Appendix

A. Elements of Style

While the ultimate test of a program is how well it performs for the user, the value of the program is greatly limited if it is difficult to understand or update. For this reason, it is very important for programmers to write their code in the most clear and understandable way possible. We call this "programming style."

Elements of Style

Perhaps the easiest way to explain coding style is this: give the bugs no place to hide. When our variable names are clearly and precisely named, we are leaving little room for confusion or misinterpretation. When things are always used the same way, then readers of the code have less difficulty understanding what they mean.

There are four components to our style guidelines: variable and function names, spacing, function and program headers, general comments, and other standards.

Variable and function names

The definitions of terms and acronyms of a software program typically consist of variable declarations. While variables are declared in more than one location, the format should be the same. Using descriptive identifiers reduces or eliminates the need for explanatory comments. For example, `sum` tells us we are adding something; `sumOfSquares` tells us specifically what we are adding. Use of descriptive identifiers also reduces the need for comments in the body of the source code. For example, `sum += x * x;` requires explanation. On the other hand, `sumOfSquares += userInput * userInput;` not only tells us where the item we are squaring came from, but also that we are creating a sum of the squares of those items. If identifiers are chosen carefully, it is possible to write understandable code with very few, if any, comments. The following are the University conventions for variable and function names:

Identifier	Example	Explanation
Variable	`sumOfSquares`	Variables are nouns so it follows that variable names should be nouns also. All the words are TitleCased except the first word. We call this style camelCase.
Function	`displayDate()`	Functions are verbs so it follows that function names should also be verbs. Like variables, we camelCase functions.
Constant	`PI`	Constants, include `#defines`, are ALL_CAPS with an underscore between the words.
Data Types	`Date`	Classes, enumeration types, type-defs, and structures are TitleCased with the first letter of each word capitalized. These are CS 165 constructs.

Function and program headers

It takes quite a bit of work to figure out what a program or function is trying to do when all the reader has is the source code. We can simplify this process immensely by providing brief summaries. The two most common places to do this are in function and program headers.

A function header appears immediately before every function. The purpose is to describe what the program does, any assumptions made about the input parameters, and describe the output. Ideally, a programmer should need no more information than is provided in the header before using a function. An example of a function header is the following:

```
/******************************************************
 * GET YEAR
 * Prompt the user for the current year. Error checking
 * will be performed to ensure the year is valid
 *    INPUT:  None (provided by the user)
 *    OUTPUT: year
 *****************************************************/
```

A program header appears at the beginning of every file. This identifies what the program does, who wrote it, and a brief description of what it was written for. Our submission program reads this program header to determine how it is to be turned in. For this reason, it is important to start every program with the template provided at `/home/cs124/template.cpp`. The header for Assignment 1.0 is:

```
/***********************************************************************
* Program:
*    Assignment 10, Hello World
*    Brother Helfrich, CS124
* Author:
*    Sam Student
* Summary:
*    Display the text “Hello world” on the screen.
*    Estimated:  0.7 hrs
*    Actual:     0.5 hrs
*      I had a hard time using emacs.
************************************************************************/
```

General Comments

We put comments in our code for several reasons:

- To describe what the next few lines of code do
- To explain to the reader of the code why code was written a certain way
- To write a note to ourselves reminding us of things that still need to be done
- To make the code easier to read and understand

Since a comment can be easily read by a programmer and source code, in many cases, must be decoded, one purpose of comments is to clarify complicated code. Comments can be used to convey information to those who will maintain the code. For example, a comment might provide warning that a certain value cannot be changed without impacting other portions of the program. Comments can provide documentation of the logic used in a program. Above all else, comments should add *value* to the code and should not simply restate what is obvious from the source code.

The following are meaningless comments and add no value to the source code:

```
int i; // declare i to be an integer
i = 2; // set i to 2
```

On the other hand, the following comments add value:

```
int i; // indexing variable for loops
i = 2; // skip cases 0 and 1 in the loop since they were processed earlier
```

With few exceptions, we use line comments (`//`) rather than block comments (`/* … */`) inside functions. Please add just enough comments to make your code more readable, but not so many that it is overly verbose. There is no hard-and-fast rule here.

"Commenting out" portions of the source code can be an effective debugging technique. However, these sections can be confusing to those who read the source code. The final version of the program should not contain segments of code that have been commented out.

Spacing

During the lexing process, the compiler removes all the spaces between keywords (such as `int`, `for`, or `if`) and operators (such as `+` or `>=`). To make the code human-readable, it is necessary to introduce spaces in a consistent way. The following are the University conventions for spaces:

Rule	Example	Explanation
Operators	`tempC = 5.0 / 9.5 (tempF - 32.0)`	There needs to be one space between all operators, including arithmetic (`+` and `%`), assignment (`=` and `+=`) and comparison (`>=` and `!=`)

Indention	`{`
	`   int answer = 42;`
	`   if (answer > 100)`
	`      cout << "Wrong answer!";`
	`}`
		With every level of indention, add three white spaces. Do not use the tab character to indent.

Functions		Put one blank line between functions. More than one results in unnecessary scrolling, less feels cramped

Related code	`   // get the data`
	`   float income;`
	`   cout << "Enter income: ";`
	`   cin  >> income;`
		Much like an essay is sub-divided into paragraphs, a function can be sub-divided into related statements. Each statement should have a blank line separating them.

Other Standards

Because of the way printers and video displays handle text, readability is improved by keeping each line of code less than 80 characters long.

Subroutines and classes should be ordered in a program such that they are easy to locate by the reader of the source code. This usually means grouping functions that perform similar operations. For example, all input functions should be next to each other in a file, as should output functions.

Each curly brace should be on its own line; this makes them easier to match up.

Please make sure there are no spelling or grammatical errors in your source code.

Style Checklist

Comments

- program introductory comment block
- identify program
- identify instructor and class
- identify author
- brief explanation of the program
- brief explanation of each class
- brief explanation of each subroutine

Variable declarations

- declared on separate lines
- comments (if necessary)

Identifiers

- descriptive
- correct use of case
- correct use of underscores

White space

- white space around operators
- white space between subroutines
- white space after key words
- each curly brace on its own line

Indentation

- statements consistently indented
- block of code within another block of code further indented

General

- code appropriately commented
- each line less than 80 characters long
- correct spelling
- no unused (e.g. commented out) code

Examples

The following are two examples of programs with excellent programming style.

```
/***********************************************************************
* Program:
*    Homework 00, Add Integers
*    Brother Twitchell, CS 124
* Author:
*    Brother Twitchell
* Summary:
*    Demonstrates the amazing ability to add a positive integer and a
*    negative integer and to display the resulting sum.
***********************************************************************/

#include <iostream>
using namespace std;

/***********************************************************************
* Prompts the user for a positive and a negative integer.
* If required input is supplied, the two integers are added and the
* sum is displayed.
***********************************************************************/
int main()
{
   int positiveIntegerFromUser;
   int negativeIntegerFromUser;
   int sumOfIntegersFromUser;

   // Prompt the user for a number
   cout << "Enter a positive integer" << endl;
   cin  >> positiveIntegerFromUser;
   if (positiveIntegerFromUser > 0)
   {
      cout << "Enter a negative integer" << endl;
      cin  >> negativeIntegerFromUser;

      if (negativeIntegerFromUser < 0)
      {
         // amazing! we have both a positive and a negative integer
         // add them and output the results
         sumOfIntegersFromUser = positiveIntegerFromUser +
                                 negativeIntegerFromUser;
         cout << "The sum of " << positiveIntegerFromUser;
         cout << " and " << negativeIntegerFromUser;
         cout << " is " << sumOfIntegersFromUser << endl;
      }
      else
      {
         // while the user has demonstrated his/her ability to enter a
         //    positive integer, he/she failed to supply a negative
         //    integer; give up!
         cout << negativeIntegerFromUser << " is not negative" << endl;
         cout << "Next time please enter a number less than zero (0)." << endl;
         cout << "Program terminating." << endl;
      }
   }
   else
   {
      // the user has not supplied a positive integer; give up!
      cout << positiveIntegerFromUser << " is not positive" << endl;
      cout << "Next time please enter a number greater than zero (0)." << endl;
      cout << "Program terminating." << endl;
   }
   return 0;
}
```

Another example:

```
/***********************************************************************
* Program:
*    Homework 00, Cube a Number
*    Brother Twitchell, CS 124
* Author:
*    Brother Twitchell
* Summary:
*    This program reads a number from a text file, cubes the number,
*    and outputs the result.
***********************************************************************/

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

/***********************************************************************
* Returns the cube of the supplied integer value.
* Receives a pointer to the value to be cubed.
***********************************************************************/
int cubedInteger(int number)
{
   // return the cube the supplied value
   return (number * number * number);
}

/***********************************************************************
* Opens the input file and reads the number to be cubed. Outputs the
* original and cubed values. Closes the input file.
***********************************************************************/
int main()
{
   int numberFromFile = 0;

   // open the input file, read a single integer from it, and close it
   ifstream inputFile("number.txt" /*filename containing the number*/);

   // yes, yes, I know we are not testing to see if we succeeded!
   //      This is only a short demonstration program.
   inputFile >> numberFromFile;
   inputFile.close();

   // output the original value and its cube
   cout << "Impress your date!\n";
   cout << "The cube of "
        << numberFromFile
        << " is "
        << cubedInteger(numberFromFile)
        << "."
        << endl;
   return 0;
}
```

B. Order of Operations

The order of operations is the evaluation order for an expression. When parentheses are not included, the following table describes which order the compiler assumes you meant. Of course, it is always better to be explicit by including parentheses. Operators in rows of the same color have the same precedence.

Name	Operator	Example
Array indexing	`[]`	`array[4]`
Function call	`()`	`function()`
Postfix increment and decrement	`++  --`	`count++  count--`
Prefix increment and decrement	`++  --`	`++count  --count`
Not	`!`	`!married`
Negative	`-`	`-4`
Dereference	`*`	`*pValue`
Address-of	`&`	`&value`
Allocate with new	`new`	`new int`
Free with delete	`delete`	`delete pValue`
Casting	`()`	`(int)4.2`
Get size of	`sizeof`	`sizeof(int)`
Multiplication	`*`	`3 * 4`
Division	`/`	`3 / 4`
Modulus	`%`	`3 % 4`
Addition	`+`	`3 + 4`
Subtraction	`-`	`3 - 4`
Insertion	`<<`	`cout << value`
Extraction	`>>`	`cin  >> value`
Greater than, etc.	`>=  <=  >  <`	`3 >= 4`
Equal to, not equal to	`==  !=`	`3 != 4`
Logical And	`&&`	`passed && juniorStatus`
Logical OR	`||`	`passed || juniorStatus`
Assignment, etc.	`=  +=  *=  -=  /=  %=`	`value += 4`
Conditional expression	`? :`	`passed ? "happy" : "sad"`

C. Lab Help

> Behold, you have not understood; you have supposed that I would give it unto you, when you took no thought save it was to ask me.
>
> But, behold, I say unto you, that you must study it out in your mind ..."
>
> *D&C 9:7-8*

The Linux lab is staffed with lab assistants to provide support for students using the lab for computer science courses. Their major role is to provide assistance, support, advice, and recommendations to students. They are not to be considered a help desk where you bring a problem to them and expect them to solve it! As the semester progresses, lab assistants become increasingly busy and are not able to provide as much tutoring. Those struggling with a class or those that have been away from programming for a significant period (e.g. mission) are encouraged to sign up for class tutors from the tutoring center.

Lab assistants can be very useful when you have encountered a brick wall and just don't know how to proceed. They can help get you going again! You must put forth effort to complete your homework assignment. Do not expect lab assistants to give you answers or source code for your specific assignments. Instead, expect questions to guide you to a solution. For example they might say, "Have you tried using a while loop?", or "Are you sure you haven't exceeded your array bounds?", or "Check the syntax of your switch statement," or "Try putting some debug statements in your code *here* to see what is happening." With this kind of help you will understand what you have done wrong, you will be less likely to make this mistake in the future, and if this mistake is made in the future, you will be in a better situation to solve the problem without assistance. As a general rule, lab assistants are *not* allowed to touch the keyboard.

Lab assistants have been hired to help all students with general questions regarding use of the computers in the Linux lab. This includes, but is not limited to, difficulties with text editors, `submit`, `styleChecker`, `testBed`, `svn`, `PuTTY`, `winscp`, and simple operating system issues. CS 124 and CS 165 students should expect appropriate assistance from lab assistants. CS 235 students, particularly those who have been away for a couple of years, should also expect appropriate assistance from lab assistants during the first few weeks of the semester. Generally speaking, however, students should have learned how to help themselves and resolve their own problems by the time they have completed CS 235. CS 213 students may expect some help from the lab assistants.

Lab assistants are expected to give first priority to CS 124 and CS 165 students. Students in upper-division classes should not expect assistance from lab assistants.

To more effectively respond to questions, a "Now Serving" system has been setup in the lab which allows a student to click on an icon and it automatically places a request for help in a queue. The lab assistants monitor the queue and help the next student in the queue. It's like taking a number and waiting for your number to be called. However, instead of calling your number they just come to your machine. To use the "Now Serving" system you will need to ask the lab assistants to set it up for you the first time. Once initially setup an icon will be visible each time you login then you simply click on the Icon to make a request for help. You won't need to keep holding your hand up waiting for a lab assistant to see you, and it makes sure that you get help in the proper order. Please ask the lab assistants to show you how to use the "Now Serving" software so they can serve you better. Students in upper-division classes should not expect assistance from lab assistants

Appendix

D. Emacs & Linux Cheat-Sheet

The emacs editor and the Linux system are most effectively used if commonly-used commands are memorized. The following are the commands used most frequently for CS 124:

Common Emacs Commands

`C-a`	Beginning of line
`C-e`	End of line
`C-k`	Kill line. Also puts the line in the buffer
`C-_`	Undo
`C-x C-f`	Load a new file or an existing file. The name will be specified in the window below
`C-x 2`	Split the window into two
`C-x 1`	Go back to one window mode
`C-x ^`	Enlarge window
`C-x o`	Switch to another window
`Alt-x shell`	Run the shell in the current window
`Alt-x goto-line`	Goto a given line
`C-x C-c`	Save buffer and kill Emacs
`C-x C-s`	Save buffer
`C-space`	Set mark
`C-w`	Cut from the cursor to the mark
`Alt-w`	Copy from the cursor to the mark
`C-y`	Paste from the buffer

Common Linux Commands

`cd`	Change Directory, move from the current directory to another
`ls`	List information about file(s)
`ll`	More verbose version of ls
`mkdir`	Create new folder(s)
`mv`	Move or rename files or directories
`cp`	Copy a file from one location to another
`rm`	Remove files
`rmdir`	Remove folder(s)
`pwd`	Print Working Directory
`cat`	Display the contents of a file to the screen
`more`	Display output one screen at a time
`clear`	Clear terminal screen
`exit`	Exit the shell
`fgrep`	Search file for lines that match a string
`kill`	Stop a process from running
`man`	Review the help page of a given command
`yppasswd`	Modify a user password
`emacs`	Common code editor
`vi`	More primitive but ubiquitous editor
`nano`	Another editor
`g++`	Compile a C++ program
`styleChecker`	Run the style checker on a file
`testBed`	Run the test bed on a file
`submit`	Turn in a file

E. C++ Syntax Reference Guide

Name	Syntax	Example

Pre-processor directives	#include *<libraryName>*
	#define *<MACRO_NAME> <expansion>*
		#include <iostream> // for CIN & COUT
		#include <iomanip> // for setw()
		#include <fstream> // for IFSTREAM
		#include <string> // for STRING
		#include <cctype> // for ISUPPER
		#include <cstring> // for STRLEN
		#include <cstdlib> // for ATOF
		
		#define PI 3.14159
		#define LANGUAGE "C++"

Function	*<ReturnType> <functionName>*(*<params>*)
	{
	   <statements>
	   return *<value>*;
	}
		int main()
		{
		   cout << "Hello world\n";
		   return 0;
		}

Function parameters	*<DataType> <passByValueVariable>*,
	<DataType> & *<passByReferenceVariable>*,
	const *<DataType> <CONSTANT_VARIABLE>*,
	<BaseType> <arrayVariable>[]
		void function(int value,
		              int &reference,
		              const int CONSTANT,
		              int array[])
		{
		}

COUT	cout << *<expression>* << … ;	cout << "Text in quotes"
		     << 6 * 7
		     << getNumber()
		     << endl;

Formatting output for money	cout.setf(ios::fixed);
	cout.setf(ios::showpoint);
	cout.precision(*<integerExpression>*);
		cout.setf(ios::fixed);
		cout.setf(ios::showpoint);
		cout.precision(2);

Declaring variables	*<DataType> <variableName>*;
	<DataType> <variableName> = *<init>*;
	const *<DataType> <VARIABLE_NAME>*;
		int integerValue;
		float realNumber = 3.14159;
		const char LETTER_GRADE = 'A';

CIN	cin >> *<variableName>*;	cin >> variableName;

IF statement	if (*<Boolean-expression>*)
	{
	   <statements>
	}
	else
	{
	   <statements>
	}
		if (grade >= 70.0)
		   cout << "Great job!\n";
		else
		{
		   cout << "Try again.\n";
		   pass = false;
		}

Asserts	assert(*<Boolean-expression>*);	#include <cassert> // at top of file
		{
		   assert(gpa >= 0.0);
		}

Name	Syntax	Example

FOR loop	`for (<initialization statement>;`
	`     <Boolean-expression>;`
	`     <increment statement>)`
	`{`
	`   <statements>`
	`}`
		`for (int iList = 0;`
		`     iList < sizeList;`
		`     iList++)`
		`   cout << list[iList];`

WHILE loop	`while (<Boolean-expression>)`
	`{`
	`  <statements>`
	`}`
		`while (input <= 0)`
		`   cin  >> input;`

DO-WHILE Loop	`do`
	`{`
	`   <statements>`
	`}`
	`while (<Boolean-expression>);`
		`do`
		`   cin >> input;`
		`while (input <= 0);`

Read from File	`ifstream <streamVar>(<fileName>);`
	`if (<streamVar>.fail())`
	`{`
	`   <statements>`
	`}`
	
	`<streamVar> >> <variableName>;`
	
	`<streamVar>.close();`
		`#include <fstream> // at top of file`
		`{`
		`   ifstream fin("data.txt");`
		`   if (fin.fail())`
		`      return false;`
		
		`   fin >> value;`
		
		`   fin.close();`
		`}`

Write to File	`ofstream <streamVar>(<fileName>);`
	`if (<streamVar>.fail())`
	`{`
	`   <statements>;`
	`}`
	
	`<streamVar> << <expression>;`
	
	`<streamVar>.close();`
		`#include <fstream> // at top of file`
		`{`
		`   ofstream fout("data.txt");`
		`   if (fout.fail())`
		`      return false;`
		
		`   fout << value << endl;`
		
		`   fout.close();`
		`}`

Fill an array	`<BaseType> <arrayName>[<size>];`
	`<BaseType> <arrayName>[] =`
	`    { <CONST_1>, <CONST_2>, … };`
	
	`for (int i = 0; i < <size>; i++)`
	`   <arrayName>[i] = <expression>;`
		`int grades[10];`
		`for (int i = 0; i < 10; i++)`
		`{`
		`   cout << "Grade " << i + 1 << ": ";`
		`   cin  >> grades[i];`
		`}`

C-Strings	`char <stringName>[<size>];`
	`cin >> <stringName>;`
	`for (char * <ptrName> = <stringName>;`
	`     *<ptrName>;`
	`     <ptrName>++)`
	`   cout << *<ptrName>;`
		`char firstName[256];`
		`cin >> firstName;`
		`for (char * p = firstName; *p; p++)`
		`   cout << *p;`

String Class	`string <stringName>;`
	`cin  >> <stringName>;`
	`cout << <stringName>;`
	`getline(<streamName>, <stringName>);`
	
	`if (<stringName1> == <stringName2>)`
	`   <statemement>;`
	
	`<stringName1> += <stringName2>;`
	`<stringName1> =  <stringName2>;`
		`string string1;          // declare`
		`string string2 = "124";   // initialize`
		
		`cin >> string1;          // input`
		`getline(cin, string2);   // getline`
		`if (string1 == string2)  // compare`
		`   string1 += string2;   // append`
		`string2 = string1;       // copy`

Name	Syntax	Example

Switch	switch (<integer-expression>)
	{
	   case <integer-constant>:
	      <statements>
	      break; // optional
	   …
	   default: // optional
	      <statements>
	}
		switch (value)
		{
		   case 3:
		      cout << "Three";
		      break;
		   case 2:
		      cout << "Two";
		      break;
		   case 1:
		      cout << "One";
		      break;
		   default:
		      cout << "None!";
		}

Conditional Expression	<Boolean-expression> ? <expression> :
	                        <expression>
		cout << "Hello, "
		     << (isMale ? "Mr. " : "Mrs. ")
		     << lastName;

Multi-dimensional array	<BaseType> <arrayName>[<SIZE>][<SIZE>];
	<BaseType> <arrayName>[][<SIZE>] =
	   {
	     { <CONST_0_0>, <CONST_0_1>, … },
	     { <CONST_1_0>, <CONST_1_1>, … },
	       …
	   };
	
	<arrayName>[<index>][<index>] =
	    <expression>;
		int board[3][3];
		
		for (int row = 0; row < 3; row++)
		   for (int col = 0; col < 3; col++)
		     board[row][col] = 10;

Allocate memory	<ptr> = new(nothrow) <DataType>;
	<ptr> = new(nothrow> <DataType>(<init>);
	<ptr> = new(nothrow) <BaseType>[<SIZE>];
		float * p1 = new(nothrow) float;
		int * p2 = new(nothrow) int(42);
		char * text = new(nothrow) char[256];

Free memory	delete <pointer>; // one value
	delete [] <arrayPointer>; // an array
		delete pNumber;
		delete [] text;

Command line parameters	int main(int <countVariable>,
	         char **<arrayVariable>)
	{
	}
		int main(int argc, char ** argv)
		{
		}

Library	Function Prototype

#include <cctype>	bool isalpha(char); // is the character an alpha ('a' - 'z' or 'A' - 'Z')?
	bool isdigit(char); // is the character a number ('0' - '9')?
	bool isspace(char); // is the character a space (' ' or '\t' or '\n')?
	bool ispunct(char); // is the character a symbol such as %#$!-_*@.,?
	bool isupper(char); // is the character uppercase ('A' - 'Z')?
	bool islower(char); // is the character lowercase ('a' - 'z')?
	int toupper(char); // convert lowercase character to uppercase. Rest unchanged
	int tolower(char); // convert uppercase character to lowercase. Rest unchanged

#include <cstring>	int strlen(const char *); // find the length of a c-string
	int strcmp(const char *, const char *); // 0 if the two strings are the same
	char * strcpy(char *<dest>, const char *<src>); // copies src onto dest

#include <cstdlib>	double atof(const char *); // parses input for a floating point number and returns it
	int atoi(const char *); // parses input for an integer number and returns it

F. Glossary

`#define`

A `#define` (pronounced “pound define”) is a mechanism to expand macros in a program. This macro expansion occurs before the program is compiled. The following example expands the macro `PI` into 3.1415

```
#define PI 3.1415
```

Chapter 2.1

`#ifdef`

The `#ifdef` macro (pronounced “if-deaf”) is a mechanism to conditionally include code in a program. If the condition is met (the referenced macro is defined), then the code is included.

```
#ifdef DEBUG
   cout << "I was here!\n";
#endif
```

Chapter 2.1

abstract

One of the three levels of understanding of an algorithm, abstract understanding is characterized by a grasp of how the parts or components of a program work together to produce a given output.

Chapter 2.4

address-of operator

The address-of operator (`&`) yields the address of a variable in memory. It is possible to use the address-of operator in front of any variable.

```
{
   int variable;
   cout << "The address of 'variable' is "
        << &variable;
}
```

Chapter 3.3

ALU

Arithmetic Logic Unit. This is the part of a CPU which performs simple mathematical operations (such as addition and division) and logical operations (such as `OR` and `NOT`)

Chapter 0.2

`argc`

When setting up a program to accept input parameters from the command line, argc is the traditional name for the number of items or parameters in the jagged array of passed data. The name “`argc`” refers to “count of arguments.”

```
int main(int argc,         // count of parameters
         char ** argv);
```

Chapter 4.3

`argv`

When setting up a program to accept input parameters from the command line, `argv` is the traditional name for the jagged array containing the passed data. The name “`argv`” refers to “argument vector” or “list of unknown arguments.”

```
int main(int argc,
         char ** argv);    // array of parameters
```

Chapter 4.3

array

An array is a data-structure containing multiple instances of the same item. In other words, it is a "bucket of variables." Arrays have two properties: all instances in the collection are the same data-type and each instance can be referenced by index (not by name).

Chapter 3.0
Chapter 3.1

```
{
   int array[4];          // a list of four integers
   array[2] = 42;         // the 3rd member of the list
}
```

assembly

Assembly is a computer language similar to machine language. It is a low-level language lacking any abstraction. The purpose of Assembly language is to make Machine language more readable. Examples of Assembly language include `LOAD M:3` and `ADD 1`.

Chapter 0.2

`assert`

An assert is a function that tests to see if a particular assumption is met. If the assumption is met, then no action is taken. If the assumption is not met, then an error message is thrown and the program is terminated. Asserts are designed to only throw in debug code. To turn off asserts for shipping code, compile with the `-DNDEBUG` switch.

Chapter 2.1

bitwise operator

A bitwise operator is an operator that works on the individual bits of a value or a variable.

Chapter 3.5

`bool`

A `bool` is a built-in datatype used to describe logical data. The name "`bool`" came from the father of logical data, George Boole.

Chapter 1.2
Chapter 1.5

```
bool isMarried = true;
```

Boolean operator

A Boolean operator is an operator that evaluates to a `bool` (`true` or `false`). For example, consider the expression (`value1 == value2`). Regardless of the data-type or value of `value1` and `value2`, the expression will always evaluate to `true` or `false`.

Chapter 1.5

data-driven

Data-driven design is a programming design pattern where most of the elements of the design are encoded in a data structure (typically an array) rather than in the algorithm. This allows a program to be modified without changing any of the code; only the data structure needs to be adjusted.

Chapter 3.1

`case`

A `case` label is part of a `switch` statement enumerating one of the options the program must select between.

Chapter 3.5

casting

The process of converting one data type (such as a `float`) into another (such as an `int`). For example, `(float)3` equals `3.0`.

Chapter 1.3

`char`

A `char` is a built-in datatype used to describe a character or glyph. The name "Char" came from "Character," being the most common use.

Chapter 1.2

```
char letterGrade = 'B';
```

comments
Comments are notes placed in a program not read by the compiler.
Chapter 0.2

compiler
A compiler is a program to translate code in a one language (say C++) into another (say machine language).
Chapter 1.0

compound statement
A compound statement is a collection of statements grouped with curly braces. The most common need for this is inside the body of an IF statement or in a loop.
Chapter 1.6

```
if (failed == true)
{                         // compound statement start
   cout << "Sorry!\n";    //   first statement
   return false;          //   second statement
}                         // compound statement end
```

cohesion
The measure of the internal strength of a module. In other words, how much a function does one thing and one thing only. The four levels of cohesion are: Strong, Extraneous, Partial, and Weak.
Chapter 2.0

coincidental
A measure of cohesion where items are in a module simply because they happen to fall together. There is no relationship.
Chapter 2.0

communicational
A measure of cohesion where all elements work on the same piece of data.
Chapter 2.0

conceptual
One of the three levels of understanding of an algorithm, conceptual understanding is characterized by a high level grasp of what the program is trying to accomplish. This does not imply an understanding of what the individual components do or even how the components work together to produce the solution.
Chapter 2.4

concrete
One of the three levels of understanding of an algorithm, concrete understanding is characterized by knowing what every step of an algorithm is doing. It does not imply an understanding of how the various steps contribute to the larger problem being solved. The desk check tool is designed to facilitate a concrete understanding of code.
Chapter 2.4

conditional expression
A conditional expression is a decision mechanism built into C++ allowing the programmer to choose between two expression, rather than two statements.
Chapter 3.5

```
cout << (grade >= 60.0 ? "pass" : "fail");
```

control
A measure of coupling where one module passes data to another that is interpreted as a command.
Chapter 2.0

counter-controlled

One of the three loop types, a counter-controlled loop keeps iterating a fixed number of types. Typically, this number is known when the loop begins. A counter-controlled loop has four components: the start, the end, the counter, and a loop body.

Chapter 2.5

coupling

Coupling is the measure of information interchange between functions. The seven levels of coupling are: Trivial, Encapsulated, Simple, Complex, Document, Interactive, and Superfluous.

Chapter 2.0

`cout`

COUT stands for Console OUTput. Technically speaking, `cout` a the destination or output stream. In other words, it in the following example, it is the destination where the insertion operator (<<) is sending data to. In this case, that destination is the screen.

```
cout << "Hello world!";
```

Chapter 1.1

CPU

Central Processing Unit. This is the part of a computer that interprets machine-language instructions

Chapter 0.2

c-string

A c-string is how strings are stored in C++: an array of characters terminated with a null (`'\0'`) character.

Chapter 3.2

data

A measure of coupling where the data passed between functions is very simple. This occurs when a single atomic data item is passed, or when highly cohesive data items are passed

Chapter 2.0

decoder

The instruction decoder is the part of the CPU which identifies the components of an instruction from a single machine language instruction.

Chapter 0.2

`default`

A `default` label is a special `case` label in a `switch` statement corresponding to the “unknown” or “not specified” condition. If none of the `case` labels match the value of the controlling expression, then the `default` label is chosen.

Chapter 3.5

`delete`

The `delete` operator serves to free memory previously allocated with `new`. When a variable is declared on the stack such as a local variable, this is unnecessary; the operating system deletes the memory for the user. However, when data is allocated with new, it is the programmer’s responsibility to delete his memory.

```
{
   int * pValue = new int;
   delete pValue;
}
```

Chapter 4.1

DeMorgan

Just as there are ways to manipulate algebraic equations using the associative and distributed properties, it is also possible to manipulate Boolean equations. One of these transformations is DeMorgan. A few DeMorgan equivalence relationships are:

```
!(p || q) == !p && !q
!(p && q) == !p || !q
a || (b && c) == (a || b) && (a || c)
a && (b || c) == (a && b) || (a && c)
```

Chapter 1.5

dereference operator

The dereference operator '*' will retrieve the data refered to by a pointer.

```
cout << "The data in the variable pValue is "
     << *pValue;
```

Chapter 3.3

desk check

A desk check is a technique used to predict the output of a given program. It accomplished by creating a table representing the value of the variables in the program. The columns represent the variables and the rows represent the value of the variables at various points in the program execution.

Chapter 2.4

`do … while`

One of the three types of loops, a DO-WHILE loop continues to execute as long as the condition in the Boolean expression evaluates to true. This is the same as a WHILE loop except the body of the loop is guaranteed to be executed at least once.

```
do
   cin >> gpa;
while (gpa > 4.0 || gpa < 0.0);
```

Chapter 2.3

`double`

A `double` is a built-in datatype use to describe large read numbers. The word "Double" comes from "Double-precision floating point number," indicating it is just like a `float` except it can represent a larger number more precisely.

```
double pi = 3.14159265359;
```

Chapter 1.2

driver

A driver is a program designed to test a given function. Usually a driver has a collection of simple prompts to feed input to the function being tested, and a simple output to display the return values of the function.

Chapter 2.1

dynamically-allocated array

A dynamically-allocated array is an array that is created at run-time rather than at compile time. Stack arrays have a size known at compile time. Dynamically-allocated arrays, otherwise known as heap arrays, can be specified at run-time.

```
{
   int * array = new int[size];
}
```

Chapter 4.1

`endl`

ENDL is short for "END of Line." It is one of the two ways to specify that the output stream (such as `cout`) will put a new line on the screen. The following example will put two blank lines on the screen:

```
cout << endl << endl;
```

Chapter 1.1

`eof`

When reading data from a file, one can detect if the end of the file is reached with the `eof()` function. Note that this will only return `true` if the end of file marker was reached in the last read.

```
if (fin.eof())
   cout << "The end of the file was reached\n";
```

Chapter 2.6

escape sequences
Escape sequences are simply indications to `cout` that the next character in the output stream is special. Some of the most common escape sequences are the newline (`\n`), the tab (`\t`), and the double-quote (`\"`)
Chapter 1.1

event-controlled
One of the three loop types, an event-controlled loop is a loop that keeps iterating until a given condition is met. This condition is called the event. There are two components to an event-controlled loop: the termination condition and the body of the loop.
Chapter 2.5

expression
A collections of values and operations that, when evaluated, result in a single value. For example, `3 * value` is an expression. If value is defined as `float value = 1.5;`, then the expression evaluates to `4.5`.
Chapter 1.3

external
A measure of coupling where two modules communicate through a global variable or another external communication avenue.
Chapter 2.0

extraction operator
The extraction operator (>>) is the operator that goes between `cin` and the variable receiving the user input. In the following example, the extraction operator is after the `cin`.
Chapter 1.2

```
cin >> data;
```

fetcher
The instruction fetcher is the part of the CPU which remembers which machine instruction is to be retrieved next. When the CPU is ready for another instruction, the fetcher issues a request to the memory interface for the next instruction.
Chapter 0.2

`for`
One of the three types of loops, the FOR loop is designed for counting. It contains fields for the three components of most counting problems: where to start (the Initialization section), where the end (the Boolean expression), and what to change with every count (the Increment section).
Chapter 2.3

```
for (int i = 0; i < num; i++)
   cout << array[i] << endl;
```

`fstream`
The `fstream` library contains tools enabling the programmer to read and write data to a file. The most important components of the `fstream` library are the `ifstream` and `ofstream` classes.
Chapter 2.6

```
#include <fstream>
```

function
One division of a program. Other names are sub-routine, sub-program, procedure, module, and method.
Chapter 1.4

functional
A measure of cohesion where every item in the function is related to a single task.
Chapter 2.0

`getline` The `getline()` method works with `cin` to get a whole line of user input. Chapter 1.2

```
char text[256];             // getline needs a string
cin.getline(text, 256);     // the size is a parameter
```

global variable A global variable is a variable defined outside a function. The scope extends to the bottom of the file, including any function that may be defined below the global. It is universally agreed that global variables are evil and should be avoided. Chapter 1.4

`ifstream` The `ifstream` class is part of the `fstream` library, enabling the programmer to write data to a file. IFSTREAM is short for "Input File STREAM." Chapter 2.6

```
#include <fstream>

{
   ifstream fin("file.txt");
   …
}
```

insertion operator The insertion operator (`<<`) is the operator that goes between `cout` and the data to be displayed. As we will learn in CS 165, the insertion operator is actually the function and `cout` is the destination of data. In the following example, the insertion operator is after the `cout`. Chapter 1.1

```
cout << "Hello world!";
```

instrumentation The process of adding counters or markers to code to determine the performance characteristics. The most common ways to instrument code is to track execution time (by noting start and completion time of a function), iterations (by noting how many times a given block of code has executed), and memory usage (by noting how much memory was allocated during execution). Chapter 4.4

`int` An `int` is a built-in datatype used to describe integral data. The word "Int" comes from "Integer" meaning "all whole numbers and their opposites." Chapter 1.2

```
int age = 19;
```

`iomanip` The IOMANIP library contains the `setw()` method, enabling a C++ program to right-align numbers. The programmer can request the IOMANIP library by putting the following code in the program: Chapter 1.1

```
#include <iomanip>
```

`iostream` The IOSTREAM library contains `cin` and `cout`, enabling a simple C++ program to display text on the screen and gather input from the keyboard. The programmer can request IOSTREAM by putting the following code in the program: Chapter 0.2

```
#include <iostream>
```

jagged array
A jagged array is a special type of multi-dimensional array where each row could be of a different size.
Chapter 4.3

lexer
The lexer is the part of the compiler to break a program into a list of tokens which will then be parsed.
Chapter 1.0

local variable
A local variable is a variable defined in a function. The scope of the variable is limited to the bounds of the function.
Chapter 1.4

logical
A level of cohesion where items are grouped in a module because they do the same kinds of things. What they operate on, however, is totally different.
Chapter 2.0

machine
Machine language is a computer language understandable by a CPU. It is language of the lowest abstraction. Machine language consists of noting but 1's and 0's.
Chapter 0.2

modularization
Modularization is the process of dividing a problem into separate tasks, each with a single purpose.
Chapter 2.0

modulus
The remainder from division. Consider 14 ÷ 3. The answer is 4 with a remainder of 2. Thus fourteen modulus 3 equals 2: 14 % 3 == 2
Chapter 1.3

multi-dimensional array
A multi-dimensional array is an array of arrays. Instead of accessing each member with a single index, more than one index is required. The following is a multi-dimensional array representing a tic-tac-toe board:

```
{
   char board[3][3];
}
```

Chapter 4.0

multi-way IF
Though an IF statement only allows the programmer to distinguish between at most two options, it is possible to specify more options through the use of more than one IF. This is called an multi-way IF.

```
if (grade >= 90.0)
   cout << "A";              // first condition
else if (grade >= 90.0)
   cout << "B";              // second condition
else
   cout << "not so good!";   // third condition
```

Chapter 1.6

nested statement
A nested statement is a statement inside the body of another statement. For example, an IF statement inside the body of another IF statement would be considered a nested IF.

```
if (grade >= 80.0)      // outer IF statement
   if (grade >= 90)     // nested IF statement
      cout << 'A';      // body of nested IF statement
   else
      cout << 'B';
```

Chapter 1.6

`new`

It is possible to allocate a block of memory with the `new` operator. This serves to issue a request to the operating system for more memory. It works with single items as well as arrays.

Chapter 4.1

```
{
   int * pValue = new int;       // one integer
   int * array = new int[10];    // ten integers
}
```

null

The null character, also known as a null terminator, is a special character marking the end of a c-string. The null character is represented as `'\0'`, which is always defined as zero.

Chapter 3.2

```
{
   char nullCharacter = '\0';  // 0x00
}
```

`NULL`

The `NULL` address corresponds to the zero address `0x00000000`. This address is guaranteed to be invalid, making it a convenient address to assign to a pointer when the pointer variable does not point to anything.

Chapter 4.1

```
{
   int * pValue = NULL;   // points to nothing
}
```

`ofstream`

The `ofstream` class is part of the `fstream` library, enabling the programmer to write data to a file. OFSTREAM is short for "Output File STREAM."

Chapter 2.6

```
#include <fstream>

{
   ofstream fout("file.txt");
   …
}
```

online desk check

An online desk check is a technique to gain an understanding of how data flows through an existing program. This is accomplished by putting COUT statements at strategic places in a program to display the value of key variables.

Chapter 2.4

parser

The parser is the part of the compiler understanding the syntax or grammar of the language. Knowing this, it is able to take all the components from the input language and place it into the format of the target or output language.

Chapter 1.0

Pascal-string

One of the two main implementations of strings, a Pascal-string is an array of characters where the length is stored in the first slot. This is not how strings are implemented in C++.

Chapter 3.2

pass-by-reference

Pass-by-reference, also known as "call-by-reference," is the process of sending a parameter to a function where the caller and the callee share the same variable. This means that changes made to the parameter in the callee will be reflected in the caller. You specify a pass-by-reference parameter with the ampersand `&`.

Chapter 1.4
Chapter 3.3

```
void passByReference(int &parameter);
```

pass-by-pointer

Pass-by-pointer, more accurately called “passing a pointer by value,” is the process of passing an address as a parameter to a function. This has much the same effect as pass-by-reference.

```
void passByPointer(int * pParameter);
```

Chapter 3.3

pass-by-value

Pass-by-value, also known as “call-by-value,” is the process of sending a parameter to a function where the caller and the callee have different versions of a variable. Data is sent one-way from the caller to the callee; no data is sent back to the caller through this mechanism. This is the default parameter passing convention in C++.

```
void passByValue(int parameter);
```

Chapter 1.4
Chapter 3.3

pointer

A pointer is a variable holding and address rather than data. A data variable, for example, may hold the value 3.14159. A pointer variable, on the other hand, will contain the address of some place in memory.

Chapter 3.3

procedural

A measure of cohesion where all related items must be performed in a certain order.

Chapter 2.0

prototype

A prototype is the name, parameter list, and return value of a function to be defined later in a file. The purpose of the prototype is to give the compiler “heads-up” as to which functions will be defined later in the file.

Chapter 1.4

pseudocode

Pseudocode is a high-level programming language designed to help people design programs. Though it has most of the elements of a language like C++, pseudocode cannot be compiled. An example of pseudocode is:

```
computeTithe(income)
  RETURN income ÷ 10
END
```

Chapter 2.2

register

The part of a CPU which stores short-term data for quick recall. A CPU typically has many registers.

Chapter 0.2

sentinel-controlled

One of the three loop types, a sentinel-controlled loop keeps iterating until a condition is met. This condition is controlled by a sentinel, a Boolean variable set by a potentially large number of divergent conditions.

Chapter 2.5

sequential

A measure of cohesion where operations in a module must occur in a certain order. Here operations depend on results generated from preceding operations

Chapter 2.0

scope

Scope is the context in which a given variable is available for use. This extends from the point where the variable is defined to the next closing braces `}`.

Chapter 1.4

`sizeof`

The `sizeof` function returns the number of bytes that a given datatype or variable requires in memory. This function is unique because it is evaluated at compile-time where all other functions are evaluated at run-time.

Chapter 1.2
Chapter 3.0

```
{
   int integerVariable;
   cout << sizeof(integerVariable) << endl;    // 4
   cout << sizeof(int)             << endl;    // 4
}
```

stack variable

A stack variable, otherwise known as a local variable, is a variable that is created by the compiler when it falls into scope and destroyed when it falls out of scope. The compiler manages the creation and destruction of stack variables wherease the programmer manages the createion and destruction of dynamically allocated (heap) variables.

Chapter 4.1

stamp

A measure of coupling where complex data or a collection of unrelated data items are passed between modules.

Chapter 2.0

string

A "string" is a computer representation of text. The term "string" is short for "an alpha-numeric string of characters." This implies one of the most important characteristics of a string: is a sequence of characters. In C++, a string is defined as an array of characters terminated with a null character.

Chapter 1.2
Chapter 3.2

```
{
   char text[256];   // a string of 255 characters
}
```

structure chart

A structure chart is a design tool representing the way functions call each other. It consists of three components: the name of the functions of a program, a line connecting functions indicating one function calls another, and the parameters that are passed between functions.

Chapter 2.0

`styleChecker`

`styleChecker` is a program that performs a first-pass check on a student's program to see if it conforms to the University style guide. The `styleChecker` should be run before every assignment submission.

Chapter 1.0
Appendix A

stub

A stub function is a placeholder for a function that is not written yet. The closest analogy is an outline in an essay: a placeholder for a chapter or paragraph to be written later. An example stub function is:

Chapter 2.1

```
void display(float value)
{
}
```

`submit`

`submit` is a program to send a student's file to the appropriate instructor. It works by reading the program header and, based on what is found, sending it to the instructor's class and assignment directory.

Chapter 1.0

`switch`

A `switch` statement is a mechanism built into most programming langauges allowing the programmer to specify between more than two options.

Chapter 3.5

tabs

The tab key on a traditional typewriter was invented to facilitate creating tabular data (hence the name). The tab character (`'\t'`) serves to move the cursor to the next tab stop. By default, that is the next 8 character increment.

```
cout << "\tTab";
```

Chapter 1.1

temporal

A measure of cohesion where items are grouped in a module because the items need to occur at nearly the same time. What they do or how they do it is not important

Chapter 2.0

`testBed`

`testBed` is a tool to compare a student's solution with the instructor's key. It works by compiling the student's assignment and running the program against a pre-specified set of input and output.

Chapter 1.0

variable

A variable is a named location where you store data. The name must be a legal C++ identifier (comprising of digits, letters, and the underscore _ but not starting with a digit) and conform to the University style guide (camelCase, descriptive, and usually a noun). The location is determined by the compiler, residing somewhere in memory.

Chapter 1.2

`while`

One of the three types of loops, a WHILE-loop continues to execute as long as the condition inside the Boolean expression is true.

```
while (grade < 70)
   grade = takeClassAgain();
```

Chapter 2.3

G. Index

Appendix