## Contents

1 Shell
  1.1 File System .................................................. 4
  1.2 Pattern Matching .............................................. 7
  1.3 Quoting .......................................................... 11
  1.4 Shell Commands ................................................ 13
  1.5 System Commands .............................................. 19
  1.6 File Permission ............................................... 32
  1.7 Input/Output Redirection .................................... 35
  1.8 Variables ........................................................ 40
  1.9 Arithmetic ..................................................... 43
  1.10 Programming .................................................. 45
    1.10.1 Routine .................................................. 49
    1.10.2 Environment Variables .................................. 52
    1.10.3 Control Structures ...................................... 54
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.10.3.1</td>
<td>Test</td>
<td>54</td>
</tr>
<tr>
<td>1.10.3.2</td>
<td>Shift</td>
<td>56</td>
</tr>
<tr>
<td>1.10.3.3</td>
<td>Selection</td>
<td>57</td>
</tr>
<tr>
<td>1.10.3.4</td>
<td>Looping</td>
<td>60</td>
</tr>
<tr>
<td>1.11</td>
<td>Cleanup Script</td>
<td>64</td>
</tr>
<tr>
<td>1.12</td>
<td>Regress Script</td>
<td>67</td>
</tr>
<tr>
<td>2</td>
<td>C++</td>
<td>70</td>
</tr>
<tr>
<td>2.1</td>
<td>First Program</td>
<td>70</td>
</tr>
<tr>
<td>2.2</td>
<td>Program Structure</td>
<td>72</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Comment</td>
<td>72</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Statement</td>
<td>74</td>
</tr>
<tr>
<td>2.3</td>
<td>Declaration</td>
<td>75</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Identifier</td>
<td>75</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Basic Types</td>
<td>76</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Variable Declaration</td>
<td>76</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Type Qualifier</td>
<td>77</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Literals</td>
<td>80</td>
</tr>
<tr>
<td>2.4</td>
<td>Expression</td>
<td>84</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Conversion</td>
<td>89</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Coercion</td>
<td>91</td>
</tr>
</tbody>
</table>
2.16 Overloading .................................................. 210
2.17 Routine Pointer ............................................. 214
2.18 Object ...................................................... 218
  2.18.1 Object Member ....................................... 220
  2.18.2 Operator Member .................................... 223
  2.18.3 Constructor .......................................... 224
    2.18.3.1 Literal ........................................ 228
  2.18.3.2 Conversion ....................................... 229
  2.18.4 Destructor ........................................... 231
  2.18.5 Copy Constructor / Assignment .................... 234
  2.18.6 Initialize const / Object Member .................. 243
  2.18.7 Static Member ...................................... 245
2.19 Random Numbers ........................................... 248
2.20 Declaration Before Use .................................... 252
2.21 Encapsulation ............................................. 258
2.22 System Modelling ........................................ 264
  2.22.1 UML ................................................ 266
2.23 Separate Compilation ..................................... 280
2.24 Inheritance .............................................. 294
  2.24.1 Implementation Inheritance ......................... 294
2.24.2 Type Inheritance ................................. 298
2.24.3 Constructor/Destructor .......................... 304
2.24.4 Copy Constructor / Assignment .................. 305
2.24.5 Overloading .................................... 307
2.24.6 Virtual Routine ................................. 309
2.24.7 Downcast ....................................... 314
2.24.8 Slicing .......................................... 315
2.24.9 Protected Members ............................... 316
2.24.10 Abstract Class ................................. 317
2.24.11 Multiple Inheritance ............................ 321
2.24.12 UML ........................................... 323
2.25 Composition / Inheritance Design ................. 324
2.26 Template .......................................... 327
  2.26.1 Standard Library ............................... 331
    2.26.1.1 Vector ................................... 333
    2.26.1.2 Map ...................................... 340
    2.26.1.3 List ..................................... 344
    2.26.1.4 for_each .................................. 347
2.27 Namespace ........................................ 349

3 Tools ................................................. 355
3.1 C/C++ Composition ........................................ 355
3.2 Compilation .................................................. 356
  3.2.1 Preprocessor ............................................. 357
  3.2.2 Translator ............................................... 358
  3.2.3 Assembler ............................................... 360
  3.2.4 Linker .................................................... 360
3.3 Compiling Complex Programs ................................. 361
  3.3.1 Dependencies ........................................... 361
  3.3.2 Make .................................................. 364
3.4 Source-Code Management ..................................... 371
  3.4.1 SVN ................................................... 372
  3.4.2 Repository ............................................. 373
  3.4.3 Checking Out ......................................... 376
  3.4.4 Adding ................................................ 377
  3.4.5 Checking In ............................................ 379
  3.4.6 Modifying .............................................. 380
  3.4.7 Revision Number ....................................... 384
  3.4.8 Updating ............................................... 385
3.5 Debugger ..................................................... 391
  3.5.1 GDB ................................................... 392
## 4 Software Engineering

4.1 Software Crisis ........................................ 403
4.2 Software Development .................................. 404
4.3 Development Processes ................................ 405
4.4 Software Methodology .................................. 409
  4.4.1 System Design ...................................... 410
  4.4.2 Top-Down ........................................... 410
4.5 Design Quality .......................................... 412
  4.5.1 Coupling ............................................ 413
  4.5.2 Cohesion ............................................ 414
4.6 Design Principles ...................................... 418
4.7 Design Patterns ........................................ 418
  4.7.1 Pattern Catalog .................................... 420
    4.7.1.1 Class Patterns ................................. 421
    4.7.1.2 Object Patterns ............................... 425
4.8 Testing .................................................. 439
  4.8.1 Human Testing ..................................... 440
  4.8.2 Machine Testing ................................... 441
  4.8.3 Testing Strategies ................................ 443
  4.8.4 Tester ............................................... 445
1 Shell

- After signing onto a computer (login), a mechanism must exist to display information and perform operations.
- The two main approaches are graphical and command line.
- **Graphical user interface** (GUI) (desktop):
  - use icons to 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):
  - use text strings to access programs (commands) and data (file names),
  - command is typed after a prompt in an interactive area to start it,
  - arguments follow the command to affect its execution.
- Graphical interface is convenient, but seldom programmable.
- Command-line interface requires more typing, but allows programming.
- A **shell** is a program that reads commands and interprets them.
- It provides a simple programming-language with *string* variables and a few statements.
Unix shells falls into two basic camps, \texttt{sh (ksh, bash)} and \texttt{csh (tcsh)}, each with slightly different syntax and semantics.

Focus on bash with some tcsh.

Area (window) where shell runs is called a \texttt{terminal} or \texttt{xterm}.

Shell line begins with a \texttt{prompt} denoted by \$ (sh) or \% (csh) (often customized).

A command is typed after the prompt but \textit{not} executed until \texttt{Enter}/Return key is pressed.

\begin{verbatim}
$ date Enter # print current date
Thu Aug 20 08:44:27 EDT 2009
$ whoami Enter # print userid
jfdoe
$ echo Hi There! Enter # print any string
Hi There!
\end{verbatim}

Comment begins with a hash (\#) and continues to the end of line.

Multiple commands can be typed on the command line separated by the semi-colon.
$ date; whoami; echo Hi There!  # 3 commands
Sat Dec 19 07:36:17 EST 2009
jfdoe
Hi There!

• Commands can be edited on the command line (not sh):
  ◦ position cursor, ✾, with ▼ and ▶ arrow keys,
  ◦ remove characters before cursor with backspace/delete key,
  ◦ type new characters before cursor,
  ◦ press Enter at any point along the command line 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 terminates with command `exit`.

```bash
$ exit # exit shell and possibly terminal
```

- when the shell of terminal/xterm terminates, the terminal/xterm terminates.
- when the login terminal/xterm terminates, you sign off the 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 are organized in an N-ary tree: directories are vertices, files are leaves.
- Information is stored at specific locations in the hierarchy.
/ root of the 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
jford 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).
- u or home : user files are located in this directory.
• Directory for a particular user is called their **home directory**.

• Each file has a unique path-name in the file system, referenced with an absolute pathname.

• An **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 short names for a file using an implicit starting location.

• At sign on, the shell creates a **current directory** variable set to the user’s home directory.

• Any file name not starting with “/” is automatically prefixed with the current directory to create the necessary absolute pathname.

• A **relative pathname** is a list of all the directory names from the current directory to the file name separated by the character “/”.

• E.g., when user `jfdoe` signs on, home and current directory are set to `/u/jfdoe`.

  `.bashrc` # => `/u/jfdoe/.bashrc`
  `cs246/a1/q1x.C` # => `/u/jfdoe/cs246/a1/q1x.C`
• 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:
  o “.” points to current directory.
    ./cs246/a1/q1x.C # => /u/jfdoe/cs246/a1/q1x.C
  o “..” points to parent directory above the current directory.
    ../../../usr/include/limits.h # => /usr/include/limits.h

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 wildcards (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 current 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)

\[
\begin{align*}
[0-3] & \quad \# \Rightarrow 0 \ 1 \ 2 \ 3 \\
[a-zA-Z] & \quad \# \Rightarrow \text{lower or upper case letter} \\
[!a-zA-Z] & \quad \# \Rightarrow \text{any character not a letter}
\end{align*}
\]

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

\[
[-?]* \quad \# \Rightarrow \text{matches file names starting with -, ?, or *}
\]

• 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** contain administrative information and start with “.” (dot).
  ○ 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.

    $ ec	ab              # cause completion of command name to echo
    $ echo q1	ab        # cause completion of file name to q1x.C

• If the completion is ambiguous (i.e., more than one):
  ○ shell “beeps”,
  ○ prints all completions if tab is pressed again,
  ○ then you type more characters to uniquely identify the name.
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 \.\gabencories\.*
# print globbing pattern
.\[!.*
```

- **Backquote** (') : execute text as a command, and replace it with command output.

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

- Globbing does NOT occur within a single/double quoted string.
• **Single quote** (’): protect everything (including newline) except single quote.

```bash
$ echo ’.\[!.[!]\]*’          # no globbing
.\[!.[!]\]*
$ echo ’\.\[\![\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!]\]*’ # no escaping
\.\[\![\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!]\]*
$ echo ’abc
> cdf’                          # prompt “>” means current line is incomplete
abc
cdf
```

**A single quote cannot appear inside a single quoted string.**

- E.g., file name containing special characters (blanks/wildcards/comment).

  ```bash
  $ echo Book Report #2
  Book Report
  $ echo ’Book Report #2’
  Book Report #2
  ```

• **Double quote** (“) : protect everything except double quote, backquote, and dollar sign (variables), which can be escaped.
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).
- Shell commands read/write shell information/state.
- `help` : display information about bash commands (not sh or cs).

```
help [command-name]
```

```
$ help cd
cd: cd [-L|P] [dir]
    Change the shell working directory. ...
```
without argument, lists all bash commands.

- **cd**: change the current 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
  ```

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

- **pwd**: print the current directory.

  ```
  $ pwd
  /u/jfdoe/cs246
  ```
• **history** and “!” (bang!) : print a numbered history of most recent commands entered and access them.

```bash
$ history
1  date
2  whoami
3  echo  Hi There
4  help
5  cd  .
6  pwd
```

```bash
$ !2  # rerun 2nd history command
$ whoami
$ whoami
```

```bash
$ !  # rerun last history command
$ !ec  # rerun last history command starting with “ec”
echo  Hi There
echo  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.

```bash
$ △ 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.*
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) `.sherrc` file,
2. place a list of `alias` commands in a file (often `.aliases`) and `source` that file from the `.sherrc` 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
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:
    
    ```
    my execution
    u | s | | u | s | | u | u | u | • • • | u
    ----
    r
    ```
    
    * **real time** is from start to end including interleavings: user + system \( \approx \) real-time
  
  ○ different shells print these values differently.
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, which usually means success.

1.5 **System Commands**

• Command programs called by shell (versus executed by shell).

• **sh / bash / csh / tcsh** : start **subshell**.
$ \ldots \quad \# \text{bash commands}
$ tcsh \quad \# \text{start tcsh in bash}
% \ldots \quad \# \text{tcsh commands}
% sh \quad \# \text{start sh in tcsh}
$ \ldots \quad \# \text{sh commands}
$ \textbf{exit} \quad \# \text{exit sh}
% \textbf{exit} \quad \# \text{exit tcsh}
$ \textbf{exit} \quad \# \text{exit original bash and terminal}

◦ Allows switching among shells for different purposes.

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

$ \textbf{echo} \$\{0\} \quad \# \text{what shell am I using?}
/bin/tcsh
$ chsh \quad \# \text{change to different shell}
Password: \textbf{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 and function.
$ man bash
... # information about “bash” command
$ man chsh
... # information about “chsh” command
$ man man
... # information about “man” command

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

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

  - `-a` lists all files, including hidden files
  - `-l` generates a long listing (details) for each file
  - no file/directory name implies current directory

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

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

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

• 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

  ◦ if the target-file does not exist, the source-file is renamed; otherwise the target-file is replaced.
  ◦ -i prompt for verification if a target file is being replaced.
$ 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 verification for each file/directory being removed.
- `-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

• Use alias for setting command options for particular commands.

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

which always uses the -i option 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.

• 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

• 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)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Owner</th>
<th>Job</th>
<th>Files</th>
<th>Total Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>rggowner</td>
<td>308</td>
<td>tt22</td>
<td>10999276 bytes</td>
</tr>
<tr>
<td>2nd</td>
<td>jfdoe</td>
<td>403</td>
<td>uml.ps</td>
<td>41262 bytes</td>
</tr>
</tbody>
</table>

$ lprm 403                 # cancel printing
services203.math: cfA403services16.student.cs dequeued
$ lpstat                  # check if cancelled
Spool queue: lp (ljp_3016)

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

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

<table>
<thead>
<tr>
<th>file x</th>
<th>file y</th>
</tr>
</thead>
<tbody>
<tr>
<td>a\n</td>
<td>a\n</td>
</tr>
<tr>
<td>b\n</td>
<td>b\n</td>
</tr>
<tr>
<td>c\n</td>
<td>c\n</td>
</tr>
<tr>
<td>d\n</td>
<td>e\n</td>
</tr>
<tr>
<td>g\n</td>
<td>h\n</td>
</tr>
<tr>
<td>h\n</td>
<td>i\n</td>
</tr>
<tr>
<td>g\n</td>
<td>g\n</td>
</tr>
</tbody>
</table>

newline is counted ⇒ 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 current directory, “.”
- if `[ expr ]` omitted, match all file names, “–name ".*"”
- recursively find file/directory names starting in current 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 `file` or `directory`
- `–maxdepth N` recursively descend at most `N` directory levels (0 ⇒ current directory)
- logical `not`, `and` and `or` (precedence order)

```
–not expr
expr –a expr
expr –o expr
```

–a assume if no operator, `expr expr ⇒ expr –a expr`
( expr ) evaluation order

- recursively find only file names starting in current 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` : (extended global regular expression print) 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”

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

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

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

list lines that match start of line “^”, match “#include”, match 1 or more space or tab “[ ]+”, match either “"” or “<”, match 1 or more characters “.+”, match either “"” or “>”, match end of line “$” in files with suffix “.h” or “.cc”
$ \text{egrep}'^\wedge\text{\#include[ ]+['<']\ldots['>']}$*.h,cc\ # why quotes ?
\text{egrep: }*.h: \text{No such file or directory}
\text{q1.cc:#include <iostream>}
\text{q1.cc:#include <iomanip>}
\text{q1.cc:#include "q1.h"}

○ \textbf{egrep pattern is different from globbing pattern.}
  Most important difference is “*” is a wildcard qualifier not a wildcard.

• \textbf{ssh:} (secure shell) safe, encrypted, remote-login between client/server hosts.
  \text{\texttt{\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:

  \texttt{$ \text{ssh -Y -l jfdoe linux.student.cs.uwaterloo.ca}$}
  \ldots # enter password, run commands (editor, programs)
  \texttt{$ \text{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:

\[ d = \text{directory} \quad - = \text{file} \]

- user permission
- group permissions
- other permissions

\[ \text{d} \quad \text{rwx} \quad \text{r-x} \quad --\]

E.g., d \text{rwx} \text{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.

\text{In general, never allow “other” users to read or write your files.}

Default permissions (usually) on:

- file: \text{rw- r-- --}, owner read/write, group only read, other none.
- directory: \text{rwx -- -- --}, owner read/write/execute, group/other none.

\text{chgrp : change group-name associated with file.}

\text{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 denoted by u for user, g for group, o for other, a for all (ugo).
- Operator + adds permission, − removes permission.
- Permissions are denoted by r for readable, w for writable and x for executable.
- Elements of the mode-list are separated by commas.
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.

```
$ 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 Variables

• syntax : `[\_a–zA–Z][\_a–zA–Z0–9]` where “*” is wildcard qualifier

• case-sensitive:

  - `VeryLongVariableName`  
  - `Page1`  
  - `Income_Tax`  
  - `75`

• Some identifiers are reserved (e.g., `if`, `while`), and hence, keywords.

• **Variables** ONLY hold string values (arbitrary length).

• Variable is declared **dynamically** by assigning a value with operator “*”.

  ```
  $ cs246assn=/u/jfdoe/cs246/a1  # declare and assign
  
  No spaces before or after “*”.
  ```

• A variable’s value is dereferenced using operators “$” or “${}”.

  ```
  $ echo $cs246assn ${cs246assn}
  /u/jfdoe/cs246/a1 /u/jfdoe/cs246/a1
  $ cd $cs246assn  # or ${cs246assn}
  ```

• Unlike alias, variable can be command or argument.
`$ 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`

- Dereferencing undefined variable returns empty string.

`$ echo $xxx`  # *no output just newline*

# empty line

- **Beware concatenation.**

  `$ cd $cs246assndata`  # *change to /u/jfdoe/cs246/a1data*

  Where does this move to?

- Always use braces to allow concatenation with other text.

  `$ cd ${cs246assn}data`  # *cd /u/jfdoe/cs246/a1data*
• **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 the 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?

• To make this work, 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

### 1.9 Arithmetic

- Shell variables have type string.

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

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

- 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 $((expression))$.

$\texttt{echo$((3 + 4 - 1))}$
6

$\texttt{echo$((3 +${i} * 2))}$
9

$\texttt{echo$((3 +${k}))} # k is unset$
bash: 3 + : syntax error: operand expected (error token is " ")

• Basic integer operations, $+$, $-$, $\ast$, $/$, $\%$ (modulus), with usual precedence, and $()$.

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

$\texttt{echo`expr 3 + 4 - 1`} # for sh, csh$
6

$\texttt{echo`expr 3 +${i} \ast 2`} # escape $\ast$
9

$\texttt{echo`expr 3 +${k}`} # k is unset$
expr: non-numeric argument
1.10 Programming

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

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

- First line should begin with magic comment: “#!” (sha-bang) with shell pathname for executing the script.

- It forces a specific shell to be used, which is run as a subshell.

- If the “#!” line is missing, a subshell of the same kind as the invoking shell is used 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   # 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:
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.

```bash
$ 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.
1.10.1 Routine

A routine is defined as follows:

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

Invoke like a command.

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

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

```bash
usage() {
    echo "Usage: ${0} -t -g -e input-file [ output-file ]"
    exit 1 # terminate script with non-zero exit code
}
```

Routine arguments are accessed the same as in the 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
• Routines/variables must be created before used, are then visible throughout the script, and can be removed.

```bash
rtn1() {
    var=3  # new variable
    rtn2  # call rtn2, see all routines
    unset rtn2  # remove routine!!!
}
rtn2() {
    echo ${var}  # see all variables
    unset var  # remove variable!!!
}
rtn1  # call
```

• **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.

```bash
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.

### 1.10.2 Environment Variables

- Each shell has a set of environment (global) and script (local/parameters) variables.
- Shell has a $N$ sets of variables: environment, local, arguments for routine calls $C_{1-i}$.

```
Shell (command)

<table>
<thead>
<tr>
<th>Envir: $E0 $E1 $E2...</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local: $L0 $L1 $L2...</td>
<td>2</td>
</tr>
<tr>
<td>Args(_1): $0 $1 $2...</td>
<td></td>
</tr>
<tr>
<td>(call stack)</td>
<td></td>
</tr>
<tr>
<td>Args(_i): $0 $1 $2...</td>
<td></td>
</tr>
</tbody>
</table>
```

- 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 its containing shell (environment variables only affect subshells).
• Only put a variable in the environment list to make it accessible by subshells.

1.10.3 Control Structures

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

1.10.3.1 Test

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

<table>
<thead>
<tr>
<th>test</th>
<th>operation</th>
<th>priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>! expr</td>
<td>not</td>
<td>high</td>
</tr>
<tr>
<td>( expr )</td>
<td>evaluation order (must be escaped)</td>
<td></td>
</tr>
<tr>
<td>expr1 –a expr2</td>
<td>logical and (not short-circuit)</td>
<td>low</td>
</tr>
<tr>
<td>expr1 –o expr2</td>
<td>logical or (not short-circuit)</td>
<td></td>
</tr>
</tbody>
</table>

• test comparison is performed using the following:
<table>
<thead>
<tr>
<th>test</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>string1 = string2</td>
<td>equal (not ==)</td>
</tr>
<tr>
<td>string1 != string2</td>
<td>not equal</td>
</tr>
<tr>
<td>integer1 –eq integer2</td>
<td>equal</td>
</tr>
<tr>
<td>integer1 –ne integer2</td>
<td>not equal</td>
</tr>
<tr>
<td>integer1 –ge integer2</td>
<td>greater or equal</td>
</tr>
<tr>
<td>integer1 –gt integer2</td>
<td>greater</td>
</tr>
<tr>
<td>integer1 –le integer2</td>
<td>less or equal</td>
</tr>
<tr>
<td>integer1 –lt integer2</td>
<td>less</td>
</tr>
<tr>
<td>–d file</td>
<td>exists and directory</td>
</tr>
<tr>
<td>–e file</td>
<td>exists</td>
</tr>
<tr>
<td>–f file</td>
<td>exists and regular file</td>
</tr>
<tr>
<td>–r file</td>
<td>exists with read permission</td>
</tr>
<tr>
<td>–w file</td>
<td>exists with write permission</td>
</tr>
<tr>
<td>–x file</td>
<td>exists with executable or searchable</td>
</tr>
</tbody>
</table>

- 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

1.10.3.2 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
#!/bin/bash
echo ${1}; shift 1
echo ${1}; shift 2
echo ${1}; shift 3
echo ${1}

$ ./scriptfile 1 2 3 4 5 6 7
1 2 4 7

1.10.3.3 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:

```bash
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.:

  ```bash
  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:

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

1.10.3.4 **Looping**

- **while** statement executes its commands zero or more times.
while test-command do
  commands
done

• test-command is evaluated; exit status of zero implies true, otherwise false.

• Check for different conditions:

  # search command-line parameters for "-x"
  while [ "$1" != "-x" ] ; do  # string compare
    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?
  ...
  i=$((i + 1))  
  file=data${i}
  # process file
  # advance to next file
done
```

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

```bash
for index [ in list ] ; do
  commands
done
for name in ric peter jo mike ; do
  echo ${name}
done
for arg in "${@}" ; do
  echo ${arg}
done
```

If no “in list”, iterate over quoted parameters, i.e., "${@}".

- Or over a set of values:
for (( init Expr; test Expr; incr Expr )); do
  commands
done

for (( i = 1; i <= $#; i += 1 )); do
  eval echo "\${${i}}"  # ${1-#}
done

- Use directly on command line:
  
  $ for file in *.C ; do cp "${file}" "${file}.old" ; done

- A while/for loop may contain break and continue to terminate loop or advance to the next loop iteration.
# process files data1, data2, ...
i=1

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

1.11 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
while [ $# -gt 0 ] ; do # process command-line arguments
    case "$1" in
        "-h") usage ;;;
        "-r" | "-R") depth="" ;;;
        "-i" | "-f") prompt="${1}" ;;;
        *)
            remove "$1"
            defaults
            ;;
    esac
    shift # remove argument
done
#!/bin/bash
#
# Compare output from two programs printing any differences.
#
# Usage: regress program1 'program1-options' program2 'program2-options' [ argument-list ]
#
# Examples:
# regress ls '' ls ''
# regress ls '' ls '' ''"a b c""
# regress cat '' cat '-n' regress regress
# regress regress "cat '"' ' cat '-n'"' regress "cat '"' ' cat '-n'"' regress regress
# regress myprog '-w' samplesoln '-w' 27 100 -2 -100

usage() {
    echo "Usage: ${0} program1 "program1-options" \
         program2 "program2-options" argument-list"
    exit 1
}

process() {
    for parm in "${@}"; do
        # process parameters
        # must use eval to reevaluate parameters
        eval ${prog1} ${opts1} ${parm} > /tmp/tmp1_${$} 2>&1 # run programs
        eval ${prog2} ${opts2} ${parm} > /tmp/tmp2_${$} 2>&1 # and save output
        diff /tmp/tmp1_${$} /tmp/tmp2_${$} # compare output from programs
        if [ $? -eq 0 ]; then
            # check return code
            echo "identical output"
        fi
        rm /tmp/tmp1_${$} /tmp/tmp2_${$} # remove temporary files
    done
}

if [ ${#} -lt 4 ]; then usage; fi # check command-line arguments
if [ ! -x "`type -P ${1}" ]; then echo "program1 is unexecutable"; usage; fi
if [ ! -x "`type -P ${3}" ]; then echo "program2 is unexecutable"; usage; fi
prog1=${1}
opts1=${2}
prog2=${3}
opts2=${4}
shift 4
if [ ${#} -eq 0 ]; then process "" # process empty argument-list
else process "${@}"; fi # process argument-list
Without `eval`:

```python
regress ls '' ls '' ''"a b c"''
```

becomes `ls a b c`, rather than `ls 'a b c'`. 
2 C++

2.1 First Program

- Java

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

- C/C++

```c
#include <stdio.h>

int main() {
    printf("Hello!\n");
    return 0;
}
```

```cpp
#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).

• **int** main() is the routine where execution starts.

• curly braces, `{ ... }`, denote a block of code, i.e., routine body of main.

• 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, `'
'` in C).

• Optional **return** 0 returns zero to the shell indicating successful completion of the program; non-zero usually indicates an error.

• **main magic! If no value is returned, 0 is implicitly returned.**

• Routine exit (Java System.exit) 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 ).
Java/C/C++ program are transformed from human readable form (text) to machine readable form (numbers) for execution, called compilation.

Compilation is performed by a compiler; several compilers exist for C++.
Compile with g++ command:

```
$ g++ firstprogram.cc    # compile program, generate executable "a.out"
$ ./a.out                # execute program; execution permission
```

C program-files use suffix .c; C++ program-files use suffixes .C / .cpp / .cc.

### 2.2 Program Structure

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

#### 2.2.1 Comment

- Comments document what a program does and how it does it.
- Comment may appear where whitespace (space, tab, newline) is allowed.
Two kinds of comments in C/C++ (same in Java):

<table>
<thead>
<tr>
<th>Java / C / C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  /* . . . */</td>
</tr>
<tr>
<td>2  // remainder of line</td>
</tr>
</tbody>
</table>

• First comment begins with the start symbol, /*, and ends with the terminator symbol, */, and hence, can extend over multiple lines.

• **Cannot be nested one within another:**

```
/* . . . */
```

```
end comment treated as statements
```

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

```c
/* attempt to comment-out a number of statements
   while ( . . . ) {
      /* . . . nested comment causes errors */
      if ( . . . ) {
         /* . . . nested comment causes errors */
      }
   }
/*
```
• Second comment begins with the start symbol, //, and continues to the end of the line, i.e., only one line long.

• Can be nested one within another:

```cpp
// … // … nested comment
```

so it can be used to comment-out code:

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

### 2.2.2 Statement

• The syntax for a C/C++ statement is a series of tokens separated by whitespace and terminated by a semicolon (except for a block, {}).
2.3 Declaration

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

2.3.1 Identifier

- name used to refer to a variable or type.
- syntax: [_a–zA–Z][_a–zA–Z0–9]* where “*” is wildcard qualifier
- case-sensitive:
  
  `VeryLongVariableName Page1 Income_Tax _75`

- Some identifiers are reserved (e.g., `if`, `while`), and hence, **keywords**.
### 2.3.2 Basic Types

<table>
<thead>
<tr>
<th>Java</th>
<th>C / C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>bool (C <code>&lt;stdbool.h&gt;</code>)</td>
</tr>
<tr>
<td>char</td>
<td>char / wchar_t</td>
</tr>
<tr>
<td>byte</td>
<td>char / wchar_t</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
</tbody>
</table>

- ASCII / unicode character
- integral types
- real-floating types
- label type, implicit

- C/C++ treat `char / wchar_t` as character and integral type.
- Java types `short` and `long` are created using type qualifiers.

### 2.3.3 Variable Declaration

- C/C++ declaration: type followed by list of identifiers, except label with an implicit type (same in Java).
Declarations may have an initializing assignment (except for fields in struct/class):

```c
int i = 3;  // a single initialization
int i = 3, j = 4, k = 5;  // a compound initialization
```

Value of an **uninitialized variable** is usually undefined.

```c
int i;
cout << i << endl;  // i has undefined value
```

Some C/C++ compilers check for uninitialized variables (use `-Wall` option).

### 2.3.4 Type Qualifier

- Other integral types are composed with type qualifiers modifying integral types `char` and `int`. 
• C/C++ provide size (\texttt{short}, \texttt{long}) and signed-ness (\texttt{signed} ⇒ positive/negative, \texttt{unsigned} ⇒ positive only) qualifiers.

• \texttt{int} provides \textit{relative} machine-specific types: usually \texttt{int} \geq 2 \text{ bytes for} 16-bit computer and 4 \text{ bytes for} 32/64-bit computer, \texttt{long} \geq \texttt{int}, \texttt{long long} \geq \texttt{long}.

• \texttt{#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:

<table>
<thead>
<tr>
<th>integral types</th>
<th>range (lower/upper bound name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{char} (\texttt{signed char})</td>
<td>\texttt{SCHAR_MIN} to \texttt{SCHAR_MAX}, e.g., -128 to 127</td>
</tr>
<tr>
<td>\texttt{unsigned char}</td>
<td>0 to \texttt{UCHAR_MAX}, e.g. 0 to 255</td>
</tr>
<tr>
<td>\texttt{short} (\texttt{signed short int})</td>
<td>\texttt{SHRT_MIN} to \texttt{SHRT_MAX}, e.g., -32768 to 32767</td>
</tr>
<tr>
<td>\texttt{unsigned short} (\texttt{unsigned short int})</td>
<td>0 to \texttt{USHRT_MAX}, e.g., 0 to 65535</td>
</tr>
<tr>
<td>\texttt{int} (\texttt{signed int})</td>
<td>\texttt{INT_MIN} to \texttt{INT_MAX}, e.g., -2147483648 to 2147483647</td>
</tr>
<tr>
<td>\texttt{unsigned int}</td>
<td>0 to \texttt{UINT_MAX}, e.g., 0 to 4294967295</td>
</tr>
<tr>
<td>\texttt{long} (\texttt{signed long int})</td>
<td>(\texttt{LONG_MIN} to \texttt{LONG_MAX}), e.g., -2147483648 to 2147483647</td>
</tr>
<tr>
<td>\texttt{unsigned long} (\texttt{unsigned long int})</td>
<td>0 to (\texttt{ULONG_MAX}, e.g. 0 to 4294967295</td>
</tr>
<tr>
<td>\texttt{long long} (\texttt{signed long long int})</td>
<td>\texttt{LLONG_MIN} to \texttt{LLONG_MAX}, e.g., -9223372036854775808 to 9223372036854775807</td>
</tr>
<tr>
<td>\texttt{unsigned long long} (\texttt{unsigned long long int})</td>
<td>0 to (\texttt{ULLONG_MAX}), e.g., 0 to 18446744073709551615</td>
</tr>
</tbody>
</table>

• \texttt{#include <stdint.h>} provides \textit{absolute} types [u]\texttt{intN_t} for \texttt{signed/unsigned} \texttt{N} = 8, 16, 32, 64 \text{ bits}.
<table>
<thead>
<tr>
<th>integral types</th>
<th>range (lower/upper bound name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8_t</td>
<td>INT8_MIN to INT8_MAX, e.g., −128 to 127</td>
</tr>
<tr>
<td>uint8_t</td>
<td>0 to UINT8_MAX, e.g., 0 to 255</td>
</tr>
<tr>
<td>int16_t</td>
<td>INT16_MIN to INT16_MAX, e.g., −32768 to 32767</td>
</tr>
<tr>
<td>uint16_t</td>
<td>0 to UINT16_MAX, e.g., 0 to 65535</td>
</tr>
<tr>
<td>int32_t</td>
<td>INT32_MIN to INT32_MAX, e.g., −2147483648 to 2147483647</td>
</tr>
<tr>
<td>uint32_t</td>
<td>0 to UINT32_MAX, e.g., 0 to 4294967295</td>
</tr>
<tr>
<td>int64_t</td>
<td>INT64_MIN to INT64_MAX, e.g., −9223372036854775808 to 9223372036854775807</td>
</tr>
<tr>
<td>uint64_t</td>
<td>0 to UINT64_MAX, e.g., 0 to 18446744073709551615</td>
</tr>
</tbody>
</table>

- C/C++ provide two basic real-float types `float` and `double`, and one real-float type generated with type qualifier.

- `#include <cfloat>` specifies names for precision and magnitude of real-float values.
### real-float types  range (precision, magnitude)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>FLT_DIG precision, FLT_MIN_10_EXP to FLT_MAX_10_EXP, 6+ digits over range 10^{-38} to 10^{38}, IEEE (4 bytes)</td>
</tr>
<tr>
<td>double</td>
<td>DBL_DIG precision, DBL_MIN_10_EXP to DBL_MAX_10_EXP, 15+ digits over range 10^{-308} to 10^{308}, IEEE (8 bytes)</td>
</tr>
<tr>
<td>long double</td>
<td>LDBL_DIG precision, LDBL_MIN_10_EXP to LDBL_MAX_10_EXP, 18-33+ digits over range 10^{-4932} to 10^{4932}, IEEE (12-16 bytes)</td>
</tr>
</tbody>
</table>

**float**:  \( \pm 1.17549435e^{-38} \) to  \( \pm 3.40282347e^{+38} \)  
**double**:  \( \pm 2.2250738585072014e^{-308} \) to  \( \pm 1.7976931348623157e^{+308} \)  
**long double**:  \( \pm 3.36210314311209350626e^{-4932} \) to  \( \pm 1.18973149535723176502e^{+4932} \)

#### 2.3.5 Literals

- Variables contain values, and each value has a **constant** (C) or **literal** (C++) meaning.
- E.g., the integral value 3 is constant/literal, i.e., it cannot change, it always means 3.

```
3 = 7;  // disallowed
```
- Every basic type has a set of literals that define its values.
• A variable’s value always starts with a literal, and changes via another literal or computation.

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

<table>
<thead>
<tr>
<th>type</th>
<th>literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>false, true</td>
</tr>
<tr>
<td>character</td>
<td>'a', ''</td>
</tr>
<tr>
<td>integral</td>
<td>decimal: 123, -456, 123456789, octal, prefix 0: 0144, -045, 04576132</td>
</tr>
<tr>
<td></td>
<td>hexadecimal, prefix 0X / 0x: 0xfe, -0X1f, 0xe89abc3d</td>
</tr>
<tr>
<td>real-floating</td>
<td>.1, 1., -1., 0.52, -7.3E3, -6.6e-2, E/e exponent</td>
</tr>
</tbody>
</table>

• 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
  ```

• Escape sequence provides quoting of special characters in a char literal using a backslash, \.
\| backslash
| ' ' | single 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 4 hexadecimal digits for \texttt{char}, up to 4 hexadecimal digits for \texttt{wchar_t} (not Java)

```cpp
cout << '\\' << endl
   << ' ' << endl
   << '\t' << '\t' << 'x' << '\n' // newline value 10
   << 'y' << '\12' // octal for 10
   << 'z' << '\xa'; // hexadecimal for 10
```

- C/C++ provide user named literals (write-once/read-only variables) with type qualifier \texttt{const} (Java \texttt{final}).
<table>
<thead>
<tr>
<th>Java</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>final char Initial = 'D';</td>
<td>const char Initial = 'D';</td>
</tr>
<tr>
<td>final short int Size = 3, SupSize;</td>
<td>const short int Size = 3, SupSize = Size + 7;</td>
</tr>
<tr>
<td>SupSize = Size + 7;</td>
<td>disallowed</td>
</tr>
<tr>
<td>final double PI = 3.14159;</td>
<td>const double PI = 3.14159;</td>
</tr>
</tbody>
</table>

- **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 practice is to name literals so all usages can be changed via its initialization value.

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

- There are trillions of literals ⇒ cannot all be stored in memory.

- Only the literals in a program use storage, some are embedded into computer instructions.
## 2.4 Expression

<table>
<thead>
<tr>
<th>Java</th>
<th>C/C++</th>
<th>priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>postfix</td>
<td>::, ., -&gt; [], call</td>
<td>high</td>
</tr>
<tr>
<td>prefix</td>
<td>+, -, !, ~, cast,</td>
<td></td>
</tr>
<tr>
<td>(unary) new</td>
<td>+, −, !, ~, &amp; *</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>*, /, %</td>
<td></td>
</tr>
<tr>
<td>bit shift</td>
<td>&lt;&lt;, &gt;&gt;, &gt;&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>relational</td>
<td>&lt;, &lt;=, &gt;, &gt;=, instanceof</td>
<td></td>
</tr>
<tr>
<td>equality</td>
<td>==, !=</td>
<td></td>
</tr>
<tr>
<td>bitwise &amp; and</td>
<td>&amp; ^</td>
<td></td>
</tr>
<tr>
<td>logical &amp; &amp;  short-circuit</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>conditional</td>
<td>?:</td>
<td></td>
</tr>
<tr>
<td>assignment</td>
<td>=, +=, −=, *=', /=, %=</td>
<td></td>
</tr>
<tr>
<td>comma</td>
<td>,</td>
<td>low</td>
</tr>
</tbody>
</table>

- Expression evaluation is like algebra:
• operations exist and are called using name with parenthesized argument(s).

\[
\begin{align*}
\text{abs}(-3); & \quad | -3 | \\
\text{sqrt}(x); & \quad \sqrt{x} \\
\text{pow}(x, y); & \quad x^y
\end{align*}
\]

• operators are prioritized and performed from high to low.

\[
\begin{align*}
x + y \times \text{sqrt}(z); & \quad \text{// call, multiple, add} \\
x + y - z; & \quad \text{// add, subtract} \\
3.0 / v \times w; & \quad \text{// divide, multiple}
\end{align*}
\]

e except for unary, ?, and assignment operators, which associate right to left.

\[
\begin{align*}
-x; & \quad \text{// complement, negate} \\
* & p; & \quad \text{// address-of, dereference} \\
x = y = z; & \quad \text{// z to y to x}
\end{align*}
\]

• parentheses control order of evaluation, i.e., override rules

\[
\begin{align*}
x + y \times z; & \quad \text{// multiple, add} \\
(x + y) \times z; & \quad \text{// add, multiple}
\end{align*}
\]
• Subexpressions and argument evaluation is unspecified (Java left to right)

\[
\begin{align*}
  (i + j) \times (k + j) &; \quad \text{// either } + \text{ done first} \\
  (i = j) + (j = i) &; \quad \text{// either } = \text{ done first} \\
  g(i) + f(k) + h(j) &; \quad \text{// } g, f, \text{ or } h \text{ called in any order} \\
  f(p++, p++, p++) &; \quad \text{// arguments evaluated in any order}
\end{align*}
\]

• **Beware of overflow.**

\[
\begin{align*}
  \text{unsigned int } a = 4294967295, b = 4294967295, c = 4294967295; \\
  (a + b) / c; \quad \text{// } \Rightarrow 0 \text{ as } a+b \text{ overflows leaving zero} \\
  a / c + b / c; \quad \text{// } \Rightarrow 2
\end{align*}
\]

Perform extra operations (2 divides) to keep numbers small.

• C++ relational/equality return **false/true**; C return 0/1.

• Referencing (address-of), &, and dereference, *, operators do not exist in Java because access to storage is restricted.

• Pseudo-routine **sizeof** returns the number of bytes for a type or variable (not in Java):

\[
\begin{align*}
  \text{long int } i; \\
  \text{sizeof}(\text{long int}); \quad \text{// type, at least 4} \\
  \text{sizeof}(i); \quad \text{// variable, at least 4}
\end{align*}
\]
The `sizeof` a pointer (type or variable) is the size of the pointer on that
particular computer and not the size of the type the pointer references.

- 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.

```cpp
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?

- Division operator, `/`, accepts integral and real-float operands, but truncates for integrals.

```
3 / 4 // 0 not 0.75
3.0 / 4.0 // 0.75
```
• Remainder (modulus) operator, %, only accepts integral operands.
  ◦ If either operand is negative, the sign of the remainder is implementation defined, e.g., 3 % -4, -3 % -4 can be 3 or -3.

• Assignment is an operator; useful for **cascade assignment** to initialize multiple variables of the same type:

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

  ◦ **Other uses of assignment in an expression are discouraged!**; i.e., assignments only on left side.

• General assignment operators, e.g., `lhs += rhs` implicitly rewritten as:

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

not:

```plaintext
lhs = lhs + rhs;
```

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

```plaintext
v[ f(3) ] += 1; // only calls f once
v[ f(3) ] = v[ f(3) ] + 1; // calls f twice
```
• Comma expression allows multiple expressions to be evaluated in a context where only a single expression is allowed.

\[ x, f + g, \sqrt{3}/2, m[i][j] \leftarrow \text{value returned} \]

○ Expressions evaluated left to right with the value of rightmost expression returned.

• Operators ++ / -- are discouraged because subsumed by general += / -=.

\[
\begin{align*}
i &+= 1; \quad \text{versus} \quad i++ \\
i &+= 3; \quad \text{versus} \quad i++ ++ +++; \quad \text{// disallowed}
\end{align*}
\]

2.4.1 Conversion

• Conversion transforms a value to another type by changing the value to the new type’s representation.

• Conversions can occur implicitly by the compiler or explicitly by the programmer using a cast operator or C++ static_cast operator.

```cpp
int i; double d;
d = i; \quad \text{// implicit (compiler)}
d = (double)i; \quad \text{// explicit with cast (programmer)}
d = \text{static_cast<double>>(i)};
```
Two kinds of conversions:

- **widening/promotion** conversion, no information is lost:
  
  \[
  \text{bool} \rightarrow \text{char} \rightarrow \text{short int} \rightarrow \text{long int} \rightarrow \text{double}
  \]

  where \( \text{false} \rightarrow 0; \text{true} \rightarrow 1 \)

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

  \[
  \text{double} \rightarrow \text{long int} \rightarrow \text{short int} \rightarrow \text{char} \rightarrow \text{bool}
  \]

  where \( 0 \rightarrow \text{false}; \text{non-zero} \rightarrow \text{true} \)

- 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 \rightarrow 3.5 
  d = i = 3.5; // d \rightarrow 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/C++ supports casting among the basic types and user defined types.

2.4.2 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 may or may not be reasonable as it might generate an invalid character.

• But the most common coercion is through pointers:
Using the explicit cast, programmer has lied to the compiler about the type of `ip`.

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

### 2.4.3 Math Operations

- `#include <cmath>` provides overloaded real-float mathematical-routines for types **float, double** and **long double**:
<table>
<thead>
<tr>
<th>operation</th>
<th>routine</th>
<th>operation</th>
<th>routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>x</td>
<td>\</td>
<td>$</td>
</tr>
<tr>
<td>arccos $x$</td>
<td>acos( $x$ )</td>
<td>$\ln x$</td>
<td>log( $x$ )</td>
</tr>
<tr>
<td>arcsin $x$</td>
<td>asin( $x$ )</td>
<td>$\log x$</td>
<td>log10( $x$ )</td>
</tr>
<tr>
<td>arctan $x$</td>
<td>atan( $x$ )</td>
<td>$x^y$</td>
<td>pow( $x$, $y$ )</td>
</tr>
<tr>
<td>$\lceil x \rceil$</td>
<td>ceil( $x$ )</td>
<td>$\sin x$</td>
<td>sin( $x$ )</td>
</tr>
<tr>
<td>cos $x$</td>
<td>cos( $x$ )</td>
<td>$\sinh x$</td>
<td>sinh( $x$ )</td>
</tr>
<tr>
<td>cosh $x$</td>
<td>cosh( $x$ )</td>
<td>$\sqrt{x}$</td>
<td>sqrt( $x$ )</td>
</tr>
<tr>
<td>$e^x$</td>
<td>exp( $x$ )</td>
<td>$\tan x$</td>
<td>tan( $x$ )</td>
</tr>
<tr>
<td>$\lfloor x \rfloor$</td>
<td>floor( $x$ )</td>
<td>$\tanh x$</td>
<td>tanh( $x$ )</td>
</tr>
</tbody>
</table>

and math literals:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_E</td>
<td>2.7182818284590452354</td>
<td>$e$</td>
</tr>
<tr>
<td>M_LOG2E</td>
<td>1.4426950408889634074</td>
<td>$\log_2 e$</td>
</tr>
<tr>
<td>M_LOG10E</td>
<td>0.43429448190325182765</td>
<td>$\log_{10} e$</td>
</tr>
<tr>
<td>M_LN2</td>
<td>0.69314718055994530942</td>
<td>$\log e 2$</td>
</tr>
<tr>
<td>M_LN10</td>
<td>2.30258509299404568402</td>
<td>$\log e 10$</td>
</tr>
<tr>
<td>M_PI</td>
<td>3.14159265358979323846</td>
<td>$\pi$</td>
</tr>
<tr>
<td>M_PI_2</td>
<td>1.57079632679489661923</td>
<td>$\pi/2$</td>
</tr>
<tr>
<td>M_PI_4</td>
<td>0.78539816339744830962</td>
<td>$\pi/4$</td>
</tr>
<tr>
<td>M_1_PI</td>
<td>0.31830988618379067154</td>
<td>$1/\pi$</td>
</tr>
<tr>
<td>M_2_PI</td>
<td>0.63661977236758134308</td>
<td>$2/\pi$</td>
</tr>
<tr>
<td>M_2_SQRTPI</td>
<td>1.12837916709551257390</td>
<td>$2/\sqrt{\pi}$</td>
</tr>
<tr>
<td>M_SQRT2</td>
<td>1.41421356237309504880</td>
<td>$\sqrt{2}$</td>
</tr>
<tr>
<td>M_SQRT1_2</td>
<td>0.70710678118654752440</td>
<td>$1/\sqrt{2}$</td>
</tr>
</tbody>
</table>
• 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 \(\Rightarrow y \geq 0\)

  \[
  \text{pow}(-2.0, 3.0); \quad -2^3 = -2 \times -2 \times -2 = -8
  \]

  \[
  \text{pow}(-2.0, 3.1); \quad -2^{3.1} = 10^{3.1 \times \log -2.0} = \text{nan (not a number)}
  \]

  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
  ```
## 2.5 Control Structures

<table>
<thead>
<tr>
<th>Java</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>block</strong></td>
<td></td>
</tr>
<tr>
<td><code>{ intermixed decls/stmts }</code></td>
<td><code>{ intermixed decls/stmts }</code></td>
</tr>
<tr>
<td><strong>selection</strong></td>
<td></td>
</tr>
<tr>
<td>if ( bool-expr1 ) stmt1 else if ( bool-expr2 ) stmt2 ... else stmtN</td>
<td>if ( bool-expr1 ) stmt1 else if ( bool-expr2 ) stmt2 ... else stmtN</td>
</tr>
<tr>
<td>switch ( integral-expr ) {</td>
<td></td>
</tr>
<tr>
<td>case c1: stmts1; break;</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>case cN: stmtsN; break;</td>
<td></td>
</tr>
<tr>
<td>default: stmts0;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td><strong>looping</strong></td>
<td></td>
</tr>
<tr>
<td>while ( bool-expr ) stmt</td>
<td>while ( bool-expr ) stmt</td>
</tr>
<tr>
<td>do stmt while ( bool-expr ) ;</td>
<td>do stmt while ( bool-expr ) ;</td>
</tr>
<tr>
<td>for (init-expr;bool-expr;incr-expr) stmt</td>
<td>for (init-expr;bool-expr;incr-expr) stmt</td>
</tr>
<tr>
<td><strong>transfer</strong></td>
<td></td>
</tr>
<tr>
<td>break [ label ]</td>
<td>break</td>
</tr>
<tr>
<td>continue [ label ]</td>
<td>continue</td>
</tr>
<tr>
<td>return [ expr ]</td>
<td>return [ expr ]</td>
</tr>
<tr>
<td>throw [ expr ]</td>
<td>throw [ expr ]</td>
</tr>
<tr>
<td>label</td>
<td>label : stmt</td>
</tr>
<tr>
<td>label</td>
<td>label : stmt</td>
</tr>
</tbody>
</table>
2.5.1 Block

- **Block** is a series of statements bracketed by braces, `{ . . . }`, which can be nested.
- A block forms a complete statement but does not have to be terminated with a semicolon.
- Block serves two purposes: bracket several statements into a single statement and introduce local declarations.
- Good practice is to always use a block versus single statement to allow easy insertion and removal of statements to or from block.

```plaintext
if ( x > y ) // no block
    x = 0; // cannot directly add statements

if ( x > y ) { // block
    x = 0; // can directly add/remove statements
}
```

Does the shell have this problem?

- Declarations may be intermixed among executable statements in a block.
- Block variables are allocated first-in first-out (FIFO) from the stack memory area.
Localizing declarations in nested blocks reduces declaration clutter at beginning of a block.

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

However, it can also make locating declarations more difficult.

- Variable names can be reused in different blocks, i.e., possibly overriding (hiding) prior variables.
```c
int i = 1; ... // first i
{
    int k = i, i = 2;... // k = first i, second i overrides first
    {
        int i = 3;... // third i (overrides second)
}
```

### 2.5.2 Selection

- C/C++ selection statements are **if** and **switch** (same as Java).
- An **if** statement selectively executes one of two alternatives based on a comparison result:
  ```c
  if ( x > y ) max = x;
  else max = y;
  ```
- 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

<table>
<thead>
<tr>
<th>Dangling Else</th>
<th>Fix Using Null Else</th>
<th>Fix Using Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( sales &lt; 10000 )&lt;br&gt;  if ( sales &lt; 5000 )&lt;br&gt; income -= penalty;&lt;br&gt; else // incorrect match!!!&lt;br&gt; income += bonus;</td>
<td>if ( sales &lt; 10000 )&lt;br&gt;  if ( sales &lt; 5000 )&lt;br&gt; income -= penalty;&lt;br&gt; else ; // null statement&lt;br&gt; else&lt;br&gt; income += bonus;</td>
<td>if ( sales &lt; 10000 ) {&lt;br&gt;  if ( sales &lt; 5000 )&lt;br&gt; income -= penalty;&lt;br&gt; } else&lt;br&gt; income += bonus;</td>
</tr>
</tbody>
</table>

- **Unnecessary equality for boolean as value is already true or false.**
  
  ```cpp
  bool b;
  if ( b == true ) ... // if ( b )
  ```

- **Conversion causes problems.**
  
  ```cpp
  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).**
  
  ```cpp
  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:
```c
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.
• 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.

2.5.3 Conditional Expression Evaluation
• Conditional expression evaluation performs partial (short-circuit) expression evaluation.

<table>
<thead>
<tr>
<th>&amp; &amp;</th>
<th>only evaluates the right operand if the left operand is true</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>only evaluates the right operand if the left operand is false</td>
</tr>
<tr>
<td>? :</td>
<td>only evaluates one of two alternative parts of an expression</td>
</tr>
</tbody>
</table>

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

    \[ d \neq 0 \quad \& \& \quad n / d > 5 \quad // \text{may not evaluate right operand} \Rightarrow \text{no division by 0} \]
false and anything is?

- Hence, short-circuit operators are control structures in the middle of an expression because \( e_1 \land e_2 \neq \land(e_1, e_2) \) (unless lazy evaluation).
- Logical & and | evaluate operands eagerly, evaluating both operands.
- Conditional ?: evaluates one of two expressions, and returns the result of the evaluated expression.
- Acts like an if statement in an expression and can eliminate temporary variables.

\[
\text{f}( ( a < 0 \ ? \ -a \ : \ a ) + 2 );
\]

\[
\text{int temp;}
\text{if ( a < 0 ) temp = -a;}
\text{else temp = a;}
\text{f( temp + 2 );}
\]

2.5.4 Looping

- C/C++ looping statements are while, do and for (same as Java).
- while statement executes its statement zero or more times.
while ( x < 5 ) {
    ... // executes 0 or more times
}

- **Beware of accidental infinite loops.**

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

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

```plaintext
do {
    ... // executes one or more times
} while ( x < 5 );
```

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

```plaintext
init-expr;
while ( bool-expr ) {
    for ( init-expr; bool-expr; incr-expr ) {
        stmts;
    }
    incr-expr;
}
```

- If **init-expr** is a declaration, the scope of its variables is the remainder of the
declaration, the other two expressions, and the loop body.

```c
for ( int i = 0, j = i; i < j; i += 1 ) { // i and j declared
    // i and j visible
} // i and j deallocated and invisible
```

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

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

for ( i = 10; 1 <= i; i -= 1 ) { // count down
    // loop 10 times
} // i has value 0 on exit

for ( p = s; p != NULL; p = p->link ) { // pointer index
    // loop through list structure
} // p has the value NULL on exit

for ( i=1, p=s; i <= 10 && p != NULL; i += 1, p=p->link ) { // 2 indices
    // loop until 10th node or end of list encountered
}
```

- Comma expression is used to initialize and increment 2 indices in a context where normally only a single expression is allowed.
- Default **true** value inserted if no conditional is specified in **for** statement.
  
  ```java
  for ( ; ; )  // rewritten as: for ( ; true ; )
  ```

- **break** statement terminates enclosing loop body.

- **continue** statement advances to the next loop iteration.

### 2.6 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).

- Developed during late 1960’s to early 1970’s to overcome the indiscriminant use of the GOTO statement.

- GOTO leads to convoluted logic in programs (i.e., does NOT support a methodical thought process).
• I.e., arbitrary transfer of control makes programs difficult to understand and maintain.

• Restricted transfer reduces the points where flow of control changes, and therefore, is easy to understand.

• There are 3 levels of structured programming:
  
  **classical**
  
  - sequence: series of statements
  - if-then-else: conditional structure for making decisions
  - while: structure for loops with test at top
  
  Can write any program (actually only need **whiles** or one **while** and **ifs**).

  **extended**
  
  - use the classical control-structures and add:
    * case/switch: conditional structure for making decisions
    * for: while with initialization/test/increment
    * repeat-until/do-while: structure for loops with test at bottom

  **modified**
  
  - use the extended control-structures and add:
    * one or more exits from arbitrary points in a loop
* exits from multiple nested control structures
* exits from multiple nested routine calls

### 2.6.1 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):
  ```ada
  while i < 10 do
  loop
    exit when i >= 10; -- loop exit
  end loop
  ```
- Repeat-loop has 1 exit located at the bottom:
  ```ada
  do
  loop
    exit when i >= 10; -- loop exit
  while ( i < 10 )
  end loop
  ```
- 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:

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

• **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):
A multi-exit loop can be written in C/C++ in the following ways:

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

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

- Allow multiple exit conditions:


```cpp
bool flag1 = false, flag2 = false;

for ( ;; ) {
    S1
    if ( i >= 10 ) break;
    S2
    if ( j >= 10 ) break;
    S3
}

while ( ! flag1 & ! flag2 ) {
    S1
    if ( C1 ) flag1 = true;
    S2
    if ( C2 ) flag2 = true;
    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.
- 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;  // rewrite: -(i<size-1, !found)
bool found = false;

while ( 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 its associated list counter is incremented.

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

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

2.6.2 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
        }
    ... }
...
L1: }
```

• 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 from the goto to statement marked with label variable.

• Label variable is declared by prefixing an identifier with a “:” to a statement.
L1: i += 1;  // associated with expression
L2: if ( ... ) ...;  // associated with if statement
L3: ;  // associated with empty statement

• Labels can only be declared in a routine, **where the label has routine scope**.
  ○ i.e., label identifier is unique within a routine body ⇒ cannot be overridden in local blocks.

```c
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.

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

• Why is it good practice to associate a label with an empty statement?
• 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 a good idea to put label on its own 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
        ...
    } // for
    if ( ... ) goto B1; // outdent
    ...
    // rest of loop
} // for

B2: ;

B1: ;

• 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?

• Other uses of multi-level exit to remove duplicate code:

<table>
<thead>
<tr>
<th>duplication</th>
<th>no duplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( C1 ) {</td>
<td>if ( C1 ) {</td>
</tr>
<tr>
<td>S1;</td>
<td>S1;</td>
</tr>
<tr>
<td>if ( C2 ) {</td>
<td>if ( C2 ) {</td>
</tr>
<tr>
<td>S2;</td>
<td>S2;</td>
</tr>
<tr>
<td>if ( C3 ) {</td>
<td>if ( C3 ) {</td>
</tr>
<tr>
<td>S3;</td>
<td>S3;</td>
</tr>
<tr>
<td>} else</td>
<td>} else</td>
</tr>
<tr>
<td>S4;</td>
<td>}</td>
</tr>
<tr>
<td>} else</td>
<td>}</td>
</tr>
<tr>
<td>S4;</td>
<td>S4; // only once</td>
</tr>
</tbody>
</table>

```c
C: ;
```
• If any conditions are false, the same code is executed (e.g., printing an error message), resulting in code duplication.

• 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.

2.7 Type Constructor

• Type constructor declaration builds more complex type from basic types.
2.7.1 Enumeration

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

```java
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 **enumerators**.
- 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.

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?

2.7.2 Pointer/Reference

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

- Used to access the value stored in the memory location at the pointer address.

- **All** variables have an address in memory, e.g., `int x = 5, y = 7:`
Two basic addressing operations:

1. **referencing**: obtain address of a variable; unary operator & in C++:
   
   100 ← &x
   200 ← &y

2. **dereferencing**: retrieve value at an address; unary operator * in C++:
   
   5 ← *(100) ← *(&x)
   7 ← *(200) ← *(&y)

Note, unary and binary use of operators &/* for reference/dereference and conjunction/multiplication.

So what does a variable name mean? For x, is it 5 or 100? It depends!

A variable name is a symbolic name for the pointer to its value, e.g., x means &x, i.e., symbol x is always replaced by pointer value 100.

What happens in this expression so it can execute?
x = x + 1;

- First, each variable name is substituted (rewritten) for its pointer value:

\[(&x) = (&x) + 1\] where \(x \equiv &x\)
\n\[(100) = (100) + 1\]

Assign into memory location 100 the value 101? Only partially correct!

- Second, when a variable name appears on the right-hand side of assignment, it implies the variable’s value not its address.

\[(&x) = *(&x) + 1\]
\n\[(100) = *(100) + 1\]
\n\[(100) = 5 + 1\]

Assign into memory location 100 the value 6? Correct!

- Hence, a variable name always means its address, and a variable name is also implicitly dereferenced on the right side of assignment.

- Exception is &x, which just means \&x not \&(&x).

- Notice, identifier x (in a particular scope) is a literal (const) pointer because it always means the same memory address (e.g., 100).

- Generalize notion of literal variable-name to variable name that can point to more than one memory location (like integer variable versus literal).
• A **pointer variable** is a (non-**const**) variable that contains different variable addresses **restricted to a specific type** in any storage location (i.e., static, stack or heap storage).

  ○ Java references can only address **object types** on the **heap**.

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

```
<table>
<thead>
<tr>
<th>p1</th>
<th>100</th>
<th>5</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>30</td>
<td></td>
<td>40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p2</th>
<th>200</th>
<th>7</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>40</td>
<td></td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p3</th>
<th>0 / 0x34fe7</th>
<th></th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>null/undefined pointer</td>
<td></td>
</tr>
</tbody>
</table>
```

• Storage is needed for different address values, so a pointer variable also has an address!

• By convention, no variable is placed at the **null address** (pointer), null in Java, 0 in C/C++.
- Hence, an address value is another variable’s address (indirection) or null address or an undefined address when uninitialized.
  - null address often means pointer is unused.
- Multiple pointers may point to the same memory address ($p2 = p1$, dashed line).
- Dereferencing null/undefined pointer is undefined as no variable at address (but not error).
- Variable pointed-at is the target variable and its value is the target value.
  - e.g., $x$ is the target variable of $p1$ with target value 5.
- Can a pointer variable point to itself?
- Same implicit reference/dereference rules apply for pointer variables.

\[
p1 = \&x; \quad \text{// pointer assignment}
\]
\[
(\&p1) \leftarrow \&x \quad \text{// no rewrite rule for x, why?}
\]
\[
(30) \leftarrow 100
\]

Assign to memory location 30 the value 100.
p2 = p1;          // pointer assignment
(&p2) ← *(&p1)   // rewrite rules
(40) ← *(30)
(40) ← 100

Assign to memory location 40 the value 100.

• Value assignment requires explicit dereferencing to access values:

  *p2 = *p1;          // value assignment, y = x
  *(&p2) ← *(*(&p1)) // rewrite rules
  *(40) ← *(30))
  200 ← *(100)
  200 ← 5

Assign to memory location 200 the value 5.

• Often the target value is used more than the pointer value.

  *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:
```c
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, extra implicit dereference generates a problem for pointer assignment.
    ```c
    r2 = r1;
    *(&r2) ← *(*(&r1)) // value 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.
    ```c
    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.

```java
int i;
int &r = i;
&r;       *(&r) ⇒ &i not &r
```

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

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

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

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

• Type qualifiers can be used to modify pointer types.
const short int \( w = 25; \)
const short int \(*p4 = \&w;\)

int \(*\text{ const} \) \( p5 = \&x; \)
int \&p5 = x;

const long int \( z = 37; \)
const long int \(*\text{ const} \) \( p6 = \&z; \)

- \( p4 \) may point at \textit{any} short int variable (\texttt{const} or non-\texttt{const}) and may not change its value.

Why can \( p4 \) point to a non-\texttt{const} variable?

- \( p5 \) may only point at the int variable \( x \) and may change the value of \( x \) through the pointer.
  - \(*\text{ const} \) and \& are literal pointers but \(*\text{ const} \) 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.
The diagram shows the variables and their values:

- `px` points to 100, `ppx` points to 108, `rx` points to 116, and `prx` points to 100.

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

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

### C++ idiom for declaring pointers/references is misleading; only works for single versus list of variables.

- **int** `* i;`  
  `double` `& x = d;`
- **int** `* i, k;`  
  `double` `& x = d, y = d;`

Gives false impression of distribution across the identifier list.
2.7.3 Aggregates

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

2.7.3.1 Array

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

```c
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.

```c
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).

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

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

- Array values (elements) are accessed by **subscripts**, “[]” (look like dimensions).

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

```cpp
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
```


- An array name without a subscript means the first element.

```cpp
array => array[0]
matrix => matrix[0][0]
cube => cube[0][0][0]
```
• C/C++ array is a contiguous set of elements not a reference to the element set as in Java.

<table>
<thead>
<tr>
<th>Java</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x[6] = 1 7 5 0 8 -1</code></td>
<td><code>x[6] = 1 7 5 0 8 -1</code></td>
</tr>
</tbody>
</table>

• **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++.

• Because no array-size information, the dimension value for an array pointer is unspecified.

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

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

```c
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.

```c
*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,
  ○ read inside parenthesis outwards,
  ○ start with variable name,
  ○ end with type name on the left.

```c
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

### 2.7.3.2 Structure
• **Structure** is a set of **heterogeneous values**, including (nested) structures.

<table>
<thead>
<tr>
<th>Java</th>
<th>C/C++</th>
</tr>
</thead>
</table>
| ```
    class Foo {
        int i = 3;
        ... // more fields
    }
``` | ```
    struct Foo {
        int i; // no initialization
        ... // more members
    }; // semi-colon terminated
``` |

• Components of a structure are called **members** subdivided into data and routine/function members\(^1\) in C++.

• All members of a structure are accessible (public) by default.

• A structure member cannot be directly initialized (unlike Java).

• A structure is terminated with a *semicolon*.

• 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
  ```

---

\(^1\)Java subdivides members into fields (data) and methods (routines).
• Structures with the same type can be assigned but not compared.

```c
struct Student {
    struct Name { // nested structure
        char first[20]; // array
        char last[20]; // array
    } name;
    double age;
    int marks[10]; // array
}
```

`s1, s2, *sp1 = &s1;
s1 = s2; // allowed
s1 == s2; // disallowed, no structure relational operations
```

Notice, arrays in the structures are copied, but there is no array copy. How?

• Structure members are accessed by **member selection**, using selection operator “.” (like Java).

```
s1.name.first[0] = 'a';
s1.age = 34;
s1.marks[3] = 99;
```

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

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

\[
\begin{align*}
(*sp1).\text{name}.\text{first}[0] &= \text{'}a\text{'}; \\
(*sp1).\text{age} &= 34; \\
(*sp1).\text{marks}[5] &= 95;
\end{align*}
\]

Alternatively, use (special) selection operator \(ightarrow\) for pointers to structures:

- performs dereference and member selection in correct order, i.e., \(p\rightarrow f\) is rewritten as \((*p).f\).

\[
\begin{align*}
\text{sp1}\rightarrow\text{name}.\text{first}[0] &= \text{'}a\text{'}; \\
\text{sp1}\rightarrow\text{age} &= 34; \\
\text{sp1}\rightarrow\text{marks}[5] &= 95;
\end{align*}
\]

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

Structures **must** be compared member by member.

- comparing bits (e.g., `memcmp`) fails as alignment padding leaves undefined values between members.

**Recursive types** (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:

```c
struct S {
    int i : 3; // 3 bits
    int j : 7; // 7 bits
    int k : 6; // 6 bits
} s;
```

- 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.
2.7.3.3 Union

- **Union** is a set of *heterogeneous values*, including (nested) structures, where all members overlay the same storage.

```c
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 << ' ' << u.s.exp << ' ' << u.s.frac << endl;

u.f = -3.5e-3; cout << u.s.sign << ' ' << u.s.exp << ' ' << u.s.frac << endl;

u.f = 3.5;     cout << u.f << ' ' << hex << u.i << endl;

u.i = 3;       cout << u.i << ' ' << u.f << endl;

produces:

0  8a 5ac000 internal structure of 3.5e3
1  76 656042 internal structure 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.**
2.7.4 String

- A **string** is a sequence of characters with specialized operations to manipulate the sequence.

- Strings are provided in C by an array of **char**, string literals, and library facilities.

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

- String literal is a double-quoted sequence of characters.

  ```c
  "abc"
  "a b c"
  ```

- Pointer to a string literal must be **const**.

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

  Why?

- Juxtaposed string literals are concatenated.

  ```c
  const char *n1 = "johndoe";
  const char *n2 = "john" "doe"; // divide literal for readability
  ```

- Character escape sequences may appear in string literal.
Sequence of octal digits is terminated by length (3) or first character not an octal digit; sequence of hex digits is arbitrarily long, but value truncated to fit character type.

How many characters?

Techniques for preventing escape ambiguity.
  o Octal escape can be written with 3 digits.
    "\01234"
  o Octal/hex escape can be written as concatenated strings.
    "\12" "34" "\xa" "abc" "\x12" "34"

Every string literal is implicitly terminated with a character '\0'.
  o e.g., string literal "abc" is actually 4 characters: 'a', 'b', 'c', and '\0', which occupies 4 bytes of storage.

Zero value is a **sentinel** used by C-string routines to locate the string end.

Drawbacks:
A string cannot contain a character with the value \'\0\'.
To find string length, must linearly search for \'\0\', which is expensive for long strings.

- 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 using a length member and managing all of the storage for the variable-sized strings (#include <string>).

- Set of powerful operations that perform actions on groups of characters.
<table>
<thead>
<tr>
<th>Java String</th>
<th>C char []</th>
<th>C++ string</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, concat</td>
<td>strcpy, strncpy</td>
<td>=</td>
</tr>
<tr>
<td>equal, compareTo</td>
<td>strcat, strncat</td>
<td>+</td>
</tr>
<tr>
<td>length</td>
<td>strcmp, strncmp</td>
<td>==, !=, &lt;, &lt;=, &gt;, &gt;=</td>
</tr>
<tr>
<td>charAt</td>
<td>strlen</td>
<td>length</td>
</tr>
<tr>
<td>substring</td>
<td>strstr</td>
<td>substr</td>
</tr>
<tr>
<td>replace</td>
<td></td>
<td>replace</td>
</tr>
<tr>
<td>indexOf, lastIndexOf</td>
<td>strcspn, strspn</td>
<td>find, rfind</td>
</tr>
</tbody>
</table>

- All of the C++ string find members return values of type string::size_type and value string::npos if a search is unsuccessful.
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, lexicographical 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).
- Member c_str converts a string to a char * pointer ( ˈ \0 ˈ terminated).
- Count and print words in string-variable line containing words separated by whitespace.
unsigned int count = 0;
string line, alpha = "abcdefghijklmnopqrstuvwxyz"
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
...
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
<table>
<thead>
<tr>
<th>Line</th>
<th>The quick brown</th>
<th>0 1 2 3 4 5 6 7 8 9 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The quick brown</td>
<td>quick brown</td>
</tr>
<tr>
<td></td>
<td>quick brown</td>
<td>brown</td>
</tr>
<tr>
<td></td>
<td>brown</td>
<td>brown</td>
</tr>
<tr>
<td></td>
<td>brown</td>
<td>npos</td>
</tr>
</tbody>
</table>

```python
\n```
It is seldom necessary to iterate through the characters of a string variable!

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

```cpp
#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?
2.7.5 Type Equivalence

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

```c
struct T1 {
    int i, j, k;
    double x, y, z;
};

structure T2 {
    int i, j, k;
    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 a 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”.

- Java provides no mechanism to alias types.

2.7.6 Type Nesting

- Type nesting is useful for organizing and controlling visibility for type names:
enum Colour { R, G, B, Y, C, M }

struct Foo {
    enum Colour { R, G, B };    // nested type
    struct Bar {
        Colour c[5];            // type defined outside (1 level)
    };
    ::Colour c[5];            // type defined outside (top level)
    Colour cc;                // type defined same level
    Bar bars[10];             // type defined same level
};

Colour c1 = R;                // type/enum defined same level
Foo::Colour c2 = Foo::R;      // type/enum defined inside
Foo::Bar bar;                 // type defined inside

• 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).

• References to types nested inside another type must be qualified with type operator "::".

• With nested types, Colour (and its enumerators) and Foo in top-level scope; without nested types need:
enum Colour { R, G, B, Y, C, M };  
enum Colour2 { R2, G2, B2 };  // prevent name clashes
struct Bar {
    Colour2 c[5];
};
struct Foo {
    Colour c[5];
    Colour2 cc;
    Bar bars[10];
};
Colour c1 = R;
Colour2 c2 = R2;
Bar bar;

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

2.7.7 Type-Constructor Literal

<table>
<thead>
<tr>
<th>enumeration</th>
<th>enumerators</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer</td>
<td>0 or NULL indicates a null pointer</td>
</tr>
<tr>
<td>structure</td>
<td>struct { double r, i; } c = { 3.0, 2.1 };</td>
</tr>
<tr>
<td>array</td>
<td>int v[3] = { 1, 2, 3 };</td>
</tr>
</tbody>
</table>

- C/C++ use 0 to initialize pointers (Java null).
• System include-file defines the preprocessor variable NULL as 0.

• 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**, which means zero-filled for basic types.

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

• g++ has a cast extension allowing construction of structure and array literals in executable statements not just declarations:

    void rtn( const int m[2][3] );
    struct Complex { double r, i; } c;
    rtn( (int [2][3]){ {93, 67, 72}, {77, 81, 86} } );  // g++ only
    c = (Complex){ 2.1, 3.4 };  // g++ only
• In both cases, a cast indicates the type and structure of the literal.
• String literals can be used as a shorthand array initializer value:

```c
char s[6] = "abcde";  // 1st dimension inferred as 6 (Why 6?)
```  
```c
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
```

2.8 Modularization

• **Modularization** is the division of a system into interconnecting parts (components), using some systematic basis, and is the foundation of software engineering.

• Medium and large systems must be modularized so people can understand them.

• Essentially any contiguous block of code or group of variables can be factored into a routine or structure and given a name (or vice versa).

• **Module**
− separates usage from implementation by enforcing logical boundaries among components,
− eliminates duplicate code by factoring common code/declarations into single location.

• **Interfaces** provide the boundaries through various programming-language mechanisms.

• Hence, modularization provides a mechanism to abstract algorithms and data structures through interfaces.

### 2.9 Routine

• Like algebra, arbitrary operations can be define and invoked, e.g., 
  \[ f(x) = 3x^2 + 2.5x - 17, \text{ where } f(4.5) = 55. \]

  ```c
  double f( double x ) { return 3.0 * x * x + 2.5 * x - 17.0; }
  f( 4.5 );  // returns 55
  ```

• A **routine** is the simplest module for factoring code (routine/procedural abstraction).

• Input and output parameters define a routine’s interface.
**Routine is either a **procedure** or a **function** based on the return type.**

**Procedure does NOT return a value that can be use in an expression, indicated with return type of **void**:  

```
void usage() {
    cout << "Usage: " << ... << endl; // some usage message
    exit( EXIT_FAILURE ); // TERMINATE PROGRAM
}
```

**Procedure can return values via the argument/parameter mechanism.**

**Procedure terminates when control runs off the end of its routine body or a return statement is executed:**
```c
void proc() {
    ... return; ...
    ... // run off end => return
}
```

- Function returns a value that can be used in an expression, and hence, **must** execute a `return` statement specifying a value:

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

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

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

  ```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`.

• 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 an implicit `static block`.

```c
int i;                     // static block, global
const double PI = 3.14159;
int rtn( double d )       // code block
{   ... return 4;        // create stack block
}
// remove stack block
int main()                // code block
{   int j;               // create stack block
    { int k;            // create stack block
        rtn( 3.0 );    // create stack block
    }                    // 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*.
- Good practise is to ONLY use static block for literals/variables accessed throughout program.

### 2.9.1 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).

<table>
<thead>
<tr>
<th>pass by value</th>
<th>pass by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>argument</strong></td>
<td>5</td>
</tr>
<tr>
<td><strong>parameter</strong></td>
<td>5</td>
</tr>
</tbody>
</table>

- Java/C, parameter passing is by value, i.e., basic types and object references are copied.
- C++, parameter passing is by value or reference depending on the type of the parameter.
- Argument expressions are evaluated *in any order*.
- For value parameters, each argument-expression result is 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.

```c
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
}
```

<table>
<thead>
<tr>
<th></th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>1.0</td>
<td>2.0</td>
<td>7.5</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>200</td>
<td>300</td>
</tr>
<tr>
<td>parameter</td>
<td>1.0</td>
<td>200</td>
<td>300</td>
</tr>
<tr>
<td></td>
<td>s</td>
<td>rs</td>
<td>ps</td>
</tr>
<tr>
<td>call</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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
• 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!

```c
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:

```c
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 [5]){ 3, 2, 7, 9, 0 } );
```

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

```cpp
void r3(int i, double g, char c = '*', double h = 3.5) {
    ...}
```

```cpp
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.

### 2.9.2 Array Parameter

• Array copy is unsupported 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 the middle form as it clearly indicates the variable can be subscripted.
- An actual declaration cannot use []; it must use *:

  double sum( double v[] ) {
      double *cv;
      cv = v;
  } // actual declaration, think cv[]

  double *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;
  }

2.10 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.
## 2.10.1 Formatted I/O

<table>
<thead>
<tr>
<th>Java</th>
<th>C</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>import java.io.*;</td>
<td>#include &lt;stdio.h&gt;</td>
<td>#include &lt;iostream&gt;</td>
</tr>
<tr>
<td>import java.util.Scanner;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>File, Scanner, PrintStream</td>
<td>FILE</td>
<td>ifstream, ofstream</td>
</tr>
<tr>
<td>Scanner in = new</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scanner( new File( &quot;f&quot; ) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PrintStream out = new</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PrintStream( &quot;g&quot; )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>in.close()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>out.close()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nextInt()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nextFloat()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nextByte()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>next()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hasNext()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hasNextT()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>skip( &quot;regexp&quot; )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>out.print( String )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Formatted I/O occurs to/from a stream file, and values are converted based
• Both I/O libraries can cascade multiple I/O operations, i.e., input or output multiple values in a single expression.

2.10.1.1 Formats

• Format of input/output values is controlled via manipulators defined in \#include `<iomanip>`.

<table>
<thead>
<tr>
<th>manipulator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oct</td>
<td>integral values in octal</td>
</tr>
<tr>
<td>dec</td>
<td>integral values in decimal</td>
</tr>
<tr>
<td>hex</td>
<td>integral values in hexadecimal</td>
</tr>
<tr>
<td>left / right</td>
<td>values with padding after / before values</td>
</tr>
<tr>
<td>boolalpha / noboolalpha</td>
<td>bool values as false/true instead of 0/1</td>
</tr>
<tr>
<td>showbase / noshowbase</td>
<td>values with / without prefix 0 for octal &amp; 0x for hex</td>
</tr>
<tr>
<td>showpoint / noshowpoint</td>
<td>print decimal point if no fraction</td>
</tr>
<tr>
<td>fixed / scientific</td>
<td>float-point values without / with exponent</td>
</tr>
<tr>
<td>setprecision(N)</td>
<td>fraction of float-point values in maximum of N columns</td>
</tr>
<tr>
<td>setfill(‘ch’)</td>
<td>padding character before/after value (default blank)</td>
</tr>
<tr>
<td>setw(N)</td>
<td>NEXT VALUE ONLY in minimum of N columns</td>
</tr>
<tr>
<td>endl</td>
<td>flush output buffer and start new line (output only)</td>
</tr>
<tr>
<td>skipws / noskipws</td>
<td>skip whitespace characters (input only)</td>
</tr>
</tbody>
</table>
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.

During input, `skipws/noskipws` toggle between ignoring whitespace between input tokens and reading the whitespace characters (i.e., tokenize versus raw input).

### 2.10.1.2 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`).
<table>
<thead>
<tr>
<th>Java</th>
<th>C</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>import java.io.*;</code></td>
<td><code>#include &lt;stdio.h&gt;</code></td>
<td><code>#include &lt;fstream&gt;</code></td>
</tr>
<tr>
<td><code>import java.util.Scanner;</code></td>
<td><code>FILE *in = fopen(&quot;f&quot;, &quot;r&quot;);</code></td>
<td><code>ifstream in( &quot;f&quot; );</code></td>
</tr>
<tr>
<td>Scanner in =</td>
<td>FILE *out = fopen(&quot;g&quot;, &quot;w&quot;);</td>
<td>ofstream out( &quot;g&quot; );</td>
</tr>
<tr>
<td><code>new Scanner(new File(&quot;f&quot;)).</code></td>
<td><code>int i, j;</code></td>
<td><code>int i, j;</code></td>
</tr>
<tr>
<td>PrintStream out =</td>
<td><code>for ( ; ; ) {</code></td>
<td><code>for ( ; ; ) {</code></td>
</tr>
<tr>
<td><code>new PrintStream(&quot;g&quot;);</code></td>
<td><code>fscanf(in, &quot;%d%d&quot;, &amp;i, &amp;j);</code></td>
<td><code>in &gt;&gt; i &gt;&gt; j;</code></td>
</tr>
<tr>
<td><code>int i, j;</code></td>
<td><code>if ( feof(in) ) break;</code></td>
<td><code>if ( in.fail() ) break;</code></td>
</tr>
<tr>
<td><code>while ( in.hasNext() ) {</code></td>
<td><code>fprintf(out, &quot;i:%d j:%d\n&quot;, i, j);</code></td>
<td><code>out &lt;&lt; &quot;i:&quot; &lt;&lt; i &lt;&lt; &quot;j:&quot; &lt;&lt; j &lt;&lt; endl;</code></td>
</tr>
<tr>
<td>i = in.nextInt(); j = in.nextInt();</td>
<td><code>}</code></td>
<td><code>// in/out closed implicitly</code></td>
</tr>
<tr>
<td>out.println(&quot;i:&quot; + i + &quot; j:&quot; + j);</td>
<td><code>}</code></td>
<td></td>
</tr>
<tr>
<td>in.close();</td>
<td><code>}</code></td>
<td></td>
</tr>
<tr>
<td>out.close();</td>
<td><code>}</code></td>
<td></td>
</tr>
</tbody>
</table>

- 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.
• To read strings containing white spaces use routine getline( stream, string, char ), which allows different delimiting characters on input:

```c++
    string s;
    getline( cin, s, ' ' ); // read characters until ' ' => cin >> c
    getline( cin, s, '@' ); // read characters until '@'
    getline( cin, s, '\n' ); // read characters until newline (default)
```
• 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.
• From a shell, typing <ctrl>-d (C-d), i.e., press <ctrl> and d keys simultaneously, causes the shell to close the 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.

```cpp
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.
• Streams also have coercion to `void *`: if `fail()`, null pointer; otherwise non-null pointer.

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

• When bad data is read, `stream must be reset and bad data cleared`:
#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(),'
'); // skip until newline
        } else {
            cout << hex << "hex: " << n << dec << " dec: " << n << endl;
        }
    }
    cout << endl;
}

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

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

```cpp
#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, so arguments to `fscanf` must be preceded with `&` (except arrays) so they can be changed.
2.10.1.3 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:
2.10.2 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 castor C++ reinterpret_cast.
```cpp
#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( reinterpret_cast<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 would be unnecessary if buffer type was void *.
```
2.11 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.
<table>
<thead>
<tr>
<th>Java</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class Prog {</code></td>
<td><code>int main( int argc, char *argv[] ) {</code></td>
</tr>
<tr>
<td><code>    public static void main( String[] args ) {</code></td>
<td><code>    switch( argc ) {</code></td>
</tr>
<tr>
<td><code>        switch ( args.length ) {</code></td>
<td><code>        case 1: ...  // no args</code></td>
</tr>
<tr>
<td><code>            case 0: ...  // no args</code></td>
<td><code>            break;</code></td>
</tr>
<tr>
<td><code>                case 1: ... args[0] ... // 1 arg</code></td>
<td><code>            case 2: ... args[1] ... // 1 arg</code></td>
</tr>
<tr>
<td><code>                break;</code></td>
<td><code>                break;</code></td>
</tr>
<tr>
<td><code>            case ... // others args</code></td>
<td><code>            case ... // others args</code></td>
</tr>
<tr>
<td><code>                break;</code></td>
<td><code>                break;</code></td>
</tr>
<tr>
<td><code>            default: ...  // usage message</code></td>
<td><code>            default: ...  // usage message</code></td>
</tr>
<tr>
<td><code>                System.exit( 1 );</code></td>
<td><code>                exit( EXIT_FAILURE );</code></td>
</tr>
<tr>
<td><code>        }</code></td>
<td><code>    }</code></td>
</tr>
<tr>
<td><code>    ...</code></td>
<td><code>    ...</code></td>
</tr>
</tbody>
</table>

- 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, dynamic allocation, stringstream (atoi does not indicate errors), and no duplicate code.
```cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <cstdlib>

using namespace std; // direct access to std

bool convert( int &val, char *buffer ) { // convert C string to integer
    std::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 integer
        // FALL THROUGH
    case 2:
        if ( ! convert( size, argv[1] ) || size < 0 ) usage( argv ); // invalid integer ?
        // FALL THROUGH
    case 1:
        break; // all defaults
    default:
        usage( argv ); // wrong number of options
}
// 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
2.12 Preprocessor

- Preprocessor is a text editor that modifies the program text before compilation.

- **Program you see is not what the compiler sees!**

- A preprocessor statement starts with a `#` character, followed by a series of tokens separated by whitespace, which is usually a single line and not terminated by punctuation.

- The three most commonly used preprocessor facilities are substitution, file inclusion, and conditional inclusion.

2.12.1 Variables/Substitution

- `#define` statement declares a preprocessor string variable, and its value is all the text after the name up to the end of line.
```c
#define Integer int
#define begin {
#define end }
#define gets =
#define set
#define with =
Integer main() begin                     // transformed
    Integer x gets 3, y;                   
    x gets 5;                             
    set y with x;                         
end                                       
```

- Preprocessor can transform the syntax of C/C++ program (discouraged).
- Preprocessor variables can be defined and initialized on the compilation command with option `\-D`.

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

Initialization value is text after `\=`.
- Same as putting the following `\#defines` 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.

• Replace `define` with `enum` for integral types; otherwise use `const` declarations (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;
```

`enum` uses no storage while `const` declarations might.

• `define` can declare macros with parameters, which expand during compilation, textually substituting arguments for parameters, e.g.:

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

• Use `inline` routines in C/C++ rather that `define` macros.

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

- File inclusion copies text from a file into a C/C++ program.
- An included file may contain anything.
- An include file normally imports preprocessor and C/C++ templates/declarations for use in a program.
- All included text goes through every compilation step, i.e., preprocessor, compiler, etc.
- Java implicitly includes by matching class names with file names in CLASSPATH directories, then extracting and including declarations.
- The `#include` statement specifies the file to be included.
- C convention uses suffix “.h” for include files containing C declarations.
- C++ convention drops suffix “.h” for its standard libraries and has special file names for equivalent C files, e.g., `cstdio` versus `stdio.h`.

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

- A file name can be enclosed in `<>` or `""`. 
• <> means preprocessor only looks in the system include directories.

• """ means preprocessor starts looking for the file in the same directory as the file being compiled, then in the system include directories (/usr/include).

• System files limits.h (climit) and stddef.h (cstddef) contain many useful #defines.
  ○ e.g., null pointer literal NULL and min/max values for types (e.g., see /usr/include/limits.h).

2.12.3 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.
```c
#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.

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

- Possible to check if a preprocessor variable is defined or not defined using `#ifdef` or `#ifndef`:

```c
#include "debug1.h"
```
#ifndef __MYDEFS_H__    // if not defined
#define __MYDEFS_H__ 1   // make it so
...
#endif

• Used in an `#include` file to ensure its contents are only expanded once.

• Note difference between checking if a preprocessor variable is defined and checking the value of the variable.

• The former capability does not exist in most programming languages, i.e., checking if a variable is declared before trying to use it.

## 2.13 Assertions

• **Assertions** document program assumptions:
  
  o pre-conditions – things true before a computation (e.g., all values are positive),
  
  o invariants – things true across the computation (e.g., all values during the computation are positive, because only `+`, `*`, `/` operations),
  
  o post-conditions – things 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, please report", distance > 0 );

a.out: test.cc:19: unsigned int stopping_distance(Car):
           Assertion ("Internal error, please report", distance > 0)

- 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

2.14 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.

### 2.14.1 Debug Print Statements

- An excellent way to debug a program is to start by inserting debug print statements (i.e., as the program is written).
- It takes more time, but the alternative is wasting hours trying to figure out what the program is doing.
- The two aspects of a program that you need to know are: where the program is executing and what values it is calculating.
- Debug print statements show the flow of control through a program and print out intermediate values.
- E.g., every routine should have a debug print statement at the beginning and end, as in:
int p( ... ) {
    // declarations
    cerr << "Enter p " << parameter variables " << endl;
    ...
    cerr << "Exit p " << return value(s) " << endl;
    return r;
}

• Result is a high-level audit trail of where the program is executing and what values are being passed around.

• Finer resolution requires more debug print statements in important control structures:

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

• By examining the control paths taken and intermediate values generated, it is possible to determine if the program is executing correctly.

• Unfortunately, debug print statements can generate enormous amounts of output.

  *It is of the highest importance in the art of detection to be able to recognize out of a number of facts which are incidental and which vital.* *(Sherlock Holmes, The Reigate Squires)*

• Gradually comment out 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:

  ```cpp
  #define DPRT( title, expr ) \
  { std::cerr << #title "\t" << __PRETTY_FUNCTION__ " \ " << "\ " " << \. expr " in " << __FILE__ " at line " << __LINE__ " << std::endl; }
  ```

for printing entry, intermediate, and exit locations and data:
```cpp
#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;
}

which generates debug output:

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
```
2.14.2 Errors

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

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

- Difficult syntax errors are:
  - missing closing " or */, as the remainder of the program is swallowed as part of the character string or comment.
missing { or }, especially if the program is properly indented (editors can help here)
missing semi-colon at end of structure

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

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

- Reason from the particular (error symptoms) to the general (error cause).
locate pertinent data: categorize as correct or incorrect
look for contradictions
list possible causes
devise a hypothesis for the cause of the problem
use data to find contradictions to eliminate hypotheses
refine any remaining hypotheses
prove hypothesis is consistent with both correct and incorrect results, and accounts for all errors

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

```c
i = 10;
while (i != 5) {
    ... 
    i += 2;
}
```
The initialization is wrong.

- Difficult semantic errors are:
  - uninitialized variable
  - invalid subscript or pointer value
  - off-by-one error
• Finally, if a statement appears not to be working properly, but looks correct, check the syntax.

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

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

### 2.15 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

```java
class Foo { char c1, c2; }
Foo r = new Foo();
r.c1 = 'X';
// r garbage collected
```

### C

```c
struct Foo { char c1, c2; };
struct Foo *p = (struct Foo *) malloc( // allocate
    sizeof(struct Foo) // size
);
p->c1 = 'X';
free( p ); // explicit free
```

### C++

```cpp
struct Foo { char c1, c2; };
Foo *p = new Foo();
p->c1 = 'X';
delete p; // 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.
- Before storage can be used, it *must* be allocated.

```c
Foo *p;  // forget to initialize pointer with “new”
p->c1 = 'R';  // places ‘R’ at some random location in memory
```

Called an uninitialized variable.

- After storage is no longer needed it *must* be explicitly deleted.
Foo *p = new Foo;
p = new Foo;      // forgot to free previous storage
Called a **memory leak**.

- After storage is deleted, it **must** not be used:

```cpp
delete p;
p->c1 = 'R';          // result of dereference is undefined
```
Called a **dangling pointer**.

- 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:
### Java

```java
{ // basic & reference
    int i;
    double d;
    AggrType agr = new AggrType();
    ...
} // garbage collected
```

### C++

```cpp
{ // all types
    int i;
    double d;
    AggrType agr;
    ...
} // implicit delete
```

- 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 i;
    ... // use i
    ... // use *ip
    delete ip;
  }
  }
  ```

- **Dynamic allocation in C++ should be used only when a variable’s storage must outlive the block in which it is allocated.**
Type *rtn(...) {
    Type *tp = new Type;       // MUST USE HEAP
    ...
    // initialize/compute using tp
    return tp;                // storage outlives block
    // tp deleted later
}

- Declaration of a pointer to an array is complex in C/C++.
- 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:**

  ```c
  delete [] parr, pvar; // => (delete [] parr), pvar;
  ```
  which is an incorrect use of a comma expression; `pvar` is not deleted.
• 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”

2.16 Overloading

- Overloading occurs 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:

```cpp
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:

```cpp
int r( int i, int j ) { ... }  // overload name r three different ways
int r( double x, double y ) { ... }
int r( int k ) { ... }
```

```cpp
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:

  \[
  r( 1, (\text{int})2.0 ) \quad // \text{1st } r \\
  r( (\text{double})1, 2.0 ) \quad // \text{2nd } r
  \]

- 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 ); \quad // \text{int}
  r( ui ); \quad // \text{unsigned int}
  r( li ); \quad // \text{long int}
  ```

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

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

<table>
<thead>
<tr>
<th>Overloading</th>
<th>Default Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int r(int i, int j) { ... }</code></td>
<td><code>int r(int i, int j = 2) { ... }</code></td>
</tr>
<tr>
<td><code>int r(int i) { int j = 2; ... }</code></td>
<td><code>r(3); // 2nd r</code></td>
</tr>
<tr>
<td><code>r(3); // default argument of 2</code></td>
<td></td>
</tr>
</tbody>
</table>

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

### 2.17 Routine Pointer

- The flexibility and expressiveness of a routine comes from the argument/parameter mechanism, which generalizes a routine across any argument variables of matching type.
- However, the code within the routine is the same for all data in these variables.
- To generalize a routine further, code can be passed as an argument, which is executed within the routine body.
- Most programming languages allow a routine pointer for further generalization and reuse.
- Java only has routines contained in class definitions so routine pointers must be accomplished indirectly via classes.

- As for data parameters, routine pointers are specified with a type (return type, and number and types of parameters), and any routine matching this type can be passed as an argument:

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

- Routine f is generalized to accept any routine argument of the form: returns an `int` and takes an `int` parameter.

- Within the body of f, the parameter p is called with an appropriate `int` argument, and the result of calling p is further modified before it is returned.

- A routine pointer is passed as a constant reference in virtually all programming languages; in general, it makes no sense to change or copy routine code, like copying a data value.

- C/C++ require the programmer to explicitly specify the reference via a pointer, while other languages implicitly create a reference.
Two common uses of routine parameters are fix-up and call-back routines.

A **fix-up routine** is passed to another routine and called if an unusual situation is encountered during a computation.

E.g., a matrix is not invertible if its determinant is 0 (singular).

Rather than halt the program for a singular matrix, invert routine calls a user supplied fix-up routine to possibly recover and continue with a correction (e.g., modify the matrix):

```c
int singularDefault( int matrix[][10], int rows, int cols ) { abort(); }
int invert( int matrix[][10], int rows, int cols,
    int (*singular)(int matrix[][10], int rows, int cols) = singularDefault ) {
    ...
    if ( determinant( matrix, rows, cols ) == 0 ) {
        correction = singular( matrix, rows, cols ); // possible correction
    }
    ...
}
int fixup( int matrix[][10], int rows, int cols ) { return 0; }
invert( matrix, 10, 10, fixup ); // fixup rather than abort
```

A fix-up parameter generalizes a routine as the corrective action is specified for each call, and the action can be tailored to a particular usage.
• Giving the fix-up parameter a default value eliminates having to provide a fix-up argument.

• A **call-back routine** is used in event programming.

• When an event occurs, one or more call-back routines are called (triggered) and each one performs an action specific for that event.

• E.g., graphical user interface has interactive “widgets”: buttons, sliders and scrollbars.

• When a user manipulates the widget, events are generated representing the new state of the widget, e.g., button down or up.

• A program registers interest in transitions for different widgets by creating and registering a call-back routine.

```c
int closedown( /* info about event */ ) {
    // close down because close button press
    // return status of callback action
}
```

// inform when close button pressed for “widget”
registerCB( widget, closeButton, closedown );

• widget maintains list of registered callbacks.
• A widget calls specific call-back routine(s) when the widget changes state, passing new state of the widget to each call-back routine.

2.18 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:

<table>
<thead>
<tr>
<th>unorganized</th>
<th>organized</th>
</tr>
</thead>
<tbody>
<tr>
<td>int people_age[30];</td>
<td><strong>struct</strong> Person {</td>
</tr>
<tr>
<td>bool people_sex[30];</td>
<td>int age;</td>
</tr>
<tr>
<td>char people_name[30][50];</td>
<td>bool sex;</td>
</tr>
<tr>
<td></td>
<td>char name[50];</td>
</tr>
<tr>
<td></td>
<td>} people[30];</td>
</tr>
</tbody>
</table>

• 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.
- C++ does not subscribe to the Java notion that everything is either a basic type or an object, i.e., routines can exist without being embedded in a `struct/class`.

<table>
<thead>
<tr>
<th>structure form</th>
<th>object form</th>
</tr>
</thead>
</table>
| `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 | `struct` Complex {  
  `double` re, im;  
  `double` abs() `const` {  
    `return` sqrt( re * re +  
    im * im );  
  }  
}  
Complex x; // object  
d = x.abs(); // call abs |

- 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 the 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.

2.18.1 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).
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 (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:

```c
struct Complex {
    ... Complex add( Complex c ) {
        return (Complex){ 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.

2.18.2 Operator Member

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

```cpp
struct Complex {
    ... Complex operator+( Complex c ) { // rename add member
        return (Complex){ 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).

```cpp
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;
```
2.18.3 Constructor

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

```c
struct Complex {
    double re, im;
    Complex() { re = 0.; im = 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.

```c
Complex x;              // implicitly
Complex *y = new Complex; // implicitly rewritten as
```

- 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.

```cpp
Complex x(); // routine prototype, no parameters returning a complex
```

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

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

- Instead use constructor(s) with parameters:

```cpp
struct Complex {
    double re, im;
    Complex( double r = 0.0, double i = 0.0 ) { re = r; im = i; }
    ...
};
```

  Note, use of default values for parameters.

- Unlike Java, constructor argument(s) can be specified after a variable for local declarations:
Complex x, y(1.0), z(6.1, 7.2); implicitly rewritten as Complex x; x.Complex(0.0, 0.0); Complex y; y.Complex(1.0, 0.0); Complex z; z.Complex(6.1, 7.2);

• Dynamic allocation is same as Java:

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

• Constructor may force dynamic allocation when initializing 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] = (Complex){ i, 2.0 }  // initialization disallowed
}
for ( int i = 0; i < 10; i += 1 ) {
    ac[i] = Complex( i, 2.0 )  // assignment, not initialization
}
Complex *ap[10];  // array of complex pointers
for ( int i = 0; i < 10; i += 1 ) {
    ap[i] = new Complex( i, 2.0 );  // initialization allowed
}

• 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:
### 2.18.3.1 Literal

- Constructors can be used to create object literals (like `g++` type-constructor literals):

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

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

  ```java
  Complex x, y, z;
  x = Complex( 3.2 );  // complex literal value 3.2+0.0i
  y = x + Complex(1.3, 7.2);  // complex literal 1.3+7.2i
  z = Complex( 2 );  // 2 widened to 2.0, complex literal value 2.0+0.0i
  ```

  ```c++
  struct Foo {
    int i, j;
    void common(int p) { i = p; j = 1; }
    Foo() { common( 2 ); }
    Foo( int p ) { common( p ); }
  };
  ```

- Previous operator `+` for `Complex` is changed because type-constructor literals are disallowed for a type with constructors:
Complex operator+( Complex c ) {
    return Complex( re + c.re, im + c.im ); // create new complex value
}

2.18.3.2 Conversion

• Constructors are implicitly used for conversions:

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

• Allows built-in 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:
```cpp
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:

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

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

```cpp
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?

### 2.18.4 Destructor

• A destructor (finalize in Java) is a special member used to perform uninitialization at object deallocation:
An object type has one destructor; its name is the character “~” followed by the type name (like a constructor).

A destructor has no parameters nor return type (not even `void`):

A destructor is only necessary if an object 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.

A destructor is invoked before an object is deallocated, either implicitly at the end of a block or explicitly by a `delete`:
For local variables in a block, destructors **must be** called in reverse order to constructors because of dependencies, e.g., y depends on x.

A destructor is more common in C++ than a finalize in Java due to the lack of garbage collection in C++.

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

```c++
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 transferred 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.

### 2.18.5 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**:

\[
\text{Complex( } \text{const Complex } &c \text{ ) } \{ \ldots \} 
\]

• Declaration initialization:

\[
\text{Complex } y = x; \quad \text{implicitly rewritten as } \text{Complex } y; \ y.\text{Complex( } x \text{ );}
\]
- “=” 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:

\[
\text{Complex } \text{rtn( Complex } a, \text{ Complex } b \text{ ) } \{ \ldots \text{ return } a; \} 
\]
\[
\text{Complex } x, \ y; \\
\text{x = rtn( } x, \ y \text{ );} \quad \quad \text{ // creates temporary before assignment}
\]
- parameter is initialized by corresponding argument using its copy constructor:

\[
\text{Complex } \text{rtn( Complex } a, \text{ Complex } b \text{ ) } \{ \\
\quad a.\text{Complex( arg1 );} \ b.\text{Complex( arg2 );} \quad \text{ // initialize parameters} \\
\quad \ldots \quad \quad \text{ // with arguments}
\]
- temporaries *may* be created for arguments and return value, initialized using copy constructor:
Complex t1( x ), t2( y );
Complex tr( rtn( t1, t2 ) );

x = rtn( . . . ); implicitly rewritten as  
 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
```cpp
struct B {
    B() { cout << "B() \n"; }
    B( const B &c ) { cout << "B(&) \n"; }
    B &operator=( const B &rhs ) { cout << "B= \n"; }
};

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:

```
<table>
<thead>
<tr>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>B()</td>
<td>B()</td>
</tr>
<tr>
<td>B(&amp;)</td>
<td>B(&amp;)</td>
</tr>
<tr>
<td>B=</td>
<td>B=</td>
</tr>
</tbody>
</table>
```

- 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.

```cpp
struct D {
    ... // same declarations
    D() { cout << "D() \n"; }
    D( const D &c ) : i( c.i ), b( c.b ), a( c.a ) { cout << "D(&) \n"; }
    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= \n";
        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.

```cpp
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
- dangling pointer
• 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
    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 an equality operator
  (operator==) performing a deep compare, i.e., compare values not pointers.
2.18.6 Initialize const / Object Member

- C/C++ const members and local objects of a structure must be initialized at declaration:

<table>
<thead>
<tr>
<th>Ideal (Java-like)</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct Bar {</td>
<td>struct Bar {</td>
</tr>
<tr>
<td>Bar( int i ) { . . . }</td>
<td>Bar( int i ) { . . . }</td>
</tr>
<tr>
<td>// no default constructor</td>
<td>// no default constructor</td>
</tr>
<tr>
<td>} bar( 3 );</td>
<td>} bar( 3 );</td>
</tr>
<tr>
<td>struct Foo {</td>
<td>struct Foo {</td>
</tr>
<tr>
<td>const int i = 3;</td>
<td>const int i;</td>
</tr>
<tr>
<td>Bar * const p = &amp;bar;</td>
<td>Bar * const p;</td>
</tr>
<tr>
<td>Bar &amp;rp = bar;</td>
<td>Bar &amp;rp;</td>
</tr>
<tr>
<td>Bar b( 7 );</td>
<td>Bar b;</td>
</tr>
<tr>
<td>} x;</td>
<td>} x = { 3, &amp;bar, bar, 7 };</td>
</tr>
</tbody>
</table>

- Left: disallowed because fields cannot be directly initialized.

- Right: disallowed because Bar has a constructor so b must use constructor syntax.

- Try using a constructor:
Constructor/assignment

```c
struct Foo {
    const int i;
    Bar * const p;
    Bar &rp;
    Bar b;
    Foo() {
        i = 3; // after declaration
        p = &bar;
        rp = bar;
        b( 7 ); // not a statement
    }
};
```

Constructor/initialize

```c
struct Foo {
    const int i;
    Bar * const p;
    Bar &rp;
    Bar b;
    Foo() : // declaration order
        i( 3 ),
        p( &bar ),
        rp( bar ),
        b( 7 )
    {};
};
```

- Left: disallowed because `const` has to be initialized at point of declaration.
- Right: special syntax to indicate initialized at point of declaration.
- 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:
```c
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.

### 2.18.7 Static Member
- Static data-member creates a single instance for object type versus for object instances.

```c
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 ofFooobjects 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 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.

2.19 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.

<table>
<thead>
<tr>
<th>bits 63-32</th>
<th>bits 31-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3e8e36</td>
</tr>
<tr>
<td>5f</td>
<td>718c25e1</td>
</tr>
<tr>
<td>ad3e</td>
<td>7b5f1dbe</td>
</tr>
<tr>
<td>bc3b</td>
<td>ac69ff19</td>
</tr>
<tr>
<td>1070f</td>
<td>2d258dc6</td>
</tr>
</tbody>
</table>

• 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):
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 are scaled using modulus; e.g., generate 10 random number between 5-21:

    PRNG prng;
    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;
    }

• By initializing PRNG with a different “seed” each time the program is run, the generated sequence is different:

    PRNG prng( getpid() );   // process id of program
    prng.seed( time() );     // current time

• #include <cstdlib> provides C random routines srand and rand to set a seed
and generate random values, respectively.

```c
srand( getpid() ); // seed random generator
r = rand();       // obtain next random value
```

### 2.20 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:

  ```c
  { 
      cout << i << endl; // prints 4 ?
      int i = 4;         // declaration after usage
  }
  ```

  but ambiguous cases make this impractical:

  ```c
  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:

```cpp
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() {
  f( 5 );
  g( 4 );
}
void g( int i ) { ... }
void f( int i ) { ... }

• E.g., allowing main routine to appear first, and for separate compilation.
• Like Java, C++ does not always require DBU within a type:
Unlike Java, C++ requires a forward declaration for mutually-recursive declarations among types:

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
</table>
| **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:
**Caution: these types cause infinite expansion as there is no base case.**

- Java version compiles because \( t_1 / t_2 \) are references not objects, and Java can look ahead at \( T_2 \); C++ version disallowed because DBU on \( T_2 \) means it does not know the size of \( T_2 \).

- 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.

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>class</strong> T1 {</td>
<td><strong>struct</strong> T2; // forward</td>
</tr>
<tr>
<td>T2 t2;</td>
<td><strong>struct</strong> T1 {</td>
</tr>
<tr>
<td>T1() { t2 = new T2(); }</td>
<td>T2 &amp;t2; // pointer, break cycle</td>
</tr>
<tr>
<td>};</td>
<td>T1() : t2( *new T2 ) {} // DBU failure, size?</td>
</tr>
<tr>
<td><strong>class</strong> T2 {</td>
<td><strong>struct</strong> T2 {</td>
</tr>
<tr>
<td>T1 t1;</td>
<td>T1 t1;</td>
</tr>
<tr>
<td>T2() { t1 = new T1(); }</td>
<td>};</td>
</tr>
<tr>
<td>};</td>
<td>};</td>
</tr>
</tbody>
</table>

- Forward declaration of \( T_2 \) allows the declaration of variable \( T_1::t_2 \).
• 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:

```cpp
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.
2.21 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.
- C++ features work both within and among source files.
- C++ provides 3 levels of access control for object types:
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*.
\begin{verbatim}
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 {
    // public:
    int z;
    private:
    int x;
    protected:
    int y;
};

class C {
    // private:
    int x;
    protected:
    int y;
    public:
    int z;
};

• Use encapsulation to preclude object copying by hiding copy constructor
  and assignment operator:
\end{verbatim}
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.
• Implementation is factored into a new type Cartesian, “+” operator is factored into a routine outside and output “<<” operator must be outside.

• 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 \texttt{struct}).

\begin{verbatim}
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;
    }
};
\end{verbatim}

Complex makes Cartesian, re, im and impl accessible to friends.

\section*{2.22 System Modelling}

• \textbf{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. \((\text{CODE} = \text{TRUTH})\)
- Currently, UML is the most popular design tool.

### 2.22.1 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.

<table>
<thead>
<tr>
<th>class name</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
<td>(data)</td>
</tr>
<tr>
<td>- name : String</td>
<td></td>
</tr>
<tr>
<td>- age : Integer</td>
<td></td>
</tr>
<tr>
<td>- sex : Boolean</td>
<td></td>
</tr>
<tr>
<td>- owns : Car [ 0..5 ]</td>
<td></td>
</tr>
<tr>
<td>operations</td>
<td>(routines)</td>
</tr>
<tr>
<td>+ getName : String</td>
<td></td>
</tr>
<tr>
<td>+ getAge : Integer</td>
<td></td>
</tr>
<tr>
<td>+ getCars : Car [ 0..5 ]</td>
<td></td>
</tr>
<tr>
<td>+ buy( in car : Car, inout card : CreditCard ) : Boolean</td>
<td></td>
</tr>
</tbody>
</table>

- **Attribute** describes a property in a class.

  [visibility] name [“:” [type] [ “[” multiplicity “]” ] [“=” default] ]
visibility : access to property
  + ⇒ public, − ⇒ private, # ⇒ protected, ∼ ⇒ 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|∗), from 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
  + ⇒ public, − ⇒ private, # ⇒ protected, ∼ ⇒ 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**.

<table>
<thead>
<tr>
<th>Person</th>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>owns : Car [0..5]</td>
<td></td>
</tr>
</tbody>
</table>

- **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)**:

<table>
<thead>
<tr>
<th>Person</th>
<th>ownership</th>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>owns 0..5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

```
<table>
<thead>
<tr>
<th>Person</th>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>owns : Car [0..5]</td>
<td>owned : Person</td>
</tr>
</tbody>
</table>
```

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) ⇒ **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 ⇒ not navigable.
• Adornments options:
  ◦ show all arrows and Xs (completely explicit)
  ◦ suppress all arrows and Xs ⇒ 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
    ⇒ 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 snapshot of class instances at one moment during execution.

Object can specify values of class: “name : class-type” (underlined), attribute values.

<table>
<thead>
<tr>
<th>object name</th>
<th>mary : Person</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>name=“Mary”</td>
</tr>
<tr>
<td>attribute</td>
<td>age=29</td>
</tr>
<tr>
<td>values</td>
<td>sex=T</td>
</tr>
<tr>
<td></td>
<td>owns=(pointer)</td>
</tr>
</tbody>
</table>

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

**Vehicle**
- make: String
- model: String
- colour: String

**Contract**
- start: Date
- end: Date

**Client**
- name: String
- phone: String
+ rate(): Double

**Insurance**
- company: String
- policy: String
- expiry: String

**Accessory**
- surcharge: Double
+ surcharge(): Double

- **Truck**
- **SUV**
- **Car**

- **Corporate**
- **Individual**

- **FloorMat**
- **GPS**
- **SatelliteRadio**

no charge during sales
Object Diagram

:Contract
start="2009/09/07"
end="2012/09/07"

jfdoe:Individual
name="John F. Doe"
phone="204 888-2020"

:Insurance
company="SUN Lite"
policy="X-JAJ1567"
expiry="2011/05/31"

:Car
make="Honda"
model="Civic"
colour="silver"

:Truck
make="Ford"
model="F150"
colour="red"

:Contract
start="2010/10/13"
end="2013/10/13"

ibm:Corporate
name="IBM"
phone="519 744-3121"

:SUV
make="Nissan"
model="Quest"
colour="black"

:Contract
start="2008/01/25"
end="2014/01/25"

:GPS
- surcharge=500

:FloorMat
- surcharge=50
Invalid Object Diagram

:jfdoe:Individual
name="John F. Doe"
phone="204 888-2020"

:Company
name="IBM"
phone="519 744-3121"

:Contract
start="2008/01/25"
end="2014/01/25"

:surcharge=50
:FloorMat

:Contract
start="2010/10/13"
end="2013/10/13"

:suv
make="Honda"
model="CRV"
colour="blue"

:surcharge=50
:GPS

:Contract
start="2009/09/07"
end="2012/09/07"

:Insurance
company="SUN Lite"
policy="X-JAJ1567"
expiry="2011/05/31"

:Insurance
company="All Gate"
policy="A012678BJK"
expiry="2010/10/01"

:Car
make="Honda"
model="Civic"
colour="silver"

:Truck
make="Ford"
model="F150"
colour="red"

:SUV
make="Nissan"
model="Quest"
colour="black"
2.23 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:
#include <iostream>
#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;

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 executable 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.
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.

• 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: preprocessor variables, C/C++ types and forward declarations.
• **extern** qualifier means variable or routine definition is located elsewhere (not for types).

• Complex implementation placed in file complex.cc.

```cpp
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.

• 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__
#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.

2.24 Inheritance

• Object-oriented languages provide inheritance for writing reusable program-components.

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class Base { ... }</code></td>
<td><code>struct Base { ... }</code></td>
</tr>
<tr>
<td><code>class Derived extends Base { ... }</code></td>
<td><code>struct Derived : public Base { ... };</code></td>
</tr>
</tbody>
</table>

• 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.

2.24.1 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>
<thead>
<tr>
<th>Composition</th>
<th>Inheritance</th>
</tr>
</thead>
</table>
| `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 |
| `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 |
• 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.

```cpp
Car d; implicitly Engine b; b.Engine(); // implicit, hidden declaration
... rewritten as ... 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.

• Still possible to access outer-scope names using "::" qualification to specify the particular nesting level.

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
</table>
| class Base1 {  
  int i;
} | struct Base1 {  
  int i;
}; |
| class Base2 extends Base1 {  
  int i;
} | struct Base2 : public Base1 {  
  int i;  
  // overrides Base1::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;
  }  
} | struct Derived : public Base2 {  
  int i;  
  // overrides Base2::i
  void r() {  
    int i = 3;  
    Derived::i = 3; // this.i
    Base2::i = 3;
    Base2::Base1::i = 3; // or Base1::i
  }  
}; |

• 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.

2.24.2 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 to allow routines to handle multiple types, called polymorphism, e.g.: 
```c
struct Foo {
    int i;
    double d;
} f;

void r( Foo f ) { ... }
r( f );  // allowed
r( b );  // disallowed, name equivalence

struct Bar {
    int i;
    double d;
    ... 
} b;
```

- Since types Foo and Bar are structurally equivalent, instances of either type should work as arguments to routine r.
- Even if type Bar has more members at the end, routine r only accesses the common ones at the beginning as its parameter is type Foo.
- However, name equivalence precludes the call r( b ).
- **Type inheritance relaxes name equivalence by aliasing the derived name with its base-type names.**
```cpp
struct Foo {
    int i;
    double d;
} f;

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:

```cpp
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:

```cpp
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. ```cpp
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!

- **public** inheritance means both implementation and type inheritance.
- **private** inheritance means only implementation inheritance.

```cpp
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.

2.24.3 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.
• To pass arguments to other constructors, use same syntax as for initializing const members.
### 2.24.4 Copy Constructor / Assignment

- Each aggregate type has a default/copy constructor, assignment operator, and destructor, so these members cannot be inherited as they exist in the derived type.

- Otherwise, copy-constructor/assignment work like composition.

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class Base {</code></td>
<td><code>struct Base {</code></td>
</tr>
<tr>
<td>Base( <code>int i</code> ) { ... }</td>
<td>Base( <code>int i</code> ) { ... }</td>
</tr>
<tr>
<td><code>};</code></td>
<td><code>};</code></td>
</tr>
<tr>
<td><code>class Derived extends Base {</code></td>
<td><code>struct Derived : public Base {</code></td>
</tr>
<tr>
<td>Derived() { <code>super( 3 );</code> ... }</td>
<td>Derived() : Base( 3 ) { ... }</td>
</tr>
<tr>
<td>Derived( <code>int i</code> ) { <code>super( i );</code> ... }</td>
<td>Derived( <code>int i</code> ) : Base( <code>i</code> ) {...}</td>
</tr>
<tr>
<td><code>};</code></td>
<td><code>};</code></td>
</tr>
</tbody>
</table>
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!

2.24.5 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();

2.24.6 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 both calls should invoke
  Derived::r because argument b and reference bp point at an object of type
  Derived.

• If variable d is replaced with b, the calls should invoke Base::r.
• To invoke routine defined in referenced object, qualify member routine with `virtual`.

• To invoke routine defined by type of 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

```
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
```

### C++

```
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 and a destructor needs to make the destructor virtual so the most derived destructor is called through a base-type pointer/reference.

• Virtual routines are normally implemented by routine pointers.

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

• May be implemented in a number of ways:
2.24.7 Downcast

- Type inheritance can mask the actual type of an object through a pointer/reference.
- 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):
<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base bp = new Derived();</td>
<td>Base *bp = new Derived;</td>
</tr>
<tr>
<td>if ( bp instanceof Derived )</td>
<td>Derived *dp;</td>
</tr>
<tr>
<td>((Derived)bp).rtn();</td>
<td>dp = dynamic_cast&lt;Derived * &gt;(bp);</td>
</tr>
<tr>
<td>if ( dp != 0 ) {</td>
<td>// 0 =&gt; not Derived</td>
</tr>
<tr>
<td>// only in Derived</td>
<td>dp-&gt;rtn();</td>
</tr>
</tbody>
</table>

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

### 2.24.8 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.

2.24.9 Protected Members

• Inherited object types can access and modify public and protected members allowing access to some of an object’s implementation.
```cpp
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
}
```

### 2.24.10 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.

```cpp
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.
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(...); ...
}

• 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, i.e. use Shape over Polygon, use Polygon over Rectangle, etc.
2.24.11  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.
<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface Polygon {</td>
<td>struct Polygon {</td>
</tr>
<tr>
<td>int sides();</td>
<td>virtual int sides() = 0;</td>
</tr>
<tr>
<td>void move( int x, int y );</td>
<td>virtual void move( int x, int y ) = 0;</td>
</tr>
<tr>
<td>}</td>
<td>};</td>
</tr>
<tr>
<td>interface Rectilinear {</td>
<td>struct Rectilinear {</td>
</tr>
<tr>
<td>final int angle = 90;</td>
<td>enum { angle = 90 };</td>
</tr>
<tr>
<td>}</td>
<td>};</td>
</tr>
<tr>
<td>class Rectangle implements Rectilinear, Polygon {</td>
<td>class Rectangle : public Polygon,</td>
</tr>
<tr>
<td>private int x1, y1, x2, y2;</td>
<td>public Rectilinear {</td>
</tr>
<tr>
<td>public void move( int x, int y ) {}</td>
<td>int x1, y1, x2, y2;</td>
</tr>
<tr>
<td>public int sides() { return 4; }</td>
<td>public:</td>
</tr>
<tr>
<td>}</td>
<td>void move( int x, int y ) {}</td>
</tr>
<tr>
<td>class Square extends Rectangle {</td>
<td>int sides() { return 4; }</td>
</tr>
<tr>
<td>public void move( int x, int y ) {}</td>
<td>};</td>
</tr>
<tr>
<td>}</td>
<td>struct Square : public Rectangle {</td>
</tr>
<tr>
<td></td>
<td>void move( int x, int y ) {}</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
</tbody>
</table>

- Multiple inheritance has many problems (beyond this course).
- **Safe if restrict multiple inheritance to one public type and one or two private types.**
Generalization: reuse through forms of inheritance.

- **Rectilinear**
  - Abstract class
  - +angle: 90

- **Polygon**
  - Multiple inheritance
  - +sides: Integer
    - move(in x: Integer, in y: Integer)

- **Rectangle**
  - Single inheritance
  - +sides: Integer
    - move(...)

- **Trapezoid**
  - Single inheritance
  - +sides: Integer
    - move(...)

- **Square**
  - Single inheritance
  - move(...)

- **superclass** (base)
  - **Rectangle**
  - **Trapezoid**

- **subclass** (derived)
  - **Square**
• 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.

2.25 Composition / Inheritance Design

• Duality between “has-a” (composition) and “is-a” (inheritance) relationship.

• Types created from multiple composite classes; types created from multiple superclasses.

<table>
<thead>
<tr>
<th>Composition</th>
<th>Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>class A {...};</td>
<td>class A {...};</td>
</tr>
<tr>
<td>class B { A a; ...};</td>
<td>class B : A {...};</td>
</tr>
<tr>
<td>class C {...};</td>
<td>class C {...};</td>
</tr>
<tr>
<td>class D { B b; C c; ...};</td>
<td>class D : B, C {...};</td>
</tr>
</tbody>
</table>
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, Backhoe
    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**.

```cpp
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.

### 2.26 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, where there is identical code but different types:

  ```c++
  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:

  ```c++
  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 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.

• **There must be a space between the two ending chevrons or >> is parsed as operator>>.**


```cpp
template<typename T> struct Foo { ... };
Foo<Stack<int>> foo;  // syntax error (fixed C++11)
Foo<Stack<int> > foo;  // space between chevrons
```

- **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.**

### 2.26.1 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 the heap.

• To provide encapsulation, containers use a nested iterator type 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
- HashSet 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 “++”.

2.26.1.1 Vector

- Vector has random access, length, subscript checking (at), and assignment (like Java array).
### std::vector<T>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector()</td>
<td>create empty vector</td>
</tr>
<tr>
<td>vector( int N )</td>
<td>create vector with N empty elements</td>
</tr>
<tr>
<td>int size()</td>
<td>vector size</td>
</tr>
<tr>
<td>bool empty()</td>
<td>size() == 0</td>
</tr>
<tr>
<td>T &amp;operator[]( int i )</td>
<td>access ith element, NO subscript checking</td>
</tr>
<tr>
<td>T &amp;at( int i )</td>
<td>access ith element, subscript checking</td>
</tr>
<tr>
<td>vector &amp;operator=( const vector &amp; )</td>
<td>vector assignment</td>
</tr>
<tr>
<td>void push_back( const T &amp;x )</td>
<td>add x after last element</td>
</tr>
<tr>
<td>void pop_back()</td>
<td>remove last element</td>
</tr>
<tr>
<td>void resize( int n )</td>
<td>add or erase elements at end so size() == n</td>
</tr>
<tr>
<td>void clear()</td>
<td>erase all elements</td>
</tr>
</tbody>
</table>

- vector is alternative to C/C++ arrays.
```cpp
#include <vector>
int i, elem;
vector<int> v; // think: int v[0]
for ( ;; ) {
    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
```

- Vector declaration *may* specify an initial size, e.g., `vector<int> v(size)`, like a dimension.

- To reduce dynamic allocation, it is more efficient to dimension, when the size is known.
```cpp
int size;
cin >> size;  // read dimension
vector<int> v(size);  // think int v[size]

• Matrix declaration is a vector of vectors:

    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.
<table>
<thead>
<tr>
<th>std::vector&lt;T&gt;::iterator</th>
<th>std::vector&lt;T&gt;::iterator</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterator begin()</td>
<td>iterator pointing to first element</td>
</tr>
<tr>
<td>iterator end()</td>
<td>iterator pointing <strong>AFTER</strong> last element</td>
</tr>
<tr>
<td>iterator rbegin()</td>
<td>iterator pointing to last element</td>
</tr>
<tr>
<td>iterator rend()</td>
<td>iterator pointing <strong>BEFORE</strong> first element</td>
</tr>
<tr>
<td>iterator insert( iterator posn, const T &amp;x )</td>
<td>insert x before posn</td>
</tr>
<tr>
<td>iterator erase( iterator posn )</td>
<td>erase element at posn</td>
</tr>
<tr>
<td>++, --, +, +=, -, -= (insertion / random order)</td>
<td>forward/backward operations</td>
</tr>
</tbody>
</table>

- 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();  // initialize 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

2.26.1.2  Map

• map (dictionary) has random access, sorted, unique-key container of pairs (Key, Val).
<table>
<thead>
<tr>
<th>std::map&lt;Key,Val&gt; / std::pair&lt;const Key,Val&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>map()</strong></td>
</tr>
<tr>
<td><strong>int size()</strong></td>
</tr>
<tr>
<td><strong>bool empty()</strong></td>
</tr>
<tr>
<td><strong>Val &amp;operator[](</strong> const Key &amp;k <strong>)</strong></td>
</tr>
<tr>
<td><strong>int count(Key key)</strong></td>
</tr>
<tr>
<td><strong>map &amp;operator=(</strong> const map &amp; <strong>)</strong></td>
</tr>
<tr>
<td><strong>insert(pair&lt;</strong> const Key,Val&gt;( k, v <strong>)</strong> )</td>
</tr>
<tr>
<td><strong>erase(Key k)</strong></td>
</tr>
<tr>
<td><strong>void clear()</strong></td>
</tr>
</tbody>
</table>

### Example

<table>
<thead>
<tr>
<th>keys</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>2</td>
</tr>
<tr>
<td>green</td>
<td>1</td>
</tr>
<tr>
<td>red</td>
<td>0</td>
</tr>
</tbody>
</table>
```cpp
#include <map>
map<string, int> m, c; // 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

c = m; // map assignment
m.insert( pair<string, int>( "yellow", 3 ) ); // m["yellow"] = 3
if ( m.count( "black" ) ) // check for key "black"
  m.erase( "blue" ); // erase pair( "blue", 2 )
```

- First subscript for key creates an entry and initializes it to default or specified value.
- Iterator can search and return values in key order.
<table>
<thead>
<tr>
<th>std::map&lt;T&gt;::iterator / std::map&lt;T&gt;::reverse_iterator</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterator begin()</td>
</tr>
<tr>
<td>iterator end()</td>
</tr>
<tr>
<td>iterator rbegin()</td>
</tr>
<tr>
<td>iterator rend()</td>
</tr>
<tr>
<td>iterator find( Key &amp;k )</td>
</tr>
<tr>
<td>iterator insert( iterator posn, <strong>const</strong> T &amp;x )</td>
</tr>
<tr>
<td>iterator erase( iterator posn )</td>
</tr>
<tr>
<td>++, -- (sorted order)</td>
</tr>
</tbody>
</table>

- Iterator returns a pointer to a pair, with fields first (key) and second (value).
```cpp
#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
```

### 2.26.1.3 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>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list()</td>
<td>create empty list</td>
</tr>
<tr>
<td>list( int n )</td>
<td>create list with n default nodes</td>
</tr>
<tr>
<td>int size()</td>
<td>list size</td>
</tr>
<tr>
<td>bool empty()</td>
<td>size() == 0</td>
</tr>
<tr>
<td>list &amp;operator=( const list &amp; )</td>
<td>list assignment</td>
</tr>
<tr>
<td>T front()</td>
<td>first node</td>
</tr>
<tr>
<td>T back()</td>
<td>last node</td>
</tr>
<tr>
<td>void push_front( const T &amp;x )</td>
<td>add x before first node</td>
</tr>
<tr>
<td>void push_back( const T &amp;x )</td>
<td>add x after last node</td>
</tr>
<tr>
<td>void pop_front()</td>
<td>remove first node</td>
</tr>
<tr>
<td>void pop_back()</td>
<td>remove last node</td>
</tr>
<tr>
<td>void clear()</td>
<td>erase all nodes</td>
</tr>
</tbody>
</table>

- Iterator returns a pointer to a node.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iterator begin()</code></td>
<td>iterator pointing to first node</td>
</tr>
<tr>
<td><code>iterator end()</code></td>
<td>iterator pointing <strong>AFTER</strong> last node</td>
</tr>
<tr>
<td><code>iterator rbegin()</code></td>
<td>iterator pointing to last node</td>
</tr>
<tr>
<td><code>iterator rend()</code></td>
<td>iterator pointing <strong>BEFORE</strong> first node</td>
</tr>
<tr>
<td><code>iterator insert( iterator posn, const T &amp;x )</code></td>
<td>insert x before posn</td>
</tr>
<tr>
<td><code>iterator erase( iterator posn )</code></td>
<td>erase node at posn</td>
</tr>
<tr>
<td><code>++</code>, <code>--</code></td>
<td>(insertion order)</td>
</tr>
<tr>
<td></td>
<td>forward/backward operations</td>
</tr>
</tbody>
</table>
#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()

2.26.1.4 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).

```cpp
#include <iostream>
#include <list>
#include <vector>
#include <algorithm>
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 ) {
    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.
• E.g., an action to print on a specified stream must store the stream and have an \texttt{operator()} allowing the object to behave like a function:

\begin{verbatim}
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) );
}
\end{verbatim}

• Expression \texttt{Print(cout)} creates a constant \texttt{Print} object, and \texttt{for\_each} calls \texttt{operator()}(Node) in the object.

2.27 Namespace

• C++ \texttt{namespace} is used to organize programs and libraries composed of multiple types and declarations \textit{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>
<thead>
<tr>
<th>Java source files</th>
<th>C++ source file</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>package Foo; // file</code></td>
<td><code>namespace Foo {</code></td>
</tr>
<tr>
<td><code>public class X ... // export one type</code></td>
<td><code>    // types / declarations</code></td>
</tr>
<tr>
<td><code>// local types / declarations</code></td>
<td><code>}</code></td>
</tr>
<tr>
<td><code>package Foo; // file</code></td>
<td><code>namespace Foo {</code></td>
</tr>
<tr>
<td><code>public enum Y ... // export one type</code></td>
<td><code>    // more types / declarations</code></td>
</tr>
<tr>
<td><code>// local types / declarations</code></td>
<td><code>}</code></td>
</tr>
<tr>
<td><code>package Bar; // file</code></td>
<td><code>namespace Bar {</code></td>
</tr>
<tr>
<td><code>public class Z ... // export one type</code></td>
<td><code>    // types / declarations</code></td>
</tr>
<tr>
<td><code>// local types / declarations</code></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

• Contents of a namespace are accessed using full-qualified names:
<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foo.T t = \texttt{new} Foo.T();</td>
<td>Foo::T *t = \texttt{new} Foo::T();</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>import Foo.T;</td>
<td>using Foo::T;</td>
</tr>
<tr>
<td>import Foo::*;</td>
<td>using namespace Foo;</td>
</tr>
</tbody>
</table>

• \texttt{using} declaration \textit{unconditionally} introduces an alias (like \texttt{typedef}) into the current scope for specified entity in namespace.
  ○ May appear in any scope.
  ○ If name already exists in current scope, \texttt{using} fails.

  namespace Foo { int i = 0; }
  int i = 1;
  using Foo::i; // \textit{i exists in scope, conflict failure}

• \texttt{using} directive \textit{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 \texttt{using} directive in current scope, \texttt{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).
3 Tools

3.1 C/C++ Composition

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

• A programmer uses the four programming languages as follows:
  user edits $\rightarrow$ **preprocessor edits** $\rightarrow$ **templates expand** $\rightarrow$ **compilation**
  ($\rightarrow$ linking/loading $\rightarrow$ execution)

• C is composed of languages 1 & 3.

• The compiler interface controls all of these steps.
3.2 Compilation

C/C++ header files  C/C++ source files

(preprocessor)  cpp

preprocessed source code

(translator)  cc1plus

assembly code

(assembler)  as

object code

other object-code files and libraries  ld (linker)

./a.out  object

-E, -D, -I

-W, -v, -g, -S, -O1/2/3, -c

-o, -l, -L
• **Compilation** is the process of translating a program from human to machine readable form.

• The translation is performed by a tool called a **compiler**.

• Compilation is subdivided into multiple steps, using a number of tools.

• Often a number of options to control the behaviour of each step.

• Option are presented for `g++`, but other compilers have similar options.

• General format:

  \[ g++ \text{ option}-\text{list} \enspace *.cc \enspace *.o \ldots \]

### 3.2.1 Preprocessor

• Preprocessor (cpp) takes a C++ source file, removes comments, and expands `#include`, `#define`, and `#if` directives.

• Options:
  
  o –E run only the preprocessor step and writes the preprocessor output to standard out.

  \[
  \$ \ g++ \enspace –E \enspace *.cc \ldots \\
  ... \enspace \text{much output from the preprocessor}
  \]
○ -D define and optionally initialize preprocessor variables from the compilation command:

```bash
$ g++ -DDEBUG=2 -DASSN ... *.cc *.o ...
```

same as putting the following `#defines` in a program without changing the program:

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

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

### 3.2.2 Translator

- Translator takes a preprocessed file and converts the C++ language into assembly language for the target machine.

- Options:
  ○ `-Wkind` generate warning message for this “kind” of situation.
  * `-Wall` print ALL warning messages.
  * `-Werror` make warnings into errors so program does not compile.
-v show each compilation step and its details:

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

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

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

- g add symbol-table information to object file for debugger
- S compile source file, writing assemble code to file `source-file.s`
- O1/2/3 optimize translation to different levels, where each level takes more compilation time and possibly more space in executable
- c compile/assemble source file but do not link, writing object code to file `source-file.o`
3.2.3 Assembler

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

3.2.4 Linker

- Linker (ld) takes the implicit .o file from translated source and explicit .o files from the command line, and combines them into a new object or executable file.

- Linking options:
  - `-L directory` is a directory containing library files of precompiled code.
  - `-l` `library` search in library directories for given `library`.
  - `-o` gives the file name where the combined object/executable is placed.
    * If no name is specified, default name `a.out` is used.

- Look in library directory “/lib” for math library “m” containing precompiled “sin” routine used in “myprog.cc” naming executable program “calc”.

  ```
  $ gcc myprog.cc -L/lib -lm -o calc
  ```
3.3 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 recompilation.

3.3.1 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)

<table>
<thead>
<tr>
<th>source code</th>
<th>dependence graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.h</td>
<td>#include &quot;y.h&quot;</td>
</tr>
<tr>
<td>x.C</td>
<td>#include &quot;x.h&quot;</td>
</tr>
<tr>
<td>y.h</td>
<td>#include &quot;z.h&quot;</td>
</tr>
<tr>
<td>y.C</td>
<td>#include &quot;y.h&quot;</td>
</tr>
<tr>
<td>z.h</td>
<td>#include &quot;y.h&quot;</td>
</tr>
<tr>
<td>z.C</td>
<td>#include &quot;z.h&quot;</td>
</tr>
</tbody>
</table>

- Cycles in #include dependences are broken by #ifndef checks (see page 190).
- 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 the 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 leaves 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?)

3.3.2 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
      command2
         ...  # rules

• **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:**

```plaintext
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}
```

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 :
    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`
<table>
<thead>
<tr>
<th>file</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.d</td>
<td>x.o: x.C x.h y.h z.h</td>
</tr>
<tr>
<td>y.d</td>
<td>y.o: y.C y.h z.h</td>
</tr>
<tr>
<td>z.d</td>
<td>z.o: z.C z.h y.h</td>
</tr>
</tbody>
</table>

- `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).

```bash
$ make clean  # remove all products (don’t create “clean”)
```

- Hence, it is possible to have a universal Makefile for a single or multiple programs.

### 3.4 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, e.g., backing out of changes because of design problems.

- 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** is used to provide versioning and control cooperative work.

### 3.4.1 SVN

- **Subversion** (SVN 1.6) is a source-code management-system using the **copy-modify-merge model**.
  - master copy of all **project** files kept in a **repository**,
  - multiple versions of the project files managed in the repository,
  - developers **checkout** a **working copy** of the project for modification,
  - developers **checkin** changes from working copy with helpful integration using **text merging**.

**SVN works on file content not file time-stamps.**
3.4.2 Repository

- The repository is a directory containing multiple projects.
- courses
  - cs246
  - assn1
    - x.h, x.C, ...
  - assn2
    - ...

- more meta-projects / projects

- `svnadmin create` command creates and initializes a repository.

  ```bash
  $ svnadmin create courses
  Committed revision 1.
  $ svn mkdir file:///u/jfdoe/courses/cs246
  Committed revision 2.
  $ svn mkdir file:///u/jfdoe/courses/cs246/assn1
  Committed revision 2.
  ```

- `svn` `mkdir` command creates subdirectories for meta-projects and projects.

  ```bash
  $ svn mkdir file:///u/jfdoe/courses/cs246
  Committed revision 1.
  $ svn mkdir file:///u/jfdoe/courses/cs246/assn1
  Committed revision 2.
  ```

- files in repository are designated using URL, so must use absolute pathname
- `-m` (message) flag documents repository change.
- if no `-m` (message) flag specified, prompts for documentation (using an editor if shell environment variable `EDITOR` set).
• `svn ls` command lists directories.

```
$ svn ls file:///u/jfdoe/courses/cs246/assn1/
$ svn ls file:///u/jfdoe/courses/cs246/assn1
```

• If project directory `assn1` already exists, it can be added directly to the repository.

• `svn import` command copies an unversioned directory of files into a repository.

```
$ svn import assn1 file:///u/jfdoe/courses/cs246/assn1
Adding assn1/z.h
Adding assn1/x.C
Adding assn1/y.C
Adding assn1/z.C
Adding assn1/Makefile
Adding assn1/x.h
Adding assn1/y.h
Committed revision 2.
```
$ svn ls file:///u/jfdoe/courses/cs246/assn1
Makefile
x.C
x.h
y.C
y.h
z.C
z.h

• For students working together, the shared repository must be made accessible in the file system.

$ chgrp -R cs246_75 courses  # set group on directory and subfiles
$ chmod -R g+rwx courses  # allow group members access to ALL files

_for the path to the repository._

• Group name cs246_75 is acquired on a per course basis for each team of students.

3.4.3 Checking Out

• svn checkout command extracts a working copy of a project from the repository.
$ svn checkout file:///u/jfdoe/courses/cs246/assn1
Checked out revision 2.
$ ls -AF assn1
  .svn/

• For first checkout, directory assn1 is created in the current directory (unless it already exists).
• Subdirectory .svn contains administrative information for SVN and must not be modified.
• Working copy is then modified before being merged back into the repository.
• Other developers do not see each others working copy, and only see modifications when committed.
• To create a working-copy off-campus, use ssh URL:

  $ svn checkout svn+ssh://jfdoe@student.cs.uwaterloo.ca/u/jfdoe/courses

  (Replace file URL in subsequent commands with ssh URL.)

3.4.4 Adding

• Introduce files into project directory assn1.
```bash
$ cd assn1
$ ... # create files: Makefile x.C x.h y.C y.h z.h z.C
$ ls -AF
  .svn/  Makefile  x.C  x.h  y.C  y.h  z.C  z.h
```

- `svn add` command **schedules** addition of files (in current directory) into the repository.

```bash
$ svn add Makefile x.C x.h y.C y.h z.h z.C
A Makefile
A x.C
A x.h
A y.C
A y.h
A z.h
A z.C
```

**Addition only occurs on next commit.**

- Forgetting `svn add` for new files is a common mistake.
- **Add only project source-files into repository.**
- Product files, e.g., *.o, *.d, a.out, do not need to be versioned.
### 3.4.5 Checking In

- `svn commit` command updates the repository with the changes in working copy.

```
$ svn commit -m "initial project files"
Adding Makefile
Adding x.C
Adding x.h
Adding y.C
Adding y.h
Adding z.C
Adding z.h
Transmitting file data .......
Committed revision 3.
```

- If no `-m` (message) flag specified, prompts for commit documentation.
Always make sure your code compiles and runs before committing; it is unfair to pollute a project with bugs.

3.4.6 Modifying

• Edited files in working copy are implicitly **scheduled** for update on next commit.

  $ vi y.h y.C

• `svn rm` command removes files from working copy and **schedules** removal of files from the repository.
$ ls -AF
.svn/ Makefile x.C x.h y.C y.h z.C z.h

$ svn rm z.h z.C
D z.h
D z.C

$ ls -AF
.svn/ Makefile x.C x.h y.C y.h

• `svn status` command displays changes between working copy and repository.

    $ svn status
    D z.h
    M y.C
    D z.C
    M y.h

Files y.h / y.C have local modifications “M”, and z.h / z.C are deleted “D”.

• Possible to undo scheduled changes by reverting to files from repository.
• `svn revert` command copies unchanged files from repository to working copy.
$ svn revert y.C z.h
Reverted ‘y.C’
Reverted ‘z.h’
$ ls -AF
   .svn/ Makefile x.C x.h y.C y.h z.h

• Commit edits and removals.

$ svn commit -m "changes to y.h and remove z.C"
Sending y.h
Deleting z.C
Transmitting file data .
Committed revision 4.
$ svn ls file:///u/jfdoe/courses/cs246/assn1
Makefile
x.C
x.h
y.C
y.h
z.h

• Files in the repository can be renamed and copied.

• `svn mv` command renames file in working copy and *schedules* renaming in
the repository.

```
$ svn mv x.h w.h
A     w.h
D     x.h
$ ls -AF
   .svn/  Makefile  w.h  x.C  y.C  y.h
```

• `svn cp` command copies file in working copy and *schedules* copying in the repository:

```
$ svn cp w.h k.h
A     k.h
$ ls -AF
   .svn/  Makefile  k.h  w.h  x.C  y.C  y.h
```

• Commit renaming and copying.
$ svn commit -m "renaming and copying"
Adding    k.h
Adding    w.h
Deleting   x.h
Committed revision 5.
$ svn ls file:///u/jfdoe/courses/cs246/assn1
Makefile
k.h
w.h
x.C
y.C
y.h

3.4.7 Revision Number

- Each commit receives a revision number (currently 5).
- Information in older versions is accessible using suffix @N on URL.
- E.g., print file z.C, which last existed in revision 3.
- `svn cat` command prints specified file from the repository.

$ svn cat file:///u/jfdoe/courses/cs246/assn1/z.C@3
#include "z.h"
Copy deleted file z.C from repository into working copy and modify.

```bash
$ svn copy file:///u/jfdoe/courses/cs246/assn1/z.C@3 z.C
A z.C
$ ls -AF
 .svn/ Makefile k.h w.h x.C y.C y.h z.C z.h
$ ... # change z.C
$ svn commit -m "bring back z.C and modify"
Adding z.C
Transmitting file data.
Committed revision 6.
```

```bash
$ svn cat file:///u/jfdoe/courses/cs246/assn1/z.C@6
#include "z.h"
new text
```

### 3.4.8 Updating

- Synchronize working copy with commits in the repository from other developers.
<table>
<thead>
<tr>
<th>jfdoe</th>
<th>kdsmith</th>
</tr>
</thead>
<tbody>
<tr>
<td>modify x.C</td>
<td>modify x.C &amp; y.C</td>
</tr>
<tr>
<td>remove k.h</td>
<td>add t.C</td>
</tr>
</tbody>
</table>

- Assume kdsmith has committed changes.
- jfdoe attempts to committed changes.

```
$ svn commit -m "modify x.C"
Sending x.C
svn: Commit failed (details follow):
svn: File `/cs246/assn1/x.C’ is out of date
```

- jfdoe must resolve differences between their working copy and the current revision in the repository.
- `svn update` command attempts to update working copy from most recent revision.
$ svn update
D  k.h  file k.h deleted
U  y.C  file y.C updated without conflicts
A  t.C  file t.C added
Conflict discovered in ‘x.C’.
Select: (p) postpone, (df) diff-full, (e) edit,
        (mc) mine-conflict, (tc) theirs-conflict,
        (mf) mine-full, (tf) theirs-full,
        (s) show all options: df
--- .svn/text-base/x.C.svn-base  Sun May  2 09:54:08 2010
+++ .svn/tmp/x.C.tmp        Sun May  2 11:28:42 2010
@@ -1 +1,6 @@
        #include "x.h"
+<<<<<<<<<<< .mine
+jfdoe new text
+=======
+kdsmithe new text
+>>>>>>> .r7
Select: (p) postpone, (df) diff-full, (e) edit, (r) resolved,
        (mc) mine-conflict, (tc) theirs-conflict,
        (mf) mine-full, (tf) theirs-full,
        (s) show all options: tc
G  x.C  file x.C merged with kdsmithe version
Updated to revision 7.
○ (p) postpone: mark conflict to be resolved later
○ (df) diff-full: show changes to merge file
○ (e) edit: change merged file in an editor
○ (r) resolved: after editing version
○ (mc) mine-conflict: accept my version for conflicts
○ (tc) theirs-conflict: accept their version for conflicts
○ (mf) mine-full: accept my file (no conflicts resolved)
○ (tf) theirs-full: accept their file (no conflicts resolved)

• Merge algorithm is generally very good if changes do not overlap.
• Overlapping changes result in a conflict, which must be resolved.
• If unsure about how to deal with a conflict, it can be postponed for each file.
$ svn update
D  k.h  file k.h deleted
U  y.C  file y.C updated without conflicts
A  t.C  file t.C added

Conflict discovered in 'x.C'.
Select: (p) postpone, (df) diff-full, (e) edit,
      (mc) mine-conflict, (tc) theirs-conflict,
      (mf) mine-full, (tf) theirs-full,
      (s) show all options: p
C  x.C  file x.C conflict

Updated to revision 7.
Summary of conflicts:
  Text conflicts: 1

• Working copy now contains the following files:
<table>
<thead>
<tr>
<th>( x.C )</th>
<th>( x.C.mine )</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include &quot;x.h&quot;</td>
<td>#include &quot;x.h&quot;</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt; .mine jfdoe new text</td>
<td>jfdoe new text</td>
</tr>
<tr>
<td>= = = = = = = = kdsmith new text</td>
<td>kdsmith new text</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt; .r7</td>
<td></td>
</tr>
</tbody>
</table>

- \( x.C \): with conflicts
- \( x.C.mine \): jfdoe version of \( x.C \)
- \( x.C.r3 \): previous jfdoe version of \( x.C \)
- \( x.C.r7 \): kdsmith version of \( x.C \) in repository

- No further commits allowed until conflict is resolved.
- `svn resolve --accept ARG` command resolves conflict with version specified by `ARG`, for `ARG` options:
  - base: \( x.C.r3 \) previous version in repository
  - working: \( x.C \) current version in my working copy (needs modification)
○ mine-conflict : x.C.mine accept my version for conflicts
○ theirs-conflict : x.C.r7 accept their version for conflicts
○ mine-full : x.C.mine accept my file (no conflicts resolved)
○ theirs-full : x.C.r7 accept their file (no conflicts resolved)

$ svn resolve --accept theirs-conflict x.C
Resolved conflicted state of ’x.C’

● Removes 3 conflict files, x.C.mine, x.C.r3, x.C.r7, and sets x.C to the ARG version.

$ svn commit -m "modified x.C"
Sending x.C
Transmitting file data.
Committed revision 8.

3.5 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.

3.5.1 GDB

• The two most common UNIX debuggers are: dbx and gdb.

• File test.cc contains:
```c
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 );
    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.
- <Enter> without a command repeats the last command.
- run 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
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.

- backtrace 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.

- print command prints variables accessible in the current routine, object, or external area.

(gdb) print i
$1 = 100000000
• Can print any C++ expression:

```c
(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 variable` command changes the value of a variable in the current routine, object or external area.

```c
(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.

• **frame [n]** command moves the **current stack frame** to the n-th routine call on the stack.

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

```plaintext
(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, **breakpoints** are used.

• **break** command establishes a point in the program where execution suspends and control returns to the debugger.

```plaintext
(gdb) break main
Breakpoint 1 at 0x10710: file test.cc, line 7.
```

```plaintext
(gdb) break test.cc:3
Breakpoint 2 at 0x106d8: file test.cc, line 3.
```
o Set breakpoint using routine name or source-file:line-number.
o info 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.
• step [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.

- next [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

- continue \([n]\) 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.

• list 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

• quit 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
4 Software Engineering

- **Software Engineering** (SE) is the social process of designing, writing, and maintaining computer programs.
- SE attempts to find good ways to help people understand and develop software.
- However, what is good for people is not necessarily good for the computer.
- Many SE approaches are counter productive in the development of high-performance software.

1. The computer does not execute the documentation!
   - Documentation is unnecessary to the computer, and significant amounts of time are spent building it so it can be ignored (program comments).
   - Remember, the *truth* is always in the code.
   - However, without documentation, developers have difficulty designing and understanding software.

2. Designing by anthropomorphizing the computer is seldom a good approach (desktops/graphical interfaces).

3. Compiler spends significant amounts of time *undoing* SE design and coding approaches to generate efficient programs.
• It is important to know these differences to achieve a balance between programs that are good for people and good for the computer.

4.1 Software Crisis

• Large software systems (> 100,000 lines of code) require many people and months to develop.
• These projects too often emerge late, over budget, and do not work well.
• Today, hardware costs are low, and people costs are high.
• While commodity software is available, someone still has to write it.
• Since people produce software ⇒ software cost is great.
• Coupled with a shortage of software personnel ⇒ problems.
• Unfortunately, software is complex and precise, which requires time and patience.
• Errors occur and cost money if not lives, e.g., Ariane 5, Therac–25, Intel Pentium division error, Mars Climate Orbiter, UK Child Support Agency, etc.
4.2 Software Development

• Techniques for program development for small, medium, and large systems.

• Objectives:
  ◦ plan and schedule project (requirements documents, UML, time-lines)
  ◦ produce reliable, flexible, efficient programs
  ◦ produce programs that are easily maintained
  ◦ reduce the cost of software
  ◦ reduce program failure

• E.g., a typical software project:
  ◦ estimate 12 months of work
  ◦ hire 3 people for 4 months
  ◦ make up milestones for the end of each month

• However, first milestone is reached after 2 months instead of 1.

• To finish on time, hire 2 more people, but:
  ◦ new people require training
  ◦ work must be redivided

This takes at least 1 month.
Now 2 months behind with 9 months of work to be done in 1 month by 5 people.

To get the project done:
  - must reschedule
  - trim project goals

Often, adding manpower to a late software project makes it later.

Illustrates the need for a methodology to aid in the development of software projects.

### 4.3 Development Processes

There are different conceptual approaches for developing software:
**waterfall**: break down project based on activities that flow (downstream) across a timeline.

- activities: (cycle of) Requirements, Analysis, Design, Coding, Testing, Debugging (RADCTD).
- timeline: assign time to accomplish each activity up to project
completion time

**iterative/spiral**: break down project based on functionality and divide functions across a timeline
- functions ($F_i$): (cycle of) acquire/verify data, process data, generate data reports
- timeline: assign time to perform software cycle on each function up to project completion time

**staged delivery**: combination of waterfall and iterative
- start with waterfall for analysis/design, and finish with iterative for coding/testing

**agile/extreme**: short, intense iterations focused largely on code (versus documentation)
- often analysis and design are done iteratively
- often coding/testing done in pairs

- Pure waterfall is problematic because all coding/testing comes at end $\Rightarrow$ major problems can appear near project deadline.
- Pure agile can leave a project with “just” working code, and little or no testing / documentation.
- Selecting a process depends on:
- kind/size of system
- quality of system (mission critical?)
- hardware/software technology used
- kind/size of programming team
- working style of teams
- nature of completion risk
- consequences of failure
- culture of company

- Meta-processes specifying the effectiveness of processes:
  - Capability Maturity Model Integration (CMMI)
  - International Organization for Standardization for Standardization (ISO) 9000

- Meta-requirements
  - procedures cover key aspects of processes
  - monitoring mechanisms
  - adequate records
  - checking for defects, with appropriate and corrective action
  - regularly reviewing processes and its quality
  - facilitating continual improvement
4.4 Software Methodology

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

• System Design
  ○ Breakdown of requirements into modules, with their relationships and data flows.
  ○ Results in a description of the various modules required, and the data interrelating these.

• Implementation
  ○ writing the program

• Testing & Debugging
  ○ get it working

• Operation & Review
  ○ was it what the customer wanted and worth the effort?

• Feedback
If possible, go back to the above steps and augment the project as needed.

4.4.1 System Design

- Two basic strategies exist to systematically modularize a system:
  - top-down or functional decomposition
  - bottom-up

- Both techniques have much in common and so examine only one.

4.4.2 Top-Down

- Start at highest level of abstraction and break down problem into cohesive units, i.e., divide & conquer.
- Then refine each unit further generating more detail at each division.
- Each subunit is divided until a level is reached where the parts are comprehensible, and can be coded directly.
- This recursive process is called successive refinement or factoring.
- Units are independent of a programming language, but ultimately must be mapped into constructs like:
  - generics (templates)
- modules
- classes
- routines

- Details look at data and control flow within and among units.
- Implementation programming language is often chosen only after the system design.

- Factoring goals:
  - reduce module size: ≈ 30-60 lines of code, i.e., 1-2 screens with documentation
  - make system easier to understand
  - eliminate duplicate code
  - localize modifications

- Stop factoring when:
  - cannot find a well defined function to factor out
  - interface becomes too complex

- Avoid having the same function performed in more than one module (create useful general purpose modules)
• Separate work from management:
  ◦ Higher-level modules only make decisions (management) and call other routines to do the work.
  ◦ Lower-level modules become increasingly detailed and specific, performing finer grain operations.
• In general:
  ◦ do not worry about little inefficiencies unless the code is executed a LARGE number of times
  ◦ put thought into readability of program

4.5 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.
4.5.1 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., \( \sin(x) \), \( \text{avg}(\text{marks}) \)
  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 calculates 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.

### 4.5.2 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., \( \sin(x) \), \( \text{avg(marks)} \), 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.

   \[
   \text{print( process( getword( word ) ) ); // read } \rightarrow \text{process} \rightarrow \text{print (shell pipe)}
   \]
   - 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.

```plaintext
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

### 4.6 Design Principles

- low coupling, high cohesion (logical modularization)
- good interfaces (abstraction and encapsulation)
- type reuse (type inheritance)
- code reuse (implementation inheritance, physical modularization)
- indirection (data/routine pointers) to generalize objects

### 4.7 Design Patterns

- **Design patterns** have existed since people/trades developed formal approaches.
• E.g., chef’s cooking meals, musician’s writing/playing music, mason’s building pyramid/cathedral.

• **Pattern** is a common/repeated issue; it can be a problem or a solution.

• Name and codify common patterns for educational and communication purposes.

• Software pattern are solutions to problems:
  - name : descriptive name
  - problem : kind of issues pattern can solve
  - solution : general elements composing the design, with relationships, responsibilities, and collaborations
  - consequences : results/trade-offs of pattern (alternative/implementation issues)

• Patterns help:
  - extend developers’ vocabulary

  **Squadron Leader** : Top hole. Bally Jerry pranged his kite right in the how’s your father. Hairy blighter, dicky-birdied, feathered back on his Sammy, took a waspy, flipped over on his Betty Harper’s and caught his can in the Bertie.
  – RAF Banter, Monty Python
- offer higher-level abstractions than routines or classes

### 4.7.1 Pattern Catalog

<table>
<thead>
<tr>
<th></th>
<th>creational</th>
<th>structural</th>
<th>behavioural</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>class</strong></td>
<td><strong>factory method</strong></td>
<td><strong>adapter</strong></td>
<td>interpreter template</td>
</tr>
</tbody>
</table>

- Scope: applies to classes or objects
  - **class pattern** – relationships among classes and subclasses (static inheritance)
  - **object pattern** – relationships among objects (dynamic creation and association)
• Purpose: what a pattern does
  • creational: classes defer construction through inheritance / objects defer creation to other objects
  • structural: composition via inherited classes or assembled objects
  • behavioural: classes describes algorithm or control-flow / objects cooperate to perform task

4.7.1.1 Class Patterns

**factory method**: generalize creation of product with multiple variants

```cpp
struct Pizza {...}; // product
struct Pizzeria { // creator
    enum Kind { It, Mg, Ch, Dd }; // styles
    virtual Pizza *order( Kind p ) = 0;
};
struct Italian : public Pizzeria { // concrete creator (factory)
    Pizza *order( Kind p ); // create italian/margarita style
};
struct Chicago : public Pizzeria { // concrete creator
    Pizza *order( Kind p ); // create chicago/deep-dish style
};
```
Italian italian; Chicago chicago;  // factories
enum Kind { It, Mg, Ch, Dd };  
Pizza * dispatch( Kind pizza ) {  // parameterized creator
    switch ( pizza ) {
        case It: case Mg: return italian.order( Pizzeria::Mg );
        case Ch: case Dd: return chicago.order( Pizzeria::Dd );
        default: ; // error
    }
}
Pizza * p = dispatch( It );
p = dispatch( Ch );

• product (Pizza) objects are consistent across all factories (could be subclassed)
• clients get a concrete product (Pizza) from the creator (directly or indirectly), but product type is unknown
• client interacts with product object through its abstract interface (Pizza)
**adapter/wrapper**: convert interface into another

```cpp
struct Stack {
    virtual void push(...);
    virtual void pop(...);
};

struct Vector {
    virtual push_back(...);
    virtual pop_back(...);
};

struct VStack : public Stack, private Vector {
    // adapter/wrapper
    void push(...) { ... push_back(...); ... }
    void pop(...) { pop_back(...); }
};

void p( Stack &s ) { ... }
VStack vs;  // use VStack code with Stack routine
p( vs );
```

- VStack is polymorphic with Stack but implements push/pop with Vector::push_back/ Vector::pop_back.
**template method**: provide algorithm but defer some details to subclass

```cpp
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
4.7.1.2 Object Patterns

abstract factory: generalize creation of family of products with multiple variants

```cpp
struct Food {...}; // abstract product
struct Pizza : public Food {...}; // concrete product
struct Burger : public Food {...}; // concrete product
struct Restaurant {
    enum Kind { Pizza, Burger };  // abstract factory product
    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)
**singleton**: single instance of class

<table>
<thead>
<tr>
<th>.h file</th>
<th>.cc file</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class Singleton {</code></td>
<td><code>#include &quot;Singleton.h&quot;</code></td>
</tr>
<tr>
<td><code>struct Impl {</code></td>
<td><code>Singleton::Impl Singleton::impl( 3, 4 );</code></td>
</tr>
<tr>
<td><code>    int x, y;</code></td>
<td><code>Singleton::Impl::Impl( int x, int y )</code></td>
</tr>
<tr>
<td><code>    Impl( int x, int y );</code></td>
<td><code>    : x(x), y(y) {}</code></td>
</tr>
<tr>
<td><code>};</code></td>
<td><code>void Singleton::m() { ... }</code></td>
</tr>
<tr>
<td><code>static Impl impl;</code></td>
<td></td>
</tr>
<tr>
<td><code>public:</code></td>
<td></td>
</tr>
<tr>
<td><code>void m();</code></td>
<td></td>
</tr>
<tr>
<td><code>};</code></td>
<td></td>
</tr>
</tbody>
</table>

Singleton x, y, z; // all access same value

- Allow different users to have their 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.
**composite**: interface for complex composite object

```cpp
struct Assembly {
    string partNo();
    string name();
    double price();
    void insert(Assembly assm);
    void remove(string partNo);
    struct Iterator {...};
};

class Engine : public Assembly {...};
class Transmission : public Assembly {...};
class Wheel : public Assembly {...};
class Car : public Assembly {...};
class Stove : public Assembly {...};

// create parts for car
Car c;                    // composite object
    c.insert(engine);
    c.insert(transmission);
    c.insert(wheel);
    c.insert(wheel);
```

- recursive assembly type creates arbitrary complex assembly object.
- vertices are subassemblies; leafs are parts
- since composite type defines both vertices and leaf, all members may not apply to both
iterator: abstract mechanism to traverse composite object

```cpp
double price = 0.0;
Assembly::Iterator c( car );
for ( part = c.begin( engine ); part != c.end(); ++part ) { // engine cost
    price += part->price();
}
```

- iteration control: multiple starting/ending locations;
  depth-first/breadth-first, forward/backward, etc.; level of traversal
- iterator may exist independently of a composite design-pattern
adapter: convert interface into another

```cpp
struct Stack {
    virtual void push(...);
    virtual void pop(...);
};

struct VecToStack : public Stack {
    // adapter/wrapper
    Vector &vec;
    VecToStack(Vector &vec) : vec(vec) {} 
    void push(...) { ... vec.push_back(...); ... }
    void pop(...) { vec.pop_back(...); }
};

void p(Stack &s) { ... }
Vector vec;
VecToStack vtos(vec); // any Vector
p(vtos);
```

- specific conversion from Vector to Stack
**proxy**: frontend for another object to control access

```cpp
struct DVD {
    void play(...);
    void pause(...);
};

struct SPVR : public DVD {
    // static
    void play(...) { ... DVD::play(...); ... }
    void pause(...) { ... DVD::pause(...); ... }
};

struct DPVR : public DVD {
    // dynamic
    DVD *dvd;
    DPVR() { dvd = NULL; }
    ~DPVR() { if ( dvd != NULL ) delete dvd; }
    void play(...) { if ( dvd == NULL ) dvd = new T; dvd->play(...); ... }
    void pause(...) { ... don’t need dvd, no pause ... }
};
```

- proxy extends object’s type
- reverse structure of template method
- dynamic approach lazily creates control object
**Decorator**: attach additional responsibilities to an object dynamically

```cpp
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
**observer**: 1 to many dependency ⇒ change updates dependencies

```cpp
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 detach( 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.detach( *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
**visitor**: perform operation on elements of heterogeneous container

```cpp
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

4.8 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.

4.8.1 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.

### 4.8.2 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

  \[
  \text{HOURS} \leq 40 \\
  40 < \text{HOURS} \leq 45 \text{ (time and a half)} \\
  45 < \text{HOURS} \text{ (double time)}
  \]
  * 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.

### 4.8.3 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 ⇔ 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**: checking if 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.

### 4.8.4 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.
Index

!, 15, 84
!=, 84, 146
"", 189
", 12, 156
#, 2
#, 185
#define, 185
#elif, 190
#else, 190
#endif, 190
#if, 190
ifdef, 190
ifndef, 190
#include, 188
$, 2, 40
${}, 40
%
%, 2
&, 84, 86, 101, 124
&&, 84, 101
&=, 84
', 12, 81
*, 84, 86, 124, 138
*/, 73
*=', 84
+=, 89, 333
+=, 84, 89
,, 84, 89, 104
-, 84
--, 89, 333
-=, 84, 89
->, 84
-L, 360
-MD, 371
-MMD, 370
-O, 359
-S, 359
-W, 358
-c, 283, 359
-g, 359, 393
-I, 360
-o, 284, 360
-v, 359
,. 84
., 138
.C, 72
.c, 72
.cc, 72, 288
.cpp, 72
.h, 188, 286
.snapshot, 23
/, 5, 84, 87
\, 11, 81
/*, 73
//, 74
/=, 84
:, 115
::, 84, 153, 257, 297
;;, 74
;;;, 59
<, 35, 84, 146
<<, 84, 87, 168, 231
<<=, 84
<=, 84, 146
<>, 189
<ctrl>-c, 13
<ctrl>-d, 36, 172
=, 16, 40, 84, 146
==, 84, 146, 242
>, 12, 35, 84, 146
>&, 35, 37
>=, 84, 146
pure, 321
abstract data-type, 258
abstract factory, 425
abstraction, 220, 258
    procedural, 157
    routine, 157
acceptance testing, 445
access control, 258
adapter, 423, 432
add, 378
ADT, 258
aggregates, 133
aggregation, 274
agile, 407
alias, 151, 300, 351
alias, 15, 24
allocation
    array, 135, 207
dynamic, 202
    array, 207
heap, 204, 206, 332
array, 207
matrix, 208
stack, 96, 206
argc, 180, 181
argument, 161
argv, 180, 182
array, 121, 133, 135, 154, 155, 165, 166, 175, 180, 207, 226
2-D, 208
deallocation, 207
dimension, 133–135, 156, 165, 207, 335
parameter, 165
as, 360
assembler, 360
assertion, 191
assignment, 85, 88, 152, 234, 305
array, 135, 333
cascade, 88
initializing, 77
operator, 260, 294
association, 269
unidirectional, 269
association class, 273
atoi, 182
attribute, 267
backquote, 11
backslash, 6, 11, 81
backspace key, 3
backtrace, 394
backward branch, 116, 120
bang, 15
bash, 2, 44, 54
bash, 19
basic types, 76, 120
bool, 76
char, 76
double, 76
float, 76
int, 76
wchar_t, 76
behavioural, 326
bit field, 140
bitwise copy, 236
black-box testing, 441
block, 71, 96, 155
{ }, 74, 96
blueprint, 266
bool, 76, 81
boolalpha, 169
boundary value testing, 442
break, 101, 105, 120
labelled, 115
break, 396
breakpoint, 396
continue, 399
next, 398
step, 397
C–c, 13

C–d, 36, 172
c_str, 146
call-back routine, 217
cascade, 169
cascade assignment, 88
case, 59, 100
;;, 59
pattern, 59
case-sensitive, 40, 75
cast, 84, 89, 92, 123, 155, 178, 212, 314
cat, 24, 384
cd, 14
cerr, 168
char, 76, 78, 81
checkin, 372
checkout, 372
checkout, 376
chevron, 84, 168, 231, 330
chgrp, 33
chmod, 34
chsh, 20
cin, 168
**class**, 219, 260
class model, 266
class pattern, 420
classes diagram, 267
clear, 174
cmp, 26
code inspection, 440
coercion, 91, 123, 178, 204, 213
cast, 178
  explicit, 92, 178
  **reinterpret_cast**, 178
cohesion, 414
coincidental, 418
comma expression, 89, 104, 208
command options, 3
command-line arguments, 180
  argc, 180, 181
  argv, 180, 182
  main, 180
command-line interface, 1
comment, 2, 72–74
  #, 2
  */, 73
  /*, 73
  //, 74
  nesting, 73, 74
  out, 73, 190
commit, 379
common coupling, 414
communicational, 415
compilation, 72, 357
g++, 72
compiler, 72, 357
  options
    -D, 186, 358
    -E, 357
    -l, 358
-L, 360
-MD, 371
-MMD, 370
-O, 359
-S, 359
-W, 358
-c, 283, 359
-g, 359, 393
-l, 360
-o, 284, 360
-v, 359

separate compilation, 191, 254
composite, 429
composition, 275, 295, 324
explicit, 295
concrete class, 320
conditional expression evaluation, 101
& & , 101
? :, 101

||, 101
partial evaluation, 101
short-circuit, 101
conditional inclusion, 189
\textbf{const}, 82, 130, 164, 187, 220
constant, 80, 83, 220, 288
initialization, 187
parameter, 164
variable, 83
construction, 296
constructor, 121, 224, 225, 296, 304
\textbf{const} member, 243
copy, 234, 260, 305
implicit conversion, 229
literal, 228
passing arguments to other constructors, 304
type, 120
container, 331
deque, 331
list, 331
map, 331
queue, 331
stack, 331
vector, 331, 333
content coupling, 414
contiguous object, 232
continue, 105
    labelled, 115
continue, 399
contra-variance, 303
control coupling, 414
control structure, 95
    block, 95, 96
    { }, 74, 96
    conditional expression evaluation, 101
    &&, 101
    ?, 101
    ||, 101
partial evaluation, 101
short-circuit, 101
looping, 95, 102
break, 63
continue, 63
do, 103
for, 62, 103
while, 60, 102
selection, 95, 98
break, 101
case, 59, 100
dangling else, 98
default, 101
else, 98
if, 57, 98
pattern, 59
switch, 99, 182
test, 54
short-circuit expression evaluation, 101
transfer, 95
conversion, 89, 123, 147, 229
cast, 84, 89
dynamic_cast, 314
explicit, 89, 212
implicit, 89, 162, 211, 229
narrowing, 90
promotion, 90
static_cast, 89
widening, 90
copy constructor, 235, 260, 294
copy-modify-merge model, 372
coupling, 413
cout, 168
cp, 22, 383
cpp, 357
create, 374
csh, 2, 45, 54
csh, 19
current directory, 6–8, 14, 21
current stack frame, 396
dangling else, 98
dangling pointer, 205, 241
data coupling, 413
data member, 137
dbx, 392
debug print statements, 195
debugger, 391
Debugging, 194
debugging, 194, 440
dec, 169
declaration, 75
  basic types, 76
  const, 187
type constructor, 120
type qualifier, 77
variable, 76
Declaration Before Use, 254
declaration before use, 252
decorator, 434
deep compare, 242
depth copy, 238, 242
default
  parameter, 214
default, 101
default constructor, 224
default initialized, 155
default value, 165, 224
  parameter, 165
delegation, 326
delete, 202
  [], 207
delete key, 3
dependence, 361
deque, 331, 344
dereference, 40, 86, 124
dereferencing, 124
design patterns, 418
desk checking, 441
desktop, 1
destruction, 296
  explicit, 232
  implicit, 232
  order, 233
destructor, 231, 296, 304
diff, 26
dimension, 133–135, 156, 165, 207, 335
do, 103
documentation, 72
double, 76, 81
double quote, 12, 47
downcast, 314
duplicate code, 157
dynamic allocation, 226
dynamic storage management, 202,
  233
dynamic_cast, 314
eager evaluation, 102
echo, 18
egrep, 29
\textbf{else}, 98
encapsulation, 258, 332
dead of file, 172
dead of line, 71
deadl, 71, 169
Enter key, 2
\textbf{enum}, 121, 187
deaderation, 121
deaderator, 121
dead, 172
equivalence
  name, 151
  structural, 151
equivalence partitioning, 442
derror guessing, 443
dead, 11
dead sequence, 143
Escape sequence, 81
deaded, 54
evaluation
  eager, 102
  lazy, 101
  partial, 101
  short-circuit, 101, 112
event programming, 217
dead, 32
dead execution error, 200
dead
  static multi-exit, 110
  static multi-level, 114
dead, 19
dead, 71
dead status, 19, 48
dead explicit coercion, 92, 178
dead explicit conversion, 89, 212
dead export, 280, 289
dead expression, 84
dead extreme, 407
Eye Candy, 108
factoring, 157, 410
factory method, 421
fail, 168, 172
false, 90
feof, 175
file
  .h, 188
    opening, 168
file inclusion, 188
file management
  file permission, 32
    input/output redirection, 35
      <, 35
      &>, 35
      >, 35
      |, 35
file permission
  execute, 32
  group, 32
  other, 32
read, 32
search, 32
user, 32
write, 32
file suffix
  .C, 72
  .c, 72
  .cc, 72, 288
  .cpp, 72
  .h, 286
  .o, 284
files, 4
  input/output redirection, 35
find, 28, 146
find_first_not_of, 146
find_first_of, 146
find_last_not_of, 146
find_last_of, 146
fix-up routine, 216
fixed, 169
flag variable, 111
**float**, 76, 80
**for**, 62, 103
for_each, 347
format
  I/O, 169
formatted I/O, 167, 168
forward branch, 116
forward declaration, 253
frame, 396
free, 202
free, 202
**friend**, 262
friendship, 262, 297
fstream, 168
function, 158
function member, 137
function-call operator, 251
functional, 415
functional testing, 444
functor, 251, 348
g++, 72, 134, 155, 228, 357
garbage collection, 202
gdb
  backtrace, 394
  break, 396
  breakpoint, 396
    continue, 399
    next, 398
    step, 397
  continue, 399
  frame, 396
  info, 397
  list, 400
  next, 398
  print, 394
  run, 393
  step, 397
gdb, 392
generalization, 323
generate, 266
globbing, 7, 11, 28, 31, 59
gmake, 365
goto, 115, 119, 120
  label, 115
graphical user interface, 1
group, 32
has-a, 296, 324
heap, 161, 204, 206, 332
  array, 207
help, 13
heterogeneous values, 137, 141
hex, 169
hidden file, 9, 21, 23
history, 15
home directory, 6, 14
homogeneous values, 133
hot spot, 194
human testing, 440

I/O
  cerr, 168
cin, 168
clear, 174
cout, 168
fail, 168
formatted, 168
fstream, 168
ifstream, 168
ignore, 175
iomanip, 169
iosstream, 168
manipulators, 169
  boolalpha, 169
dec, 169
decl, 169
fixed, 169
hex, 169
left, 169
boolalpha, 169
no\textbf{showbase}, 169
no\textbf{showpoint}, 169
no\textbf{skipws}, 169
\textbf{oct}, 169
\textbf{right}, 169
\textbf{scientific}, 169
\textbf{setfill}, 169
\textbf{setprecision}, 169
\textbf{setw}, 169
\textbf{showbase}, 169
\textbf{showpoint}, 169
\textbf{skipws}, 169
\textbf{ofstream}, 168
\textbf{identifier}, 75, 115
\textbf{if}, 57, 98
\textbf{?:}, 101
dangling \textbf{else}, 98
\textbf{else}, 98
\textbf{ifstream}, 168
\textbf{ignore}, 175
implementation, 286
\textbf{implementation inheritance}, 294
implicit conversion, 89, 162, 211, 229
\textbf{import}, 280, 286
\textbf{import}, 375
\textbf{indirection}, 127
\textbf{info}, 397
\textbf{Inheritance}, 327
\textbf{inheritance}, 294, 324
implementation, 294
type, 294, 298
\textbf{initialization}, 154, 224, 228, 234, 243, 296, 304, 305
array, 155
forward declaration, 257
\textbf{string}, 156
\textbf{structure}, 155
\textbf{inline}, 187
\textbf{input}, 71, 166, 170
>>, 231
end of file, 172
eof, 172
fail, 172
feof, 175
formatted, 168
manipulators
  iomanip, 169
  noskipws, 169
  skipws, 169
standard input
  cin, 168
input/output redirection, 35
  filter
    |, 35
input
  <, 35
output
  >, 35
  >&, 35

int, 76, 78, 81
  INT16_MAX, 79, 80
  INT16_MIN, 79, 80
  int16_t, 79, 80
  INT32_MAX, 79
  INT32_MIN, 79
  int32_t, 79
  INT64_MAX, 79
  INT64_MIN, 79
  int64_t, 79
  INT8_MAX, 79
  INT8_MIN, 79
  int8_t, 79
  INT_MAX, 78
  INT_MIN, 78
integral type, 140
integration testing, 444
interaction model, 266
interface, 157, 220, 286
interface class, 321
interfaces, 157
iomanip, 169
iostream, 71, 168
is-a, 298, 299, 324
iteration statement
  break, 105
  continue, 105
iterative, 407
iterator, 332, 431
  ++, 333
  --, 333
  for_each, 347
keyword, 75
keywords, 40
ksh, 2
label, 115
label variable, 115
language
  preprocessor, 355
programming, 355
template, 355
lazily, 101
lazy evaluation, 101
ld, 360
left, 169
less, 24
linker, 360
list, 331, 344, 400
  back, 345
  begin, 346
  clear, 345
  empty, 345
  end, 346
  erase, 346
  front, 345
  insert, 346
  pop_back, 345
  pop_front, 345
  push_back, 345
  push_front, 345
begin, 346
end, 346
size, 345
literal, 80–82, 143, 154, 171
  bool, 81
  char, 81
  double, 81
  escape sequence, 81
  initialization, 154
  int, 81
  string, 81, 143
    type constructor, 154
literals, 121
LONG_MAX, 78
LONG_MIN, 78
loop
  mid-test, 107
  multi-exit, 107
  looping statement, 102
    break, 63
    continue, 63
    do, 103
    for, 62, 103
    while, 60, 102
lp, 25
lpstat, 25
ls, 21, 32, 375
machine testing, 441
macros, 187
main, 71, 180, 254
make, 364
make, 365
malloc, 202
man, 20
managed language, 202
manipulators, 169
map, 331, 340
  begin, 343
  end, 343
  erase, 343
  find, 343
  insert, 343
  begin, 343
  end, 343
math library, 360
matrix, 134, 166, 208, 336
member, 137
  anonymous, 296
  const, 243
  constructor, 224
destruction, 231, 296, 304
initialization, 224, 296, 304
object, 220
operator, 223
overloading, 223
pure virtual, 317, 320
static member, 245
virtual, 311, 315
member selection, 138
memberwise copy, 236
memory leak, 205, 208, 241
mid-test loop, 107
mixin, 324
mkdir, 21, 374
modularization, 156
modularize, 410
module, 156, 157
more, 24
multi-exit
  loop, 107
  mid-test, 107
multi-level exit
  static, 114
multiple inheritance, 321
mutually recursive, 253, 255
mv, 22, 382
name equivalence, 151, 299, 300, 302, 327
namespace, 71, 349
    std, 71
narrowing, 90
navigable, 271
nesting, 297
    blocks, 96, 97
    comments, 73, 74
    initialization, 155
    preprocessor, 189
    routines, 160
type, 152
new, 202
next, 398
noboolalpha, 169
non-contiguous, 232, 233
noshowbase, 169
noshowpoint, 169
noskipws, 169
npos, 146
NULL, 154, 189
null address, 126
null character, 144
object, 218, 219
    anonymous member, 296
    assignment, 234, 305
    const member, 243
    constructor, 224, 296, 304
    copy constructor, 234, 260, 305
    default constructor, 224
destructor, 231, 296, 304
    initialization, 224, 304
    literal, 228
    member, 220
    pure virtual member, 317, 320
    static member, 245
virtual member, 311, 315
object code, 360
object diagram, 272
object model, 266
object pattern, 420
object-oriented, 218, 294
observer, 436
oct, 169
ofstream, 168
open, 168
  file, 168
operation, 268
operators
  *, 86, 124
  <<, 168, 231
  >>, 168, 231
  & , 86, 124
  arithmetic, 84
  assignment, 84
  bit shift, 84
bitwise, 84
cast, 84
comma expression, 84
control structures, 84
logical, 84
overloading, 168, 223
pointer, 84, 86, 124
priority, 85
relational, 84
selection, 153, 297
string, 146
struct, 84
selection, 257
other, 32
output, 71, 166, 176
  <<, 231
endl, 71
formatted, 168
manipulators
  boolalpha, 169
dec, 169
dendl, 169
fixed, 169
hex, 169
iomanip, 169
left, 169
nboolalpha, 169
nshowbase, 169
nshowpoint, 169
oct, 169
right, 169
scientific, 169
setfill, 169
setprecision, 169
setw, 169
showbase, 169
showpoint, 169
standard error
cerr, 168
standard output
cout, 71, 168
overflow, 86
overload, 180
overloading, 168, 210, 223, 224, 231
override, 297, 302, 309, 311
overriding, 97
paginate, 25
parameter, 161
array, 165
constant, 164
default value, 165
pass by reference, 162
pass by value, 162
prototype, 254
parameter passing
array, 165
pass by reference, 162
pass by value, 162
pattern, 59, 419
pattern matching, 7
performance testing, 445
pointer, 121, 123, 154
  0, 154
  array, 135, 207
  matrix, 208
  NULL, 154, 189
pointer variable, 126
polymorphic, 314
polymorphism, 299
Polymorphism, 325
preprocessor, 185, 355, 357, 370
  #define, 185
  #elif, 190
  #else, 190
  #endif, 190
  #if, 190
  #ifdef, 190
  #ifndef, 190
  #include, 188
file inclusion, 188
  macros, 187
  variable, 186, 358
  print, 394
  priority, 85
  private, 259
  procedural, 416
  procedural abstraction, 157
  procedure, 158
  program
    structure, 72
  program structure, 72
    block, 71
    main, 71
  project, 372
  promotion, 90
  prompt, 1, 2
  $, 2
  %, 2
  >, 12
  protected, 259
prototype, 252, 253
proxy, 433
pseudo random-number generator, 249
pseudo random-numbers, 249
**public**, 137, 259
pure abstract-class, 321
pure virtual member, 317, 320
**pwd**, 14
queue, 331, 344
quoting, 11
random number, 248
generator, 249
pseudo-random, 249
seed, 251
Random-number generator, 249
read, 32
real time, 18
recursive type, 139
reference, 86, 121, 123, 124
initialization, 129
reference parameter, 162
referencing, 124
regression testing, 444
regular expressions, 7
**reinterpret_cast**, 178
relative pathname, 6
replace, 146
repository, 372, 373
resolve, 390
**return**, 71, 158
return code, 19
Return key, 2
return type, 158
reuse, 294
revert, 381
rfind, 146
right, 169
rm, 23, 380
routine, 157
argument/parameter passing, 161
array parameter, 165
function, 158
member, 220
parameter, 159
   pass by reference, 162
   pass by value, 162
procedure, 158
prototype, 252
return, 158
return type, 158
routine overloading, 211
routine prototype
   forward declaration, 253
scope, 220
routine abstraction, 157
routine member, 137
routine pointer, 214
routine prototype, 253
run, 393

scientific, 169
scope, 220, 257, 350
script, 45
search, 32
security testing, 445
sed, 39
selection operator, 153
selection statement, 98
   break, 101
   case, 59, 100
   default, 101
   else, 98
   if, 57, 98
   pattern, 59
   switch, 99, 182
self-assignment, 241
semantic error, 200
semi-colon, 57
semicolon, 74, 96, 137
sentinel, 144
separate compilation, 191, 280
  -c, 283
sequential, 415
setfill, 169
setprecision, 169
setw, 169
sh, 2, 45
sh, 19
sha-bang, 45
shell, 1
  bash, 2, 54
csh, 2, 54
ksh, 2
login, 53
prompt, 2
  $, 2
  %, 2
  >, 12
sh, 2
tcsh, 2

shell program, 45
shift, 56
short, 78
short-circuit, 54, 101
short-circuit expression evaluation, 101
||, 101
showbase, 169
showpoint, 169
SHRT_MAX, 78
SHRT_MIN, 78
signature, 253
signed, 78
single quote, 12
singleton, 428
size_type, 146
sizeof, 86
sketch, 266
skipws, 169
slicing, 315
software development
   .cc, 288
   .h, 286
   .o, 284
   separate compilation, 280
software engineering, 156, 402
source, 51
source file, 254, 258
source-code management, 371
source-code management-system, 372
spiral, 407
ssh, 31
stack, 96, 161
stack, 331, 344
stack allocation, 206
staged delivery, 407
stamp coupling, 413
statement, 74
static, 291
static block, 160, 246
static exit
   multi-exit, 110
   multi-level, 114
static multi-level exit, 114
static_cast, 89
status, 381
std, 71
stderr, 168
stdin, 168
stdout, 168
step, 397
strcat, 146
strcpy, 146
strcspn, 146
stream
   cerr, 168
   cin, 168
   clear, 174
cout, 168
fail, 168
formatted, 168
fstream, 168
ifstream, 168
ignore, 175
input, 71
cin, 168
end of file, 172
eof, 172
fail, 172
manipulators
boolalpha, 169
dec, 169
endl, 169
fixed, 169
hex, 169
iomanip, 169
left, 169
nboolalpha, 169
nshowbase, 169
nshowpoint, 169
nskipws, 169
oct, 169
right, 169
scientific, 169
setfill, 169
setprecision, 169
setw, 169
showbase, 169
showpoint, 169
skipws, 169
ofstream, 168
output, 71
cout, 71
endl, 71
stream file, 168
stress testing, 445
string, 81, 143
C++
!=, 146
+, 146
<, 146
<=, 146
=, 146
==, 146
>, 146
>=, 146
[], 146
c_str, 146
find, 146
find_first_not_of, 146
find_first_of, 146
find_last_not_of, 146
find_last_of, 146
npos, 146
replace, 146
rfind, 146
size_type, 146
substr, 146
C

[], 146
strcat, 146
strcpy, 146
strcspn, 146
strlen, 146
strncat, 146
strncpy, 146
strspn, 146
strstr, 146
literal, 143
null termination, 144
stringstream, 182
strlen, 146
strncat, 146
strncpy, 146
strspn, 146
strstr, 146
struct, 219, 260
structurally equivalent, 151
structure, 121, 136, 137, 154, 155, 218, 219
member, 137, 219
data, 137
function, 137
initialization, 137
routine, 137
visibility
default, 137
**public**, 137
**struct**, 84
structured programming, 105
subscript, 134
subshell, 19, 45, 54
substitutability, 325
substr, 146
subversion, 372
successive refinement, 410
suffix
  .C, 72
switch, 99, 182
  break, 101
  case, 100
  default, 101
syntax error, 199
system command, 364
system modelling, 264
system testing, 444
system time, 18
tab key, 10
target value, 127
target variable, 127
tcsh, 2
tcsh, 19
template, 327, 355
  routine, 327
  type, 328
template method, 424
template routine, 327
template type, 328
temporal, 416
terminal, 2, 4
test, 54
test harness, 443
test-case design, 441
Testing
  Integration, 444
testing, 439
    acceptance, 445
    black-box, 441
    functional, 444
    gray-box, 442
    harness, 443
    human, 440
    machine, 441
    performance, 445
    regression, 444
    security, 445
    stress, 445
    system, 444
unit, 444
usability, 445
volume, 445
white-box, 442
text merging, 372
this, 220
time, 18
time stamp, 363
token, 185
translation unit, 280
translator, 358
true, 90
type, 17
type aliasing, 151
type coercion, 178
Type constructor, 120
type constructor
  array, 133
  enumeration, 121, 187
  literal, 154
pointer, 123
reference, 123
structure, 136
type aliasing, 151
union, 141
type conversion, 89, 211, 229, 314
type equivalence, 299, 302
type inheritance, 294, 299
type nesting, 152
type qualifier, 77, 79, 130
  const, 82, 130
  long, 78
  short, 78
  signed, 78
  static, 291
  unsigned, 78
type-constructor literal
  array, 154
  pointer, 154
  structure, 154
typedef, 151, 351
UINT16_MAX, 79
uint16_t, 79
UINT32_MAX, 79
uint32_t, 79
UINT64_MAX, 79
uint64_t, 79
UINT8_MAX, 79, 80
uint8_t, 79, 80
UINT_MAX, 78
ULLONG_MAX, 78
ULONG_MAX, 78
undefined, 127
unformatted I/O, 167, 177
unidirectional association, 269
unified modelling language, 266
uninitialization, 231
uninitialized variable, 77, 127, 201, 204
union, 141
unit testing, 444
unmanaged language, 202
unsigned, 78
update, 386
usability testing, 445
user, 32
user time, 18
USHRT_MAX, 78
using
    declaration, 351
directive, 351
value parameter, 162
variable declarations
type qualifier, 77, 79
variables
    constant, 83
dereference, 86, 124
reference, 86, 124
vector, 331, 333
[], 334
at, 334
begin, 338
clear, 334
empty, 334
end, 338
erase, 338
insert, 338
pop_back, 334
push_back, 334
rbegin, 338
rend, 338
resize, 334, 337
size, 334
version control, 371
virtual, 311, 315
virtual members, 311, 315, 317, 320
visibility, 152
default, 137
private, 259
protected, 259
public, 137, 259
visitor, 438
void, 158
void *, 204
volume testing, 445
walkthrough, 441
waterfall, 406
wchar_t, 76
which, 17
while, 60, 102
white-box testing, 442
whitespace, 72, 171, 185
widening, 90
wildcard, 7, 31
qualifier, 31
working copy, 372
wrapper, 423
wrapper member, 308
write, 32
xterm, 2, 4
zero-filled, 155