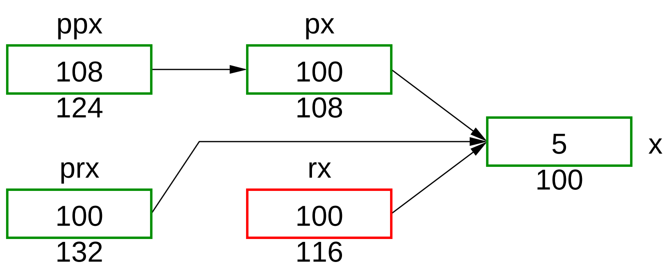
- Type qualifiers (see Section 2.4.3, p. 50) can be used to modify pointer types.

```
const short int w = 25;
const short int *p4 = &w;

int * const p5 = &x;
int &p5 = x;

const long int z = 37;
const long int * const p6 = &z;
const long int &p6 = z;
```



- p4 may point at ***any*** **short int** variable (**const** or non-**const**) and may not change its value.
  Why can p4 point to a non-**const** variable?
- p5 may only point at the **int** variable x and may change the value of x through the pointer.
  - * **const** and & are literal pointers but * **const** has no implicit dereferencing like &.
- p6 may only point at the **long int** variable z and may not change its value.
- Pointer variable has memory address, so it is possible for a pointer to address another pointer or object containing a pointer.

```
int *px = &x, **ppx = &px,
    &rx = x, *prx = &rx;
```



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

`int* p1, p2;`	p1 is a pointer, p2 is an integer	`int *p1, *p2;`
`int& rx = i, ry = i;`	rx is a reference, ry is an integer	`int &rx =i, &ry = i;`

2.10.3 Aggregates

- Aggregates are a set of homogeneous/heterogeneous values and a mechanism to access the values in the set.

2.10.3.1 Array

- **Array** is a set of **homogeneous values**.

```
int array[10];        // 10 int values
```

- Array type, **int**, is the type of each set value; array **dimension**, 10, is the maximum number of values in the set.
- An array can be structured to have multiple dimensions.

```
int matrix[10][20];   // 10 rows, 20 columns => 200 int values
char cube[5][6][7];   // 5 rows, 6 columns, 7 deep => 210 char values
```

  Common dimension mistake: matrix[**10, 20**]; means matrix[20] because 10, 20 is a comma expression not a dimension list.
- Number of dimensions is fixed at compile time, but dimension size may be:
  - static (compile time),

- ◦ block dynamic (static in block),
- ◦ or dynamic (change at any time, see vector Section 2.36.1.1, p. 167).

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

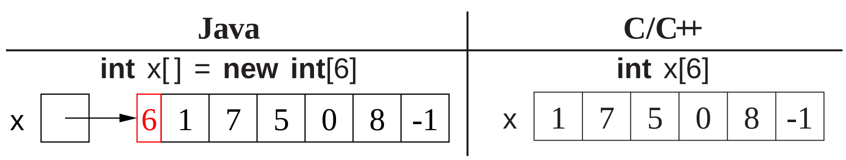
```
int r, c;
cin >> r >> c;          // input dimensions
int array[r];           // dynamic dimension, C/g++ only
int matrix[r][c];       // dynamic dimension, C/g++ only
```

- A dimension is subscripted from 0 to dimension-1.

```
array[5] = 3;           // location at column 5
i = matrix[0][2] + 1;   // value at row 0, column 2
c = cube[2][0][3];      // value at row 2, column 0, depth 3
```

Common subscript mistake: matrix[**3, 4**] means matrix[4], 4th row of matrix.

- **Do not use pointer arithemetic to subscript arrays: error prone and no more efficient.**
- C/C++ array is a contiguous set of elements not a reference to the element set as in Java.



- **C/C++ do not store dimension information in the array!**
- Hence, cannot query dimension sizes, ***no subscript checking***, and no array assignment.
- Declaration of a pointer to an array is complex in C/C++ (see also page 92).
- Because no array-size information, the dimension value for an array pointer is unspecified.

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

- However, no dimension information results in the following ambiguity:

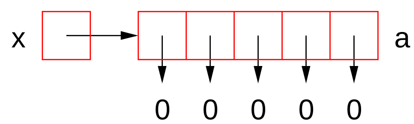
```
int *pvar = &i;         // think pvar[] and i[1]
int *parr = arr;        // think parr[]
```

- ***Variables* pvar *and* parr *have same type but one points at a variable and other an array!***
- Programmer decides if variable or array by not using or using subscripting.

```
*pvar                   // variable
*parr                   // variable, arr[0]
parr[0], parr[3]        // array, many
pvar[3]                 // array, but wrong
```

- ASIDE: Practise reading a complex declaration:
  - parenthesize type qualifiers based on operator priority (see Section 2.6, p. 61),
  - read inside parenthesis outwards,
  - start with variable name,
  - end with type name on the left.

```
const long int * const a[5] = {0,0,0,0,0};
const long int * const (&x)[5] = a;
const long int ( * const ( (&x)[5] ) ) = a;
```



x : reference to an array of 5 constant pointers to constant long integers

2.10.3.2 Structure

- **Structure** is a set of **heterogeneous values**, including (nested) structures.

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

- Structure fields are called **members** subdivided into data and routines[1].
- All members of a structure are accessible (public) by default.
- Member ***CANNOT*** be directly initialized (C++11 allowed with constructor) (see Section 2.13, p. 85 and 2.21.3, p. 108).

```
struct foo {
    int i = 3, j = 4;   // disallowed
}; // semi-colon
```

- Structure can be defined and instances declared in a single statement.

```
struct Complex { double re, im; } s; // definition and declaration
```

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

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

- Pointers to structures have a problem:
  - C/C++ are unique in having the priority of selection operator "." higher than dereference operator "*". (Java is coorect.)
  - Hence, *p.f executes as *(p.f), which is incorrect. Why?

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

- ◦ To get the correct effect, use parenthesis: (*p).f.

```
(*sp1).name.first[0] = 'a';
(*sp1).age = 34;
(*sp1).marks[5] = 95;
```

- Alternatively, use (special) selection operator -> for pointers to structures:
  - ◦ performs dereference and member selection in correct order, i.e., p->f rewritten as (*p).f.

```
sp1->name.first[0] = 'a';
sp1->age = 34;
sp1->marks[5] = 95;
```

  - ◦ for reference pointers, -> is unnecessary because r.f means (*r).f, so r.f makes more sense than (&r)->f.

- **Makes switching from a pointer to reference difficult** (. ↔ ->).
- Structures ***must*** be compared member by member.
  - ◦ comparing bits (e.g., memcmp) fails as alignment padding leaves undefined values between members.
- **Recursive type**s (lists, trees) are defined using a self-referential pointer in a structure:

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

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

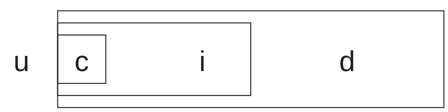
```
struct S {
    int i : 3;     // 3 bits
    int j : 7;     // 7 bits
    int k : 6;     // 6 bits
} s;
s.i = 2;     // 010
s.j = 5;     // 0000101
s.k = 9;     // 001001
```

- A bit field must be an integral type.
- Unfortunately allocation of bit-fields is implementation defined ⇒ not portable (maybe left to right or right to left!).
- Hence, the bit-fields in variable s above must be reversed.
- While it is unfortunate C/C++ bit-fields lack portability, they are the highest-level mechanism to manipulate bit-specific information.

2.10.3.3 Union

- **Union** is a set of **heterogeneous values**, including (nested) structures, **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 {      // IEEE floating-point representation
        unsigned int sign : 1;   // may need to be reversed
        unsigned int exp : 8;
        unsigned int frac : 23;
    } s;
    int i;
} u;
u.f = 3.5e3;     cout << u.s.sign << '\t' << u.s.exp << '\t' << u.s.frac << endl;
u.f = -3.5e-3;   cout << u.s.sign << '\t' << u.s.exp << '\t' << u.s.frac << endl;
u.f = 3.5;       cout << u.f << '\t' << hex << u.i << endl;
u.i = 3;         cout << u.i << '\t' << u.f << endl;
```

produces:

```
0   8a  5ac000       internal representation of 3.5e3
1   76  656042       internal representation of -3.5e-3
3.5 40600000         coerce double to int
3   4.2039e-45       coerce int to double
```

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

2.11 Type Equivalence

- In Java/C/C++, types are equivalent if they have the same name, called **name equivalence**.

```
struct T1 {                     struct T2 {   // identical structure
    int i, j, k;                    int i, j, k;
    double x, y, z;                 double x, y, z;
};                              };
T1 t1, t11 = t1;    // allowed, t1, t11 have compatible types
T2 t2 = t1;         // disallowed, t2, t1 have incompatible types
T2 t2 = (T2)t1;     // disallowed, no conversion from type T1 to T2
```

- Types T1 and T2 are **structurally equivalent**, but have different names so they are incompatible, i.e., initialization of variable t2 is disallowed.

- An **alias** is a different name for same type, so alias types are equivalent.
- C/C++ provides **typedef** to create an alias 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**".
- Use to prevent repetition of large type names:

```
void f( map<string, pair<vector<string>, map<string,string> > > p1,
        map<string, pair<vector<string>, map<string,string> > > p2 );

typedef map<string, pair<vector<string>, map<string,string> > > StudentInfo;
void f( StudentInfo p1, StudentInfo p2 );
```

- Java provides no mechanism to alias types.

2.12 Type Nesting

- Type nesting is used to organize and control visibility of type names (see Section 2.26, p. 130):

```
enum Colour { R, G, B, Y, C, M };
struct Person {
    enum Colour { R, G, B };     // nested type
    struct Face {                // nested type
        Colour Eyes, Hair;       // type defined outside (1 level)
    };
    ::Colour shirt;              // type defined outside (top level)
    Colour pants;                // type defined same level
    Face looks[10];              // type defined same level
};
Colour c = R;                    // type/enum defined same level
Person::Colour pc = Person::R;   // type/enum defined inside
Person::Face pretty;             // type defined inside
```

- **C can nest types but not name them! (useless)**
- Variables/types at top nesting-level are accessible with unqualified "::".
- References to types inside the nested type do not require qualification (like declarations in nested blocks, see Section 2.4.2, p. 49).
- References to types nested inside another type are qualified with "::".

- Without nested types need:

```
enum Colour { R, G, B, Y, C, M };
enum Colour2 { R2, G2, B2 };    // prevent name clashes
struct Face {
    Colour2 Eyes, Hair;
};
struct Person {
    Colour shirt;
    Colour2 pants;
    Face looks[10];
};
Colour c = R;
Colour2 pc = R2;
Face pretty;
```

- ***Do not pollute lexical scopes with unnecessary names (name clashes).***

2.13 Type-Constructor Literal

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 (Java null).
- System include-file defines the preprocessor variable NULL as 0 (see Section 2.17, p. 94).
- Structure and array initialization can 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
```

- A nested structure or multidimensional array is created using nested braces.
- Initialization values are placed into a variable starting at beginning of the structure or array.
- Not all the members/elements must be initialized.
  - If not explicitly initialized, a variable is **default initialized** (see also Section 2.21.3, p. 108), which means zero-filled for basic types.

```
int b[10];          // uninitialized
int b[10] = {};     // zero initialized
```

- Cast allows construction of structure and array literals in statements:

```
void rtn( const int m[2][3] );
struct Complex { double r, i; } c;
rtn( (const int [2][3]){ {93, 67, 72}, {77, 81, 86} } );  // g++ only
c = (Complex){ 2.1, 3.4 };  // g++ only
c = { 2.1, 3.4 };  // C++11 only
```

- In both cases, a cast indicates the type and structure of the literal.
- String literals 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 literals 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
```

2.14 Routine

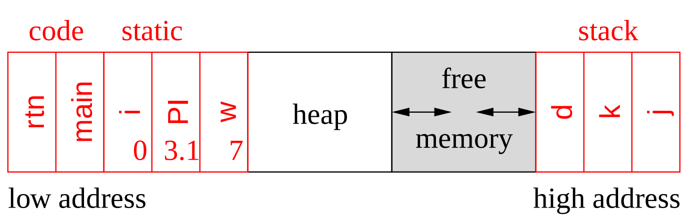
- Routine with no parameters has parameter **void** in C and empty parameter list in C++:

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

  - In C, empty parameters mean no information about the number or types of the parameters is supplied.

- If a routine is qualified with **inline**, the routine is expanded (maybe) at the call site, i.e., unmodularize, to increase speed at the cost of storage (no call).
- Routine cannot be nested in another routine (possible in gcc).
- Java requires all routines to be defined in a **class** (see Section 2.21.1, p. 106).
- Each routine call creates a new block on the stack containing its parameters and local variables, and returning removes the block.
- Variables declared outside of routines are defined in implicit **static block**.

```
int i;                     // static block, global
const double PI = 3.14159;
void rtn( double d )       // code block
{   static const int w = 7; // create static block
}                          // remove stack block
int main()                 // code block
{   int j;                 // create stack block
    {   int k;             // create stack block
        rtn( 3.0 );
    }                      // remove stack block
}                          // remove stack block
```



Where is the program executing?

- Static block is a separate memory area from stack and heap areas and ***is always zero filled***.
- Otherwise variables are **uninitialized**.
- **Good practise is to ONLY use static block for**
  - constants (anywhere) set to literals

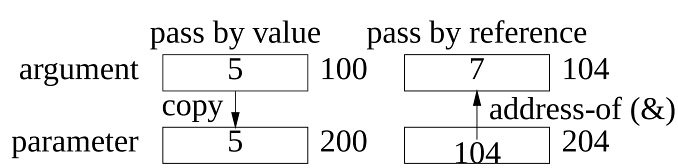
    ```
    bool check( int key ) {
        static const int vals[] = { 12, 15, 34, 67, 88 }; // allocated ONCE
        . . .
    ```

  - global variables accessed throughout program

    ```
    int callCounter = 0;
    int rtn1( int key ) { callCounter += 1; . . . }
    int rtn2( int key ) { callCounter += 1; . . . }
    . . .
    ```

2.14.1 Argument/Parameter Passing

- Modularization without communication is useless; information needs to flow from call to routine and back to call.
- Communication is achieved by passing arguments from a call to parameters in a routine and back to arguments or return values.
  - **value parameter**: parameter is initialized by copying argument (input only).
  - **reference parameter**: parameter is a reference to the argument and is initialized to the argument's address (input/output).



- 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.
- For value parameters, each argument-expression result is copied into the corresponding parameter in the routine's block on the stack, ***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 as the corresponding reference parameter.

```
struct S { double d; };
void r1( S s, S &rs, S * const ps ) {
    s.d = rs.d = ps->d = 3.0;
}
int main() {
    S s1 = {1.0}, s2 = {2.0}, s3 = {7.5};
    r1( s1, s2, &s3 );
    // s1.d = 1.0, s2.d = 3.0, s3.d = 3.0
}
```

call

	s1	s2	s3
argument	1.0	2.0	7.5
	100	200	300
parameter	1.0	200	300
	s	rs	ps

return

	s1	s2	s3
argument	1.0	3.0	3.0
	100	200	300
parameter	3.0	200	300
	s	rs	ps

- C-style pointer-parameter simulates the reference parameter, but requires & on argument and use of -> with parameter.

- Value passing is most efficient for small values or for large values with high referencing because the values are accessed directly (not through pointer).

- Reference passing is most efficient for large values with low/medium referencing because the values are not duplicated in the routine but accessed via pointers.

- Problem: cannot change a literal or temporary variable via parameter!

```
void r2( int &i, Complex &c, int v[] );
r2( i + j, (Complex){ 1.0, 7.0 }, (int [3]){ 3, 2, 7 } );  // disallowed!
```

- Use type qualifiers to create read-only reference parameters so the corresponding argument is guaranteed not to change:

```
void r2( const int &i, const Complex &c, const int v[] ) {
    i = 3;    // disallowed, read only!
    c.re = 3.0;
    v[0] = 3;
}
r2( i + j, (Complex){ 1.0, 7.0 }, (int [3]){ 3, 2, 7 } );  // allowed!
```

- Provides efficiency of pass by reference for large variables, security of pass by value as argument cannot change, and allows literals and temporary variables as arguments.

- **Good practise uses reference parameters rather than pointer.**

- C++ parameter can have a **default value**, which is passed as the argument value if no argument is specified at the call site.

```
void r3( int i, double g, char c = '*', double h = 3.5 ) { ... }
r3( 1, 2.0, 'b', 9.3 );    // maximum arguments
r3( 1, 2.0, 'b' );         // h defaults to 3.5
r3( 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.

2.14.2 Array Parameter

- Array copy is unsupported (see Section 2.10, p. 76) so arrays cannot be passed by value.
- Instead, array argument is a pointer to the array that is copied into the corresponding array parameter (pass by value).
- 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 practice uses middle form as it clearly indicates variable can 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[] ) {
    double total = 0.0;
    for ( int c = 0; c < cols; c += 1 )
        total += v[c];
    return total;
}
```

```
double sum( int rows, int cols, double *m[] ) {
    double 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;
}
```

2.15 Overloading

- **Overloading** is when a name has multiple meanings in the same context.
- Most languages have overloading, e.g., most built-in operators are overloaded on both integral and real-floating operands, i.e., + operator is different for `1 + 2` than for `1.0 + 2.0`.
- Overloading requires disambiguating among identical names based on some criteria.

- Normal criterion is type information.
- In general, overloading is done on operations not variables:

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

- ***Power of overloading occurs when programmer changes a variable's type: operations on the variable are implicitly reselected for new type.***
- E.g., after changing a variable's type from **int** to **double**, all operations implicitly change from integral to real-floating.
- Number and ***unique*** parameter types ***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 ambiguities:

```
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
```

- Subtle cases:

```
int i;  unsigned int ui;  long int li;
void r( int i ) { ... }  // overload name r three different ways
void r( unsigned int i ) { ... }
void r( long int i ) { ... }
r( i );             // int
r( ui );            // unsigned int
r( li );            // long int
```

- Parameter types with qualifiers other than **short**/**long**/**signed**/**unsigned** are ambiguous at definition:

```
int r( int i ) {...}              // rewritten: int r( signed int )
int r( signed int i ) {...}       // disallowed : redefinition of first r
int r( const int i ) {...}        // disallowed : redefinition of first r
int r( volatile int i ) {...}     // disallowed : redefinition of first r
```

- Reference parameter types with same base type are ambiguous at call:

```
int r( int i ) {...}          // cannot be called
int r( int &i ) {...}         // cannot be called
int r( const int &i ) {...} // cannot be called
int i = 3;
const int j = 3;
r( i );     // disallowed : ambiguous
r( j );     // disallowed : ambiguous
```

  Cannot cast argument to select r(**int** i), r(**int** &i) or r(**const int** &i).

- Overload/conversion confusion: I/O operator << is overloaded with **char** * to print a C string and **void** * to print pointers.

```
char c;  int i;
cout << &c << " " << &i << endl;  // print address of variables
```

  type of &c is char *, so printed as C string, which is undefined; type of &i is **int** *, which is converted to **void** *, so printed as an address.

- Fix using coercion.

```
cout << (void *)&c << " " << &i << endl;  // print address of variables
```

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

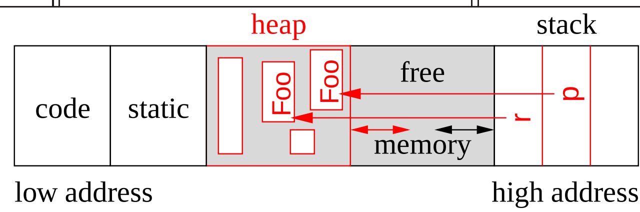
Overloading	Default Argument
int r(**int** i, **int** j) { ... } **int** r(**int** i) { **int** j = 2; ... } r(3); *// 2nd r*	**int** r(**int** i, **int** j = 2) { ... } r(3); *// default argument of 2*

  If the overloaded routine bodies are essentially the same, use a default argument, otherwise use overloaded routines.

2.16 Dynamic Storage Management

- Java/Scheme are **managed languages** because the language controls all memory management, e.g., **garbage collection** to free dynamically allocated storage.
- C/C++ are **unmanaged languages** because the programmer is involved in memory management, e.g., no garbage collection so dynamic storage must be explicitly freed.
- 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++
class Foo { **char** c1, c2; } Foo r = **new** Foo(); r.c1 = 'X'; *// r garbage collected*	**struct** Foo { **char** c1, c2; }; **struct** Foo *p = (**struct** Foo *) *// coerce* malloc(*// allocate* **sizeof**(**struct** Foo) *// size*); p->c1 = 'X'; free(p); *// explicit free*	**struct** Foo { **char** c1, c2; }; Foo *p = **new** Foo(); p->c1 = 'X'; **delete** p; *// explicit free* Foo &r = ***new** Foo(); r.c1 = 'X'; **delete** &r; *// explicit free*



Unallocated memory in heap is also free.

- Allocation has 3 steps:
  1. determine size of allocation,
  2. allocate heap storage of correct size/alignment,
  3. coerce undefined storage to correct type.
- C++ operator **new** performs all 3 steps implicitly; each step is explicit in C.
- Coercion cast is required in C++ for malloc but optional in C.
  - C has implicit cast from **void** * (pointer to anything) to specific pointer (***dangerous!***).
  - Good practise in C is to use a cast so compiler can verify type compatibility on assignment.
- Parenthesis after the type name in the **new** operation are optional.
- For reference r, why is there a "*" before **new** and an "&" in the **delete**?
- Storage for dynamic allocation comes from a memory area called the **heap**.
- If heap is full (i.e., no more storage available), malloc returns 0, and **new** terminates program with an error.
- Unlike Java, C/C++ allow ***all*** types to be dynamically allocated not just object types, e.g., **new int**.
- As well, C/C++ allow ***all*** types to be allocated on the stack, i.e., local variables of a block.
- Declaration of a pointer to an array is complex in C/C++ (see also page 80).
- Because no array-size information, no dimension for an array pointer.

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

- No dimension information results in the following ambiguity:

```
int *pvar = new int;        // basic "new"
int *parr = new int[10];    // parr[ ], array "new"
```

- Variables pvar and parr have the same type but one is allocated with the basic **new** and the other with the array **new**.

- Special syntax ***must*** be used to call the corresponding deletion operation for a variable or an array (any dimensions):

```
delete pvar;      // basic delete : single element
delete [ ] parr;  // array delete : multiple elements (any dimension)
```

- If basic **delete** is used on an array, only the first element is freed (memory leak).

- If array **delete** is used on a variable, storage after the variable is also freed (often failure).

- **Never do this:**

```
delete [ ] parr, pvar; // => (delete [ ] parr), pvar;
```

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

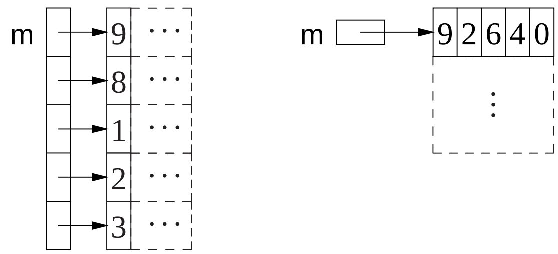
- **Dynamic allocation should be used only when a variable's storage must outlive the block in which it is allocated** (see also page 109).

```
Type *rtn(. . .) {
    Type *tp = new Type;   // MUST USE HEAP
    . . .                  // initialize/compute using tp
    return tp;             // storage outlives block
}                          // tp deleted later
```

- Stack allocation eliminates explicit storage-management (simpler) and is more efficient than heap allocation — **use it whenever possible.**

```
{ // good, use stack                    { // bad, unnecessary dynamic allocation
    int size;                               int *sizep = new int;
    cin >> size;                            cin >> *sizep;
    int arr[size]                           int *arr = new int[*sizep];
    . . .                                   . . .
                                            delete [ ] arr;
                                            delete sizep;
} // size, arr implicitly deallocated   }
```

- Declaration of a pointer to a matrix is complex in C/C++, e.g., **int** *m[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.
- Dimension is higher priority so declaration is interpreted as **int** (*(m[5])) (left).
- Right example cannot be generalized to a dynamically-sized matrix.

```
int R = 5, C = 4;              // 5 rows, 4 columns
int (*m)[C] = new int[R][C];  // disallowed, C must be literal, e.g, 4
```

Compiler must know the stride (number of columns) to compute row.

- Left example can be generalized to a dynamically-sized matrix.

```
int main() {
    int R = 5, C = 4;             // or cin >> R >> C;
    int *m[R];                    // R rows
    for ( int r = 0; r < R; r += 1 ) {
        m[r] = new int[C];        // C columns per row
        for ( int c = 0; c < C; c += 1 ) {
            m[r][c] = r + c;      // initialize matrix
        }
    }

    for ( int r = 0; r < R; r += 1 ) { // print matrix
        for ( int c = 0; c < C; c += 1 ) {
            cout << m[r][c] << ", ";
        }
        cout << endl;
    }
    for ( int r = 0; r < R; r += 1 ) {
        delete [] m[r];           // delete each row
    }
}                                 // implicitly delete array “m”
```

2.17 Preprocessor

- Preprocessor is a text editor that modifies the program text ***before*** compilation.
- **Program you see is not what the compiler sees!**
- -E run only the preprocessor step and write preprocessor output to standard out.

```
$ g++ -E *.cc ...
... much output from the preprocessor
```

2.17.1 File Inclusion

- File inclusion copies text from a file into a C/C++ program.
- Java implicitly includes by matching class names with file names in CLASSPATH directories, then extracting and including declarations.
- **#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"
```

- -v show each compilation step and its details:

```
$ g++ -v *.cc *.o ...
... much output from each compilation step
```

E.g., include directories where cpp looks for system includes.

```
#include <...> search starts here:
 /usr/include/c++/4.6
 /usr/include/c++/4.6/x86_64-linux-gnu/.
 /usr/include/c++/4.6/backward
 /usr/lib/gcc/x86_64-linux-gnu/4.6/include
 /usr/local/include
 /usr/lib/gcc/x86_64-linux-gnu/4.6/include-fixed
 /usr/include/x86_64-linux-gnu
 /usr/include
```

- -I***directory*** search directory for include files;
  - files within the directory can now be referenced by relative name using **#include** <file-name>.

2.17.2 Variables/Substitution

- **#define** statement declares a preprocessor string variable or macro, and its value/body is the text after the name up to the end of line.
- Preprocessor can transform the syntax of C/C++ program (**discouraged**).

```
#define Malloc( T ) (T *)malloc( sizeof( T ) )
int *ip = Malloc( int );

#define For( v, N ) for ( unsigned int v = 0; i < N; i += 1 )
For( i, 10 ) { ... }

#define Exit( c ) if ( c ) break
for ( ;; ) {
    ...
  Exit( a > b );
    ...
}
```

- Replace **#define** constants with **enum** (see Section 2.10.1, p. 76) for integral types; otherwise use **const** declarations (see Section 2.4.3, p. 50) (Java **final**).

```
enum { arraySize = 100 };         #define arraySize 100
enum { PageSize = 4 * 1024 };     #define PageSize (4 * 1024)
const double PI = 3.14159;        #define PI 3.14159
int array[arraySize], pageSize = PageSize;
double x = PI;
```

- Use **inline** routines in C/C++ rather that **#define** macros (see page 165).

```
inline int MAX( int a, int b ) { return a > b ? a : b; }
```

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

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

  Initialization value is text after =.

- Same as following **#define**s in a program without changing the program:

```
#define DEBUG 2
#define ASSN 1
```

- **Cannot have both `-D` and #define for the same variable.**

- Predefined preprocessor-variables exist identifying hardware and software environment, e.g., mcpu is kind of CPU.

2.17.3 Conditional Inclusion

- Preprocessor has an **if** statement, which may be nested, to conditionally add/remove code from a program.

- Conditional **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
```

- Like Shell, possible to check if a preprocessor variable is defined or not defined 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 (see Section 2.27, p. 133).
- 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.

2.18 Assertions

- **Assertion**s document program assumptions:
  - pre-conditions – true before a computation (e.g., all values are positive),
  - invariants – true across the computation (e.g., all values during the computation are positive, because only +,*, / operations),
  - post-conditions – true after the computation (e.g., all results are positive).
- Assumptions cannot reflect external usage, where there is no control.
  - E.g., at interface points, a routine call can be made with incorrect values.

- ◦ Checking interface parameters is not an assumption about program behaviour, rather about user behaviour.

- Assertions occur ***after*** usage checks when a program has control over its computation.
  - ◦ E.g., after checking a "car" is passed to a routine to calculate braking distance, an assumption of correct behaviour is a positive braking distance.
  - ◦ Therefore, routine can assert post-condition "braking distance is greater than zero" before returning.

- Macro assert tests a boolean expression representing a logical assumption:

```
#include <cassert>
unsigned int stopping_distance( Car car ) {
    if ( car != ... ) exit( EXIT_FAILURE );  // check parameter

    brakes = ... ;
    assert( brakes > 0 );       // pre-condition

    distance = brakes ... ;
    assert( distance > 0 );     // invariant
    distance = ... ;
    assert( distance > 0 );     // invariant

    distance = ... ;
    assert( distance > 0 );     // post-condition
    return distance;
}
```

- If assert fails (false result), it aborts program and prints expression:

```
a.out: test.cc:19: unsigned int stopping_distance(Car):
    Assertion 'distance > 0' failed.
```

- Use comma expression to add documentation to assertion message.

```
assert( ("Internal error", distance > 0) );
a.out: test.cc:19: unsigned int stopping_distance(Car):
    Assertion ('"Internal error", distance > 0)' failed.
```

- Assertions in **hot spot**, i.e., point of high execution, can significantly increase program cost.

- Compiling a program with preprocessor variable NDEBUG defined removes all asserts.

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

- Therefore, never put computations needed by a program into an assertion.

```
assert( needed_computation(...) > 0 );  // may not be executed
```

2.19 Debugging

- **Debugging** is the process of determining why a program does not 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.

2.19.1 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;
    for ( . . . ) {
        cerr << "x=" << x << ", y=" << y << endl;
        . . .
    }
} else {
    cerr << "a <= b" << endl;
    . . .
}
```

- 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 generate lots 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 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.
- When you go for help, your program should contain debug print-statements to indicate some attempted at understanding the problem.
- Use a preprocessor macro to simplify debug prints for printing entry, intermediate, and exit locations and data:

```
#define DPRT( title, expr ) \
 { std::cerr << #title "\t\"" << __PRETTY_FUNCTION__ << "\" " << \
 expr << " in " << __FILE__ << " at line " << __LINE__ << std::endl; }
```

```
#include <iostream>
#include "DPRT.h"
int test( int a, int b ) {
    DPRT( ENTER, "a:" << a << " b:" << b );
    if ( a < b ) DPRT( a < b, "a:" << a << " b:" << b );
    DPRT( , a + b );                   // empty title
    DPRT( HERE, "" );                  // empty expression
    DPRT( EXIT, a );
    return a;
}
```

```
ENTER "int test(int, int)" a:3 b:4 in test.cc at line 14
a < b   "int test(int, int)" a:3 b:4 in test.cc at line 16
    "int test(int, int)" 7 in test.cc at line 18
HERE   "int test(int, int)"  in test.cc at line 19
EXIT   "int test(int, int)" 3 in test.cc at line 20
```

2.20 Debugger

- An interactive, symbolic **debugger** effectively allows debug print statements to be added and removed to/from a program dynamically.
- Do not rely solely on a debugger to debug a program.
- Some systems do not have 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 a program, it merely helps in the debugging process.
- Therefore, you must have some idea (hypothesis) about what is wrong with a program before starting to look.

2.20.1 GDB

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

```
int r( int a[] ) {
    int i = 100000000;
    a[i] += 1;  // really bad subscript error
    return a[i];
}
int main() {
    int a[10] = { 0, 1 };
    r( a );
}
```

- 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.

GDB Command	Action
<Enter>	repeat last command
run *[shell-arguments]*	start program with shell arguments
backtrace	print current stack trace
print *variable-name*	print value in variable-name
frame *[n]*	go to stack frame n
break *routine / file-name:line-no*	set breakpoint at routine or line in file
info breakpoints	list all breakpoints
delete *[n]*	delete breakpoint n
step *[n]*	execute next n lines (into routines)
next *[n]*	execute next n lines of current routine
continue *[n]*	skip next n breakpoints
list	list source code
quit	terminate gdb

- <Enter> without a command repeats the last command.
- **r**un command begins execution of the program:

```
(gdb) run
Starting program: /u/userid/cs246/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.
  - If program is not compiled with -g flag, only routine names given.

- **ba**cktrace command prints a stack trace of called routines.

```
(gdb) backtrace
#0  0x000106f8 in r (a=0xffbefa08) at test.cc:3
#1  0x00010764 in main () at test.cc:8
```

  - stack has 2 frames main (#1) and r (#0) because error occurred in call to r.

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

```
(gdb) print i
$1 = 100000000
```

- 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
```

- **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
$6 = 7
(gdb) set var a[0] = 3
(gdb) p a[0]
$7 = 3
```

  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.

- **f**rame *[n]* command moves the **current stack frame** to the nth routine call 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:8
8        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.

- To trace program execution, **breakpoint**s are used.

- **b**reak 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 7.
(gdb) break test.cc:3
Breakpoint 2 at 0x106d8: file test.cc, line 3.
```

  - Set breakpoint using routine name or source-file:line-number.
  - **i**nfo breakpoints command prints all breakpoints currently set.

```
(gdb) info break
Num Type           Disp Enb Address      What
1   breakpoint     keep y   0x00010710 in main at test.cc:7
2   breakpoint     keep y   0x000106d8 in r(int*) at test.cc:3
```

- 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: /u/userid/cs246/a.out
Breakpoint 1, main () at test.cc:7
7        int a[10] = { 0, 1 };
(gdb) p a[7]
$8 = 0
```

- Once a breakpoint is reached, execution of the program can be continued in several ways.

- **s**tep *[n]* command executes the next n lines of the program and stops, so control enters routine calls.

```
(gdb) step
8           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) <Enter>
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.
  - If the next line is a routine call, control enters the routine and stops at the first line.

- **n**ext *[n]* command executes the next n lines of the current routine and stops, so routine calls are not entered (treated as a single statement).

```
(gdb) run
...
Breakpoint 1, main () at test.cc:7
7           int a[10] = { 0, 1 };
(gdb) next
8           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 *[n]* command continues execution until the next breakpoint is reached.

```
(gdb) run
...
Breakpoint 1, main () at test.cc:7
7           int a[10] = { 0, 1 };
(gdb) c
Breakpoint 2, r (a=0x7fffffffe7d0) at test.cc:3
3           a[i] += 1;     // really bad subscript error
(gdb) p i
$9 = 100000000
(gdb) set var i = 3
(gdb) c
Continuing.
Program exited normally.
```

- **l**ist command lists source code.

```
(gdb) list
1    int r( int a[] ) {
2          int i = 100000000;
3          a[i] += 1;    // really bad subscript error
4          return a[i];
5    }
6    int main() {
7          int a[10] = { 0, 1 };
8          r( a );
9    }
```

  - 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:7
7          int a[10] = { 0, 1 };
1: a[0] = 67568
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

2.21 Object

- ***Object***-oriented programming was developed in the mid-1960s by Dahl and Nygaard and first implemented in SIMULA67.
- Object programming is based on structures, used for organizing logically related data (see Section 2.10.3, p. 79):

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 a structure by allowing routine members.

- Java has either a basic type or an object, i.e., all routines are embedded in a **struct**/**class** (see Section 2.14, p. 86).

structure form	object form
<pre>struct Complex { double re, im; }; double abs(const Complex &This) { return sqrt(This.re * This.re + This.im * This.im); } Complex x; // structure d = abs(x); // call abs</pre>	<pre>struct Complex { double re, im; double abs() const { return sqrt(re * re + im * im); } }; Complex x; // object d = x.abs(); // call abs</pre>

- ***An object provides both data and the operations necessary to manipulate that data in one self-contained package.***
- Both approaches use routines as an abstraction mechanism to create an interface to the information in the structure.
- Interface separates usage from implementation at the interface boundary, allowing an object's implementation to change without affecting usage.
- E.g., if programmers do not access Complex's implementation, it can change from Cartesian to polar coordinates and maintain same interface.
- ***Developing good interfaces for objects is important.***
  - e.g., mathematical types (like complex) should use value semantics (functional style) versus reference to prevent changing temporary values.

2.21.1 Object Member

- A routine member in a class 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 T * **const** this parameter, implicitly passed to each routine member (like left example).

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

Since implicit parameter "this" is a const pointer, it should be a reference.

- Except for the syntactic differences, the two forms are identical.

- ***The use of implicit parameter* this*, e.g.,* this->f*, is seldom necessary.***
- Member routine declared **const** is read-only, i.e., cannot change member variables.
- 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.
- ***Nesting of object types only allows static not dynamic scoping*** (see Section 2.12, p. 84) (Java allows dynamic scoping).

```
struct Foo {
    int g;
    int r() { . . . }
    struct Bar {                 // nested object type
        int s() { g = 3; r(); }  // disallowed, dynamic reference
    };                           //    to specific object
} x, y, z;
```

References in s to members g and r in Foo disallowed because must know the **this** for specific Foo object, i.e., which x, y or z.

- Extend type Complex by inserting an arithmetic addition operation:

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

- To sum x and y, write x.add(y), which looks different from normal addition, x + y.
- Because addition is a binary operation, add needs a parameter as well as the implicit context in which it executes.
- Like outside a type, C++ allows overloading members in a type.

2.21.2 Operator Member

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

```
struct Complex {
    . . .
    Complex operator+( Complex c ) { // rename add member
        return { re + c.re, im + c.im };
    }
};
```

- Addition routine is called +, and x and y can be added by x.**operator**+(y) or y.**operator**+(x), which looks slightly better.

- Fortunately, C++ implicitly 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;   // rewritten as  x.operator+( y )
cout << "sum:" << sum.re << "+" << sum.im << "i" << endl;
```

2.21.3 Constructor

- A **constructor** member ***implicitly*** performs initialization after object allocation to ensure the object is valid before use.

```
struct Complex {
    double re, im;
    Complex() { re = im = 0.0; } // default constructor
    . . . // other members
};
```

- Constructor member-name is overloaded with structure name.
- Constructor without parameters is the **default constructor**, for initializing a new object.

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.
- Constructor normally initializes members ***not initialized via other constructors***, i.e., some members are objects with their own 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**).
- Never put parentheses to invoke default constructor for declaration.

```
Complex x(); // routine prototype, no parameters returning a complex
```

- Once constructor specified, structure initialization requires a corresponding constructor:

```
struct Complex {
    double re = 0.0, im = 0.0;
    Complex( double r = 0.0, double i = 0.0 ) { re = r; im = i; }
    . . .
};
Complex x, y( 3.2 ), z( 3.2, 4.5 );
Complex x, y = { 3.2 }, z = { 3.2, 4.5 }; // C++11, rewrite to first
Complex x, y{ 3.2 }, z{ 3.2, 4.5 }; // C++11, rewrite to first
```

Use field initialization or default parameters.

- Compiler allocates and initializes.

`Complex x, y( 3.2 ), z( 3.2, 4.5 );`	implicitly rewritten as	`Complex x; x.Complex( 0.0, 0.0 );` `Complex y; y.Complex( 3.2, 0.0 );` `Complex z; z.Complex( 3.2, 4.5 );`

(see declaring stream files page 56)

- Dynamic allocation (same in Java):

```
Complex *x = new Complex; // parentheses/braces optional
Complex *y = new Complex( 3.2, 4.5 );
Complex *z = new Complex{ 3.2 }; // C++11
```

- Constructor may force dynamic allocation when initializating an array of objects.

```
Complex ac[10];          // complex array default initialized to 0.0+0.0i
for ( int i = 0; i < 10; i += 1 ) {
    ac[i] = { i, i + 2.0 }; // assignment, constructor already called
}
Complex *ap[10];         // array of complex pointers
for ( int i = 0; i < 10; i += 1 ) {
    ap[i] = new Complex( i, i + 2.0 );  // initialization, constructor called
}
for ( int i = 0; i < 10; i += 1 ) {
    delete ap[i];
}
#include <memory>
{
    unique_ptr<Complex> uac[10]; // C++11
    for ( int i = 0; i < 10; i += 1 ) {
        // objs[i].reset( new Obj( i ) ); // C++11
        uac[i] = make_unique<Complex>( i, i + 2.0 ); // initialization, C++14
    }
} // automatically delete objs for each uac
```

See Section 2.21.5, p. 112 for difference between initialization and assignment.

- ***If only non-default constructors are specified, i.e., ones with parameters, an object cannot be declared without an initialization value:***

```
struct Foo {
    // no default constructor
    Foo( int i ) { . . . }
};
Foo x;   // disallowed!!!
Foo x( 1 );  // allowed
```

- Unlike Java, constructor cannot be called explicitly in another constructor, so constructor reuse is 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 ); }` `};`

2.21.3.1 Literal

- Constructors can be used to create object literals (see Section 2.13, p. 85):

```
Complex x, y, z;
x = { 3.2 };                    // Complex( 3.2 )
y = x + { 3.2, 4.5 };           // disallowed
y = x + (Complex){ 3.2, 4.5 };  // g++
z = x + Complex( 2 ) + y;       // 2 widened to 2.0, Complex( 2.0 )
```

2.21.3.2 Conversion

- Constructors are implicitly used for conversions (see Section 2.6.1, p. 62):

```
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 literals and types to interact with user-defined types.
- Note, two implicit conversions are performed on variable i in x + i: **int** to **double** and then **double** to Complex.
- Can require only explicit conversions with qualifier **explicit** on constructor:

```
struct Complex {
    // turn off implicit conversion
    explicit Complex( double r = 0.0, double i = 0.0 ) { re = r; im = i; }
    . . .
};
```

- Problem: implicit conversion disallowed for commutative binary operators.
- 1.3 + x, disallowed 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 infix form (see Section 2.15, p. 89):

```
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;                          operator+(x, y)
1.3 + x;      implicitly        operator+(Complex(1.3), x)
x + 1.3;      rewritten as      operator+(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.
- 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?

2.21.4 Destructor

- A **destructor** (finalize in Java) member ***implicitly*** performs uninitialization at object deallocation:

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

- Object type has one destructor; its name is the character "~" followed by the type name (like a constructor).
- Destructor has no parameters nor return type (not even **void**):
- ***Destructor is only necessary if an object is non-contiguous, i.e., composed of multiple pieces within its environment***, e.g., files, dynamically allocated storage, etc.

- A **contiguous object**, like a Complex object, requires no destructor as it is self-contained (see Section 2.27, p. 133 for a version of Complex requiring a destructor).
- Destructor is invoked *before* an object is deallocated, either implicitly at the end of a block or explicitly by a **delete**:

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

- For local variables in a block, destructors ***must be*** called in reverse order to constructors because of dependencies, e.g., y depends on x.
- Destructor is more common in C++ than finalize in Java as no garbage collection in C++.
- ***If an object type performs dynamic storage allocation, it is non-contiguous and 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
    . . .
};
```

  Exception is when the dynamic object is transfered to another object for deallocation.

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

2.21.5 Copy Constructor / Assignment

- There are multiple contexts where an object is copied.
  1. declaration initialization (ObjType obj2 = obj1)
  2. pass by value (argument to parameter)
  3. return by value (routine to temporary at call site)
  4. assignment (obj2 = obj1)
- Cases 1 to 3 involve a newly allocated object with undefined values.
- Case 4 involves an existing object that may contain previously computed values.
- C++ differentiates between these situations: initialization and assignment.

- Constructor with a **const** reference parameter of class type is used for initialization (declarations/parameters/return), called the **copy constructor**:

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

- Declaration initialization:

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

  - "=" is misleading as copy constructor is called not assignment operator.
  - value on the right-hand side of "=" is argument to copy constructor.

- Parameter/return initialization:

```
Complex rtn( Complex a, Complex b ) { . . . return a; }
Complex x, y;
x = rtn( x, y );        // creates temporary before assignment
```

  - parameter is initialized by corresponding argument using its copy constructor:

```
Complex rtn( Complex a, Complex b ) {
    a.Complex( arg1 ); b.Complex( arg2 ); // initialize parameters
    . . .                                 //    with arguments
```

  - temporaries *may* be created for arguments and return value, initialized using copy constructor:

`x = rtn(. . .);`	implicitly rewritten as	`Complex t1( x ), t2( y );` `Complex tr( rtn( t1, t2 ) );` `x.Complex( tr );` *or* `x.Complex( rtn( x, y ) );`

  - Note, assignment converted to initialization.

- Assignment routine is used for assignment:

```
Complex &operator=( const Complex &rhs ) { . . . }
```

  - usually most efficient to use reference for parameter and return type.
  - value on the right-hand side of "=" is argument to assignment operator.

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

- If a copy constructor or assignment operator is not defined, an implicit one is generated that does a **memberwise copy** of each subobject.
  - basic type, **bitwise copy**
  - class type, use class's copy constructor
  - array, each element is copied appropriate to the element type

```
struct B {
    B() { cout << "B() "; }
    B( const B &c ) { cout << "B(&) "; }
    B &operator=( const B &rhs ) { cout << "B= "; }
};
struct D {                  // implicit copy and assignment
    int i;                  // basic type, bitwise
    B b;                    // object type, memberwise
    B a[5];                 // array, element/memberwise
};
int main() {
    D i;      cout << endl; // B′s default constructor
    D d = i; cout << endl;  // D′s default copy-constructor
    d = i;    cout << endl; // D′s default assignment
}
```

outputs the following:

```
b   a                          // member variables
B()  B() B() B() B() B()       // D i
B(&) B(&) B(&) B(&) B(&) B(&)  // D d = i
B=  B= B= B= B= B=             // d = i
```

- Often only a bitwise copy as subobjects have no copy constructor or assignment operator.
- If D defines a copy-constructor/assignment, it overrides that in any subobject.

```
struct D {
    . . . // same declarations
    D() { cout << "D() "; }
    D( const D &c ) : i( c.i ), b( c.b ), a( c.a ) { cout << "D(&) "; }
    D &operator=( const D &rhs ) {
        i = rhs.i;  b = rhs.b;
        for ( int i = 0; i < 5; i += 1 ) a[i] = rhs.a[i]; // array copy
        cout << "D= ";
        return *this;
    }
};
```

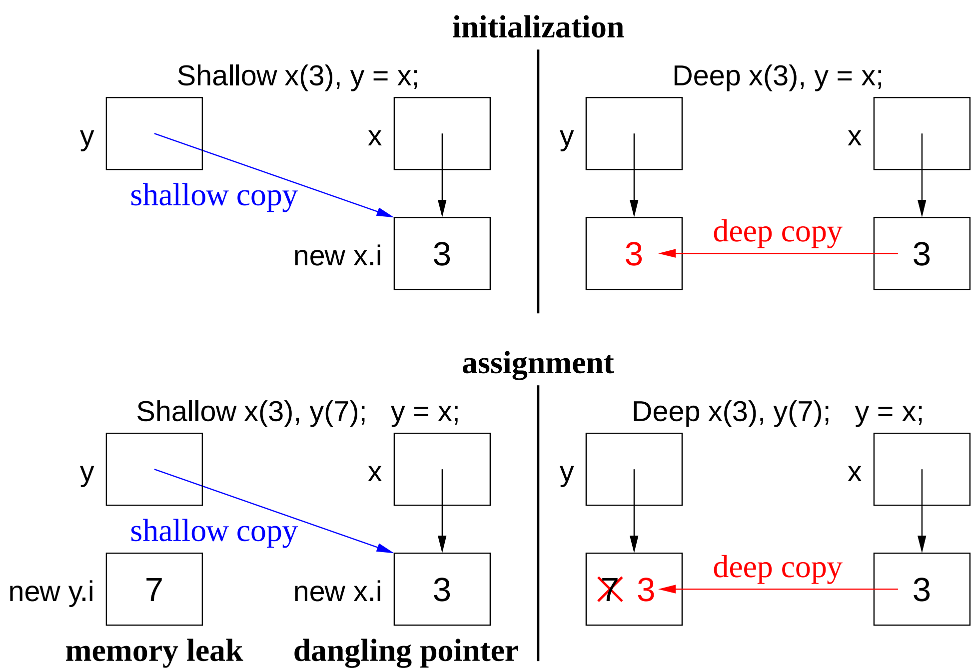
outputs the following:

```
B() B() B() B() B() B() D()            // D i
B(&) B(&) B(&) B(&) B(&) B(&) D(&)     // D d = i
B= B= B= B= B= B= D=                   // d = i
```

Must copy each subobject to get same output.

- When an object type has pointers, it is often necessary to do a deep copy, i.e, copy the contents of the pointed-to storage rather than the pointers (see also Section 2.27, p. 133).

```
struct Shallow {
    int *i;
    Shallow( int v ) { i = new int; *i = v; }
    ~Shallow() { delete i; }
};
struct Deep {
    int *i;
    Deep( int v ) { i = new int;  *i = v; }
    ~Deep() { delete i; }
    Deep( Deep &d ) { i = new int; *i = *d.i; }    // copy value
    Deep &operator=( const Deep &rhs ) {
        *i = *rhs.i;  return *this;                // copy value
    }
};
```



- For shallow copy:
  - memory leak occurs on the assignment
  - dangling pointer occurs after x or y is deallocated; when the other object is deallocated, it reuses this pointer to delete the same storage.
- Deep copy does not change the pointers only the values associated within the pointers.
- Beware **self-assignment** for variable-sized types:

```
struct Varray {                     // variable-sized array
    unsigned int size;
    int *a;
    Varray( unsigned int s ) { size = s; a = new int[size]; }
    . . . // other members
    Varray &operator=( const Varray &rhs ) {  // deep copy
        delete [] a;                // delete old storage
        size = rhs.size;            // set new size
        a = new int[size];          // create storage for new array
        for ( unsigned int i = 0; i < size; i += 1 ) // copy values
            a[i] = rhs.a[i];
        return *this;
    }
};
Varray x( 5 ), y( 10 );
x = y;   // works
y = y;   // fails
```

- How can this problem be fixed?
- Which pointer problem is this, and why can it go undetected?
- For deep copy, it is often necessary to define a equality operator (**operator**==) performing a deep compare, i.e., compare values not pointers.

2.21.6 Initialize const / Object Member

- C/C++ **const** members and local objects of a structure must be initialized at declaration:

```
struct Bar {
    Bar( int i ) {}
    // no default constructor
} bar( 3 ), baz( 4 );
struct Foo {
    const int i = 3;  // C++11
    Bar * const p = &bar;
    Bar &rp = bar;
    Bar b = { 7 };  // b( 7 )
};
Foo w1,
     w2 = { 2, &baz, baz, 9 }, // disallowed, no constructor
     w3( 2, &baz );                     // disallowed, no constructor
```

- Add constructor:

```
Foo( const int i = 3, Bar * const p = &bar, Bar &rp = bar, Bar b = { 7 } ) {
    Foo::i = i;          // disallowed
    Foo::p = p;
    Foo::rp = rp;
    Foo::b = b;
}
```

- Assignment disallowed because constants could be used before initialization:

```
cout << i << endl;    // no value in constant
Foo::i = i;
```

- Special syntax to initialize at point of declaration.

```
Foo( const int i = 3, Bar * const p = &bar, Bar &rp = bar, Bar b = { 7 } )
        : i( i ), p( p ), rp( rp ), b( b ) { // special initialization syntax
    cout << i << endl;  // now value in constant
}
```

- Ensures **const**/object members are initialized before used in constructor body.

- ***Must be initialized in declaration order to prevent use before initialization.***

- Syntax may also be used to initialize any local members:

```
struct Foo {
    Complex c;
    int k;
    Foo() : c( 1, 2 ), k( 14 ) {    // initialize c, k
        c = Complex( 1, 2 );        // or assign c, k
        k = 14;
    }
};
```

  Initialization may be more efficient versus default constructor and assignment.

2.21.7 Static Member

- Static data-member creates a single instance for object type versus for object instances.

```
struct Foo {
    static int cnt;    // one for all objects
    int i;             // one per object
    . . .
};
```

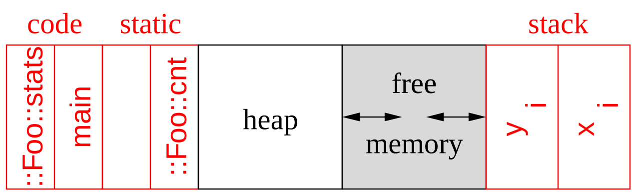
  - exist even if no instances of object exist
  - ***must still be declared (versus defined in the type) in a .cc file.***
  - allocated in static block not in object.

- Static routine-member, used to access static data-members, has no **this** parameter (i.e., like a regular routine)

- E.g., count the number of Foo objects created.

```
int cnt;

void stats() {
	cout << cnt;
}
struct Foo {
	int i;
	Foo() {
		::cnt += 1;
		::stats();
	}
};

int main() {
	Foo x, y;
}
```

```
struct Foo {
	static int cnt;
	int i;
	static void stats() {
		cout << cnt;  // allowed
		i = 3;        // disallowed
		mem();        // disallowed
	}
	Foo() {
		cnt += 1;     // allowed
		stats();      // allowed
	}
};
int Foo::cnt; // declaration (optional initialization)
int main() {
	Foo x, y;
}
```



- Object members mem can reference j and rtn in static block.
- Static member rtn not logically nested in type foo, so it cannot reference members i and mem.

2.22 Random Numbers

- **Random numbers** are values generated independently, i.e., new values do not depend on previous values (independent trials).
- E.g., lottery numbers, suit/value of shuffled cards, value of rolled dice, coin flipping.
- While programmers spend much time ensuring computed values are not random, random values are useful:
  - gambling, simulation, cryptography, games, etc.
- **Random-number generator** is an algorithm computing independent values.
- If algorithm uses deterministic computation (predictable sequence), it generates **pseudo random-numbers** versus “true” random numbers.
- All **pseudo random-number generators** (PRNG) involve some technique that scrambles the bits of a value, e.g., multiplicative recurrence:

```
seed_ = 36969 * (seed_ & 65535) + (seed_ >> 16); // scramble bits
```

- Multiplication of large values adds new least-significant bits and drops most-significant bits.

bits 63-32	bits 31-0
0	3e8e36
5f	718c25e1
ad3e	7b5f1dbe
bc3b	ac69ff19
1070f	2d258dc6

- By dropping bits 63-32, bits 31-0 become scrambled after each multiply.

- E.g., class PRNG generates a ***fixed*** sequence of LARGE random values that repeats after 2^{32} values (but might repeat earlier):[2]

```
class PRNG {
    uint32_t  seed_;     // same results on 32/64-bit architectures
  public:
    PRNG( uint32_t  s = 362436069 ) {
        seed_ = s;                       // set seed
    }
    uint32_t seed() {                    // read seed
        return seed_;
    }
    void seed( uint32_t  s ) {           // reset seed
        seed_ = s;                       // set seed
    }
    uint32_t operator()() {              // [0,UINT_MAX]
        seed_ = 36969*(seed_ & 65535)+(seed_ >> 16); // scramble bits
        return seed_;
    }
    uint32_t operator()( uint32_t u ) {    // [0,u]
        return operator()() % (u + 1);     // call operator()()
    }
    uint32_t operator()( uint32_t l, uint32_t u ) {  // [l,u]
        return operator()( u - l ) + l;     // call operator()( uint32_t )
    }
};
```

- Creating a member with the function-call operator name, (), (**functor**) allows these objects to behave like a routine.

```
PRNG prng;       // often create single generator
prng();          // [0,UINT_MAX]
prng( 5 );       // [0,5]
prng( 5, 10 );   // [5,10]
```

- Large values scaled using modulus. e.g., random number between 5-21:

[2] http://www.bobwheeler.com/statistics/Password/MarsagliaPost.txt

```
for ( int i = 0; i < 10; i += 1 ) {
    cout << prng() % 17 + 5 << endl; // values 0-16 + 5 = 5-21
    cout << prng( 16 ) + 5 << endl;
    cout << prng( 5, 21 ) << endl;
}
```

- Initializing PRNG with external "seed" generates different sequence:

```
PRNG prng( getpid() );   // process id of program
prng.seed( time() );     // current time
```

- **#include** <cstdlib> provides C random routines srand and rand.

```
srand( getpid() );       // seed random genrator
r = rand();              // obtain next random value
```

2.23 System Modelling

- **System modelling** involves describing a complex system in an abstract way to help understand, design and construct the system.
- Modelling is useful at various stages:
  - analysis : system function, services, requirements (outline for design)
  - design : system parts/structure, interactions, behaviour (outline for programming)
  - programming : converting model into implementation
- Model grows from nothing to sufficient detail to be transformed into a functioning system.
- Model provides high-level documentation of the system for understanding (education) and for making changes in a systematic manner.
- Top-down successive refinement is a foundational mechanism used in system design.
- 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:
  - **sketch** out high-level design or complex parts of a system,
  - **blueprint** the system abstractly with high accuracy,

- ○ **generate** interfaces/code directly.

- Key advantage is design tool provides a generic, abstract model of a system, which is transformable into different formats.

- Key disadvantage is design tool seldom linked to implementation mechanism so two often differ. **(CODE = TRUTH)**

- Currently, UML is the most popular design tool.

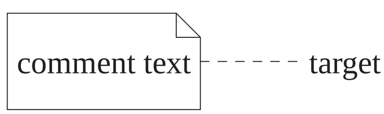
2.23.1 UML

- **Unified Modelling Language** (UML) is a graphical notation for describing and designing software systems, with emphasis on the object-oriented style.

- UML modelling has multiple viewpoints:
  - ○ **class model** : describes static structure of the system for creating objects
  - ○ **object model** : describes dynamic (temporal) structure of system objects
  - ○ **interaction model** : describes the kinds of interactions among objects

  Focus on class and object modelling.

- Note / comment



- **Classes diagram** defines class-based modelling, where a class is a type for instantiating objects.

- Class has a name, attributes and operations, and may participate in inheritance hierarchies (see Section 2.29.3, p. 146).

class name	Person	
	- name : String	
attributes	- age : Integer	optional
(data)	- sex : Boolean	
	- owns : Car [0..5]	
	+ getName : String	
operations	+ getAge : Integer	optional
(routines)	+ getCars : Car [0..5]	
	+ buy(in car : Car, inout card : CreditCard) : Boolean	

- **Attribute** describes a property in a class.

  [visibility] name [“:” [type] [“[” multiplicity “]”] [“=” default]]

- ○ visibility : access to property
  $+ \Rightarrow$ public, $- \Rightarrow$ private, # $\Rightarrow$ protected, $\sim \Rightarrow$ package
- ○ name : identifier for property (like field name in structure)
- ○ type : kind of property Boolean, Integer, Float, String, class-name
- ○ multiplicity : cardinality for instantiation of property: $0..(N|*)$, 0 to N or unlimited, N short for $N..N$, $*$ short for $0..*$, **defaults to 1**
- ○ default : expression that evaluates to default value (or values) for property

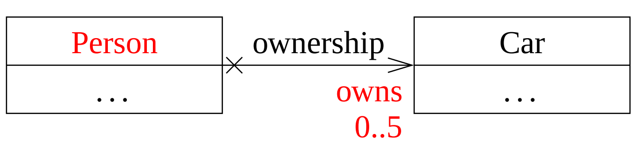
- **operation** : action invoked in context of object from the class
  [visibility] name ["(" [parameter-list] ")"] [":" return-type] ["[" multiplicity "]"]
  - ○ visibility : access to operation
    $+ \Rightarrow$ public, $- \Rightarrow$ private, # $\Rightarrow$ protected, $\sim \Rightarrow$ package
  - ○ name : identifier for operation (like method name in structure)
  - ○ parameter-list : comma separated list of input/output types for operation
    [direction] parameter-name ":" type ["[" multiplicity "]"]
    ["=" default] ["{" modifier-list "}"]]
  - ○ direction : direction of parameter data flow
    "in" (default) | "out" | "inout"
  - ○ return-type : output type from operation

- **Only specify attributes/operations useful in modelling: no flags, counters, temporaries, constructors, helper routines, etc.**

- Attribute with type other than basic type has an **association**.

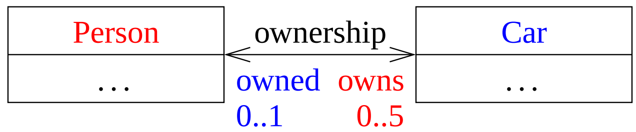
  - ○ Class Person has attribute owns with multiplicity constraint 0..5 forming unidirectional association with class Car, i.e., person owns (has) 0 to 5 cars.

- Alternatively, association can be represented via a line (possibly named):



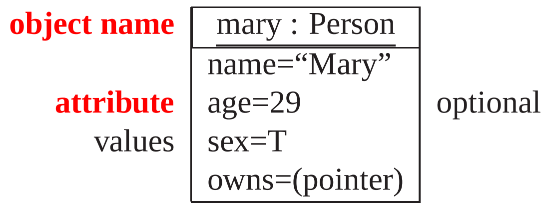
  - ○ Class Person has attribute owns with multiplicity constraint 0..5 (at target end) forming a unidirectional association with class Car and association is named "ownership".

- Association can also be bidirectional.



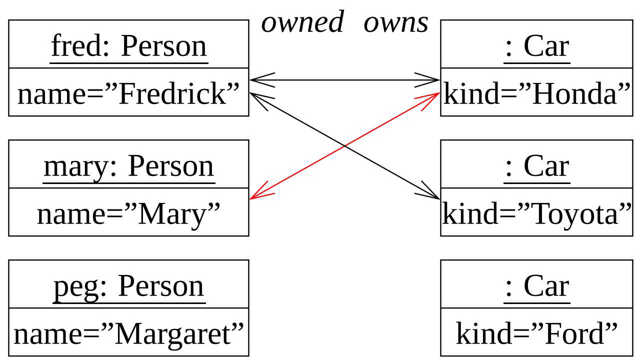
  - Association "ownership" also has class Car having attribute owned with multiplicity constraint 0..1 person, i.e., a car can be unowned or owned by 1 person.

- If UML graph is cluttered with lines, create association in class rather than using a line.
  - E.g., if 20 classes associated with Car, replace 20 lines with attributes in each class.
- Alternatively, multiple lines to same aggregate may be merged into a single segment.
  - Any adornments on that segment apply to all of the aggregation ends.
- $<$ (arrowhead) $\Rightarrow$ **navigable**:
  - instances of association can be accessed efficiently at the association end (arrowhead) (car is accessible from person)
  - opposite association end "owns" the association's implementation (person has a car)
- X $\Rightarrow$ not navigable.
- Adornments options:
  - show all arrows and Xs (completely explicit)
  - suppress all arrows and Xs $\Rightarrow$ no inference about navigation
    often convenient to suppress some of the arrows/Xs and only show special cases
  - show only unidirectional association arrows, and suppress bidirectional associations
    $\Rightarrow$ two-way navigability cannot be distinguished from no navigation at all, but latter case occurs rarely in practice.
- Navigability may be implemented in a number of ways:
  - pointer/reference from one object to another
  - elements in arrays
- **Object diagram** : is a snaphot of class instances at one moment during execution.
- Object can specify values of class : "name : class-type" (underlined), attribute values.



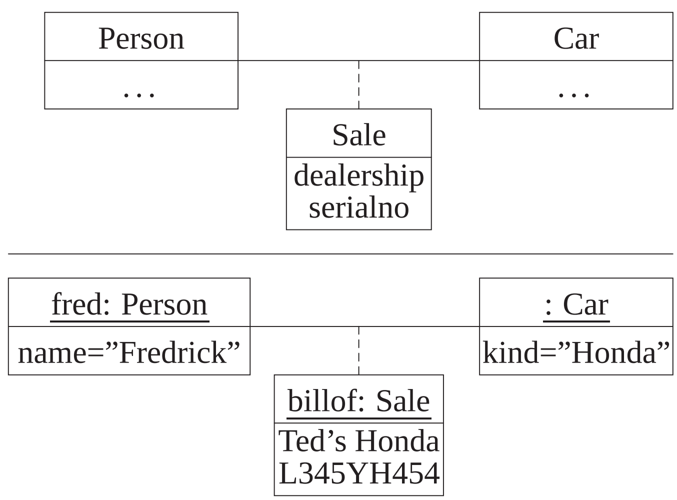
Object may not have a name (dynamically allocated).

- Objects associated with "ownership" are linked.



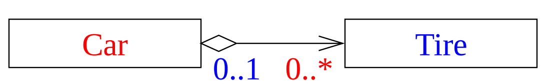
Which associations are valid/invalid/missing?

- **Association Class** : optional aspects of association (dashed line).



  - cars sold through dealership (versus gift) need bill of sale
  - association class cannot exist without association (no owner)

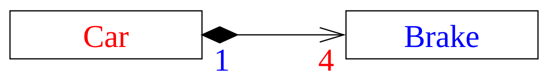
- **Aggregation** (◇) is an association between an aggregate attribute and its parts.



- car can have 0 or more tires and a tire can only be on 0 or 1 car
- aggregate may not create/destroy its parts, e.g., many different tires during car's lifetime and tires may exist after car's lifetime (snow tires).

```
class Car {
    Tires *tires[4];  // array of pointers to tires
```

- **Composition** (♦) is a stronger aggregation where a part is included in at most one composite at a time.

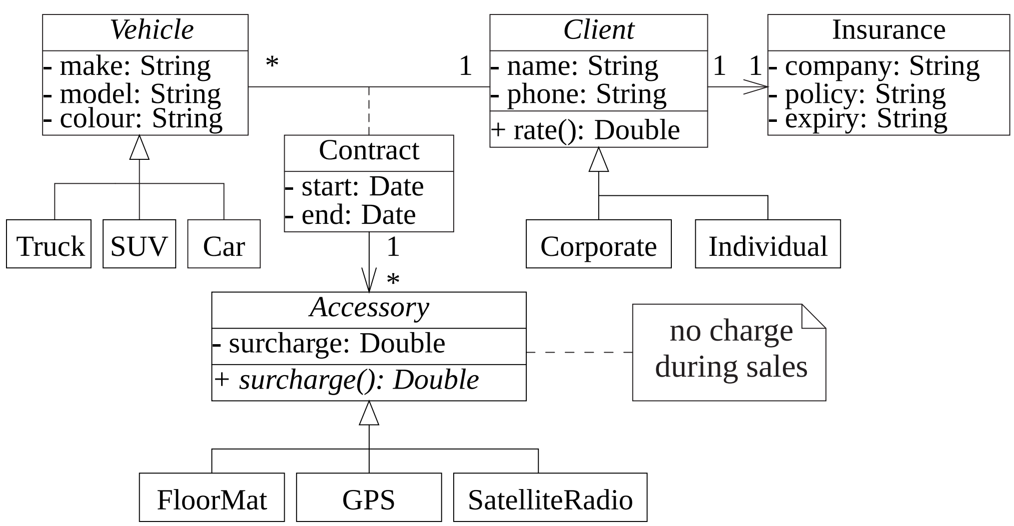


- car has 4 brakes and each brake is on 1 car
- composite aggregate often does create/destroy its parts, i.e., same brakes for lifetime of car and brakes deleted when car deleted (unless brakes removed at junkyard)

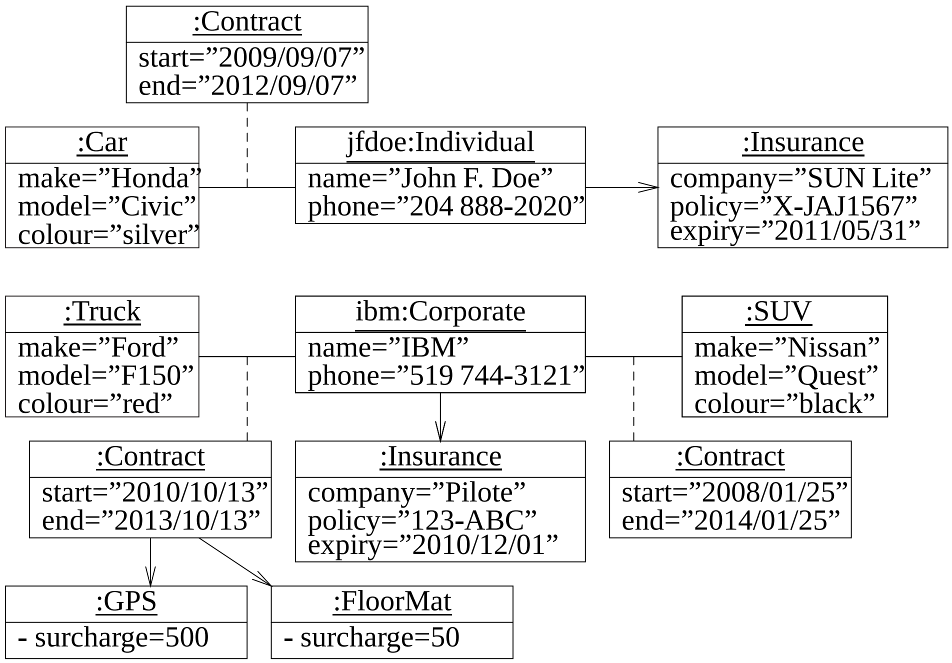
```
class Car {
    DiscBrake brakes[4];  // array of brakes
```

- UML has many more facilities, supporting very complex descriptions of relationships among entities.
  - VERY large visual mechanisms, with several confusing graphical representations.
- **UML diagram is too complex if it contains more than about 25 boxes.**

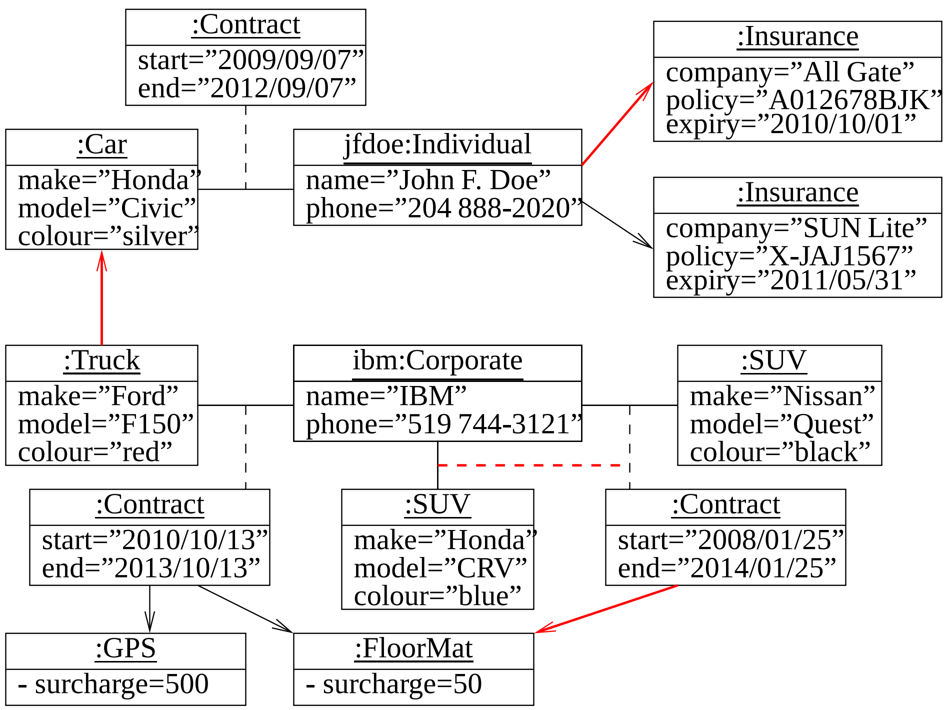
Classes Diagram



Object Diagram



Invalid Object Diagram



2.24 Singleton Pattern

- **singleton** patterns single instance of class

```
#include <Database.h>
class Database {
    static Relational_Database *database; // 0 initialized => NULL
  public:
    void Database() {
        if ( database == NULL ) database = new Relational_Database;
    }
    . . . // members to access database
};
```

- Allow each users to have they own declaration but still access same value.

```
Database database; // user 1
Database db;       // user 2
Database info;     // user 3
```

- Alternative is global variable, which forces name and may violate abstraction.

2.25 Declaration Before Use

- **Declaration Before Use** (DBU) means a variable declaration must appear before its usage in a block.
- In theory, a compiler could handle some DBU situations:

```
{
    cout << i << endl;    // prints 4 ?
    int i = 4;            // declaration after usage
}
```

  but ambiguous cases make this impractical:

```
int i = 3;
{
    cout << i << endl;    // which i?
    int i = 4;
    cout << i << endl;
}
```

- 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
}
```

Caution: these calls cause infinite recursion as there is no base case.

- Cannot type-check the call to g in f to ensure matching number and type of arguments and the return value is used correctly.
- Interchanging the two routines does not solve the problem.
- A **forward declaration** introduces a routine's type (called a **prototype**/**signature**) 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.

```
int main();             // forward declarations, any order
void g( int i );
void f( int i );
int main() {            // actual declarations, any order
    f( 5 );
    g( 4 );
}
void g( int i ) { ... }
void f( int i ) { ... }
```

- E.g., allowing main routine to appear first, and for separate compilation (see Section 2.27, p. 133).
- Like Java, C++ does not always require DBU within a type:

Java	C++
`class T {` `    void f() { c = Colour.R; g(); }` `    void g() { c = Colour.G; f(); }` `    Colour c;` `    enum Colour { R, G, B };` `};`	`void g() {}  // not selected by call in T::f` `struct T {` `    void f() { c = R; g(); }  // c, R, g not DBU` `    void g() { c = G; f(); }  // c, G not DBU` `    enum Colour { R, G, B };  // type must be DBU` `    Colour c;` `};`

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

Java	C++
`class T1 {` `    T2 t2;` `    T1() { t2 = new T2(); }` `};` `class T2 {` `    T1 t1;` `    T2() { t1 = new T1(); }` `};` `T1 t1 = new T1();`	`struct T1 {` `    T2 t2; // DBU failure, T2 size?` `};` `struct T2 {` `    T1 t1;` `};` `T1 t1;`

Caution: these types cause infinite expansion as there is no base case.

- Java version compiles because t1/t2 are references not objects, and Java can look ahead at T2; C++ version disallowed because DBU on T2 means it does not know the size of T2.
- An object declaration and usage requires the object's size and members so storage can be allocated, initialized, and usages type-checked.
- Solve using Java approach: break definition cycle using a forward declaration and pointer.

Java	C++
 `class T1 {` `    T2 t2;` `    T1() { t2 = new T2(); }` `};` `class T2 {` `    T1 t1;` `    T2() { t1 = new T1(); }` `};`	`struct T2;   // forward` `struct T1 {` `    T2 &t2; // pointer, break cycle` `    T1() : t2( *new T2 ) {} // DBU failure, size?` `};` `struct T2 {` `    T1 t1;` `};`

- Forward declaration of T2 allows the declaration of variable T1::t2.
- Note, a forward type 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.
- C++'s solution still does not work as the constructor cannot use type T2.
- Use forward declaration and syntactic trick to move member definition ***after both types are defined***:

```
struct T2;   // forward
struct T1 {
    T2 &t2; // pointer, break cycle
    T1();   // forward declaration
};
struct T2 {
    T1 t1;
};
T1::T1() : t2( *new T2 ) {} // can now see type T2
```

- Use of qualified name `T1::T1` allows a member to be logically declared in `T1` but physically located later (see Section 2.27, p. 133).

2.26 Encapsulation

- **Encapsulation** hides implementation to support abstraction (**access control**).
- Access control applies to types NOT objects, i.e., all objects of the same type have identical levels of encapsulation.
- ***Abstraction and encapsulation are neither essential nor required to develop software.***
- E.g., programmers could follow a convention of not directly accessing the implementation.
- However, relying on programmers to follow conventions is dangerous.
- **Abstract data-type** (ADT) is a user-defined type practicing abstraction and encapsulation.
- 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 (see Section 2.27, p. 133).
- C++ features work both within and among source files.
- C++ provides 3 levels of access 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.
- Encapsulation supports abstraction by making implementation members private and interface members public.
- ***Note, private/protected members are still visible to programmer but inaccessible*** (see page 138 for invisible implementation).

```
struct Complex {
  private:
    double re, im;  // cannot access but still visible
  public:
    // interface routines
};
```

- **struct** has an implicit **public** inserted at beginning, i.e., by default all members are public.
- **class** has an implicit **private** inserted at beginning, i.e., by default all members are private.

```
struct S {         | class C {
  // public:       |   // private:
    int z;         |     int x;
  private:         |   protected:
    int x;         |     int y;
  protected:       |   public:
    int y;         |     int z;
};                 | };
```

- Use encapsulation to preclude object copying by hiding copy constructor and assignment operator:

```
class Lock {
    Lock( const Lock & );    // definitions not required
    Lock &operator=( Lock & );
  public:
    Lock() {...}
    ...
};
void rtn( Lock f ) {...}
Lock x, y;
rtn( x );  // disallowed, no copy constructor for pass by value
x = y;     // disallowed, no assignment operator for assignment
```

- Prevent object forgery (lock, boarding-pass, receipt) or copying that does not make sense (file, database).
- Encapsulation introduces problems when factoring for modularization, e.g., previously accessible data becomes inaccessible.

```
class Complex {
    double re, im;
  public:
    Complex operator+(Complex c);
    . . .
};
ostream &operator<<(ostream &os,
                    Complex c);
```

```
class Cartesian { // implementation type
    double re, im;
};
class Complex {
    Cartesian impl;
  public:
    . . .
};
Complex operator+(Complex a, Complex b);
ostream &operator<<(ostream &os,
                    Complex c);
```

- Implementation is factored into a new type Cartesian, "+" operator is factored into a routine outside and output "<<" operator must be outside (see Section 2.21.3.2, p. 110).
- Both Complex and "+" operator need to access Cartesian implementation, i.e., re and im.
- Creating get and set interface members for Cartesian provides no advantage over full access.
- C++ provides a mechanism to state that an outside type/routine is allowed access to its implementation, called **friendship** (similar to package visibility in Java).

```
class Complex; // forward
class Cartesian { // implementation type
    friend Complex operator+( Complex a, Complex b );
    friend ostream &operator<<( ostream &os, Complex c );
    friend class Complex;
    double re, im;
};
class Complex {
    friend Complex operator+( Complex a, Complex b );
    friend ostream &operator<<( ostream &os, Complex c );
    Cartesian impl;
  public:
    . . .
};
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";
}
```

- Cartesian makes re/im accessible to friends, and Complex makes impl accessible to friends.
- Alternative design is to nest the implementation type in Complex and remove encapsulation (use **struct**).

```
class Complex {
    friend Complex operator+( Complex a, Complex b );
    friend ostream &operator<<( ostream &os, Complex c );
    struct Cartesian { // implementation type
        double re, im;
    } impl;
  public:
    Complex( double r = 0.0, double i = 0.0 ) {
        impl.re = r; impl.im = i;
    }
};
. . .
```

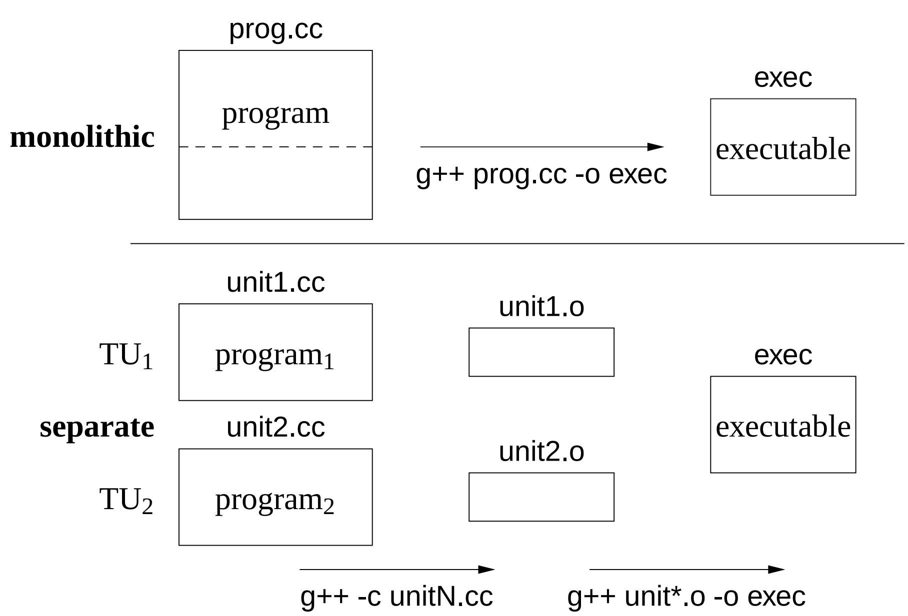
Complex makes Cartesian, re, im and impl accessible to friends.

2.27 Separate Compilation

- As program size increases, so does cost of compilation.
- **Separate compilation** divides a program into units, where each unit can be independently compiled.
- Advantage: saves time by recompiling only program unit(s) that change.
  - In theory, if an expression is changed, only that expression needs to be recompiled.
  - In practice, compilation unit is coarser: **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.
  - ***So a TU should not be too big and not be too small.***
- Disadvantage: TUs depend on each other because a program shares many forms of information, especially types (done automatically in Java).
  - Hence, need mechanism to **import** information from referenced TUs and **export** information needed to referencing TUs.
- For example, simple program in file prog.cc using complex numbers:

```
prog.cc
    #include <iostream>          // import
    #include <cmath>             // sqrt
    using namespace std;
    class Complex {
        friend Complex operator+( Complex a, Complex b );
        friend ostream &operator<<( ostream &os, Complex c );
        static int objects;      // shared counter
        double re, im;
      public:
        Complex( double r = 0.0, double i = 0.0 ) { objects += 1; ...}
        double abs() const { return sqrt( re * re + im * im ); };
        static void stats() { cout << objects << endl; }
    };
    int Complex::objects;        // declare
    Complex operator+( Complex a, Complex b ) {...}
    ... // other arithmetic and logical operators
    ostream &operator<<( ostream &os, Complex c ) {...}
    const Complex C_1( 1.0, 0.0 );
    int main() {
        Complex a( 1.3 ), b( 2., 4.5 ), c( -3, -4 );
        cout << a + b + c + C_1 << c.abs() << endl;
        Complex::stats();
    }
```

- TU prog.cc has references to items in iostream and cmath.
- As well, there are many references within the TU, e.g., main references Complex.
- Subdividing program into TUs in C/C++ is complicated because of import/export mechanism.



- TU_i is NOT a program; program formed by combining TUs.

- Compile each TU$_i$ with -c compiler flag to generate executable code in .o file (Java has .class file).

```
$ g++ -c unit1.cc ... // compile only modified TUs
```

generates files unit1.o containing a compiled version of source code.

- Combine TU$_i$ with -o compiler flag to generate executable program.

```
$ g++ unit*.o -o exec   // create new excutable program "exec"
```

- Separate original program into two TUs in files complex.cc and prog.cc:

```
complex.cc
    #include <iostream>          // import
    #include <cmath>
    using namespace std;
    class Complex {
        friend Complex operator+( Complex a, Complex b );
        friend ostream &operator<<( ostream &os, Complex c );
        static int objects;      // shared counter
        double re, im;           // implementation
      public:
        Complex( double r = 0.0, double i = 0.0 ) { objects += 1; ...}
        double abs() const { return sqrt( re * re + im * im ); }
        static void stats() { cout << objects << endl; }
    };
    int Complex::objects;        // declare
    Complex operator+( Complex a, Complex b ) {...}
    ... // other arithmetic and logical operators
    ostream &operator<<( ostream &os, Complex c ) {...}
    const Complex C_1( 1.0, 0.0 );
```

TU complex.cc has references to items in iostream and cmath.

```
prog.cc
    int main() {
        Complex a( 1.3 ), b( 2., 4.5 ), c( -3, -4 );
        cout << a + b + c + C_1 << c.abs() << endl;
        Complex::stats();
    }
```

TU prog.cc has references to items in iostream and complex.cc.

- How can TU prog.cc access Complex? By importing description of Complex.

- How are descriptions imported?

  TU imports information using preprocessor **#include** (see Section 2.17.1, p. 95).

- Why not include complex.cc into prog.cc?

  Because all of complex.cc is compiled each time prog.cc is compiled so there is no advantage to the separation (program is still monolithic).

- Hence, must separate complex.cc into interface for import and implementation for code.
- Complex interface placed into file complex.h, for inclusion (import) into TUs.

complex.h
```
#ifndef __COMPLEX_H__
#define __COMPLEX_H__  // protect against multiple inclusion
#include <iostream>    // import
// NO "using namespace std", use qualification to prevent polluting scope
class Complex {
    friend Complex operator+( Complex a, Complex b );
    friend std::ostream &operator<<( std::ostream &os, Complex c );
    static int objects;     // shared counter
    double re, im;          // implementation
  public:
    Complex( double r = 0.0, double i = 0.0 );
    double abs() const;
    static void stats();
};
extern Complex operator+( Complex a, Complex b );
. . . // other arithmetic and logical operator descriptions
extern std::ostream &operator<<( std::ostream &os, Complex c );
extern const Complex C_1;
#endif // __COMPLEX_H__
```

- (Usually) no code, just descriptions : preprecessor variables, C/C++ types and forward declarations (see Section 2.25, p. 127).
- **extern** qualifier means variable or routine definition is located elsewhere (not for types).
- Complex implementation placed in file complex.cc.

complex.cc
```
#include "complex.h"       // do not copy interface
#include <cmath>           // import
using namespace std;       // ok to pollute implementation scope
int Complex::objects;      // defaults to 0
void Complex::stats() { cout << Complex::objects << endl; }
Complex::Complex( double r, double i ) { objects += 1; . . .}
double Complex::abs() const { 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";
}
const Complex C_1( 1.0, 0.0 );
```

- Implementation is composed of actual declarations and code.
- ***.cc file includes the .h file so that there is only one copy of the constants, declarations, and prototype information.***

- Why is **#include** <cmath> in complex.cc instead of complex.h?
- Compile TU complex.cc to generate complex.o.

```
$ g++ -c complex.cc
```

- What variables/routines are exported from complex.o?

```
$ nm -C complex.o | egrep ' T | B '
C_1
Complex::stats()
Complex::objects
Complex::Complex(double, double)
Complex::Complex(double, double)
Complex::abs() const
operator<<(std::ostream&, Complex)
operator+(Complex, Complex)
```

- In general, type names are not in the .o file?
- To compile prog.cc, it must import complex.h

```
prog.cc
    #include "complex.h"
    #include <iostream>          // included twice!
    using namespace std;

    int main() {
        Complex a( 1.3 ), b( 2., 4.5 ), c( -3, -4 );
        cout << a + b + c + C_1 << c.abs() << endl;
        Complex::stats();
    }
```

- Why is **#include** <iostream> in prog.cc when it is already imported by complex.h?
- Compile TU prog.cc to generate prog.o.

```
$ g++ -c prog.cc
```

- Link together TUs complex.o and prog.o to generate exec.

```
$ g++ prog.o complex.o -o exec
```

- ***All .o files MUST be compiled for the same hardware architecture, e.g., all x86.***
- To hide global variables/routines (but NOT class members) in TU, qualify with **static**.

```
complex.cc
    ...
    static Complex C_1( 1.0, 0.0 );
    static Complex operator+( Complex a, Complex b ) {...}
    static ostream &operator<<( ostream &os, Complex c ) {...}
```

  - here **static** means linkage NOT allocation (see Section 2.21.7, p. 117).

- These variables/routines are now only accessible in the TU! (not good for users of complex)

- Encapsulation is provided by giving a user access to the include file(s) (.h) and the compiled source file(s) (.o), but not the implementation in the source file(s) (.cc).

- Note, while the .h file encapsulates the implementation, the implementation is still visible.

- To completely hide the implementation requires a (more expensive) reference:

complex.h

```
#ifndef __COMPLEX_H__
#define __COMPLEX_H__  // protect against multiple inclusion
#include <iostream>            // import
// NO "using namespace std", use qualification to prevent polluting scope
class Complex {
        friend Complex operator+( Complex a, Complex b );
        friend std::ostream &operator<<( std::ostream &os, Complex c );
        static int objects;          // shared counter
        struct ComplexImpl;         // hidden implementation, nested class
        ComplexImpl &impl;          // indirection to implementation
    public:
        Complex( double r = 0.0, double i = 0.0 );
        Complex( const Complex &c );  // copy constructor
        ~Complex();                  // destructor
        Complex &operator=( const Complex &c ); // assignment operator
        double abs() const;
        static void stats();
};
extern Complex operator+( Complex a, Complex b );
extern std::ostream &operator<<( std::ostream &os, Complex c );
extern const Complex C_1;
#endif // __COMPLEX_H__
```

```
complex.cc
    #include "complex.h"       // do not copy interface
    #include <cmath>           // import
    using namespace std;       // ok to pollute implementation scope
    int Complex::objects;      // defaults to 0
    struct Complex::ComplexImpl { double re, im; }; // implementation
    Complex::Complex( double r, double i ) : impl(*new ComplexImpl) {
        objects += 1; impl.re = r; impl.im = i;
    }
    Complex::Complex( const Complex &c ) : impl(*new ComplexImpl) {
        objects += 1; impl.re = c.impl.re; impl.im = c.impl.im;
    }
    Complex::~Complex() { delete &impl; }
    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);}
    void Complex::stats() { cout << Complex::objects << endl; }
    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 are used because complex objects now contain a reference pointer to the implementation (see page 114).

2.28 Design Quality

- System design is a general plan for attacking a problem, but leads to multiple solutions.
- Need the ability to compare designs.
- 2 measures: coupling and cohesion
- Low (loose) coupling is a sign of good structured and design; high cohesion supports readability and maintainability.

2.28.1 Coupling

- **Coupling** measures the degree of interdependence among programming "modules".
- Aim is to achieve lowest coupling or highest independence (i.e., each module can stand alone or close to it).
- A module can be read and understood as a unit, so that changes have minimal effect on other modules and possible to isolate it for testing purposes (like stereo components).
- 5 types of coupling in order of loose to tight (low to high):
    1. **Data** : modules communicate using arguments/parameters containing minimal data.

    - E.g., `sin( x )`, `avg( marks )`

2. **Stamp** : modules communicate using only arguments/parameters containing extra data.
    - E.g., pass aggregate data (array/structure) with some elements/fields unused
    - problem: accidentally change other data
    - modules may be less general (e.g., average routine passed an array of records)
    - stamp coupling is common because data grouping is more important than coupling
3. **Control** : pass data using arguments/parameters to effect control flow.
    - E.g., module calculate 2 different things depending on a flag
    - bad when flag is passed down, worse when flag is passed up
4. **Common** : modules share global data.
    - cannot control access since scope rule allows many modules to access the global variables
    - difficult to find all references reading/writing global variables
5. **Content** : modules share information about type, size and structure of data, or methods of calculation
    - changes effect many different modules (good/bad)
    - avoid **friend** routine/class unless friend module is logically nested but extracted for technical reasons.

2.28.2 Cohesion

- **Cohesion** measures degree of association among elements within a module (how focused).
- Elements can be a statement, group of statements, or calls to other modules.
- Alternate names for cohesion: binding, functionality, modular strength.
- Highly cohesive module has strongly and genuinely related elements.
- If modules have low cohesion (module elements are related) $\Rightarrow$ tight coupling.
- If modules have high cohesion (module elements are NOT related) $\Rightarrow$ loose coupling.
- 7 types of cohesion (high to low):
    1. **Functional** : modules elements all contribute to computation of one and only one problem related task (Single Responsibility Principle).
        - E.g., `sin( x )`, `avg( marks )`, `Car {...}`, `Driver {...}`
        - coupling is excellent
    2. **Sequential** : module elements interact as producer/consumer, i.e., output data from one activity is input data to next.

```
print( process( getword( word ) ) ); // read -> process -> print (shell pipe)
```

- similar to functional, except possibly mandates sequences of use
- coupling is good

3. **Communicational** : module elements contribute to activities that use the same data.

```
find( book, title );
find( book, price );
find( book, ISBN );
find( book, author );
```

- all have same input data
- like sequential but order is not important
- coupling is acceptable
- usually improve maintainability by splitting common module into separate, functional ones

4. **Procedural** : module elements involved in different and possibly unrelated activities, but which flow from one activity to the next.

```
file = open( filename );      // open connection to file name
read( file );                 // read file contents
close( file );                // close connection to file name
```

- related by order of execution rather than by any single problem-related function
- typically data sent to procedure modules is unrelated to data sent back
- procedural modules pass around partial results

5. **Temporal** : module elements involved in activities related in time.

```
initialization
  - turn things on
  - turn things off
  - set things to 0
  - set things to 1
  - set things to ' '
```

- unrelated except carried out at particular time
- each initialization is more closely related to the modules that make use of it ⇒ tight coupling
- want to re-initialize only some of the entities in initialization routine
- like procedural, except order of execution is more important in procedural

6. **Logical** : module elements contribute to same general category, where activity is selected from outside the module.

```
#include <algorithms>
find ...
swap ...
search ...
sort ...
inner_product ...
```

- modules contain number of activities of some general kind

- to use, pick out just one of the pieces needed
- interface weak, and contains code sharing common lines of code and/or data areas

7. **Coincidental** : module elements grouped arbitrarily.
   - activities are related neither by flow of data nor control
   - like logical, internal activity must be externally selected, but worse since categories in the module are very weakly related

2.29 Inheritance

- Object-***oriented*** languages provide **inheritance** for writing 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.
- Implementation inheritance provides reuse of code ***inside*** an object type; type inheritance provides reuse ***outside*** the object type by allowing existing code to access the base type.

2.29.1 Implementation Inheritance

- 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 generate and debug a new object type.
- One way to understand implementation inheritance is to model it via composition:

<table>
<tr><th>Composition</th><th>Inheritance</th></tr>
<tr><td>

```
struct Engine { // Base
    int cyls;
    int r(...) { ... }
    Engine() { ... }
};
struct Car { // Derived
    Engine e;  // explicit composition
    int s(...) { e.cyls = 4; e.r(...); ... }
    Car() { ... }
} vw;
vw.e.cyls = 6; // composition reference
vw.e.r(...); // composition reference
vw.s(...); // direct reference
```

</td><td>

```
struct Engine { // Base
    int cyls;
    int r(...) { ... }
    Engine() { ... }
};
struct Car : public Engine { // implicit
                             // composition
    int s(...) { e.cyls = 4; e.r(...); ... }
    Car() { ... }
    Derived() { ... }
} vw;
vw.cyls = 3; // direct reference
vw.r(...); // direct reference
vw.s(...); // direct reference
```

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

- Composition explicitly creates object member, e, to aid in implementation.

- ○ A Car "has-a" Engine.
- ○ A Car is not an Engine nor is an Engine a Car, i.e., they are not logically interchangable.

- Inheritance, "**public** Engine" clause, implicitly:
  - ○ creates an anonymous base-class object-member,
  - ○ ***opens*** the scope of anonymous member so its members are accessible without qualification, both inside and outside the inheriting object type.

- E.g., Car declaration creates invisible Engine object in Car object, like composition, and allows direct access to variables Engine::i and Engine::r in Car::s.

- Constructors and destructors must be invoked for all implicitly declared objects in inheritance hierarchy as done for an explicit member in composition.

Car d; . . .	implicitly rewritten as	Engine b; b.Engine(); *// implicit, hidden declaration* Car d; d.Car(); . . . d.~Car(); b.~Engine(); *// reverse order of construction*

- If base type has members with the same name as derived type, it works like nested blocks: inner-scope name overrides outer-scope name (see Section 2.4.2, p. 49).

- Still possible to access outer-scope names using "::" qualification (see Section 2.12, p. 84) to specify the particular nesting level.

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

```
class Base1 {
    int i;
}
class Base2 extends Base1 {
    int i;
}
class Derived extends Base2 {
    int i;
    void s() {
        int i = 3;
        this.i = 3;
        ((Base2)this).i = 3; // super.i
        ((Base1)this).i = 3;
    }
}
```

</td><td>

```
struct Base1 {
    int i;
};
struct Base2 : public Base1 {
    int i;            // overrides Base1::i
};
struct Derived : public Base2 {
    int i;            // overrides Base2::i
    void r() {
        int i = 3;    // overrides Derived::i
        Derived::i = 3; // this.i
        Base2::i = 3;
        Base2::Base1::i = 3; // or Base1::i
    }
};
```

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

- ***Friendship is not inherited.***

```
class C {
    friend class Base;
    . . .
};
class Base {
    // access C's private members
    . . .
};
class Derived : public Base {
    // not friend of C
};
```

- 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.

2.29.2 Type Inheritance

- Type inheritance establishes an "**is-a**" relationship among types.

```
class Employee {
    . . . // personal info
};
class FullTime : public Employee {
    . . . // wage & benefits
};
class PartTime : public Employee {
    . . . // wage
};
```

  - A FullTime "is-a" Employee; a PartTime "is-a" Employee.
  - A FullTime and PartTime are logically interchangable with an Employee.
  - A FullTime and PartTime are not logically interchangable.

- Type inheritance extends name equivalence (see Section 2.10, p. 76) 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 );   // allowed
r( b );   // disallowed, name equivalence
```

- Since types Foo and Bar are structurally equivalent, instances of either type should work as arguments to routine r (see Section 2.11, p. 83).

- 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).
- ***Type inheritance relaxes name equivalence by aliasing the derived name with its base-type names.***

```
struct Foo {                  struct Bar : public Foo { // inheritance
    int i;                        // remove Foo members
    double d;
                                  . . .
} f;                          } b;
void r( Foo f ) { . . . }
r( f );   // valid call, derived name matches
r( b );   // 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.
- Why is the qualification Complex:: necessary in Mycomplex::abs?
- 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.
- Redeclare Complex variables to Mycomplex to get new abs, and member calls returns the current number of calls to abs for any Mycomplex object.
- 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;   // disallowed
}
```

- However, result type from **operator+** is Complex, not Mycomplex.
- Assignment of a complex (base type) to Mycomplex (derived type) disallowed 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! (see Section 2.29.7, p. 149)

- **public** inheritance means both implementation and type inheritance.
- **private** inheritance means only implementation inheritance.

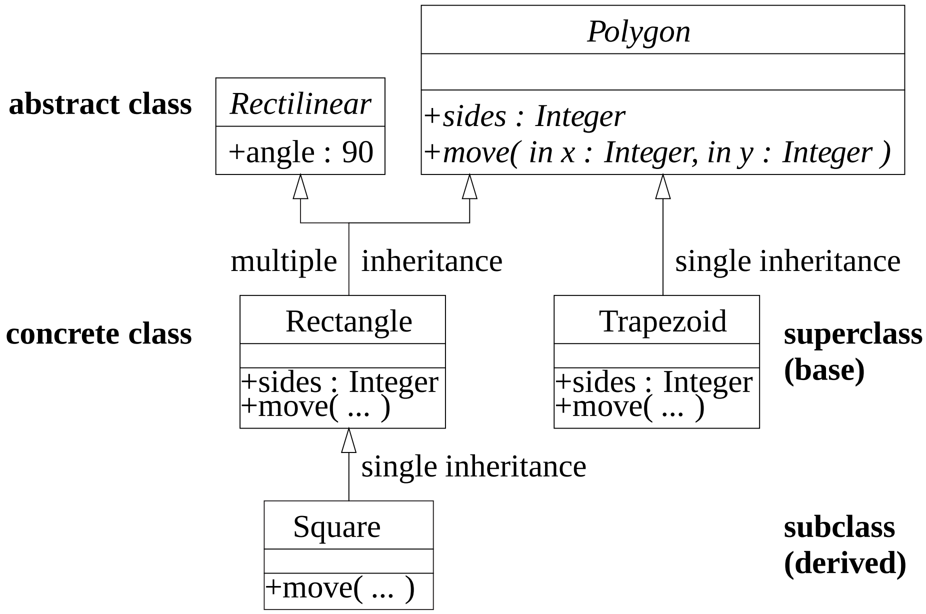
```
class bus : private car { . . .
```

  Use implementation from car, but bus is not a car.

- No direct mechanism in C++ for type inheritance without implementation inheritance.

2.29.3 UML inheritance

- **Generalization** : reuse through forms of inheritance.



  - Represent inheritance by arrowhead △ to establish is-a relationship on type, and reuse of attributes and operations.
  - Association class can be implemented with forms of multiple inheritance (mixin).

- For abstract class, the class name and abstract operations are *italicized*.
- For concrete class, abstract operations that are implemented appear in the class diagram.

2.29.4 Constructor/Destructor

- Constructors are 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 executed bottom-up, from most derived to base type.
- 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 (see Section 2.21.6, p. 116).
- To pass arguments to other constructors, use 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) {. . .} };

2.29.5 Copy Constructor / Assignment

- Each aggregate type has a default/copy constructor, assignment operator, and destructor (see page 113), so these members cannot be inherited as they exist in the derived type.
- Otherwise, copy-constructor/assignment work like composition (see Section 2.21.5, p. 112)

```
struct B {
    B() { cout << "B() "; }
    B( const B &c ) { cout << "B(&) "; }
    B &operator=( const B &rhs ) { cout << "B= "; }
};
struct D : public B {
    int i;                      // basic type, bitwise
};
int main() {
    D i;      cout << endl;     // B's default constructor
    D d = i; cout << endl;      // D's default copy-constructor
    d = i;    cout << endl;     // D's default assignment
}
```

outputs the following:

```
B()        // D i
B(&)       // D d = i
B=         // d = i
```

- If D defines a copy-constructor/assignment, it overrides that in any subobject.

```
struct D : public B {
    ... // same declarations
    D() { cout << "D() "; }
    D( const D &c ) : i( c.i ), B( c ) { cout << "D(&) "; }
    D &operator=( const D &rhs ) {
        i = rhs.i;  (B &)*this = rhs;
        cout << "D= ";
        return *this;
    }
};
```

outputs the following:

```
B() D()     // D i
B(&) D(&)   // D d = i
B= D=       // d = i
```

Must copy each subobject to get same output. **Note coercion!**

2.29.6 Overloading

- Overloading a member routine in a derived class overrides all overloaded routines in the base class with the same name.

```
class Base {
  public:
    void mem( int i ) {}
    void mem( char c ) {}
};
class Derived : public Base {
  public:
    void mem() {}   // overrides both versions of mem in base class
};
```

- Hidden base-class members can still be accessed:

  - Provide explicit wrapper members for each hidden one.

    ```
    class Derived : public Base {
      public:
        void mem() {}
        void mem( int i ) { Base::mem( i ); }
        void mem( char c ) { Base::mem( c ); }
    };
    ```

  - Collectively provide implicit members for all of them.

    ```
    class Derived : public Base {
      public:
        void mem() {}
        using Base::mem; // all base mem routines visible
    };
    ```

  - Use explicit qualification to call members (violates abstraction).

    ```
    Derived d;
    d.Base::mem( 3 );
    d.Base::mem( 'a' );
    d.mem();
    ```

2.29.7 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 object type, but expectation is that 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, expectation is that the calls should invoke Base::r.

- To invoke a routine defined in a referenced object, qualify member routine with **virtual**.

- To invoke a routine defined by the type of a pointer/reference, do not qualify 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***.

- 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.

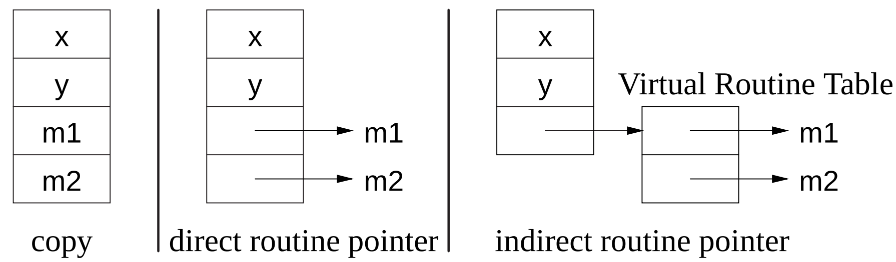
- C++ provides mechanism to override the default at the call site.

Java	C++
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* **public void** e() {} *// 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*	**struct** Base { **void** f() {} *// non-virtual* **void** g() {} *// non-virtual* **virtual void** h() {} *// virtual* }; **struct** Derived : **public** Base { **void** g() {}; *// replace, non-virtual* **void** h() {}; *// replace, virtual* **void** e() {}; *// extension, non-virtual* }; Base &bp = *****new** Derived(); *// polymorphic assignment* 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* *// cannot access "e" through bp*

- 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.
- C++ virtual members are qualified in base type as opposed to derived type.
- Hence, C++ requires the base-type definer to presuppose how derived definers might want the call default to work.
- **Good practice for inheritable types is to make all routine members virtual.**
- Any type with virtual members needs to make the destructor virtual (even if empty) so the most derived destructor is called through a base-type pointer/reference.
- Virtual routines are normally 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:



2.29.8 Downcast

- Type inheritance can mask the actual type of an object through a pointer/reference (see Section 2.29.2, p. 144).
- A **downcast** dynamically determines the actual type of an object pointed to by a polymorphic pointer/reference.
- The Java operator instanceof and the C++ **dynamic_cast** operator perform a dynamic check of the object addressed by a pointer/reference (not coercion):

Java	C++
`Base bp = new Derived();` `if ( bp instanceof Derived )` `    ((Derived)bp).rtn();`	`Base *bp = new Derived;` `Derived *dp;` `dp = dynamic_cast<Derived *>(bp);` `if ( dp != 0 ) { // 0 => not Derived` `    dp->rtn();   // only in Derived`

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

2.29.9 Slicing

- Polymorphic copy or assignment can result in object truncation, called **slicing**.

```
struct B {
    int i;
};
struct D : public B {
    int j;
};
void f( B b ) {...}
int main() {
    B b;
    D d;
    f( d );      // truncate D to B
    b = d;       // truncate D to B
}
```

- ***Avoid polymorphic value copy/assignment; use polymorphic pointers.***

2.29.10 Protected Members

- Inherited object types can access and modify public and protected members allowing access to some of an object's implementation.

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

2.29.11 Abstract Class

- **Abstract class** combines type and implementation inheritance for structuring new types.

- Contains at least one pure virtual member that ***must*** be implemented by derived class.

```
class Shape {
     int colour;
  public:
     virtual void move( int x, int y ) = 0; // pure virtual member
};
```

- Strange initialization to 0 means pure virtual member.

- Define type hierarchy (taxonomy) of abstract classes moving common data and operations are high as possible in the hierarchy.

Java	C++
``abstract class Shape { protected int colour = White; public abstract void move(int x, int y); } abstract class Polygon extends Shape { protected int edges; public abstract int sides(); } class Rectangle extends Polygon { protected int x1, y1, x2, y2; public Rectangle(...) {...} public void move(int x, int y) {...} public int sides() { return 4; } } class Square extends Rectangle { // check square Square(...) { super(...); ...} }```	```class Shape { protected: int colour; public: Shape() { colour = White; } virtual void move(int x, int y) = 0; }; class Polygon : public Shape { protected: int edges; public: virtual int sides() = 0; }; class Rectangle : public Polygon { protected: int x1, y1, x2, y2; public: Rectangle(...) {...} // init corners void move(int x, int y) {...} int sides() { return 4; } }; struct Square : public Rectangle { // check square Square(...) : Rectangle(...) {...} };```

- Use **public**/**protected** to define interface and implementation access for derived classes.
- Provide (pure) virtual member to allow overriding and force implementation by derived class.
- Provide default variable initialization and implementation for **virtual** routine (non-abstract) to simplify derived class.
- Provide non-virtual routine to ***force*** specific implementation; ***derived class should not override these routines***.
- **Concrete class** inherits from one or more abstract classes defining all pure virtual members, i.e., can be instantiated.
- ***Cannot instantiate abstract class, but can declare pointer/reference to it.***
- Pointer/reference used to write polymorphic data structures and routines:

```
void move3D( Shape &s ) { ... s.move(...); ... }
Polygon *polys[10] = { new Rectangle(), new Square(), ... };
for ( unsigned int i = 0; i < 10; i += 1 ) {
    cout << polys[i]->sides() << endl;  // polymorphism
    move3D( *polys[i] );  // polymorphism
}
```

- To maximize polymorphism, ***write code to the highest level of abstraction***[3], i.e. use Shape over Polygon, use Polygon over Rectangle, etc.

2.29.12 Multiple Inheritance

- **Multiple inheritance** allows a new type to apply type and implementation inheritance multiple times.

```
class X : public Y, public Z, private P, private Q { ... }
```

- X type is aliased to types Y and Z with implementation, and also uses implementation from P and Q.
- **Interface class** (**pure abstract-class**) provides only types and constants, providing type inheritance.
- Java only allows multiple inheritance for interface class.

Java	C++
<pre>interface Polygon { int sides(); void move(int x, int y); } interface Rectilinear { final int angle = 90; } class Rectangle implements Rectilinear, Polygon { private int x1, y1, x2, y2; public void move(int x, int y) {} public int sides() { return 4; } } class Square extends Rectangle { public void move(int x, int y) {} }</pre>	<pre>struct Polygon { virtual int sides() = 0; virtual void move(int x, int y) = 0; }; struct Rectilinear { enum { angle = 90 }; }; class Rectangle : public Polygon, public Rectilinear { int x1, y1, x2, y2; public: void move(int x, int y) {} int sides() { return 4; } }; struct Square : public Rectangle { void move(int x, int y) {} };</pre>

- Multiple inheritance has ***many*** problems (beyond this course).
- ***Safe if restrict multiple inheritance to one public type and one or two private types.***

2.30 Composition / Inheritance Design

- Duality between “has-a” (composition) and “is-a” (inheritance) relationship (see page 142).
- Types created from multiple composite classes; types created from multiple superclasses.

[3] Also called “program to an interface not an implementation”, which does not indicate the highest level of abstraction.

Composition	Inheritance
class A {...};	**class** A {...};
class B { A a; ...};	**class** B : A {...};
class C {...};	**class** C {...};
class D { B b; C c; ...};	**class** D : B, C {...};

- Both approaches:
  - remove duplicated code (variable/code sharing)
  - have separation of concern into components/superclasses.
- Choose inheritance when evolving hierarchical types (taxonomy) needing polymorphism.

```
Vehicle
  Construction
    Heavy Machinery
      Crane, Grader, Back-hoe
    Haulage
      Semi-trailer, Flatbed
  Passenger
    Commercial
      Bus, Fire-truck, Limousine, Police-motorcycle
    Personal
      Car, SUV, Motorcycle
```

- For maximum reuse and to eliminate duplicate code, place variables/operations as high in the hierarchy as possible.
- ***Polymorphism requires derived class maintain base class's interface (substitutability).***
  - derived class should also have **behavioural** compatibility with base class.
- However, all taxonomies are an organizational compromise: when is a car a limousine and vice versa.
- Not all objects fit into taxonomy: flying-car, boat-car.
- Inheritance is rigid hierarchy.
- Choose composition when implementation can be **delegated**.

```
class Car {
    SteeringWheel s;     // fixed
    Donut spare;
    Wheel *wheels[4];    // dynamic
    Engine *eng;
    Transmission *trany;
  public:
    Car( Engine *e = fourcyl, Transmission *t = manual ) :
        eng( e ), trany( t ) { wheels[i] = ...}
    rotate() {...}             // rotate tires
    wheels( Wheels *w[4] ) {...} // change wheels
    engine( Engine *e ) {...}  // change engine
};
```

- Composition may be fixed or dynamic (pointer/reference).
- Composition still uses hierarchical types to generalize components.
  - Engine is abstract class that is specialized to different kinds of engines, e.g., 3,4,6,8 cylinder, gas/diesel/hybrid, etc.

2.31 Observer Pattern

- **observer** pattern : 1 to many dependency ⇒ change updates dependencies

```
struct Fan {                          // abstract
    Band &band;
    Fan( Band &band ) : band( band ) {}
    virtual void update( CD cd ) = 0;
};
struct Band {
    list<Fan *> fans;                 // list of fans
    static void perform( Fan *fan ) { fan->update(); }
    void attach( Fan &fan ) { fans.push_back( &fan ); }
    void deattach( Fan &fan ) { fans.remove( &fan ); }
    void notify() { for_each( fans.begin(), fans.end(), perform ); }
};
struct Groupie : public Fan {         // specialize
    Groupie( Band &band ) : Fan( band ) { band.attach( *this ); }
    ~Groupie() { band.deattach( *this ); }
    void update( CD cd ) { buy/listen new cd }
};
Band dust;
Groupie g1( dust ), g2( dust );       // register
dust.notify();                        // inform fans about new CD
```

- manage list of interested objects, and push new events to each
- alternative design has interested objects pull the events from the observer
  - ⇒ observer must store events until requested

2.32 Decorator Pattern

- **decorator** pattern : attach additional responsibilities to an object dynamically

```
struct Window {
    virtual void move(...) {...}
    virtual void lower(...) {...}
    ...
};                                   };
struct Scrollbar : public Window {   // specialize
    enum Kind { Hor, Ver };
    Window &window;
    Scrollbar( Window &window, Kind k ) : window( &window ), ... {}
    void scroll( int amt ) {...}
};

struct Title : public Window {       // specialize
    ...
    Title( Window &window, ... ) : window( window ), ... {}
    setTitle( string t ) {...}
};
Window w;
Title( Scrollbar( Scrollbar( w, Ver ), Hor ), "title" ) decorate;
```

- decorator only mimics object's type through base class

- allows decorator to be dynamically associated with different object's, or same object to be associated with multiple decorators

2.33 Factory Pattern

- **factory** pattern : generalize creation of family of products with multiple variants

```
struct Food {...};                    // abstract product
struct Pizza : public Food {...};     // concrete product
struct Burger : public Food {...};    // concrete product
struct Restaurant {                   // abstract factory product
    enum Kind { Pizza, Burger };
    virtual Food *order( Kind f ) = 0;
    virtual int staff() = 0;
};
struct Pizzeria : public Restaurant {  // concrete factory product
    Food *order( Kind f ) {}
    int staff() {...}
};
struct Burgers : public Restaurant {   // concrete factory product
    Food *order( Kind f ) {}
    int staff() {...}
};
```

```
enum Type { PizzaHut, BugerKing };
struct RestaurantFactory {              // abstract factory
    Restaurant *create( Type t ) {}
};
struct PizzeriaFactory : RestaurantFactory { // concrete factory
    Restaurant *create( Type t ) {}
};
struct BurgerFactory : RestaurantFactory { // concrete factory
    Restaurant *create( Type t ) {}
};

PizzeriaFactory pizzeriaFactory;
BurgerFactory burgerFactory;
Restaurant *pizzaHut = pizzeriaFactory.create( PizzaHut );
Restaurant *burgerKing = burgerFactory.create( BugerKing );
Food *dispatch( Restaurant::Kind food ) { // parameterized creator
  switch ( food ) {
    case Restaurant::Pizza: return pizzaHut->order(Restaurant::Pizza);
    case Restaurant::Burger: return burgerKing->order(Restaurant::Burger);
    default: ; // error
  }
}
```

- use factory-method pattern to construct generated product (Food)
- use factory-method pattern to construct generated factory (Restaurant)
- clients obtains a concrete product (Pizza, Burger) from a concrete factory (PizzaHut, BugerKing), but product type is unknown
- client interacts with product object through its abstract interface (Food)

2.34 Template Method

- **template method** : provide algorithm but defer some details to subclass

```
class PriceTag {                        // template method
    virtual string label() = 0;         // details for subclass
    virtual string price() = 0;
    virtual string currency() = 0;
  public:
    string tag() { return label() + price() + currency(); }
};
class FurnitureTag : public PriceTag {  // actual method
    string label() { return "furniture "; }
    string price() { return "$1000 "; }
    string currency() { return "Cdn"; }
};
FurnitureTag ft;
cout << ft.tag() << endl;
```

- template-method routines are non-virtual, i.e., not overridden

2.35 Compiling Complex Programs

- As number of TUs grow, so do the references to type/variables (dependencies) among TUs.
- When one TU is changed, other TUs that depend on it must change and be recompiled.
- ***For a large numbers of TUs, the dependencies turn into a nightmare with respect to re-compilation.***

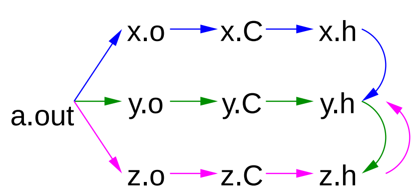
2.35.1 Dependencies

- A **dependence** occurs when a change in one location (entity) requires a change in another.
- Dependencies can be:
  - loosely coupled, e.g., changing source code may require a corresponding change in user documentation, or
  - tightly coupled, changing source code may require recompiling of some or all of the components that compose a program.
- Dependencies in C/C++ occur as follows:
  - executable depends on .o files (linking)
  - .o files depend on .C files (compiling)
  - .C files depend on .h files (including)

source code

```
x.h    #include "y.h"
x.C    #include "x.h"

y.h    #include "z.h"
y.C    #include "y.h"

z.h    #include "y.h"
z.C    #include "z.h"
```

dependence graph

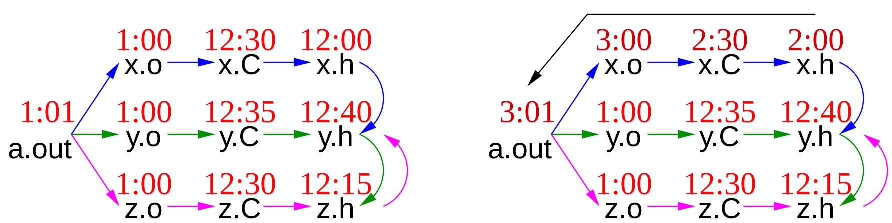


- Cycles in **#include** dependences are broken by **#ifndef** checks (see page 97).
- The executable (a.out) is generated by compilation commands:

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

- However, it is inefficient and defeats the point of separate compilation to recompile all program components after a change.
- If a change is made to y.h, what is minimum recompilation necessary? (all!)

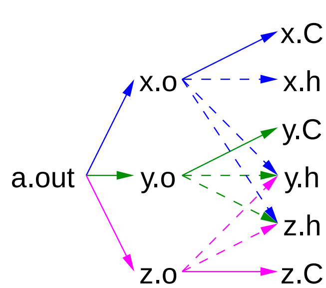
- Does ***any*** change to y.h require these recompilations?
- Often no mechanism to know the kind of change made within a file, e.g., changing a comment, type, variable.
- Hence, “change” may be coarse grain, i.e., based on ***any*** change to a file.
- One way to denote file change is with **time stamps**.
- UNIX stores in the directory the time a file is last changed, with second precision.
- Using time to denote change means the dependency graph is a temporal ordering where the root has the newest (or equal) time and the leafs the oldest (or equal) time.



  - Files x.o, y.o and z.o created at 1:00 from compilation of files created before 1:00.
  - File a.out created at 1:01 from link of x.o, y.o and z.o.
  - Changes are subsequently made to x.h and x.C at 2:00 and 2:30.
  - Only files x.o and a.out need to be recreated at 3:00 and 3:01. (Why?)

2.35.2 Make

- **make** is a system 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 understand relationships among sources, one that exists at the source-code level and is crucial.**
- Hence, make dependence-graph loses some of the relationships (dashed lines):



- 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.
- 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.
- 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 name is specified.
- Each target has a list of dependencies, and possibly a set of commands specifying how to re-establish the target.

```
variable = value                             # variable
target : dependency1 dependency2 ...         # target / dependencies
	command1                                 # rules
	command2
	...
```

- ***Commands must be indented by one tab character.***
- make is invoked with a target, which is the root or subnode of a dependence graph.
- make builds the dependency graph and decorates the edges with time stamps for the specified files.
- If any of the dependency files (leafs) is newer than the target file, or if the target file does not exist, the commands are executed by the shell to update the target (generating 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
```

- Check dependency relationship (assume source files just created):

```
$ make -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
```

All necessary commands are triggered to bring target a.out up to date.

- ◦ -n builds and checks the dependencies, showing rules to be triggered (leave off to execute rules)
- ◦ -f Makefile is the dependency file (leave off if named [Mm]akefile)
- ◦ a.out target name to be updated (leave off if first target)

- Generalize and eliminate duplication using variables:

```
CXX = g++                      # compiler
CXXFLAGS = -g -Wall -c         # compiler flags
OBJECTS = x.o y.o z.o          # object files forming executable
EXEC = a.out                   # executable name

${EXEC} : ${OBJECTS}           # link step
    ${CXX} ${OBJECTS} -o ${EXEC}
x.o : x.C x.h y.h z.h          # targets / dependencies / commands
    ${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:
  - ◦ make can deduce simple rules when dependency files have specific suffixes.
  - ◦ E.g., given target with dependencies:

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

    make deduces the following rule:

    ```
    ${CXX} ${CXXFLAGS} -c -o x.o     # special variable names
    ```

    where -o x.o is redundant as it is implied by -c.
  - ◦ This rule use variables ${CXX} and ${CXXFLAGS} for generalization.
  - ◦ Therefore, all rules for x.o, y.o and z.o can be removed.

    ```
    CXX = g++                      # compiler
    CXXFLAGS = -g -Wall            # compiler flags, remove -c
    OBJECTS = x.o y.o z.o          # object files forming executable
    EXEC = a.out                   # executable name

    ${EXEC} : ${OBJECTS}           # link step
        ${CXX} ${OBJECTS} -o ${EXEC}
    x.o : x.C x.h y.h z.h          # targets / dependencies
    y.o : y.C y.h z.h
    z.o : z.C z.h y.h
    ```

- Because dependencies are extremely complex in large programs, programmers seldom construct them correctly or maintain them.

- **Without complete and update dependencies, make is useless.**
- Automate targets and dependencies:

```
CXX = g++                        # compiler
CXXFLAGS = -g -Wall -MMD           # compiler flags
OBJECTS = x.o y.o z.o            # object files forming executable
DEPENDS = ${OBJECTS:.o=.d}         # substitute ".o" with ".d"
EXEC = a.out                     # executable name

${EXEC} : ${OBJECTS}             # link step
    ${CXX} ${OBJECTS} -o ${EXEC}

-include ${DEPENDS}              # copies files x.d, y.d, z.d (if exists)

.PHONY : clean                   # not a file name
clean :                          # remove files that can be regenerated
    rm -rf ${DEPENDS} ${OBJECTS} ${EXEC}   # alternative *.d *.o
```

  - Preprocessor traverses all include files, so it knows all source-file dependencies.
  - g++ flag -MMD writes out a dependency graph for user source-files to file *source-file*.d

file	contents
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/system source-files.
  - **-include** reads the .d files containing dependencies.
  - **.PHONY** indicates a target that is not a file name and never created; it is a recipe to be executed every time the target is specified.
    - A phony target avoids a conflict with a file of the same name.
  - Phony target clean removes product files that can be rebuilt (save space).

```
$ make clean          # remove all products (don't create "clean")
```

- Hence, it is possible to have a universal Makefile for a single or multiple programs.

2.36 Template

- Inheritance provides reuse for types organized into a hierarchy that extends name equivalence.
- **Template** provides alternate kind of reuse with no type hierarchy and types are not equivalent.
- E.g., overloading (see Section 2.15, p. 89), where there is identical code but different types:

```
int max( int a, int b ) { return a > b ? a : b; }
double max( double a, double b ) { return a > b ? a : b; }
```

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

```
template<typename T>  T max( T a, T b ) { return a > b ? a : b; }
```

- **template** introduces type parameter T used to declare return and parameter types.

- Template routine is called with value for T, and compiler constructs a routine with this type.

```
cout << max<int>( 1, 3 );     // T -> int
cout << max<double>( 1.1, 3.5 ); // T -> double
```

- In many cases, the compiler can infer type T from argument(s):

```
cout << max( 1, 3 );           // T -> int
cout << max( 1.1, 3.5 );       // T -> double
```

- Inferred type must supply all operations used within the template routine.
  - e.g., types used with template routine max must supply **operator**>.

- **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, unsigned int N=10> // default type/value
  struct Stack {                       // NO ERROR CHECKING
    T elems[N];                        // maximum N elements
    unsigned int size;                 // position of free element after top
    Stack() { size = 0; }
    T top() { return elems[size - 1]; }
    void push( T e ) { elems[size] = e; size += 1; }
    T pop() { size -= 1; return elems[size]; }
};
template<typename T, unsigned int N>  // print stack
  ostream &operator<<( ostream &os, const Stack<T, N> &stk ) {
    for ( int i = 0; i < stk.size; i += 1 ) os << stk.elems[i] << " ";
    return os;
}
```

- Type parameter, T, specifies the element type of array elems, and return and parameter types of the member routines.

- Integer parameter, N, denotes the maximum stack size.

- Unlike template routines, type cannot be inferred by compiler because type is created at declaration before any member calls.

```
Stack<> si;                   // stack of int, 10
si.push( 3 );                 // si : 3
si.push( 4 );                 // si : 3 4
cout << si.top() << endl;     // 4
int i = si.pop();             // i : 4, si : 3
Stack<double> sd;             // stack of double, 10
sd.push( 5.1 );               // sd : 5.1
sd.push( 6.2 );               // sd : 5.1 6.2
cout << sd << endl;           // 5.1 6.2
double d = sd.pop();          // d : 6.2, sd : 5.1
Stack<Stack<int>,20> ssi;     // stack of (stack of int, 10), 20
ssi.push( si );               // ssi : (3 4)
ssi.push( si );               // ssi : (3 4) (3 4)
ssi.push( si );               // ssi : (3 4) (3 4) (3 4)
cout << ssi << endl;          // 3 4  3 4  3 4
si = ssi.pop();               // si : 3 4, ssi : (3 4) (3 4)
```

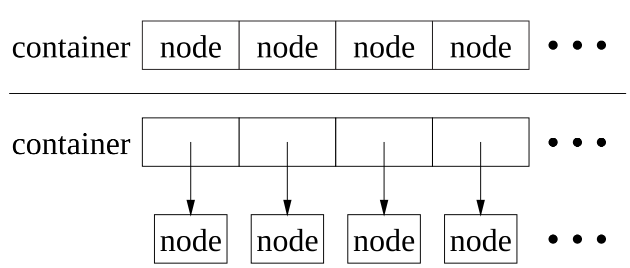
Why does `cout << ssi << endl` have 2 spaces between the stacks?

- Specified type must supply all operations used within the template type.
- ***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 and separate compilation.***
- ***C++03 requires space between the two ending chevrons or >> is parsed as* operator>>.**

```
template<typename T> struct Foo { ... };
Foo<Stack<int>> foo;  // syntax error (fixed C++11)
Foo<Stack<int> > foo; // space between chevrons
```

2.36.1 Standard Library

- C++ Standard Library is a collection of (template) classes and routines providing: I/O, strings, data structures, and algorithms (sorting/searching).
- Data structures are called **containers**: vector, map, list (stack, queue, deque).
- In general, nodes of a data structure are either in a container or pointed-to from the container.



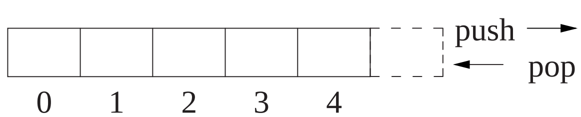
- To copy a node into a container requires its type have a default and/or copy constructor so instances can be created without constructor arguments.

- ***Standard library containers use copying ⇒ node type must have default constructor.***
- **All containers are dynamic sized so nodes are allocated in heap** (arrays can be on stack).
- To provide encapsulation (see Section 2.26, p. 130), containers use a nested **iterator** type (see Section 2.12, p. 84) to traverse nodes.
  - Knowledge about container implementation is completely hidden.
- Iterator capabilities often depend on kind of container:
  - singly-linked list has unidirectional traversal
  - doubly-linked list has bidirectional traversal
  - hashing list has random traversal
- Iterator operator "++" moves forward to the next node, until ***past*** the end of the container.
- For bidirectional iterators, operator "--" moves in the reverse direction to "++".

2.36.1.1 Vector

- vector has random access, length, subscript checking (at), and assignment (like Java array).

std::vector<T>	
vector()	create empty vector
vector(size, *[initialization]*)	create vector with N empty/initialized elements
int size()	vector size
bool empty()	size() == 0
T &**operator**[](**int** i)	access ith element, NO subscript checking
T &at(**int** i)	access ith element, subscript checking
vector &**operator**=(**const** vector &)	vector assignment
void push_back(**const** T &x)	add x after last element
void pop_back()	remove last element
void resize(**int** n)	add or erase elements at end so size() == n
void clear()	erase all elements



- Vector declaration ***may*** specify an initial size, e.g., vector<**int**> v(size), like a dimension.
- When size is known, more efficient to dimension and initialize to reduce dynamic allocation.

```
int size;
cin >> size;                 // read dimension
vector<int> v1( size );      // think int v1[size]
vector<int> v2( size, 0 );   // think int v2[size] = { 0 }
int a[] = { 16, 2, 77, 29 };
vector<int> v3( a, &a[4] );  // think int v3[4]; v3 = a;
vector<int> v4( v3 );        // think int v4[size]; v4 = v3
```

- vector is alternative to C/C++ arrays (see Section 2.10.3.1, p. 79).

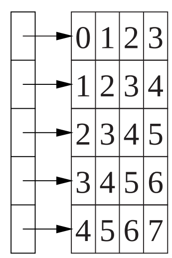
```
#include <vector>
int i, elem;
vector<int> v;                // think: int v[0]
for ( ;; ) {                  // create/assign vector
    cin >> elem;
  if ( cin.fail() ) break;
    v.push_back( elem );      // add elem to vector
}
vector<int> c;                // think: int c[0]
c = v;                        // array assignment
for ( i = c.size() - 1; 0 <= i; i -= 1 ) {
    cout << c.at(i) << " ";   // subscript checking
}
cout << endl;
v.clear();                    // remove ALL elements
```

- Matrix declaration is a vector of vectors (see also page 94):

```
vector< vector<int> > m;
```

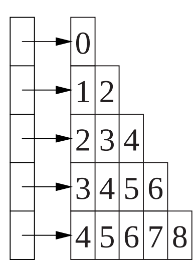
- Again, it is more efficient to dimension, when size is known.

```
#include <vector>
vector< vector<int> > m( 5, vector<int>(4) );
for ( int r = 0; r < m.size(); r += 1 ) {
    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;
}
```



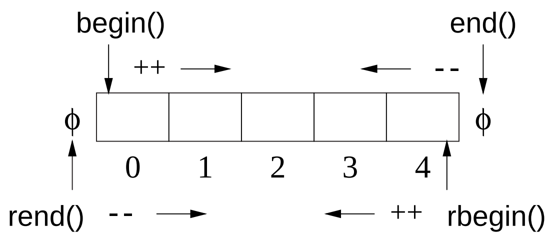
- Optional second argument is initialization value for each element, i.e., 5 rows of vectors each initialized to a vector of 4 integers initialized to zero.
- All loop bounds use dynamic size of row or column (columns may be different length).
- Alternatively, each row is dynamically dimensioned to a specific size, e.g., triangular matrix.

```
vector< vector<int> > m( 5 ); // 5 empty rows
for ( int r = 0; r < m.size(); r += 1 ) {
    m[r].resize( r + 1 ); // different length
    for ( int c = 0; c < m[r].size(); c += 1 ) {
        m[r][c] = r+c;     // or m.at(r).at(c)
    }
}
```



- Iterator allows traversal in insertion order or random order.

std::vector<T>::iterator	
iterator begin()	iterator pointing to first element
iterator end()	iterator pointing **AFTER** last element
iterator rbegin()	iterator pointing to last element
iterator rend()	iterator pointing **BEFORE** first element
iterator insert(iterator posn, **const** T &x)	insert x before posn
iterator erase(iterator posn)	erase element at posn
++, - -, +, +=, -, -= (insertion / random order)	forward/backward operations



- Iterator's value is a pointer to its current vector element ⇒ dereference to access element.

```
vector<int> v(3);
vector<int>::iterator it;
v[0] = 2;              // initialize first element
it = v.begin();        // intialize iterator to first element
cout << v[0] << " " << *v.begin() << " " << *it << endl;
```

- ***If* erase *and* insert *took subscript argument, no iterator necessary!***

- Use iterator like subscript by adding/subtracting from begin/end.

```
v.erase( v.begin() );      // erase v[0], first
v.erase( v.end() - 1 );    // erase v[N - 1], last (why "- 1"?)
v.erase( v.begin + 3 );    // erase v[3]
```

- ***Insert or erase during iteration using an iterator causes failure.***

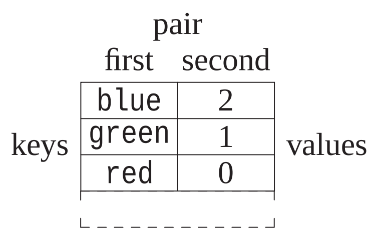
```
vector<int> v;
for ( int i = 0 ; i < 5; i += 1 ) // create
    v.push_back( 2 * i );          // values: 0, 2, 4, 6, 8

v.erase( v.begin() + 3 );          // remove v[3] : 6

int i;    // find position of value 4 using subscript
for ( i = 0; i < 5 && v[i] != 4; i += 1 );
v.insert( v.begin() + i, 33 );     // insert 33 before value 4

// print reverse order using iterator (versus subscript)
vector<int>::reverse_iterator r;
for ( r = v.rbegin(); r != v.rend(); r ++ ) // ++ move towards rend
    cout << *r << endl;           // values: 8, 4, 33, 2, 0
```

2.36.1.2 Map

- map (dictionary) has random access, sorted, unique-key container of pairs (Key, Val).

- set (dictionary) is like map but the value is also the key (array maintained in sorted order).

std::map<Key,Val> / std::pair<**const** Key,Val>	
map(*[initialization]*)	create empty/initialized map
int size()	map size
bool empty()	size() == 0
Val &**operator**[](**const** Key &k)	access pair with Key k
int count(Key key)	0 ⇒ no key, 1 ⇒ key (unique keys)
map &**operator**=(**const** map &)	map assignment
insert(pair<**const** Key,Val>(k, v))	insert pair
erase(Key k)	erase key k
void clear()	erase all pairs




```
#include <map>
map<string, int> m;             // Key => string, Val => int
m["green"] = 1;                 // create, set to 1
m["blue"] = 2;                  // create, set to 2
m["red"];                       // create, set to 0 for int
m["green"] = 5;                 // overwrite 1 with 5
cout << m["green"] << endl; // print 5
m.insert( pair<string,int>( "yellow", 3 ) ); // m[“yellow”] = 3
map<string, int> c( m );        // Key => string, Val => int
c = m;                          // map assignment
if ( c.count( "black" ) )       // check for key “black”
c.erase( "blue" );              // erase pair( “blue”, 2 )
```

- First key subscript creates entry; initialized to default or specified value.

- Iterator can search and return values in key order.

std::map<T>::iterator / std::map<T>::reverse_iterator	
iterator begin()	iterator pointing to first pair
iterator end()	iterator pointing **AFTER** last pair
iterator rbegin()	iterator pointing to last pair
iterator rend()	iterator pointing **BEFORE** first pair
iterator find(Key &k)	find position of key k
iterator insert(iterator posn, **const** T &x)	insert x before posn
iterator erase(iterator posn)	erase pair at posn
++, - - (sorted order)	forward/backward operations

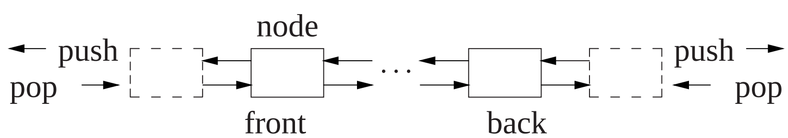
- Iterator returns a pointer to a pair, with fields first (key) and second (value).

```
#include <map>
map<string,int>::iterator f = m.find( "green" ); // find key position
if ( f != m.end() )                 // found ?
    cout << "found " << f->first << ' ' << f->second << endl;

for ( f = m.begin(); f != m.end(); f ++ )          // increasing order
    cout << f->first << ' ' << f->second << endl;

map<string,int>::reverse_iterator r;
for ( r = m.rbegin(); r != m.rend(); r ++ )        // decreasing order
    cout << r->first << ' ' << r->second << endl;
m.clear();                          // remove ALL pairs
```

2.36.1.3 List

- In certain cases, it is more efficient to use a single (stack/queue/deque) or double (list) linked-list container than random-access container.
- Examine list (arbitrary removal); stack, queue, deque are similar (restricted insertion/removal).

std::list<T>	
list()	create empty list
list(size, *[initialization]*)	create list with N empty/initialized elements
int size()	list size
bool empty()	size() == 0
list &**operator**=(**const** list &)	list assignment
T front()	first node
T back()	last node
void push_front(**const** T &x)	add x before first node
void push_back(**const** T &x)	add x after last node
void pop_front()	remove first node
void pop_back()	remove last node
void clear()	erase all nodes



- Like vector, list declaration ***may*** specify an initial size, like a dimension.

- When size is known, more efficient to dimension and initialize to reduce dynamic allocation.

```
int size;
cin >> size;                    // read dimension
list<int> l1( size );
list<int> l2( size, 0 );
int a[ ] = { 16, 2, 77, 29 };
list<int> l3( a, &a[4] );
list<int> l4( l3 );
```

- Iterator returns a pointer to a node.

std::list<T>::iterator / std::list<T>::reverse_iterator	
iterator begin()	iterator pointing to first node
iterator end()	iterator pointing **AFTER** last node
iterator rbegin()	iterator pointing to last node
iterator rend()	iterator pointing **BEFORE** first node
iterator insert(iterator posn, **const** T &x)	insert x before posn
iterator erase(iterator posn)	erase node at posn
++, -- (insertion order)	forward/backward operations

```
#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> dl;                          // doubly linked list
for ( int i = 0; i < 10; i += 1 ) {     // create list nodes
    dl.push_back( Node( 'a'+i, i, i+0.5 ) ); // push node on end of list
}
list<Node>::iterator f;
for ( f = dl.begin(); f != dl.end(); f ++ ) { // forward order
    cout << "c:" << (*f).c << " i:" << f->i << " d:" << f->d << endl;
}
while ( 0 < dl.size() ) {               // destroy list nodes
    dl.erase( dl.begin() );             // remove first node
} // same as dl.clear()
```

2.36.1.4 for_each

- Template routine for_each provides an alternate mechanism to iterate through a container.

- An action routine is called for each node in the container passing the node to the routine for processing (Lisp apply).

```
#include <iostream>
#include <list>
#include <vector>
#include <algorithm>                    // for_each
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 );
}
```

- 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 using a functor (see page 119).
- 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.

2.37 Namespace

- C++ **namespace** is used to organize programs and libraries composed of multiple types and declarations ***to deal with naming conflicts***.
- 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.

- Analogy in Java is a package, but **namespace** does NOT provide abstraction/encapsulation (use .h/.cc files).
- C++ allows multiple namespaces to be defined in a file, as well as among files (unlike Java packages).
- Types and declarations do not have to be added consecutively.

<table>
<tr><th>Java source files</th><th>C++ source file</th></tr>
<tr><td>package Foo; // file
public class X . . . // export one type
// local types / declarations<hr>package Foo; // file
public enum Y . . . // export one type
// local types / declarations<hr>package Bar; // file
public class Z . . . // export one type
// local types / declarations</td><td>namespace Foo {
    // types / declarations
}
namespace Foo {
    // more types / declarations
}
namespace Bar {
    // types / declarations
}</td></tr>
</table>

- Contents of a namespace are 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 importing all of the namespace content.

Java	C++
import Foo.T; **import** Foo.*;	**using** Foo::T; *// declaration* **using namespace** Foo; *// directive*

- **using** declaration ***unconditionally*** introduces an alias (like **typedef**, see Section 2.11, p. 83) into the current scope for specified entity in namespace.
  - May appear in any scope.
  - If name already exists in current scope, **using** fails.

```
namespace Foo { int i = 0; }
int i = 1;
using Foo::i; // i exists in scope, conflict failure
```

- **using** directive ***conditionally*** introduces aliases to current scope for all entities in namespace.

- If name already exists in current scope, alias is ignored; if name already exists from **using** directive in current scope, **using** fails.

```
namespace Foo { int i = 0; }
namespace Bar { int i = 1; }
{
    int i = 2;
    using namespace Foo; // i exists in scope, alias ignored
}
{
    using namespace Foo;
    using namespace Bar; // i exists from using directive
    i = 0;    // conflict failure, ambiguous reference to 'i'
}
```

- May appear in namespace and block scope, but not class scope.

```
namespace Foo {             // start namespace
    enum Colour { R, G, B };
    int i = 3;
}
namespace Foo {             // add more
    class C { int i; };
    int j = 4;
    namespace Bar {         // start nested namespace
        typedef short int shrint;
        char j = 'a';
        int C();
    }
}
int j = 0;                  // external
int main() {
    int j = 3;              // local
    using namespace Foo; // conditional import: Colour, i, C, Bar (not j)
    Colour c;               // Foo::Colour
    cout << i << endl;      // Foo::i
    C x;                    // Foo::C
    cout << ::j << endl;    // external
    cout << j << endl;      // local
    cout << Foo::j << " " << Bar::j << endl;  // qualification
    using namespace Bar;  // conditional import: shrint, C() (not j)
    shrint s = 4;           // Bar::shrint
    using Foo::j;           // disallowed : unconditional import
    C();                    // disallowed : ambiguous "class C" or "int C()"
}
```

- Never put a **using** declaration/directive in a header file (.h) (pollute local namespace) or before **#include** (can affect names in header file).

2.38 Decorator Pattern

- **visitor** pattern : perform operation on elements of heterogeneous container

```
struct PrintVisitor {
    void visit( Wheel &w ) { print wheel }
    void visit( Engine &e ) { print engine }
    void visit( Transmission &t ) { print transmission }
    . . .
};
struct Part {
    virtual void action( Visitor &v ) = 0;
};
struct Wheel : public Part {
    void action( Visitor &v ) { v.visit( *this ); } // overload
};
struct Engine : public Part {
    void action( Visitor &v ) { v.visit( *this ); } // overload
};
. . .

PrintVisitor pv;
list<Part *> ps;
for ( int i = 0; i < 10; i += 1 ) {
    ps.push_back( add different car parts );
}
for ( list<Part *>::iterator pi = ps.begin(); pi != ps.end(); ++pi ) {
    (*pi)->action( pv );
}
```

- each part has a general action that is specialized by visitor
- different visitors perform different actions or dynamically vary the action
- compiler statically selects appropriate overloaded version of visit in action

Index

!, 8, 61
!=, 53, 61
", 6, 86
#, 2
#define, 95
#elif, 97
#else, 97
#endif, 97
#if, 97
#ifdef, 97
#ifndef, 97
#include, 95
$, 2, 22
${}, 22
%, 2
&, 61, 62
&&, 61
&=, 61
′, 6, 51
*, 61, 62, 81
*=, 61
+, 53, 61
++, 167
+=, 61
,, 61
-, 61
--, 167
-=, 61
->, 61
-MD, 164
-MMD, 164
-W, 48
-c, 135
-g, 48, 101
-o, 48, 135
-std, 48
-v, 95
., 61
., 81
.C, 48
.c, 48
.cc, 48, 136
.cpp, 48
.h, 95, 136
.snapshot, 12
/, 3, 61
\, 6, 51
/=, 61
:, 72
::, 61, 84, 85, 130, 143
;;, 29
<, 17, 53, 61
<<, 56, 61, 62, 111
<<=, 61
<=, 53, 61
<ctrl>-c, 7
<ctrl>-d, 18, 58
=, 8, 22, 53, 61
==, 53, 61, 116
>, 6, 17, 53, 61
>&, 17, 18
>=, 53, 61
>>, 56, 61, 62, 111
>>=, 61
?:, 61
[], 26, 53, 93
%, 61
%=, 61
&, 62
^, 61
^=, 61
`, 6
|, 17, 61
|=, 61

~, 3, 61

a.out, 74
absolute pathname, 4
abstract class, 153
 pure, 155
abstract data-type, 130
abstraction, 106, 130
acceptance testing, 46
access control, 130
add, 37
ADT, 130
aggregates, 79
aggregation, 124
alias, 84, 145, 174
alias, 8, 12
allocation
 array, 80, 92
 dynamic, 91
  array, 93
 heap, 92, 93, 167
  array, 93
 matrix, 93
 stack, 66, 93
argc, 74
argument, 87
argv, 74
array, 59, 74, 76, 79, 80, 85, 89, 92, 93, 109
 2-D, 93
 deallocation, 93
 dimension, 79, 80, 86, 89, 92, 167, 172
 parameter, 89
assertion, 97
assignment, 62, 84, 112, 148
 array, 80, 167
 initializing, 50
 operator, 131, 139
association, 122
 unidirectional, 122
association class, 124
atoi, 74
attribute, 121

backquote, 6
backslash, 4, 6, 51
backspace key, 2
backtrace, 102
backward branch, 72, 73
bang, 8
bash, 1, 25, 26
bash, 10
basic types, 49, 76
 bool, 49
 char, 49
 double, 49
 float, 49
 int, 49
 wchar_t, 49
behavioural, 156
bit field, 82
bitwise copy, 113
black-box testing, 44
block, 65, 85
blueprint, 120
bool, 49, 51
boolalpha, 57
boundary value testing, 45
break, 67, 73
 labelled, 71
break, 103
breakpoint, 103
 continue, 104
 next, 104
 step, 103

C-c, 7
C-d, 18, 58
c_str, 53
cascade, 56
case, 28, 67
 ;;, 29
 pattern, 29
case-sensitive, 21
cast, 60–63, 77, 85, 90, 152
cat, 11
cd, 7
cerr, 55
char, 49–51
chevron, 56, 61, 111, 166

chgrp, 17
chmod, 17
chsh, 10
cin, 55
class, 106, 131
class model, 121
classes diagram, 121
clear, 59
cmp, 13
code inspection, 44
coercion, 60, 63, 77, 91, 92
  cast, 60
  explicit, 60, 63
  reinterpret_cast, 60
cohesion, 140
coincidental, 142
comma expression, 93
options, 2
command-line arguments, 73
  argc, 74
  argv, 74
  main, 73
command-line interface, 1
comment, 2
  #, 2
  nesting, 49
  out, 49, 97
commit, 37
Common, 140
communicational, 141
compilation
  g++, 48
compiler
  options
    -D, 96
    -E, 94
    -I, 95
    -MD, 164
    -MMD, 164
    -W, 48
    -c, 135
    -g, 48, 101
    -o, 48, 135
    -std, 48
    -v, 95
  separate compilation, 97, 128
composition, 125, 142, 155
  explicit, 142
concrete class, 154
conditional inclusion, 96
config, 36
const, 52, 78, 88, 96, 106
constant, 51, 52, 106, 136
  initialization, 96
  parameter, 88
  variable, 52
construction, 143
constructor, 76, 108, 143, 147
  const member, 116
  copy, 112, 131, 148
  implicit conversion, 110
  literal, 110
  passing arguments to other constructors, 147
  type, 76
container, 166
  deque, 166
  list, 166
  map, 166
  queue, 166
  stack, 166
  vector, 166, 167
Content, 140
contiguous object, 112
continue
  labelled, 71
continue, 104
contra-variance, 146
Control, 140
control structure, 65
  block, 65
  looping, 65
    break, 30
    continue, 30
    for, 30
    while, 29
  selection, 65, 66
    break, 67
    case, 28, 67
    dangling else, 66

- **default**, 67
- **if**, 27
- pattern, 29
- **switch**, 67, 74
- test, 26
- transfer, 65
- conversion, 53, 62, 77, 110
  - cast, 61, 62
  - **dynamic_cast**, 152
  - explicit, 62, 90
  - implicit, 62, 87, 90, 110
  - narrowing, 63
  - promotion, 62
  - **static_cast**, 62
  - widening, 62
- copy constructor, 113, 131, 139
- copy-modify-merge, 36
- coupling, 139
- cout, 55
- cp, 11
- csh, 1, 19, 26
- csh, 10
- current directory, 3, 4
- current stack frame, 103

- dangling else, 66
- dangling pointer, 115
- Data, 139
- data member, 81
- dbx, 101
- debug print statements, 99
- debugger, 100
- debugging, 43, 99
- dec, 57
- declaration, 49
  - basic types, 49
  - **const**, 96
  - type constructor, 76
  - type qualifier, 50
  - variable, 49
- declaration before use, 128
- Declaration Before Use, 127
- decorator, 158
- deep compare, 116
- deep copy, 114, 116
- default
  - parameter, 91
- **default**, 67
- default constructor, 108
- default initialized, 85
- default value, 88, 108
  - parameter, 88
- delegated, 156
- **delete**, 91
  - [ ], 93
- delete key, 2
- dependence, 160
- deque, 166, 171
- dereference, 22, 62
- desk checking, 44
- desktop, 1
- destruction, 143
  - explicit, 112
  - implicit, 112
  - order, 112
- destructor, 111, 143, 147
- diff, 13
- dimension, 79, 80, 86, 89, 92, 167, 172
- **double**, 49, 51
- double quote, 6, 20
- downcast, 152
- dynamic allocation, 109
- dynamic storage management, 91, 112
- **dynamic_cast**, 152

- **echo**, 9
- egrep, 14
- encapsulation, 130, 167
- end of file, 58
- end of line, 48
- endl, 48, 57
- Enter key, 2
- **enum**, 76, 96
- enumeration, 76
- enumerator, 76
- eof, 58
- equivalence
  - name, 83
  - structural, 83
- equivalence partitioning, 45

error guessing, 45
escape, 6
escape sequence, 51
escaped, 27
evaluation
  short-circuit, 70
execute, 16
exit
  static multi-exit, 69
  static multi-level, 71
exit, 10
exit, 48
exit status, 10, 21
explicit coercion, 60, 63
explicit conversion, 62, 90
export, 133, 137
expression, 61
eye candy, 68

factory, 158
fail, 56, 58
false, 63
feof, 59
file, 3
  .h, 95
  opening, 56
file inclusion, 95
file management
  file permission, 16
  input/output redirection, 17
    <, 17
    >&, 17
    >, 17
    |, 17
file permission
  execute, 16
  group, 16
  other, 16
  read, 16
  search, 16
  user, 16
  write, 16
file suffix
  .C, 48
  .c, 48
  .cc, 48, 136
  .cpp, 48
  .h, 136
  .o, 135
file system, 3
files
  input/output redirection, 17
find, 14, 53
find_first_not_of, 53
find_first_of, 53
find_last_not_of, 53
find_last_of, 53
fixed, 57
flag variable, 69
float, 49, 50
for, 30
for_each, 172
format
  I/O, 57
formatted I/O, 55
Formatted I/O, 55
forward branch, 72
forward declaration, 128
frame, 103
free, 91
free, 91
friend, 132
friendship, 132, 143
fstream, 56
function-call operator, 119
functional, 140
functional testing, 46
functor, 119, 173

g++, 48, 80
garbage collection, 91
gdb
  backtrace, 102
  break, 103
  breakpoint, 103
    continue, 104
    next, 104
    step, 103
  continue, 104
  frame, 103

- info, 103
- list, 105
- next, 104
- print, 102
- run, 102
- step, 103

gdb, 101
generalization, 146
generate, 121
git, 36
config
- config, 36

git, 36
- add, 37
- commit, 37
- mv, 41
- pull, 38
- push, 38
- remote, 36
- rm, 39
- status, 37

init
- init, 36

gitlab, 34
gitlab, 34
global repository, 36
globbing, 4, 6, 14, 15, 28
gmake, 162
goto, 72, 73
- label, 72

graphical user interface, 1
gray-box testing, 44
group, 16

has-a, 143, 155
heap, 87, 92, 93, 167
- array, 93

help, 7
heterogeneous values, 81, 83
hex, 57
hidden file, 3, 5, 11, 12
history, 8
home directory, 3, 4, 7
homogeneous values, 79
hot spot, 98
human testing, 44

I/O
- cerr, 55
- cin, 55
- clear, 59
- cout, 55
- fail, 56
- formatted, 55
- fstream, 56
- ifstream, 56
- ignore, 59
- iomanip, 57
- iostream, 55
- manipulators, 57
  - boolalpha, 57
  - dec, 57
  - endl, 57
  - fixed, 57
  - hex, 57
  - left, 57
  - noboolalpha, 57
  - noshowbase, 57
  - noshowpoint, 57
  - noskipws, 57
  - oct, 57
  - right, 57
  - scientific, 57
  - setfill, 57
  - setprecision, 57
  - setw, 57
  - showbase, 57
  - showpoint, 57
  - skipws, 57
- ofstream, 56

identifier, 72
if, 27
- dangling else, 66

ifstream, 56
ignore, 59
implementation, 136
implementation inheritance, 142
implicit conversion, 62, 87, 90, 110
import, 133, 135
info, 103

inheritance, 142, 155, 164
  implementation, 142
  type, 142, 144
init, 36
initialization, 85, 108, 110, 112, 116, 143, 147, 148
  array, 85
  forward declaration, 129
  string, 86
  structure, 85
inline, 96
input, 48, 55, 57
  >>, 111
  end of file, 58
  eof, 58
  fail, 58
  feof, 59
  formatted, 55
  manipulators
    iomanip, 57
    noskipws, 57
    skipws, 57
  standard input
    cin, 55
input/output redirection, 17
  filter
    |, 17
  input
    <, 17
  output
    >, 17
    >&, 17
int, 49–51
INT16_MAX, 50
INT16_MIN, 50
int16_t, 50
INT_MAX, 50
INT_MIN, 50
integral type, 82
integration testing, 45
interaction model, 121
interface, 106, 136
interface class, 155
iomanip, 57
iostream, 48, 55
is-a, 144, 147, 155
iterator, 167
  ++, 167
  - -, 167
  for_each, 172

keywords, 21
ksh, 1

label, 72
label variable, 72
language
  preprocessor, 47
  programming, 47
  template, 47
left, 57
less, 11
list, 105, 166, 171
  back, 171
  begin, 172
  clear, 171
  empty, 171
  end, 172
  erase, 172
  front, 171
  insert, 172
  pop_back, 171
  pop_front, 171
  push_back, 171
  push_front, 171
  begin, 172
  end, 172
  size, 171
literal, 51, 52, 58, 85
  bool, 51
  char, 51
  double, 51
  escape sequence, 51
  initialization, 85
  int, 51
  string, 51
  type constructor, 85
literals, 76
LLONG_MAX, 50
LLONG_MIN, 50

local repository, 36
logical, 141
login, 2
login shell, 23
logout, 2
long, 50
LONG_MAX, 50
LONG_MIN, 50
loop
    mid-test, 68
    multi-exit, 68
looping statement
    break, 30
    continue, 30
    for, 30
    while, 29
lp, 13
lpstat, 13
ls, 11, 16

machine testing, 44
main, 73, 128
make, 161
make, 162
malloc, 91
man, 10
managed language, 91
manipulators, 57
map, 166, 170
    begin, 171
    end, 171
    erase, 171
    find, 171
    insert, 171
    begin, 171
    end, 171
matrix, 80, 89, 93, 168
member, 81
    anonymous, 143
    const, 116
    constructor, 108
    destruction, 111, 143, 147
    initialization, 108, 143, 147
    object, 106
    operator, 107
    overloading, 107
    pure virtual, 153, 154
    static member, 117
    virtual, 150, 152
memberwise copy, 113
memory leak, 93, 115
mid-test loop, 68
mixin, 147
mkdir, 11
more, 11
multi-exit
    Multi-exit loop, 68
    mid-test, 68
multi-level exit
    static, 71
multiple inheritance, 155
mutually recursive, 127, 128
mv, 11, 41

name equivalence, 83, 144, 145, 164
namespace, 48, 173
    std, 48
narrowing, 63
navigable, 123
nesting, 143
    blocks, 66
    comments, 49
    initialization, 85
    preprocessor, 96
    routines, 86
    type, 84
new, 91
next, 104
noboolalpha, 57
non-contiguous, 111, 112
noshowbase, 57
noshowpoint, 57
noskipws, 57
npos, 53
NULL, 85
null character, 51

object, 105
    anonymous member, 143
    assignment, 112, 148

- **const** member, 116
- constructor, 108, 143, 147
- copy constructor, 112, 131, 148
- default constructor, 108
- destructor, 111, 143, 147
- initialization, 108, 147
- literal, 110
- member, 106
- pure virtual member, 153, 154
- **static** member, 117
- virtual member, 150, 152

object diagram, 123
object model, 121
object-oriented, 105, 142
observer, 157
oct, 57
ofstream, 56
open, 56
- file, 56

operation, 122
operators
- *, 62
- <<, 56, 111
- >>, 56, 111
- &, 62
- arithmetic, 61
- assignment, 61
- bit shift, 61
- bitwise, 61
- cast, 61
- comma expression, 61
- control structures, 61
- logical, 61
- overloading, 56, 107
- pointer, 61, 62
- relational, 61
- selection, 84, 85, 143
- string, 53
- **struct**, 61

selection, 130
other, 16
output, 48, 55, 60
- <<, 111
- endl, 48
- formatted, 55
- manipulators
  - boolalpha, 57
  - dec, 57
  - endl, 57
  - fixed, 57
  - hex, 57
  - iomanip, 57
  - left, 57
  - noboolalpha, 57
  - noshowbase, 57
  - noshowpoint, 57
  - oct, 57
  - right, 57
  - scientific, 57
  - setfill, 57
  - setprecision, 57
  - setw, 57
  - showbase, 57
  - showpoint, 57
- standard error
  - cerr, 55
- standard output
  - cout, 48, 55

overflow, 62
overload, 73
overloading, 56, 89, 107, 108, 111
override, 143, 145, 149, 150
overriding, 66

paginate, 11
parameter, 87
- array, 89
- constant, 88
- default value, 88
- pass by reference, 87
- pass by value, 87
- prototype, 128

parameter passing
- array, 89

pass by reference, 87
pass by value, 87
pattern, 29
pattern matching, 4
performance testing, 46
pointer, 76, 77, 85

0, 85
array, 80, 92
matrix, 93
NULL, 85
polymorphic, 152
polymorphism, 144, 156
preprocessor, 47, 49, 94, 164
#define, 95
#elif, 97
#else, 97
#endif, 97
#if, 97
#ifdef, 97
#ifndef, 97
#include, 95
comment-out, 49
file inclusion, 95
variable, 96
print, 102
private, 131
procedural, 141
project, 36
promotion, 62
prompt, 1, 2
$, 2
%, 2
>, 6
protected, 131
prototype, 127, 128
pseudo random-number generator, 118
pseudo random-numbers, 118
public, 81, 131
pull, 36
pull, 38
pure abstract-class, 155
pure virtual member, 153, 154
push, 36
push, 38
pwd, 8

queue, 166, 171
quoting, 6

random number, 118
generator, 118
pseudo-random, 118
seed, 120
random-number generator, 118
read, 16
real time, 10
recursive type, 82
reference, 62, 76, 77
initialization, 78
reference parameter, 87
regression testing, 46
regular expressions, 4
reinterpret_cast, 60
relative pathname, 4
remote, 36
replace, 53
return code, 10
Return key, 2
reuse, 142
rfind, 53
right, 57
rm, 12, 39
routine, 86
argument/parameter passing, 87
array parameter, 89
member, 106
parameter, 86
pass by reference, 87
pass by value, 87
prototype, 127
routine overloading, 90
routine prototype
forward declaration, 128
scope, 106
routine member, 81
routine prototype, 128
run, 102

scientific, 57
scope, 106, 130, 173
script, 19, 25
search, 16
security testing, 46
sed, 19
selection operator, 84
selection statement, 66

- **break**, 67
- **case**, 28, 67
- **default**, 67
- **if**, 27
- pattern, 29
- **switch**, 67, 74

self-assignment, 115
semi-colon, 28
sentinel, 51
separate compilation, 97, 133
- -c, 135

sequential, 140
set, 170
setfill, 57
setprecision, 57
setw, 57
sh, 1, 19
sh, 10
sha-bang, 19
shell, 1
- bash, 1, 26
- csh, 1, 26
- ksh, 1
- login, 23
- prompt, 2
  - $, 2
  - %, 2
  - >, 6
- sh, 1
- tcsh, 1

shell program, 19
shift, 21
short, 50
short-circuit, 27
showbase, 57
showpoint, 57
SHRT_MAX, 50
SHRT_MIN, 50
signature, 128
signed, 50
single quote, 6
singleton, 126
size_type, 53
sketch, 120
skipws, 57
slicing, 152
software development
- .cc, 136
- .h, 136
- .o, 135
- separate compilation, 133

source, 26
source file, 128, 130
source-code management, 34
source-code management-system, 34
ssh, 15
stack, 66, 87
stack, 166, 171
stack allocation, 93
Stamp, 140
static, 137
static block, 86, 117
static exit
- multi-exit, 69
- multi-level, 71

Multi-level exit, 71
static_cast, 62
status, 37
std, 48
stderr, 55
stdin, 55
stdout, 55
step, 103
strcat, 53
strcpy, 53
strcspn, 53
stream
- cerr, 55
- cin, 55
- clear, 59
- cout, 55
- fail, 56
- formatted, 55
- fstream, 56
- ifstream, 56
- ignore, 59
- input, 48
  - cin, 55
  - end of file, 58
  - eof, 58

fail, 58
manipulators
boolalpha, 57
dec, 57
endl, 57
fixed, 57
hex, 57
iomanip, 57
left, 57
noboolalpha, 57
noshowbase, 57
noshowpoint, 57
noskipws, 57
oct, 57
right, 57
scientific, 57
setfill, 57
setprecision, 57
setw, 57
showbase, 57
showpoint, 57
skipws, 57
ofstream, 56
output, 48
cout, 48
endl, 48
stream file, 55
stress testing, 46
string, 51, 52
C++
!=, 53
+, 53
<, 53
<=, 53
=, 53
==, 53
>, 53
>=, 53
[ ], 53
c_str, 53
find, 53
find_first_not_of, 53
find_first_of, 53
find_last_not_of, 53
find_last_of, 53
npos, 53
replace, 53
rfind, 53
size_type, 53
substr, 53
C
[ ], 53
strcat, 53
strcpy, 53
strcspn, 53
strlen, 53
strncat, 53
strncpy, 53
strspn, 53
strstr, 53
null termination, 51
stringstream, 74
strlen, 53
strncat, 53
strncpy, 53
strspn, 53
strstr, 53
struct, 106, 131
structurally equivalent, 83
structure, 76, 81, 85, 105, 106
member, 81, 105
data, 81
initialization, 81
routine, 81
visibility
default, 81
public, 81
struct, 61
structured programming, 68
subshell, 10, 19, 23
substitutability, 156
substr, 53
suffix
.C, 48
.c, 48
.cc, 48
.cpp, 48
switch, 67, 74
break, 67
case, 67

    default, 67
system command, 161
system modelling, 120
system testing, 46
system time, 9

tab key, 5
tcsh, 1
tcsh, 10
template, 47, 164
    routine, 165
    type, 165
template method, 159
template routine, 165
template type, 165
temporal, 141
terminal, 1
test, 26
test harness, 45
test-case design, 44
testing, 43
    acceptance, 46
    black-box, 44
    functional, 46
    gray-box, 44
    harness, 45
    human, 44
    integration, 45
    machine, 44
    performance, 46
    regression, 46
    security, 46
    stress, 46
    system, 46
    unit, 45
    usability, 46
    volume, 46
    white-box, 44
text merging, 36
this, 106
time, 9
time stamp, 161
translation unit, 133
true, 63
type, 9
type aliasing, 83
type coercion, 60
type constructor, 76
    array, 79
    enumeration, 76, 96
    literal, 85
    pointer, 77
    reference, 77
    structure, 81
    type aliasing, 83
    union, 83
type conversion, 62, 90, 110, 152
type equivalence, 144, 145
type inheritance, 142, 144
type nesting, 84
type qualifier, 50, 78
    const, 52, 78
    long, 50
    short, 50
    signed, 50
    static, 137
    unsigned, 50
type-constructor literal
    array, 85
    pointer, 85
    structure, 85
typedef, 84, 174

UINT8_MAX, 50
uint8_t, 50
UINT_MAX, 50
ULLONG_MAX, 50
ULONG_MAX, 50
unformatted I/O, 60
Unformatted I/O, 55
unidirectional association, 122
unified modelling language, 121
uninitialization, 111
uninitialized, 87
union, 83
unit testing, 45
unmanaged language, 91
unsigned, 50
usability testing, 46
user, 16

user time, 9
USHRT_MAX, 50
using
  declaration, 174
  directive, 174

value parameter, 87
variable declarations
  type qualifier, 50
variables
  constant, 52
  dereference, 62
  reference, 62
vector, 166, 167
  [], 167
  at, 167
  begin, 169
  clear, 167
  empty, 167
  end, 169
  erase, 169
  insert, 169
  pop_back, 167
  push_back, 167
  rbegin, 169
  rend, 169
  resize, 167, 168
  size, 167
version control, 34
virtual, 150, 152
virtual members, 150, 152–154
visibility, 84
  default, 81
  private, 131
  protected, 131
  public, 81, 131
visitor, 175
void *, 92
volume testing, 46

walkthrough, 44
wchar_t, 49
which, 9
while, 29
white-box testing, 44
whitespace, 49, 58
widening, 62
wildcard, 4, 15, 21
  qualifier, 15
working directory, 4, 7, 8, 11
wrapper member, 149
write, 16

xterm, 1

zero-filled, 85