CISC 181 midterm overview

- This Thursday – 75 minutes
- Worth 20% of your grade
- Covers topics from class in chapters listed on course page up to March 18 inclusive
  - Will not test on ncurses, specific time functions, cerr, Makefile/header file stuff (aka Chap. 11.1), formatting numbers for output, overloading as member, sorting, “static” functions/variables…
- Question types
  - Language feature/concept definitions and explanations
  - Write a function that does X
  - If we call function f() with args a, b, what does it return/print?
  - Probably some “self-test exercises” from textbook

Topic list

• C++ basics	Chap. 1
• Control structures	Chap. 2
• Functions	Chap. 3
• Parameters	Chap. 4-4.2
• Arrays & C strings	Chap. 5 (skip 5.3), 9-9.1
• Structs, pointers, & dynamic allocation	Chap. 6.1, 10-10.2
• File I/O	Chap. 9.2, 12-12.2
• Classes, C++ strings	Chap. 6.2, 7-7.2, 8, 9.3

Chap. 1: C++ basics

- Introduction to C++
  - Origins, Object-Oriented Programming, Terms
- Variables, Expressions, and Assignment Statements
- Console Input/Output
- Program Style
- Libraries and Namespaces

Chap. 2: Flow of Control

- Boolean Expressions
  - Building, Evaluating & Precedence Rules
- Branching Mechanisms
  - if-else
  - switch
  - Nesting if-else
- Loops
  - While, do-while, for
  - Nesting loops

Chap. 3: Functions

- Predefined Functions
  - Those that return a value and those that don't
- Programmer-defined Functions
  - Defining, Declaring, Calling
  - Recursive Functions
- Scope Rules
  - Local variables
  - Global constants and global variables
  - Blocks, nested scopes

Chap. 4-4.2: Parameters

- Parameters
  - Call-by-value
  - Call-by-reference
  - Mixed parameter-lists
- Overloading and Default Arguments
  - Examples, Rules

Chap. 5 (skip 5.3), 9-9.1: Arrays & C strings

- Introduction to Arrays
  - Declaring and referencing arrays
  - For-loops and arrays
  - Arrays in memory
- Arrays in Functions
  - Arrays as function arguments, return values
- Multidimensional Arrays
- An Array Type for Strings
  - C-Strings

Chap. 6.1, 10-10.2: Structs, pointers, & dynamic allocation

- Structures
  - Structure types
  - Structures as function arguments
  - Initializing structures
- Pointers
  - Pointer variables
  - Memory management
- Dynamic Arrays
  - Creating and using
  - Pointer arithmetic

Chap. 9.2, 12-12.2

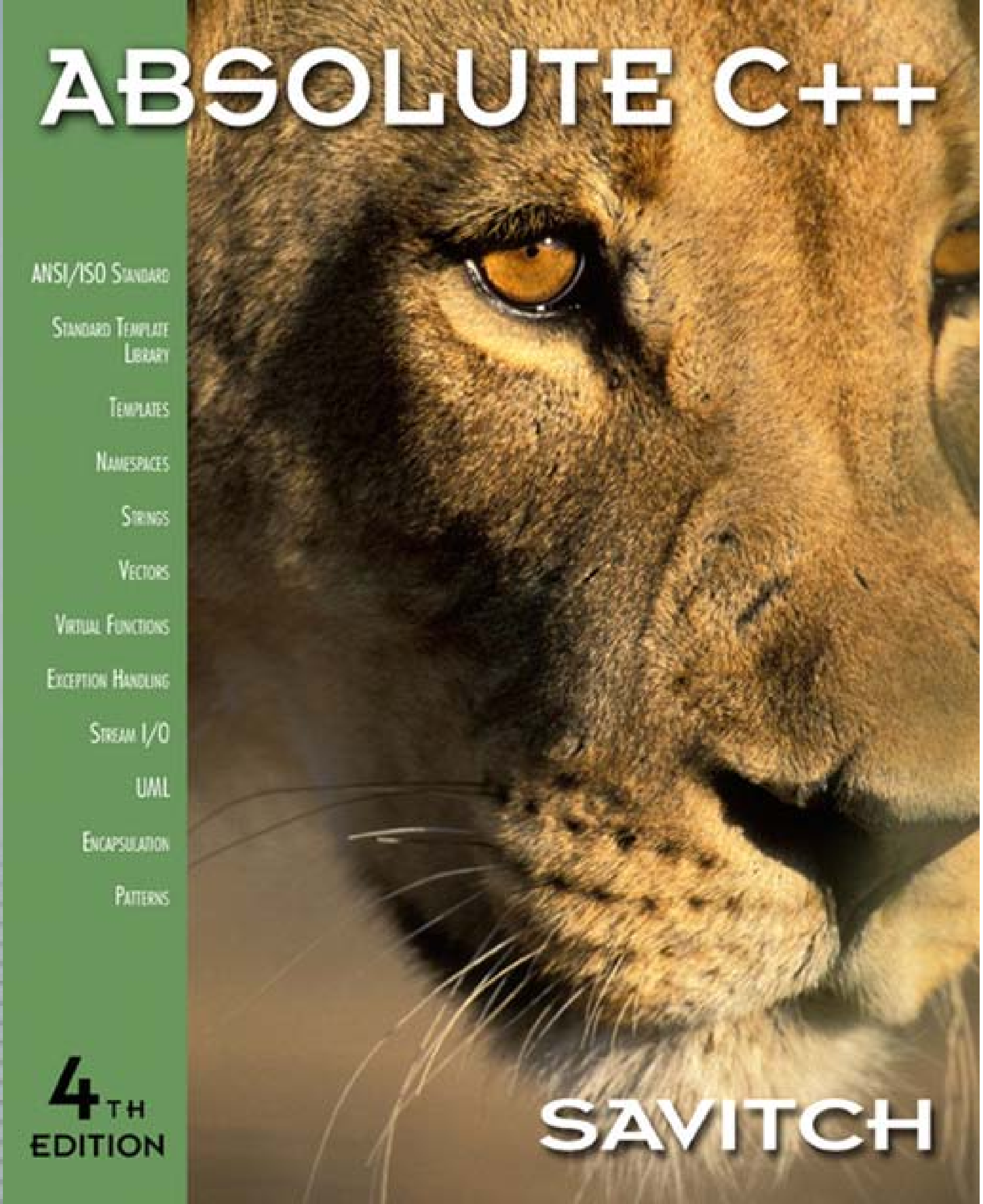
- Character Manipulation Tools
  - Character I/O
  - get, put member functions
- I/O Streams
  - File I/O
  - Character I/O
- Tools for Stream I/O
  - File names as input

Chap. 6.2, 7-7.2: Class basics

- Classes
  - Defining, member functions
  - Public and private members
  - Accessor and mutator functions
  - Structures vs. classes
- Constructors
  - Definitions
  - Calling
- More Tools
  - const parameter modifier
  - Inline functions

Chap. 8, 9.3: More on classes, C++ strings

- Basic Operator Overloading
  - Unary operators
  - As member functions
- Friends and Automatic Type Conversion
  - Friend functions, friend classes
  - Constructors for automatic type conversion
- References and More Overloading
  - << and >>
  - Not = , [], ++, --
- Standard Class string
  - String processing



Chapter 1

C++ Basics

Learning Objectives

- Introduction to C++
  - Origins, Object-Oriented Programming, Terms
- Variables, Expressions, and Assignment Statements
- Console Input/Output
- Program Style
- Libraries and Namespaces

Display 1.1
A Sample C++ Program (1 of 2); notice library include & namespace directive

Display 1.1 A Sample C++ Program

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

int main( )
{
    int numberOfLanguages;

    cout << "Hello reader.\n"
         << "Welcome to C++.\n";

    cout << "How many programming languages have you used? ";
    cin >> numberOfLanguages;

    if (numberOfLanguages < 1)
        cout << "Read the preface. You may prefer\n"
             << "a more elementary book by the same author.\n";
    else
        cout << "Enjoy the book.\n";

    return 0;
}
```

Data Types:
Display 1.2 Simple Types (1 of 2)

Display 1.2 Simple Types

TYPE NAME	MEMORY USED	SIZE RANGE	PRECISION
`short` (also called `short int`)	2 bytes	−32,768 to 32,767	Not applicable
`int`	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`long` (also called `long int`)	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`float`	4 bytes	approximately 10^{-38} to 10^{38}	7 digits
`double`	8 bytes	approximately 10^{-308} to 10^{308}	15 digits

Data Types:
Display 1.2 Simple Types (2 of 2)

`long double`	10 bytes	approximately 10^{-4932} to 10^{4932}	19 digits
`char`	1 byte	All ASCII characters (Can also be used as an integer type, although we do not recommend doing so.)	Not applicable
`bool`	1 byte	`true, false`	Not applicable

The values listed here are only sample values to give you a general idea of how the types differ. The values for any of these entries may be different on your system. *Precision* refers to the number of meaningful digits, including digits in front of the decimal point. The ranges for the types `float`, `double`, and `long double` are the ranges for positive numbers. Negative numbers have a similar range, but with a negative sign in front of each number.

Assigning Data

- Initializing data in declaration statement
  - Results "undefined" if you don’t!
    - int myValue = 0;
- Assigning data during execution
  - **Lvalues** (left-side) & **Rvalues** (right-side)
    - Lvalues must be variables
    - Rvalues can be any expression
    - Example:
      distance = rate * time;
      Lvalue: "distance"
      Rvalue: "rate * time"

Assigning Data: Shorthand Notations

- Display, page 14

EXAMPLE	EQUIVALENT TO
`count += 2;`	`count = count + 2;`
`total -= discount;`	`total = total - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rushFactor;`	`time = time/rushFactor;`
`change %= 100;`	`change = change % 100;`
`amount *= cnt1 + cnt2;`	`amount = amount * (cnt1 + cnt2);`

Data Assignment Rules

- Compatibility of Data Assignments
  - Type mismatches
    - General Rule: Cannot place value of one type into variable of another type
  - intVar = 2.99; // 2 is assigned to intVar!
    - Only integer part "fits", so that's all that goes
    - Called "implicit" or "automatic type conversion"
  - Literals
    - 2, 5.75, "Z", "Hello World"
    - Considered "constants": can't change in program

Literal Data (& comments)

- Literals
  - Examples:
    - 2 // Literal constant int
    - 5.75 /* Literal constant double */
    - "Z" // Literal constant char
    - "Hello World" // Literal constant string
- Cannot change values during execution
- Called "literals" because you "literally typed" them in your program!

Escape Sequences

- "Extend" character set
- Backslash, \ preceding a character
  - Instructs compiler: a special "escape character" is coming
  - Following character treated as "escape sequence char"
  - Commonly-used: \n, \\ (not a comment!), \’, \”

Constants

- Naming your constants
  - Literal constants are "OK", but provide little meaning
    - e.g., seeing “24” in a program tells nothing about what it represents
- Use named constants instead
  - Meaningful name to represent data
    const int NUMBER_OF_STUDENTS = 24;
    - Called a "declared constant" or "named constant"
    - Now use its name wherever needed in program
    - Added benefit: changes to value result in one fix

Arithmetic Precision

- Precision of Calculations
  - VERY important consideration!
    - Expressions in C++ might not evaluate as you'd "expect"!
  - "Highest-order operand" determines type of arithmetic "precision" performed
  - Common pitfall!

Arithmetic Precision Examples

- Examples:
  - 17 / 5 evaluates to 3 in C++!
    - Both operands are integers
    - Integer division is performed!
  - 17.0 / 5 equals 3.4 in C++!
    - Highest-order operand is "double type"
    - Double "precision" division is performed!
  - int intVar1 =1, intVar2=2;
    intVar1 / intVar2;
    - Performs integer division!
    - Result: 0!

Type Casting

- Casting for Variables
  - Can add ".0" to literals to force precision arithmetic, but what about variables?
    - We can’t use "myInt.0"!
  - static_cast<double>intVar
  - Explicitly "casts" or "converts" intVar to double type
    - Result of conversion is then used
    - Example expression:
      doubleVar = static_cast<double>intVar1 / intVar2;
      - Casting forces double-precision division to take place among two integer variables!

Type Casting

- Two types
  - Implicit—also called "Automatic"
    - Done FOR you, automatically
      17 / 5.5
      This expression causes an "implicit type cast" to take place, casting the 17 → 17.0
  - Explicit type conversion
    - Programmer specifies conversion with cast operator
      (double)17 / 5.5
      Same expression as above, using explicit cast
      (double)myInt / myDouble
      More typical use; cast operator on variable

Shorthand Operators

- Increment & Decrement Operators
  - Just short-hand notation
  - Increment operator, ++
    intVar++; is equivalent to
    intVar = intVar + 1;
  - Decrement operator, --
    intVar--;   is equivalent to
    intVar = intVar – 1;

Shorthand Operators: Two Options

- Post-Increment
  intVar++
  - Uses current value of variable, THEN increments it
- Pre-Increment
  ++intVar
  - Increments variable first, THEN uses new value
- "Use" is defined as whatever "context" variable is currently in
- No difference if "alone" in statement:
  intVar++; and ++intVar; → identical result

Console Input/Output

- I/O objects cin, cout
- Defined in the C++ library called <iostream>
- Must have these lines (called pre-processor directives) near start of file:
  - #include <iostream>
    using namespace std;
  - Tells C++ to use appropriate library so we can use the I/O objects cin, cout

Console Output

- What can be outputted?
  - Any data can be outputted to display screen
    - Variables
    - Constants
    - Literals
    - Expressions (which can include all of above)
  - cout << numberOfGames << " games played.";
    2 values are outputted:
    "value" of variable numberOfGames,
    literal string " games played."
- Cascading: multiple values in one cout

Separating Lines of Output

- New lines in output
  - Recall: "\n" is escape sequence for the char "newline"
- A second method: object **endl**
- Examples:

  cout << "Hello World\n";

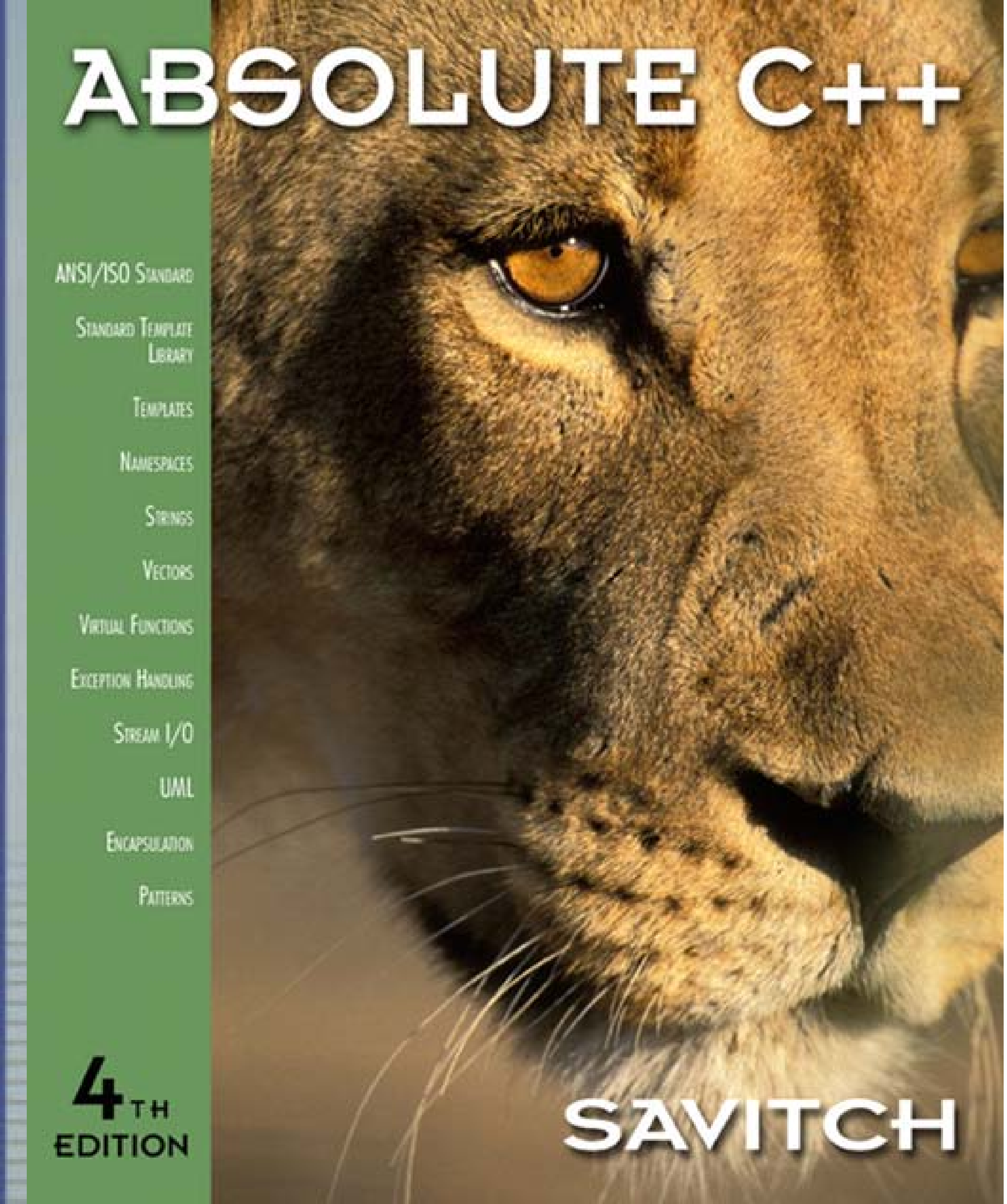
    - Sends string "Hello World" to display, & escape sequence "\n", skipping to next line

  cout << "Hello World" << endl;

    - Same result as above

Input Using cin

- cin for input, cout for output
- Differences:
  - ">>" (extraction operator) points opposite
    - Think of it as "pointing toward where the data goes"
  - Object name "cin" used instead of "cout"
  - No literals allowed for cin
    - Must input "to a variable"
- cin >> num;
  - Waits on-screen for keyboard entry
  - Value entered at keyboard is "assigned" to num



Chapter 2

Flow of Control

Learning Objectives

- Boolean Expressions
  - Building, Evaluating & Precedence Rules
- Branching Mechanisms
  - if-else
  - switch
  - Nesting if-else
- Loops
  - While, do-while, for
  - Nesting loops

Boolean Expressions: **Display 2.1** Comparison Operators

- Data type bool (**true** or **false**)
- Logical Operators
  - Logical AND (&&)
  - Logical OR (||)

Display 2.1 **Comparison Operators**

MATH SYMBOL	ENGLISH	C++ NOTATION	C++ SAMPLE	MATH EQUIVALENT
=	Equal to	`==`	`x + 7 == 2*y`	$x + 7 = 2y$
≠	Not equal to	`!=`	`ans != 'n'`	$ans \neq 'n'$
<	Less than	`<`	`count < m + 3`	$count < m + 3$
≤	Less than or equal to	`<=`	`time <= limit`	$time \leq limit$
>	Greater than	`>`	`time > limit`	$time > limit$
≥	Greater than or equal to	`>=`	`age >= 21`	$age \geq 21$

Display 2.3
Precedence of Operators (1 of 4)

Display 2.3 Precedence of Operators

::	Scope resolution operator
.	Dot operator
->	Member selection
[]	Array indexing
( )	Function call
++	Postfix increment operator (placed after the variable)
--	Postfix decrement operator (placed after the variable)
++	Prefix increment operator (placed before the variable)
--	Prefix decrement operator (placed before the variable)
!	Not
-	Unary minus
+	Unary plus
*	Dereference
&	Address of
new	Create (allocate memory)
delete	Destroy (deallocate)
delete[]	Destroy array (deallocate)
sizeof	Size of object
( )	Type cast

Highest precedence (done first)

Display 2.3
Precedence of Operators (2 of 4)

* / %	Multiply Divide Remainder (modulo)
+ −	Addition Subtraction
<< >>	Insertion operator (console output) Extraction operator (console input)

Lower precedence (done later)

Display 2.3
Precedence of Operators (3 of 4)

Display 2.3 Precedence of Operators

All operators in part 2 are of lower precedence than those in part 1.

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal
!=	Not equal
&&	And
||	Or

Display 2.3
Precedence of Operators (4 of 4)

`=`	Assignment
`+=`	Add and assign
`-=`	Subtract and assign
`*=`	Multiply and assign
`/=`	Divide and assign
`%=`	Modulo and assign
`? :`	Conditional operator
`throw`	Throw an exception
`,`	Comma operator

Lowest precedence (done last)

Precedence Examples

- Arithmetic before logical
  - x + 1 > 2 || x + 1 < -3 means:
    - (x + 1) > 2 || (x + 1) < -3
- Short-circuit evaluation
  - (x >= 0) && (y > 1)
  - Be careful with increment operators!
    - (x > 1) && (y++)
- Integers as boolean values
  - All non-zero values → true
  - Zero value → false

Branching Mechanisms

- if-else statements
  - Choice of two alternate statements based on condition expression
  - Example:

```
if (hrs > 40)
  grossPay = rate*40 + 1.5*rate*(hrs-40);
else
  grossPay = rate*hrs;
```

Compound/Block Statement

- Only "get" one statement per branch
- Must use compound statement { }
  for multiples
  - Also called a "block" stmt
- Each block should have block statement
  - Even if just one statement
  - Enhances readability

Compound Statement in Action (one style of indenting)

- ```
  if (myScore > yourScore)
  {
        cout << "I win!\n";
        wager = wager + 100;
  }
  else
  {
        cout << "I wish these were golf scores.\n";
        wager = 0;
  }
  ```
  ```

Common Pitfalls

- Operator "=" vs. operator "=="
- One means "assignment" (=)
- One means "equality" (==)
  - VERY different in C++!
  - Example:
    ```
    if (x = 12)  ←Note operator used!
        Do_Something
    else
        Do_Something_Else
    ```

The Optional else

- else clause is optional
  - If, in the false branch (else), you want "nothing" to happen, leave it out
  - Example:

```
if (sales >= minimum)
    salary = salary + bonus;
cout << "Salary = %" << salary;
```

  - Note: nothing to do for false condition, so there is no else clause!
  - Execution continues with cout statement

Nested Statements

- if-else statements contain smaller statements
  - Compound or simple statements (we've seen)
  - Can also contain any statement at all, including another if-else stmt!
  - Really should use { } to make block for clarity
  - Example:

```
if (speed > 55)
    if (speed > 80)
        cout << "You're really speeding!";
    else
        cout << "You're speeding.";
```

    - Note proper indenting!

Multiway if-else

- Not new, just different indenting
- Avoids "excessive" indenting
  - Syntax:

Multiway if-else Statement

SYNTAX

```
if (Boolean_Expression_1)
    Statement_1
else if (Boolean_Expression_2)
    Statement_2
        .
        .
        .
else if (Boolean_Expression_n)
    Statement_n
else
    Statement_For_All_Other_Possibilities
```

switch Statement Syntax

switch Statement

SYNTAX

```
switch (Controlling_Expression)
{
    case Constant_1:
        Statement_Sequence_1
        break;
    case Constant_2:
        Statement_Sequence_2
        break;
                    .
                    .
                    .
    case Constant_n:
        Statement_Sequence_n
        break;
    default:
        Default_Statement_Sequence
}
```

You need not place a **break** *statement in each case. If you omit a* **break***, that case continues until a* **break** *(or the end of the* **switch** *statement) is reached.*

The switch Statement in Action

EXAMPLE

```
int vehicleClass;
double toll;
cout << "Enter vehicle class: ";
cin >> vehicleClass;

switch (vehicleClass)
{
    case 1:
        cout << "Passenger car.";
        toll = 0.50;
        break;
    case 2:
        cout << "Bus.";
        toll = 1.50;
        break;
    case 3:
        cout << "Truck.";
        toll = 2.00;
        break;
    default:
        cout << "Unknown vehicle class!";
}
```

If you forget this **break**, *then passenger cars will pay $1.50.*

The switch: multiple case labels

- Execution "falls through" until break
  - switch provides a "point of entry"
  - Example:

```
case "A":
case "a":
    cout << "Excellent: you got an "A"!\n";
    break;
case "B":
case "b":
    cout << "Good: you got a "B"!\n";
    break;
```

  - Note multiple labels provide same "entry"

Conditional Operator

- Also called "ternary operator"
  - Allows embedded conditional in expression
  - Essentially "shorthand if-else" operator
  - Example:
    ```
    if (n1 > n2)
        max = n1;
    else
        max = n2;
    ```
  - Can be written:
    ```
    max = (n1 > n2) ? n1 : n2;
    ```
    - "?" and ":" form this "ternary" operator

Loops

- 3 Types of loops in C++
  - while
    - Most flexible
    - No "restrictions"
  - do-while
    - Least flexible
    - Always executes loop body at least once
  - for
    - Natural "counting" loop

while Loops Syntax

Syntax for while and do-while Statements

A while STATEMENT WITH A SINGLE STATEMENT BODY

```
while (Boolean_Expression)
    Statement
```

A while STATEMENT WITH A MULTISTATEMENT BODY

```
while (Boolean_Expression)
{
    Statement_1
    Statement_2
       .
       .
       .
    Statement_Last
}
```

while Loop Example

- Consider:

```
count = 0;                  // Initialization
while (count < 3)           // Loop Condition
{
    cout << "Hi ";          // Loop Body
    count++;                // Update expression
}
```

  - Loop body executes how many times?

do-while Loop Syntax

A do–while STATEMENT WITH A SINGLE-STATEMENT BODY

```
do
    Statement
while (Boolean_Expression);
```

A do–while STATEMENT WITH A MULTISTATEMENT BODY

```
do
{
    Statement_1
    Statement_2
      .
      .
      .
    Statement_Last
} while (Boolean_Expression);
```

Do not forget the final semicolon.

do-while Loop Example

- ```
  count = 0;              // Initialization
  do
  {
      cout << "Hi ";      // Loop Body
      count++;            // Update expression
  } while (count < 3);    // Loop Condition
  ```
  - Loop body executes how many times?
  - do-while loops always execute body at least once!
  ```

for Loop Syntax

for (Init_Action; Bool_Exp; Update_Action)
    Body_Statement

- Like if-else, Body_Statement can be a block statement
  - Much more typical

for Loop Example

- for (count=0;count<3;count++)
  ```
  {
      cout << "Hi ";          // Loop Body
  }
  ```
- How many times does loop body execute?
- Initialization, loop condition and update all "built into" the for-loop structure!
- A natural "counting" loop

Loop Pitfalls: Misplaced ;

- Watch the misplaced ; (semicolon)
  - Example:

    ```
    while (response != 0) ;←
    {
        cout << "Enter val: ";
        cin >> response;
    }
    ```
  - Notice the ";" after the while condition!
- Result here: INFINITE LOOP!

Loop Pitfalls: Infinite Loops

- Loop condition must evaluate to false at some iteration through loop
  - If not → infinite loop.
  - Example:
    ```
    while (1)
    {
        cout << "Hello ";
    }
    ```
  - A perfectly legal C++ loop → always infinite!
- Infinite loops can be desirable
  - e.g., "Embedded Systems"

The break and continue Statements

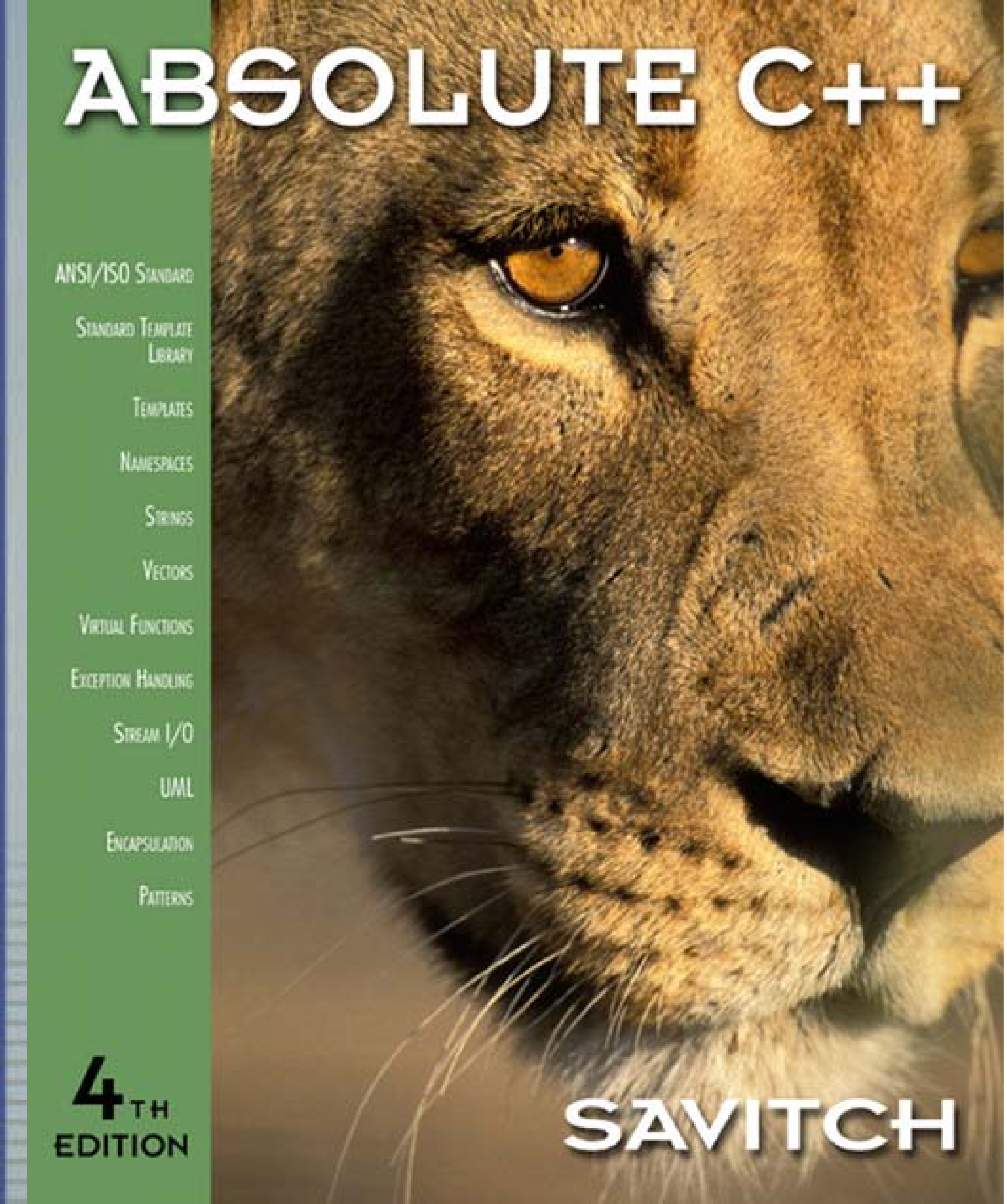
- Flow of Control
  - Recall how loops provide "graceful" and clear flow of control in and out
  - In RARE instances, can alter natural flow
- break;
  - Forces loop to exit immediately.
- continue;
  - Skips rest of loop body
- These statements violate natural flow
  - Only used when absolutely necessary!

Nested Loops

- Recall: ANY valid C++ statements can be inside body of loop
- This includes additional loop statements!
  - Called "nested loops"
- Requires careful indenting:

```
for (outer=0; outer<5; outer++)
    for (inner=7; inner>2; inner--)
        cout << outer << inner;
```

  - Notice no { } since each body is one statement
  - Good style dictates we use { } anyway



Chapter 3

Function Basics

Learning Objectives

- Predefined Functions
  - Those that return a value and those that don't
- Programmer-defined Functions
  - Defining, Declaring, Calling
  - Recursive Functions
- Scope Rules
  - Local variables
  - Global constants and global variables
  - Blocks, nested scopes

Predefined Functions

- Libraries full of functions for our use!
- Two types:
  - Those that return a value
  - Those that do not (void)
- Must "#include" appropriate library
  - e.g.,
    - <cmath>, <cstdlib> (Original "C" libraries)
    - <iostream> (for cout, cin)

Using Predefined Functions

- Math functions very plentiful
  - Found in library <cmath.h>
  - Most return a value (the "answer")
- Example: theRoot = sqrt(9.0);
  - Components:
    sqrt = name of library function
    theRoot = variable used to assign "answer" to
    9.0 = argument or "starting input" for function
  - In I-P-O:
    - I = 9.0
    - P = "compute the square root"
    - O = 3, which is returned & assigned to theRoot

The Function Call

- Back to this assignment:

  theRoot = sqrt(9.0);

  - The expression "sqrt(9.0)" is known as a function *call*, or function *invocation*
  - The argument in a function call (9.0) can be a literal, a variable, or an expression
  - The call itself can be part of an expression:
    - bonus = sqrt(sales)/10;
    - A function call is allowed wherever it's legal to use an expression of the function's return type

Even More Math Functions: **Display 3.2** Some Predefined Functions (1 of 2)

Display 3.2 Some Predefined Functions

NAME	DESCRIPTION	TYPE OF ARGUMENTS	TYPE OF VALUE RETURNED	EXAMPLE	VALUE	LIBRARY HEADER
`sqrt`	Square root	`double`	`double`	`sqrt(4.0)`	`2.0`	`cmath`
`pow`	Powers	`double`	`double`	`pow(2.0,3.0)`	`8.0`	`cmath`
`abs`	Absolute value for `int`	`int`	`int`	`abs(-7)` `abs(7)`	`7` `7`	`cstdlib`
`labs`	Absolute value for `long`	`long`	`long`	`labs(-70000)` `labs(70000)`	`70000` `70000`	`cstdlib`
`fabs`	Absolute value for `double`	`double`	`double`	`fabs(-7.5)` `fabs(7.5)`	`7.5` `7.5`	`cmath`

Even More Math Functions: **Display 3.2** Some Predefined Functions (2 of 2)

ceil	Ceiling (round up)	double	double	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	Floor (round down)	double	double	floor(3.2) floor(3.9)	3.0 3.0	cmath
exit	End program	int	void	exit(1);	None	cstdlib
rand	Random number	None	int	rand( )	Varies	cstdlib
srand	Set seed for rand	unsigned int	void	srand(42);	None	cstdlib

Remember use of time functions to get different seed for each program run

Predefined Void Functions

- No returned value
- Performs an action, but sends no "answer"
- When called, it's a statement itself
  - exit(1);   // No return value, so not assigned
    - This call terminates program
    - void functions can still have arguments
- All aspects same as functions that "return a value"
  - They just don't return a value!

Random Number Generator

- Return "randomly chosen" number
- Used for simulations, games
  - rand()
    - Takes no arguments
    - Returns value between 0 & RAND_MAX
  - Scaling
    - Squeezes random number into smaller range
      rand() % 6
    - Returns random value between 0 & 5
  - Shifting
    rand() % 6 + 1
    - Shifts range between 1 & 6 (e.g., die roll)
  - Random double between 0.0 & 1.0:
    (RAND_MAX – rand())/static_cast<double>(RAND_MAX)
    - Type cast used to force double-precision division

Random Number Seed

- Pseudorandom numbers
  - Calls to rand() produce given "sequence" of random numbers
- Use "seed" to alter sequence
  srand(seed_value);
  - void function
  - Receives one argument, the "seed"
  - Can use any seed value, including system time:
    srand(time(0));
  - time() returns system time as numeric value
  - Library <time> contains time() functions

Programmer-Defined Functions

- Write your own functions!
- Building blocks of programs
  - Divide & Conquer
  - Readability
  - Re-use
- Your "definition" can go in either:
  - Same file as main()
  - Separate file so others can use it, too

Components of Function Use

- 3 Pieces to using functions:
  - Function Declaration/prototype
    - Information for compiler
    - To properly interpret calls
  - Function Definition
    - Actual implementation/code for what function does
  - Function Call
    - Transfer control to function

Function Declaration

- Also called function prototoype
- An "informational" declaration for compiler
- Tells compiler how to interpret calls
  - Syntax:
    <return_type> FnName(<formal-parameter-list>);
  - Example:
    ```
    double totalCost(    int numberParameter,
                         double priceParameter);
    ```
- Placed before any calls
  - In declaration space of main()
  - Or above main() in global space

Function Definition

- Implementation of function
- Just like implementing function main()
- Example:

```
double totalCost(       int numberParameter,
                        double priceParameter)
{
    const double TAXRATE = 0.05;
    double subTotal;
    subtotal = priceParameter * numberParameter;
    return (subtotal + subtotal * TAXRATE);
}
```

- Notice proper indenting

Function Definition Placement

- Placed after function main()
  - NOT "inside" function main()!
- Functions are "equals"; no function is ever "part" of another
- Formal parameters in definition
  - "Placeholders" for data sent in
    - "Variable name" used to refer to data in definition
- return statement
  - Sends data back to caller

Function Call

- Just like calling predefined function
  bill = totalCost(number, price);
- Recall: totalCost returns double value
  - Assigned to variable named "bill"
- Arguments here: number, price
  - Recall arguments can be literals, variables, expressions, or combination
  - In function call, arguments often called "actual arguments"
    - Because they contain the "actual data" being sent

Alternative Function Declaration

- Recall: Function declaration is "information" for compiler
- Compiler only needs to know:
    - Return type
    - Function name
    - Parameter list
- Formal parameter names not needed:
  double totalCost(int, double);
    - Still "should" put in formal parameter names
        - Improves readability

Functions Calling Functions

- We're already doing this!
  - main() IS a function!
- Only requirement:
  - Function's declaration must appear first
- Function's definition typically elsewhere
  - After main()"s definition
  - Or in separate file
- Common for functions to call many other functions
- Function can even call itself → "Recursion"

Boolean Return-Type Functions

- Return-type can be any valid type
  - Given function declaration/prototype:
    bool appropriate(int rate);
  - And function's definition:
    ```
    bool appropriate (int rate)
    {
            return (((rate>=10)&&(rate<20))||(rate==0);
    }
    ```
  - Returns "true" or "false"
  - Function call, from some other function:
    ```
    if (appropriate(entered_rate))
        cout << "Rate is valid\n";
    ```

Declaring Void Functions

- Similar to functions returning a value
- Return type specified as "void"
- Example:
  - Function declaration/prototype:

    ```
    void showResults(    double fDegrees,
                         double cDegrees);
    ```
    - Return-type is "void"
    - Nothing is returned

Declaring Void Functions

- Function definition:
```
void showResults(double fDegrees, double cDegrees)
{
        cout.setf(ios::fixed);
        cout.setf(ios::showpoint);
        cout.precision(1);
        cout    << fDegrees
                << " degrees fahrenheit equals \n"
                << cDegrees << " degrees celsius.\n";
}
```
- Notice: no return statement
  - Optional for void functions

Calling Void Functions

- Same as calling predefined void functions
- From some other function, like main():
  - showResults(degreesF, degreesC);
  - showResults(32.5, 0.3);
- Notice no assignment, since no value returned
- Actual arguments (degreesF, degreesC)
  - Passed to function
  - Function is called to "do its job" with the data passed in

More on Return Statements

- Transfers control back to "calling" function
  - For return type other than void, MUST have return statement
  - Typically the LAST statement in function definition
- return statement optional for void functions
  - Closing } would implicitly return control from void function

Scope Rules

- Local variables
  - Declared inside body of given function
  - Available only within that function
- Can have variables with same names declared in different functions
  - Scope is local: "that function is its scope"
- Local variables preferred
  - Maintain individual control over data
  - Need to know basis
  - Functions should declare whatever local data needed to "do their job"

Global Constants and Global Variables

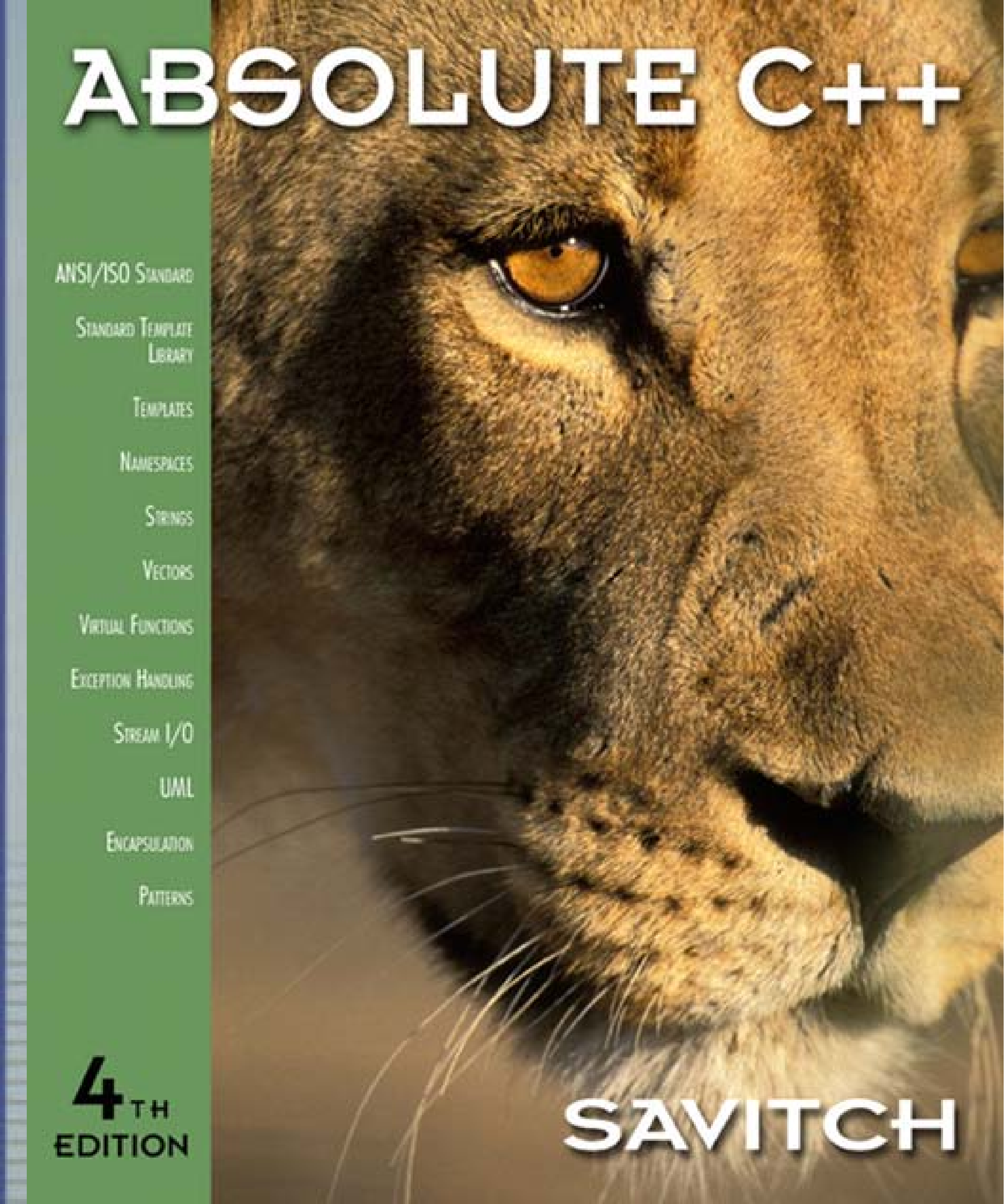
- Declared "outside" function body
  - Global to all functions in that file
- Declared "inside" function body
  - Local to that function
- Global declarations typical for constants:
  - const double TAXRATE = 0.05;
  - Declare globally so all functions have scope
- Global variables?
  - Possible, but SELDOM-USED
  - Dangerous: no control over usage!

Blocks

- Declare data inside compound statement
  - Called a "block"
  - Has "block-scope"
- Note: all function definitions are blocks!
  - This provides local "function-scope"
- Loop blocks:

```
for (int ctr=0;ctr<10;ctr++)
{
    sum+=ctr;
}
```

  - Variable ctr has scope in loop body block only



Chapter 4-4.2

Parameters and Overloading

Learning Objectives

- Parameters
  - Call-by-value
  - Call-by-reference
  - Mixed parameter-lists
- Overloading and Default Arguments
  - Examples, Rules

Parameters

- Two methods of passing arguments as parameters
- Call-by-value
  - "copy" of value is passed
- Call-by-reference
  - "address of" actual argument is passed

Call-by-Value Parameters

- Copy of actual argument passed
- Considered "local variable" inside function
- If modified, only "local copy" changes
  - Function has no access to "actual argument" from caller
- This is the default method
  - Used in all examples thus far

Call-By-Reference Parameters

- Used to provide access to caller's actual argument
- Caller's data can be modified by called function!
- Typically used for input function
  - To retrieve data for caller
  - Data is then "given" to caller
- Specified by ampersand, &, after type in formal parameter list

Call-By-Reference Example:
Display 4.1 Call-by-Reference Parameters (1 of 3)

Display 4.2 Call-by-Reference Parameters

```
//Program to demonstrate call-by-reference parameters.
#include <iostream>
using namespace std;

void getNumbers(int& input1, int& input2);
//Reads two integers from the keyboard.

void swapValues(int& variable1, int& variable2);
//Interchanges the values of variable1 and variable2.

void showResults(int output1, int output2);
//Shows the values of variable1 and variable2, in that order.

int main( )
{
    int firstNum, secondNum;

    getNumbers(firstNum, secondNum);
    swapValues(firstNum, secondNum);
    showResults(firstNum, secondNum);
    return 0;
}
```

Call-By-Reference Example:
Display 4.1 Call-by-Reference Parameters (2 of 3)

```
void getNumbers(int& input1, int& input2)
{
    cout << "Enter two integers: ";
    cin >> input1
        >> input2;
}

void swapValues(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}

void showResults(int output1, int output2)
{
    cout << "In reverse order the numbers are: "
         << output1 << " " << output2 << endl;
}
```

Call-By-Reference Details

- What's really passed in?
- A "reference" back to caller's actual argument!
  - Refers to memory location of actual argument
  - Called "address", which is a unique number referring to distinct place in memory

Constant Reference Parameters

- Reference arguments inherently "dangerous"
  - Caller’s data can be changed
  - Often this is desired, sometimes not
- To "protect" data and still pass by reference:
  - Use const keyword
    - void sendConstRef(const int &par1,
      const int &par2);
    - Makes arguments "read-only" by function
    - No changes allowed inside function body

Mixed Parameter Lists

- Can combine passing mechanisms
- Parameter lists can include pass-by-value and pass-by-reference parameters
- Order of arguments in list is critical:
  void mixedCall(int & par1, int par2, double & par3);
  – Function call:
    mixedCall(arg1, arg2, arg3);
    - arg1 must be integer type, is passed by reference
    - arg2 must be integer type, is passed by value
    - arg3 must be double type, is passed by reference

Overloading

- Same function name
- Different parameter lists
- Two separate function definitions
- Function "signature"
  - Function name & parameter list
  - Must be "unique" for each function definition
- Allows same task performed on different data

Overloading Example: Average

- Function computes average of 2 numbers:
  double average(double n1, double n2)
  ```
  {
          return ((n1 + n2) / 2.0);
  }
  ```

- Now compute average of 3 numbers:
  double average(double n1, double n2, double n3)
  ```
  {
          return ((n1 + n2) / 2.0);
  }
  ```

- Same name, two functions

Overloaded Average() Cont'd

- Which function gets called?
- Depends on function call itself:
  - avg = average(5.2, 6.7);
    - Calls "two-parameter average()"
  - avg = average(6.5, 8.5, 4.2);
    - Calls "three-parameter average()"
- Compiler resolves invocation based on signature of function call
  - "Matches" call with appropriate function
  - Each considered separate function

Overloading Resolution

- 1st: Exact Match
  – Looks for exact signature
    - Where no argument conversion required
- 2nd: Compatible Match
  – Looks for "compatible" signature where automatic type conversion is possible:
    - 1st with promotion (e.g., int→double)
      – No loss of data
    - 2nd with demotion (e.g., double→int)
      – Possible loss of data

Overloading Resolution Example

- Given following functions:
  - 1. void f(int n, double m);
    2. void f(double n, int m);
    3. void f(int n, int m);
  - These calls:
    f(98, 99); → Calls #3
    f(5.3, 4); → Calls #2
    f(4.3, 5.2); → Calls ???
- Avoid such confusing overloading

Automatic Type Conversion and Overloading

- Numeric formal parameters typically made "double" type
- Allows for "any" numeric type
  - Any "subordinate" data automatically promoted
    - int → double
    - float → double
    - char → double *More on this later!
- Avoids overloading for different numeric types

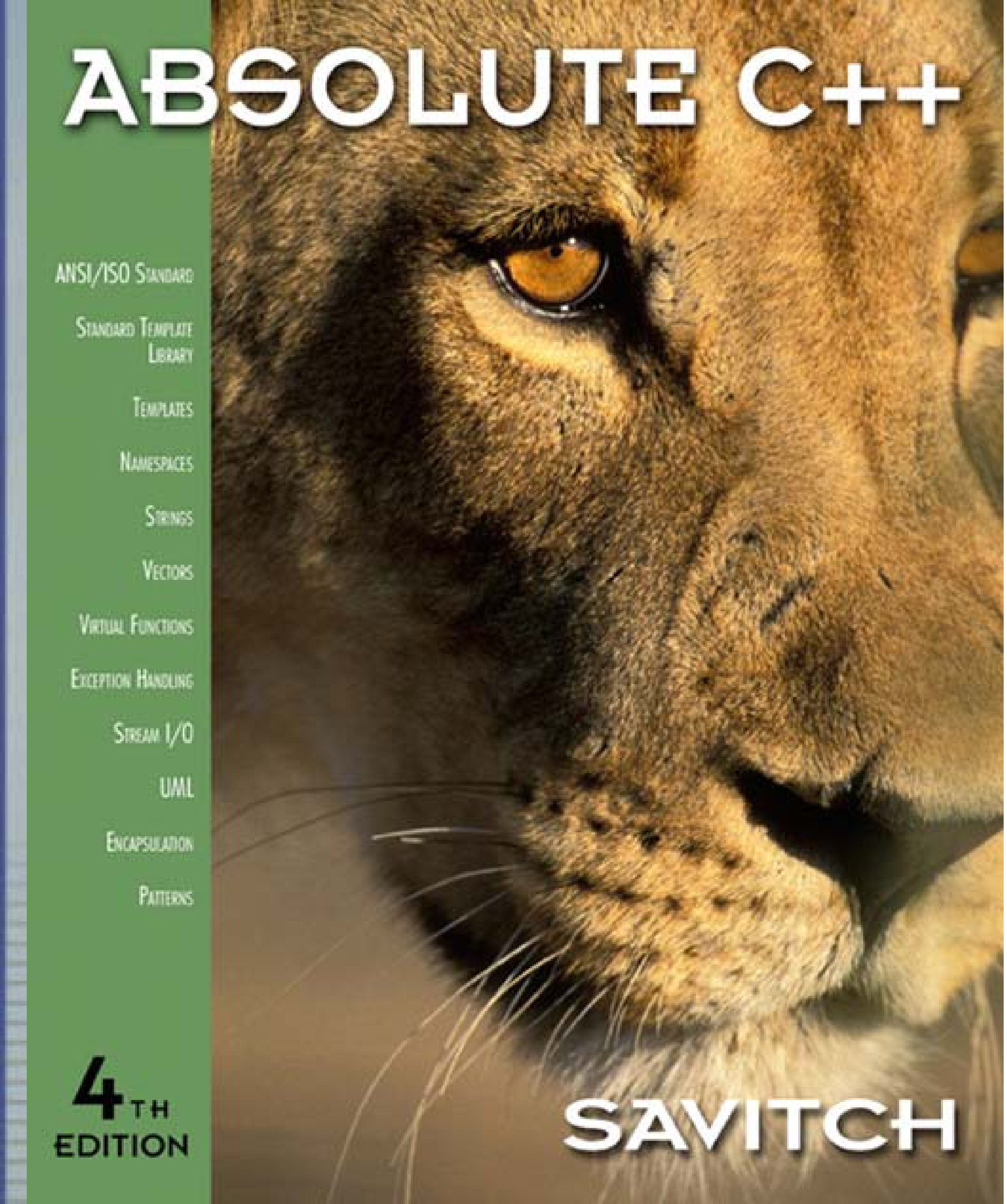
Automatic Type Conversion and Overloading Example

- double mpg(double miles, double gallons)
```
{
	return (miles/gallons);
}
```

- Example function calls:
  - mpgComputed = mpg(5, 20);
    - Converts 5 & 20 to doubles, then passes
  - mpgComputed = mpg(5.8, 20.2);
    - No conversion necessary
  - mpgComputed = mpg(5, 2.4);
    - Converts 5 to 5.0, then passes values to function

Default Arguments

- Allows omitting some arguments
- Specified in function declaration/prototype
  - void showVolume(int length,
    int width = 1,
    int height = 1);
    - Last 2 arguments are defaulted
  - Possible calls:
    - showVolume(2, 4, 6); //All arguments supplied
    - showVolume(3, 5); //height defaulted to 1
    - showVolume(7); //width & height defaulted to 1



Chapter 5 (skip 5.3), 9-9.1

Arrays & C strings

Learning Objectives

- Introduction to Arrays
  - Declaring and referencing arrays
  - For-loops and arrays
  - Arrays in memory
- Arrays in Functions
  - Arrays as function arguments, return values
- Multidimensional Arrays
- An Array Type for Strings
  - C-Strings

Introduction to Arrays

- Array definition:
  - A collection of data of same type
- First "aggregate" data type
  - Means "grouping"
  - int, float, double, char are simple data types
- Used for lists of like items
  - Test scores, temperatures, names, etc.
  - Avoids declaring multiple simple variables
  - Can manipulate "list" as one entity

Declaring Arrays

- Declare the array → allocates memory
  int score[5];
  - Declares array of 5 integers named "score"
  - Similar to declaring five variables:
    int score[0], score[1], score[2], score[3], score[4]
- Individual parts called many things:
  - Indexed or subscripted variables
  - "Elements" of the array
  - Value in brackets called index or subscript
    - Numbered from 0 to size - 1

Accessing Arrays

- Access using index/subscript
  - cout << score[3];
- Note two uses of brackets:
  - In declaration, specifies SIZE of array
  - Anywhere else, specifies a subscript
- Size, subscript need not be literal
  - int score[MAX_SCORES];
  - score[n+1] = 99;
    - If n is 2, identical to: score[3]

Array Program Example:
Display 5.1 Program Using an Array (1 of 2)

Display 5.1 Program Using an Array

```
//Reads in five scores and shows how much each
//score differs from the highest score.
#include <iostream>
using namespace std;

int main( )
{
    int i, score[5], max;

    cout << "Enter 5 scores:\n";
    cin >> score[0];
    max = score[0];
    for (i = 1; i < 5; i++)
    {
        cin >> score[i];
        if (score[i] > max)
            max = score[i];
        //max is the largest of the values score[0],..., score[i].
    }
```

Array Program Example:
Display 5.1 Program Using an Array (2 of 2)

```
    cout << "The highest score is " << max << endl
         << "The scores and their\n"
         << "differences from the highest are:\n";
    for (i = 0; i < 5; i++)
        cout << score[i] << " off by "
             << (max – score[i]) << endl;

    return 0;
}
```

Sample Dialogue

Enter 5 scores:
5 9 2 10 6
The highest score is 10
The scores and their
differences from the highest are:
5 off by 5
9 off by 1
2 off by 8
10 off by 0
6 off by 4

for-loops with Arrays

- Natural counting loop
  - Naturally works well "counting thru" elements of an array
- Example:

```
for (idx = 0; idx<5; idx++)
{
      cout << score[idx] << "off by "
               << max – score[idx] << endl;
}
```

  - Loop control variable (idx) counts from 0 – 5

Major Array Pitfall

- Array indexes always start with zero!
- Zero is "first" number to computer scientists
- C++ will "let" you go beyond range
  - Unpredictable results
  - Compiler will not detect these errors!
- Up to programmer to "stay in range"

Major Array Pitfall Example

- Indexes range from 0 to (array_size – 1)
  – Example:
  double temperature[24];   // 24 is array size
  // Declares array of 24 double values called temperature
    - They are indexed as:
      temperature[0], temperature[1] … temperature[23]

  – Common mistake:
  temperature[24] = 5;
    - Index 24 is "out of range"!
    - No warning, possibly disastrous results

Defined Constant as Array Size

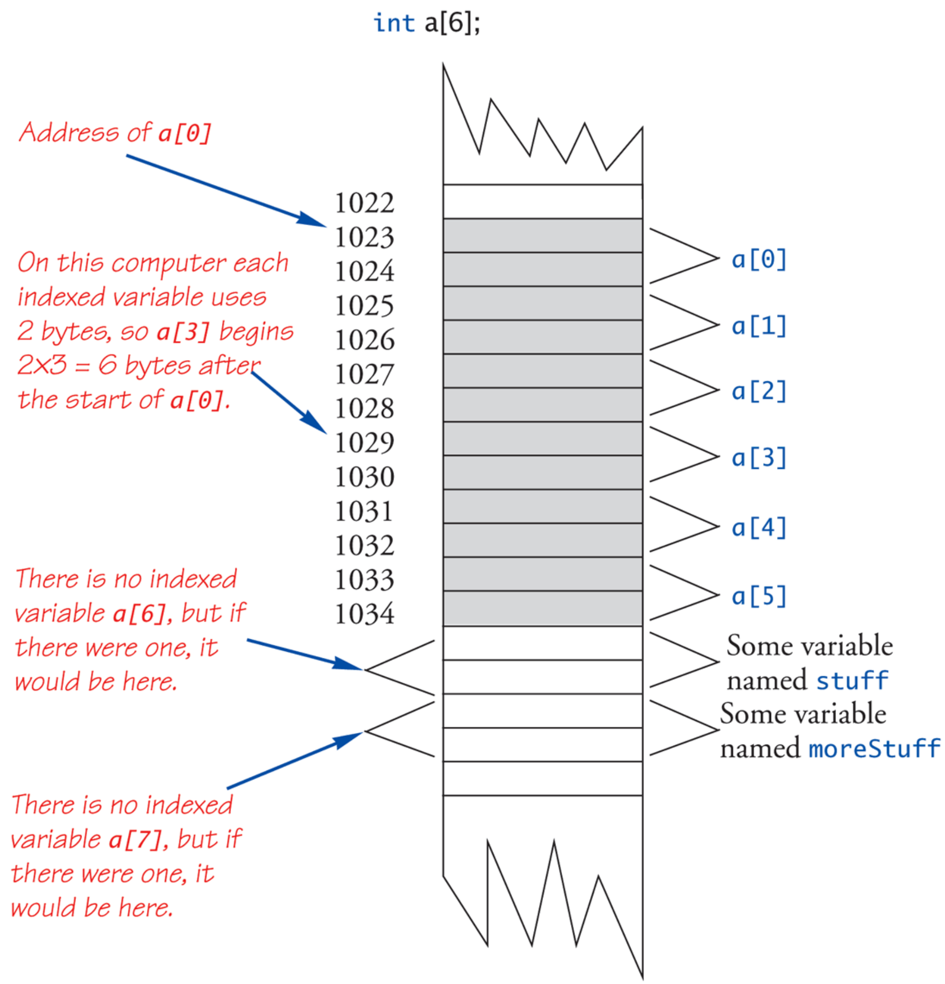
- Always use defined/named constant for array size
- Example:
  const int NUMBER_OF_STUDENTS = 5;
  int score[NUMBER_OF_STUDENTS];
- Improves readability
- Improves versatility
- Improves maintainability

Arrays in Memory

- Recall simple variables:
  - Allocated memory in an "address"
- Array declarations allocate memory for entire array
- Sequentially-allocated
  - Means addresses allocated "back-to-back"
  - Allows indexing calculations
    - Simple "addition" from array beginning (index 0)

An Array in Memory

Display 5.2 An Array in Memory



Initializing Arrays

- As simple variables can be initialized at declaration:
  int price = 0;   // 0 is initial value
- Arrays can as well:
  int children[3] = {2, 12, 1};
  - Equivalent to following:
    int children[3];
    children[0] = 2;
    children[1] = 12;
    children[2] = 1;

Auto-Initializing Arrays

- If fewer values than size supplied:
  - Fills from beginning
  - Fills "rest" with zero of array base type
- If array-size is left out
  - Declares array with size required based on number of initialization values
  - Example:
    int b[] = {5, 12, 11};
    - Allocates array b to size 3

Arrays in Functions

- As arguments to functions
  - Indexed variables
    - An individual "element" of an array can be function parameter
  - Entire arrays
    - All array elements can be passed as "one entity"
- As return value from function
  - Can be done → chapter 10

Indexed Variables as Arguments

- Indexed variable handled same as simple variable of array base type
- Given this function declaration:
  void myFunction(double par1);
- And these declarations:
  int i; double n, a[10];
- Can make these function calls:
  myFunction(i);   // i is converted to double
  myFunction(a[3]);   // a[3] is double
  myFunction(n); // n is double

Entire Arrays as Arguments

- Formal parameter can be entire array
  - Argument then passed in function call is array name
  - Called "array parameter"
- Send size of array as well
  - Typically done as second parameter
  - Simple int type formal parameter

Entire Array as Argument Example: **Display 5.3** Function with an Array Parameter

Display 5.3 Function with an Array Parameter

SAMPLE DIALOGUEFUNCTION DECLARATION

```
void fillUp(int a[], int size);
//Precondition: size is the declared size of the array a.
//The user will type in size integers.
//Postcondition: The array a is filled with size integers
//from the keyboard.
```

SAMPLE DIALOGUEFUNCTION DEFINITION

```
void fillUp(int a[], int size)
{
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> a[i];
    cout << "The last array index used is " << (size – 1) << endl;
}
```

Entire Array as Argument Example

- Given previous example:
- In some main() function definition, consider this calls:

```
int score[5], numberOfScores = 5;
fillup(score, numberOfScores);
```

  - 1st argument is entire array
  - 2nd argument is integer value
- – Note no brackets in array argument!

Array as Argument: How?

- What's really passed?
- Think of array as 3 "pieces"
  - Address of first indexed variable (arrName[0])
  - Array base type
  - Size of array
- Only 1st piece is passed!
  - Just the beginning address of array
  - Very similar to "pass-by-reference"

Array Parameters

- May seem strange
  - No brackets in array argument
  - Must send size separately
- One nice property:
  - Can use SAME function to fill any size array!
  - Exemplifies "re-use" properties of functions
  - Example:
    ```
    int score[5], time[10];
    fillUp(score, 5);
    fillUp(time, 10);
    ```

The const Parameter Modifier

- Recall: array parameter actually passes address of 1st element
  - Similar to pass-by-reference
- Function can then modify array!
  - Often desirable, sometimes not!
- Protect array contents from modification
  - Use "const" modifier before array parameter
    - Called "constant array parameter"
    - Tells compiler to "not allow" modifications

Functions that Return an Array

- Functions cannot return arrays same way simple types are returned
- Requires use of a "pointer"
- Will be discussed in chapter 10…

Multidimensional Arrays

- Arrays with more than one index
  - char page[30][100];
    - Two indexes: An "array of arrays"
    - Visualize as [row][col]:
      page[0][0], page[0][1], …, page[0][99]
      page[1][0], page[1][1], …, page[1][99]
      …
      page[29][0], page[29][1], …, page[29][99]
- C++ allows any number of indexes
  - Typically no more than two

Multidimensional Array Parameters

- Similar to one-dimensional array
  - 1st dimension size not given
    - Provided as second parameter
  - 2nd dimension size IS given
- Example:

```
void DisplayPage(const char p[][100], int sizeDimension1)
{
    for (int index1=0; index1<sizeDimension1; index1++)
    {
        for (int index2=0; index2 < 100; index2++)
            cout << p[index1][index2];
        cout << endl;
    }
}
```

C-Strings

- Array with base type *char*
  - One character per indexed variable
  - One extra character: "\0"
    - Called "null character"
    - End marker
- We’ve used c-strings
  - Literal "Hello" stored as c-string

C-String Variable

- Array of characters:
  char s[10];
  - Declares a c-string variable to hold up to 9 characters
  - + one null character
- Typically "partially-filled" array
  - Declare large enough to hold max-size string
  - Indicate end with null
- Only difference from standard array:
  - Must contain null character

C-String Storage

- A standard array:
  char s[10];
  - If s contains string "Hi Mom", stored as:

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]
H	i		M	o	m	!	\0	?	?

C-String Initialization

- Can initialize c-string:
  char myMessage[20] = "Hi there.";
  - Needn't fill entire array
  - Initialization places "\0" at end
- Can omit array-size:
  char shortString[] = "abc";
  - Automatically makes size one more than length of quoted string
  - NOT same as:
    char shortString[] = {"a", "b", "c"};

C-String Index Manipulation

- Can manipulate indexed variables
  char happyString[7] = "DoBeDo";
  happyString[6] = "Z";
  - Be careful!
  - Here, "\0" (null) was overwritten by a "Z"!
- If null overwritten, c-string no longer "acts" like c-string!
  - Unpredictable results!

= and == with C-strings

- C-strings not like other variables
  - Cannot assign or compare:
    ```
    char aString[10];
    aString = "Hello";          // ILLEGAL!
    ```
    - Can ONLY use "=" at declaration of c-string!
- Must use library function for assignment:
  ```
  strcpy(aString, "Hello");
  ```
  - Built-in function (in <cstring>)
  - Sets value of aString equal to "Hello"
  - NO checks for size!
    - Up to programmer, just like other arrays!

Comparing C-strings

- Also cannot use operator ==

```
char aString[10] = "Hello";
char anotherString[10] = "Goodbye";
```

  – aString == anotherString;   // NOT allowed!

- Must use library function again:

```
if (strcmp(aString, anotherString))
      cout << "Strings NOT same.";
else
      cout << "Strings are same.";
```

The <cstring> Library: **Display 9.1** Some Predefined C-String Functions in <cstring> (1 of 2)

- Full of string manipulation functions

Display 9.1 Some Predefined C-String Functions in `<cstring>`

FUNCTION	DESCRIPTION	CAUTIONS
`strcpy(`*Target_String_Var*`,` *Src_String*`)`	Copies the C-string value *Src_String* into the C-string variable *Target_String_Var*.	Does not check to make sure *Target_String_Var* is large enough to hold the value *Src_String*.
`strcpy(`*Target_String_Var*`,` *Src_String, Limit*`)`	The same as the two-argument `strcpy` except that at most *Limit* characters are copied.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcpy`. Not implemented in all versions of C++.
`strcat(`*Target_String_Var*`,` *Src_String*`)`	Concatenates the C-string value *Src_String* onto the end of the C-string in the C-string variable *Target_String_Var*.	Does not check to see that *Target_String_Var* is large enough to hold the result of the concatenation.

(continued)

The <cstring> Library: **Display 9.1** Some Predefined C-String Functions in <cstring> (2 of 2)

Display 9.1 Some Predefined C-String Functions in <cstring>

FUNCTION	DESCRIPTION	CAUTIONS
`strcat(`*Target_String_Var*`,` *Src_String, Limit*`)`	The same as the two argument `strcat` except that at most *Limit* characters are appended.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcat`. Not implemented in all versions of C++.
`strlen(`*Src_String*`)`	Returns an integer equal to the length of *Src_String*. (The null character, `'\0'`, is not counted in the length.)	
`strcmp(`*String_1*`,`*String_2*`)`	Returns `0` if *String_1* and *String_2* are the same. Returns a value < 0 if *String_1* is less than *String_2*. Returns a value > 0 if *String_1* is greater than *String_2* (that is, returns a nonzero value if *String_1* and *String_2* are different). The order is lexicographic.	If *String_1* equals *String_2*, this function returns `0`, which converts to `false`. Note that this is the reverse of what you might expect it to return when the strings are equal.
`strcmp(`*String_1*`,` *String_2, Limit*`)`	The same as the two-argument `strcat` except that at most *Limit* characters are compared.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcmp`. Not implemented in all versions of C++.

C-string Arguments and Parameters

- Recall: c-string is array
- So c-string parameter is array parameter
  - C-strings passed to functions can be changed by receiving function!
- Like all arrays, typical to send size as well
  - Function "could" also use "\0" to find end
  - So size not necessary if function won't change c-string parameter
  - Use "const" modifier to protect c-string arguments

C-String Output

- Can output with insertion operator, <<
- As we've been doing already:
  cout << news << " Wow.\n";
  - Where *news* is a c-string variable
- Possible because << operator is overloaded for c-strings!

C-String Input

- Can input with extraction operator, >>
  - Issues exist, however
- Whitespace is "delimiter"
  - Tab, space, line breaks are "skipped"
  - Input reading "stops" at delimiter
- Watch size of c-string
    - Must be large enough to hold entered string!
    - C++ gives no warnings of such issues!

C-String Input Example

- ```
  char a[80], b[80];
  cout << "Enter input: ";
  cin >> a >> b;
  cout << a << b << "END OF OUTPUT\n";
  ```
- Dialogue offered:

  Enter input: <u>Do be do to you!</u>
  DobeEND OF OUTPUT

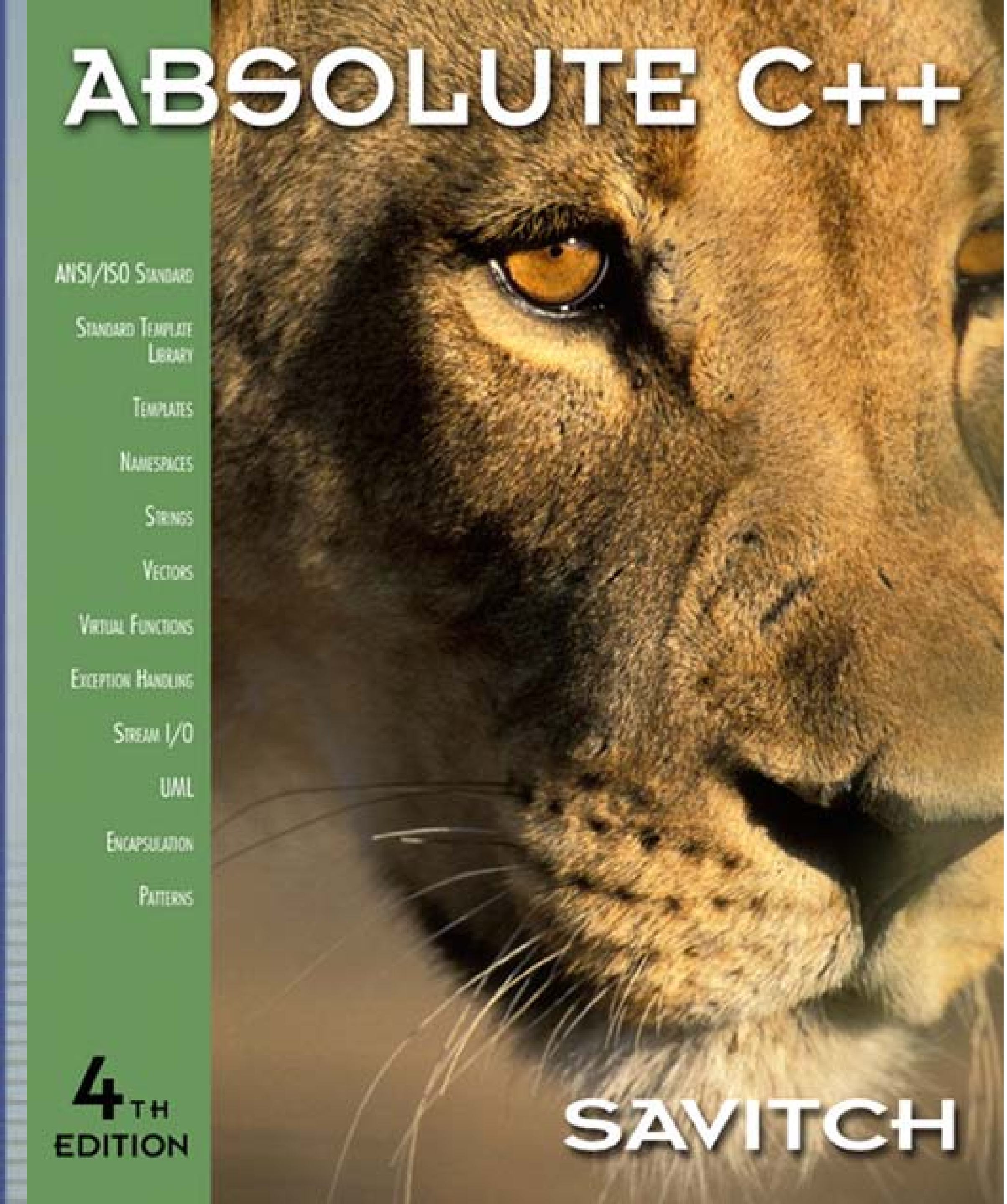
  – Note: Underlined portion typed at keyboard
- C-string *a* receives: "do"
- C-string *b* receives: "be"
  ```

Example: Command Line Arguments

- Programs invoked from the command line (e.g. a UNIX shell, DOS command prompt) can be sent arguments
  - Example: COPY C:\FOO.TXT D:\FOO2.TXT
    - This runs the program named "COPY" and sends in two C-String parameters, "C:\FOO.TXT" and "D:\FOO2.TXT"
    - It is up to the COPY program to process the inputs presented to it; i.e. actually copy the files
- Arguments are passed as an array of C-Strings to the main function

Example: Command Line Arguments

- Header for main
  - int main(int argc, char *argv[])
  - argc specifies how many arguments are supplied. The name of the program counts, so argc will be at least 1.
  - argv is an array of C-Strings.
    - argv[0] holds the name of the program that is invoked
    - argv[1] holds the name of the first parameter
    - argv[2] holds the name of the second parameter
    - Etc.



Chapter 6-6.1, 10-10.2

Structures, pointers, & dynamic allocation

PEARSON Addison Wesley

Learning Objectives

- Structures
  - Structure types
  - Structures as function arguments
  - Initializing structures
- Pointers
  - Pointer variables
  - Memory management
- Dynamic Arrays
  - Creating and using
  - Pointer arithmetic

Structures

- 2^{nd} aggregate data type: struct
- Recall: aggregate meaning "grouping"
  – Recall array: collection of values of same type
  – Structure: collection of values of different types
- Treated as a single item, like arrays
- Major difference: Must first "define" struct
  – Prior to declaring any variables

Structure Types

- Define struct globally (typically)
- No memory is allocated
  - Just a "placeholder" for what our struct will "look like"
- Definition:

```
struct CDAccountV1   ←Name of new struct "type"
{
        double balance;        ← member names
        double interestRate;
        int term;
};   // ← REQUIRED semicolon!
```

Declare Structure Variable

- With structure type defined, now declare variables of this new type:
  CDAccountV1 account;
  - Just like declaring simple types
  - Variable *account* now of type CDAccountV1
  - It contains "member values"
    - Each of the struct "parts"

Accessing Structure Members

- Dot Operator to access members
  - account.balance
  - account.interestRate
  - account.term
- Called "member variables"
  - The "parts" of the structure variable
  - Different structs can have same name member variables
    - No conflicts

Structure Assignments

- Given structure named CropYield
- Declare two structure variables:
  CropYield apples, oranges;
  - Both are variables of "struct type CropYield"
  - Simple assignments are legal:
    apples = oranges;
    - Simply copies each member variable from apples into member variables from oranges

Structures as Function Arguments

- Passed like any simple data type
  - Pass-by-value
  - Pass-by-reference
  - Or combination
- Can also be returned by function
  - Return-type is structure type
  - Return statement in function definition sends structure variable back to caller

Initializing Structures

- Can initialize at declaration
  - Example:

```
struct Date
{
        int month;
        int day;
        int year;
};
Date dueDate = {12, 31, 2003};
```

  - Declaration provides initial data to all three member variables

Pointer Introduction

- Pointer definition:
  – Memory address of a variable
- Recall: memory divided
  – Numbered memory locations
  – Addresses used as name for variable
- You've used pointers already!
  – Call-by-reference parameters
    - Address of actual argument was passed

Pointer Variables

- Pointers are "typed"
  - Can store pointer in variable
  - Not int, double, etc.
    - Instead: A POINTER to int, double, etc.!
- Example:
  double *p;
  - p is declared a "pointer to double" variable
  - Can hold pointers to variables of type double
    - Not other types!

Declaring Pointer Variables

- Pointers declared like other types
  - Add "*" before variable name
  - Produces "pointer to" that type
- "*" must be before each variable
- int *p1, *p2, v1, v2;
  - p1, p2 hold pointers to int variables
  - v1, v2 are ordinary int variables

Addresses and Numbers

- Pointer is an address
- Address is an integer
- Pointer is NOT an integer!
  - Not crazy → abstraction!
- C++ forces pointers be used as addresses
  - Cannot be used as numbers
  - Even though it "is a" number

Pointing to …

- int *p1, *p2, v1, v2;
  p1 = &v1;
  - Sets pointer variable p1 to "point to" int variable v1
- Operator, &
  - Determines "address of" variable
- Read like:
  - "p1 equals address of v1"
  - Or "p1 points to v1"

Pointing to …

- Recall:
  int *p1, *p2, v1, v2;
  p1 = &v1;
- Two ways to refer to v1 now:
  - Variable v1 itself:
    cout << v1;
  - Via pointer p1:
    cout *p1;
- Dereference operator, *
  - Pointer variable "derereferenced"
  - Means: "Get data that p1 points to"

"Pointing to" Example

- Consider:
```
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```
- Produces output:
  42
  42
- p1 and v1 refer to same variable

& Operator

- The "address of" operator
- Also used to specify call-by-reference parameter
  - No coincidence!
  - Recall: call-by-reference parameters pass "address of" the actual argument
- Operator's two uses are closely related

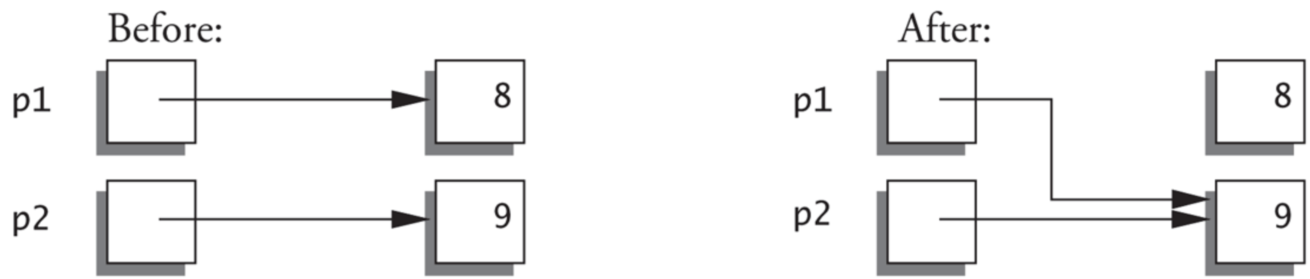
Pointer Assignments

- Pointer variables can be "assigned":
  int *p1, *p2;
  p2 = p1;
  - Assigns one pointer to another
  - "Make p2 point to where p1 points"
- Do not confuse with:
  *p1 = *p2;
  - Assigns "value pointed to" by p1, to "value pointed to" by p2

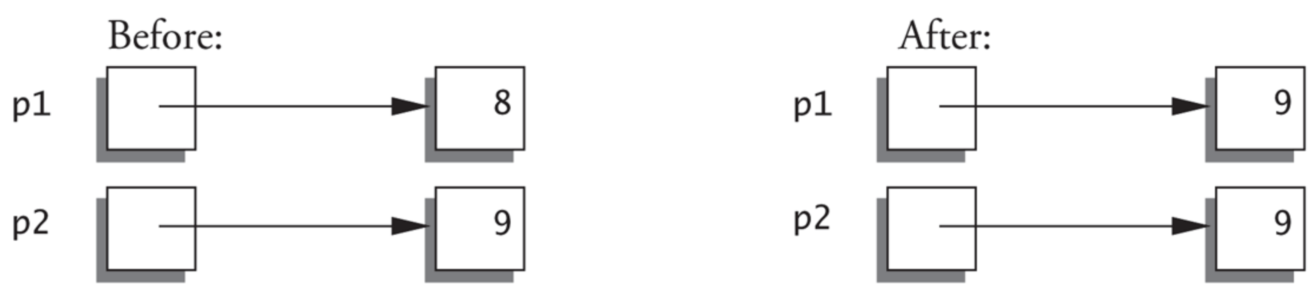
Pointer Assignments Graphic: **Display 10.1** Uses of the Assignment Operator with Pointer Variables

Display 10.1 Uses of the Assignment Operator with Pointer Variables

```
p1 = p2;
```




```
*p1 = *p2;
```



The new Operator

- Since pointers can refer to variables…
  - No "real" need to have a standard identifier
- Can dynamically allocate variables
  - Operator *new* creates variables
    - No identifiers to refer to them
    - Just a pointer!
- p1 = new int;
  - Creates new "nameless" variable, and assigns p1 to "point to" it
  - Can access with *p1
    - Use just like ordinary variable

Basic Pointer Manipulations Example: **Display 10.2** Basic Pointer Manipulations (1 of 2)

Display 10.2 Basic Pointer Manipulations

```
//Program to demonstrate pointers and dynamic variables.
#include <iostream>
using std::cout;
using std::endl;

int main( )
{
    int *p1, *p2;

    p1 = new int;
    *p1 = 42;
    p2 = p1;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    *p2 = 53;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;
```

Pointers and Functions

- Pointers are full-fledged types
  - Can be used just like other types
- Can be function parameters
- Can be returned from functions
- Example:
  int* findOtherPointer(int* p);
  - This function declaration:
    - Has "pointer to an int" parameter
    - Returns "pointer to an int" variable

Memory Management

- Heap
  - Also called "freestore"
  - Reserved for dynamically-allocated variables
  - All new dynamic variables consume memory in freestore
    - If too many → could use all freestore memory
- Future "new" operations will fail if freestore is "full"

Checking new Success

- Older compilers:
  - Test if null returned by call to *new:*

```
int *p;
p = new int;
if (p == NULL)
{
    cout << "Error: Insufficient memory.\n";
    exit(1);
}
```

  - If new succeeded, program continues

new Success – New Compiler

- Newer compilers:
  - If new operation fails:
    - Program terminates automatically
    - Produces error message
- Still good practice to use NULL check

delete Operator

- De-allocate dynamic memory
  - When no longer needed
  - Returns memory to freestore
  - Example:
```
int *p;
p = new int(5);
… //Some processing…
delete p;
```
  - De-allocates dynamic memory "pointed to by pointer p"
    - Literally "destroys" memory

Dangling Pointers

- delete p;
  - Destroys dynamic memory
  - But p still points there!
    - Called "dangling pointer"
  - If p is then dereferenced (*p)
    - Unpredicatable results!
    - Often disastrous!
- Avoid dangling pointers
  - Assign pointer to NULL after delete:

    delete p;
    p = NULL;

Dynamic and Automatic Variables

- Dynamic variables
  - Created with new operator
  - Created and destroyed while program runs
- Local variables
  - Declared within function definition
  - Not dynamic
    - Created when function is called
    - Destroyed when function call completes
  - Often called "automatic" variables
    - Properties controlled for you

Dynamic Arrays

- Array variables
  - Really pointer variables!
- Standard array
  - Fixed size
- Dynamic array
  - Size not specified at programming time
  - Determined while program running

Array Variables

- Recall: arrays stored in memory addresses, sequentially
  - Array variable "refers to" first indexed variable
  - So array variable is a kind of pointer variable!
- Example:
  ```
  int a[10];
  int * p;
  ```
  - a and p are both pointer variables!

Array Variables → Pointers

- Recall previous example:
  ```
  int a[10];
  typedef int* IntPtr;
  IntPtr p;
  ```
- a and p are pointer variables
  - Can perform assignments:
    `p = a;    // Legal.`
    - p now points where a points
      - To first indexed variable of array a
  - `a = p;    // ILLEGAL!`
    - Array pointer is CONSTANT pointer!

Array Variables → Pointers

- Array variable
  int a[10];
- MORE than a pointer variable
  - "const int *" type
  - Array was allocated in memory already
  - Variable *a* MUST point there…always!
    - Cannot be changed!
- In contrast to ordinary pointers
  - Which can (& typically do) change

Dynamic Arrays

- Array limitations
  - Must specify size first
  - May not know until program runs!
- Must "estimate" maximum size needed
  - Sometimes OK, sometimes not
  - "Wastes" memory
- Dynamic arrays
  - Can grow and shrink as needed

Creating Dynamic Arrays

- Very simple!
- Use new operator
  - Dynamically allocate with pointer variable
  - Treat like standard arrays
- Example:

```
typedef double * DoublePtr;
DoublePtr d;
d = new double[10];    //Size in brackets
```

  - Creates dynamically allocated array variable *d*, with ten elements, base type double

Deleting Dynamic Arrays

- Allocated dynamically at run-time
  – So should be destroyed at run-time
- Simple again. Recall Example:
  d = new double[10];
  … //Processing
  delete [] d;
  – De-allocates all memory for dynamic array
  – Brackets indicate "array" is there
  – Recall: *d* still points there!
    - Should set d = NULL;

Function that Returns an Array

- Array type NOT allowed as return-type of function
- Example:
  int [] someFunction(); // ILLEGAL!
- Instead return pointer to array base type:
  int* someFunction(); // LEGAL!

Pointer Arithmetic

- Can perform arithmetic on pointers
  - "Address" arithmetic
- Example:
  ```
  typedef double* DoublePtr;
  DoublePtr d;
  d = new double[10];
  ```
  - d contains address of d[0]
  - d + 1 evaluates to address of d[1]
  - d + 2 evaluates to address of d[2]
    - Equates to "address" at these locations

Alternative Array Manipulation

- Use pointer arithmetic!
- "Step thru" array without indexing:

```
for (int i = 0; i < arraySize; i++)
        cout << *(d + I) << " " ;
```

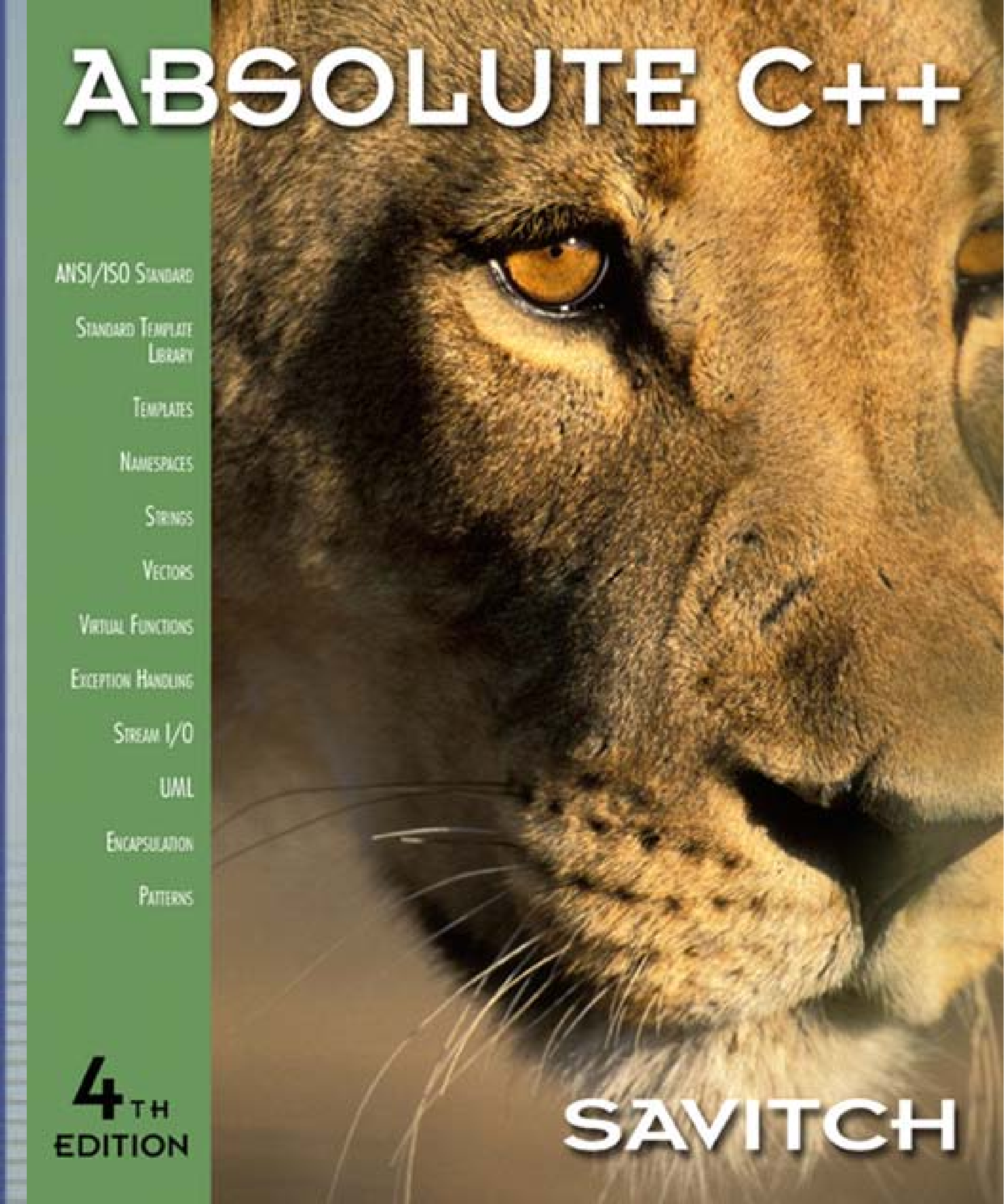
- Equivalent to:

```
for (int i = 0; i < arraySize; i++)
        cout << d[I] << " " ;
```

- Only addition/subtraction on pointers
  - No multiplication, division
- Can use ++ and -- on pointers

Multidimensional Dynamic Arrays

- Yes we can!
- Recall: "arrays of arrays"
- Type definitions help "see it":
  typedef int* IntArrayPtr;
  IntArrayPtr *m = new IntArrayPtr[3];
  - Creates array of three pointers
  - Make each allocate array of 4 ints
- for (int i = 0; i < 3; i++)
  m[i] = new int[4];
  - Results in three-by-four dynamic array!



Chapter 9.2, 12-12.2

File I/O

Learning Objectives

- Character Manipulation Tools
  - Character I/O
  - get, put member functions
- I/O Streams
  - File I/O
  - Character I/O
- Tools for Stream I/O
  - File names as input

C-String Line Input

- Can receive entire line into c-string
- Use getline(), a predefined member function:

```
char a[80];
cout << "Enter input: ";
cin.getline(a, 80);
cout << a << "END OF OUTPUT\n";
```

  – Dialogue:

    Enter input: <u>Do be do to you!</u>
    Do be do to you!END OF INPUT

More getline()

- Can explicitly tell length to receive:
```
char shortString[5];
cout << "Enter input: ";
cin.getline(shortString, 5);
cout << shortString << "END OF OUTPUT\n";
```
  - Results:
    Enter input: <u>dobedowap</u>
    dobeEND OF OUTPUT
  - Forces FOUR characters only be read
    - Recall need for null character!

Character I/O

- Input and output data
  - ALL treated as character data
  - e.g., number 10 outputted as "1" and "0"
  - Conversion done automatically
    - Uses low-level utilities
- Can use same low-level utilities ourselves as well

Member Function get()

- Reads one char at a time
- Member function of cin object:
  char nextSymbol;
  cin.get(nextSymbol);
  - Reads next char & puts in variable nextSymbol
  - Argument must be char type
    - Not "string"!

Member Function put()

- Outputs one character at a time
- Member function of cout object:
- Examples:
  cout.put("a");
  - Outputs letter "a" to screen

  char myString[10] = "Hello";
  cout.put(myString[1]);
  - Outputs letter "e" to screen

Character-Manipulating Functions: **Display 9.3** Some Functions in <cctype> (1 of 3)

Display 9.3 Some Functions in <cctype>

FUNCTION	DESCRIPTION	EXAMPLE

`toupper(`*Char_Exp*`)`	Returns the uppercase version of *Char_Exp* (as a value of type `int`).	`char c = toupper('a');`
		`cout << c;`
		Outputs: A

`tolower(`*Char_Exp*`)`	Returns the lowercase version of *Char_Exp* (as a value of type `int`).	`char c = tolower('A');`
		`cout << c;`
		Outputs: a

`isupper(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is an uppercase letter; otherwise, returns `false`.	`if (isupper(c))`
		`    cout << "Is uppercase.";`
		`else`
		`    cout << "Is not uppercase.";`

Character-Manipulating Functions: **Display 9.3** Some Functions in <cctype> (2 of 3)

Display 9.3 Some Functions in <cctype>

FUNCTION	DESCRIPTION	EXAMPLE

`islower(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a lowercase letter; otherwise, returns `false`.	`char c = 'a';`
		`if (islower(c))`
		`    cout << c << " is lowercase.";`
		Outputs: `a is lowercase.`

`isalpha(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a letter of the alphabet; otherwise, returns `false`.	`char c = '$';`
		`if (isalpha(c))`
		`    cout << "Is a letter.";`
		`else`
		`    cout << "Is not a letter.";`
		Outputs: `Is not a letter.`

`isdigit(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is one of the digits `'0'` through `'9'`; otherwise, returns `false`.	`if (isdigit('3'))`
		`    cout << "It's a digit.";`
		`else`
		`    cout << "It's not a digit.";`
		Outputs: `It's a digit.`

`isalnum(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is either a letter or a digit; otherwise, returns `false`.	`if (isalnum('3') && isalnum('a'))`
		`    cout << "Both alphanumeric.";`
		`else`
		`    cout << "One or more are not.";`
		Outputs: `Both alphanumeric.`

Character-Manipulating Functions: **Display 9.3** Some Functions in <cctype> (3 of 3)

`isspace(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a whitespace character, such as the blank or newline character; otherwise, returns `false`.	`//Skips over one "word" and sets c`
		`//equal to the first whitespace`
		`//character after the "word":`
		`do`
		`{`
		`    cin.get(c);`
		`} while (! isspace(c));`

`ispunct(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a printing character other than whitespace, a digit, or a letter; otherwise, returns `false`.	`if (ispunct('?'))`
		`    cout << "Is punctuation.";`
		`else`
		`    cout << "Not punctuation.";`

`isprint(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a printing character; otherwise, returns `false`.	
`isgraph(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a printing character other than whitespace; otherwise, returns `false`.	
`isctrl(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a control character; otherwise, returns `false`.	

Streams

- A flow of characters
- Input stream
  - Flow into program
    - Can come from keyboard
    - Can come from file
- Output stream
  - Flow out of program
    - Can go to screen
    - Can go to file

Streams Usage

- We've used streams already
  - cin
    - Input stream object connected to keyboard
  - cout
    - Output stream object connected to screen
- Can define other streams
  - To or from files
  - Used similarly as cin, cout

Streams Usage Like cin, cout

- Consider:
  - Given program defines stream inStream that comes from some file:

    int theNumber;
    inStream >> theNumber;

    - Reads value from stream, assigned to *theNumber*
  - Program defines stream outStream that goes to some file

    outStream << "theNumber is " << theNumber;

    - Writes value to stream, which goes to file

Files

- We'll use text files
- Reading from file
  - When program takes input
- Writing to file
  - When program sends output
- Start at beginning of file to end
  - Other methods available
  - We'll discuss this simple text file access here

File Connection

- Must first connect *file* to *stream object*
- For input:
  - File → ifstream object
- For output:
  - File → ofstream object
- Classes ifstream and ofstream
  - Defined in library <fstream>
  - Named in std namespace

File I/O Libraries

- To allow both file input and output in your program:

```
#include <fstream>
using namespace std;
              OR
#include <fstream>
using std::ifstream;
using std::ofstream;
```

Declaring Streams

- Stream must be declared like any other class variable:

  ```
  ifstream inStream;
  ofstream outStream;
  ```

- Must then "connect" to file:

  ```
  inStream.open("infile.txt");
  ```

  – Called "opening the file"
  – Uses member function *open*
  – Can specify complete pathname

Streams Usage

- Once declared → use normally!

```
int oneNumber, anotherNumber;
inStream >> oneNumber >> anotherNumber;
```

- Output stream similar:

```
ofstream outStream;
outStream.open("outfile.txt");
outStream       << "oneNumber = " << oneNumber
                << " anotherNumber = "
                << anotherNumber;
```

  – Sends items to output file

File Names

- Programs and files
- Files have two names to our programs
  - External file name
    - Also called "physical file name"
    - Like "infile.txt"
    - Sometimes considered "real file name"
    - Used only once in program (to open)
  - Stream name
    - Also called "logical file name"
    - Program uses this name for all file activity

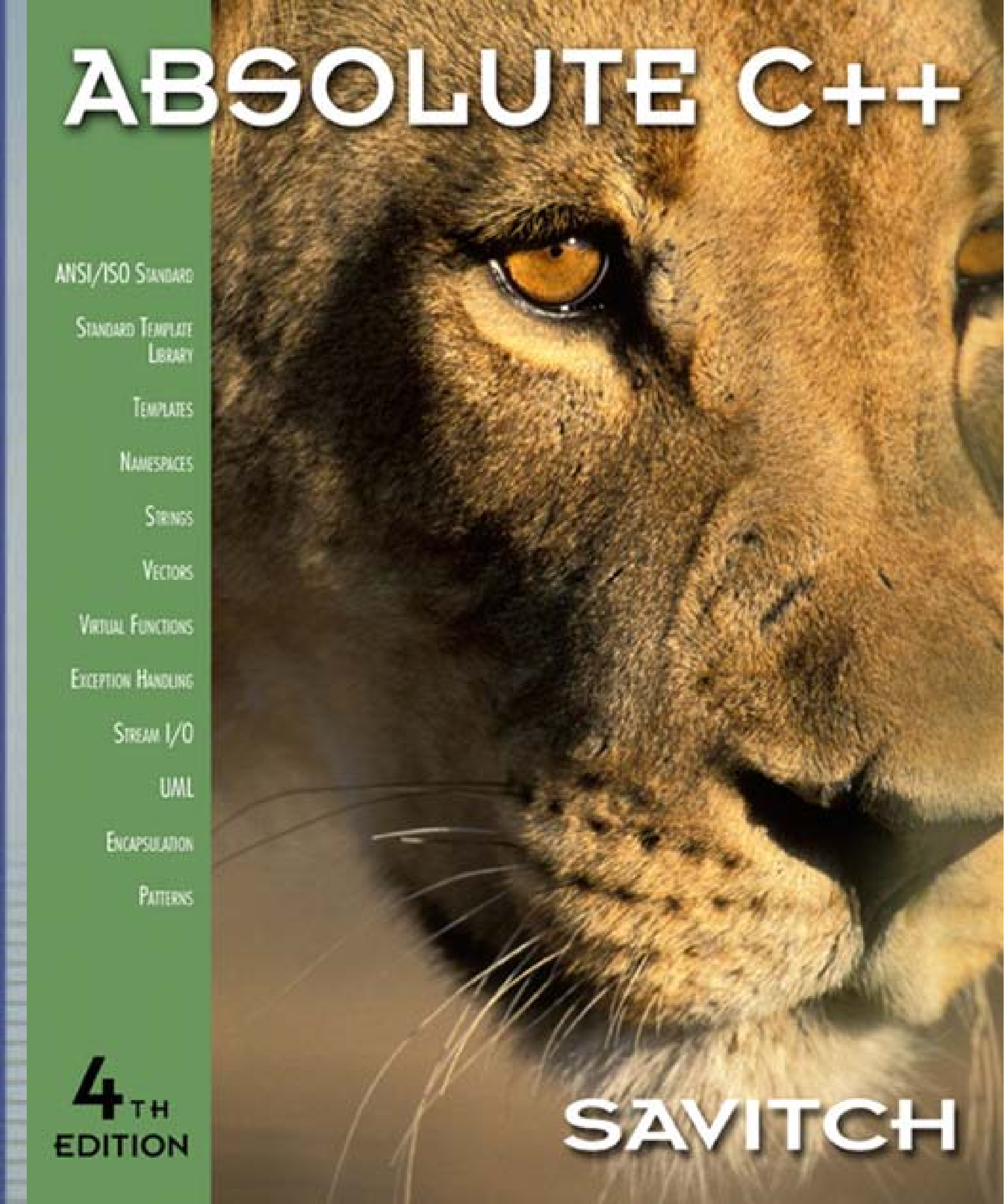
Closing Files

- Files should be closed
  - When program completed getting input or sending output
  - Disconnects stream from file
  - In action:

    ```
    inStream.close();
    outStream.close();
    ```
    - Note no arguments
- Files automatically close when program ends

File Flush

- Output often "buffered"
  - Temporarily stored before written to file
  - Written in "groups"
- Occasionally might need to force writing:
  outStream.flush();
  - Member function *flush*, for all output streams
  - All buffered output is physically written
- Closing file automatically calls flush()



Chapter 6.2, 7-7.2

Classes (definition, members, constructors)

Learning Objectives

- Classes
  - Defining, member functions
  - Public and private members
  - Accessor and mutator functions
  - Structures vs. classes
- Constructors
  - Definitions
  - Calling
- More Tools
  - const parameter modifier
  - Inline functions

Classes

- Similar to structures
  - Adds member FUNCTIONS
  - Not just member data
- Integral to object-oriented programming
  - Focus on objects
    - Object: Contains data and operations
    - In C++, variables of class type are objects

Class Definitions

- Defined similar to structures
- Example:

```
class DayOfYear    ← name of new class type
{
public:
        void output();    ← member function!
        int month;
        int day;
};
```

- Notice only member function's prototype
  - Function's implementation is elsewhere

Declaring Objects

- Declared same as all variables
  - Predefined types, structure types
- Example:

  DayOfYear today, birthday;

    - Declares two objects of class type DayOfYear
- Objects include:
  - Data
    - Members month, day
  - Operations (member functions)
    - output()

Class Member Access

- Members accessed same as structures
- Example:

  today.month
  today.day

  – And to access member function:
    today.output(); ← Invokes member function

Class Member Functions

- Must define or "implement" class member functions
- Like other function definitions
  - Can be after main() definition
  - Must specify class:
    void DayOfYear::output()
    {…}
    - :: is **scope resolution operator**
    - Instructs compiler "what class" member is from
    - Item before :: called type qualifier

Class Member Functions Definition

- Notice output() member function’s definition (in next example)
- Refers to member data of class
  - No qualifiers
- Function used for all objects of the class
  - Will refer to "that object’s" data when invoked
  - Example:
    today.output();
    - Displays "today" object’s data

Dot and Scope Resolution Operator

- Used to specify "of what thing" they are members
- Dot operator:
  - Specifies member of particular object
- Scope resolution operator:
  - Specifies what class the function definition comes from

A Class's Place

- Class is full-fledged type!
  – Just like data types int, double, etc.
- Can have variables of a class type
  – We simply call them "objects"
- Can have parameters of a class type
  – Pass-by-value
  – Pass-by-reference
- Can use class type like any other type!

Principles of OOP

- Information Hiding
  - Details of how operations work not known to "user" of class
- Data Abstraction
  - Details of how data is manipulated within class not known to user
- Encapsulation
  - Bring together data and operations, but keep "details" hidden

Public and Private Members

- Data in class almost always designated private in definition!
  - Upholds principles of OOP
  - Hide data from user
  - Allow manipulation only via operations
    - Which are member functions
- Public items (usually member functions) are "user-accessible"

Public and Private Example

- Modify previous example:

```
class DayOfYear
{
public:
        void input();
        void output();
private:
        int month;
        int day;
};
```

- Data now private
- Objects have no direct access

Public and Private Style

- Can mix & match public & private
- More typically place public first
  - Allows easy viewing of portions that can be USED by programmers using the class
  - Private data is "hidden", so irrelevant to users
- Outside of class definition, cannot change (or even access) private data

Accessor and Mutator Functions

- Object needs to "do something" with its data
- Call accessor member functions
  - Allow object to read data
  - Also called "get member functions"
  - Simple retrieval of member data
- Mutator member functions
  - Allow object to change data
  - Manipulated based on application

Separate Interface and Implementation

- User of class need not see details of how class is implemented
  - Principle of OOP → encapsulation
- User only needs "rules"
  - Called "interface" for the class
    - In C++ → public member functions and associated comments
- Implementation of class hidden
  - Member function definitions elsewhere
  - User need not see them

Constructors

- Initialization of objects
  - Initialize some or all member variables
  - Other actions possible as well
- A special kind of member function
  - Automatically called when object declared
- Very useful tool
  - Key principle of OOP

Constructor Definitions

- Constructors defined like any member function
  - Except:
    1. Must have same name as class
    2. Cannot return a value; not even void!

Constructor Definition Example

- Class definition with constructor:
  - class DayOfYear

```
{
public:
    DayOfYear(int monthValue, int dayValue);
            //Constructor initializes month & day
    void input();
    void output();
    …
private:
    int month;
    int day;
}
```

Constructor Notes

- Notice name of constructor: DayOfYear
  - Same name as class itself!
- Constructor declaration has no return-type
  - Not even void!
- Constructor in public section
  - It's called when objects are declared
  - If private, could never declare objects!

Calling Constructors

- Declare objects:
  ```
  DayOfYear date1(7, 4),
            date2(5, 5);
  ```
- Objects are created here
  - Constructor is called
  - Values in parens passed as arguments to constructor
  - Member variables month, day initialized:
    date1.month → 7 date2.month → 5
    date1.dat → 4 date2.day → 5

Constructor Code

- Constructor definition is like all other member functions:

```
DayOfYear::DayOfYear(int monthValue, int dayValue)
{
    month = monthValue;
    day = dayValue;
}
```

- Note same name around ::
  – Clearly identifies a constructor
- Note no return type
  – Just as in class definition

Alternative Definition

- Previous definition equivalent to:

```
DayOfYear::DayOfYear(          int monthValue,
                               int dayValue)
          : month(monthValue), day(dayValue) ←
{…}
```

- Third line called "Initialization Section"
- Body left empty
- Preferable definition version

Constructor Additional Purpose

- Not just initialize data
- Body doesn't have to be empty
  - In initializer version
- Validate the data!
  - Ensure only appropriate data is assigned to class private member variables
  - Powerful OOP principle

Overloaded Constructors

- Can overload constructors just like other functions
- Recall: a signature consists of:
  - Name of function
  - Parameter list
- Provide constructors for all possible argument-lists
  - Particularly "how many"

Constructor with No Arguments

- Can be confusing
- Standard functions with no arguments:
  - Called with syntax: callMyFunction();
    - Including empty parentheses
- Object declarations with no "initializers":
  - DayOfYear date1;		// This way!
  - DayOfYear date();		// NO!
    - What is this really?
    - Compiler sees a function declaration/prototype!
    - Yes! Look closely!

Explicit Constructor Calls

- Can also call constructor AGAIN
  - After object declared
    - Recall: constructor was automatically called then
  - Can call via object's name; standard member function call
- Convenient method of setting member variables
- Method quite different from standard member function call

Explicit Constructor Call Example

- Such a call returns "anonymous object"
  - Which can then be assigned
  - **<u>In Action</u>**:
    DayOfYear holiday(7, 4);
    - Constructor called at object's declaration
    - Now to "re-initialize":
      holiday = DayOfYear(5, 5);
      - Explicit constructor call
      - Returns new "anonymous object"
      - Assigned back to current object

Default Constructor

- Defined as: constructor w/ no arguments
- One should always be defined
- Auto-Generated?
  - Yes & No
  - If no constructors AT ALL are defined → Yes
  - If any constructors are defined → No
- If no default constructor:
  - Cannot declare: MyClass myObject;
    - With no initializers

Class Type Member Variables

- Class member variables can be any type
  - Including objects of other classes!
  - Type of class relationship
    - Powerful OOP principle
- Need special notation for constructors
  - So they can call "back" to member object's constructor

Parameter Passing Methods

- Efficiency of parameter passing
  - Call-by-value
    - Requires copy be made → Overhead
  - Call-by-reference
    - Placeholder for actual argument
    - Most efficient method
  - Negligible difference for simple types
  - For class types → clear advantage
- Call-by-reference desirable
  - Especially for "large" data, like class types

The const Parameter Modifier

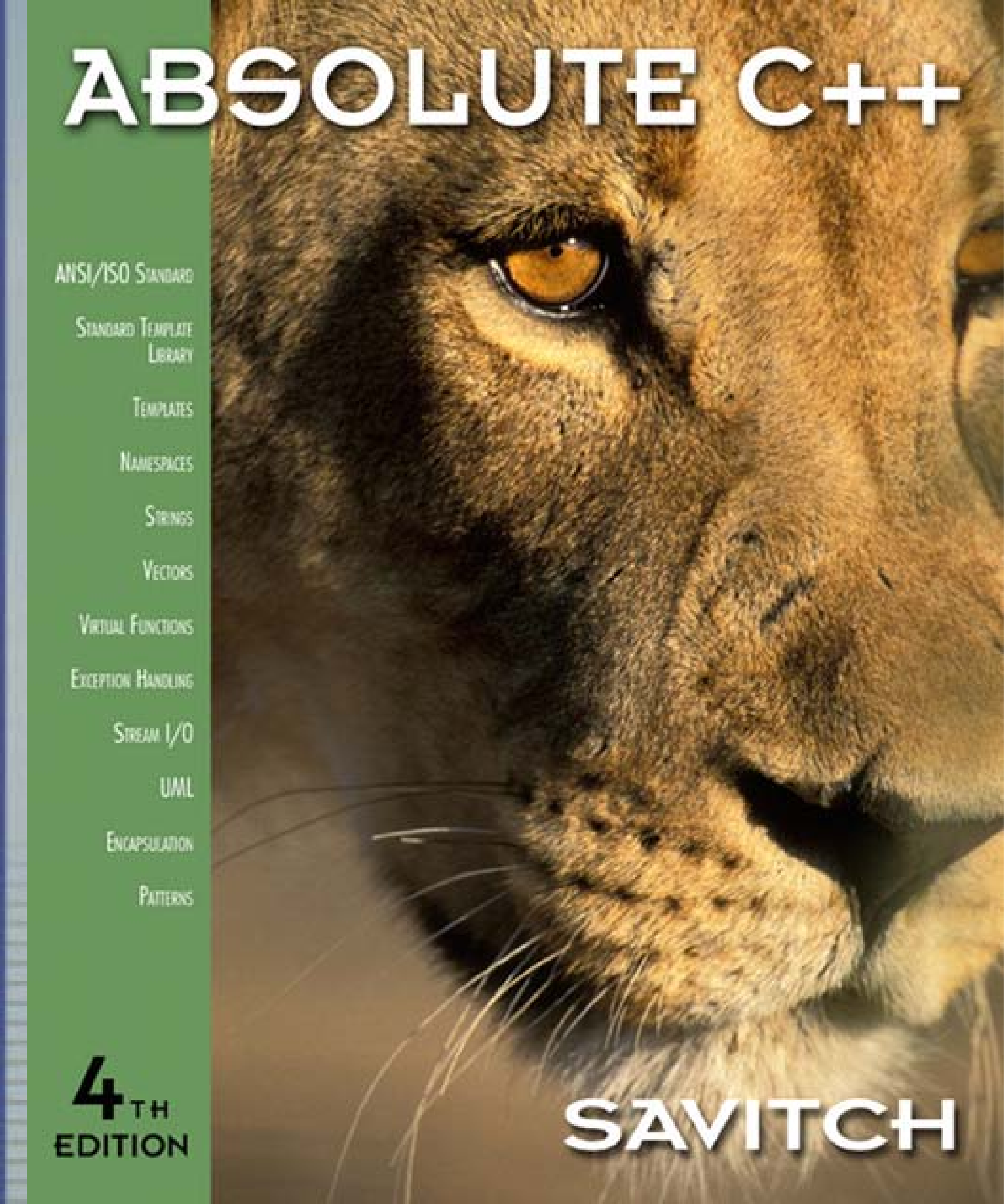
- Large data types (typically classes)
  - Desirable to use pass-by-reference
  - Even if function will not make modifications
- Protect argument
  - Use constant parameter
    - Also called constant call-by-reference parameter
  - Place keyword *const* before type
  - Makes parameter "read-only"
  - Attempts to modify result in compiler error

Use of const

- All-or-nothing
- If no need for function modifications
  – Protect parameter with const
  – Protect ALL such parameters
- This includes class member function parameters

Inline Member Functions

- Member function definitions
  - Typically defined separately, in different file
  - Can be defined IN class definition
    - Makes function "in-line"
- Again: use for very short functions only
- More efficient
  - If too long → actually less efficient!



Chapter 8, 9.3

Classes (operator overloading, C++ strings)

Learning Objectives

- Basic Operator Overloading
  - Unary operators
  - As member functions
- Friends and Automatic Type Conversion
  - Friend functions, friend classes
  - Constructors for automatic type conversion
- References and More Overloading
  - << and >>
  - Not = , [], ++, --
- Standard Class string
  - String processing

Operator Overloading Introduction

- Operators +, -, %, ==, etc.
  - Really just functions!
- Simply "called" with different syntax:
  x + 7
  - "+" is binary operator with x & 7 as operands
  - We "like" this notation as humans
- Think of it as:
  +(x, 7)
  - "+" is the function name
  - x, 7 are the arguments
  - Function "+" returns "sum" of its arguments

Operator Overloading Perspective

- Built-in operators
  - e.g., +, -, = , %, ==, /, *
  - Already work for C++ built-in types
  - In standard "binary" notation
- We can overload them!
  - To work with OUR types!
  - To add "Chair types", or "Money types"
    - As appropriate for our needs
    - In "notation" we're comfortable with
- Always overload with similar "actions"!

Overloading Basics

- Overloading operators
  - VERY similar to overloading functions
  - Operator itself is "name" of function
- Example Declaration:

```
const Money operator +(     const Money& amount1,
                            const Money& amount2);
```

  - Overloads + for operands of type Money
  - Uses constant reference parameters for efficiency
  - Returned value is type Money
    - Allows addition of "Money" objects

Overloaded "+"

- Given previous example:
  - Note: overloaded "+" NOT member function
  - Definition is "more involved" than simple "add"
    - Requires issues of money type addition
    - Must handle negative/positive values
- Operator overload definitions generally very simple
  - Just perform "addition" particular to "your" type

Overloaded "=="

- Equality operator, ==
  - Enables comparison of Money objects
  - Declaration:
    ```
    bool operator ==(const Money& amount1,
                     const Money& amount2);
    ```
    - Returns bool type for true/false equality
  - Again, it's a non-member function
    (like "+" overload)

Overloaded "==" for Money: **Display 8.1** Operator Overloading

- Definition of "==" operator for Money class:

```
bool operator ==(const Money& amount1, const Money& amount2)
{
    return ((amount1.getDollars( ) == amount2.getDollars( ))
            && (amount1.getCents( ) == amount2.getCents( )));
}
```

Constructors Returning Objects

- Constructor a "void" function?
  - We "think" that way, but no
  - A "special" function
    - With special properties
    - CAN return a value!
- Recall return statement in "+" overload for Money type:
  - return Money(finalDollars, finalCents);
    - Returns an "invocation" of Money class!
    - So constructor actually "returns" an object!
    - Called an "anonymous object"

Returning by const Value

- Consider "+" operator overload again:
  const Money operator +(const Money& amount1,
  const Money& amount2);
  - Returns a "constant object"?
  - Why?
- Consider impact of returning "non-const" object to see…→

Returning by non-const Value

- Consider "no const" in declaration:
  ```
  Money operator +(    const Money& amount1,
                       const Money& amount2);
  ```
- Consider expression that calls:
  m1 + m2
  - Where m1 & m2 are Money objects
  - Object returned is Money object
  - We can "do things" with objects!
    - Like call member functions…

What to do with Non-const Object

- Can call member functions:
  - We could invoke member functions on object returned by expression m1+m2:
    - (m1+m2).output(); //Legal, right?
      - Not a problem: doesn't change anything
    - (m1+m2).input(); //Legal!
      - PROBLEM! //Legal, but MODIFIES!
    - Allows modification of "anonymous" object!
    - Can't allow that here!
- So we define the return object as const

Overloading Unary Operators

- C++ has unary operators:
  - Defined as taking one operand
  - e.g., - (negation)
    - x = -y;   // Sets x equal to negative of y
  - Other unary operators:
    - ++, --
- Unary operators can also be overloaded

Overload "-" for Money

- Overloaded "-" function declaration
  - Placed outside class definition:
    const Money operator –(const Money& amount);
  - Notice: only one argument
    - Since only 1 operand (unary)
- "-" operator is overloaded twice!
  - For two operands/arguments (binary)
  - For one operand/argument (unary)
  - Definitions must exist for both

Overloaded "-" Definition

- Overloaded "-" function definition:

```
const Money operator –(const Money& amount)
{
      return Money(-amount.getDollars(),
                   -amount.getCents());
}
```

- Applies "-" unary operator to built-in type
  - Operation is "known" for built-in types
- Returns anonymous object again

Other Overloads

- &&, ||, and comma operator
  - Predefined versions work for bool types
  - Recall: use "short-circuit evaluation"
  - When overloaded no longer uses short-circuit
    - Uses "complete evaluation" instead
    - Contrary to expectations
- Generally should not overload these operators

Friend Functions

- Nonmember functions
  - Recall: operator overloads as nonmembers
    - They access data through accessor and mutator functions
    - Very inefficient (overhead of calls)
- Friends can directly access private class data
  - No overhead, more efficient
- So: best to make nonmember operator overloads friends!

Friend Functions

- Friend function of a class
  - Not a member function
  - Has direct access to private members
    - Just as member functions do
- Use keyword *friend* in front of function declaration
  - Specified IN class definition
  - But they're NOT member functions!

Friend Function Uses

- Operator Overloads
  - Most common use of friends
  - Improves efficiency
  - Avoids need to call accessor/mutator member functions
  - Operator must have access anyway
    - Might as well give full access as friend
- Friends can be any function

Friend Classes

- Entire classes can be friends
  - Similar to function being friend to class
  - Example:
    class F is friend of class C
    - All class F member functions are friends of C
    - NOT reciprocated
    - Friendship granted, not taken
- Syntax: friend class F
  - Goes inside class definition of "authorizing" class

Overloading >> and <<

- Enables input and output of our objects
  - Similar to other operator overloads
  - New subtleties
- Improves readability
  - Like all operator overloads do
  - Enables:
    cout << myObject;
    cin >> myObject;
  - Instead of need for:
    myObject.output(); …

Standard Class string

- Defined in library:
  #include <string>
  using namespace std;
- String variables and expressions
  – Treated much like simple types
- Can assign, compare, add:

```
string s1, s2, s3;
s3 = s1 + s2;          //Concatenation
s3 = "Hello Mom!"  //Assignment
```

  – Note c-string "Hello Mom!" automatically converted to string type!

Display 9.4
Program Using the Class string

Display 9.4 Program Using the Class `string`

```
//Demonstrates the standard class string.
#include <iostream>
#include <string>
using namespace std;

int main( )
{
    string phrase;
    string adjective("fried"), noun("ants");
    string wish = "Bon appetite!";

    phrase = "I love " + adjective + " " + noun + "!";
    cout << phrase << endl
         << wish << endl;

    return 0;
}
```

Initialized to the empty string.

Two equivalent ways of initializing a string variable

Sample Dialogue

I love fried ants!
Bon appetite!

I/O with Class string

- Just like other types!
- string s1, s2;
  cin >> s1;
  cin >> s2;
- Results:
  User types in:
  May the hair on your toes grow long and curly!
- Extraction still ignores whitespace:
  s1 receives value "May"
  s2 receives value "the"

getline() with Class string

- For complete lines:

```
string line;
cout << "Enter a line of input: ";
getline(cin, line);
cout << line << "END OF OUTPUT";
```

- Dialogue produced:

  Enter a line of input: <u>Do be do to you!</u>
  Do be do to you!END OF INPUT

  – Similar to c-string's usage of getline()

Class string Processing

- Same operations available as c-strings
- And more!
  - Over 100 members of standard string class
- Some member functions:
  - .length()
    - Returns length of string variable
  - .at(i)
    - Returns reference to char at position i

Display 9.7 Member Functions of the Standard Class string (1 of 2)

Display 9.7 Member Functions of the Standard Class string

EXAMPLE	REMARKS
Constructors	
`string str;`	Default constructor; creates empty `string` object `str`.
`string str("string");`	Creates a `string` object with data `"string"`.
`string str(aString);`	Creates a `string` object `str` that is a copy of `aString`. `aString` is an object of the class `string`.
Element access	
`str[i]`	Returns read/write reference to character in `str` at index `i`.
`str.at(i)`	Returns read/write reference to character in `str` at index `i`.
`str.substr(position, length)`	Returns the substring of the calling object starting at `position` and having `length` characters.
Assignment/Modifiers	
`str1 = str2;`	Allocates space and initializes it to `str2`'s data, releases memory allocated for `str1`, and sets `str1`'s size to that of `str2`.
`str1 += str2;`	Character data of `str2` is concatenated to the end of `str1`; the size is set appropriately.
`str.empty( )`	Returns `true` if `str` is an empty `string`; returns `false` otherwise.

(continued)

Display 9.7 Member Functions of the Standard Class string (2 of 2)

Display 9.7 **Member Functions of the Standard Class `string`**

EXAMPLE	REMARKS
`str1 + str2`	Returns a `string` that has `str2`'s data concatenated to the end of `str1`'s data. The size is set appropriately.
`str.insert(pos, str2)`	Inserts `str2` into `str` beginning at position pos.
`str.remove(pos, length)`	Removes substring of size `length`, starting at position pos.
Comparisons	
`str1 == str2` `str1 != str2`	Compare for equality or inequality; returns a Boolean value.
`str1 < str2` `str1 > str2`	Four comparisons. All are lexicographical comparisons.
`str1 <= str2` `str1 >= str2`	
`str.find(str1)`	Returns index of the first occurrence of `str1` in `str`.
`str.find(str1, pos)`	Returns index of the first occurrence of string `str1` in `str`; the search starts at position `pos`.
`str.find_first_of(str1, pos)`	Returns the index of the first instance in `str` of any character in `str1`, starting the search at position `pos`.
`str.find_first_not_of (str1, pos)`	Returns the index of the first instance in `str` of any character *not* in `str1`, starting search at position pos.

C-string and string Object Conversions

- Automatic type conversions
  - From c-string to string object:
    ```
    char aCString[] = "My C-string";
    string stringVar;
    stringVar = aCstring;
    ```
    - Perfectly legal and appropriate!
  - aCString = stringVar;
    - ILLEGAL!
    - Cannot auto-convert to c-string
  - Must use explicit conversion:
    ```
    strcpy(aCString, stringVar.c_str());
    ```