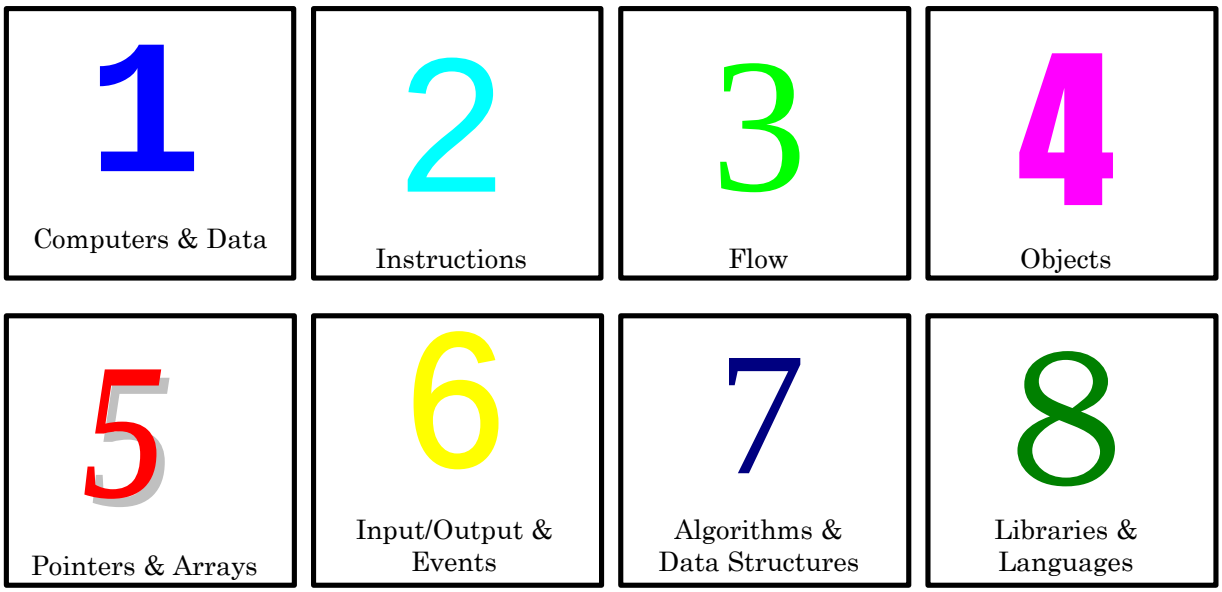


Introduction to Programming

UCSD Extension

Philip J. Mercurio
mercurio@acm.org

9th Edition

1

Introduction to Programming

Computers & Data

Session 1

Phil Mercurio
UCSD Extension
mercurio@acm.org

Today's Session

- Course Overview
- Introduction
- Computers
- Data

Course Overview: Readings

- *Rescued By C++ (3nd edition)*, Kris Jamsa, Jamsa Press, 1997. (*RBC++*)
- *How Computer Programming Works,* Daniel Appleman, APress, 2000 *(recommended)*

Session 1	Session 2	Session 3	Session 4
Computers & Data RBC 1-5	Instructions RBC 6	Flow RBC 8-10	Objects RBC 20, 23-28 *Midterm*

Session 5	Session 6	Session 7	Session 8
Pointers & Arrays RBC 18, 19, 22 *Midterm Due*	Input/Output & Events RBC 7	Algorithms – *Final*	Libraries & Languages – *Final Due*

When was the last time you gave someone directions?

- To get to UCSD Extension:
  - Take Hwy 5 North or South to Genesee
    - If you were on 5N, turn left
    - If you were on 5S, turn right
  - Follow Genesee around campus until you get to the light at Almahurst Drive
  - Turn left
  - Turn left again
  - Park

You're Already A Programmer

- These directions contain many of the elements of a computer program
- Anytime you give explicit instructions
  - Driving directions
  - A recipe
  - Setting the timer on a VCR
- … you're programming!

Languages

- Whenever we program, we give instructions in a language
- The thing being programmed has to understand the language
- In our driving directions, the language was English
- In recipes, we use a subset of English
  - We use abbreviations like **tsp** (teaspoon)
  - We use special verbs like **sauté**

1 Programming a Computer

- To learn how to program a computer, we have to learn one of the languages computers understand
- Human languages contain a lot of ambiguities
- Computer languages are very precise
  - Computers are very obedient, but very stupid
- To help us learn how to program, we'll study how a computer operates and how it deals with information
- We'll also study programming techniques that apply to any computer language

7 Session 1

1 Why C++?

- C is the most popular computer language in use today
  - Available on almost every computer
  - Contains concepts used in all programming languages
- C++ is an extension of C
  - All C programs are valid C++ programs
  - Not only did they extend the language, but they fixed some deficiencies

8 Session 1

1 Goals of this course

- Like any field of human study, programming can get very complex
  - But the basics of programming can be learned by anyone
- This is a survey course intended to teach people who can use a computer how to program
- Like *RBC++*, we'll use C++ to express the concepts we're studying
- We'll cover the first half of *RBC++*
- You'll be prepared to take a first-level course in any programming language

9 Session 1

1 What You'll Learn

- This course will cover the fundamentals of computer programming
- You'll learn many principles and techniques that will provide a basis for studying any language
- To make these ideas concrete, we'll also learn a subset of C++
  - Almost all of the C++ features we'll study are really part of C
  - We'll use C++-specific features that are simpler than their C counterparts

10 Session 1

1 After taking this course, you can:

- Stop here and use what you've learned
  - Good for managers and others that need to understand programming but not practice it
  - Understanding programming will make you a better user
- Continue studying on your own to become an amateur programmer
  - Completing *RBC++* will teach you enough of C++ to write programs recreationally
- Proceed with formal study with the goal of becoming a professional programmer
  - Take formal courses in C and then C++
  - Or study an entirely different language

11 Session 1

1 Computers Are Stupid

- Computers are mind-bogglingly stupid
- Hollywood portrays computers as intelligent
  - Even fiction set in current times exaggerates computer intelligence
- Users attribute intelligence and common sense to computers
  - Many new user difficulties are due to expecting too much
- In fact, computers can only react to situations that the programmer has anticipated

12 Session 1

1 Why Do Computers Appear Smart?

- A $50 chess program running on a $500 computer can beat most people
- Computers excel, not in intelligence, but in speed and accuracy
- The intelligence comes from the programmer, who figured out a precise set of instructions which, when followed accurately and quickly, plays a strong game of chess

1 Fundamental Rule of Programming

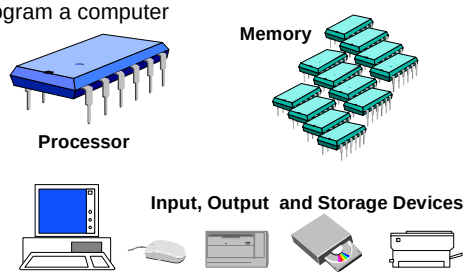
- **You can't program a computer to do something you don't know how to do!**
- However, just because you know how to do something doesn't mean you could do it
- Computers perform many tasks beyond the capabilities of humans, because they can process many more instructions per second, without errors

1 Hardware & Software

- Hardware is what you can see and touch, consisting of:
  - The Processor (or Central Processing Unit, CPU)
  - Memory
  - Input, Output, and Storage Devices
- Software can't be seen or touched, it's information
  - Stored on some media (disk, tape, etc.)
  - Loaded into your computer's memory for processing

1 Computer Hardware

- We don't need to know much about hardware to program a computer



1 The Processor

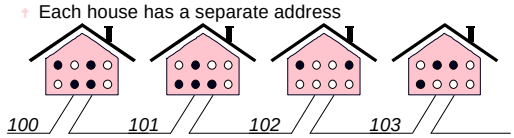
- The Processor (or Central Processing Unit, CPU) controls all of the data and performs all of the calculations
  - It usually consists of one chip (or a few chips)
  - Examples: Intel Pentium and 486, Motorola 68000 and PowerPC series
- The processor chip normally has a large number of pins, because it sends and receives electrical signals from many parts of the computer

1 Memory

- Memory is storage for data, usually lots of identical chips
- There are two types of memory:
  - RAM (Random-Access Memory) memory that can be read and changed by the processor
  - ROM (Read-Only Memory) memory that the processor can read but can't change
- Storage media like floppies, hard disks, CD-ROMs, etc. can be thought of as memory
  - But when we say "memory", we're usually talking about RAM

1 What Does Memory Look Like?

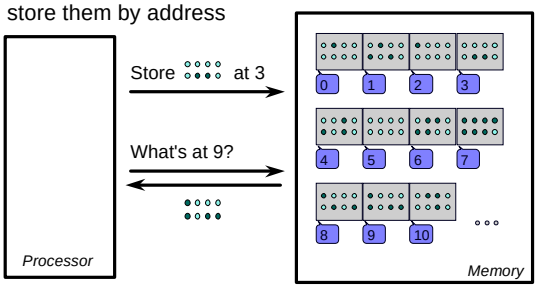
- RAM chips contain millions of switches, each capable of storing a bit (on or off)
- Bits are grouped by 8 into bytes
- All of your RAM put together can be thought of as a long street of byte-sized houses
  - Each house can contain one byte (8 bits)
  - Each house has a separate address



19 Session 1

1 The Processor and Memory

- The processor can retrieve bytes from memory and store them by address



20 Session 1

1 The Address Space

- A fundamental property of a processor is how large of an address it can represent at once
  - A 16-bit processor can address 2^{16} (~65,000) different locations
  - A 32-bit processor can address 2^{32} (~4 billion) different locations
- Most computers made today have a 32-bit address space
  - They could handle, theoretically, 4 gigabytes of RAM
  - More likely, your computer has 16 to 256 megabytes of RAM

21 Session 1

1 Devices

- Your computer has many devices for getting data in and out of memory
- Input devices: keyboard, mouse, sound card, scanner, network interface or modem
- Output devices: monitor, sound card, printer, network interface or modem
- Storage devices: ROM, hard disks, floppies, tape, CD-ROM, etc.

22 Session 1

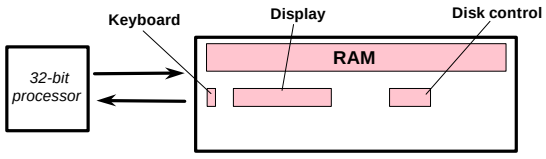
1 Memory Mapping: The Great Simplification

- Dealing with each device separately would require lots of connections from different parts of the computer to the Processor
- Instead, the computer hardware makes a device look like it's a set of locations in memory: the device is *mapped* onto memory
- For example, the processor might find out which key is typed by reading a certain location in memory
  - About 1 megabyte of memory stores the pixel values for your video display
- All the processor has to do is read and write locations in memory to control a device

23 Session 1

1 Our Model of a Computer

- We can choose to think of a computer as simply a processor and memory
  - This simplification covers everything programmers really need to know about hardware



4,294,967,296-byte Address Space

24 Session 1

1 Memory & Data

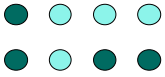
- What sorts of data can be stored in memory?
- All memory, ultimately, is on/off switches: bits
- We can choose to view a sequence of bits of a certain length as a single entity
- When we specify how many bits and how we're going to interpret them, we're defining a new type of data: a *data type*
- Underneath, every data type is still just bits
  - We can look at the same set of bits interpreted via different data types, if we choose to do so

1 Data Types

- True/False Values
- Numbers (many different kinds)
- Text
- Addresses
- Instructions
- Groups of different data types
- Lists of some data type

1 True/False (Boolean) Values

- The simplest data type is 1 bit, representing True if the bit is on, False if it is off
- George Boole, in 1854, developed the logic of true/false values, hence they're called Boolean values
- We can store up to 8 Booleans in one byte



=

False	True	True	True	
False	True	False	False	

- We might use the whole byte, in which case all off is **false**, any bits on is **true**

1 Numbers

- The most common type of data is numbers
- There are several different ways of using bits to represent numbers, depending on what kind of numbers you're dealing with:
  - Real numbers (any number, with or without a decimal point)
  - Rational numbers (one integer divided by another, a fraction)
  - Whole numbers (integers)
  - Non-negative whole numbers (zero + positive integers)
- There are also several different sizes of numbers
  - The most common are 8, 16, 32, or 64 bits

1 Bases

- We normally represent numbers in *decimal* (base 10)
  - What we call the "decimal point" is more appropriately called the "radix point"

4	0	6	2	.	1	2	5
x	x	x	x		x	x	x
10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}
||	||	||	||		||	||	||
4000 +	0 +	60 +	2 +		1/10 +	2/100 +	5/1000

1 Binary

- Computers store numbers in *binary* (base 2)

1	0	1	0	0	0	1	1	.
x	x	x	x	x	x	x	x	
2^7 (128)	2^6 (64)	2^5 (32)	2^4 (16)	2^3 (8)	2^2 (4)	2^1 (2)	2^0 (1)	
||	||	||	||	||	||	||	||	
128 +	0 +	32 +	0 +	0 +	0 +	2 +	1 =	

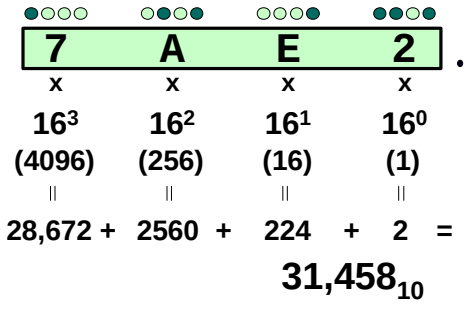
163_{10}

Hexadecimal

- Long sequences of 1s and 0s can be difficult to type or read
- Another handy base is *hexadecimal* (base 16)
- One *hex* digit represents exactly 4 bits (see table, following)
- We need symbols for 10_{10}, 11_{10}, 12_{10}, 13_{10}, 14_{10}, and 15_{10}, all of which are single digits in hex
- We'll use the first six letters of the alphabet, so the valid hex digit values are:

0_{16} 1_{16} 2_{16} 3_{16} 4_{16} 5_{16} 6_{16} 7_{16} 8_{16} 9_{16} A_{16} B_{16} C_{16} D_{16} E_{16} F_{16}

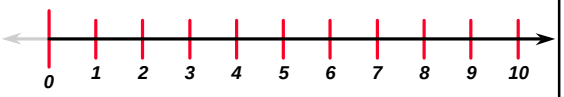
Hex



Handy Hex Table

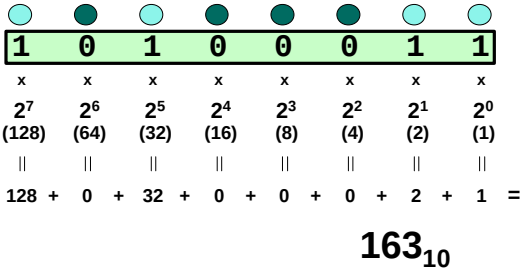
hex	binary	hex	binary	hex	binary	hex	binary
0	0000	1	0001	2	0010	3	0011
4	0100	5	0101	6	0110	7	0111
8	1000	9	1001	A	1010	B	1011
C	1100	D	1101	E	1110	F	1111

Unsigned Integers



- Unsigned Integers are used to store positive whole numbers and zero
- "unsigned" means no negative values allowed, the lowest value is 0
- We interpret the bits as the digits of a number in binary (base 2)
- If a bit is on, it represents 1, if off it represents 0

8-bit Unsigned Integer

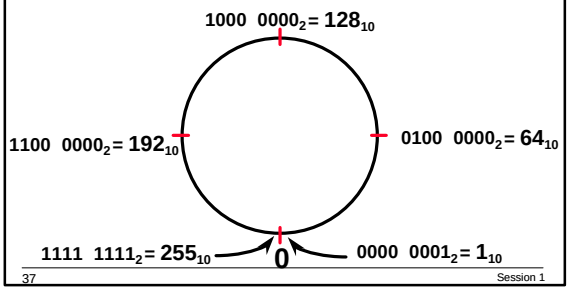


More 8-bit Unsigned Integers

- There are 2^8 (256) possible values for an 8-bit integer
- $0000\ 0000_2 = 0_{10}$
  - The smallest 8-bit unsigned integer
- $0000\ 0001_2 = 1_{10}$
  - The next 8-bit unsigned integer
- ...
- $1111\ 1110_2 = 254_{10} = 2^8\text{-}2$
  - The next-to-largest 8-bit unsigned integer
- $1111\ 1111_2 = 255_{10} = 2^8\text{-}1$
  - The largest 8-bit unsigned integer

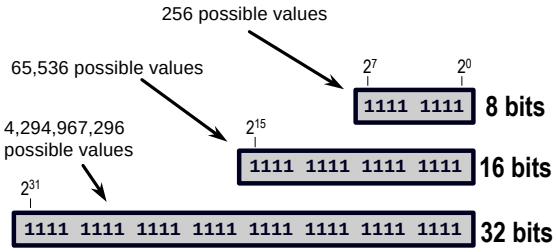
The Number Circle

- $1111\ 1111_2 + 0000\ 0001_2 = 0000\ 0000_2$
  - Computer integers are not infinite!



Bigger Numbers

- By using more bits, we can represent larger numbers



8-bit Signed Integers

- To represent both positive and negative numbers, let's take over 1 bit as a Boolean value
  - If it's True, the number is negative, if it's False the number is positive
- The remaining 7 bits store the number
  - If negative, we invert the bits (take the 1's complement)

0_{10}= 0000 0000
+000 0000

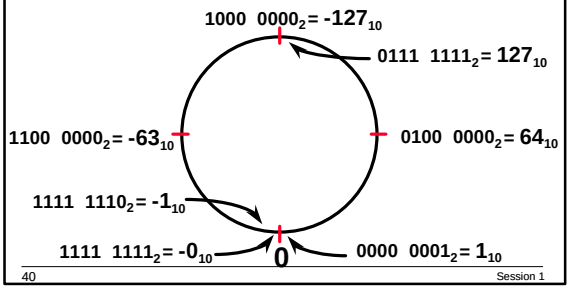
3_{10}= 0000 0011
+000 0011

-127_{10}= 1000 0000
-111 1111

-3_{10}= 1111 1100
-000 0011

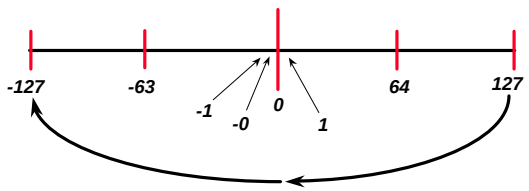
The Signed Number Circle

- This is why we take the 1s complement



...Becomes the Number Line

- By using the 1s complement for negative numbers, we keep the same order as the Number Line
- But what's with the -0?



Correct 8-bit Signed Integers

- Let's get rid of the -0 by adding 1 to all our negative numbers (this is called 2s complement)
- To make a negative number: invert the bits, add 1, set the sign bit to True

0_{10}= 0000 0000
+000 0000

3_{10}= 0000 0011
+000 0011

-1_{10}= 1111 1111
+1
-000 0001

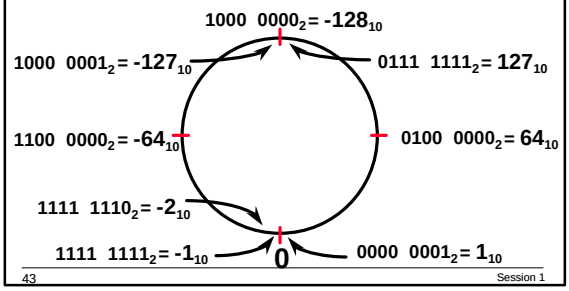
-3_{10}= 1111 1101
+1
-000 0011

-127_{10}= 1000 0001
+1
-111 1111

-128_{10}= 1000 0000
+1
-1000 0000

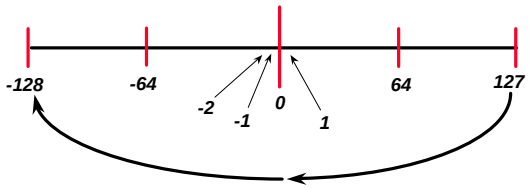
The Correct Signed Number Circle

- Notice that this appears more symmetric



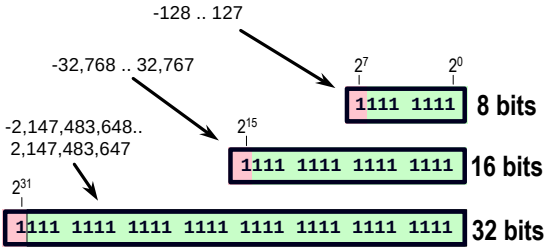
...Becomes the Correct Number Line

- Note that it's still not infinite!
- For 8-bit signed integers, 127 + 1 = -128



Bigger Signed Numbers

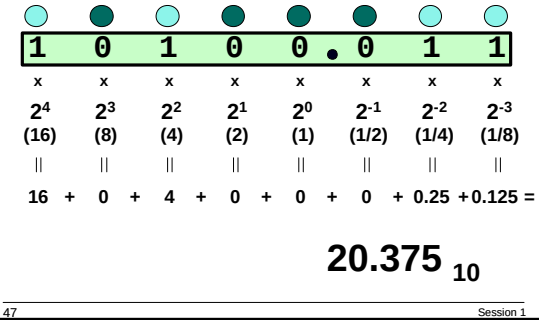
- By using more bits, we can represent larger numbers



Rational Numbers

- One way to represent rational numbers (numbers with a fractional component) using integers is to imagine that some of the digits are to the right of the radix point
- We do this all the time, in decimal, with percentages:
  - 1/2 = 0.50 = 50%
  - 5/4 = 1.25 = 125%
  - When we see %, we imagine a radix point 2 digits from the right
- Since the radix point never moves, we call this *fixed point* notation

8-bit Fixed Point



Too Fixed

- Fixed point numbers are too limiting
- We can borrow a solution from math: *Scientific Notation*
- Scientific Notation breaks a number down into two components:
  - *mantissa*: the digits of the number (-1.0 ?? mantissa ?? +1.0)
  - *exponent*: what power of 10 to multiply the mantissa by to get the number

0.986 x 10^2 = 98.6

mantissa exponent

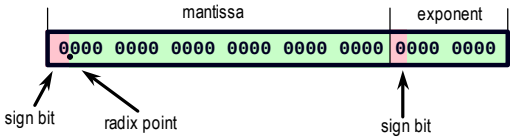
0.314159 x 10^1 = 3.14159

0.9296 x 10^8 = 92,960,000

-0.12 x 10^{-4} = -0.000012

1 Floating Point Numbers

- Most computers store rational numbers in *floating point* notation, breaking them down into 2 parts:
  - A group of bits representing a fixed-point number between -1 and 1 (the mantissa)
  - A group of bits representing a signed integer (the exponent)



49 Session 1

1 Floats and Doubles

- Floating point notation is actually more complicated (to make arithmetic easier)
- All we need to know is:
  - Precision: how many bits (or digits) in the mantissa
  - Range: the smallest and largest possible exponents
- There are usually two sizes of floating point numbers:

	size (bits)	mantissa (bits)	mantissa (digits)	smallest exponent	largest exponent
float	32	24	~7	-45	+38
double	64	53	~16	-324	+308

50 Session 1

1 Precision

- Note that a float can only store about 7 decimal digits.
- For example, the number 123,456,789 would be stored as 0.1234567×10^9 or 123,456,700
- If we used floats to compute the U.S. national debt, we'd only be accurate to within a few million dollars
- However, many computers can do arithmetic with floats faster than doubles
- You can trade speed for accuracy

51 Session 1

1 Real Numbers

- Even doubles are limited in what real numbers they can store
- Some real numbers, like π, are irrational and can't be represented exactly anyway
- Some numbers are repeating fractions, and can't be represented precisely in a finite number of digits
  - An example in decimal: 1/3 is the repeating decimal 0.333333333...
- Since mantissas are stored in binary, the set of fractions that repeat is different
  - $1/400 = 0.0025_{10} = 0.001010010100101\ldots_2$

52 Session 1

1 Text

- Modern computers probably process text as much as numbers
- We can use bits to represent text by establishing a code for individual characters
- There are about 100 characters on your keyboard, that would require 7 bits ($2^7 = 128$)
- Such a code already exists, it's called ASCII (American Standard Code for Information Interchange)
- We actually store ASCII characters in 8-bit bytes, the leftmost bit is always 0

53 Session 1

1 ASCII Table

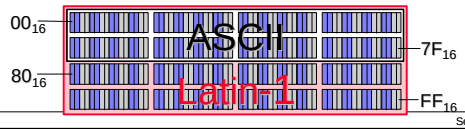
0000 0000

		0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111
0	000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	001	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	010		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	011	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	100	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	101	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	110	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	111	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

54 Session 1

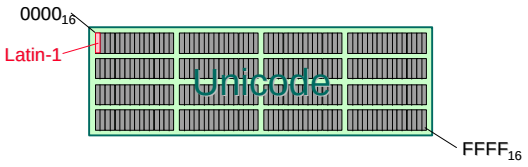
Latin-1

- 7-bit ASCII can only encode English text
  - It can't even handle all English words, e.g. résumé
- ISO Latin-1 uses 8 bits to encode most European languages
  - The first 128 codes (0_{16} to $7F_{16}$) are the same as in ASCII
  - The codes 80_{16} and above are accented letters: ç (c cedilla), è (e grave), ñ (n tilde), ü (u umlaut), etc., and additional symbols: § ¶ © £ ¢ ± ¿



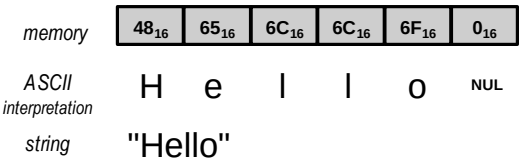
Unicode

- Unicode uses 16 bits to encode most major languages written today
  - The first 256 codes (0_{16} to FF_{16}) are the same as Latin-1
  - Can be extended to millions of codes, to handle any text, modern or historical



Strings

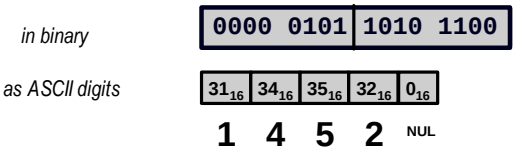
- Text is usually stored as strings, sequences of characters in adjacent bytes in memory
- In C, we mark the end of a string with an ASCII **NUL** ($0000\ 0000_2$)



Numbers and Strings

- There's a big difference between a number stored as a binary value vs. a string of digits

1,452



Addresses

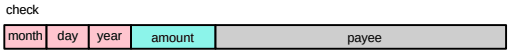
- Another important data type we'll need to store in memory is an *address* of a location in memory
- In most computers, an address will be an unsigned integer, with 0 referring to the first location in memory
  - Some computers represent addresses differently, but all have some way of storing addresses in memory
- A location in memory that stores an address is called a *pointer*
  - It, indirectly, points to another location

Representing Objects in Memory

- Every program has to deal with some objects, usually representing something in the real world
- A checkbook balancing program, for example, would need to store a representation of a check in memory
- Integers, floats, ASCII characters, and addresses are the *primitive* data types, not too useful by themselves
- To represent an object in memory, we'll need to use many primitives
  - For example, to represent a check, we might have 3 integers storing the date, a float storing the amount, a string for the payee, etc.

1 Groups and Lists

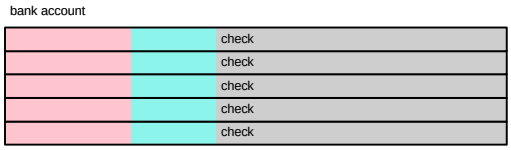
- There are two basic ways of organizing larger chunks of memory:
- Groups: combines primitives of different types
  - For example, 3 integers, a float, and a string to represent a check



61 Session 1

1 Groups and Lists (continued)

- Lists: sequences of one primitive type
  - Strings are lists of characters
- We can even get more complicated by having lists of lists, groups of groups, lists of groups, groups of lists, etc.



62 Session 1

1 So Far

- We've now seen almost all of the fundamental ways we'll be structuring memory
- Memory can also be viewed as coded instructions for the processor
- This is the topic of Session 2!

63 Session 1

2

Introduction to Programming

Instructions

Session 2

Phil Mercurio
UCSD Extension
mercurio@acm.org

2 Last Session

- Last session we began our model of a computer
- For our purposes, we can view a computer as a processor and memory
- We then discussed all the data types we're going to use to structure memory
  - Except for instructions

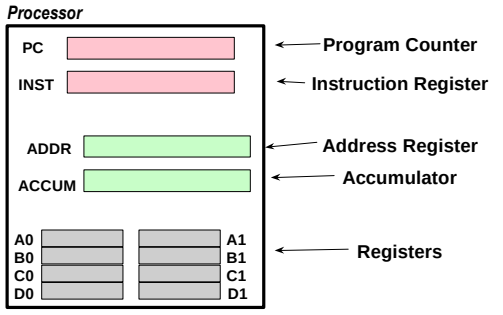
2 Today's Session

- Processor Instructions
- Languages
- C++

2 The Processor (A Simplified Model)

- The processor consists of millions of switches
- The switches are organized into circuits which perform arithmetic, communicate with the rest of the computer, and store values from memory
- Memory locations built into the processor are called *registers*
  - In our reference model, all registers are 32 bits (size of an address)
- We don't need to concern ourselves with most of the circuitry
- We only need to watch the contents of the registers

2 Processor Registers



2 Instructions

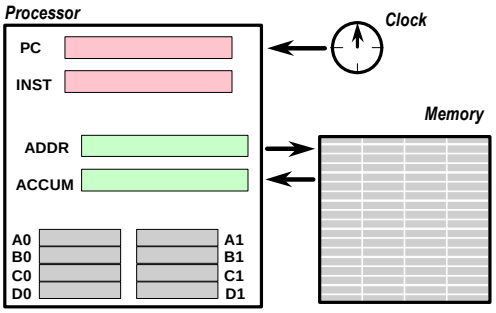
- An *instruction* is a coded 32-bit value
- When an instruction is placed in **INST** (the instruction register), the processor performs an operation
- Operations:
  - transfer data between memory and the registers
  - perform arithmetic on registers
  - make decisions based on register values
- An instruction consists of two parts:
  - **opcode**: operation code
  - **operand(s)**: depends on the opcode; might be a number, might refer to registers or a memory address, etc.

2 Processor's Environment

- The processor has output and input connections to memory
- The timing is controlled by the *clock*
- To start the processor, load the address of an instruction into the **PC** (program counter)
- Each cycle of the clock, the instruction stored at the address in **PC** is loaded into **INST** and the operation is performed
- Then the pointer in **PC** is advanced 4 bytes ahead to the next instruction in memory

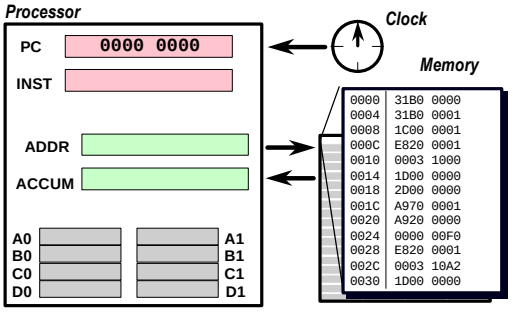
7 Session 2

2 Processor's Environment



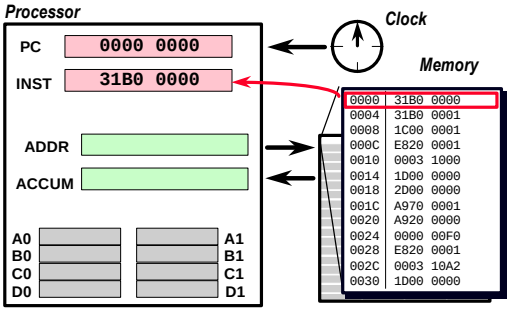
8 Session 2

2 Load Processor



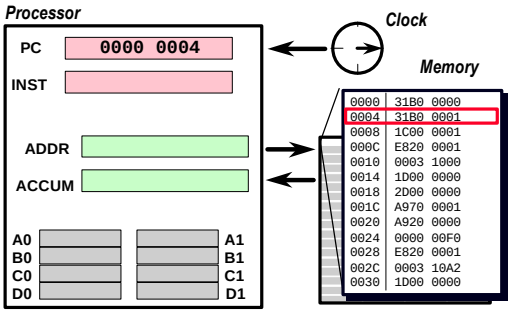
9 Session 2

2 Fetch Instruction



10 Session 2

2 Increment **PC**



11 Session 2

2 Instruction Set

- The set of all of the instructions understood by a particular processor is the *instruction set* or *machine language* of the processor
- Processors from different designers have different instruction sets, number of registers, maximum clock speeds, etc.
- We'll look at generic instructions that are common to most modern computers

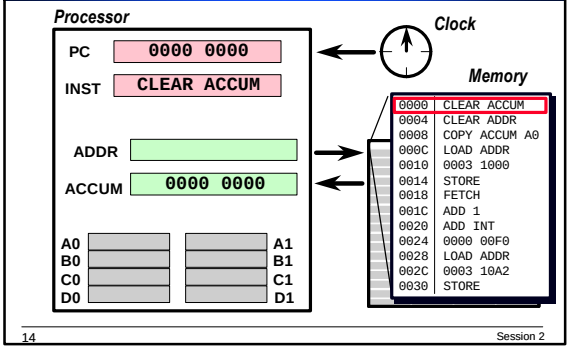
12 Session 2

2 Representing Instructions

- Instead of representing the opcode portion of an instruction in hex, we'll assign short names to each opcode
  - An opcode is still just a pattern of bits
- If the operand is a number or address, we'll show it in hex
- If the operand is a code for a register or pair of registers, we'll show the name of the register(s)

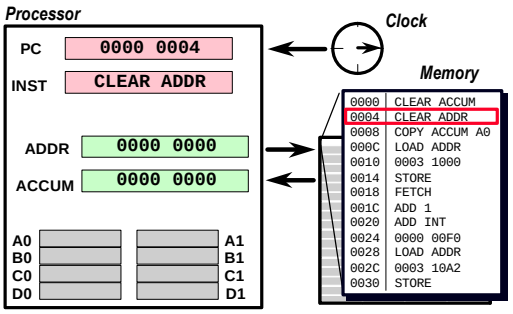
13 Session 2

2 CLEAR ACCUM



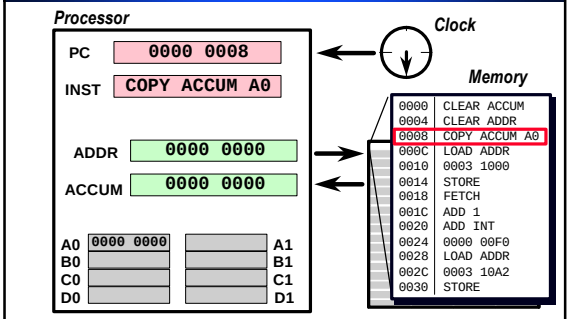
14 Session 2

2 CLEAR ADDR



15 Session 2

2 COPY ACCUM A0



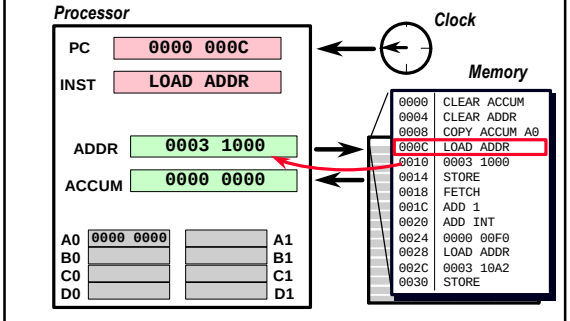
16 Session 2

2 Instructions and Data

- The simplest operations are those which clear registers, copy one register to another, etc.
  - Both the opcode and operand fit in one 32-bit chunk
- What if you need a bigger operand, like an address?
- There's a set of special opcodes which use the memory location(s) after the current instructions as data
- For example, LOAD ADDR
  - gets the value at the location after the current instruction
  - copies it to **ADDR**
  - increments **PC** by 4, *twice*

17 Session 2

2 LOAD ADDR

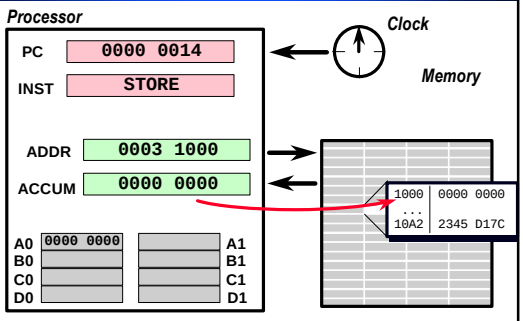


18 Session 2

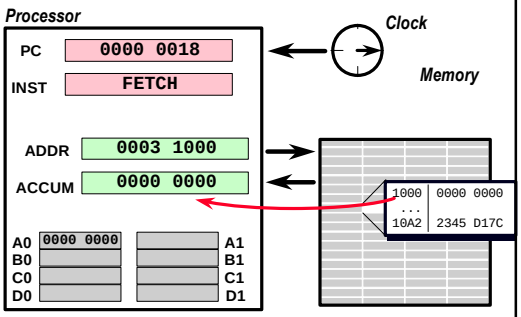
2 Data Transfer

- **ADDR** (the address register) and **ACCUM** (the accumulator) are used together to copy values into and out of memory
- STORE copies the value in **ACCUM** to the memory location in **ADDR**
- FETCH retrieves the value stored at the address in **ADDR** and copies it to **ACCUM**

2 STORE



2 FETCH



2 Programs

- A *program* is a sequence of instructions (interspersed with data)
- We've just written a program which clears some registers, stores a 0 in location 31000_{16}, and then retrieves that same number from memory

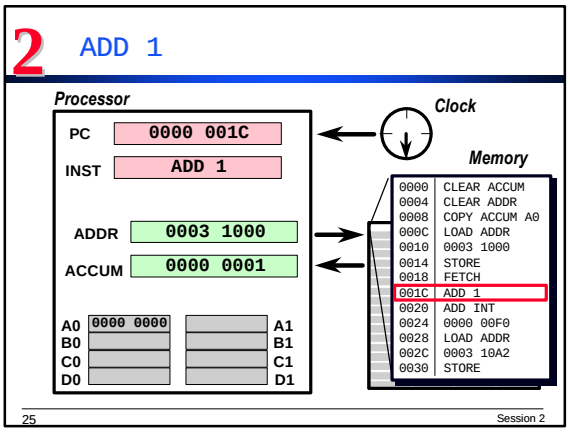
```
0000  CLEAR ACCUM
0004  CLEAR ADDR
0008  COPY ACCUM A0
000C  LOAD ADDR
0010  0003 1000
0014  STORE
0018  FETCH
```

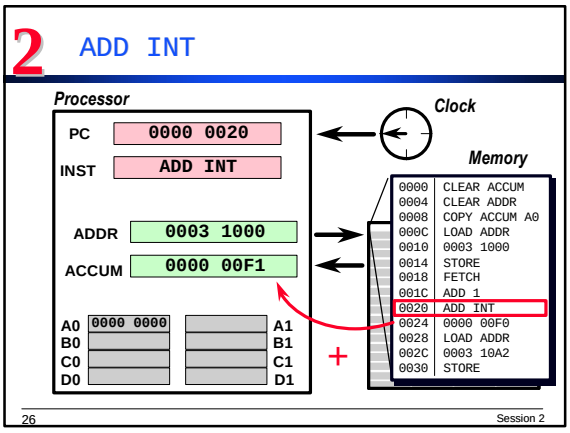

2 Running a Program

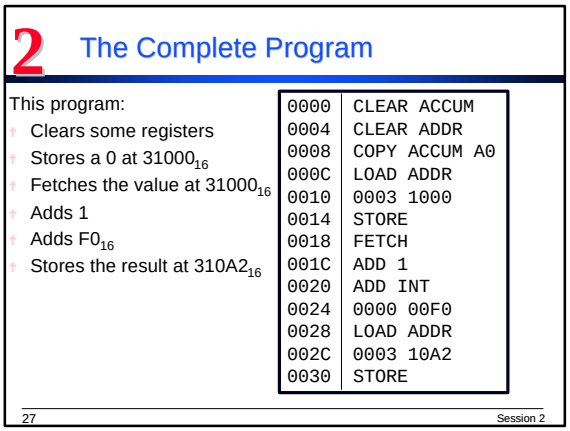
- Once a program has been copied into some sequential locations in memory, we can execute it by loading the first address into the **PC**
- How does the processor know which portions of the program are instructions and which are data?
- It assumes everything is an instruction, unless an instruction tells it to do otherwise!

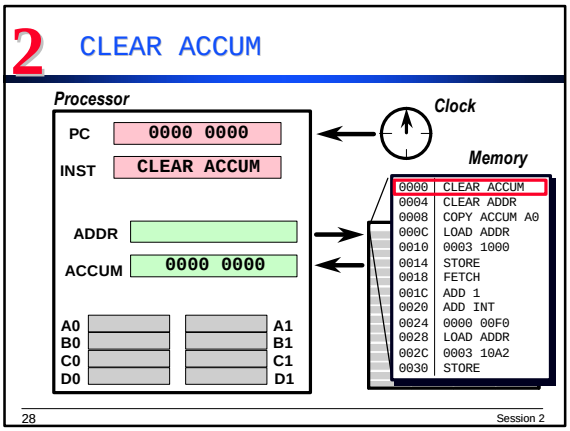
2 Arithmetic

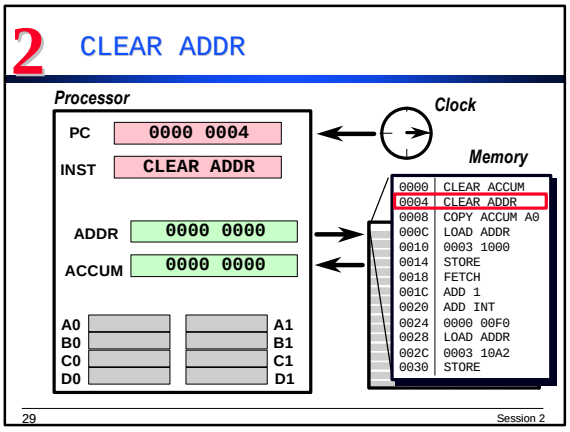
- The processor performs arithmetic storing the result in the accumulator
  - You can think of the accumulator as being similar to the display on a hand-held calculator
- The operand for the arithmetic operation can come from the instruction, a register, or the program
- On simpler processors, multiplication is done by repeated addition, and division is done by repeated subtraction
- Processors without floating point circuitry have to perform operations on floats by using many integer operations

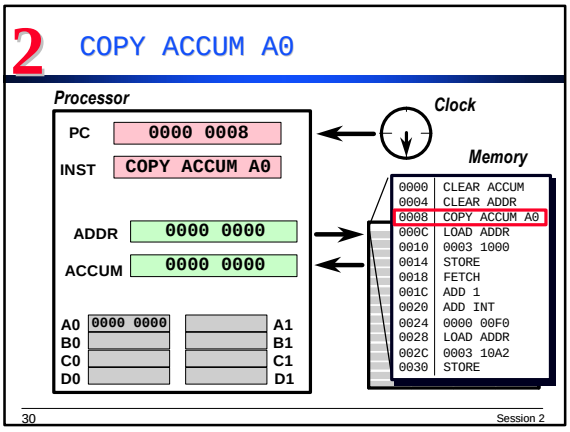


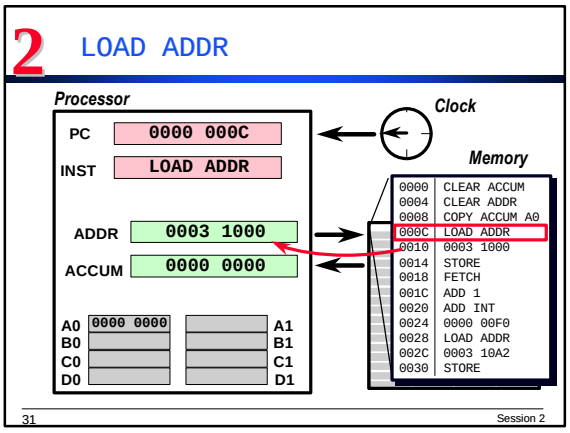


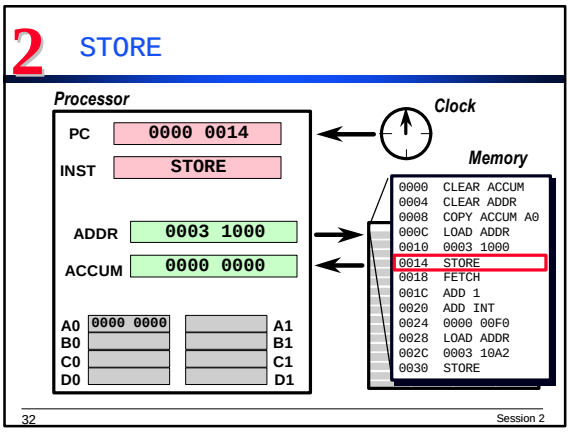


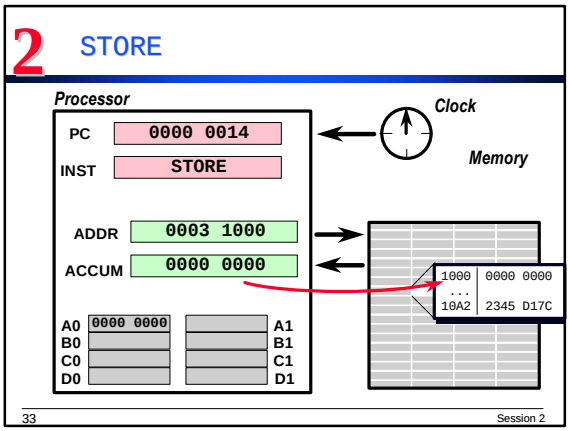


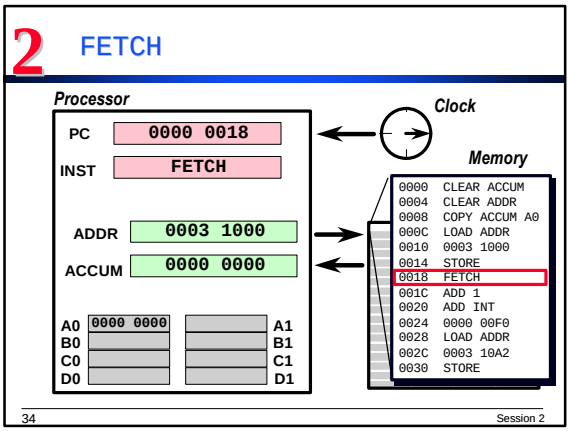


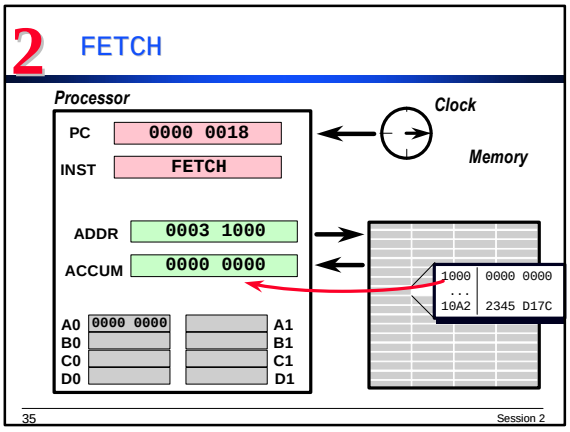


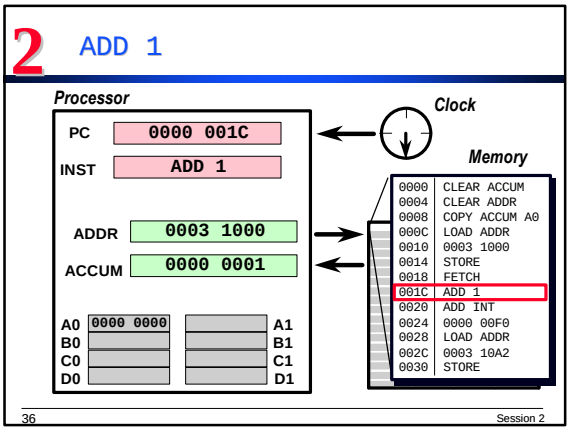


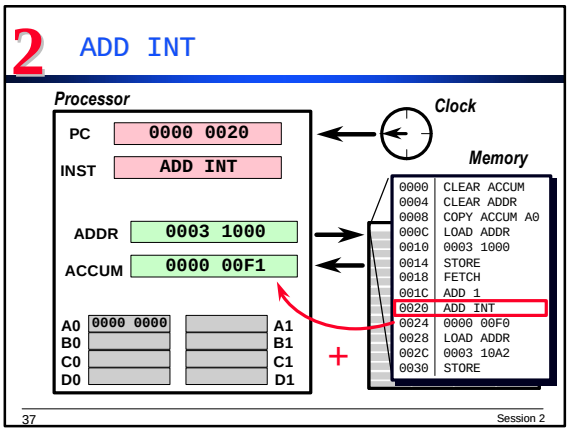


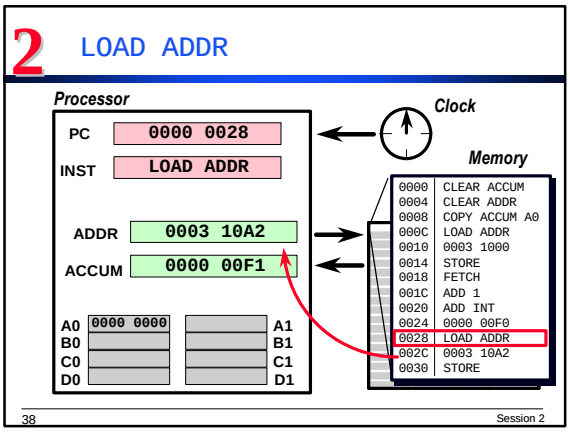


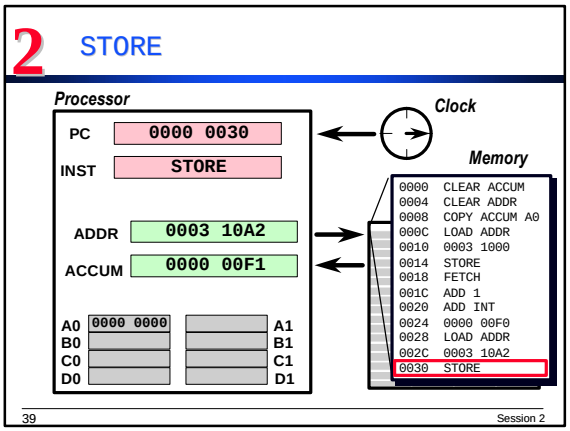


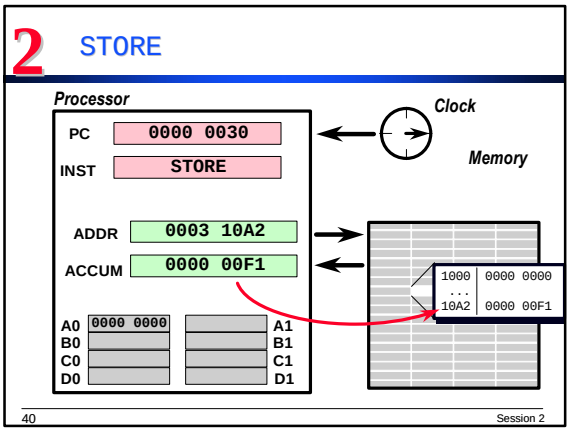












Languages

- The processor only understands instructions written in its machine language
- Expressing a program in machine language is very tedious, we need languages that are more like human languages
- Our previous examples have been in *assembly language*
  - Words represent the opcodes and operands
  - Each line of text represents one machine language instruction
- There are many other computer languages

41 Session 2

What is a Computer Language?

- Computer languages have much in common with written human languages
- **Alphabet**: the set of symbols used to express the language
- **Vocabulary**: the words of the language and their meanings
- **Grammar**: the rules for combining words to produce meaningful sentences

42 Session 2

2 Assembly Language

- The alphabet for assembly language is the ASCII characters (all the letters and symbols on your keyboard)
  - This is true for almost all computer languages
- The vocabulary consists of words like
  - ADD which represent opcodes,
  - ADDR which represent operands, and
  - numbers, also used as operands
- The grammar is very simple
  - The first word is an opcode
  - Depending on the opcode, there will be 0, 1, or 2 operands

43 Session 2

2 What is Meaning?

- A word in a human language is useful when a group of people all give it the same meaning
- Computer languages are similar, what's important is who (or what) understands the language
- Your computer's processor only understands machine language
- There are programs (called *assemblers*) which "understand" assembly language
  - They read a file containing the text of an assembly language program
  - They create a file containing the machine language translation of the program

44 Session 2

2 Low-Level and High-Level

- Assembly language is considered to be a low-level computer language
- By writing more sophisticated translation programs, we can create higher-level languages
- In a high-level language
  - One statement (sentence) will usually translate to a sequence of several machine language instructions
  - There are numerous ways to refer to memory and impose structure on it

45 Session 2

2 Compilers and Interpreters

- A program which translates a text file in a high-level language to a machine language program is called a *compiler*
- There are also interactive *interpreters* which translate commands to machine language and execute them immediately

46 Session 2

2 Many Tongues

- There are dozens of high-level computer languages
- Some are meant to be translated by a compiler, some by an interpreter, a few by either
- Some languages are specialized for particular types of programs
- C++ is a compiled, general-purpose language
- Even though it's a high-level language, it has features that allow it to do anything that can be done in machine language

47 Session 2

2 C++

- The alphabet for C++ is the ASCII character set.
  - Much of the punctuation (#, ;. {}, ( ), etc) has important meaning in C++
- The vocabulary consists of words made from
  - The letters of the alphabet (a-z A-Z)
  - The digits (0-9) and the underscore (_)
  - Any word which begins with a digit is taken to be a number
- The grammar consists of rules for different types of *statements* (the C++ equivalent of a sentence)

48 Session 2

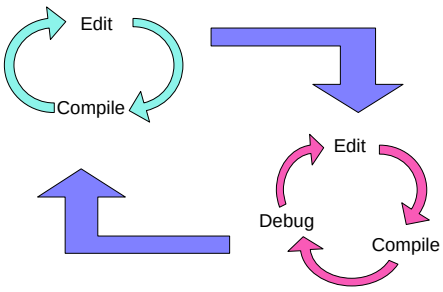
2 Our C++

- Our goal in this class is to immerse you into the C++ language
- We're not going to cover the entire language, or be rigorous in our specifications
- We'll be learning a subset of C++
- More importantly, we'll be learning how a computer language works

2 The Mechanics

- When programming in C++, we type in the text of our program using a text editor
- We then run the compiler on the file(s) we've created
- If we've obeyed all the rules of C++ grammar and not used any unknown vocabulary, the compiler will generate an executable program
- Many packages include a integrated editor and compiler (Microsoft Visual C++, Metrowerks Code Warrior, etc)

2 Code Development Cycles



2 C++ Words

- C++ words are made of letters, digits, and the underscore (_), and may not begin with a digit:
  - `while if int salary Employee33 x0 y0 z0 p q address Address`
  - Note that C++ distinguishes between upper and lower case letters, so `address` and `Address` are two different words
- Words are separated by punctuation or spaces, so we use _ to combine multiple English words into one C++ word:
  - `cost_of_living date_of_hire`
- Many people use capital letters for the same purpose:
  - `costOfLiving dateOfHire`

2 Reserved Words & Your Words

- There are a number of words that are pre-defined in the C++ language, these are called *reserved words* or *keywords* (here are *some* of them):

```
char class do double else float if include  int
  long new private public return short signed
  unsigned void while
```


2 Reserved Words & Your Words (continued)

- There are also many punctuation marks and combinations of punctuation marks that have defined meanings:

```
~ ! # ^ & * ( ) [ ] { } - + = | . , ; ' " < > / %
== <= >= != -> << >> /* */ // ++ -- += -=  *= /= %=
```

- The rest of the vocabulary consists of words that you and other programmers create

2 Numbers

- C++ represents numbers in base 10 just like we do:
  - `10 -3 98.6 -0.0012 0`
- Numbers can also be expressed in hexadecimal using the characters 0x to indicate base 16:
  - `0xA12` = $A12_{16} = 2578_{10}$
- Very large and very small numbers can be expressed in scientific notation by writing the mantissa and the exponent with an e in-between:
  - `0.986e2` = $0.986 \times 10^2 = 98.6$
  - `-0.12e-4` = $-0.12 \times 10^{-4} = -0.000012$

2 Constants

- Numbers in a C++ program represent constant values, values that won't change during the course of your program
- It doesn't matter what base you use to represent a constant, it's represented by the compiler as either an integer or floating point value
  - `0xA12` and `2578` are two ways of expressing the same integer

2 Text Constants

- We can also represent a single byte in ASCII by enclosing it in single quotes:
  - `'A'` is a constant representing the 8-bit integer value 41_{16} (65_{10})
- We can represent a string of characters with double quotes:
  - `"Hello"` is the sequence of bytes:
    - 48_{16} 65_{16} $6C_{16}$ $6C_{16}$ $6F_{16}$ 0_{16}

2 Statements

- The C++ analogue to a sentence is a *statement*
- A statement consists of a sequence of C++ words and constants separated by punctuation and white space
  - white space is any number of spaces or tabs, even blank lines
  - C++ ignores all of the space between words
- Since a statement may be spread among many lines, the end of a statement is marked with a semicolon **;**
- The simplest statement (it does nothing):

2 Operations and Declarations

- Most C++ statements you'll write will be *operations* you want performed by the processor
  - Each C++ statement will translate to multiple machine language instructions
- But there are statements which are not meant to be translated directly into machine language, they are *declarations* made to the compiler
- Imagine that we're talking through a human translator to a third party (the processor)
- A declaration is analogous to speaking to the translator

2 Back to Memory

- The compiler does more than translate C++ operations into machine language
- It also *manages* memory by
  - Setting aside locations in memory for the use of our program (we think of this as *creating* memory)
  - Allowing us to name locations in memory
  - Allowing us to specify what data type we choose to view memory as
  - Releasing memory when we're done with it (*destroying* memory)
- The primary use for declarations is to tell the compiler to manage some memory for us

2 Variables

- A piece of memory managed for us by the compiler is called a *variable*
- Before we can operate on a variable in C++, we have to declare the variable to the compiler
- Here's our first C++ statement:

```
short int a;
```

ends with a semicolon

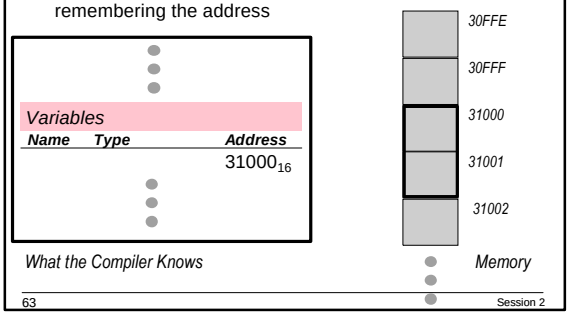
Data Type — Variable name

2 Variable Declaration

- `short int` are two C++ reserved words which together mean "a signed 16-bit integer value" (a number from -32,768 to 32,767)
- a is a word (variable name) we created to refer to this variable
- We can imagine the compiler encountering this declaration in the text of our program and:

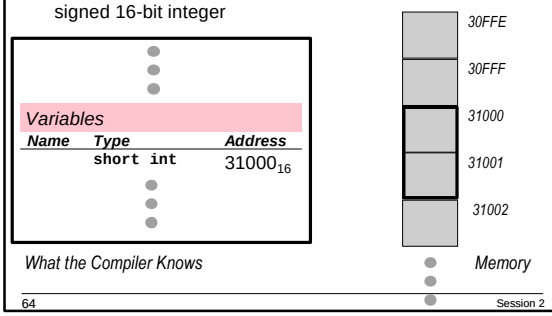
2 Variable Declaration Step 1

- Compiler sets aside 16-bits (2 bytes) of memory, remembering the address



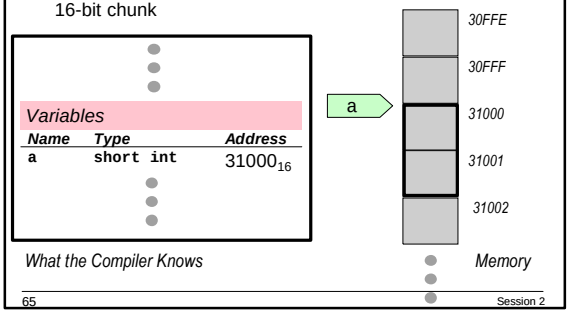
2 Variable Declaration Step 2

- Compiler records that we'll be looking at this memory as a signed 16-bit integer



2 Variable Declaration Step 3

- Compiler records that the name a is going to refer to this 16-bit chunk



2 C++ Data Types

- C++ has reserved words which declare all of the primitive data types described in Session 1
- For integer arithmetic, we declare an `int`
  - An `int` will be 16 bits on a 16-bit machine, 32 bits on a 32-bit machine
- If we need to be specific, we use either `short int` (16 bits) or `long int` (32 bits)
- `int`s are normally signed, we can declare an unsigned one with
  - `unsigned int`
  - `unsigned short int`
  - `unsigned long int`

2 C++ Data Types (continued)

- Boolean variables are declared with `bool`
  - Usually use 1 byte of storage
- Floating point variables are declared with `float` (single precision) and `double` (double precision)
- A single byte, which is usually used to store a character of text encoded via ASCII, is declared with `char`
  - A `char` can also be thought of as a signed 8-bit integer
  - An unsigned 8-bit integer can be declared with `unsigned char`

2 Assignment Statement

<variable> = *<expression>* ;

- Our first operation in C++ is the assignment statement, indicated by the equal sign (=)
- The value of the *expression* on the right of the equal sign is stored in the *variable* on the left

```
a = 0;
```

- `0` is recognized as a constant with the value 0_{10}
- The value 0 is stored in the memory location we've named a

2 Expressions

- The *<expression>* on the right-side of an assignment statement can be more than just a constant
- It can be another variable, or
- An arithmetic expression combining variables and constants with the operators:
  - + Addition
  - - Subtraction
  - * Multiplication
  - / Division
- Parentheses are used for grouping operations

2 Expression Examples

Expression	Value
`42`	A constant, value is 42_{10}
`a`	A variable, the value is whatever is stored in `a`
`6 / 2`	Divides 6 by 2 (evaluates to 3)
`a - 7`	Retrieves the value stored in a and subtracts 7
`a * b + 4`	Multiplies a times b, adds 4
`a * (b + 4)`	Adds 4 to `b`, then multiplies by a

2 Expression Evaluation

- Without parentheses, multiplications and divisions are performed before additions and subtractions
- Otherwise, evaluation proceeds from left to right
- You can use parentheses within parentheses, the innermost parens are evaluated first

`1 / ((a+b) * (c+d))`	`1 / a+b * c+d`
$\frac{1}{(a ? b)(c ? d)}$	$\frac{1}{a} ? bc ? d$

2 Assignment Examples

```
int a, b;
```

- Declares two integers, a and b

```
a = 2;
```

- Stores a 2 in a

```
b = a + 4;
```

- Gets the value stored in a (2)
- Adds 4
- Stores the result (6) in b

2 More Assignment Examples

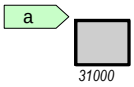
```
a = a + 1;
```

- Retrieves the value in a (2)
- Adds 1
- Stores the result back in a

- We now know enough C++ to express the same sequence of operations as in our machine-language example in the first half of this session

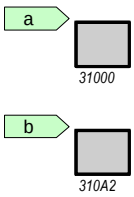
2 Program Animation #1

```
int a, b;
```



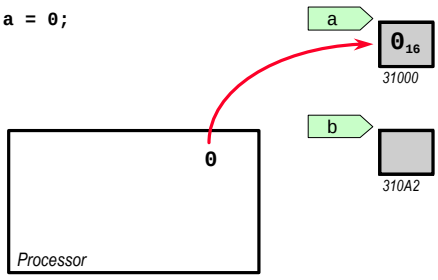
2 Program Animation #2

```
int a, b;
```



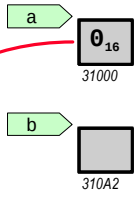
2 Program Animation #3

```
int a, b;
a = 0;
```



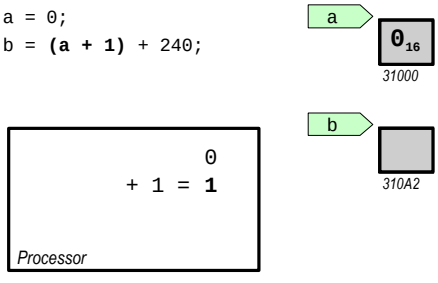
2 Program Animation #4

```
int a, b;
a = 0;
b = (a + 1) + 240;
```



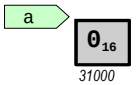
2 Program Animation #5

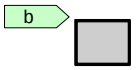
```
int a, b;
a = 0;
b = (a + 1) + 240;
```



2 Program Animation #6

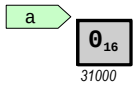
```
int a, b;
a = 0;
b = (a + 1) + 240;
```

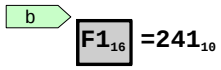


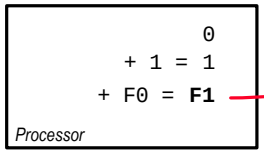


2 Program Animation #7

```
int a, b;
a = 0;
b = (a + 1) + 240;
```







2 Printing an Expression

- The value of any expression can be printed (as text) using the `cout` instruction:

```
cout << 6/3;
```

  - Prints 2

```
cout << "The answer is " << b << endl;
```

  - Prints `The answer is`
  - Prints the value of b
  - Prints a carriage return

2 The Whole Program

Prog2-1.cpp

```
#include <iostream.h>

void main()
{
     int a, b;

     a = 0;
     b = (a + 1) + 240;

     cout << "The answer is " << b << endl;
}
```

Output

```
The answer is 241
```


2 Flow

- In this session we've seen how a computer can store and execute a sequence of instructions
- And how we can express instructions in a high-level language
- How do we control which instructions get executed?
- How do we build a program out of smaller parts?
- These topics will be covered when we discuss *Flow*, in Session 3

3

Introduction to Programming

Flow

Session 3

Phil Mercurio
UCSD Extension
mercurio@acm.org

3

Last Session

- Last session we added the notion of instructions to our model of a computer
- We also saw how to use a compiler to convert C++ instructions into machine language, which can be stored in memory and executed by the processor
- We also saw that the compiler can manage memory for us in the form of *variables*

3

Today's Session

- Flow
- Functions, Procedures, Routines
- Decisions
- Loops
- Streams

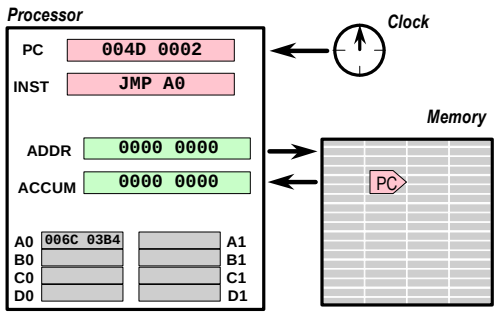
3

Flow

- Our programs would be pretty boring if all they did was perform a single sequence of instructions from start to finish
- Our programs need to respond differently to different inputs
- We need to control which instructions get executed, the *flow* of execution
- First, let's see how this is achieved at the machine language level

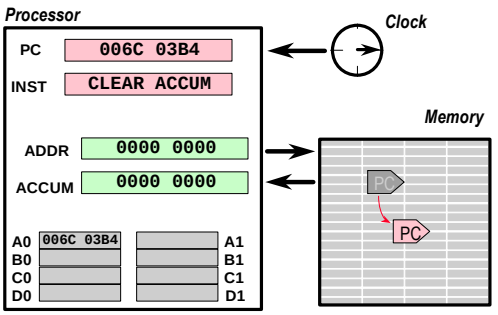
3

JMP A0 (Before)



3

JMP A0 (After)



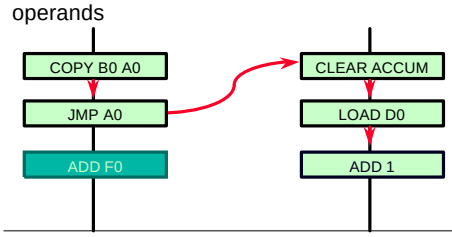
3 Jump!

- JMP A0 is one of many *jump* instructions
- A jump instruction loads a new address into the Program Counter (**PC**)
  - The address may come from a register, as in this example
  - Or it might follow the instruction in memory (the next 32-bit number)
- The flow of execution jumps to the new location in memory

7 Session 3

3 Flowcharts

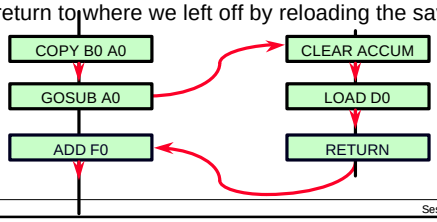
- We can represent the flow of execution diagramatically with arrows in a *flowchart*
- Here each box represents an instruction and its operands



8 Session 3

3 Getting Back

- The JMP instruction takes us to a new location in our program, ignoring the instructions that follow
- What if, instead, we *save* the current **PC** before jumping?
- We return to where we left off by reloading the saved **PC**



9 Session 3

3 Routines and Subroutines

- The current sequence of instructions is a *routine*
  - Routines are also called *functions* and *procedures*--all these names are the same for our purposes
- When we jump to another piece of our program and return, that piece is called a *subroutine* of this routine
- Every processor provides machine language instructions like
  - GOSUB, which saves the **PC** and jumps, and
  - RETURN, which jumps back to the saved **PC**
- In C++, every operation statement is part of a routine
- Routines have names, just like variable names

10 Session 3

3 C++ Routines

`<routine>() { <statements ...> }`

- This is the simplest form of a routine in C++
- *<routine>* is the name of the routine, following the same rules as for variable names
- `()` are an empty set of parens (we'll see what goes here later)
- The statements which make up the routine are enclosed in curly brackets { and }

11 Session 3

3 main()

- When the C++ compiler processes the text of your program, it looks for a routine called `main()`.
- When you run your program, it begins with the first instructions in `main()`
- When the last instruction in `main()` is complete, your program is finished
- Every program has exactly one `main()` routine
- Here's the example from Session 2 expressed as a complete C++ program (that doesn't print anything):

12 Session 3

3 Program 3-1

```
main()
{
     int a, b;

     a = 0;
     b = (a + 1) + 240;
}
```


3 Automatic Variables

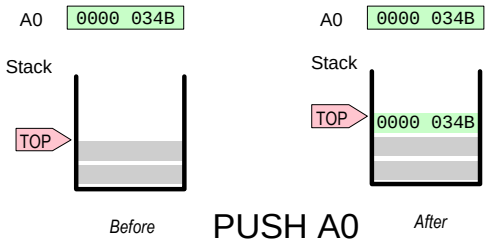
- `int a, b;` declares two integer variables, a and b
- These variables are *automatically* managed by the compiler
- When compiling a routine, the compiler inserts instructions to
  - Create all the declared variables when the routine starts
  - Destroy all the declared variables when it ends
- The rest of the program performs some arithmetic on the variables a and b

3 The Stack

- How does RETURN know where to jump back? Where are automatic variables stored?
- The compiler creates a large chunk of memory for use exclusively by your program, called the *stack*
- The stack operates like a stack of food trays at a cafeteria: the last tray (value) placed on the stack is the first one retrieved
- There are two operations we can perform on the stack (usually available as machine language instructions):

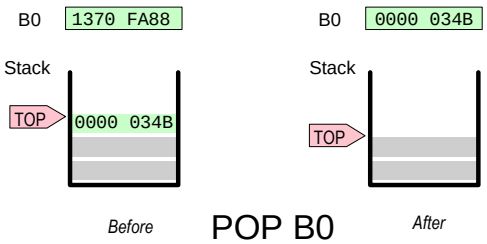
3 PUSH

PUSH Place a value on the top of the stack



3 POP

POP Retrieve the top value from the stack (removing it from the stack)



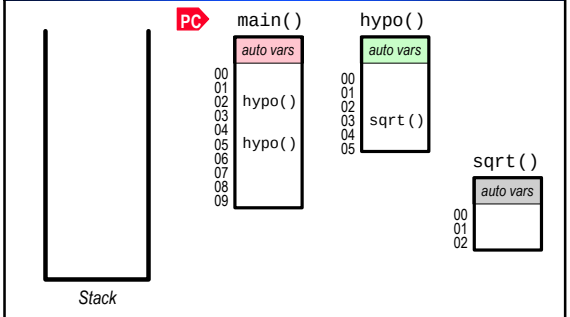
3 The Stack (continued)

- GOSUB stores the **PC** (actually, it stores the address of the instruction *after* the GOSUB) by pushing it on the stack
- The memory needed for automatic variables is also pushed on the stack
- At the end of the routine, the memory used for the automatic variables is popped off the stack
- RETURN pops the top address from the stack and jumps to it
- This lets us call a subroutine, which can in turn call another, etc.

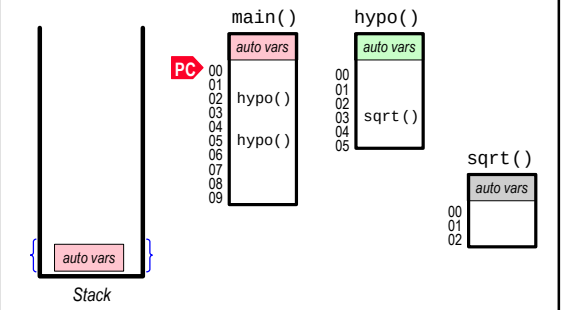
3 Hypotenuses

- Let's say that we've measured the sides (a and b) of two right triangles and we need the hypotenuses (square root of $a^2 + b^2$)
- We'll create a routine called `hypo()` to compute one hypotenuse for us
- We'll also create a routine called `sqrt()` to compute a square root
- We can animate the flow of control like this:

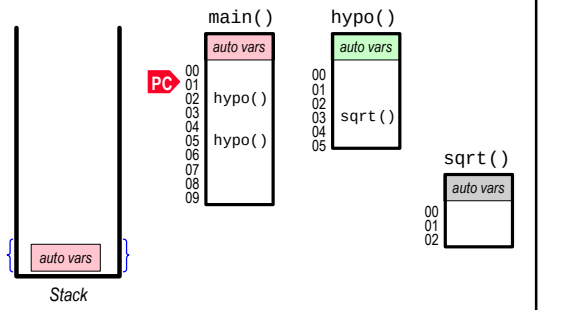
3 Stack Animation #1



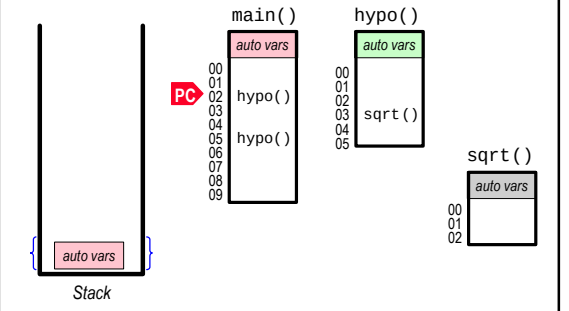
3 Stack Animation #2



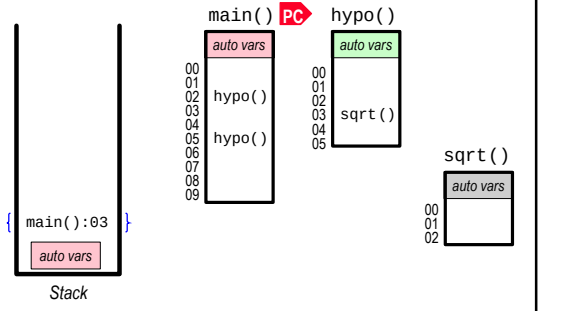
3 Stack Animation #3

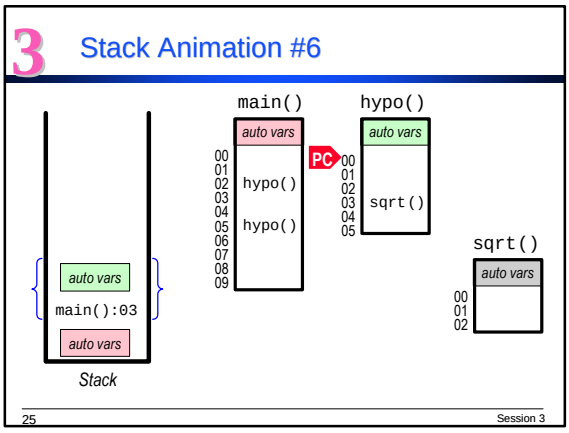


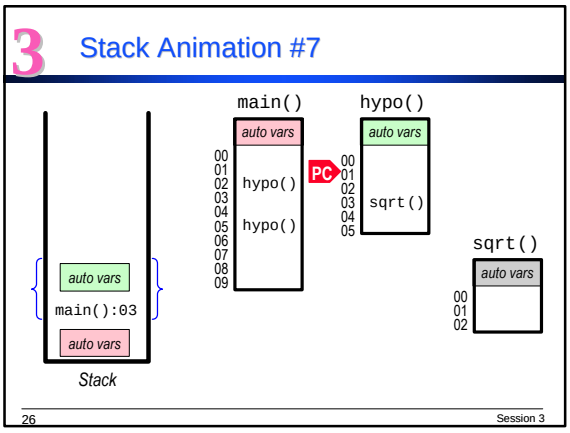
3 Stack Animation #4

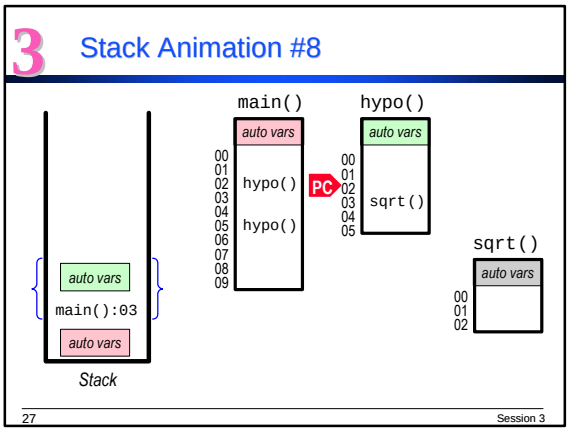


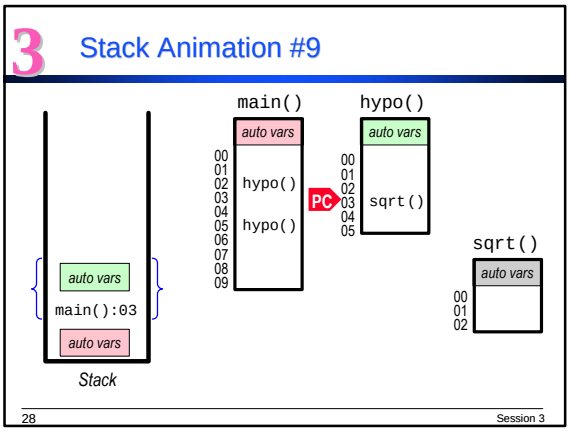
3 Stack Animation #5

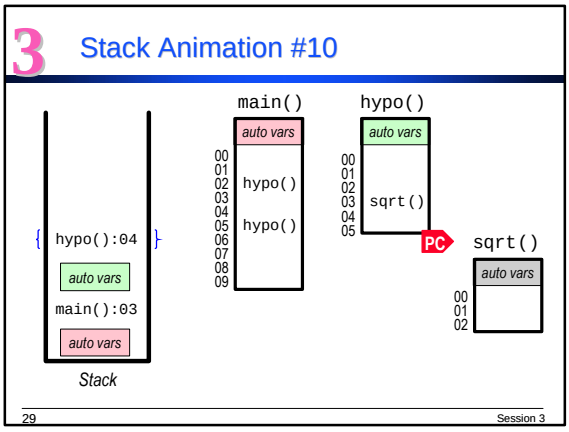


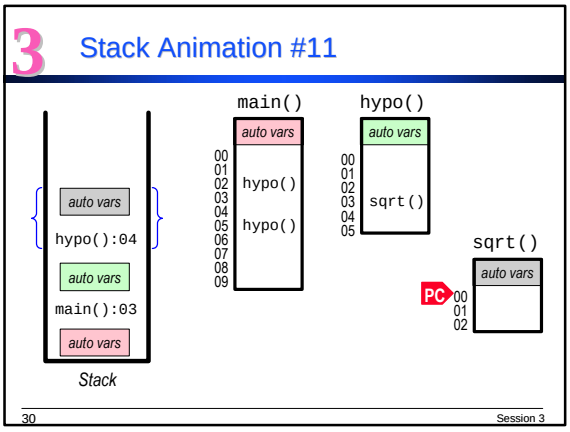


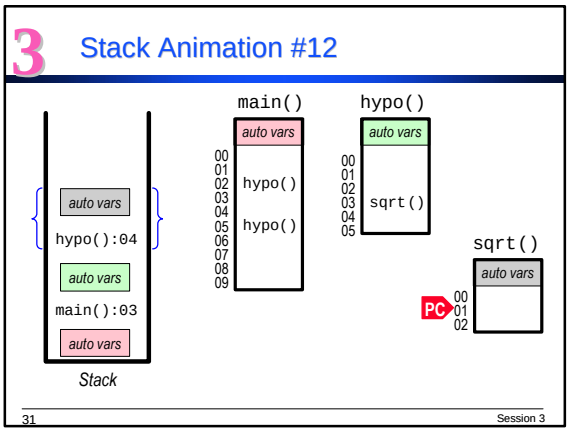


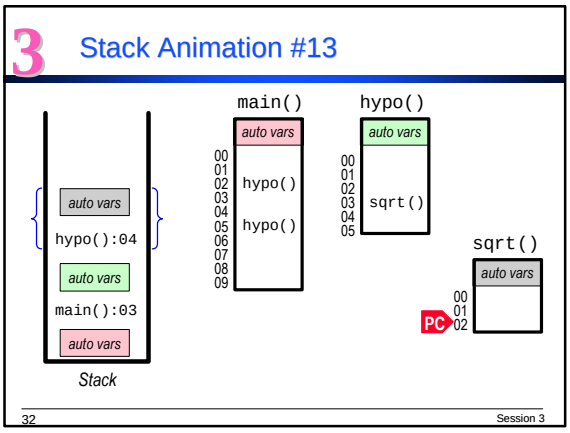


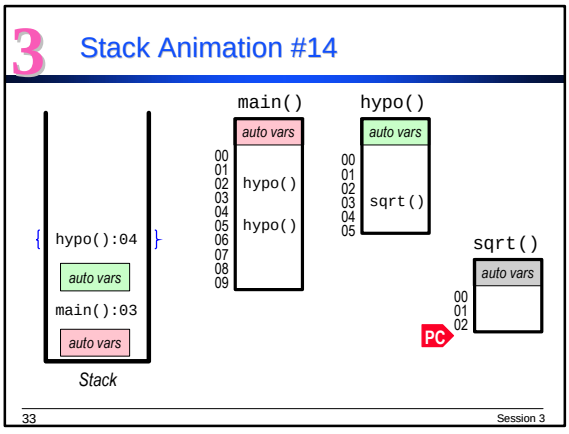


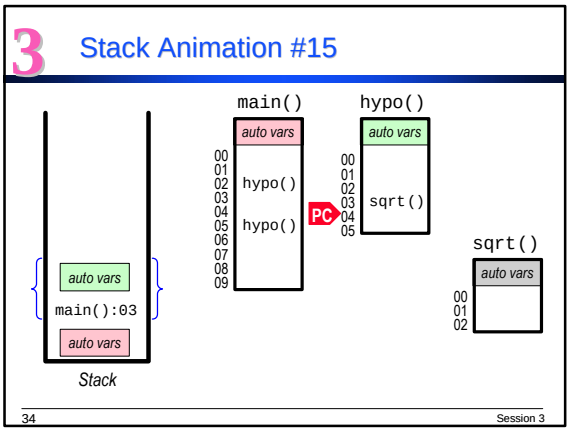


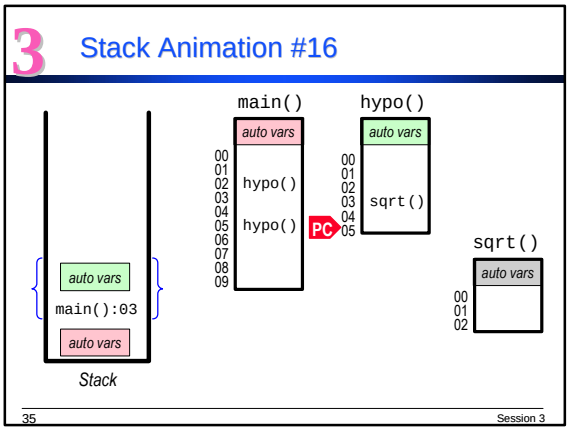


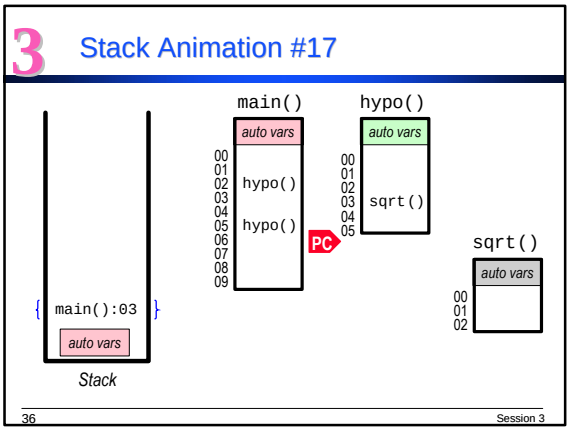


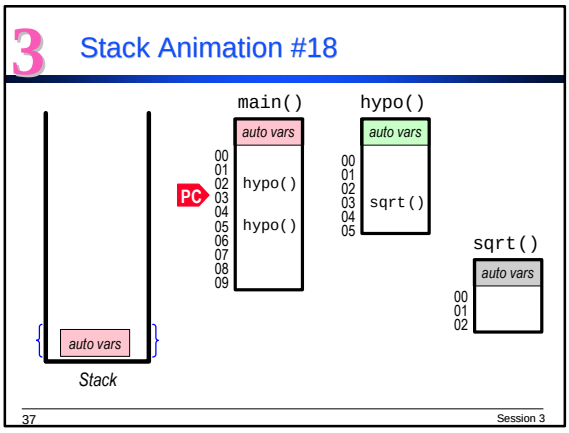


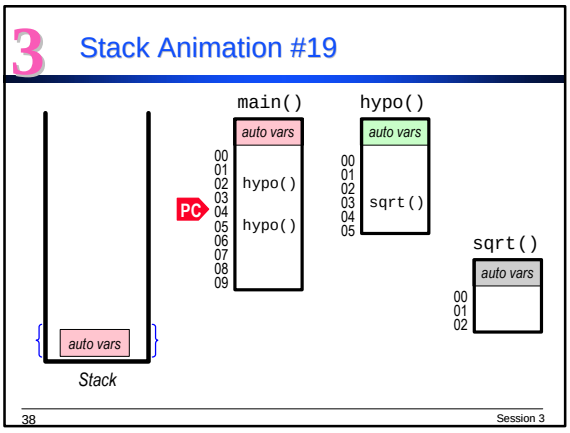


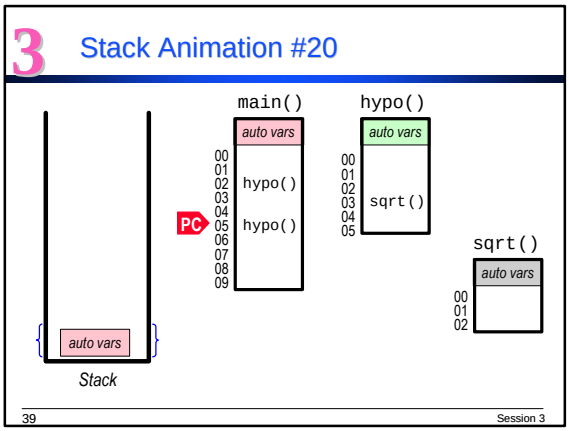


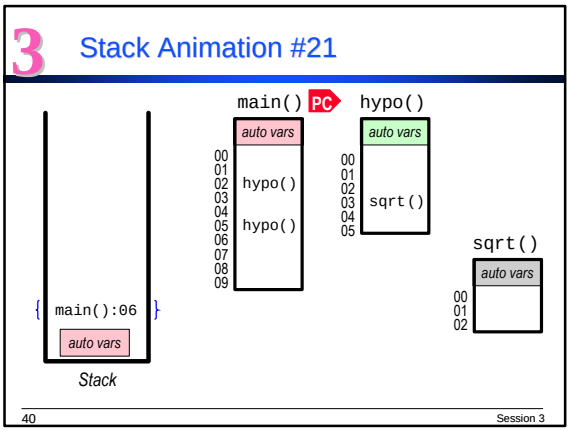


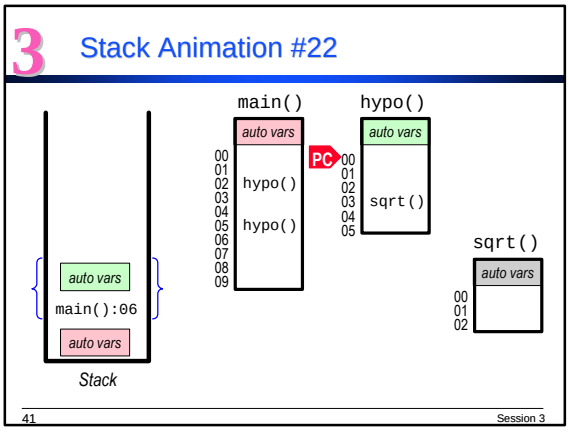


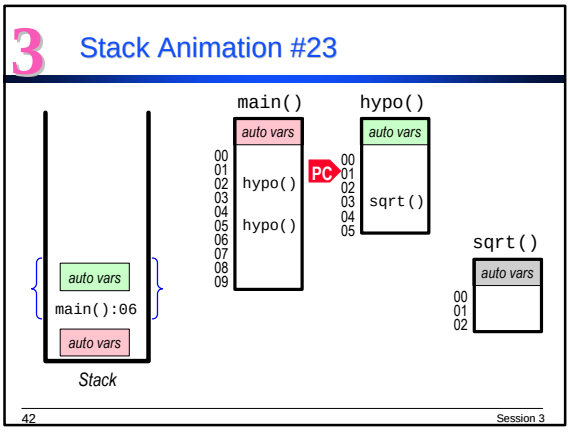


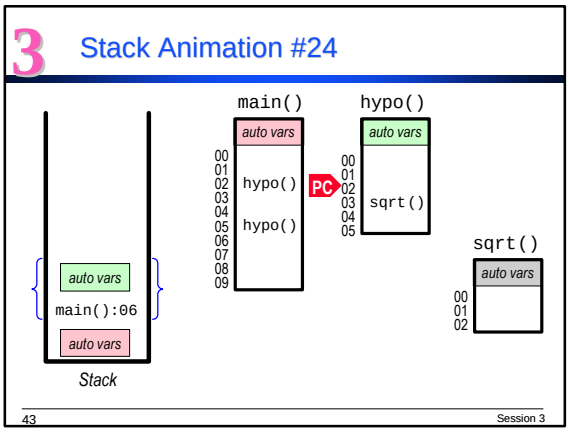


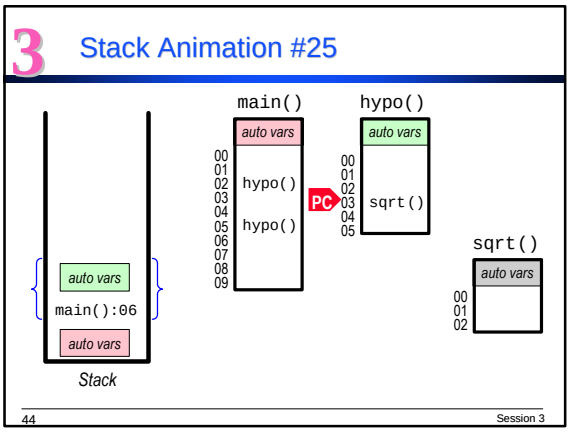


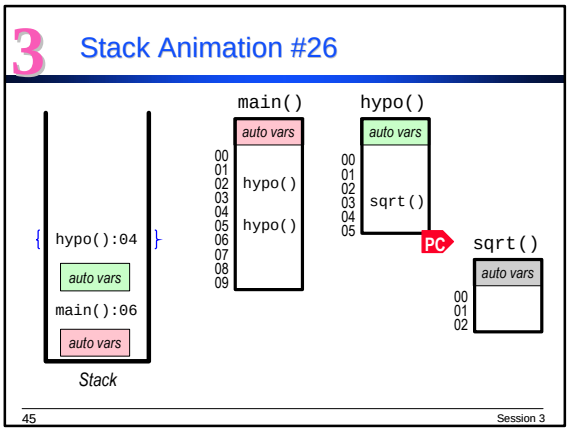


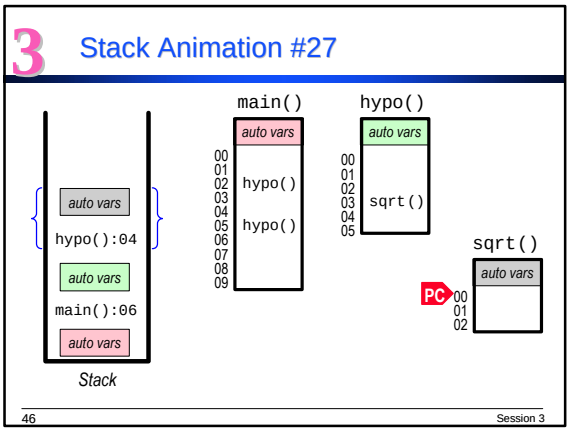


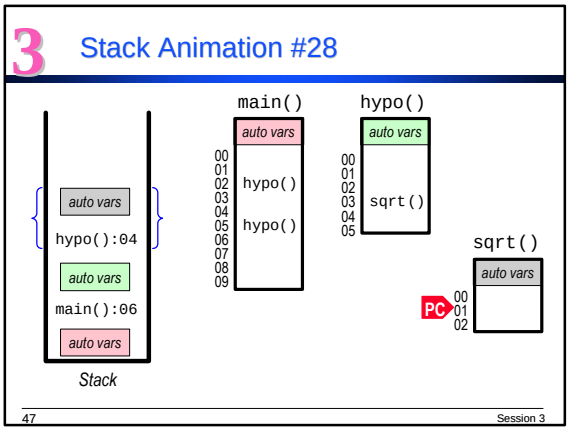


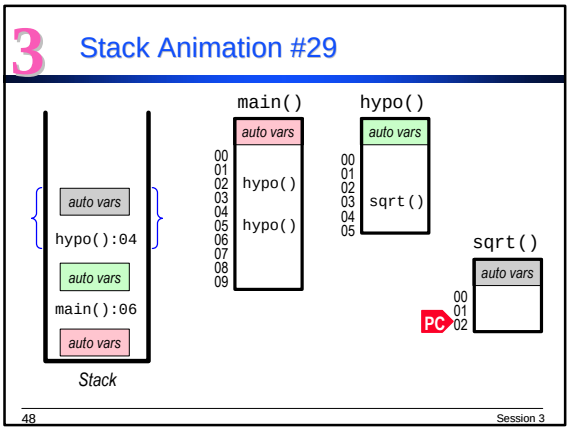


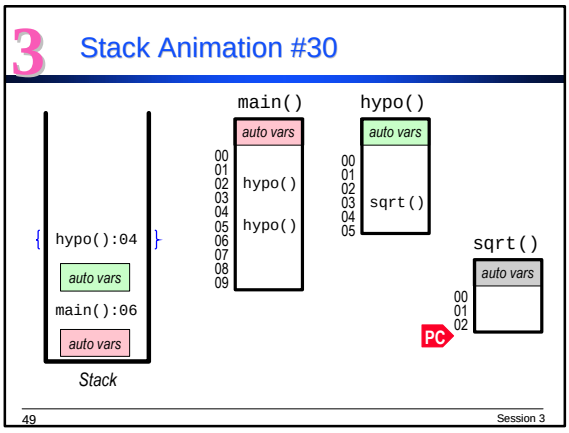


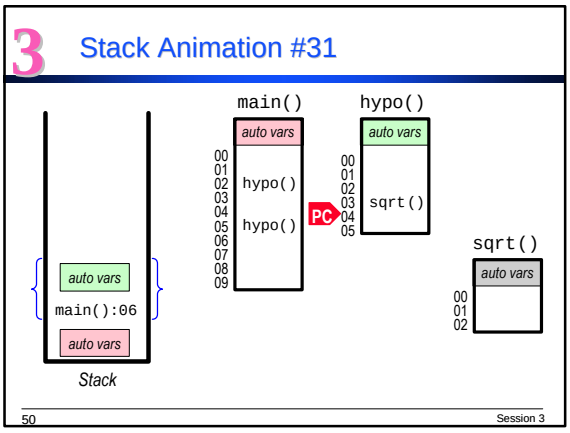


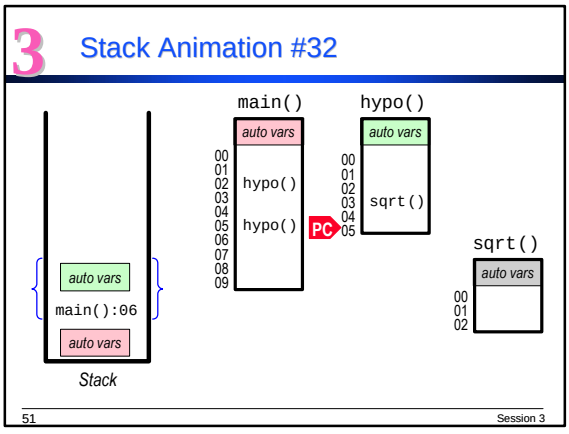


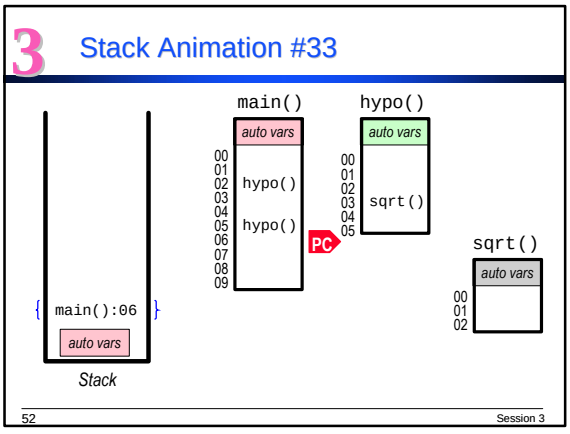


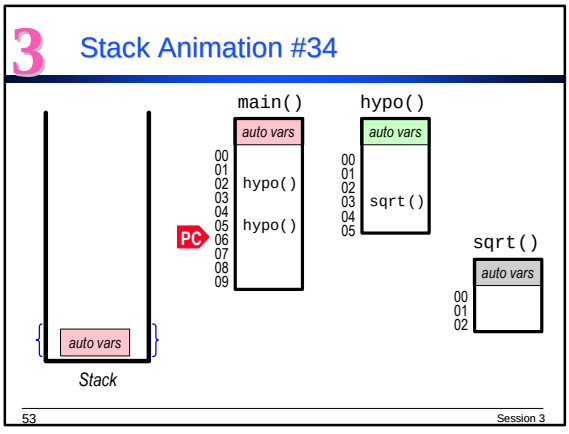


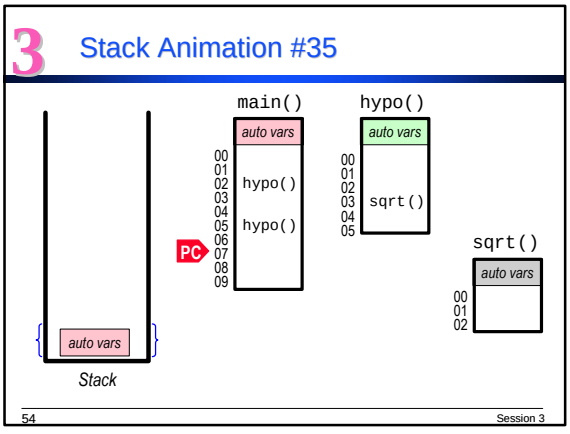




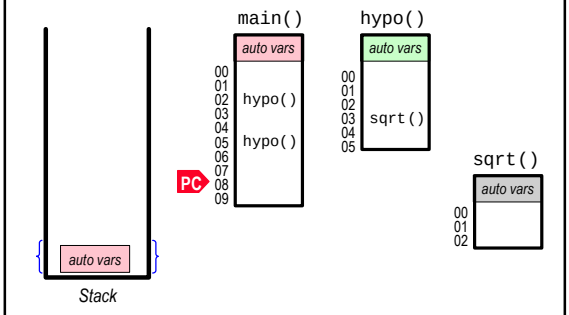




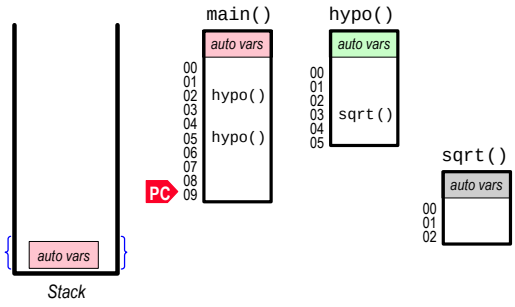




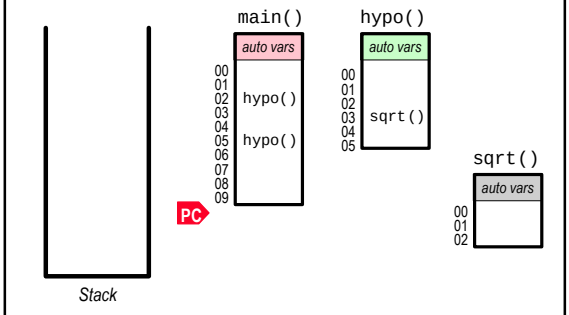
3 Stack Animation #36



3 Stack Animation #37



3 Stack Animation #38



3 The Routine's the Thing

- The routine is a self-contained unit of a program
  - Each routine has its own private, automatic variables
  - One routine can't access another's private variables
- The memory stored on the stack is called the *context* in which the routine operates
  - Going in to and out of subroutines is called *context switching*
- By adding information to the context, we can copy the values of variables from a routine to a subroutine
- We can also have the subroutine *return* a value to the routine

3 Arguments

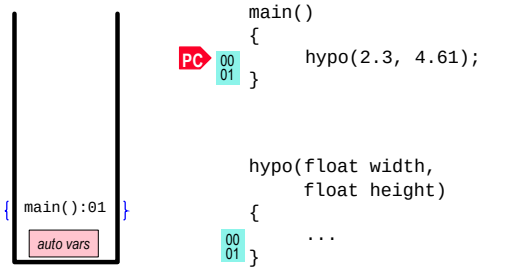
- When we define a routine we can specify one or more *arguments*, variables copied from the parent routine to the subroutine

```
hypo(float width, float height) { ... }
```

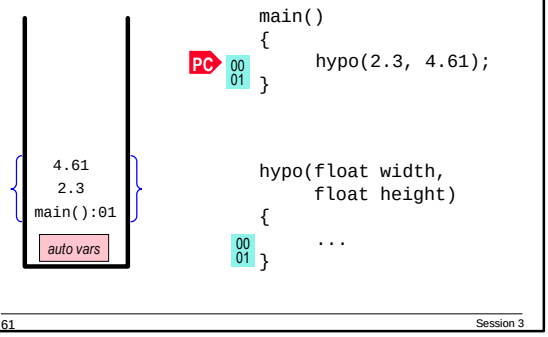
- This is the definition of a routine which expects to find two floating point numbers on the stack
- Here's how that routine might be called from `main()`:

```
main()
{
    hypo(2.3, 4.61);
}
```


3 Arguments Animation #1

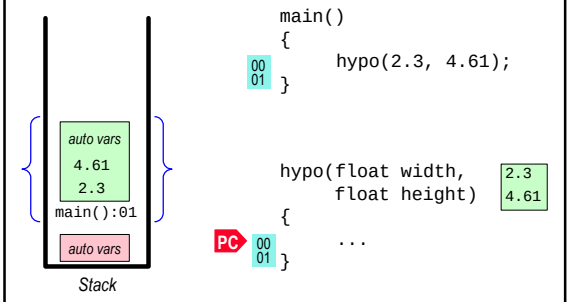


3 Arguments Animation #2



61 Session 3

3 Arguments Animation #3



62 Session 3

3 Return Values

- A subroutine communicates with the routine that called it by leaving a value on the stack
- Only one value can be returned, but of any type

```
float hypo(float width, float height) {
  ... }
```

- The complete definition of `hypo()`:
  - Two `float` arguments width and height
  - Returns a `float` value
- The value returned will be the result of the calculation
- We return a value with the `return(`*<expression>*`);` statement

63 Session 3

3 Routines in Expressions

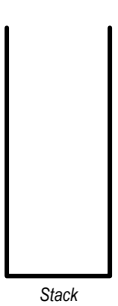
- We can think of the arguments to a routine as its inputs, and the return value as its output
- This makes C++ routines similar to a function in mathematics: $y = f(x)$
  - C++ routines are often called functions
- The value of a function can be used in an expression, along with variables and constants

```
answer = hypo(2.4, 4.61);
```

- Calls the routine `hypo()` with the arguments 2.4 and 4.61
- Stores the returned value in `answer`

64 Session 3

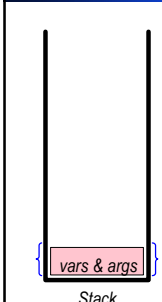
3 Functions Animation #1




```
PC main()
{
     float answer;
00   answer = hypo(2.3, 4.61);
}
float hypo(float width,
           float height)
{
     float a,b;
00   a = width*width +
         height*height;
01   b = sqrt(a);
02   return(b);
}
float sqrt(float x)
{
00   ...
}
```

65 Session 3

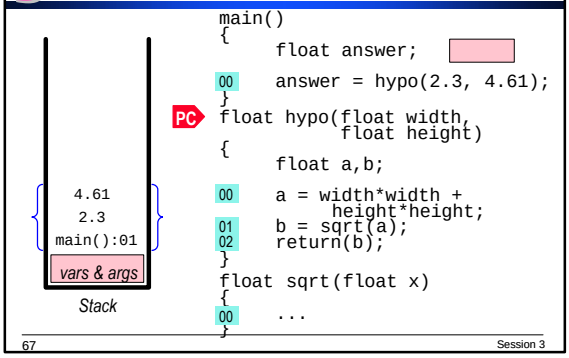
3 Functions Animation #2



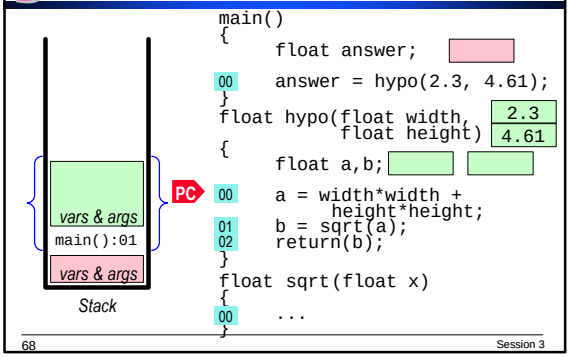

```
main()
{
     float answer;
PC 00  answer = hypo(2.3, 4.61);
}
float hypo(float width,
           float height)
{
     float a,b;
00   a = width*width +
         height*height;
01   b = sqrt(a);
02   return(b);
}
float sqrt(float x)
{
00   ...
}
```

66 Session 3

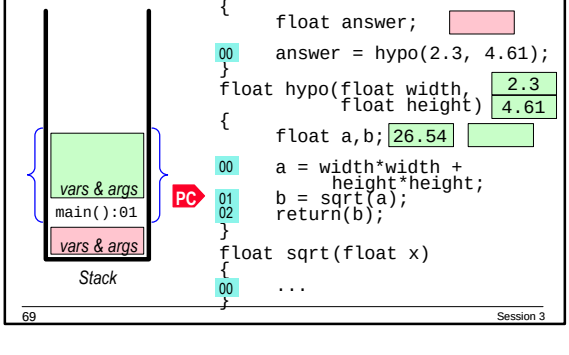
3 Functions Animation #3



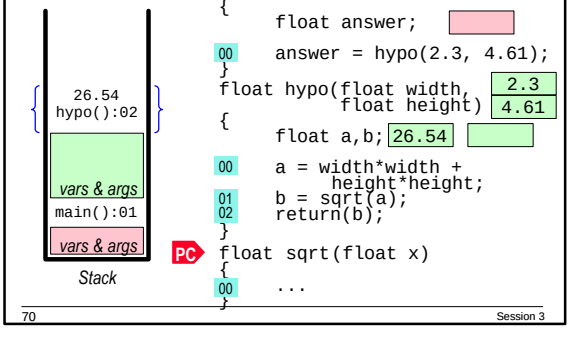
3 Functions Animation #4



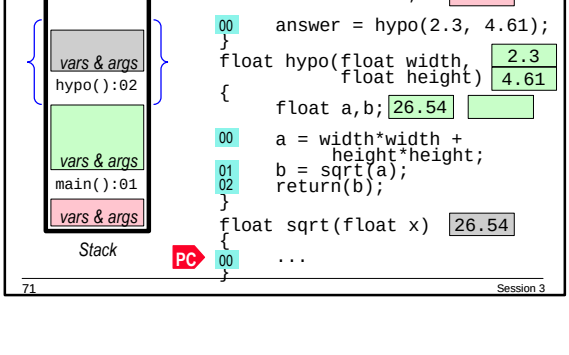
3 Functions Animation #5



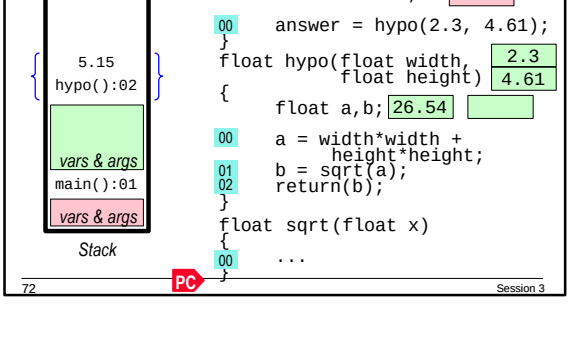
3 Functions Animation #6



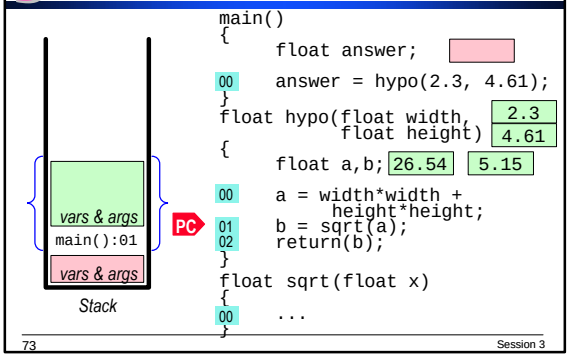
3 Functions Animation #7



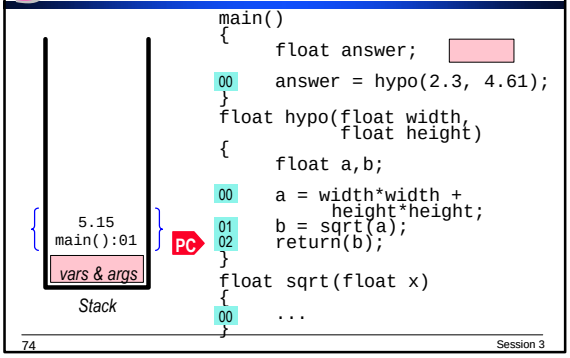
3 Functions Animation #8



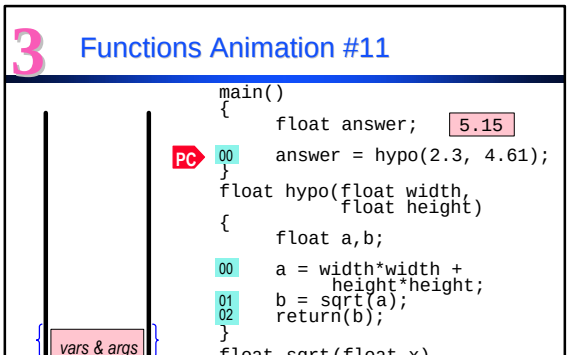
3 Functions Animation #9



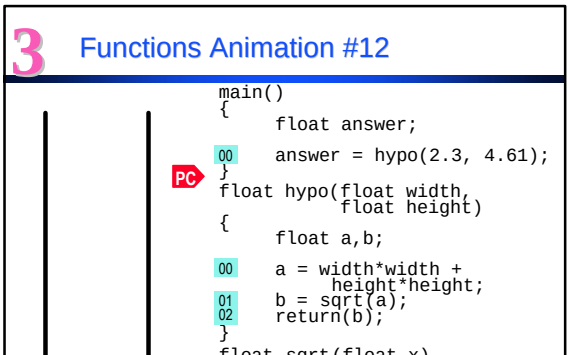
3 Functions Animation #10



3 Functions Animation #11



3 Functions Animation #12



3 Defining Routines

<return type> <name>(<args...>) {

<declarations>

<operations>

}

- A routine is defined when we specify
  - *<return type>* the data type it will return
  - *<name>* its name
  - *<args...>* 0 or more arguments separated by commas
  - *<declarations>* automatic variables
  - *<operations>* C++ statements that do this routine's work

3 Declaring Routines

<return type> <name>(<args...>);

- Just as with variables, we need to declare routines before we can use them
- The declaration looks like the beginning of the definition
  - Everything but the *<statements>* themselves
- Each function may only be defined once, but it may be declared any number of times

3 Declaring vs. Defining

- Defining a routine adds a new C++ word to the vocabulary of your program
  - We can think of the name of the routine as meaning: "perform this group of instructions and return"

3 Declaring vs. Defining (continued)

- Declaring a routine is an aside to the compiler that a routine is defined somewhere, while specifying how to use it grammatically
  - It tells the compiler how to pass arguments to the routine and what type the result will be
  - Without a declaration, the subroutine may not be called correctly, and the stack might get messed up
  - Similar to stating that an English word is a noun or a verb, without giving a definition

3 The Void

- C++ has a special data type called `void`
- `void` is used to indicate that a routine doesn't return anything, or takes no arguments
- `void saveGame(void);`
- Declares a routine called saveGame() which takes no arguments and does not return a value
- We return from a void routine with `return;`
  - Or by coming to the closing brace }

3 Namespaces

- A *namespace* is a collection of **unique** names
  - When trying to understand what, exactly, a variable name or routine name refers to, consider what namespace it is in
- Each routine has a private namespace for its local variables
- Variables declared outside of any routine are part of the *global* namespace, they are valid in all routines
  - In case of conflict, the local namespace overrides the global namespace
- Routines are always part of the global namespace, regardless of where the declaration appears

3 Namespace Example

Global Namespace

```
float hypo(float width, float height);
float answer;
main()
{
     float answer;
     ...
}                                   main()'s namespace
float sqrt(float x);
float hypo(float width, float height)
{
     float a,b;
     ...
}                                   hypo()'s namespace
float sqrt(float x)
{
     ...
}                                   sqrt()'s namespace
```


3 Scope

- A C++ declaration is said to have *scope*, the portion of the text of the program where the declaration is valid
- The automatic variables and arguments of a routine have *local* scope, they are defined only within their routine's braces { }
- Declarations made outside of all the routines have *global* scope, they are valid in all routines
- The scope of a declaration flows downward from where it appears in the file
  - In the text above the declaration, that name is not *in scope* and can't be used

3 Scope Example

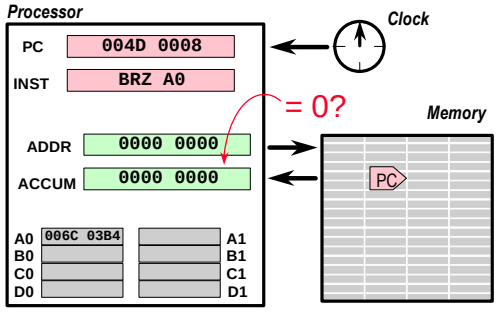
Global Scope

```
float hypo(float width, float height);
float answer;
main()
{
     float answer;
     ...
}                                   Local to main()
float sqrt(float x);
float hypo(float width, float height)
{
     float a,b;
     ...
}                                   Local to hypo()
float sqrt(float x)
{
     ...
}                                   Local to sqrt()
```

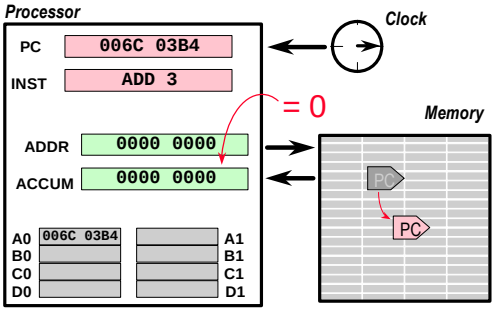

3 Still Linear

- We now know everything we need to know about how to break a program up into routines
- But our routines always do the same thing, each time they are called
- We need to be able to make decisions based on values in memory
- The processor has machine language instructions called branch instructions which load a new value into **PC**, but only if a certain condition is met

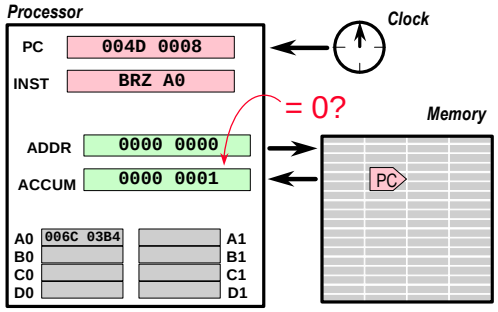
3 BRZ A0 (Before)



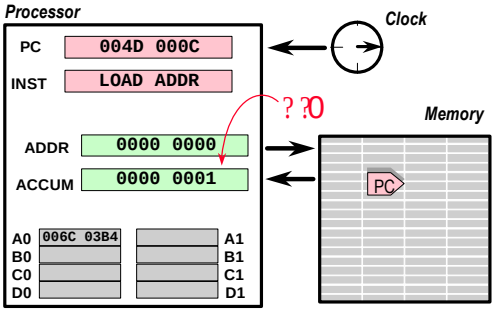
3 BRZ A0 (After)



3 BRZ A0 (Before)



3 BRZ A0 (After)



3 Branching

- The previous example was an instruction called BRZ (Branch if Zero)
- BRZ A0 tests the accumulator
  - if **ACCUM** contains a 0, then **A0** is copied into **PC**
  - Otherwise, we proceed with the next instruction
- In C++, we can express a branch with the `if` statement

3 The if Statement

```
if (<condition>) {
   <statements if true>
}
else {
   <statements if false>
}
```

- If the <*condition*> is true, <*statements if true*> are executed, otherwise <*statements if false*>
- Only one of the two groups of statements is executed
- A group of statements in `{ }` is called a *block*

3 What is Truth?

- In C++, false is defined as a value of 0
- True is any non-zero value, like 1
- The <*condition*> can be any combination of variables, constants, and subroutine calls, connected by operators
- The <*condition*> expression is evaluated, any non-zero value indicates truth
- There are a group of operators which can be used to compare two values
  - They evaluate to true or false (1 or 0)

3 Relational Operators

- C++ has six *relational operators*:

Operator	Meaning	Example
`==`	Equal to	`count == 10`
`!=`	Not equal to	`flag != DONE`
`<`	Less than	`a < b`
`<=`	Less than or equal to	`low <= high`
`>`	Greater than	`pointer > endOfList`
`>=`	Greater than or equal to	`j >= 0`

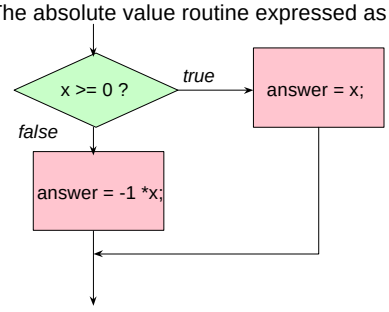
3 if Example

- This routine returns the absolute value of a floating point number

```
float  absoluteValue(float x)
{
     float answer;

     if(x >= 0) {
          answer = x;
     }
     else {
          answer = -1 * x;
     }

     return(answer);
}
```


3 absoluteValue Flowchart

- The absolute value routine expressed as a flowchart:



3 Testing a Boolean

- Here's a routine which tests the global Boolean variable `gameOver`
  - Note that we don't need an `else`

global variable → `bool gameOver = false;`

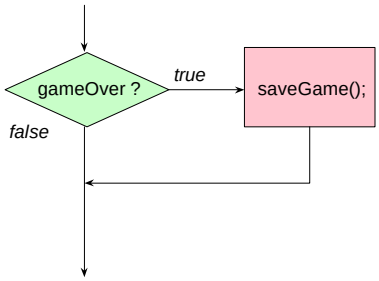
```
...

void  endOfRound(void)
{
      if(gameOver){
            saveGame();
      }
}
```

no return value → `void`

97 Session 3

3 Boolean Flowchart

- `endOfRound()` expressed as a flowchart:



98 Session 3

3 Comparison Function

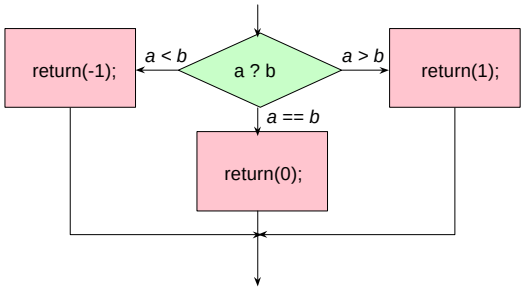
- Let's create a routine to help us sort numbers
- This routine will take two `floats` as arguments and return
  - 0 if they are equal
  - -1 if the first is less than the second
  - 1 if the first is greater than the second

```
int compare(float a, float b);
```

- The declaration for the routine

99 Session 3

3 compare() Flowchart



100 Session 3

3 compare() Version 1

```
int  compare(float a, float b)
{
     if(a == b){
          return(0);
     }
     else {
          if(a < b) {
               return(-1);
          }
          else {
               return(1);
          }
     }
}
```

101 Session 3

3 compare() Version 2

```
int  compare(float a, float b)
{
     if(a == b){
          return(0);
     }
     else if(a < b) {
          return(-1);
     }
     else {
          return(1);
     }
}
```

102 Session 3

3 Complicated `if`'s

- The `else if` construct allows us to select from more than 2 branches
- We can have as many else-ifs as we want, for an *n-way branch*
- In this example, only one of the three possible blocks of statements will be executed

3 Logical Expressions

- C++ provides operators to combine truth values in logical expressions:
- <*operand*> && <*operand*> logical AND
- <*operand*> || <*operand*> logical OR
- ! <*operand*> logical NOT
- The <*operands*> are treated as Boolean values (0 is false, non-zero is true)
- The result of the operation is a Boolean value

3 ! Truth Table

a	! a
T	F
F	T

3 && Truth Table

a	b	a && b
T	T	T
T	F	F
F	T	F
F	F	F

3 || Truth Table

a	b	a || b
T	T	T
T	F	T
F	T	T
F	F	F

3 `isLowerCase()`

- This function returns a Boolean indicating whether the argument is a lower case letter or not
  - This works because the ASCII table has the letters in order
  - c must be >= `'a'` AND <= `'z'` to be lower case

```
bool isLowerCase(char c)
{
     if(c >= 'a' && c <= 'z') {
          return(true);
     }
     else {
          return(false);
     }
}
```

3 matchedPair()

- This function returns a Boolean indicating whether the two arguments are a matched pair: `( )`, `{ }`, or `[ ]`

```
bool matchedPair(char a, char b)
{
     if(( a == '(' && b == ')' ) ||
        ( a == '[' && b == ']' ) ||
        ( a == '{' && b == '}' ) ) {
          return(true);
     }
     else {
          return(false);
     }
}
```


3 Loops

- We can also use the branch instructions to implement a *loop*
- A loop is a block of instructions that is repeated
- Every loop must have a condition to test to determine whether or not to continue looping
- In C++ there are two forms of loop statements:
  - `while`, which tests the condition at the beginning of the block
  - `do-while`, which tests the condition at the end of the block

3 while Statement

`while (`*<condition>*`) {`

<statements>

`}`

- The *<condition>* is evaluated just like the *<condition>* in an `if` statement
- As long as *<condition>* is true, the *<statements>* are executed
- Each time the block of *<statements>* is executed, the *<condition>* is tested again
- If the *<condition>* starts out false, the *<statements>* never get executed

3 do-while Statement

`do {`

<statements>

`} while (`*<condition>*`);`

- Just like the `while` statement, except that the *<statements>* are executed first
- Then the *<condition>* is evaluated
- As long as *<condition>* is true, the *<statements>* are executed again
- Even if the *<condition>* starts out false, the *<statements>* get executed at least once

3 The operator-equals Shorthand

- `answer = answer + 1;` is a common C++ idiom
- It's so common, that C++ provides a shorthand: `answer += 1;`
- Any operator can be combined with `=` to mean "use the variable on the left and the expression on the right as operands, and store the result in the variable on the left"

`x *= a + 1;`

- Multiplies x times `(a + 1)` and stores the result in x

`b -= 1;`

- Subtracts one from b

3 The Increment and Decrement Operators

- Even `b += 1;` is too much typing for a C++ programmer!
- For the special case of adding 1 (incrementing) or subtracting 1 (decrementing) a variable, C++ provides these shorthands:
- `a++;` is the same as `a += 1;` (which is the same as `a = a + 1;`)
- `a--;` is the same as `a -= 1;`

3 power() Function

- This function raises a `float` to a power and returns the result (pow must be >= 0)

```
float power(float x,  int pow)
{
     float result = 1.0;

     while(pow > 0) {
          result *= x;
          pow--;
     }

     return(result);
}
```


3 Variable Table for power()

- Let's look at the contents of the variables when we call `power(10.26,5)`

x	pow	pow > 0	result
10.26	5	True	10.26
10.26	4	True	105.27
10.26	3	True	1080.05
10.26	2	True	11081.27
10.26	1	True	113693.81
10.26	0	False	DONE!!

3 power() Version 2

- This version combines `if` and `while` statements to handle both positive and negative powers

```
float power(float x,  int pow)
{
     float result = 1.0;

     if(pow < 0) {
          while(pow < 0) {
               result /= x;
               pow++;
          }
     }
     else {
          while(pow > 0) {
               result *= x;
               pow--;
          }
     }

     return(result);
}
```


3 Streams

- A **stream** is a sequence of bytes, we read or write to a stream in chunks of 1 or more bytes
- You normally have at least two streams, one for input and one for output
- The input stream is usually the keyboard, and the output stream is the screen
- In a windowing environment like MS Windows or MacOS, your compiler can add on code to create a separate window for your program's input and output streams

3 Using Streams

`#include <iostream.h>`

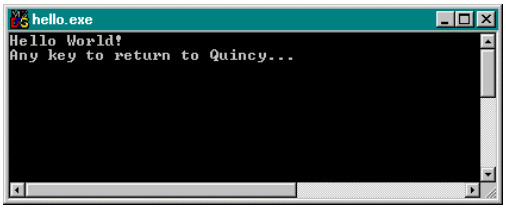
- This command, when placed at the beginning of our program, tells the compiler to include the contents of `iostream.h`
  - The <> indicates that the header file was installed with the compiler
- All the declarations necessary to use streams are in `iostream.h`
- The routines which implement streams have already been compiled, and are automatically included in your program

3 cin and cout

- There are two instructions for using streams:
  - `cin >> <variable>;`
    - Read the value of <*variable*> from the keyboard
  - `cout << <expression>;`
    - Write the value of <*expression*> to the display
- Input and output are both in the form of text

3 Program 3-2

```
#include <iostream.h>

void main()
{
     cout << "Hello World!";
}
```



3 Notes on Program 3-2

- The expression being sent to `cout` is the constant string `"Hello World!"`
  - `cout` prints the text to the output window
- Most languages are introduced using a similar example

3 Program 3-3

```
#include <iostream.h>

void main()
{
     float age;

     cout << "Please enter your age: ";
     cin >> age;

     cout << "That would be " << age / 7.0
          << " years in a dog's life!" <<  endl;
}
```

Output

```
Please enter your age:  43
That would be 6.14286 years in a dog's life!
```


3 Notes on Program 3-3

`cin >> age;`

- Reads, from the keyboard, a value to store in `age`
- The program flow doesn't proceed past this instruction until the user provides the input

- Note that we can have multiple `<<`s in a single `cout` statement
- `age / 7.0` divides `age` by 7, `cout` then prints it
- `endl` is a constant (defined in `iostream.h`) equivalent to an end-of-line character (carriage return)

3 So Far

- We've seen how to control the flow of execution using functions, branches, and loops
- We've seen how to do simple input and output with `cin` and `cout`
- This is how instructions (our program) are structured
- In Session 4 we'll see how both data and instructions can be structured to represent *objects*

4

Introduction to Programming

Objects

Session 4

Phil Mercurio
UCSD Extension
mercurio@acm.org

Recap

- We've seen how a computer stores data and instructions in memory
- We've also seen how instructions can be structured into routines
- Now let's see how memory and instructions can be structured together to model objects in the real world

Object-Oriented Programming

- Key concepts of OOP:
- ***Abstract Data Types***: creating a structure for data (a **class**) to model a set of **objects**
- ***Encapsulation***: expressing all the behavior for an ADT in one place, hiding the details
- ***Overloading***: using the same name for the same concept in different contexts (let the compiler figure it out)
- ***Inheritance***: being able to say *this* ADT is just like *that* ADT, except for *these* differences

Models

- Every program is a model of a system, either real or imaginary
- A flight simulator models physics and aerodynamics as realistically as possible
- A computer game might have some basis in physics, but would also model many imaginary things
- A word processor models something entirely imaginary, a "document"
- All computer models are ultimately stored as bits: data and instructions

Primitive Models

- A computer doesn't really store numbers, it *models* them
  - `int`s and `float`s are two different ways of modeling a real number
- For example, a `float` is a model that consists of
  - A scheme for using bits to represent a real number, along with
  - The operations (`+`, `-`, `*`, `/`, etc.) that can be performed
- `char`s and strings are ways of modeling text
- Booleans are models of true/false values
- Each primitive data type, along with its operators, is a model

Building Models with Models

- We can combine primitive data types and primitive operations to form more sophisticated models
- For example, we might model a calendar date as:
  - A group of 3 integers: month, day, and year
  - A set of routines to operate on dates, such as
    - advancing to the next date
    - counting the number of days between two dates
    - determining which day of the week a date falls on

4 Classes and Objects

- A description of a model is called an Abstract Data Type
- In C++, an ADT is called a `class`
- A `class` consists of C++ declarations for
  - Variables of various data types used to represent the model
  - Routines which operate on objects in the class
- The `class` becomes a new C++ data type
- When we declare a variable of this data type, we are creating an *object*
  - The object is an *instance* of the class

4 Classes and Objects (continued)

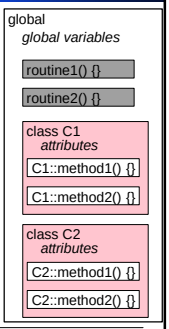
- A class is a blueprint for objects
  - Variables defined in a class are called *attributes*
  - Routines defined in a class are called *methods*

```
class Date {
  int    month;
  int    day;
  int    year;
           ⋮
  void nextDay(void);
  int  dayOfWeek(void);
  void addDays(int x);
           ⋮
};
```

```
Date d1;        month: 11
                day:    2
                year: 1958

Date d2;        month:  6
                day:   24
                year: 2001

Date d3;        month:  2
                day:   15
                year: 1992
```


4 Classes and Namespaces

- In C, each routine has its own local namespace, plus there is one global namespace
- In C++, the support for classes allows us to have more namespaces
- Each class is a separate namespace



4 Other People's Objects

- Before seeing how to build our own classes, let's look at some pre-defined classes (streams)
- When your computer boots, it loads a boot program from ROM which in turn loads the operating system
- The operating system is always running
  - Every program we run can be thought of as a subroutine of the OS
- The OS defines classes and creates objects that our programs can use to communicate with the computer
  - The classes and objects provided for our program are called its *runtime environment*

4 Streams

- Different OSs provide different runtime environments
  - Every OS provides basic input/output classes called *streams*
- `istream` models input streams we can use to read what the user types or from a file
- `ostream` models output streams for displaying text to the user or writing to a file
- `iostream.h` declares the streams classes, and two stream objects, ready for use
  - `istream cin;` for reading the keyboard
  - `ostream cout;` for writing to the display

4 Encapsulation

- The `istream` operator `>>` is implemented via a method belonging to the `istream` class: `operator>>()`
- The `ostream` operator `<<` is the method `operator<<()`
- Streams know how to do sophisticated conversions to/from text
  - Numbers read in are converted from text to integer or floating point values
  - Numbers output are converted from `int`s or `float`s to text
- All of this behavior is ***encapsulated*** in the stream classes, you never have to deal with the details

4 Our Virtual Dogs

- Let's imagine we're writing a program to animate and simulate virtual dogs
  - This would be a fairly large program, we'll be looking at small pieces
- Each dog is an object comprised of other objects, which in turn is comprised of objects ... all the way down to the primitives (`chars`, `ints`, `floats`, pointers)
- The class `Dog` expresses the abstract concept of a dog (what's common to all dog objects)

Snapshot from Dogz² , PF Magic Inc.

4 Where It's At

- Let's model one aspect of the state of our virtual dog: it's location in the playground
  - Imagine the playground is a field of 30 x 20 squares
- Each `Dog` has two *attributes*, `x` and `y`, to record its location
  - `x` and `y` are each short integers


```
class Dog
{
public:
      short int x, y;
};
```


4 Declaring the `Dog` class

`class Dog {`

- The class declaration begins with the reserved word `class`, followed by the name of the class and a `{`

`public:`

- Portions of a class can be hidden, so other classes can't modify them. This statement makes the contents of this class unhidden

`short int x, y;`

- These are the two attributes of this `class`

`};`

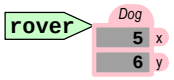
- Note that the class declaration ends with a semicolon

4 Creating an Object

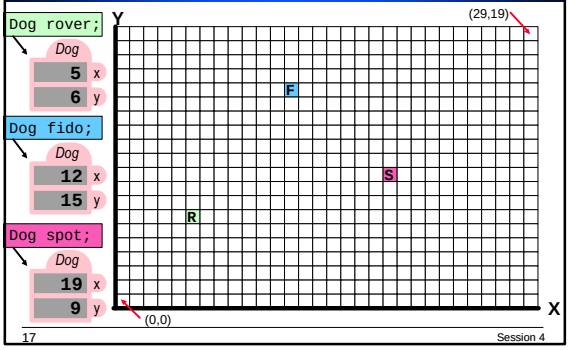
- We've just described what the class of `Dogs` looks like
- To create an object (*instance*) of this class, we declare a variable of this type
  - Our simulation will have many dogs
  - Each dog has its own values for the `Dog` attributes

`Dog rover;`

- Declares a variable of type `Dog`, called `rover`



4 Playground



4 Attributes (Instance Variables)

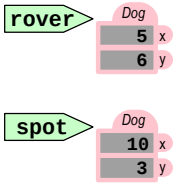
`rover.x`

- This is how we refer to the `x` attribute of the `rover` object
- attributes are also called *instance variables*
  - each instance of the `Dog` class has its own `x` and `y`

```
Dog rover, fido, spot;

rover.x = 5;
rover.y = 6;
fido.x = 12;
fido.y = 15;
...
```

4 Program 4-1

```
#include <iostream.h>

class Dog {
public:
      short int x,y;
};

void main()
{
      Dog   rover, spot;

      rover.x = 5;
      rover.y = 6;
      spot.x = 10;
      spot.y = 3;
```



4 Program 4-1 (continued)

```
      cout << "Rover is at (" << rover.x
           << "," << rover.y << ")" <<  endl;
      cout << "Spot is at (" << spot.x
           << "," << spot.y << ")" <<  endl;

      rover.x++;
      spot.y += 2;
      cout << "Rover is now at (" << rover.x
           << "," << rover.y << ")" <<  endl;
      cout << "Spot is now at (" << spot.x
           << "," << spot.y << ")" <<  endl;
}
```

Output

```
Rover is at (5,6)
Spot is at (10,3)
Rover is now at (6,6)
Spot is now at (10,5)
```


4 Weak Encapsulation

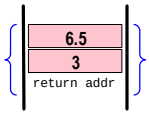
- The advantage of object-oriented design is to encapsulate *everything* relevant to an object into one unit
- Our description of the Dog class should include not only
  - the attributes of an object in the class, but also
  - the methods which operate on an object in the class
- Our first design for Dog is poor, since all the work is done outside the class (in `main()`)

4 Methods

- A *method* is a routine associated with a class
- A method is always invoked *on* an object, it can't be called without the object

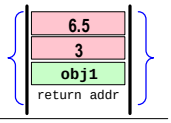
Routine

```
Declaration:
  void routine1( int a, float b);
Use:
  routine1(3,6.5);
```



Method

```
Declaration:
  void Class1::method1( int a, float b);
Use:
  Class1 obj1;
  obj1.method1(3,6.5);
```



4 Methods: What's In Scope

- A variable name in a method can refer to:
  - a global variable
  - an attribute (instance variable) of the object
  - an argument to the method
  - a local variable

```
Dog rover;
rover.MoveX(2);
```

```
int dogMoved = 0;

void Dog::MoveX(int amount)
{
      int true = 1;

      x += amount;
      dogMoved = true;
}
```

attribute `rover.x` → `x += amount;`

argument → `amount`

local → `true`

global → `dogMoved = true;`

4 Code Files

- The text of our programs (also called the source code, or just code) will often be split among many files
- For each object Foo, there's a header file **Foo.h** which has the class declaration
  - The class declaration includes the declarations for all of the method routines
- The actual instructions for the methods go in **Foo.cpp**
  - Some people use **Foo.cxx**, some use **Foo.cc**
- Short methods can be defined as part of the class declaration in the **Foo.h** file

4 Virtual Dog Methods

- The first methods we'd like to add to our `Dog` class are:
- `void Dog::Home(void)`
  - Relocate the dog at the home position (0,0)
- `void Dog::MoveX(int amount)`
  - Move the dog in the X direction by `amount` squares
- `void Dog::MoveY(int amount)`
  - Move the dog in the Y direction by `amount` squares

4 Comments

- Before we show the next program, we'd like to briefly mention *comments*
- A comment is text you've written in a C++ code file that you want the compiler to totally ignore
  - We use comments to write notes for ourselves, mostly to capture what was going through our heads when writing the program
- C++ has two types of comments:
- Everything after `//` is ignored, to the end of the line
- Everything between `/*` and `*/` is ignored, even across multiple lines

4 Program 4-2

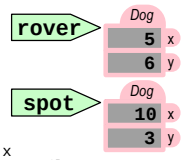
```
/*
 * Dog location Version 2
 */
#include <iostream.h>

class Dog {
public:
      short int x,y;

      void Home(void);
      void MoveX(int amount);
      void MoveY(int amount);
};
```

4 Program 4-2 (continued)

```
/** Methods **/

// Send a Dog home (0,0)
void Dog::Home(void)
{
      x = 0;
      y = 0;
}

// Move a Dog in the X direction
void Dog::MoveX(int amount)
{
      x += amount;
}

// Move a Dog in the Y direction
void Dog::MoveY(int amount)
{
      y += amount;
}
```


4 Program 4-2 (continued)

```
void main()
{
      Dog rover, spot;

      rover.x = 5;
      rover.y = 6;
      spot.x = 10;
      spot.y = 3;
      cout << "Rover is at (" << rover.x
           << "," << rover.y << ")" <<  endl;
      cout << "Spot is at (" << spot.x
           << "," << spot.y << ")" <<  endl;
```



4 Program 4-2 (end)

```
      rover.MoveX(1);
      cout << "Rover is at (" << rover.x
           << "," << rover.y << ")" << endl;

      rover.MoveY(-3);
      cout << "Rover is at (" << rover.x
           << "," << rover.y << ")" << endl;

      spot.Home();
      cout << "Spot is at (" << spot.x
           << "," << spot.y << ")" << endl;
}
```

Output

```
Rover is at (5,6)
Spot is at (10,3)
Rover is at (6,6)
Rover is at (6,3)
Spot is at (0,0)
```

4 Notes on Program 4-2

`rover.MoveX(1);`

- Invokes the method `Dog::MoveX()` on the object `rover`, with the argument 1

- Note that inside a `Dog` method, we refer to the current object's X attribute as simply `x`
  - The object being operated on is part of the context for each of the methods

4 Controlling Attribute Values

- We want to move as much of the details regarding dogs into the `Dog` class
- Currently, we're still setting the initial value for the location outside of the class, in `main()`
  - What if someone using our class forgets to initialize it?
- We also don't want to allow `main()` to move the dog outside the playground
  - `x` should be in the range [0..29]
  - `y` should be in the range [0..19]
- Our encapsulation is not yet complete

4 Constructors

- C++ classes can define a method that is called when the object is created
  - This method is called the class's *constructor*
  - The constructor is a method with the same name as the class
  - Constructors do not have a return value, not even `void`
- By requiring that the constructor takes one or more arguments, we can insure that a user of our class provides the info we need

```
Dog::Dog(short int x0, short int y0);
```


4 Constructors (continued)

- In `main()`:

```
Dog rover(5,6);
```

  - creates a `Dog` object called `rover`, while
  - providing the arguments required by the constructor
- Any attempt to create a `Dog` without specifying the arguments will generate an error

4 Fence() method

- To keep the dog within the playground, we can write a method to check the values of `x` and `y` and change them if they're out of range
- Inside a method, we can call another method from the same class without specifying an object
  - the submethod operates on the same object as the method, accessing the same attributes
- We'll call this method `Fence()`, because it keeps the dog in the playground, and we'll call it each time we move the dog

4 Program 4-3

```
/*
 * Dog location Version 3
 */
#include <iostream.h>

class Dog {
public:
      short int x,y;

      // Constructor
      Dog(short int x0, short int y0);

      // Movement methods
      void Home(void);
      void MoveX(int amount);
      void MoveY(int amount);
      void Fence();
};
```

4 Program 4-3 (continued)

```
// Constructor
//
Dog::Dog(short int x0, short int y0)
{
      x = x0;
      y = y0;
      Fence();
}

/** Methods **/

// Send a Dog home (0,0)
void Dog::Home(void)
{
      x = 0;
      y = 0;
}
```


4 Program 4-3 (continued)

```
// Move a Dog in the X direction
//
void Dog::MoveX(int amount)
{
      x += amount;
      Fence();
}

// Move a Dog in the Y direction
//
void Dog::MoveY(int amount)
{
      y += amount;
      Fence();
}
```


4 Program 4-3 (continued)

```
// Contain the Dog within a 30 x 20 playground
//
void Dog::Fence(void)
{
      if(x < 0)
            x = 0;

      if(x > 29)
            x = 29;

      if(y < 0)
            y = 0;

      if(y > 19)
            y = 19;
}
```


4 Program 4-3 (end)

```
void main()
{
      Dog rover(5,6), spot(10,3);

      cout << "Rover is at (" << rover.x
           << "," << rover.y << ")" <<  endl;
      cout << "Spot is at (" << spot.x
           << "," << spot.y << ")" <<  endl;

      rover.MoveX(1);
      cout << "Rover is at (" << rover.x
           << "," << rover.y << ")" <<  endl;

      spot.MoveY(-9);
      cout << "Spot is at (" << spot.x
           << "," << spot.y << ")" << endl;

      spot.Home();
      cout << "Spot is at (" << spot.x
           << "," << spot.y << ")" <<  endl;
}
```


4 Program 4-3 Output

```
Rover is at (5,6)
Spot is at (10,3)
Rover is at (6,6)
Spot is at (10,0)
Spot is at (0,0)
```


4 True Encapsulation

- When we encapsulate an object, we want to hide how it's implemented
  - That way we can change the implementation later, without affecting the rest of the program
- The constructor and methods for `Dog` should be public, but the fact that the location is stored as two short integers named `x` and `y` is part of the implementation
- We want to hide `x` and `y` , but we'll need to add public methods to retrieve the values called `Dog::GetX()`and `Dog::GetY()`

4 Program 4-4

```
/** Dog location Version 4 **/
#include <iostream.h>

class Dog {
private:
      short int x,y;

public:
      // Constructor
      Dog(short int x0, short int y0);

      // Access to private attributes
      short int GetX(void);
      short int GetY(void);

      // Movement methods
      void Home(void);
      void MoveX(int amount);
      void MoveY(int amount);
      void Fence(void);
};
```

4 Program 4-4 (continued)

```
// Constructor
Dog::Dog(short int x0, short int y0)
{
      x = x0;
      y = y0;
      Fence();
}

/** Methods **/

// Return the value of the private attribute x
short int Dog::GetX(void)
{
      return(x);
}

// Return the value of the private attribute y
short int Dog::GetY(void)
{
      return(y);
}
```


4 Program 4-4 (continued)

```
// Send a Dog home (0,0)
void Dog::Home(void)
{
      x = 0;
      y = 0;
}

// Move a Dog in the X direction
void Dog::MoveX(int amount)
{
      x += amount;
      Fence();
}

// Move a Dog in the Y direction
void Dog::MoveY(int amount)
{
      y += amount;
      Fence();
}
```


4 Program 4-4 (continued)

```
// Contain the Dog within a 30 x 20 playground
//
void Dog::Fence(void)
{
      if(x < 0)
            x = 0;

      if(x > 29)
            x = 29;

      if(y < 0)
            y = 0;

      if(y > 19)
            y = 19;
}
```


4 Program 4-4 (end)

```
void main()
{
      Dog rover(5,6), spot(10,3);

      cout << "Rover is at (" << rover.GetX()
           << "," << rover.GetY() << ")" << endl;
      cout << "Spot is at (" << spot.GetX()
           << "," << spot.GetY() << ")" << endl;

      rover.MoveX(1);
      cout << "Rover is at (" << rover.GetX()
           << "," << rover.GetY() << ")" << endl;

      spot.MoveY(-9);
      cout << "Spot is at (" << spot.GetX()
           << "," << spot.GetY() << ")" << endl;

      spot.Home();
      cout << "Spot is at (" << spot.GetX()
           << "," << spot.GetY() << ")" << endl;
}
```


4 Program 4-4 Output

```
Rover is at (5,6)
Spot is at (10,3)
Rover is at (6,6)
Spot is at (10,0)
Spot is at (0,0)
```

4 Notes on Program 4-4

- The instance variables, `short int x,y;`, are now declared in a section of the class marked `private`
  - If `main()` or any other routine that's not a method of `Dog` attempts to access the `x` or `y` variable, the compiler will generate an error
- All the methods, including `Dog::GetX()` and `Dog::GetY()`, are public

4 Interface

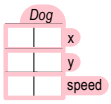
- The public portions of a class are the *interface* seen by other classes
  - We are declaring that certain methods are valid for objects of a class, and how they are called
- In order to break a large project up into smaller pieces, we want to avoid changing the interfaces between classes
- As long as we don't change the names, arguments, or return values of our public methods, we can change anything else

4 Extending an Interface

- For example, let's say our `Dog` class needs another attribute: the speed at which a particular dog travels
- We'll need to add a public method for changing the speed
  - This extends the `Dog` interface
- However, we'll set a default speed of 1 in the constructor
  - Programs written using the old interface won't need to change
- This is called *backwards compatibility*

4 Program 4-5

```
/*
 * Dog location Version  5
 */
#include <iostream.h>

class Dog {
private:
      short int x,y;
      short int speed;

public:
      // Constructor
      Dog(short int x0, short int y0);

      // Access to private attributes
      short int GetX(void);
      short int GetY(void);

      // Set a new speed
      void ChangeSpeed(int s);
```



4 Program 4-5 (continued)

```
      // Movement methods
      void Home(void);
      void MoveX(int amount);
      void MoveY(int amount);
      void Fence(void);
};

// Constructor
//
Dog::Dog(short int x0, short int y0)
{
      x = x0;
      y = y0;
      speed = 1;

      Fence();
}
```


4 Program 4-5 (continued)

```
/** Methods **/

// Set the private attribute speed
//
void Dog::ChangeSpeed(int s)
{
      speed = s;
}

// Return the value of the private attribute x
short int Dog::GetX(void)
{
      return(x);
}

// Return the value of the private attribute y
short int Dog::GetY(void)
{
      return(y);
}
```

4 Program 4-5 (continued)

```
// Send a Dog home (0,0)
void Dog::Home(void)
{
      x = 0;
      y = 0;
}

// Move a Dog in the X direction
void Dog::MoveX(int amount)
{
      x += (amount * speed);
      Fence();
}

// Move a Dog in the Y direction
void Dog::MoveY(int amount)
{
      y += (amount * speed);
      Fence();
}
```


4 Program 4-5 (continued)

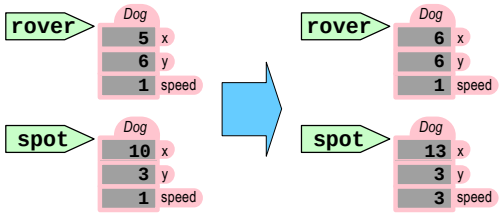
```
// Contain the Dog within a 30 x 20 playground
//
void Dog::Fence(void)
{
      if(x < 0)
            x = 0;

      if(x > 29)
            x = 29;

      if(y < 0)
            y = 0;

      if(y > 19)
            y = 19;
}
```


4 Program 4-5 (end)

```
void main()
{
      Dog rover(5,6), spot(10,3);

      cout << "Rover is at (" << rover. GetX()
           << "," << rover. GetY() << ")" << endl;
      cout << "Spot is at (" << spot. GetX()
           << "," << spot.GetY() << ")" << endl;

      rover.MoveX(1);
      spot.ChangeSpeed(3);
      spot.MoveX(1);

      cout << "Rover is at (" << rover.GetX()
           << "," << rover.GetY() << ")" << endl;
      cout << "Spot is at (" << spot.GetX()
           << "," << spot.GetY() << ")" << endl;
}
```


4 Program 4-5 Output

```
Rover is at (5,6)
Spot is at (10,3)
Rover is at (6,6)
Spot is at (13,3)
```



4 Notes on Program 4-5

- Note that we've changed the instance variables and the insides of many of the methods, but not the name, arguments, or return values of the methods
- The first part of `main()` is unchanged, and uses a `Dog` with the default speed of 1
- The last part of `main()` tests the new feature by changing the speed for `spot`

4 More Enhancements

- The current version of `Dog` can only handle speeds greater than one
- Whenever we do arithmetic with integers, and the result has a fractional portion, that fraction is lost:
  - 5/2 is 2.5, which would get stored as 2 in an integer
  - In fact, 2/3 = 0!
- To handle rates like 0.5, we need to store the attributes using floating point
- Our `Dog` class will look, on the outside, as if the location were two integers, but internally it will be represented as two floats

Rounding Numbers

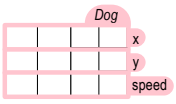
- We can force a floating point expression to be converted to an integer using a *typecast*, the name of a data type in parentheses:

```
float f = 98.6;
int i = (int) f;
```

- When a `float` or `double` is converted to an `int`, the fractional portion is lost
  - `(int) 1.999 = 1`
- To round off to the nearest `int`, we add 0.5 first
  - `(int) (1.999 + 0.5) = (int) (2.499) = 2`
  - `(int) (1.111 + 0.5) = (int) (1.611) = 1`

Program 4-6

```
/* Dog location Version 6 */
#include <iostream.h>

class Dog {
private:
      float x,y;
      float speed;

public:
      // Constructor
      Dog(short int x0, short int y0);

      // Access to private attributes
      short int GetX(void);
      short int GetY(void);

      // Set a new speed, two versions
      void ChangeSpeed(short int s);
      void ChangeSpeed(double s);
```



Notes on Program 4-6

- We've changed the declarations of `x, y` and `speed` to `floats`
- Note that there are two versions of the `ChangeSpeed ( )` method, one which takes an integer as an argument, and one that takes a floating point value
- This is called *overloading* (see below)
- Since the old `ChangeSpeed ( )` is still available, we're still backwards compatible

Program 4-6 (continued)

```
      // Movement methods
      void Home(void);
      void MoveX(int amount);
      void MoveY(int amount);
      void Fence(void);
};

// Constructor
//
Dog::Dog(short int x0, short int y0)
{
      x = x0;
      y = y0;
      speed = 1.0;

      Fence();
}
```


Program 4-6 (continued)

```
/** Methods **/

// Set the private attribute speed,
// given an integer argument.
//
void Dog::ChangeSpeed(short int s)
{
      speed = (float) s;
}

// Set the private attribute speed,
// given a floating point argument.
//
void Dog::ChangeSpeed(double s)
{
      speed = s;
}
```


Program 4-6 (continued)

```
// Return the value of the private attribute x,
// rounded to the nearest integer value.
short int Dog::GetX(void)
{
      return((short int) (x + 0.5));
}

// Return the value of the private attribute y
// rounded to the nearest integer value.
short int Dog::GetY(void)
{
      return((short int) (y + 0.5));
}

// Send a Dog home (0,0)
void Dog::Home(void)
{
      x = 0;
      y = 0;
}
```

Notes on Program 4-6 (continued)

- `Dog::GetX()` and `Dog::GetY()` have been redefined to round off the internal floating point values and return integers

Program 4-6 (continued)

```
// Move a Dog in the X direction
//
void Dog::MoveX(int amount)
{
      x += (amount * speed);
      Fence();
}

// Move a Dog in the Y direction
//
void Dog::MoveY(int amount)
{
      y += (amount * speed);
      Fence();
}
```


Program 4-6 (continued)

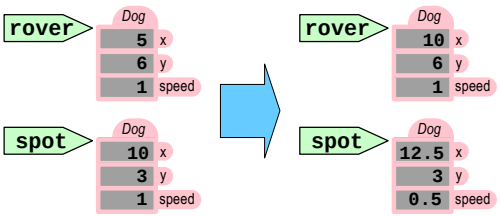
```
// Contain the Dog within a 30 x 20 playground
//
void Dog::Fence(void)
{
      if(x < 0)
            x = 0;

      if(x > 29)
            x = 29;

      if(y < 0)
            y = 0;

      if(y > 19)
            y = 19;
}
```


Program 4-6 (end)

```
void main()
{
      Dog rover(5,6), spot(10,3);

      cout << "Rover is at (" << rover.GetX()
           << "," << rover.GetY() << ")" << endl;
      cout << "Spot is at (" << spot.GetX()
           << "," << spot.GetY() << ")" << endl;

      rover.MoveX(5);
      spot.ChangeSpeed(0.5);
      spot.MoveX(5);

      cout << "Rover is at (" << rover.GetX()
           << "," << rover.GetY() << ")" << endl;
      cout << "Spot is at (" << spot.GetX()
           << "," << spot.GetY() << ")" << endl;
}
```


Program 4-6 Output

```
Rover is at (5,6)
Spot is at (10,3)
Rover is at (10,6)
Spot is at (13,3)
```



Overloading

- One big advantage that methods give us over plain-vanilla routines is *overloading*
- We can have more than one method with the same name, if the arguments are of different types
- The compiler automatically picks the right version of an overloaded method
- `rover.ChangeSpeed(2);` invokes
  - `void Dog::ChangeSpeed(int s);`
- `rover.ChangeSpeed(2.5);` invokes
  - `void Dog::ChangeSpeed(double s);`

4 Overloading (continued)

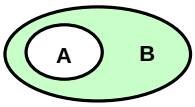
- Overloading is an important part of object-oriented programming, it is much less complex than having to give each `ChangeSpeed ( )` method a different name

4 Dealing With Complexity

- As programs get more complex, the classes get more complex
- Grouping data and routines together to form objects is the first step in dealing with the complexity
- C++ also provides a means of expressing the relationships between classes, called *inheritance*
- Inheritance is a means of saying "this class is just like that class, *except*..."

4 Inheritance

- Let's say we have a class **A**
- We can define a new class **B** that *inherits from* **A**
- Every attribute of **A** is also an attribute of **B**
  - B may have additional attributes of its own
- Every method of **A** is also a method of **B**
  - B may have additional methods of its own
- **A** is called the *base* class, **B** is the *derived* class



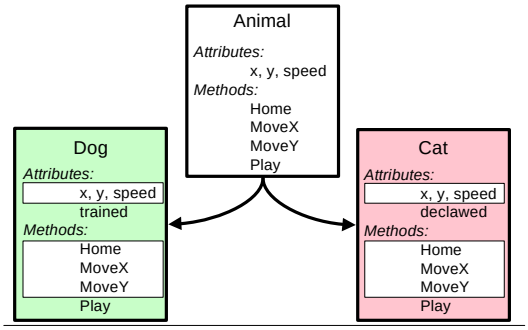
4 Dogs and Cats

- Let's extend our simulation to include cats as well as dogs
  - We'll create a new class, `Cat`, to describe a cat
- In the course of thinking about modeling dogs and cats, we realize they have many similarities
- We'll create a base class, `Animal`, to model the similarities
- The classes `Dog` and `Cat` will be derived from `Animal`

4 Dogs and Cats (continued)

- The features that dogs and cats share are modeled as attributes and methods of `Animal`
  - attributes: location, speed
  - methods: movement
- The features that differentiate dogs and cats are implemented in the `Dog` and `Cat` classes
  - attributes: only dogs can be "trained"
  - attributes: only cats can be "declawed"
  - methods: dogs and cats prefer to play with different things

4 `Animal` Hierarchy



4 Program 4-7

```
/** Animal, Dog and Cat  **/
#include <iostream.h>

class Animal {
private:
      float x,y;
      float speed;

public:
      // Constructor
      Animal(short int x0, short int y0);

      // Access to private attributes
      short int GetX(void);
      short int GetY(void);

      // Set a new speed, two versions
      void ChangeSpeed(int s);
      void ChangeSpeed(double s);
```


4 Program 4-7 (continued)

```
      // Movement methods
      void Home(void);
      void MoveX(int amount);
      void MoveY(int amount);
      void Fence(void);
};

class Dog : public Animal {
public:
      bool trained;

      // Constructor
      Dog(void);

      // Methods
      void Play(void);
};
```


4 Program 4-7 (continued)

```
class Cat : public Animal {
public:
      bool declawed;

      // Constructor
      Cat(void);

      //Methods
      void Play(void);
};

/** Animal methods  **/
// Constructor
Animal::Animal(short int x0, short int y0)
{
      x = x0;
      y = y0;
      speed = 1.0;
      Fence();
}
```


4 Program 4-7 (continued)

```
/*
 * Methods
 */

// Set the private attribute speed,
// given an integer argument.
//
void Animal::ChangeSpeed(int s)
{
      speed = (float) s;
}

// Set the private attribute speed,
// given a double argument.
//
void Animal::ChangeSpeed(double s)
{
      speed = s;
}
```


4 Program 4-7 (continued)

```
// Return the value of the private attribute x,
// rounded to the nearest integer value.
short int Animal::GetX(void)
{
      return((short int) (x + 0.5));
}

// Return the value of the private attribute y
// rounded to the nearest integer value.
short int Animal::GetY(void)
{
      return((short int) (y + 0.5));
}

// Send an Animal home (0,0)
void Animal::Home(void)
{
      x = 0;
      y = 0;
}
```


4 Program 4-7 (continued)

```
// Move an Animal in the X direction
//
void Animal::MoveX(int amount)
{
      x += (amount * speed);
      Fence();
}

// Move an Animal in the Y direction
//
void Animal::MoveY(int amount)
{
      y += (amount * speed);
      Fence();
}
```

4 Program 4-7 (continued)

```
// Contain the Animal within a 30 x 20 playground
//
void Animal::Fence(void)
{
      if(x < 0)
            x = 0;

      if(x > 29)
            x = 29;

      if(y < 0)
            y = 0;

      if(y > 19)
            y = 19;
}
```


4 Program 4-7 (continued)

```
/*
 * Dog Methods
 */

// Construct a Dog, at (0,0)
//
Dog::Dog(void) : Animal(0,0)
{
      trained = false;
}

// Play with a Dog
//
void Dog::Play(void)
{
      cout << "Let's play fetch!"  << endl;
}
```


4 Program 4-7 (continued)

```
/*
 * Cat Methods
 */

// Construct a Cat, at (29,19)
//
Cat::Cat(void) : Animal(29,19)
{
      declawed = false;
}


// Play with a Cat
//
void Cat::Play(void)
{
      cout << "Let's play with a ball of yarn!"  << endl;
}
```


4 Program 4-7 (end)

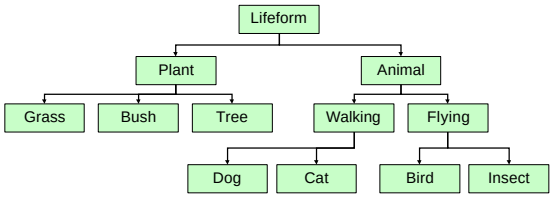
```
/*
 * Main routine to test Animal, Dog and Cat classes
 */
void main()
{
      Dog rover;
      Cat fluffy;

      cout << "Rover is at (" << rover.GetX()
           << "," << rover.GetY() << ")" << endl;
      cout << "Fluffy is at (" << fluffy.GetX()
           << "," << fluffy.GetY() << ")" << endl;

      cout << "Rover says: ";
      rover.Play();

      cout << "Fluffy says: ";
      fluffy.Play();
}
```


4 Program 4-7 Output

```
Rover is at (0,0)
Fluffy is at (29,19)
Rover says: Let's play fetch!
Fluffy says: Let play with a ball of yarn!
```


4 Inheritance Trees

- The animal example consists of one ancestor and two descendants
- Here's a larger view of more of the lifeforms we might want to model:



Great Ancestors

- A class can access any of the attributes and methods of its ancestor, and its ancestor's ancestor, and its ancestor's ancestor's ancestor...
- For example, if `Lifeform` contains an attribute `float age`, then every class in the previous diagram contains an `age`
  - Each `Lifeform` has its own `age`
- We can treat any object in any subordinate class as if it were a `Lifeform`

91 Session 4

is-a and has-a

- Two of the ways classes X and Y can be related:
- *is-a*: X inherits from Y (X *is a* Y)
- *has-a*: X contains a Y (X *has a* Y)
- For example, how would we fit a `Color` class into our `Lifeform` tree?
  - Each lifeform **has a** color
  - It would not make sense to say "a lifeform **is a** color"
- `Color` would be a separate class. A `Color` object would be one of the attributes of `Lifeform`, so every lifeform would have a color
  - Think in terms of *is-a* and *has-a* when designing classes

92 Session 4

So Far

- We've now covered four main features of object oriented programming:
  - Abstract Data Types
  - Encapsulation
  - Overloading
  - Inheritance
- Next session we'll look at how C++ helps us manipulate memory addresses via pointers

93 Session 4

5

Introduction to Programming

Pointers & Arrays

Session 5

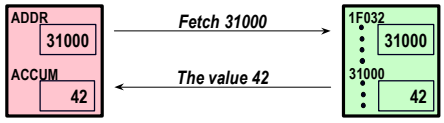
Phil Mercurio
UCSD Extension
mercurio@acm.org

5 Recap

- We've seen how to structure memory in the form of objects: data and instructions grouped together
- In this session, we'll see
  - how to deal with memory directly (*pointers*), and
  - how to handle lists of objects (*arrays*)

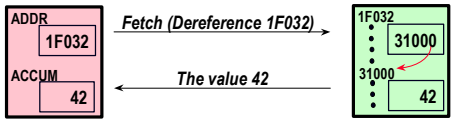
5 Indirection

- The fundamental principle we'll be covering today is *indirection*
- When the processor requests the contents of an address from memory, it is accessing that value *directly*
  - The value's address is explicitly stated



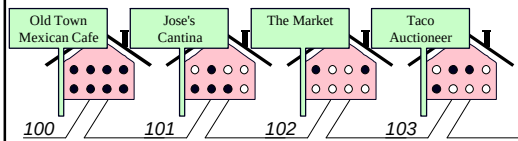
5 Indirection (continued)

- Instead of stating the address explicitly, what if we store the address somewhere else in memory?
- That stored address is a *pointer*, and accessing a value indirectly, through a pointer, is called *dereferencing* (the pointer is a *reference* to the value)



5 Example of Indirection

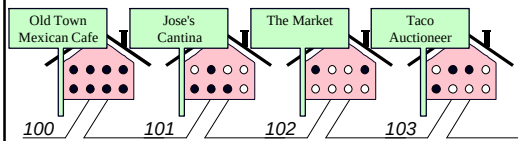
- Let's return to our model of memory as a street of houses
  - Instead of houses, we'll use restaurants
- Each restaurant has a name as well as an address
  - This is analogous to variables in C++, each of which has a name and an address in memory



5 Example of Indirection (continued)

- Let's say you're planning to get a group of friends together for a Mexican dinner tomorrow night
- Tom knows of a great restaurant, but he can't remember its name or address
  - He can find out tomorrow, though

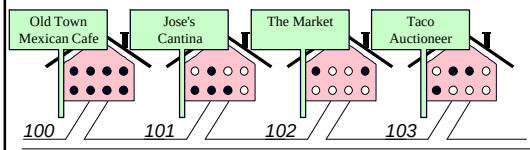
Rick
Sally
John
Jane
Tom ???



5 Example of Indirection (continued)

- Everyone knows where the Old Town Mexican Cafe is
- You plan to meet at the Cafe

Rick OTMC
Sally OTMC
John OTMC
Jane OTMC
Tom OTMC

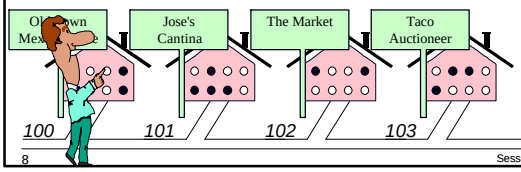


7 Session 5

5 Example of Indirection (continued)

- By the time Tom gets to the Cafe, he's found out the name of the restaurant (the Market) and its address (102)
- Tom redirects everyone to the Market by storing 102 at the Cafe

Rick OTMC
Sally OTMC
John OTMC
Jane OTMC
Tom Market

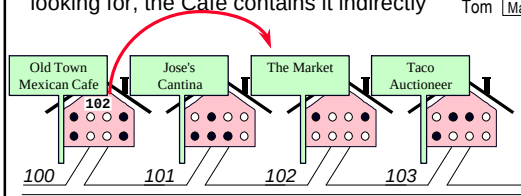


8 Session 5

5 Example of Indirection (continued)

- We've specified the Market indirectly, by storing a reference to it (102) at the Cafe
- The Cafe *points to* the Market
- The Market contains the value we're looking for, the Cafe contains it indirectly

Rick Market
Sally Market
John Market
Jane Market
Tom Market



9 Session 5

5 Why Indirection?

- This example shows some of the reasons indirection is so powerful
- Instead of hungry friends, we're concerned with different parts of our program
  - Different objects
  - Different routines
- Pointers allow us to manage access to information stored in memory

10 Session 5

5 Sharing Information

- We can use pointers to share information among different parts of a program
- In this example, the Market's location was shared by giving each friend a reference to it (the Cafe)

11 Session 5

5 Filling In Later

- Note that, at the time each friend was supplied with the reference (told to go to the Cafe), the required information (the location of the Market) was not stored there yet
- Pointers make it possible to fill in information later
- They also make it easy to change information (Tom could have decided to redirect everyone to Jose's instead)

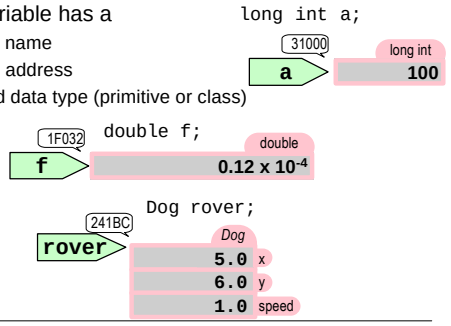
12 Session 5

5 What is a Variable?

- We previously (Session 2) defined a variable as "a piece of memory managed by the compiler"
  - More precisely, it is managed by instructions the compiler adds to our program
- A variable can be either a primitive or an object, so it can be any amount of bytes of memory
- A variable has a name, which is unique within its scope
- But a variable also has an address in memory, which is unique everywhere

5 Variables

- Each variable has a
  - unique name
  - unique address
  - defined data type (primitive or class)



5 Pointers vs. Addresses

- Up to now, we've been using the terms *address* and *pointer* interchangeably
- An address is the location of a byte in memory (we can think of it as a numerical value)
- A pointer is a C++ variable that can contain an address
  - Similar to how a `long int` is a variable that can contain a 32-bit signed integer value
- A pointer is more powerful than just an address, because it also includes a data type: the type of the data stored at the address

5 Declaring Pointers

*<data type> *<pointer>;*

- Pointers are declared just like variables, with the pointer name preceded by a *

```
short int count = 10;
```

- Declares a `short int` called `count`, and initializes it to 10

```
short int *intPtr;
```

- Declares a variable called `intPtr`, of type *pointer to* `short int`
- All pointers are the same size, 32 bits (the size of an address)

5 Pointer Data Types

- Remember, the important concept here is indirection (dereferencing): using a pointer to get or set a value in memory
- Because a pointer has an associated data type, the compiler knows *how* we want to dereference the pointer

```
short int *intPtr;
```

  - Creates a variable called `intPtr` which can contain an address
  - The two bytes at the address are to be interpreted as a 16-bit signed integer
  - When we dereference `intPtr`, we'll get back a `short int`

5 Pointer Operators

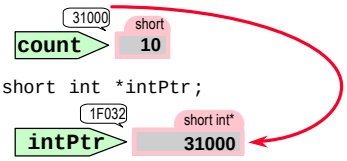
- Pointers can be used in expressions, just like any other variable
- There are two special operators for pointers:
  - & The Address operator
  - * The Indirection operator

5 & The Address Operator

- &<*variable*> gets the address of <*variable*>
  - `intPtr = &count;` stores the address of `count` in the pointer `intPtr`

```
short int count = 10;
short int *intPtr;
```

intPtr = &count;

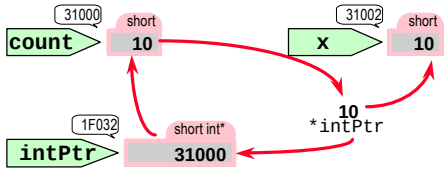


19 Session 5

5 * The Indirection Operator

- *<*pointer*> dereferences <*pointer*>
  - `x = *intPtr;` takes the address in `intPtr`, gets the value stored at that address, and copies the value into x

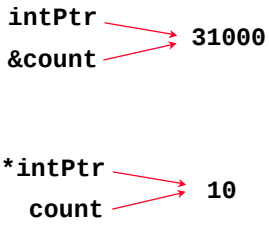
x = *intPtr;



20 Session 5

5 Duality

- If `intPtr` is set equal to `&count`, and `count` contains the value 10, then:
  - There are two ways to refer to the address 31000
  - There are two ways to refer to the value 10



21 Session 5

5 Program 5-1

```
/* Program to illustrate pointers */
#include <iostream.h>

void main()
{
      short int   x, count = 10;
      short int   *intPtr;

      intPtr = &count;
      x = *intPtr;
      *intPtr = 12;

      cout << "count = " << count
           << " x = " << x << endl;
}
```

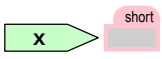
Output

```
count = 12 x = 10
```

22 Session 5

5 Program 5-1 Animation #1

```
short int x
```



23 Session 5

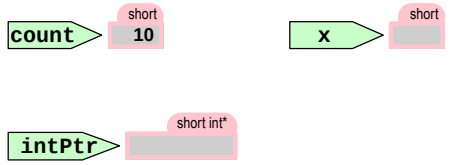
5 Program 5-1 Animation #2

```
short int x, count = 10;
```

24 Session 5

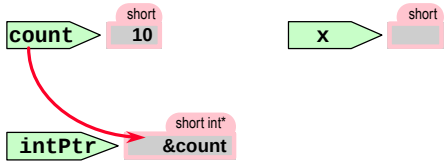
5 Program 5-1 Animation #3

```
short int x, count = 10;
short int *intPtr;
```



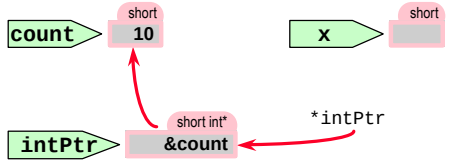
5 Program 5-1 Animation #4

```
short int x, count = 10;
short int *intPtr;
intPtr = &count;
```



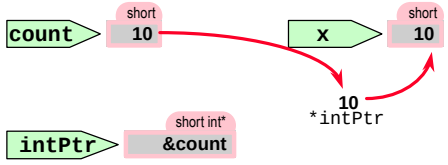
5 Program 5-1 Animation #5

```
short int x, count = 10;
short int *intPtr;
intPtr = &count;
x = *intPtr;
```



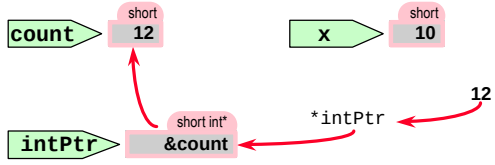
5 Program 5-1 Animation #6

```
short int x, count = 10;
short int *intPtr;
intPtr = &count;
x = *intPtr;
```



5 Program 5-1 Animation #7

```
short int x, count = 10;
short int *intPtr;
intPtr = &count;
x = *intPtr;
*intPtr = 12;
```



5 Pointers and Arguments

- A routine passes the value of a variable to a subroutine by copying it onto the stack
- The subroutine has no access to the routine's variable, and can't change its value
- By passing a pointer to the variable, the subroutine *can* change its value
- There are other ways of passing info from a subroutine back to its routine, but this is a good way to return more than one value

5 Squaring a Complex Number

- Dealing with complex numbers is one instance where we might want a function to return two values
- $a + bi$ is a complex number, a is the real part and b is the imaginary part (i is the square root of -1)

$$(a + bi)^2 =$$

$$(a + bi)(a + bi) =$$

$$a(a + bi) + bi(a + bi) =$$

$$a^2 + abi + abi + (bi)^2 =$$

real — imaginary

$$a^2 - b^2 + 2abi$$

5 Program 5-2

```
/* Square a complex number */
#include <iostream.h>

void squareComplex(double *aPtr, double *bPtr)
{
      double a, b, aAns, bAns;

      // Retrieve a and b from the pointers
      a = *aPtr;
      b = *bPtr;

      // Compute the real and imaginary parts of answer
      aAns = (a*a) - (b*b);
      bAns = 2*a*b;

      // Return the answer via the pointers
      *aPtr = aAns;
      *bPtr = bAns;
}
```


5 Program 5-2 (continued)

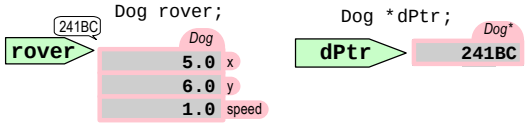
```
void main()
{
      double      a = 4.0, b = 1.5;

      squareComplex(&a, &b);

      cout << "Answer: (" << a << " + "
           << b << "i)" << endl;
}
```

Output

```
Answer: (13.75 + 12i)
```


5 Pointers and Objects

- Pointers can be created for any C++ data type, including classes
- `Dog rover;`
  - Declares an object of class `Dog`, called `rover`
- `Dog *dPtr;`
  - Declares a pointer to a `Dog`, called `dPtr`



5 Pointers and Objects (continued)

- `dPtr = &rover;`
  - Sets `dPtr` to point to `rover`
- `(*dPtr).x` now refers to the same `float` as `rover.x`
- This is such a common idiom, C++ has a special operator for it:
- `dPtr->x` is another way of saying `(*dPtr).x`
- `->` is the dereferencing operator, it retrieves an attribute of an object, given a pointer to it

5 Pointers and Objects Example

- Here's a piece of a C++ routine that sets a pointer to the leftmost of three dogs and moves it around:

```
Dog rover, spot, tippy;
Dog *leftmost;

if(rover.GetX() < spot.GetX())
      leftmost = &rover;
else
      leftmost = &spot;

if(tippy.GetX() < leftmost->GetX())
      leftmost = &tippy;

leftmost->ChangeSpeed(2);
leftmost->MoveX(3);
leftmost->MoveY(-4);
leftmost->ChangeSpeed(0.5);
leftmost->MoveX(-3);
```

5 Managing Memory

- The compiler sets aside memory for variables as they are needed
  - Global variables are created before the program begins
  - Local variables are stored on the stack while their routine is active
- Using pointers and objects, we can manage memory directly
- C++ has two keywords for memory management:
- (For the examples that follow, assume `Dog *dPtr;`)

5 new

`new <class>(<constructor args>)`

- new creates a new object of the specified *<class>*
- If the constructor for the class takes arguments, they are included in the new command
- the result of the expression is a pointer to the new object

```
dPtr = new Dog(5,6);
```

- Allocates memory for a new `Dog`
- Initializes it by calling its constructor with (5,6) as the arguments
- Stores a pointer to the new `Dog` in `dPtr`

5 delete

`delete <pointer>;`

- `delete` destroys (frees) memory that was created by `new`
- a class that has a constructor method may also have a *destructor* method, if so, `delete` calls the destructor first

```
class Dog {
   Dog(short int x, short int y); // constructor
   ~Dog(); // Destructor
}
```

- The name of the destructor method is the same as that of the class, with a ~ in front

5 Dynamic Memory

- The ability to create and destroy objects in memory is called *dynamic memory*
- Dynamic memory is a powerful way to model objects in the real world
- We can imagine a virtual dog simulation with dozens of `Dog` objects
- The simulation cycles in a `while` loop, updating all the currently active `Dogs` each time through the loop

5 Dynamic Memory (continued)

- When a new dog is born, an object is created with `new`
  - The `Dog` constructor takes care of creating all the parts of a dog
- When that dog's life is over, the object is deleted
  - The `Dog`'s destructor deletes all the dog parts

5 Arrays

- How would we manage a list of pointers to all the active dogs?
- For that matter, how would we manage a list of anything?
- C++ provides a means for handling lists, called *arrays*
- An array is a sequence of primitives or objects:
  - all of the same data type*, and
  - stored in consecutive locations in memory
- (*By contrast, a class is used to group objects or primitives of *different* types)

5 Arrays (continued)

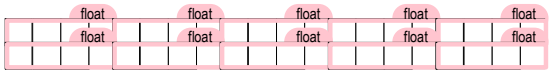
<data type> <array name>`[`*<size>*`];`

- This creates an array called *<array name>*, with *<size>* elements, each of which is a *<data type>*

```
float grades[10];
```

- Declares an array of 10 `float`s, called `grades`

- Arrays are managed automatically by the compiler, so this is like having 10 `float` variables:



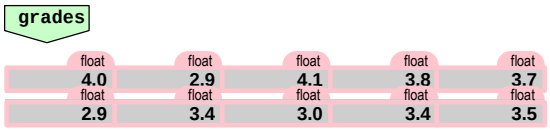
5 Using Arrays

- To access an element of the array, we specify its *index*
- If the size of the array is n:
  - the index of the first element is `0`
  - the index of the last element is `n-1`
- The array index is given in `[]` after the name of the array:
  - `grades[0]` is the first element of the array `grades`
  - `grades[9] *= 3;` multiplies the last element of `grades` by 3

5 Initializing Arrays

- When declaring an array, we can also initialize each of the elements by listing the values in `{}`:

```
float grades[10] = {4.0, 2.9, 4.1, 3.8,
  3.7, 2.9, 3.4, 3.0, 3.4, 3.5};
```



5 Program 5-3

```
/* Average 10 floats */
#include <iostream.h>

void main()
{
      float grades[10] = {4.0, 2.9, 4.1, 3.8, 3.7,
            2.9, 3.4, 3.0, 3.4, 3.5};
      int   index;
      float result = 0.0;

      // Total the 10 grades
      index = 0;
      while(index < 10) {
            result += grades[index];
            index++;
      }

      // Compute the average
      result /= 10.0;
      cout << "The average is " << result <<  endl;
}
```

Output

```
The average is 3.47
```


5 Notes on Program 5-3

- The `while` loop counts from 0 to 9 using the variable `index`
- Each successive element of the array `grades` is added to `result`
- To compute the average, `result` is divided by the number of elements in the array (10)

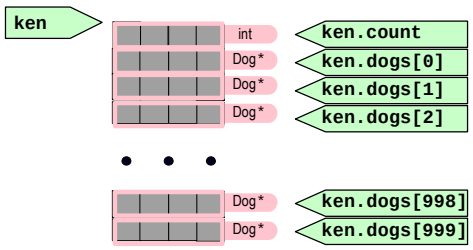
5 The Dog Kennel

- We can create arrays of any data type, including pointers
- Let's model a kennel full of dogs
  - We'll use an array of pointers to `Dog` to represent the dogs occupying the kennel
  - and an integer to store how many dogs are in the kennel
- Our kennel can hold a maximum of 1000 dogs

```
class Kennel {
public:
      int       count;
      Dog       *dogs[1000];
};
```

5 Declaring a Kennel

- `Kennel ken;` declares an `int` followed by an array of 1000 pointers to the `Dog` class



5 Kennel Constructor

- The constructor for the `Kennel` class takes an argument, the number of dogs to populate the kennel with

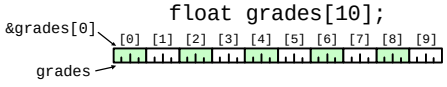
```
Kennel::Kennel( int howMany )
{
      int i;

      i = 0;
      while(i < howMany) {
            dogs[i] = new Dog;
            i++;
      }

      count = howMany;
}
```


5 Kennel Destructor

- The destructor for the `Kennel` class cleans up by deleting all the dogs

```
Kennel::~Kennel()
{
      int i;

      i = 0;
      while(i < count) {
            delete dogs[i];
            i++;
      }
}
```


5 Arrays and Pointers

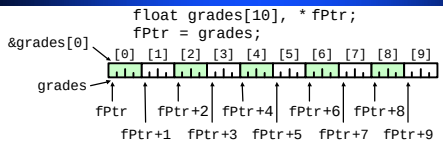


When we declare an array, we're asking the compiler to set aside a chunk of memory for us

- In this case, 10 `floats` (4 bytes each)

- `grades` can be thought of as a pointer (`float *grades;`) to the chunk of memory
  - With the exception that `grades` is a fixed location in memory, the array doesn't move once created
  - `grades` is the same as `&grades[0]`

5 Arrays and Pointers (continued)



- We can use pointers to index through arrays
- `*fPtr` and `grades[0]` are the same float
- *Pointer Arithmetic*: when an integer is added to a pointer, the integer is multiplied by the size of the thing pointed to (4 bytes, in this example)
  - `fPtr++` points to the next float (not the next byte)

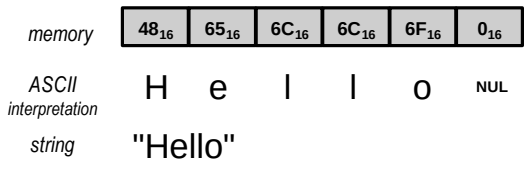
5 Walking the Dogs

- Here's a Kennel method for walking the dogs:

```
Kennel::WalkDogs()
{
      int i = 0;
      Dog *d = dogs;

      while(i < count) {
            if(d->GetStamina() > 10.0) {
                  d->Run();
            }
            else {
                  d->Walk();
            }

            d++;
            i++;
      }
}
```

5 Strings

- As we saw in Session 1, computers store text as strings of ASCII characters, terminated by a **NUL** (byte value of 0_{16})



5 String Constants

- In C++, a string is represented as an array of `chars`
- A constant string is typed using `""` double quotes
  - We've seen this in most of our `cout` statements
- The compiler manages storing the string in an array
- The compiler also adds the **NUL** at the end

5 String Variables

```
char str[80];
```

- To declare a variable string, we declare an array of `chars`

```
char hi[6] = "Hello";
```

- C++ will let us initialize a `char` array with a string
  - We have to make sure to leave room for the **NUL** at the end
- We could also list each of the characters in `{}`:

```
char hi[6] ={'H', 'e', 'l', 'l', 'o', 0};
```


5 String Variables (continued)

- Since the name of an array is the same as the address of the beginning of the array
  - `hi` is equivalent to `&hi[0]`
- We can use the name of the array like a string constant

```
cout << "This is the contents of the " <<
  "string hi: " << hi;
```

- Output:
  - `This is the contents of the string hi: Hello`

5 Lines of Text

- When we're manipulating text, quite often we deal with it a line at a time
- It's handy to pick a common size for most of your strings
- We'll use 128
  - Room enough for most lines of text we might type
  - Programmers like powers of 2

```
char a[128], b[128];
```

- Declares two strings, a and b, large enough for a line of text

5 Pointers and Strings

- We can also refer to strings using pointers

```
char a[128] = "Hello", *p;
```

- Declares a string a, initialized to "Hello", and a pointer to a `char`

```
p = a;  (same as p = &a[0];)
```

- Set p to the address where the array a begins
- Either a or p can be used to refer to the string

```
cout << p << a << p;
```

- Output: `HelloHelloHello`

5 Copying Strings

- Let's write a routine to copy one string to another
- When we pass an array as a argument to a subroutine, we use the name of the array
- This is how we'll use our string copy routine:

```
char destination[128];
char source[128] =
  "Twas brillig, and the slithy toves";
stringCopy(destination, source);
cout << destination;
```

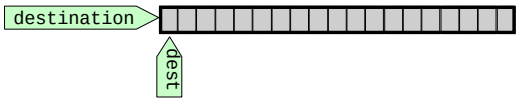
- Output: `Twas brillig, and the slithy toves`

5 stringCopy()

- We declare the arguments to our subroutine using pointers:

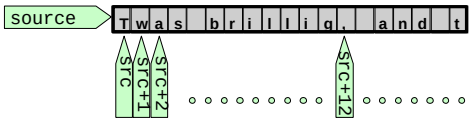
```
void stringCopy(char *dest, char *src);
```

- In this case, the pointer `dest` will get a copy of the address where the array `destination` begins,
  - while `src` is initialized with a copy of `source`



5 stringCopy() (continued)

- Since `dest` and `src` are copies of the original pointers, `destination` and `source`, we can change them without affecting the originals
  - `destination` and `source` will always point to the beginnings of the two 128-byte arrays
- We can increment the pointers to step to the next character



5 stringCopy() (continued)

- We can use `dest` and `src` to step through each of their respective strings
- To move a pointer to the next character, we increment it by one: `dest++;`
- If `dest` and `src` are each pointing to a location in their respective strings, we can copy the `chars` using: `*dest = *src;`
  - `*src` is the value of the `char src` is currently pointing to
  - Assigning that value to `*dest` means "store this value at the character `dest` is currently pointing to"
- We continue until `src` points to the **NUL** at the end

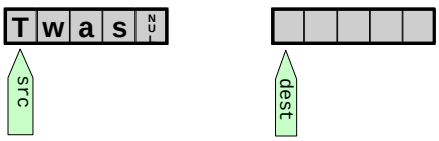
5 stringCopy() (continued)

- Here's the complete `stringCopy()` routine

```
void stringCopy(char *dest, char *src)
{
     while(*src != 0) {
          *dest = *src;
          dest++;
          src++;
     }

     *dest = 0;
}
```

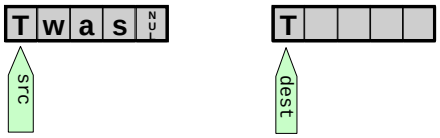

5 stringCopy() Animation #1

```
while(*src != 0) {
     *dest = *src;
     dest++;
     src++;
}

*dest = 0;
```



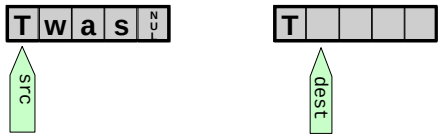
5 stringCopy() Animation #2

```
while(*src != 0) {
     *dest = *src;
     dest++;
     src++;
}

*dest = 0;
```



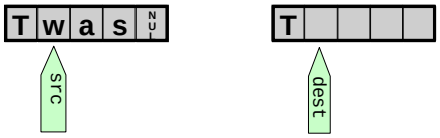
5 stringCopy() Animation #3

```
while(*src != 0) {
     *dest = *src;
     dest++;
     src++;
}

*dest = 0;
```



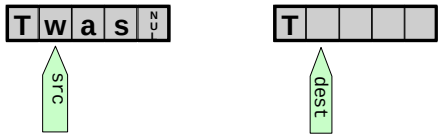
5 stringCopy() Animation #4

```
while(*src != 0) {
     *dest = *src;
     dest++;
     src++;
}

*dest = 0;
```



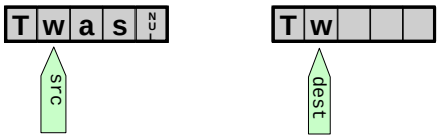
5 stringCopy() Animation #5

```
while(*src != 0) {
     *dest = *src;
     dest++;
     src++;
}

*dest = 0;
```



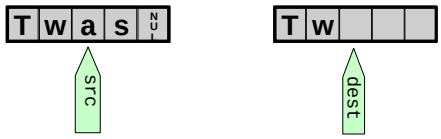
5 stringCopy() Animation #6

```
while(*src != 0) {
     *dest = *src;
     dest++;
     src++;
}

*dest = 0;
```



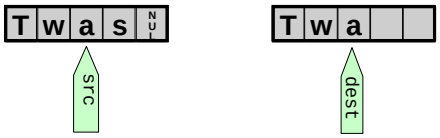
5 stringCopy() Animation #7

```
while(*src != 0) {
     *dest = *src;
     dest++;
     src++;
}

*dest = 0;
```



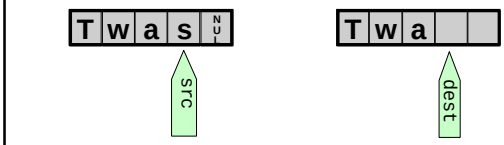
5 stringCopy() Animation #8

```
while(*src != 0) {
     *dest = *src;
     dest++;
     src++;
}

*dest = 0;
```



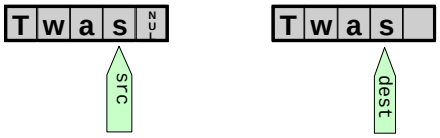
5 stringCopy() Animation #9

```
while(*src != 0) {
     *dest = *src;
     dest++;
     src++;
}

*dest = 0;
```



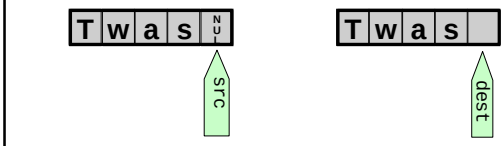
5 stringCopy() Animation #10

```
while(*src != 0) {
     *dest = *src;
     dest++;
     src++;
}

*dest = 0;
```



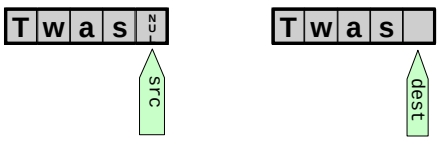
5 stringCopy() Animation #11

```
while(*src != 0) {
     *dest = *src;
     dest++;
     src++;
}

*dest = 0;
```



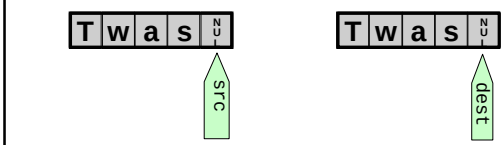
5 stringCopy() Animation #12

```
while(*src != 0) {
     *dest = *src;
     dest++;
     src++;
}

*dest = 0;
```



5 stringCopy() Animation #13

```
while(*src != 0) {
     *dest = *src;
     dest++;
     src++;
}

*dest = 0;
```



5 Built-in String Routines

- C++ has a collection of built-in routines for manipulating strings
- To use these routines, we need to add `#include <string.h>` to our programs
- In the examples that follow, we'll assume that we have two strings already declared:

```
char str1[128] = "Hello";
char str2[128] = "World";
```


5 String Routines

`int strlen(char *s);`

- Returns the length of the string `s`, not counting the **NUL** at the end
- `strlen(str1)` would return 5

`char *strcpy(char *dest, char *src);`

- Copies `src` to `dest` (just like our `stringCopy()`)
- Also returns a pointer to the beginning of the `dest` string
  - We rarely use the return value
- After `strcpy(str2, str1)`, `str2` would contain "Hello"

5 String Routines (continued)

`char *strncpy(char *dest, char *src, int n);`

- Just like `strcpy()`, but copies no more than n characters
- After `strncpy(str2, str1, 3)`, `str2` would contain "Hel"
- Like all the string routines, it makes sure the destination string is terminated with a **NUL**

`char *strcat(char *s1, char *s2);`

- Concatenates `s2` onto the end of `s1`
- After `strcat(str1, str2)`, `str1` would contain "HelloWorld"
  - `str2` would be unchanged

5 String Routines (continued)

`char *strncat(char *s1, char *s2, int n);`

- Like `strcat()`, but only appends n characters
- After `strncat(str1, str2, 2)`, `str1` would contain "HelloWo"

`int strcmp(char *s1, char *s2);`

- Compares `s1` and `s2` to see if they're equal
- Returns 0 if `s1` and `s2` are identical: `strcmp(str1, "Hello");`
- Returns -1 if `s1` would precede `s2` in alphabetical order: `strcmp("apple", "orange");`
- Returns 1 if `s1` would follow `s2`: `strcmp("flagpole", "flag");`

5 String Routines (continued)

`char *strstr(char *s1, char *s2);`

- Searches `s1` for the first occurrence of `s2`, starting from the left
- If it finds the string `s2` inside `s1`, it returns a pointer to the location in `s1`
- `strstr(str1, "ll")` would return a pointer two characters past the beginning of `str1` (where the "ll" appears in `str1`)
- If `s2` is not contained anywhere in `s1`, it returns 0

- There are many more string routines, but these are the most common

5 So Far

- Today we've seen how to manipulate memory using pointers
- We've also seen how to manage a list of some data type using arrays, and how to manipulate text using strings
- Using these tools, we can now do just about anything we might want to do with memory
  - Since memory is our interface to the entire computer, we can now control the entire computer
- In the next session we'll study the tools C++ provides for interacting with the user and with files

6

Introduction to Programming

Input/Output & Events

Session 6

Phil Mercurio
UCSD Extension
mercurio@acm.org

6 Recap

- We've seen how to deal with memory by structuring it into objects
- And how to deal directly with memory via pointers
- Today we'll study the tools available for interacting with the user

6 Operating Systems

- Your computer's operating system provides an environment for running programs
- The operating system provides a set of data types and routines that you can use
  - Some are organized into objects (like `cin` and `cout`)
  - Some aren't (like the string routines)
- These routines are packaged into *libraries*

6 Libraries

- A library is a file containing a collection of compiled routines, written by someone else
- Associated with a library are one or more header files, files ending in **.h**, which contain the declarations of the data types and routines used in the library
  - Most C++ libraries contain classes and their methods, but there are non-object-oriented libraries that we'll use too
- To use a library, you
  - `#include` the header files you need, and
  - Tell the compiler that you want it to use the library

6 Libraries (continued)

- The `cin` and `cout` objects we've been using belong to a library the compiler automatically includes, called the *standard C++ library*
- **iostream.h** is the header file for one set of classes in the standard library
- There are several other classes in the standard library, along with many routines not organized into classes
- There are also many other libraries available to you, some are included in your OS, while others you obtain from third parties

6 Operating Systems (continued)

- The operating system acts as an intermediary between our program and the user
  - When we think of interacting with the user, we're really interacting with the OS
- OSes provide two types of interface to the user:
  - Command-line interfaces (MS-DOS, Unix)
  - Window interfaces (MS-Windows, MacOS, Unix)
- In addition, the OS is responsible for managing files stored on hard disks or networks
  - We can think of reading and writing files as an indirect way of communicating with the user

6 What is an Interface?

- An *interface* is the boundary between two systems
- In this session, we're concerned with the interface between a program and a user
- The interface consists of
  - One or more means of providing input to the program
  - One or more means of getting output from the program
- As far as the user is concerned, the interface *is* the program

7 Session 6

6 Command-Line Interfaces

- Before computers were powerful enough to support graphical user interfaces, people interacted with computers by typing commands
- The computer displays a text prompt, like `C:\>` or `unix 1%`
- The user runs a program by typing a single line of text: a command, possibly with arguments, followed by a RETURN:
  - `type readme.txt`
- The program's output is also text, and the user may provide further input via the keyboard

8 Session 6

6 Command-Line Interfaces (continued)

- Today, most OSes (except MacOS prior to OS X) still have a command-line interface
  - Windows: DOS
  - Unix, Linux, and Mac OS X: various shells
- Command-line arguments are often either
  - names of files or
  - program options:
- DOS: `dir /p`
  - Lists files, pausing after each screenful
- Unix: `ls -t`
  - Lists files, sorted chronologically

9 Session 6

6 Streams

- As we've already seen, C++ deals with text input and output via streams
- A stream of text can be thought of like a stream of water: a bunch of drops of water (characters) all moving in the same direction
- An input stream is a *source* of text, while an output stream is a *sink*
- Streams can be used not only with the keyboard and screen, but also with files and other programs

10 Session 6

6 C++ CLI Environment

- A C++ program running in a CLI environment uses three I/O streams:
- *Standard Input:* input text, usually coming from the keyboard
  - We read the standard input via `cin`
- *Standard Output:* output text, usually going to the screen
  - We write to the standard output via `cout`
- *Standard Error Output:* a second output stream for error messages, always goes to the screen
  - We write to the standard error output via `cerr`

11 Session 6

6 Redirecting I/O

- In DOS and Unix, you can choose to take the standard input from a file or send it to a file:
  - `more < readme.txt`
    - Reads from the file **readme.txt** rather than the keyboard
  - `dir > list.txt`
    - Writes to **list.txt** rather than the screen
- You can also send the output of one program to the input of another:
  - `dir | more`
    - "Pipes" the output of the dir program through more, which displays it one screenful at a time

12 Session 6

6 Redirecting I/O (continued)

- Output sent to the standard error output is always displayed on the screen, even when the standard output is being redirected to a file or another program
- `cerr` is a safe way to print messages that the user has to see, such as
  - The user specified the wrong arguments
  - A needed file couldn't be found

6 Output Streams

- Output streams are described by the class `ostream`
  - `cout` and `cerr` are two `ostream` objects created for your program by the OS
- The output streams are accessed with the << operator
  - This is just a special way of invoking a method from the `ostream` class on an `ostream`
- An `ostream` can output all of the primitive types
  - `chars` and strings are output in ASCII
  - `ints` and `floats` are output as decimal values, in ASCII text
  - `bools` are output as 0 for false, 1 for true

6 Output Streams (continued)

- When outputting to a stream, the textual representation of each expression follows the previous one immediately:
- `cout << 5 << "Hello";` would output `5Hello`
- To add spaces, we have to be explicit:
  - `cout << 5 << ' ' << "Hello";` would output `5 Hello`
- To start a new line, we have to use `endl`:
  - `cout << 5 << endl << "Hello";` would output

```
5
Hello
```


6 Input Streams

- Input streams are described by the class `istream`
  - `cin` is an `istream` object created by the OS
- Input streams are read from using the >> operator
  - >> is also just a method of the `istream` class
- An `istream` knows how to read all of the primitive types
  - `chars` and strings are read as ASCII text
  - `ints` and `floats` are read as ASCII text and interpreted using base 10
  - `bools` are read like `ints`, 0 is false, anything else is true

6 Input Streams (continued)

- When an input stream reads the value of a variable, it consumes text from the input until satisfied
- All spaces, and tabs, and newlines in the input are ignored

```
int a,b;
cin >> a >> b;
```

- The input will be scanned to find a value for a
  - Spaces, tabs or newlines are skipped
  - Characters are collected up to the first non-digit
  - That string of characters is interpreted as a decimal integer

6 Input Streams (continued)

- After reading a, `cin` then reads b in the same way
  - Any intervening spaces, tabs, or newlines are discarded
- These are all valid ways to type the input for `cin >> a >> b;`

```
100 200
```

```
100        200
```

```
100
        200
```

6 Input Streams (continued)

- The input to your program is *buffered*: the OS waits until you've typed a RETURN before sending the text to your program
  - The place where the text you're typing is stored, prior to sending it to `cin`, is called a *buffer*
  - This is why you're able to backspace and correct your input line before hitting the RETURN--the program hasn't seen the input yet
- When you put an instruction like `cin >> a;` in your program, it won't proceed until you've typed an integer and a RETURN

6 Reading Text

- When reading a single `char`, an `istream` behaves the same as when reading `int`s or `float`s
  - Spaces, tabs, and newlines separating one `char` from another are ignored
- When reading a string, an `istream`
  - Skips spaces, tabs, and newlines
  - Consumes all of the characters up to the next space, tab or newline
- In other words, an `istream` fills a string by reading a single word of text

6 Program 6-1

```
#include <iostream.h>

void main()
{
      int i;
      float f;
      char c1, c2;
      char str1[128], str2[128], str3[128];

      cout << "Please type an int and a float: ";
      cin >> i >> f;
      cout << endl << "You typed: " << i
            << " and " << f << endl;

      cout << "Please type two characters: ";
      cin >> c1 >> c2;
      cout << endl << "You typed: " << c1
            << " and " << c2 << endl;
```


6 Program 6-1 (continued)

```
      cout << "Please type three words: ";
      cin >> str1 >> str2 >> str3;
      cout << "You typed: " << str1
            << "|" << str2 << "|" << str3
            << endl;
}
```

Output

```
Please type an int and a float: 3   4.78

You typed: 3 and 4.78
Please type two characters: af

You typed: a and f
Please type three words: alpha beta
gamma
You typed: alpha|beta|gamma
```


6 Program 6-1 (alternate input)

Output, alternate input

```
Please type an int and a float: 2.1

You typed: 2 and 0.1
Please type two characters: afx

You typed: a and f
Please type three words: alpha beta
You typed: x|alpha|beta
```


6 Reading Lines

- To read a whole line of text, `istream` provides a method called `getline()`:

```
istream::getline(char *ptr, int size);
```

- `ptr` is a string we're going to use as an input buffer, to store one line at a time
- `size` is the number of chars in our buffer

```
char buffer[128];
cin.getline(buffer,128);
```

6 End of File

- How do we know when we're done reading the input?
  - If we're reading from a file, we'll eventually get to the end of the file
  - If we're reading from the keyboard, the user can type control-Z (DOS) or control-D (Unix) to end the input
- `istreams` have a special property: when tested in an `if` or `while` statement they evaluate to a Boolean value:
  - True if there is data left to be read
  - False if we've gotten to the end of the input

6 Program 6-2

- This program reads each line of text you type and repeats it twice

```
#include <iostream.h>

void main()
{
      char  buffer[128];

      do {
            cin.getline(buffer,128);
            if(cin) {
                  cout << buffer << endl;
                  cout << buffer << endl;
            }
      } while(cin);

      cout << "OK, bye!" << endl;
}
```


6 Program 6-2 Output

```
Hello
Hello
Hello
Goodbye
Goodbye
Goodbye
^Z
OK, bye!
```


6 Reading & Writing Objects

- The iostreams know how to input and output the C++ primitives, but how do we do I/O on our classes?
- Recall overloaded methods from Session 4: multiple methods with the same name but different data types as arguments
- >> and << are methods of the `istream` and `ostream` classes
- We need to define additional >> and << methods that take our class as an argument
- We're actually extending what the `istream` and `ostream` classes can do

6 Extending istream

- Let's create a class for handling dates:

```
class Date {
public:
   int month, day, year;
};
```

- To read a `Date` from an `istream`, we need to write a >> method that works on a `Date`
- Unfortunately, the grammar for doing this is a little tricky, it uses a couple of C++ features that are beyond the scope of this course

6 Extending `istream` (continued)

- To create a >> method for your own class, follow this example:
  - This is your class
  - Insert here the command(s) to read each of the parts of the object `obj` from the istream `in`
  - This reads dates like 8/1/1942

```
istream& operator>>( istream& in, Date& obj )
{
      char slash;
      in >> obj.month >> slash >> obj.day
           >> slash >> obj.year;
      return(in);
}
```

6 Extending ostream

- To create a << method for your own class, follow this example:
  - Insert here the command(s) to write each of the parts of the object obj to the ostream out, in the same order as the input method

```
ostream& operator<<( ostream& out, Date& obj)
{
     out << obj.month << "/"
         << obj.day << "/"
         << obj.year;
     return(out);
}
```


6 Reading & Writing Objects (continued)

- Creating input and output methods for a class requires some strange syntax, but the result is very simple

```
void main()
{
     Date today;

     cout << "Please enter a date: ";
     cin >> today;
     cout << "You entered: " << today <<  endl;
}
```

Output

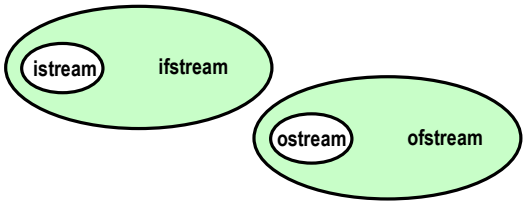
```
Please enter a date:  5/16/1959
You entered: 5/16/1959
```


6 File I/O

- Using the standard input and output, our program can only be reading from one file at a time, and writing to one other file
  - The OS opens the files for us when we use < and > to redirect I/O
- To open files ourselves, we use the classes ifstream (input file stream) and ofstream (output file stream)
- We'll need to #include <fstream.h>

6 File I/O (continued)

- ifstream inherits from istream, and ofstream inherits from ostream
- Using file streams is exactly like using the standard streams



6 Opening a File

- The ifstream and ofstream classes each take an argument in their constructors: the name of the file to open

```
ifstream in("readme.txt");
```

  - Creates an ifstream by opening the file **readme.txt** for reading

```
ofstream out("output.txt");
```

  - Creates an ofstream by opening the file **output.txt** for writing
  - If the file doesn't exist, it's created
  - If it does exist, it's erased and writing starts at the beginning

6 Closing a File

- As long as the ifstream or ofstream exists, the file is open
- When the stream variable is destroyed (when the routine exits), the file is closed
- This routine counts the characters in a file

```
int countFile(char *filename)
{
     ifstream in(filename);
     int  count = 0;
     char c;

     do {
          in >> c;
          if(in) { count++; }
     } while(in);

     return(count);
}
```

6 Command-Line Arguments

- When the user types a command line to invoke your program, the OS splits the line up into words (separated by spaces or tabs) and provides it to your program
- The command-line arguments are passed as arguments to `main()`
- The arguments are a list of strings, so they're passed as an array of strings
- Another argument is provided to count the number of strings

6 Program 6-3

- This program prints out each of its arguments

```
#include <iostream.h>

void main(int argc, char *argv[])
{
      int   i = 0;

      cout << "There are " << argc
            << " arguments: " << endl;

      while(i < argc) {
            cout << "Argument #" << i
                  << ": " << argv[i] << endl;

            i++;
      }
}
```


6 Program 6-3 (Output)

- `argc` is the argument count, and `argv` is the argument values (array of strings)
- If we run the program with the arguments **file1.txt** and **file2.txt** we get the output below
- Note that `argv[0]` is the name of the program itself
- The argument count includes the program name

```
c:\work> prog6-3 file1.txt file2.txt
There are 3 arguments:
Argument #0: prog6-3
Argument #1: file1.txt
Argument #2: file2.txt
c:\work>
```


6 File Merger

- Let's write a program to merge two files, line by line
- The output will consist of a line from file1, then one from file2, then the next from file1, and so on, until both of the files run out
- Our program will take three arguments, the first and second input files and the output file

6 Program 6-4

- We'll have to check the argument count to make sure the user specifies enough file names

```
#include <fstream.h>

void main(int argc, char *argv[])
{
      if(argc != 4) {
            cerr << "Usage: " << argv[0] <<
                  " file1 file2 output" << endl;

            return;
      }
```


6 Program 6-4 (continued)

```
      ifstream a(argv[1]), b(argv[2]);
      ofstream out(argv[3]);
      char  buffer[128];

      while(a || b) {
            if(a && a.getline(buffer,128)) {
                  out << buffer << endl;
            }

            if(b && b.getline(buffer,128)) {
                  out << buffer << endl;
            }
      }
}
```

6 Program 6-4 Output

```
c:\work> prog6-4
Usage: prog6-4 file1 file2 output
c:\work> prog6-4 a.txt b.txt out.txt
c:\work>
```

a.txt:
a1
a2
a3
a4
a5

b.txt:
b1
b2
b3

out.txt:
a1
b1
a2
b2
a3
b3
a4
a5

6 From Commands to Windows

- We've just covered most of what we'll need to deal with programs running from a command line
- Command-line interfaces will be around forever, but most new software is being developed in graphical environments
- Graphical user interfaces (GUIs) include a display (usually color) and a pointing device (usually a mouse)
- Programs are represented by icons and create one or more windows when executed

6 From Commands to Windows (continued)

- Most C++ compilers in GUI environments provide a means of emulating a CLI by creating a window for your program
  - Text typed in the window is the standard input
  - The standard output and standard error output are displayed in the window
  - In Visual C++, this is called a Console (DOS) application
  - In Metrowerks Code Warrior (Mac), this is called a SIOUX application
- Under Windows, you can edit the properties of your program's icon to specify the equivalent of command-line arguments

6 Getting GUI

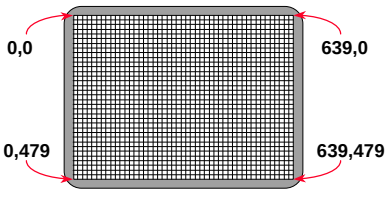
- GUI programs are usually fired up by an unmodified icon (they have no arguments) and do not have a standard input or output
- The program creates windows containing menus, buttons, and other controls for getting input from the user
- Every GUI environment (MacOS, Windows, X Windows) is different, and there are many ways to program GUI applications under each OS
- We can't cover the specifics of any particular GUI environment, but we can discuss many of the common, basic principles

6 Bit-Mapped Displays

- All modern graphics computers have a bit-mapped display
- The screen, like that of a television, is composed of thousands of individual colored dots, called *pixels*
- Pixels are arranged in a rectangular grid
  - A typical desktop computer screen is 640 (horizontal) by 480 (vertical) pixels or more
  - A typical graphics workstation would be 1280 x 1024

6 Bit-Mapped Displays (continued)

- The screen can be thought of as a piece of graph paper, with the squares colored in
  - We can refer to each pixel by giving its coordinates in the horizontal (X) and vertical (Y) directions



6 Pixels

- The pixel values for your entire display are stored together in an area of memory called *video RAM*
  - Video RAM may be physically separate from your main memory, but it's still in the same address space
- The computer hardware takes care of reading this memory and converting it to the voltages which control your monitor, at least 30 times a second
- The number of different colors you can display at once depends on your graphics hardware and the amount of memory allocated for each pixel

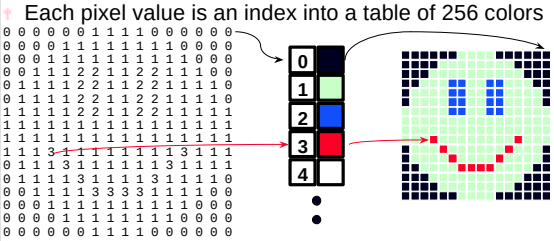
6 Pixels (continued)

- In a monochrome display, each pixel consists of only 1 bit, black or white
- High-end graphics workstations have *true color* displays, with 3 bytes per pixel
- Each pixel consists of 3 unsigned 8-bit integers, storing the amount of red, green, and blue in the color
  - Each pixel can be any one of 2^{24} (~16.7 million) colors

	R	G	B
white	255	255	255
black	0	0	0
gray	128	128	128
red	255	0	0
magenta	255	0	255
pink	255	192	203
dark green	0	103	94
turquoise	54	180	190
pale turquoise	200	240	240

6 Color Tables

- Most computers have a *pseudocolor* display, with only 8 bits (1 byte) per pixel
- Each pixel value is an index into a table of 256 colors

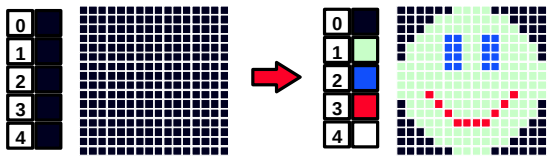


6 Color Tables (continued)

- Graphics on a pseudocolor display is like coloring with a box of 256 crayons
  - You can have any 256 colors you want, but no blending!
- 256 colors are fine for most non-graphics applications, but dealing with realistic images requires a true color display
- Some pseudocolor displays have 12 or 16 bits per pixel, for 4096 or 65,536 colors (respectively)
- Many computers can operate at different screen and pixel sizes, trading the number of colors for the number of pixels, or vice versa

6 Color Table Animation

- Pseudocolor displays can do color table animation, changing the image just by changing color entries in the table



6 Graphics Libraries

- Graphics libraries consist of a set of routines and data structures for dealing with the display, color table, etc.
- There are many different libraries for doing graphics, some general purpose, some specialized
- There may be more than one graphics library in use on the same machine, simultaneously
- We'll consider a simplified graphics library, organized as a set of C++ classes

6 Display

- The basis of our graphics library will be the `Display` class
  - In this case, we'll assume a 640x480 display, 8-bit pseudocolor
- `Display` contains the memory for the pixel values and for the color table
- C++ arrays can be extended to two dimensions by adding another set of brackets:
- `unsigned char screen[640][480];` declares a 2-D array consisting of 307,200 pixels

6 Display (continued)

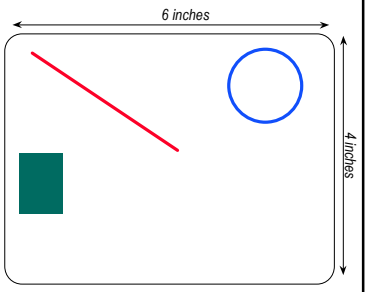
- `screen[0][0]` is the upper left corner, `screen[639][479]` is the lower right corner
- There are two fundamental ways of creating images on a `Display`: drawing and painting

6 Drawing

- The `Display` class contains a group of methods for drawing lines and shapes (rectangles, circles, ellipses, etc.)
- You can redefine the coordinates of the screen to be whatever you want, so you can deal in inches, centimeters, etc.
- For example, the `Circle` method would take the coordinates and radius of a circle, along with the user-defined coordinates for the screen, and compute which pixels should be changed to represent that circle

6 Drawing (continued)

- `Line(0.5,0.5, 3.0,2.0,red);`
- `Rectangle(0.0, 2.0,1.0,3.0, green, filled);`
- `Circle(4.5, 1.0,0.75,blue, unfilled);`



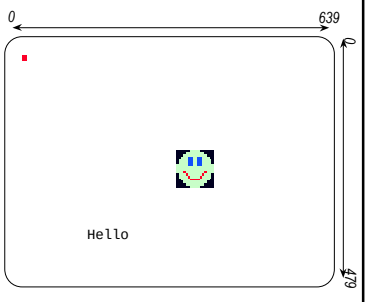
6 Painting

- Painting is setting the values of pixels directly, and is usually done in real screen coordinates
  - The simplest painting method sets the value of one pixel, given the coordinates
- Often painting consists of copying a rectangular area from off-screen memory onto the screen
- A chunk of memory containing an image is called a *bitmap*
  - The happy face from a few slides earlier might be `unsigned char happy[16][16];`
- Text fonts are also stored as bitmaps, and are painted onto the display

6 Painting (continued)

- `SetPixel(20, 20,red);`
- `PaintText( "Hello", courier, 18, 100,400);`
- `PaintBitmap (happy,16,16, 320,200);`

happy



6 Graphical Input

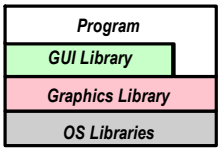
- To get input from the user, `Display` also has a pair of methods for getting the current position of the mouse
  - `GetMouseX()` and `GetMouseY()`
- If we've remapped the coordinates, the `Display` will do the inverse mapping and report the mouse position in, for example, inches
- `Display` also provides methods for checking the state of the mouse button(s)

6 Primitive Graphics

- The drawing and painting methods, along with the methods for reading input devices, form the primitives for a graphics library
- Attempting to build a modern graphical user interface, with dozens of window, popups, buttons, menus, drawing windows, typing windows, scrollbars, etc. , would be very difficult using these tools
- Fortunately, there are many libraries, usually called user interface toolkits, designed to help create GUIs

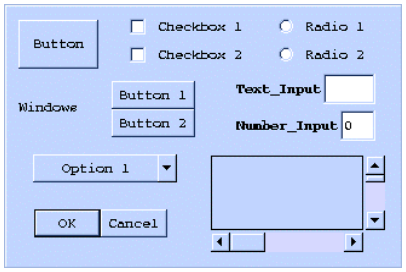
6 UI Toolkits

- A GUI toolkit consists of routines and data types (often organized into objects) for creating window-based interfaces
- The toolkit uses the primitive graphics library
- We can think of our program consisting of layers of libraries:



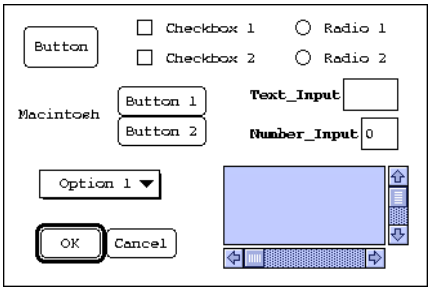
6 UI Toolkits (Windows)

- The toolkit lets us "draw" with more sophisticated objects (often called *widgets*)



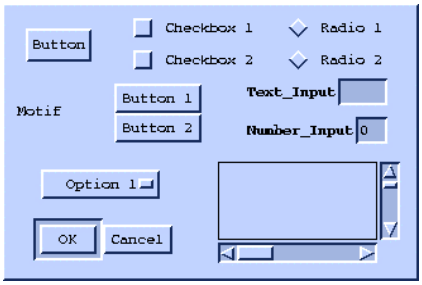
6 UI Toolkits (Macintosh)

- Here are the same widgets in a Macintosh application:



6 UI Toolkits (Motif)

- Here are the same widgets in a Motif application:



6 GUI Builders

- Writing the C++ code to create a complicated user interface can be very tedious
  - Having to edit your program and recompile just to move a button over by a few pixels slows down development
- Most toolkits come with interactive programs for creating interfaces
  - Similar to a drawing program, a GUI builder lets you select widgets from a palette and place them in a representation of your window
- The GUI builder then creates files containing the C++ instructions to create the interface, it actually writes C++ for you!

67 Session 6

6 Many Toolkits

- There are many GUI toolkits and builders, especially for the Unix and Windows operating systems
- User interface development is a big field of research, and there are many approaches to designing a toolkit
  - C++ is a popular language for GUIs because object-orientation is a good way to design a window interface
- You may be running applications developed with different toolkits at the same time
  - The different programs might even have similar-looking widgets

68 Session 6

6 Look & Feel vs. API

- A toolkit actually consists of two "interfaces":
- Look & Feel (L&F)
  - What the widgets look like to the user, and how they operate
- Application Programmer Interface (API)
  - What the library looks like to the programmer: the classes or other data types and routines in the library
- Users are concerned with having a consistent, familiar L&F
- Programmers are concerned with having an API that's easy to learn and flexible

69 Session 6

6 Look & Feel vs. API (continued)

- In the MS Windows OS, the Microsoft Foundation Classes are the API for applications with a Windows L&F
- Under Unix, the standard for L&F is called Motif
  - The Motif designers also created an API, but their libraries are a commercial product
  - There are many freely available Unix toolkits which create Motif applications, but have entirely different APIs

70 Session 6

6 Separating Look & Feel from API

- There are also toolkits which include different versions for different OSes
  - The API for all versions is the same, but
  - The look & feel is different on each OS, to match the standard for that OS
  - One version of your program can be compiled and run on several different OSes
- There are even toolkits that masquerade as other toolkits!
  - For example, there are Unix toolkits which have the same API as the Microsoft Foundation Classes
  - A MFC program could run under Unix with a Motif L&F

71 Session 6

6 GUI Example

- Let's consider a simplified GUI C++ API
- There would be classes corresponding to each of the widgets:
  - `Window`, `Button`, `Menu`, `Label`, `Scrollbar`, etc.
- We would start by creating a window, specifying its size

```
Window myWin(300,200);
```

- When creating a widget for inside a window, one of the constructor arguments is the window

```
Button offButton(&myWin,"Off");
```

72 Session 6

6 GUI Example (continued)

- All of the widgets have methods to cause the widget to change its appearance or behavior
- `offButton.SetColor(red);`
  - Changes the `offButton`'s color to red
- `scroller1.SetRange(0,200);`
  - Changes the range of values for a `Scrollbar` called `scroller1` to 0 to 200
  - We might get the value later using `scroller1.GetValue();` and it would be in the range 0 to 200
- Most of these instructions to create and configure widgets would be written using a GUI builder

6 1-D vs. 2-D Interfaces

- With a text-only interface, the flow through your program is pretty much linear
  - The program starts with the command line arguments as input
  - Perhaps it prompts the user for additional input, or reads some files
  - The result is computed and output is generated, to the screen or files
- A program with a linear interface is pretty much in control of the dialog with the user
  - The program presents output, and the user responds

6 1-D vs. 2-D Interfaces (continued)

- A GUI is opposite, the user is in control
- The program displays windows filled with widgets, and is required to respond correctly to whatever widget the user chooses to operate
- Instead of reading input and writing output, a GUI program
  - Processes input events (typing, clicking an on-screen button with the mouse, moving a scrollbar with the mouse, etc.) and
  - Generates displays (or other output)
- Programming a GUI is much different from programming a text-based program

6 A GUI `main()`

- The main routine for a GUI program usually consists of two parts:
- Object Creation
  - Many, many lines of C++ to create and configure all of the windows and all of the widgets they contain
- Event Handling
  - A loop that runs continuously, collecting input events from the user and calling the methods you've written to generate displays
  - The event handler is usually a routine provided by the toolkit, when it's finished your program is over

6 Event Handling

- The event handling routine figures out, for example, that the user has clicked the left mouse button while the cursor was positioned over a checkbox
- It can also do more complicated tasks, like managing all of the behavior of a scrollbar
  - Clicking on the slider and dragging it
  - Clicking on the arrows
  - The result is a value in the range you specified when creating the scrollbar
- But you need to somehow specify what you want to happen when, for example, a particular button is pressed

6 Inheriting Buttons

- The `Button` class, as contained in the toolkit, has a method called `Action()`
  - When the event handler notices that the user has clicked on a `Button`, it calls its `Action()` method
- When we create a button for our application, we make it a descendant of `Button` and write a new `Action` for it:

```
class QuitButton : public Button {
      ....
      void Action() {
            saveFiles();
            exit();
      }
};
```

6 Inheriting Buttons (continued)

`QuitButton qb(&myWin,"Quit");`

- Creates a `QuitButton` as part of the `Window myWin`

- When the event handler gets the mouse click on your `QuitButton`, it calls its `Action()` method instead of the default `Button::Action()`
  - `exit()` is a built-in C++ routine that quits your program immediately
  - `saveFiles()` is a routine you wrote to write everything to the output files before exiting
- More complicated widgets, like a text window, might have many different methods for different actions (typing, selecting text, backspacing, etc)

6 Event-Driven Programming

- When writing an event-driven GUI program, you can't just list all the steps in order as in a linear, text-only program
- Instead, each widget is programmed as a unit, as if it had a life of its own
  - Each widget has its own actions
  - Usually, an action is a pretty short routine
- Underneath everything are the classes you write to implement the non-GUI portion of your program

6 Event-Driven Programming (continued)

- For example, a checkbook-balancing program would have classes to represent a check, to total the checks in an account, to sort the checks, to print them, etc.
- The action methods contain statements operating on the non-GUI classes
- For example, a menu item called **`sort`** might call the method `account.SortChecks();`
- Each action method has to be written as a self-contained unit, keeping in mind that you don't know when the user will cause the action to happen

6 So Far

- In this session we saw the two primary means of interacting with the user:
- Command-Line Interfaces and Files
- Graphical User Interfaces
- Now that we can talk with our programs, next session we'll look at how a program performs sophisticated computations

7

Introduction to Programming

Algorithms & Data Structures

Session 7

Phil Mercurio
UCSD Extension
mercurio@acm.org

7 Recap

- In the course so far, we've developed a model of a computer sufficient for our needs as programmers
- We've also learned a good portion of the C++ language, and
- How to interact with the user
- But we still don't know how to solve real problems

7 Pizza Making

- Let's imagine that we've assembled all of the ingredients for a pizza: flour, water, yeast, sugar, oil, mozzarella, tomato sauce, pepperoni, veggies, spices, etc.
- We might also have all the tools and utensils we need to make the pizza: a pan, bowls, knives, an oven, etc.
- But unless we know *how* to make a pizza, we can't proceed
- A good cookbook would describe the general procedure for making many different types of pizzas, rather than one pizza recipe

7 More Pizza

- To make a pizza:
  - Combine the flour, water, yeast, sugar, and oil to make dough
  - Let the dough rise, stretch it flat in the pizza pan, repeat until the whole pan is covered
  - Combine the tomato sauce and spices and simmer to make pizza sauce
  - Layer the sauce, cheese, and other ingredients on the dough and bake
- We haven't described how to make a particular pizza, rather just the overall design of pizza-making
  - We're also assuming knowledge of bread-making and sauce-making

7 Pizza Analogy

- The data structures (groups and lists of data types) we define are the ingredients
- The C++ commands are the utensils and tools we use
- The description of pizza-making is the *algorithm*
  - It tells us how to make a pizza, but it's not as specific as a particular recipe (program)
  - In fact, we can use the pizza algorithm and our imagination to create many different pizza recipes

7 Algorithms

- An algorithm describes how to perform a particular programming task
- But it's not a detailed program in a specific language
- Algorithms are a big field of computer science research
- There are many books and articles written on computer algorithms
  - A good book covering many common algorithms is *Algorithms in C++* by Robert Sedgewick
- Sharing algorithms is how computer programmers build on each other's work

7 Algorithms (continued)

- Algorithms are often expressed in *pseudocode*
  - Pseudocode is a made-up language that's basically English
  - Pseudocode isn't precise enough to be a real computer language
  - Each author makes up their own pseudocode
- By creating a pseudocode for describing algorithms, the author makes it possible for many programmers to use their algorithms, regardless of what language they're using
- We'll describe some algorithms, but we'll also express them precisely in C++

7 Data Structures and Algorithms

- A data structure is a general term for the grouping and listing of data types we've already seen
- The design of most algorithms include the design of data structures as well as the steps of the algorithm
- Object-oriented programming is a good approach for expressing algorithms, because a C++ class combines
  - A data structure (the attributes of the class) and
  - Algorithms expressed as class methods

7 Back to the Kennel

- We'll be looking at several algorithms which operate on lists of objects
- Recall our Virtual Dog Kennel example from Session 5, which consisted of an array of pointers to Dogs, and an `int` containing the number of Dogs in the array

```
class Kennel {
public:
   Dog *dogs[1000];
   int count;
   ....
};
```


7 Dogs

- Our Dogs will have many attributes, we'll be looking at
  - The dog's name, stored as a character string
  - The dog's age in (human) years, stored as a float

```
class Dog {
public:
   char name[128];
   float age;
   ....
};
```


7 An Assumption

- We'll assume that our Kennel has already been filled with a collection of Dogs
  - Each `Dog` will already have a name and age
  - This data may have been read from a file, entered by the user, etc.

Creating a Kennel:

```
Kennel ken;
```

Calling a method:

```
ken.oldestDog()
```

In a Kennel method:

```
count
dogs[0]->name
dogs[9]->age
```


7 The `for` statement

- Let's take this opportunity to introduce a new C++ statement, the `for` loop

`for(`*<initialization>*`; `*<condition>*`; `*<increment>*`) {`*<body>*`}`

  - The statements in *<body>* are repeated until *<condition>* is no longer true
  - Before starting the loop, the *<initialization>* statement is executed
  - Each time through the loop, after the *<body>*, the *<increment>* statement is executed
- *<initialization>*, *<condition>* and *<increment>* usually all operate on the same variable, called the *index variable*

7 The for statement (continued)

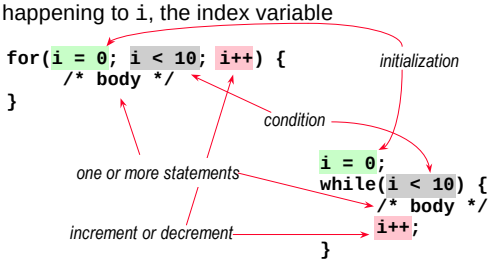
- A `for` loop can be looked at as a shortcut for a common `while` loop

```
for(i = 0; i < 10; i++) {
     /* body */
}
```

```
i = 0;
while(i < 10) {
     /* body */
     i++;
}
```


7 The for statement (continued)

- Note that the `for` version makes it clearer what's happening to `i`, the index variable



7 Scanning and Searching

- Two simple and common operations on lists are
  - Searching for an item
  - Scanning through the list performing some operation on each of the items
- Scanning and searching are nicely expressed using `for` loops

7 Oldest Dog Pseudocode

- We want to write a method that finds the age of the oldest dog in the kennel
- We need to scan the entire list of dogs, examining each dog's age
- First, create a variable `oldest` to store the age of the oldest dog so far
  - We initialize `oldest` to an absurd value: 0.0
- For each dog in the kennel, if it is older than `oldest`, store its age in `oldest`
- When we're done, `oldest` contains the age of the oldest dog in the kennel

7 Oldest Dog

- This method returns the age of the oldest dog in the Kennel
  - In `if`, `while`, and `for` statements, if the *<body>* is only one statement, you don't need the {}

```
float Kennel::oldestDog(void)
{
     int i;
     float oldest = 0.0;

     for(i=0; i < count; i++) {
          if(dogs[i]->age > oldest)
               oldest = dogs[i]->age;
     }

     return(oldest);
}
```


7 Average Dog

- An average of a list of values is defined as the sum of values divided by the count of how many values are in the list
- This method returns the average of the dogs' ages

```
float Kennel::averageAge(void)
{
     int i;
     float sum = 0.0;

     for(i=0; i < count; i++)
          sum += dogs[i]->age;

     return(sum/count);
}
```

7 Where's My Dog?

- This method searches the Kennel for a dog with a particular name, and returns a pointer to it
  - Recall that `strcmp()` returns 0 when the two strings are equal

```
Dog *Kennel::findDog(char *who)
{
     int i;

     for(i=0; i < count; i++) {
          if(strcmp(dogs[i]->name,who) == 0)
               return(dogs[i]);
     }

     /* If we get here, we failed */
     return(0);
}
```


7 Sorting

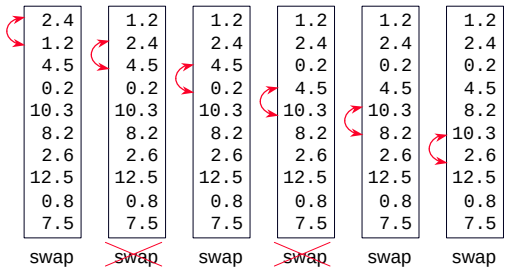
- When searching for something in a list, we can be more efficient if the list is sorted in order first
  - Consider trying to find a word in a dictionary that's not in any particular order
  - You'd have to look at each and every item!
- Sorting is one of the most heavily researched areas of computer algorithms
- Let's look at a simple but effective sorting algorithm called Bubble Sort
  - To make our example a little simpler, we'll sort by age

7 Bubble Sort

- Here's a pseudocode description of the Bubble Sort algorithm:
- Start at the beginning of the array and compare the first pair of items
  - If the items are in the wrong order, swap them
  - Proceed with the next pair, swapping each pair that's out of order
  - At the end of the array, go back to the beginning for another pass
- Keep going through the array until you make a full pass without any swaps
- You're done!

7 Bubble Sort (continued)

- Here's a diagram of the first pass through an array



7 Bubble Sort (continued)

- Here's the result after each pass

1.2	1.2	0.2	0.2	0.2	0.2	0.2	0.2
2.4	0.2	1.2	1.2	1.2	1.2	0.8	0.8
0.2	2.4	2.4	2.4	2.4	0.8	1.2	1.2
4.5	4.5	2.6	2.6	0.8	2.4	2.4	2.4
8.2	2.6	4.5	0.8	2.6	2.6	2.6	2.6
2.6	8.2	0.8	4.5	4.5	4.5	4.5	4.5
10.3	0.8	7.5	7.5	7.5	7.5	7.5	7.5
0.8	7.5	8.2	8.2	8.2	8.2	8.2	8.2
7.5	10.3	10.3	10.3	10.3	10.3	10.3	10.3
12.5	12.5	12.5	12.5	12.5	12.5	12.5	12.5
6 swaps	4 swaps	4 swaps	1 swap	1 swap	1 swap	1 swap	0 swaps

7 Kennel::bubbleSort()

- This method performs a Bubble Sort of a Kennel
- We simplify this method by assuming we'll write another method, `bSortPass()` to do one pass
- This matches the pseudocode "make passes until a pass is made with no swaps"

```
void Kennel::bubbleSort(void)
{
     int swapped;

     do {
          swapped = bSortPass();
     } while(swapped > 0);
}
```

7 Kennel::bSortPass()

- This method makes one pass through the array

```
int Kennel::bSortPass(void)
{
      int swaps = 0, i;
      Dog *temp;

      for(i=0; i < count-1; i++) {
            if(dogs[i]->age > dogs[i+1]->age) {
                  temp = dogs[i];
                  dogs[i] = dogs[i+1];
                  dogs[i+1] = temp;

                  swaps++;
            }
      }

      return(swaps);
}
```


7 Quick Sort

- Bubble Sort is easy to understand and implement, but it's not that fast
- One of the most frequently used sorting algorithms is Quick Sort
- The standard C++ library provides a routine called `qsort()` which can sort any array, using the Quick Sort algorithm

7 Quick Sort (continued)

- Calling `qsort()` requires C++ features beyond the scope of this course, but the arguments are
  - the array
  - the number of elements in the array
  - the size (in bytes) of each element
  - A routine which compares two elements, and returns the same values `strcmp()` returns:
    - -1 if the two elements are in order
    - 0 if they are equal
    - 1 if they are out of order

7 Smart Searching

- The minimal searching algorithm described earlier, starting at the beginning and looking at every item in the list, is the worst approach to searching
- For a small list, it's not a problem
- For a large list, like the entries of a dictionary, it can take a significant amount of time to perform a search
- Search algorithms are as well researched as sorting algorithms
- We can use the knowledge that the array has been sorted to perform a more efficient search

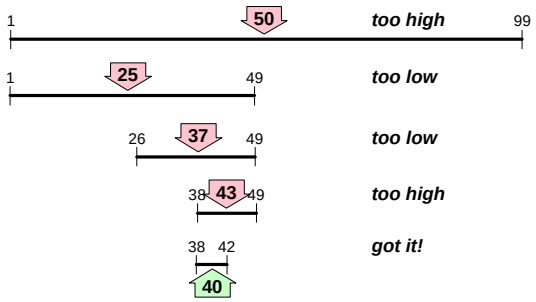
7 Binary Searching

- As an example, let's say you're trying to guess a number between 1 and 99
- Your goal is to find the number in the fewest number of guesses
- Each time you guess, I'll tell you if you're correct, or if you're too high or too low
- Your best strategy is to guess 50, the number in the middle
  - My answer will eliminate half of the array

7 Binary Searching (continued)

- Now pick the number in the middle of the remaining half, etc.
- This is called binary searching, because each step cuts the array in half

Binary Search Example: 40



Binary Search Algorithm

- Searching for `x` in array `M`, size `n`:
- *Step 1:* Set `low` to `0`, `high` to `n - 1`
- *Step 2:* If `low > high`, `x` is not in `M` at all; we're done
- *Step 3:* Set `mid` to `(low + high) / 2`
- *Step 4:* If `M[mid] == x`, return `mid`, we're done
- *Step 5:* If `M[mid] < x`, set `low` to `mid + 1` and go to Step 2
- *Step 6:* If `M[mid] > x`, set `high` to `mid - 1` and go to Step 2

`Kennel::bubbleSortByName()`

- Before we search for a `Dog` by name, we'll need to sort the `Kennel` by name
- We'll have a method called `Kennel::bubbleSortByName()` which calls `bSortByNamePass()`
- `Kennel::bSortByNamePass()` is identical to to `bSortPass()`, except the `if` statement is

```
if(strcmp(dogs[i]->name,
          dogs[i+1]->name) > 0) ...
```


Where's My Dog? #2

```
Dog *Kennel::findDog(char *who)
{
     int low = 0, high = count - 1;
     int mid, result;

     while(low <= high) {
          mid = (low + high) / 2;
          result = strcmp(dogs[mid]->name,who);
          if(result == 0)      // got it!
               return(dogs[mid]);
          else if(result < 0) // too low
               low = mid + 1;
          else                 // too high
               high = mid - 1;
     }

     return(0);  // dog gone
}
```


Stacks and Queues

- Stacks and queues are specialized lists
- In fact, we restrict the ways we can add and remove items from the list
  - In contrast to arrays, where we can access any element directly
- A stack has two operations
  - **Push**: add an item to the end of the list
  - **Pop**: remove the last item from the list
- A stack is referred to as LIFO: Last In, First Out
  - We saw a stack used for switching subroutine context in Session 3

Stacks and Queues (continued)

- A queue is like waiting in line at the grocery store
- There are two operations:
  - **Append**: add an item to the end of the queue
  - **Remove**: remove the first item from the queue
- A queue is FIFO, First In First Out
- Both stacks and queues can be implemented using arrays

7 Stacks vs. Queues

- Stacks are handy for saving data we need to get back to later
- As an example, consider a World-Wide Web hypertext browser like Netscape
  - Or the hypertext help in MS Windows
- You're looking at a home page on the WWW
- When you select a link, the page you're looking at is pushed on the stack, and the link is followed to a new page
- When you select the **Back** button, the browser pops the stack

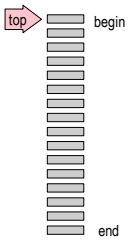
7 Stacks vs. Queues (continued)

- Notice that you can follow many links in succession, but still back all the way out
- A queue is handy for processing data in order
- For example, a bank's central computer would use a queue to store automated teller transactions as they're received over the network
- Each transaction is processed in turn, in the order they came in

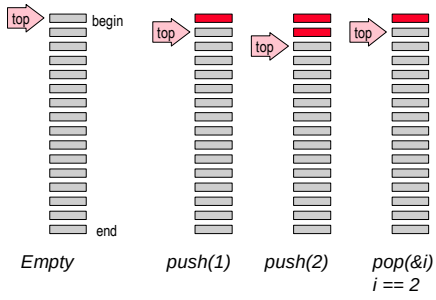
7 IntStack

- We can implement a stack using an array
  - The stack won't be able to grow indefinitely, it's limited by the size of the array

```
class IntStack {
public:
     int stack[1000];
     int *begin, *end, *top;

     IntStack() {   // Constructor
          begin = stack;
          end = &stack[999];
          top = begin;
     }
     ...
};
```

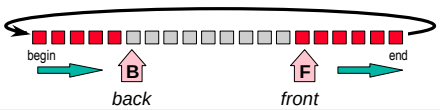


7 IntStack Diagram



7 IntQueue

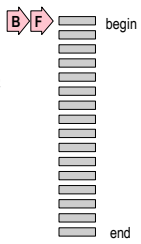
- We can also implement a queue using an array, but it's a little trickier
  - When we remove the first element of the queue, we don't want to have to copy each of the elements over by one
- We have to keep track of both the front and back
  - `begin` and end are still the boundaries of the array, we have to wrap around when one of our pointers hits a boundary



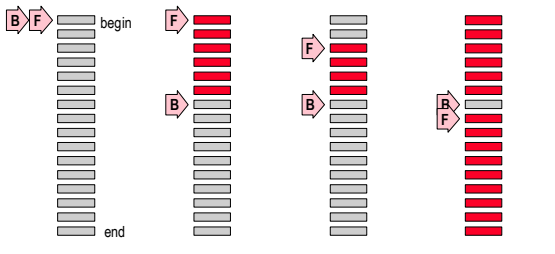
7 IntQueue (continued)

- Remember that `begin` and end never move, they mark the array
- `back` behaves just like a stack's `top`

```
class IntQueue {
public:
     int queue[10];
     int *begin, *end, *front, *back;
     char full;  // is queue full?

     IntQueue() {   // Constructor
          begin = queue;
          end = &queue[9];
          front = begin;
          back = begin;
          full = 0;
     } ...
```



7 IntQueue Diagram



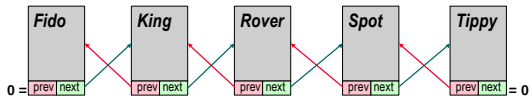
7 The Problem with Arrays

- Using an array to represent a list is limited, because an array has a fixed size
  - What if we don't know how large our list is going to be?
  - What if we're going to have many lists, some large and some small, and we don't want to use a large array for the small lists?
  - How can we insert an item in the middle of a list without all the copying involved in moving the existing elements over by one?
- To solve this problem, we use a *linked list*
  - A linked list can grow to any length because each item keeps track of the next, like a string of children holding hands

7 Linked List

- To make a linked list out of Dogs, we add two more attributes to the class, pointers to the previous Dog and the next Dog in the list

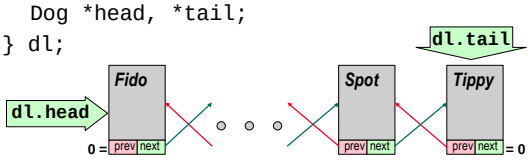
```
class Dog {
   ... lots of attributes ...
   Dog *prev, *next;
};
```



7 Linked List (continued)

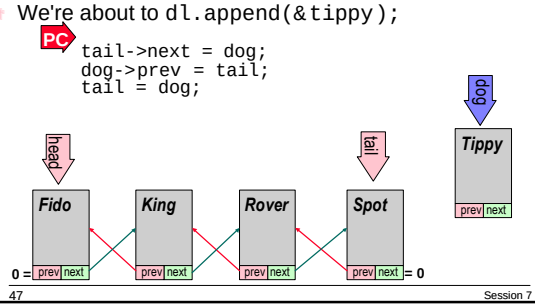
- In order to keep track of a linked list, we need to store a pointer to the first item in the list (the *head*)
- It's also handy to keep track of the last item in the list (the *tail*)

```
class DogList {
   Dog *head, *tail;
} dl;
```



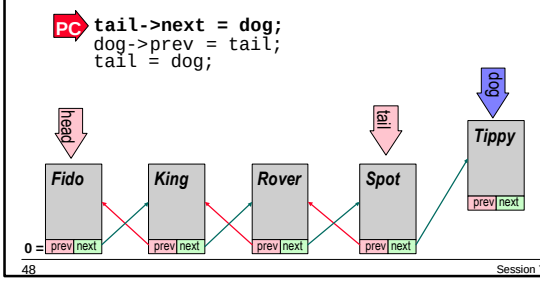
7 DogList::append Before

- Here we have four dogs in a linked list `DogList dl;`
- We're about to `dl.append(&tippy);`



7 DogList::append During 1

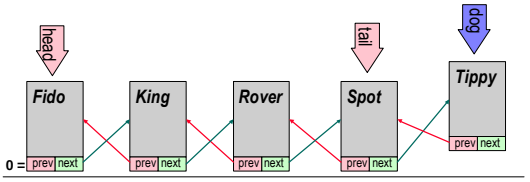
- The new Dog gets appended after the current `tail`



7 DogList::append During 2

- The `prev` pointer needs to be set

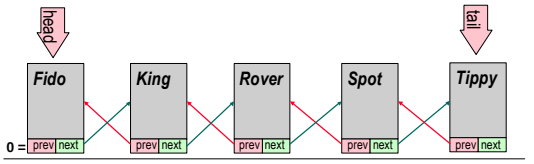
```
tail->next = dog;
dog->prev = tail;
tail = dog;
```



7 DogList::append After

- Then we update the `tail` attribute of the `DogList`

```
tail->next = dog;
dog->prev = tail;
tail = dog;
```



7 DogList::append

- To add a Dog to the list, we attach it to the tail
  - `head` and `tail` are initialized to 0 in the `DogList` constructor

```
void DogList::append(Dog *dog)
{
      if(tail == 0) {    // empty list
            head = dog;
            tail = dog;
      }
      else {
            tail->next = dog;  // link up
            dog->prev = tail;
            tail = dog;        // new tail
      }

      tail->next = 0;          // terminate
}
```


7 DogList::oldestDog

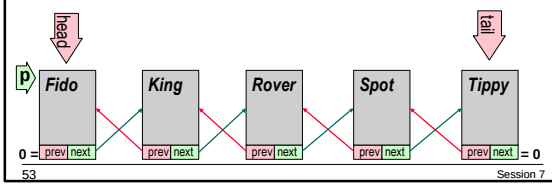
- Since we know our DogList is terminated by a null pointer, here's how we can express the oldest dog scan

```
float DogList::oldestDog(void)
{
      Dog *p;
      float oldest = 0.0;

      for(p=head; p != 0; p = p->next) {
            if(p->age > oldest)
                  oldest = p->age;
      }

      return(oldest);
}
```


7 Traversing a DogList #1

- Initialization

```
for(p=head; p != 0; p = p->next) {}
```

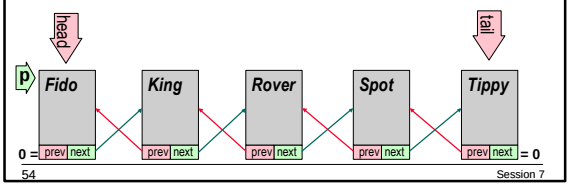


7 Traversing a DogList #2

- Condition

```
for(p=head; p != 0; p = p->next) {}
```

True

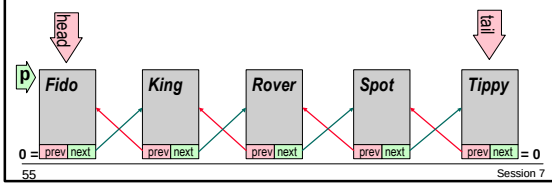


7 Traversing a DogList #3

- Body

```
for(p=head; p != 0; p = p->next) {}
```

PC

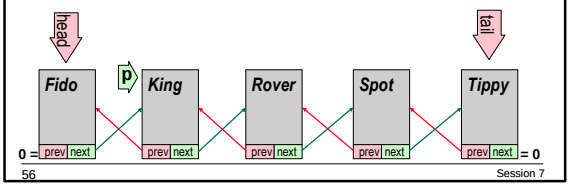


7 Traversing a DogList #4

- Increment

```
for(p=head; p != 0; p = p->next) {}
```

PC

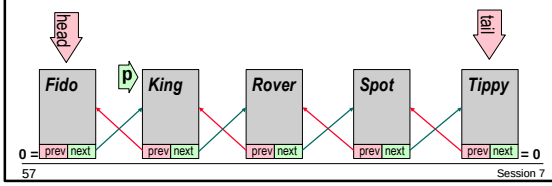


7 Traversing a DogList #5

- Condition, Body

```
for(p=head; p != 0; p = p->next) {}
```

True

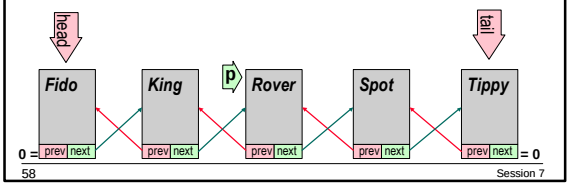


7 Traversing a DogList #6

- Increment, Condition, Body

```
for(p=head; p != 0; p = p->next) {}
```

True

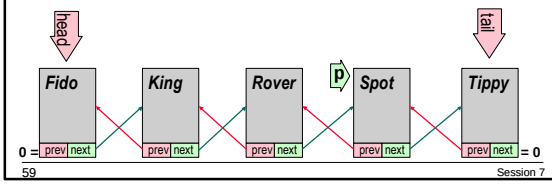


7 Traversing a DogList #7

- Increment, Condition, Body

```
for(p=head; p != 0; p = p->next) {}
```

True

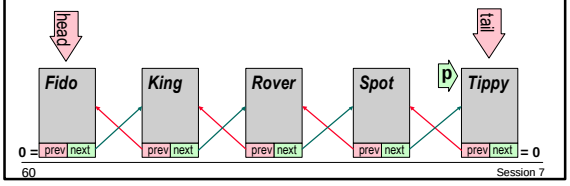


7 Traversing a DogList #8

- Increment, Condition, Body

```
for(p=head; p != 0; p = p->next) {}
```

True



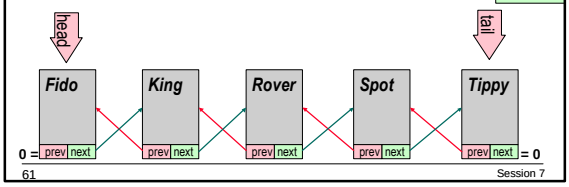
7 Traversing a DogList #9

- Final Increment

```
for(p=head; p != 0; p = p->next) {}
```

False

p = 0



7 DogList::insert

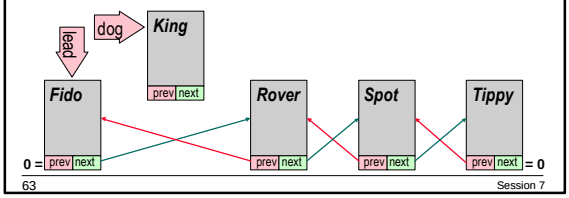
- We can insert a dog anywhere in the list, if we provide a pointer to the dog we want it to follow

```
void DogList::insert(Dog *lead, Dog *dog)
{
     if(lead == 0) { // insert at head
          dog->next = head;
          head->prev = dog;
          head = dog;
     }
     else {               // insert in middle
          dog->next = lead->next;
          lead->next = dog;
          dog->prev = lead;
          if(dog->next) dog->next-> prev = dog;
          if(tail == lead) tail = dog; //at end
     }
}
```


7 DogList::insert Animation #1

- Here we have four dogs in a linked list `DogList dl;`
- We're about to `dl.insert(&fido,&king);`

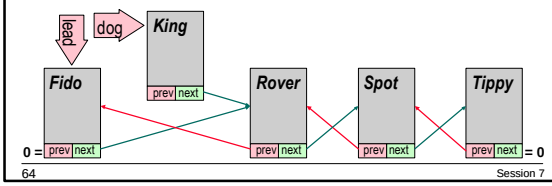
```
PC dog->next = lead->next;
   lead->next = dog;
   dog->prev = lead;
   if(dog->next) dog->next-> prev = dog;
```



7 DogList::insert Animation #2

- After the first statement, the dog is attached to its follower

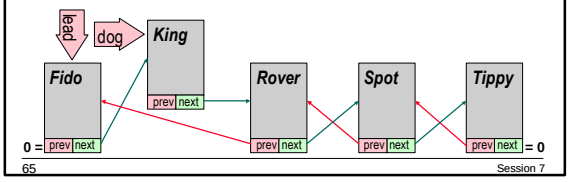
```
PC dog->next = lead->next;
   lead->next = dog;
   dog->prev = lead;
   if(dog->next) dog->next-> prev = dog;
```



7 DogList::insert Animation #3

- Now we can change the `next` pointer on the `lead` dog

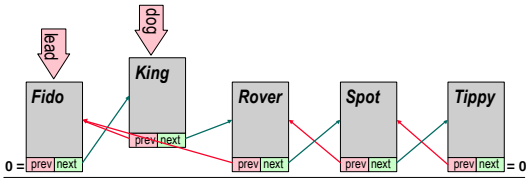
```
   dog->next = lead->next;
PC lead->next = dog;
   dog->prev = lead;
   if(dog->next) dog->next-> prev = dog;
```



7 DogList::insert Animation #4

- We have to make certain the `prev` links are correct also

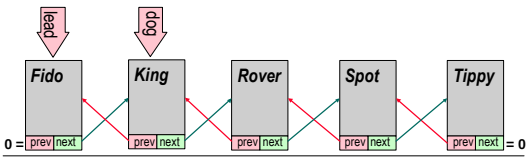
```
   dog->next = lead->next;
   lead->next = dog;
PC dog->prev = lead;
   if(dog->next) dog->next-> prev = dog;
```



7 DogList::insert Animation #5

- If there's a dog after dog, set its `prev` pointer to dog

```
dog->next = lead->next;
lead->next = dog;
dog->prev = lead;
if(dog->next) dog->next->prev = dog;
```



7 DogList::remove

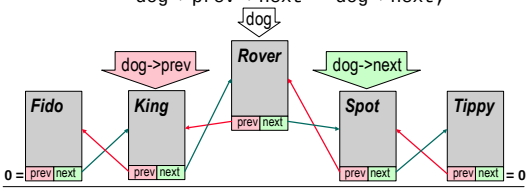
- We can remove a dog from anywhere in the list

```
void DogList::remove(Dog *dog)
{
     if(dog->next)
          dog->next->prev = dog->prev;

     if(dog->prev)
          dog->prev->next = dog->next;

     // Fix the head and tail
     if(dog == head)
          head = dog->next;

     if(dog == tail)
          tail = dog->prev;
}
```


7 DogList::remove Animation #1

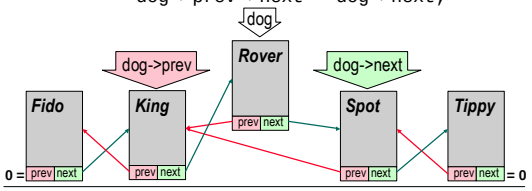
- Let's now remove Rover: `dl.remove(&rover);`

```
if(dog->next)
     dog->next->prev = dog->prev;

if(dog->prev)
     dog->prev->next = dog->next;
```



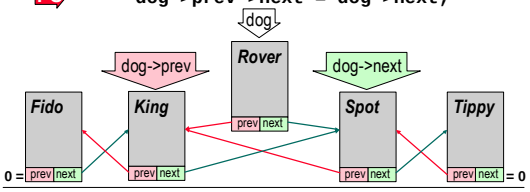
7 DogList::remove Animation #2

```
if(dog->next)
     dog->next->prev = dog->prev;

if(dog->prev)
     dog->prev->next = dog->next;
```



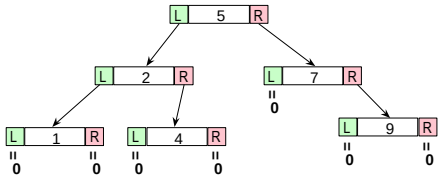
7 DogList::remove Animation #3

```
if(dog->next)
     dog->next->prev = dog->prev;

if(dog->prev)
     dog->prev->next = dog->next;
```



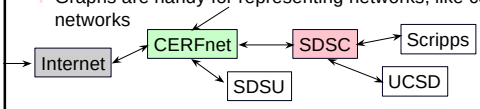
7 Trees

- We can use pointers to connect objects up in many different configurations
  - One powerful structure is the *tree*
  - This is a *binary tree*, each entry has a left and right attachment



7 Trees and Graphs

- There are many different tree structures, with differing numbers of attachments
  - Trees can be used to model hierarchies, like a file system or the organization chart of a corporation
- A tree is a specific type of *graph*, where an entry in a graph can have any number of attachments to any number of other entries
  - Graphs are handy for representing networks, like computer networks



73 Session 7

7 Recursion

re-cur-sion **ri-'ker-zhen**\ *n*

1. See *recursion*

- Recursion is one of the most elegant algorithms
- A C++ routine can call itself, which can call itself again, etc.
  - The only limit is the size of the context-switching stack
- Whenever we can break a problem down into a subset that's easier to solve, we can use recursion

74 Session 7

7 Factorials

- The *factorial* in mathematics is a classic example of a recursive problem
- The factorial of an integer n is defined as all the integers from 1 to n multiplied together
  - The factorial is written n!
- 5! = 5 x 4 x 3 x 2 x 1 = 120
- 6! = 6 x 5 x 4 x 3 x 2 x 1 = 6 x 5! = 720
- 0! is defined to be 1
  - It just is
- Factorials are used heavily in statistics and other areas of math

75 Session 7

7 Factorials (continued)

- We could compute a factorial using a `for` loop

```
int factorial(int n)
{
     int i;
     int result = 1;

     for(i=1; i <= n; i++)
          result *= i;

     return(result);
}
```

76 Session 7

7 Factorials (continued)

- The factorial of n can be defined in terms of the factorial of a smaller number, n-1
- n! = n x (n-1)!
- We can keep on breaking it down until we get to something we know, the factorial of 0:

n! = n x (n-1)! = n x (n-1) x (n-2)! = ...

= n x (n-1) x (n-2) ... x 2 x 1 x 0!

- This leads to a very elegant factorial routine

77 Session 7

7 Recursive Factorials

- If n is 0, we return 1
  - By definition
- Otherwise, we return n times the result of calling factorial on the smaller number `n-1`

```
int factorial(int n)
{
     if(n == 0)
          return(1);
     else
          return(n * factorial(n-1));
}
```

78 Session 7

7 Factorial Animation #1

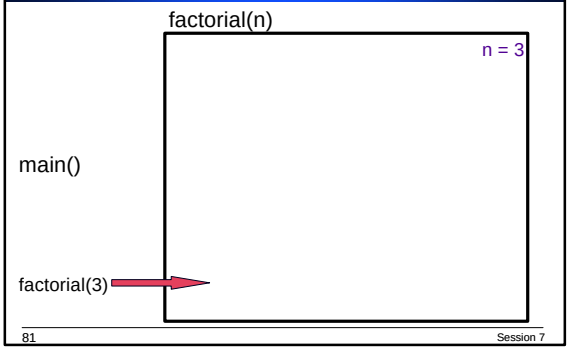
main()

7 Factorial Animation #2

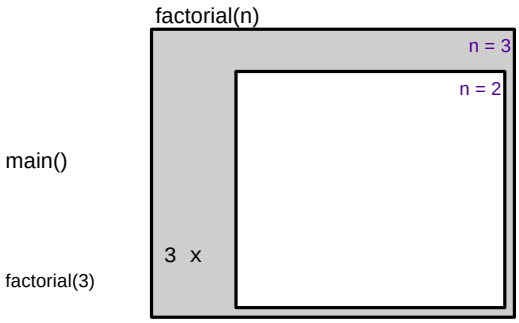
main()

factorial(3)

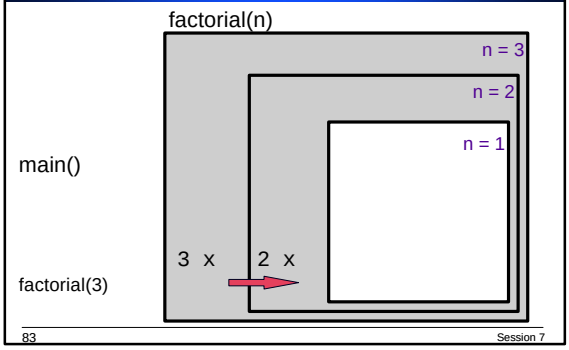
7 Factorial Animation #3



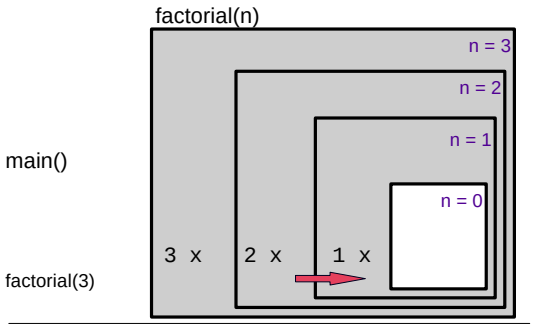
7 Factorial Animation #4

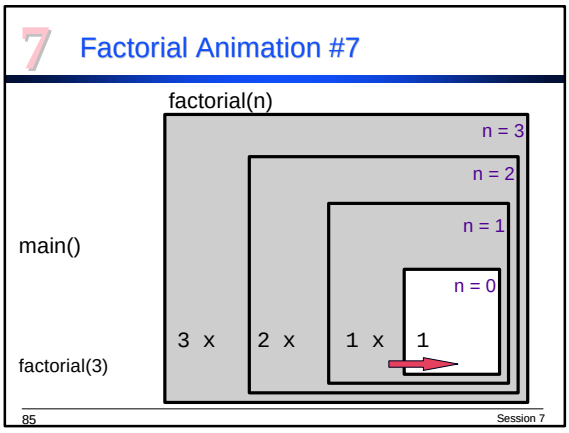


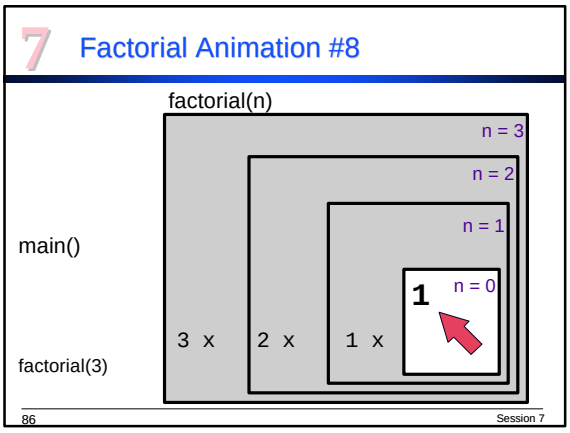
7 Factorial Animation #5

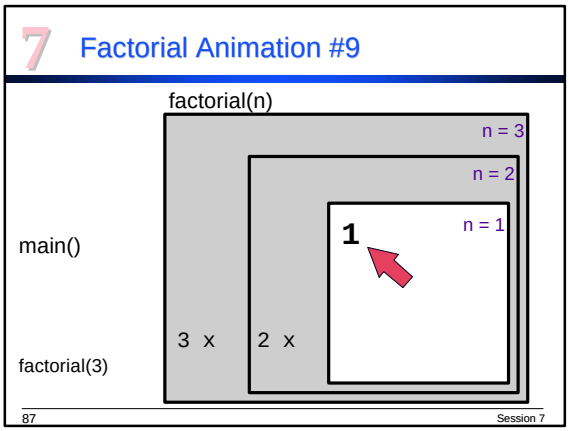


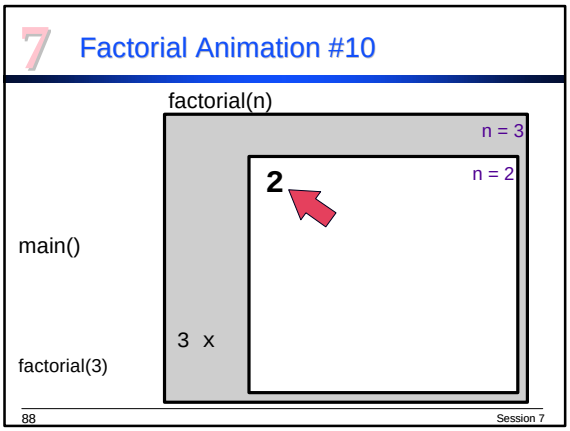
7 Factorial Animation #6

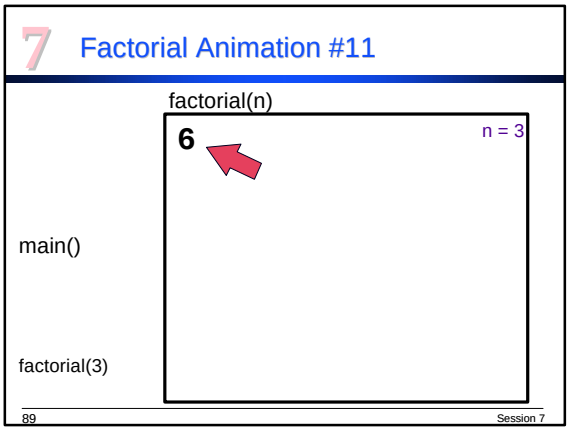


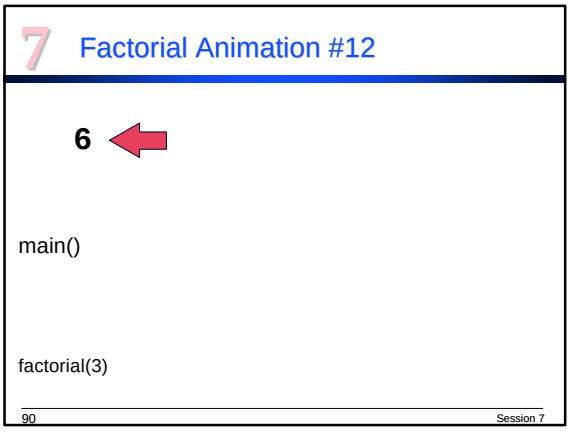






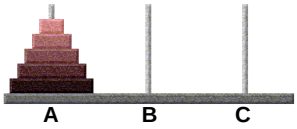




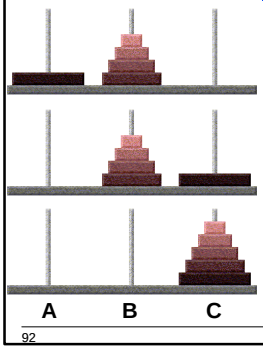


7 Towers of Hanoi

- Another problem best solved via recursion
- Move 5 disks from Tower A to Tower C
  - Only the top disk of a tower can be moved
  - You may never place a larger disk on top of a smaller one



7 Towers of Hanoi Solution



To move 5 disks from A to C:

Move 4 disks from A to B

Move disk #5 from A to C

Move 4 disks from B to C

7 Hanoi.cpp

```
#include <iostream.h>

void hanoi(int nDisks, char from, char to, char spare);

void main(int ac, char** av)
{
      hanoi(5,'A','C','B');
}

// Move one disk
void move(int disk, char from, char to)
{
      cout << "Move disk " << disk
           << " from peg " << from
           << " to peg " << to
           << endl;
}
```


7 Hanoi.cpp (continued)

```
// Move a stack n disks tall from Tower 'from' to
// Tower 'to', using 'spare' as a spare tower.
//
void hanoi(int n, char from, char to, char spare)
{
      if(n > 1) {
            hanoi(n-1,from,spare,to);
            move(n,from,to);
            hanoi(n-1,spare,to,from);
      }
      else {
            move(n,from,to);
      }
}
```


7 Algorithm Toolbelt

- Recursion is one of many powerful approaches toward solving a problem
- The algorithms described here and elsewhere are tools that can be used together to solve computational problems
  - For example, many of the routines used to manipulate a graph are easily expressed using recursion
- A working programmer is researching, modifying, and inventing algorithms daily
- In the final session, we'll discuss how to use algorithms written by others, via libraries, and some other tools and languages for programming

8

Introduction to Programming

Libraries & Languages

Session 8

Phil Mercurio
UCSD Extension
mercurio@acm.org

8 Recap

- The preceding 7 sessions provided an introduction to the theory and design of programming
- In this final session, we'll discuss the tools that a working programmer uses to get the job done
  - Other than pizza and caffeine
- We'll also take a short survey of other programming languages

8 Modularity

- The text of a C++ program is normally distributed among many files
- Usually, each class Foo is one *module*:
  - **Foo.h** header file containing the declaration of the class
  - **Foo.cpp** or **Foo.cc** code file containing the source code for the methods of the Foo class
- Some programmers like to combine several classes in one pair of **.h** and **.cpp** files, especially if the classes are closely related to each other
- How does the compiler combine the text from multiple files to create one executable program?

8 Symbols

- The vocabulary of your program, the names of the variables and routines you've written, are called *symbols*
- In the process of compiling, the compiler has to resolve symbols to the memory addresses they represent
  - variables are resolved to the address of the memory the compiler set aside to store that value
  - routine names are resolved to the address in the program where that routine begins
- If there are any symbols left unresolved, the compiler can't produce an executable program

8 Symbols (continued)

- The local variable symbols are all resolved before the compiler gets to the end of the routine
  - They're always declared within the same routine, before they're used
- But subroutine calls often refer to routines in other modules, or in libraries
- When the compiler reaches the end of a **.cpp** file, there are normally symbols left unresolved

8 Object Code

- The text of your program, in C++ or whatever language, is called the *source code*
- When the compiler operates on a source code file like **Foo.cpp**, it creates a file called **Foo.o**, containing the *object code*
- Object code is the machine language version of your program, but with the unresolved symbols still stored as text

Foo.cpp

```
#include <iostream.h>

void main()
{
  double f = 42.0;
  double x;

  x = sqrt(f);
}
```

Foo.o

```
7F45 4C46 0102 0100
0000 0000 0000 0000
0002 0002 0000 0001
0001 08E4 000 0034
0000 2930 0000 0000
0014 1D11 2F75 7372
2F6C 6992 2F54 642E
6F70 sqrt 7463
4E41 4D49 6D6F 6465
5F63 7869 7400 7F66
6D61 714D 714D 714D
```

Compiling and Linking

- The process of compilation actually consists of several steps, or *phases*
- The first phase is called the *preprocessor*, it operates on the text of your program before it's compiled
  - `#includes` are handled by the preprocessor
- In the second phase the actual compilation occurs, turning **.cpp** files into **.o** files (object code)
- The final phase is the linker, which takes many object code files and resolves the symbols

Compiling and Linking (continued)

- If a **.cpp** file calls a routine, like `sqrt()`, which is in another file, an unresolved reference is retained in the **.o** file
- During linking, all of the **.o** files are scanned for the definitions of unresolved symbols
- The final result is a program with all of the symbols turned into addresses, ready to be loaded into memory
- The compiler performs all the phases of the compilation process automatically

Separate Compilation

- When we modify parts of our program and recompile, only those **.cpp** files that changed need to be recompiled
- Most C++ development environments include a tool that checks the dates of the **.cpp** files against the **.o** files, and only recompiles those **.cpp** files that are newer
  - This tool is called **make** in the Unix OS, and is built into packages like Visual C++
- Working on large projects would be impractical otherwise!

Back to the Libraries

- We introduced the concept of a library in Session 6
- A library is a file containing multiple object files produced from some one else's source code
  - There are also header files containing the declarations of the classes, data structures, and routines in the library
- Libraries are
  - Provided with your C++ compiler
  - Provided with your OS
  - Purchased as third party products
  - Public domain or shareware

The Standard C Libraries

- C has been around, in various forms, since the early '70s
- Since 1983, the American National Standards Institute (ANSI) has been working on standardizing C
- This means both finalizing the definition of the language, as well as specifying the routines that must be available in the standard libraries
- Any C compiler purchased within the past couple of years will be an ANSI C compiler with ANSI standard libraries
  - All C++ compilers are also ANSI C compilers

The Standard C Libraries (continued)

- The standard C libraries include the tools to deal with strings (`strcpy()`, `strcmp()`, etc.) that we've already seen
- C also has standard file I/O routines (not as easy to use as C++'s iostreams)
- The standard libraries also include support for
  - handling dates and times
  - finding out the limits of floating point arithmetic on your machine
  - dealing with errors
  - managing memory
  - sorting and searching, etc.

The Math Library

- The standard C libraries are automatically linked with our programs by the compiler
- C also provides a library for doing mathematics including
  - square and cube roots
  - trig functions (sine, cosine, etc.)
  - log and exponentiation functions, etc.
- The headers files for the math library are included with `#include <math.h>`
- You usually have to explicitly instruct the compiler to link with the math library

The Standard C++ Libraries

- The C++ language has recently completed the ANSI standardization process
  - As a result, you might still have problems getting a program written for one compiler to work with another
- In the long process of developing a standard, pieces of it solidify and become readily available
- For example, the streams classes described in Session 6 can be expected to be the same in any modern C++ compiler

The Standard C++ Libraries (continued)

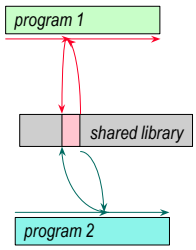
- ANSI C++ also includes standard classes for dealing with strings
- The ideal of C++ is that, by organizing library routines into well-designed classes, they become easier to use

Operating System Libraries

- The C++ language can only go so far in creating a standard environment for your programs
- There are many features of your computer and your operating system that are handled by libraries provided with your OS
- Some libraries are used quite heavily, appearing in almost every program
- Having the same library file included in numerous program files is a waste of disk space

Shared Libraries

- Libraries only contain routines (or methods)
- Any data structures or objects associated with a library are created as part of your program
  - Either as variables or via dynamic memory allocation (`new` and `delete`)
- It's possible to share one copy of a library, loaded into memory, among many running programs



Shared Libraries (continued)

- In Unix, they're called *shared libraries*
- In Windows, they're called *Dynamic Link Libraries* (DLLs)
- When compiling a program to use shared libraries, the linker adds instructions to query the OS to find where the library currently resides in memory
- Or to load the library, if this is the first program to use it
- Since the library isn't included in your executable, a user needs to have a copy of the shared library as well

Device Drivers

- To integrate a piece of hardware into your computer, the manufacturer needs to write a set of routines for controlling the device, called a *device driver*
- A driver is essentially a shared library, but it may be more tightly integrated into the OS
- Writing device drivers is a specialized skill, it requires a strong knowledge of the OS and intimate knowledge of the hardware

Third-Party Libraries

- Many companies market libraries for
  - graphics and graphic user interfaces
  - databases
  - networking
  - just about any aspect of programming
- The best sources for information on what's available are current magazines for programmers
  - Most programming magazines are specific to a particular OS or language
  - Check any large bookstore's computer magazine rack

Public Domain & Shareware Libraries

- There are hundreds of libraries available on the Internet, other networks, and CD collections
- Many of these libraries come from academia and hobbyists, and are in the public domain or made freely available as open source
  - Libraries aren't usually distributed as shareware
  - Most of these libraries include the source code
- Open source software sites like `sourceforge.net` and `freshmeat.net` are good places to start searching

The Life of a Programmer

- Each day in the life of a working programmer includes:
- **Designing**: Taking a problem and figuring out how to implement a solution
- **Coding and Documentation**: Writing source code and the documentation (comments in the code, design documents, user manuals) to accompany it
- **Testing and Debugging**: Putting your program through its paces and fixing problems

The Second Fundamental Rule of Programming

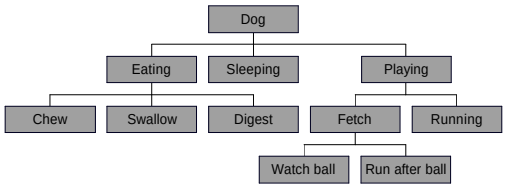
- In Session 1, we introduced the First Fundamental Rule of Programming:
  - You can't program something you don't know how to do
- Here's the Second Fundamental Rule:
- ***Someone else has probably figured it out already***
- Before starting a big programming task, do some research into existing algorithms and libraries
- There are also many complete programs which include the source code, study one similar to what you want to write

Designing a Program

- There are two modes of designing
  - Figuring out how to solve the problem
  - Figuring out how to implement the solution
- Designing a solution that's too hard to code doesn't solve the problem
- The most effective way to attack a problem is to break it down into smaller problems

Top-Down Design

- Starting at the top and breaking a problem down into pieces is called *top-down* design
- For example, in designing a Dog brain we might come up with this breakdown of activities:



Bottom-Up Design

- Not all design should proceed from the top down
- When you recognize that your problem is going to require a certain set of tools, it might be a good idea to design the tools next--this is *bottom-up* design
  - For example, while designing a business application you may realize that you'll need to do many types of statistics
  - By designing your toolkit of statistics routines (averages, standard deviation, etc.) you'll make subsequent top-down design easier
  - Recognizing when to shift from top-down to bottom-up and back is one of the arts of software design

Design Tools

- When designing, you don't want to slow yourself down by expressing your designs in a programming language (yet)
- Many people use flowcharts to design a program, pseudocode is another good approach
- For larger, multi-person projects, there's a whole field called **CASE** (Computer-Aided Software Engineering) which focuses on tools for software design
- Many CASE tools allow you to visually sketch the objects in your design and their relationships to each other

Prototyping

- Before going to the implementation stage, your design will need to be refined and made precise
- Here it's handy to use tools for rapid prototyping, implementing something quick-and-dirty to test out the design
- Interpreted languages like BASIC, TCL, and Python can be used for prototyping
- Most advanced GUI design tools will allow you to prototype your interface visually

Coding and Documentation

- The process of writing code consists of more than just typing
- Always remember that your audience, when writing code, is really the next programmer to work on the project
  - That might be you!
- When writing, make heavy use of comments to document what you're doing
- There will be ideas in your head as you write, such as how the routine you're writing fits into the Big Picture, that will be lost otherwise

Coding and Documentation (continued)

- Each class and each method should be prefaced by a short paragraph explaining its purpose
- Someone reading your code should be able to understand it by reading only the comments
- They should only have to read the C++ code to see how you implemented what you've described in the comments

The code implements the comments

Source Code Control

- In any good-sized project, you'll need to keep track of multiple versions of your program
  - You never know when you might need to go back to an older version
- When multiple programmers are working on the same program, you'll need to coordinate the individual contributions
- There are tools which help you do this, many are commercial products and some are quite elaborate
- There are also tools that will extract the comments from your source code and assemble them together as documentation

Testing and Debugging

- A program never behaves as expected the first time
- The process of programming consists of iteratively designing, coding, and debugging
- Some bugs are easy to find, because they are easily repeated
- Some bugs are sneakier and don't show up every time the program is run
- Many of these bugs are due to either
  - Failing to delete memory allocated with new
  - Indexing past the end of an array, or some other pointer operation that accesses the wrong memory location

Safe Compilers

- There are numerous tools for making programs safer
- For example, there is a bounds checker for the GNU compiler that catches attempts to go beyond the end of an array
- Two popular commercial packages, available on many OSes, are **Purify** and **Insure**

Safe Compilers (continued)

- Each of these adds code to your program to check boundaries, memory leaks, and many other potential errors
- Once your program has passed the tests of one of these tools, you can recompile without the tool to create a releasable version

Coverage

- Tools like Purify and Insure can do more than catch memory errors, they can also report on *coverage*
- Coverage is what portions of your program were actually executed this time
- For any given input, only a certain subset of your code is executed
- You need to design several different test inputs to exercise all the branches in your code
- Until you've tested as much of the code as possible, you're not ready to release the product

Debuggers

- Many bugs can't be found during compilation, even by the smartest compilers
- A valuable tool is a *debugger*, a program which allows you to step through your program one instruction at a time
- All modern debuggers can display the source code, highlighting each line of code as it is executed
  - You can mark a line and tell the debugger to stop when it gets there, this is called a *breakpoint*
  - They can also display the values of variables, and stop only when certain variables have certain values
- Tools like MS Visual C++ have a debugger built in

Other Languages

- C++ is only one of at least a hundred computer programming languages
- A working programmer should be familiar with other languages for several reasons:
  - Some languages are better suited for certain tasks than others
  - Understanding of other languages might improve your programming in your main language
  - The best way to solve a particular problem might be by combining more than one language

Multilingual Programs

- When a source code file is compiled into object code, there are no remnants of the original language
- By following strict rules for communicating between languages, object code from very different languages can be combined in one program
- The major concern is dealing with the context-switching stack consistently
- Some language interpreters are themselves written in C, and can be extended by writing additional C code

ANSI C

- The **C** programming language was written at AT&T Bell Labs in the '70s
  - By Dennis Ritchie and others
  - Successor to a language called BCPL
- C, along with the Unix OS, became very popular in academia and then spread to the entire industry
  - The ANSI C standard version of the language can now be found on almost every computer hardware and operating system
- C is C++ without the support for classes and objects
- You'll need a background in C to be effective in any of its derived languages

ANSI C (continued)

- C has a funkier I/O mechanism than C++ streams, called `stdio`
- Here's a C program which counts from 0 to 9

```
#include <stdio.h>

void main()
{
     int  i;

     for(i=0; i <= 9; i++)
          printf("%d ", i);

     printf("\n");
}
```


Objective C

- C++ is one approach to adding object-oriented techniques to C, **Objective C** is a different approach
- Objective C has a simpler OO mechanism: instead of calling a method on an object (C++), you send the object a *message*
  - If the object doesn't recognize the message, it passes it to its ancestor

Objective C (continued)

- Here's an Objective C example

```
#import <objc/Object.h>
#import "List.h"          // A list class

main()
{
  id list;                // id is a data type for
                           // representing an object

  list = [List new];      // create an instance of List
  [list addEntry: 5];     // send a message to list
  [list print];
  [list addEntry: 6];
  [list addEntry: 3];
  [list print];
  [list free];            // get rid of object
}
```

Objective C vs. C++

- Messages are more flexible than methods, but they're slower since your program has to do the routing
  - In C++, most of the work involved in calling a method is done in the compiler
  - In Objective C, a message is routed by routines incorporated into your program, obtained from the Objective C runtime library
- Objective C and C++ are so distinct that they can be combined without interfering with each other's grammar

Java

- **Java** is another object-oriented language derived from C, invented at Sun Microsystems
  - Java has many of the best features of both Objective C and C++
- It's designed to be used as a language for transmitting small programs (*applets*) via the WWW
  - It has many features to support secure programs
- Java is compiled into a special **bytecode** (not machine language) which is interpreted
  - This code consists of simple instructions similar to machine language, but not specific to any particular machine

Java (continued)

- All major web browsers understand Java, and most operating systems have Java implementations
- Java is an important language for cross-platform software development
- The Web address for Java is:
  - **`java.sun.com`**

Java Example

- This example displays the text "Hello World" on any Java-equipped browser, on any computer

```
import java.awt.*;
import java.applet.Applet;

public class helloworld extends Applet {

    public void init() {
      resize(150,25);
    }

    public void paint(Graphics g) {
      g.drawString("Hello World",50,25);
    }
}
```


Procedural Languages

- C and its derivatives are one family of a type of language called *procedural*
- Procedural languages are structured around the procedure, another name for routine
- Each of the steps in a procedural language are specified explicitly
- There are many other procedural languages in popular use

Basic

- **BASIC** (Beginner's All-Purpose Symbolic Instruction Code) has been around since the mid-60's
- Early versions of BASIC were crude, variables had short one- or two-character names, every line had a line number, subroutines couldn't have names, etc.
- BASIC has proliferated into many forms and on most machines
  - It is usually interpreted, though it can be compiled
- Modern BASIC (like MS Visual Basic) has all the bells and whistles of C or any other modern language
- Many beginners choose BASIC as a first language

Pascal & Its Descendants

- **Pascal** was designed in the 60s by Niklaus Wirth in Switzerland
- Pascal was intended as a language for teaching computer science, but it gained popularity and was used for application development
- In 1982, Wirth developed a descendant of Pascal, **Modula 2**, designed for business and scientific use
  - The current project from Wirth is **Oberon**
- Modula 2 pioneered many of the mechanisms for modularity that migrated to other languages like C++ and Ada

49 Session 8

Other Procedural Languages

- **Fortran** (Formula Translation): Developed in 1956 by John Backus; specialized for scientific and engineering computations, still in use in many high-performance applications
- **COBOL** (Common Business-Oriented Language): Developed in 1959 by Grace Hopper; very verbose language for business, still in use
- **Ada** (named after Ada Augusta, Lady Lovelace): Dept. of Defense, 1981; general-purpose language designed to enforce strict modularity and more robust programs, primarily used by defense contractors

50 Session 8

Forth

- **Forth**, developed in the late 70s by Charles Moore, is a small language with a very simple grammar

```
3 4 + .
```

- Forth words operate on a stack, the statement above adds the numbers 3 and 4
  - 3 and 4 each push a value onto the stack
  - + pops two values off the stack, adds them, and pushes the answer
  - `.` pops the top value off the stack and prints it
- Forth is usually interpreted

51 Session 8

Forth (continued)

- Forth programs consist of definitions of new words to manipulate the stack, essentially the same concept as routines in other languages
- `: ADD2 2 + ;` defines a new word which adds 2 to whatever's on the stack
- A Forth interpreter is a relatively easy program to write, so Forth has been used in many applications and embedded into other software and hardware

52 Session 8

Forth (continued)

- Many of the concepts in Forth are used throughout computer science
- For example, the printer description format **Postscript** is actually a fully-functional programming language, similar to Forth

53 Session 8

Smalltalk

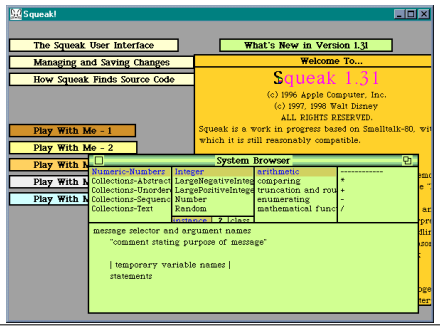
- **Smalltalk** is an object-oriented language developed by Xerox in the 70s and 80s
- Everything in Smalltalk is an object, even a number like 2
- From the first versions, Smalltalk environments have had GUI tools for finding and using classes stored in libraries
  - These browsers listed all the classes available
  - Select a class and a list of its attributes and methods is displayed

54 Session 8

Smalltalk (continued)

- Smalltalk is considered to be one of the best environments for object-oriented programming, and is largely responsible for the rise of OOP in the 80s and 90s
- Concerns that Smalltalk was too slow led to languages like C++ and Objective C, but modern Smalltalk is quite fast and has a strong following
- A free, multi-platform version of Smalltalk called **Squeak** is available via:
  - `www.squeak.org`

Squeak screen shot



Languages for Artificial Intelligence

- **Lisp** (List Processing) was developed in the late 50's as a language for symbolic computation
- Lisp is very good at dealing with words and lists of words
  - Typically, Lisp implementations have not been strong on numerical computations
- Much research in artificial intelligence has used Lisp and its descendants like **Common Lisp** and **Scheme**
- The language **Logo** is essentially a derivative of Lisp, it was designed as a language for teaching programming to children

Prolog

- **Prolog** (Programming in Logic) is a much different approach to programming than any of the procedural (*imperative*) languages
- Prolog is a *declarative* language, the programmer makes statements and queries, and the computer figures out how to resolve the queries
- Prolog programming requires a different mindset, since you're not telling the computer what to do step-by-step
- Instead, you state facts and the built-in reasoning engine digests them and answers queries

Prolog (continued)

```
is_male(bob).                 states that bob is a male
is_sibling(mike,bob).         bob is mike's sibling
is_brother(A,B) :-            B is A's brother if
   is_sibling(A,B),           B is a sibling of A
   is_male(B).                and B is male
```

- The query **`?- is_brother(mike,bob).`** would produce the answer yes
- Prolog and similar languages can be combined with C to add artificial intelligence to an application

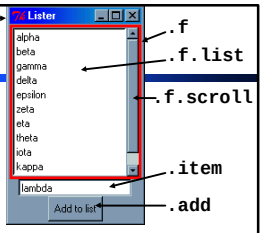
Scripting Languages

- There are a number of languages used for rapid software development or *scripting*
- They're often used for short programs, or for gluing together various libraries and tools
- Programming for the WWW often involves scripting
- The most popular scripting languages:
  - PERL `www.perl.org`
  - Python `www.python.org`
  - Tcl/Tk `tcl.sourceforge.net` `tcl.activestate.com`

TCL/TK

- **TCL** (Tool Command Language) and **TK** (Toolkit) were developed by John Ousterhout at UC Berkeley
  - TCL is the language (a procedural language like C or BASIC) and TK is a GUI toolkit
- TCL/TK is freely available on many OSes (Unix, Mac, Windows) and is a good language for developing small applications on multiple OSes
- The TCL interpreter can be incorporated into your own program and used as an end-user programming language

TCL/TK Example



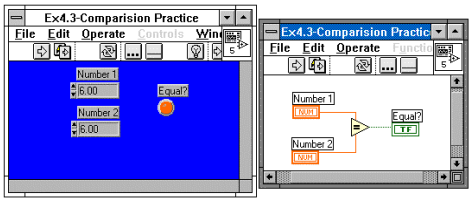

```
wm title . "Lister"

frame .f -borderwidth 3 \
      -background red
listbox .f.list -yscrollcommand ".f.scroll set" \
      -background white
scrollbar .f.scroll -command {.f.list  yview}
pack .f.list .f.scroll -side left -fill both -expand true

entry .item -width 20 -relief sunken - textvariable newItem
button .add  -text "Add to list" \
      -command {.f.list insert end $ newItem}
pack .f .item .add -side top
```


Visual Programming

- This is by no means an exhaustive list of computer languages
  - Not all languages are expressed using text
- Visual languages: languages depicted (and edited) graphically
- LabView: `www.natinst.com/labview`
- Prograph: `www.pictorius.com/prograph.html`
- Stagecast: `www.stagecast.com`
- AgentSheets: `www.agentsheets.com`

LabView

- This picture shows both the user interface and the LabView source for a simple program to compare two numbers



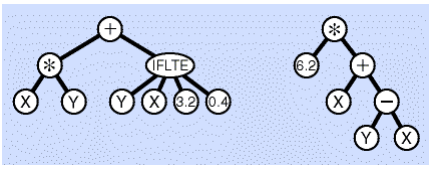
Automatic Programming

- Why write any code at all?
- Ultimately, we'd like to specify just the goals of a program, and have the computer figure out how to do it
- There are many approaches to machine learning: expert systems, neural networks, fuzzy logic, evolutionary algorithms, artificial life
- One promising area is Genetic Programming

`www.genetic-programming.org`

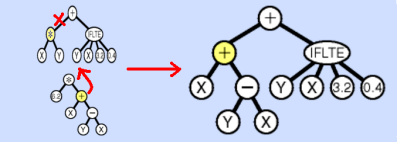
Genetic Programming

- In GP programs are represented by trees
- The programmer specifies:
  - the number and types of inputs — `X Y (floats)`
  - the operations available — `+ - * / IFs`
  - a goal for the output — `Z = func(X,Y)`



Genetic Programming (continued)

- The GP system creates a population of random programs and has them compete for survival
  - programs are tested with many input values, the best performers reproduce via crossover
  - random mutations are also introduced
- A winning solution is evolved by survival of the fittest



67 Session 8

The Future

- In 1999, a GP system for designing electrical circuits invented several patentable circuits
  - This could be considered competitive with human intelligence, at least within this limited domain
- Truly intelligent and autonomous programs are still in the future, human programmers aren't obsolete yet!

68 Session 8

Introduction to Programming

- The goal of this course was to introduce key topics in computer programming
- You'll revisit all of these topics in depth as you study a programming language thoroughly
- Learn a language well, then learn others
- Always keep your mind open to better ways to write software

69 Session 8