

CSC 221: Computer Programming I

Fall 2001

- C++ libraries
 - cmath, ctype
- classes and objects
 - abstract data types, classes and objects
 - data fields, member functions
- string class
 - data fields and member functions
 - logical connectives, short-circuit evaluation
- user-defined classes
 - Die class, projects and separate compilation

C++ libraries

C++ provides numerous libraries of useful code

- cmath functions for manipulating numbers, including:

```
double fabs(double x);           // returns absolute value of x
double sqrt(double x);          // returns square root of x
double ceil(double x);          // returns x rounded up
double floor(double x);         // returns x rounded down
double pow(double x, double y); // returns x^y
```

- ctype functions for testing and manipulating characters, including:

```
bool isalpha(char ch); // returns true if ch is a letter
bool islower(char ch); // returns true if ch is a lower-case letter
bool isupper(char ch); // returns true if ch is an upper-case letter
bool isdigit(char ch); // returns true if ch is a digit
bool isspace(char ch); // returns true if ch is whitespace
                        // (space, tab, newline, . . .)

char tolower(char ch); // returns lower-case equivalent of ch
char toupper(char ch); // returns upper-case equivalent of ch
                        // NOTE: both return ch if not a letter
```

character examples

suppose you wanted to prompt the user for a y/n answer

```
char response;
cout << "Do you want to play again? (y/n) ";
cin >> response;

if (tolower(response) == 'y') {
    PlayGame();
}
else {
    cout << "Thanks for playing." << endl;
}
```

suppose you wanted to read in a middle initial and verify it

```
char middleInitial;
cout << "Enter your middle initial: ";
cin >> middleInitial;

if (isalpha(middleInitial) == false) {
    cout << "That is not a valid initial." << endl;
}
```

abstract data types

for primitive types such as numbers and characters,
libraries of stand-alone functions suffice

for defining/manipulating more complex types,
a better (more generalizable) approach is needed

an *abstract data type (ADT)* is a collection of data and the associated operations that can be applied to that data

e.g., an integer is a number (series of digits) with assumed operations: addition, subtraction, multiplication, division, compare with other integers, ...

e.g., a string is a sequence of characters enclosed in quotes with operations: concatenation, determine its length, access a character, access a substring, ...

C++ classes and ADT's

in C++, new abstract data types can be defined as *classes*

- once a class has been defined and #included into a program, it is indistinguishable from the predefined data types of C++
 - by defining a new class, you can extend the expressibility of C++
- when you create an instance of a class, that *object* encapsulates the data and operations of that class (a.k.a. *member functions*)

EXAMPLE: the C++ string class in <string>

DATA: a sequence of characters (enclosed in quotes)

MEMBER FUNCTIONS:

```
+ >> << // operators for concatenation, input, and output
int length(); // returns number of chars in string
char at(int index); // returns character at index (first index is 0)
string substr(int pos, int len); // returns substring starting at pos of length len
int find(string substr); // returns position of first occurrence of substr,
// returns constant string::npos if not found
int find(char ch); // similarly finds index of character ch
. . .
```

string class

to create an object,
declare a variable of
that type (as before)

to call a member
function on that
object,

object.funtionCall

e.g.,

```
word.length()
word.at(0)
```

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

string Capitalize(string str);

int main()
{
    string word = "foobar";

    cout << word << " contains " << word.length()
         << " characters." << endl;

    cout << "Capitalized, its: " << Capitalize(word) << endl;

    char ch;
    cout << "Enter a character to search for: ";
    cin >> ch;

    int index = word.find(ch);
    if (index == string::npos) {
        cout << word << " does not contain " << ch << endl;
    }
    else {
        cout << ch << " is found at index " << index << endl;
    }

    return 0;
}

////////////////////////////////////
string Capitalize(string str)
// Assumes: str is a single word (no spaces)
// Returns: a copy of str with the first char capitalized
{
    char cap = toupper(str.at(0));
    return cap + str.substr(1, str.length()-1);
}
```

Pig Latin (v. 1)

suppose we want to convert a word into Pig Latin

- simplest version

nix → ixnay

pig → igpay

latin → atinlay

banana → ananabay

- to convert a word, move the last letter to the end and add "ay"

```
// piglatin.cpp      Dave Reed      9/30/01
//
// First version of Pig Latin translator.
//
#include <iostream>
#include <string>
using namespace std;

string PigLatin(string word);

int main()
{
    string word;
    cout << "Enter a word: ";
    cin >> word;

    cout << "That translates to: " << PigLatin(word) << endl;

    return 0;
}

//
string PigLatin(string word)
// Assumes: word is a single word (no spaces)
// Returns: the Pig Latin translation of the word
{
    return word.substr(1, word.length()-1) + word.at(0) + "ay";
}
```

opsoay?

using our program,

oops → opsoay

apple → ppleaay

ARE THESE RIGHT?

for "real" Pig Latin, you must consider first letter of word

- if a consonant, then convert as before (move first letter to end then add "ay")
- if a vowel, simply add "way" to the end

oops → oopsway

apple → appleway

so, we need to be able to determine whether the first letter is a vowel

HOW?

IsVowel function (v. 1)

whenever you have a well-defined, self-contained task to perform, consider defining a function

- IsVowel function takes a character as input, returns true if it is a vowel (else false)
- uses cascading if-else to check all possible letters (lower-case and upper-case)

CAN WE SIMPLIFY THIS?

```
bool IsVowel(char ch)
// Assumes: ch is a letter
// Returns: true if ch is a vowel
{
    if (ch == 'a') {
        return true;
    }
    else if (ch == 'A') {
        return true;
    }
    else if (ch == 'e') {
        return true;
    }
    else if (ch == 'E') {
        return true;
    }
    .
    .
    else if (ch == 'U') {
        return true;
    }
    else {
        return false;
    }
}
```

IsVowel function (v. 2)

could make use of the tolower function from <cctype>

- assume <cctype> library has been #included
- first convert the character to lower-case, then only have to check against lower-case vowels

STILL TEDIOUS!

LOTS OF TESTS WITH THE SAME RESULT

```
bool IsVowel(char ch)
// Assumes: ch is a letter
// Returns: true if ch is a vowel
{
    ch = tolower(ch);
    if (ch == 'a') {
        return true;
    }
    else if (ch == 'e') {
        return true;
    }
    else if (ch == 'i') {
        return true;
    }
    else if (ch == 'o') {
        return true;
    }
    else if (ch == 'u') {
        return true;
    }
    else {
        return false;
    }
}
```

Boolean logical connectives

C++ provides logical connectives for complex Boolean expressions

&& represents conjunction (AND)

```
if (ch >= 'A' && ch <= 'Z') {  
    cout << ch << " is a capital letter." << endl;  
}
```

note: characters are comparable, and letters are consecutive

|| represents disjunction (OR)

```
if (grade < 0 || grade > 100) {  
    cout << "That is not a valid grade!" << endl;  
}
```

! represents negation (NOT)

```
if (!isalpha(middleInitial)) {  
    cout << "That is not a valid initial." << endl;  
}
```

short-circuit evaluation

&& and || both employ short-circuit evaluation

- as soon as a test in a conjunction fails, stop and evaluate to false

```
if (ch >= 'A' && ch <= 'Z') {  
    cout << ch << " is a capital letter." << endl;  
}
```

if $ch < 'A'$, then no point in comparing with 'Z', return false

- as soon as a test in a disjunction succeeds, stop and evaluate to true

```
if (grade < 0 || grade > 100) {  
    cout << "That is not a valid grade!" << endl;  
}
```

if $grade < 0$, then no point in comparing with 100, return true

in addition to avoiding unnecessary computation, can be useful as a filter

```
if (numberOfScores > 0 && sumOfScores/numberOfScores >= 90) {  
    cout << "You get an A." << endl;  
}
```

IsVowel function (v. 3)

in IsVowel, can combine all cases that return true using OR ('||')

```
bool IsVowel(char ch)
// Assumes: ch is a letter
// Returns: true if ch is a vowel
{
    ch = tolower(ch);
    if (ch == 'a') {
        return true;
    }
    else if (ch == 'e') {
        return true;
    }
    else if (ch == 'i') {
        return true;
    }
    else if (ch == 'o') {
        return true;
    }
    else if (ch == 'u') {
        return true;
    }
    else {
        return false;
    }
}
```

```
bool IsVowel(char ch)
// Assumes: ch is a letter
// Returns: true if ch is a vowel ("aeiouAEIOU")
{
    ch = tolower(ch);
    if (ch == 'a' || ch == 'e' ||
        ch == 'i' || ch == 'o' || ch == 'u') {
        return true;
    }
    else {
        return false;
    }
}
```

IsVowel function (v. 4)

whenever you see an if-else of this pattern,

```
if (SOME_TEST) {
    return true;    // if SOME_TEST is true, returns true
}
else {
    return false;  // if SOME_TEST is false, returns false
}
```

the same effect could be achieved by returning the TEST value

```
return SOME_TEST;    // returns value of SOME_TEST directly
```

much cleaner and clearer

```
bool IsVowel(char ch)
// Assumes: ch is a letter
// Returns: true if ch is a vowel ("aeiouAEIOU")
{
    ch = tolower(ch);
    return (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u');
}
```

Pig Latin (v. 2)

```
// piglatin.cpp    Dave Reed    9/30/01
//
// Second version of Pig Latin translator.
///////////////////////////////////////////////////////////////////

#include <iostream>
#include <string>
#include <cctype>
using namespace std;

string PigLatin(string word);
bool IsVowel(char ch);

int main()
{
    string word;
    cout << "Enter a word: ";
    cin >> word;

    cout << "That translates to: "
         << PigLatin(word) << endl;

    return 0;
}

/////////////////////////////////////////////////////////////////
```

```
string PigLatin(string word)
// Assumes: word is a single word (no spaces)
// Returns: the Pig Latin translation of the word
{
    if (IsVowel(word.at(0))) {
        return word + "way";
    }
    else {
        return word.substr(1, word.length()-1) +
            word.at(0) + "ay";
    }
}

bool IsVowel(char ch)
// Assumes: ch is a letter
// Returns: true if ch is a vowel ("aeiouAEIOU")
{
    ch = tolower(ch);

    return (ch == 'a' || ch == 'e' ||
            ch == 'i' || ch == 'o' || ch == 'u');
}
```

IsVowel function (v. 5 & v. 6)

could utilize another string member function: `find`

recall: `find` searches for the first occurrence of a character or substring in a string, and returns its index (`string::npos` if not found)

create a string of all vowels, then search for `ch` using `find`

ADVANTAGES?

```
bool IsVowel(char ch)
// Assumes: ch is a letter
// Returns: true if ch is a vowel ("aeiouAEIOU")
{
    const string VOWELS = "aeiouAEIOU";

    if (VOWELS.find(ch) != string::npos) {
        return true;
    }
    else {
        return false;
    }
}
```

```
bool IsVowel(char ch)
// Assumes: ch is a letter
// Returns: true if ch is a vowel ("aeiouAEIOU")
{
    const string VOWELS = "aeiouAEIOU";

    return (VOWELS.find(ch) != string::npos);
}
```

Pig Latin (v. 3)

```
// piglatin.cpp    Dave Reed    9/30/01
//
// Third version of Pig Latin translator.
//
#include <iostream>
#include <string>
using namespace std;

string PigLatin(string word);
bool IsVowel(char ch);

int main()
{
    string word;
    cout << "Enter a word: ";
    cin >> word;

    cout << "That translates to: "
         << PigLatin(word) << endl;

    return 0;
}
//
```

```
string PigLatin(string word)
// Assumes: word is a single word (no spaces)
// Returns: the Pig Latin translation of the word
{
    if (IsVowel(word.at(0))) {
        return word + "way";
    }
    else {
        return word.substr(1, word.length()-1) +
            word.at(0) + "ay";
    }
}

bool IsVowel(char ch)
// Assumes: ch is a letter
// Returns: true if ch is a vowel ("aeiouAEIOU")
{
    const string VOWELS = "aeiouAEIOU";
    return (VOWELS.find(ch) != string::npos);
}
```

reightoncay?

using our program,

creighton → reightoncay

thrill → hrilltay

ARE THESE RIGHT?

for "real" Pig Latin, if the word starts with a sequence of consonants,
must move the entire sequence to the end then add "ay"

creighton → eightoncray

thrill → illthray

so, we need to be able to find the first occurrence of a vowel

HOW?

Handling multiple consonants

modifications to PigLatin are relatively easy

can use the find member function to find a specific character, but how do you search for *any* vowel

```
string PigLatin(string word)
// Assumes: word is a single word (no spaces)
// Returns: the Pig Latin translation of the word
{
    int vowelIndex = FindVowel(word);

    if (vowelIndex == 0 || vowelIndex == string::npos) {
        return word + "way";
    }
    else {
        return word.substr(vowelIndex, word.length()-vowelIndex) +
            word.substr(0, vowelIndex) + "ay";
    }
}

int FindVowel(string str)
// Assumes: str is a single word (no spaces)
// Returns: the index of the first vowel in str, or
// string::npos if no vowel is found
{
    ????
}
```

UGLY!!!

FindVowel can be done using the tools we know, but not easily

- must search for each vowel (lower-case and upper-case)
- then must compare all to find smallest index (if any)

WE REALLY NEED LOOPS!

```
int FindVowel(string str)
// Assumes: str is a single word (no spaces)
// Returns: the index of the first vowel in str, or
// string::npos if no vowel is found
// NOTE: THIS IS REALLY UGLY, REPETITIVE CODE -- CALLS FOR LOOPING!
{
    int indexOfa = str.find('a'), indexOfA = str.find('A');
    int indexOfe = str.find('e'), indexOfE = str.find('E');
    int indexOfi = str.find('i'), indexOfI = str.find('I');
    int indexOfo = str.find('o'), indexOfO = str.find('O');
    int indexOfu = str.find('u'), indexOfU = str.find('U');

    int indexSoFar = string::npos;
    if (indexOfa != string::npos) {
        if (indexSoFar == string::npos || indexOfa < indexSoFar) {
            indexSoFar = indexOfa;
        }
    }
    if (indexOfA != string::npos) {
        if (indexSoFar == string::npos || indexOfA < indexSoFar) {
            indexSoFar = indexOfA;
        }
    }
    if (indexOfe != string::npos) {
        if (indexSoFar == string::npos || indexOfe < indexSoFar) {
            indexSoFar = indexOfe;
        }
    }
    .
    .
    .
    if (indexOfU != string::npos) {
        if (indexSoFar == string::npos || indexOfU < indexSoFar) {
            indexSoFar = indexOfU;
        }
    }
    return indexSoFar;
}
```

[link to entire source code](#)

User-defined classes

similar to string, you can define your own classes (types) in C++

example: suppose we want to simulate dice rolls for experimentation

- built-in `rand` function uses a complex algorithm to generate a sequence of uniformly distributed numbers

```
double rand();           returns a random number
```

Note: each time a program is run, the same sequence is generated!

- `srand` initializes the number generation algorithm so that a different sequence is obtained (but same seed → same sequence)

```
void srand(unsigned int seed); initializes the algorithm with seed
```

- can use the `time` function from `<ctime>` to specify a unique seed

```
srand(unsigned(time(0)));
```

dice simulation

ugly code

- lots of detail
- repetition

also, we would like have additional functionality

- allow for dice with different # of sides
- for a particular die, be able to determine
 - # of sides
 - # times rolled

```
// rollem.cpp
//
// Simulates rolling two dice
////////////////////////////////////

#include <iostream>
#include <ctime>
#include <cmath>
using namespace std;

int main()
{
    srand(unsigned(time(0)));

    int roll1 = int(fabs(rand()))%6 + 1;
    int roll2 = int(fabs(rand()))%6 + 1;

    cout << "You rolled " << (roll1 + roll2) << endl;

    return 0;
}
```

Die class

can encapsulate behavior of a die in a new class (type)

DATA: number of sides, count of number of rolls

OPERATIONS: roll the die, determine # sides, determine # rolls so far

in order to use a class (type), you must

- #include its definition

```
#include "Die.h"    quotes around file name specify a user-defined  
                   class (compiler will look in local directory)
```

- know how to create an object and apply the member functions

```
int Roll();         returns random roll of die  
int NumSides();    returns number of sides on die  
int numRolls();    returns number of times has been rolled
```

dice simulation (with Die objects)

much cleaner

- hides ugly details of getting time, calling srand, and squeezing the random number into the range 1 to 6
- also hides inclusion of low-level libraries

```
// rollem.cpp  
//  
// Simulates rolling two dice  
////////////////////////////////////  
  
#include <iostream>  
#include "Die.h"  
using namespace std;  
  
int main()  
{  
    Die d1, d2;  
  
    int roll1 = d1.Roll();  
    int roll2 = d2.Roll();  
  
    cout << "You rolled " << (roll1 + roll2) << endl;  
  
    return 0;  
}
```

dice simulation (cont.)

also more general

- can declare dice with diff. # of sides

Roll function automatically adjusts

- encapsulates the code for keeping track of # sides and # of rolls

each die keeps track of itself!

```
// rollem.cpp
//
// Simulates rolling two dice
////////////////////////////////////
#include <iostream>
#include "Die.h"
using namespace std;

int main()
{
    Die sixSided, eightSided(8);

    int sum1 = sixSided.Roll() + sixSided.Roll();
    cout << "Two 6-sided dice: " << sum1 << endl;

    int sum2 = sixSided.Roll() + eightSided.Roll();
    cout << "6- and 8-sided dice:" << sum2 << endl;

    cout << "The " << sixSided.NumSides()
         << "-sided die has been rolled "
         << sixSided.NumRolls() << " times." << endl;

    return 0;
}
```

ADT/class summary

an Abstract Data Type (ADT) is a collection of data along with the operations that are allowed on that data

C++ provides several built-in ADTs: int, double, char, bool, ...

in C++, can define a new ADT by defining a class

- a class encapsulates data fields (variables) and member functions

in order to use an ADT defined as a class,

- must be able to #include the class definition
- must know how to create an object & call the appropriate member functions

```
#include <string>

int length();
char at(int index);
int find(char ch);
. . .
```

```
#include "Die.h"

int Roll();
int NumSides();
int NumRolls();
```