

Writing Your First Programs Chapter 2

Mr. Dave Clausen

La Cañada High School

<http://www.lcusd.net/lchs/dclausen/>

Program Development

Top Down Design

- Planning is a critical issue
 - Don't type in code “off the top of your head”
- Programming Takes Time
 - Plan on writing several revisions
 - Debugging your program
- Programming requires precision
 - One misplaced semi-colon will stop the program

Exercise in Frustration

- Plan well (using paper and pencil)
- Start early
- Be patient
- Handle Frustration
- Work Hard
- Don't let someone else do part of the program for you. Understand the Concepts Yourself!



Six Steps To Good Programming Habits #1

- 1. Analyze the Problem
 - Formulate a clear and precise statement of what is to be done.
 - Know what data are available
 - Know what may be assumed
 - Know what output is desired & the form it should take
 - Divide the problem into subproblems

Six Steps To Good Programming Habits #2

● 2. Develop an Algorithm

– Algorithm:

- a finite sequence of *effective* statements that when applied to the problem, will solve it.

– Effective Statement:

- a clear unambiguous instruction that can be carried out.

– Algorithms should have:

- a specific beginning and ending that is reached in a reasonable amount of time (Finite amount of time).



Six Steps To Good Programming Habits #3

- 3. Document the Program
 - Programming Style
 - Comments
 - Descriptive Variable Names
 - Pre & Post Conditions
 - Output



Six Steps To Good Programming Habits #4-5

- 4. Code the Program
 - After algorithms are correct
 - Desk check your program
- 5. Run the Program
 - Syntax Errors (semi colon missing, etc.)
 - Logic Errors (divide by zero, etc.)



Six Steps To Good Programming Habits

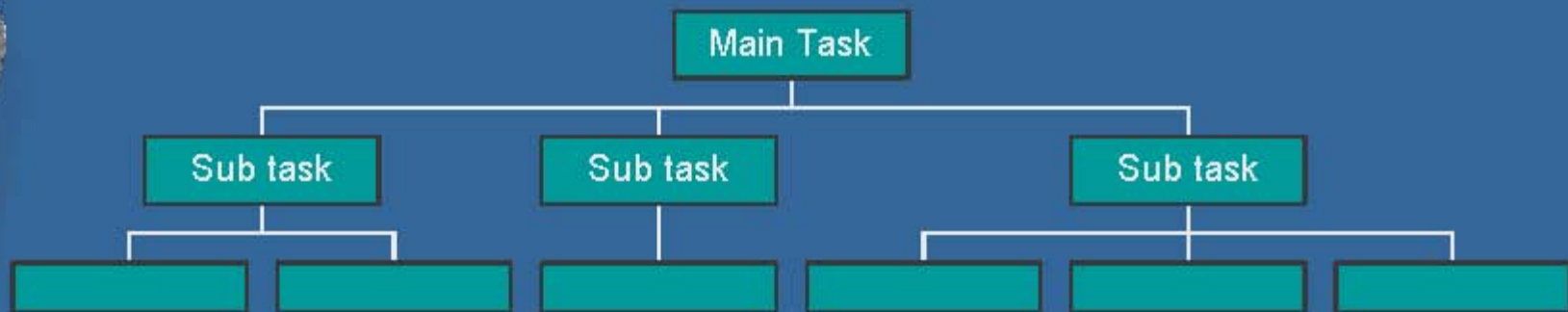
- 6. Test the Results
 - Does it produce the correct solution?
 - Check results with paper and pencil.
 - Does it work for all cases?
 - Border, Edge, Extreme Cases
 - Revise the program if not correct.

Top Down Design

- Subdivide the problem into major tasks
 - Subdivide each major task into smaller tasks
 - Keep subdividing until each task is easily solved.
- Each subdivision is called stepwise refinement.
- Each task is called a module
- We can use a structure chart to show relationships between modules.

Top Down Design

Structure Chart



Top Down Design

- Pseudocode
 - is written in English with C++ like sentence structure and indentations.
 - Major Tasks are numbered with whole numbers
 - Subtasks use decimal points for outline.

Pseudocode

1. Get Information
 - 1.1. Get starting balance
 - 1.2. Get transaction type
 - 1.3. Get transaction amount
2. Perform computations
 - 2.1. If deposit then
 add to balance
 - Else
 subtract from balance
3. Display the results
 - 3.1. Display starting balance
 - 3.2. Display transaction
 - 3.2.1. Display transaction type
 - 3.2.2. Display transaction amount
 - 3.3. Display ending balance

[Checkbook.cpp](#)

Writing Programs

- C++ Vocabulary
 - reserved words
 - have a predefined meaning that can't be changed
 - library identifiers
 - words defined in standard C++ libraries
 - programmer supplied identifiers
 - defined by the programmer following a well defined set of rules

Writing Programs

- Words are CaSe SeNsItIvE
 - For constants use ALL CAPS (UPPERCASE)
 - For reserved words and identifiers use lowercase
- Syntax
 - rules for construction of valid statements, including
 - order of words
 - punctuation

Library Identifiers

- Predefined words whose meanings could be changed.
- Examples:
 - `iostream`
 - `cin` `cout`
 - `iomanip`
 - `setprecision` `setw`
 - `math`
 - `pow` `sin` `sqrt`

Identifiers

- Must start with a letter of the alphabet or underscore `_` (we will not use underscores to start identifiers)
- length determined by compiler
 - Turbo C++ Win 4.5 32 characters
 - Borland C++ unlimited
 - Codewarrior 255 characters
 - aim for 8 to 15 characters
- common use is to name variables & constants

Basic Program Components

- Comments
- Preprocessor Directives
- Constant Declaration Section
- Type Declaration Section
- Function Declarations
- Main Program Heading: `int main()`
 - Declaration Section (eg. variables)
 - Statement Section



A Sample Program reserved words

[Reswords.doc](#)

[Reswords.cpp](#)

Writing Code in C++

- Executable Statement
 - basic unit of grammar
 - library identifiers, programmer defined identifiers, reserved words, numbers and/or characters
 - A semicolon almost always terminates a statement
 - usually not needed AFTER a right curly brace }
 - Exception: declaring user defined types.
- Programs should be readable

[noformat.cpp](#)

[format.cpp](#)

Simple Data Types

- Type int
 - represent integers or whole numbers
 - Some rules to follow:
 - Plus signs do not need to be written before the number
 - Minus signs must be written when using negative #'s
 - Decimal points cannot be used
 - Commas cannot be used
 - A comma is a character and will “crash” your program, no joke.
 - Leading zeros should be avoided (octal or base 8 #'s)
 - `limits.h` `int_max` `int_min`

Simple Data Types

- Type double
 - used to represent real numbers
 - many programmers use type float, the AP Board likes the extra precision of double
 - avoid leading zeros, trailing zeros are ignored
 - limits.h, float.h
 - `dbl_max`, `dbl_min`, `dbl_dig`

Simple Data Types

- Type char

- used to represent character data
 - a single character which includes a space
 - See Appendix 4 in our text
- must be enclosed in single quotes eg. ‘d’
- Escape sequences treated as single char
 - ‘\n’ newline
 - ‘\’ apostrophe
 - ‘\”’ double quote
 - ‘\t’ tab
 - ‘\\’ pathnames for files

Simple Data Types

● Strings

- used to represent textual information
- string constants must be enclosed in double quotation marks eg. “Hello world!”
 - empty string “”
 - new line char or string “\n”
 - “the word \”hello\”” (puts quotes around “hello”)
- String variables use:
`#include “apstring.cpp”`
 - use quotes for user supplied libraries

Output

- `#include <iostream.h>`
 - cout pronounced see-out
 - `cout << '\n';`
 - `cout << endl;`
 - `cout << "Hello world!";`
 - `cout << "Hello world!" << endl;`

[printadd.cpp](#)

Formatting Integers

- `#include <iomanip.h>` (input/output manipulators)
- right justify output
 - `cout << setiosflags (ios::right);`
- specify field width
 - `cout << setw(10) << 100` (output: `*****100`, where `*` represents a space.)
- specify decimal precision
 - `cout<<setiosflags (ios::fixed | ios::showpoint | ios::right)<< setprecision (2);`

Setprecision

- Precision is set and will remain until the programmer specifies a new precision
 - The decimal uses one position
 - Trailing zeros are printed the specified number of places
 - Leading plus signs are omitted
 - Leading minus signs are printed and use 1 position
 - Digits are rounded, not truncated.

Test Programs

- Test programs are short programs written to provide an answer to a specific question.
- You can try something out
- Play with C++
- Ask “what if” questions
- Experiment: try and see