1

C++ Basics

CSC 340 - Appendices A, G, H, I, Credit to Hui Yang

February 10, 2016

2

Overview

- Comments, keywords, variables, data types, typedef
- I/O
- Flow Control
- **Functions**
- Arrays
- Structures
- Strings
- File I/O
- Program Style and Documentation

3

Functions in C++

- Two types
  - Member functions, if declared in a class
  - Standalone functions, if declared outside of a class (unlike Java!)
- Function Components
  - return type
  - function name
  - parameter list
  - *function body*

```
bool isGreater( int first, int second ) {
  return ( first > second );
}
```

Function Terminology

- Function signature (used by compiler for overload resolution):
  function name and parameter list
  `isGreater(int, int);`
- Function declaration (prototype for a function):
  return type, function name, and parameter list
  `bool isGreater(int, int);`
- Function definition:
  function declaration with the function body

4

Function signatures do not include return type, because that does not participate in overload resolution

The main() function

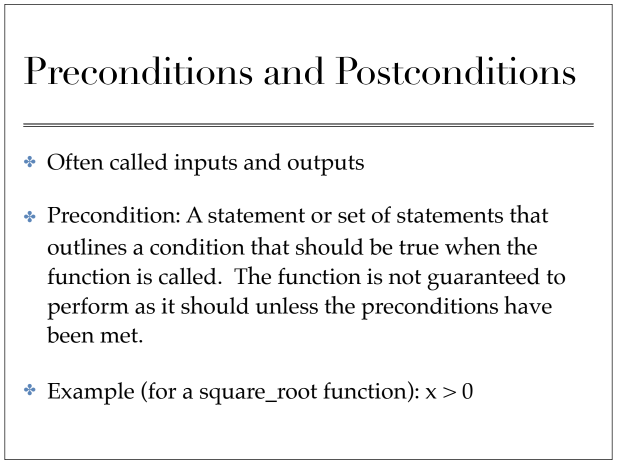
- `main()` is a function
  `int main()`
  `int main( int argc, char* argv[] )`
- Special in that one and only one function called main() will exist in a program
- Who calls main?
  - Operating system
  - Tradition holds it should have a return statement
    - Should return int or void
    - Special meaning to the operating system:
      0 is "normal exit",
      non-zero is "abnormal termination" (not standardized)

5

Function definition

- Placed outside of the function main()
- Functions are "equals" - no function is ever "part of" another
- Formal parameters in definition
  - "Placeholder" variables for data passed into function
- `return` statement
  - sends data back to caller

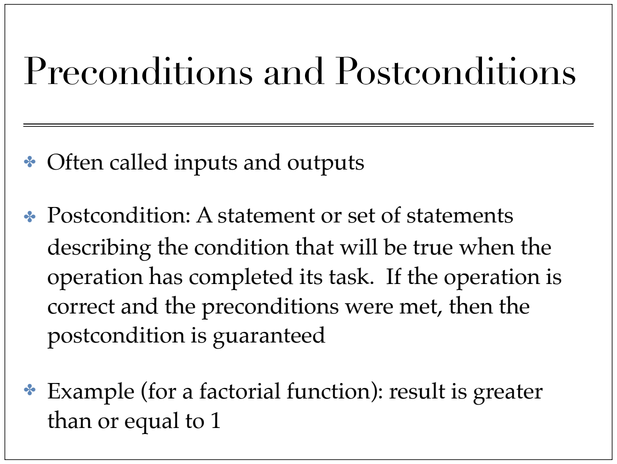
6



Preconditions and Postconditions

- Often called inputs and outputs
- Precondition: A statement or set of statements that outlines a condition that should be true when the function is called. The function is not guaranteed to perform as it should unless the preconditions have been met.
- Example (for a square_root function): x > 0

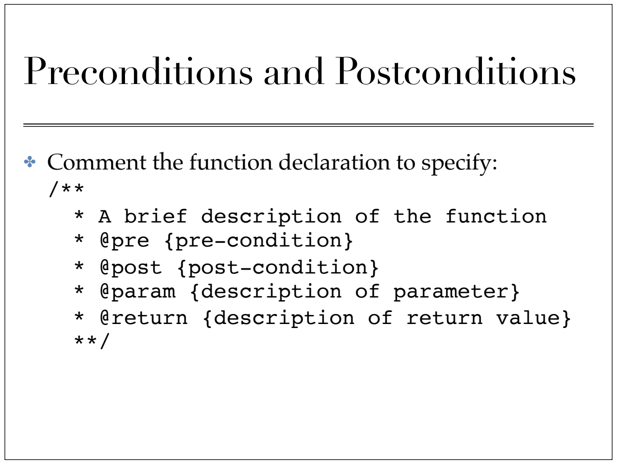
7



Preconditions and Postconditions

- Often called inputs and outputs
- Postcondition: A statement or set of statements describing the condition that will be true when the operation has completed its task. If the operation is correct and the preconditions were met, then the postcondition is guaranteed
- Example (for a factorial function): result is greater than or equal to 1

8



Preconditions and Postconditions

- Comment the function declaration to specify:

```
/**
  * A brief description of the function
  * @pre {pre-condition}
  * @post {post-condition}
  * @param {description of parameter}
  * @return {description of return value}
  **/
```

9

Example of precondition for a square_root function: x > 0

10

Default function parameters

- C++ allows users to specify default values for formal parameters
  - They must be the last parameters
    isGreater(int a, int b = 0); // legal
    isGreater(int a = 0, int b); // illegal
  - Do not overuse (can lead to ambiguity):

```
isGreater( int a, int b = 0 );
isGreater( int a ); // overloaded, compares a to 10000
isGreater( 10 ); // ambiguous invocation - which function called?
```

11

Parameter Passing - by value

- Call by value: `bool isGreater( int a, int b );`
  - Copy of a variable passed to function
  - Becomes "local" variable in function - modifications scoped only to function

12

Parameter Passing - by reference

- Call by reference: `bool isGreater( int & a, int b );`
  - Efficient when passing a large data type (like a big array)
  - Provides location of actual variable to function (refers to memory location where variable resides)
  - Modification scoped to local and calling function
  - Specified by & in formal parameter list
  - Can use const keyword to prevent data from being modified in function
    `bool isGreater( const int & a, int b );`

Functions calling functions

- Function’s declaration must appear before it can be called
- Function’s definition is typically placed elsewhere
  - After definition of main()
  - Separate file
- Common for functions to call many other functions
- A function can call itself: recursion

The return statement

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

Scope

- Local scope
  - Variable declared inside the body of a given function
  - Available only within that function
- Class scope
  - Defined within a class, outside any function
- Namespace scope
  - Defined within a namespace, outside any function, or class

Function Overloading

16

- Function overloading same as in Java
- Several functions can share the same function name as long as they have different signatures

Function Invocation

17

- Refer to the function name and provide a list of required types of arguments
  `isGreater( 20, 30 );`

Overview

18

- Comments, keywords, variables, data types, typedef
- I/O
- Flow Control
- Functions
- **Arrays**
- Structures
- Strings
- File I/O
- Program Style and Documentation

19

Arrays in C++

- Arrays in C++ are not objects!!
- Fixed size array
```
double d_array[20];
```
- Dynamic size array
    - Allocated at run time
    - Must use a C++ unique feature - the pointer
    - The programmer must write code to manage the memory
    - More on this later
- Do not go beyond the boundary!!

20

C++ vs. Java Reminders

- C++ does not perform any boundary checking at runtime
- Sometimes an out of range index will cause a runtime error, but not always
- Unlike Java, C++ allows a programmer to create an array of objects
    - In Java, you're really creating an array of references

21

Fixed size array declaration

- Declare the array:
  int score[5];
    - Declares an array of 5 integers named score
- Each part of the array called different things
    - Indexed or subscripted variables (Value in brackets called index or subscript)
    - Elements of the array
- Numbered from 0 to size - 1 (just like Java)

Accessing Arrays

- Access using index/subscript
  `cout << score[3];`
- Note two uses of brackets:
  - In declaration, specifies the size of the array
  - Anywhere else, specifies an index
- Size, subscript need not be literal:
  ```
  int score[ MAX_SCORES ];
  score[ n + 1 ] = 99;
  ```

Entire Arrays as Parameters

- Formal parameter can be an 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
- What's really passed?
  - Think of the array as three pieces: address of first indexed variable (array[0]), array base type, size of array

Using the Vector Library

- Requires an include directive:
  #include <vector>
  vector<double> score(10); // a vector of 10 doubles
- A vector maintains its current size and capacity just like an ArrayList in Java
- A programmer can shrink or expand a vector by calling the resize() member function
- Unlike C++ arrays, vectors are objects

Initializing vector elements

- Elements are added to a vector using the member function `push_back`
  - Adds an element in the next available position
    ```
    vector<double> sample;
    sample.push_back(0.0);
    sample.push_back(1.1);
    sample.push_back(2.2);
    ```

Accessing vector elements

- Vector elements are indexed starting with 0
- [] are used to read or change the value of an item
  ```
  v[i] = 42;
  cout << v[i];
  ```
- [] can not be used to initialize an arbitrary vector element

Alternate vector initialization

- A vector constructor exists that takes an integer argument and initializes that number of elements
  ```
  vector<int> v(10);
  ```
  - Initializes the first 10 elements to 0
  - `v.size()` would return 10
  - [] can now be used to assign elements index 0 through 9
  - `push_back` is used to assign elements to indexes greater than 9

Vector Initialization with Classes

- Constructor with an integer argument and a number type initializes elements to 0
- Constructor with an integer argument and a class type initializes elements using the default constructor for the class
  - Effect is to create an array of objects

Size of a vector

- The member function size returns the number of elements in a vector

```
for( int i = 0; i < sample.size(); i++ )
  cout << sample[i] << endl;
```

- A vector's capacity is the number of elements allocated in memory
  - Accessible using the capacity() member function

vector Issues

- Attempting to use [] to set a value beyond the size of a vector may not generate an error
  - The program will probably not behave as expected…
- The assignment operator with vectors does an element by element copy of the right hand vector
  - For class types, the assignment operator must make independent copies

31

vector Efficiency

- A vector's capacity is the number of elements allocated in memory
- Size is the number of elements initialized
- When a vector runs out of space, the capacity is automatically increased
  - Common approach is to double the size of a vector
  - More efficient than allocating smaller chunks of memory

32

Controlling vector capacity

- When efficiency is an issue
- Member function reserve can increase the capacity of a vector

```
v.reserve( 32 ); // at least 32 elements
v.reserve( v.size() + 10 ); // 10 more than current size
```

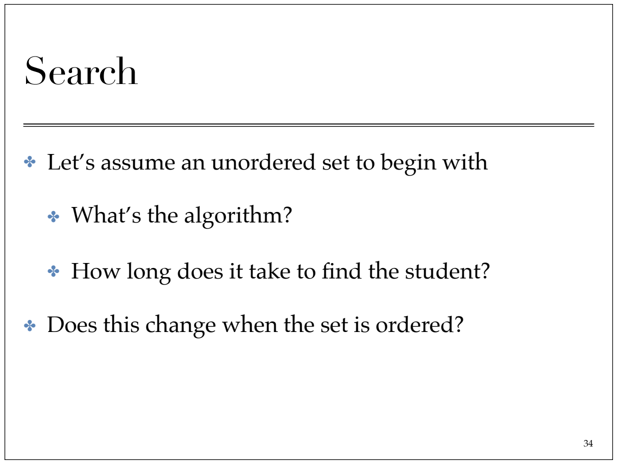
- resize can be used to shrink a vector

```
v.resize( 24 ); // elements beyond 24 are truncated
```

33

Brief detour

- Set up an example where we will want to pass a vector to a function…
- Search
  - Given the set of student IDs for students in this classroom, how can I determine if there is a student with ID 912361928?
  - (We'll be exploring search a little more later in the semester)

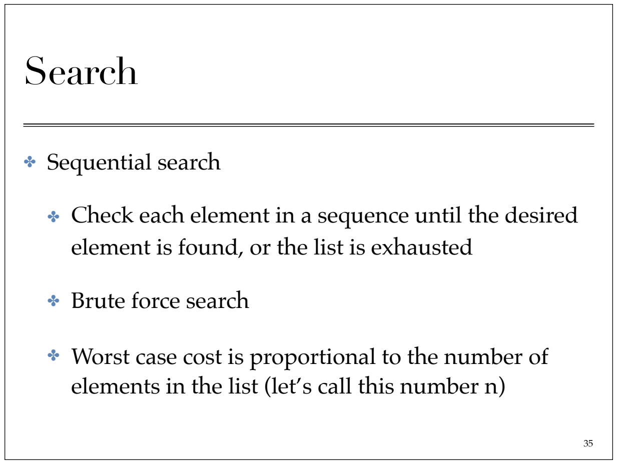


Search

- Let's assume an unordered set to begin with
  - What's the algorithm?
  - How long does it take to find the student?
- Does this change when the set is ordered?

34

34

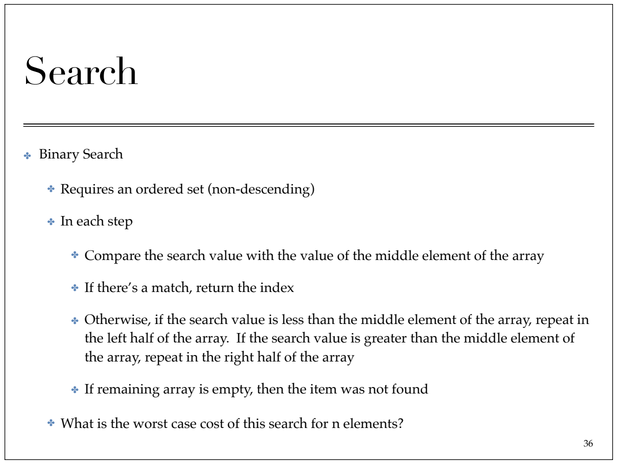


Search

- Sequential search
  - Check each element in a sequence until the desired element is found, or the list is exhausted
  - Brute force search
  - Worst case cost is proportional to the number of elements in the list (let's call this number n)

35

35



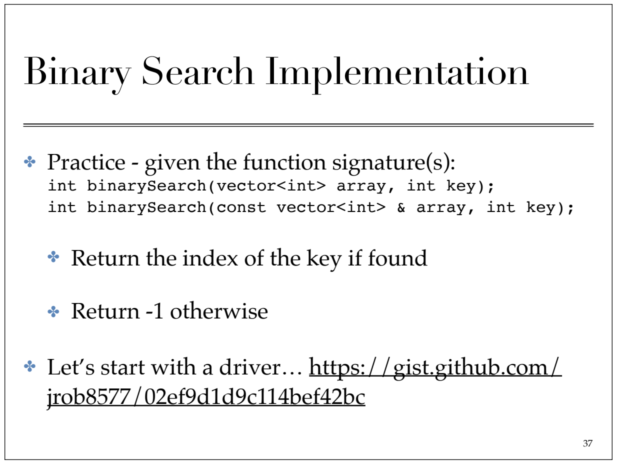
Search

- Binary Search
  - Requires an ordered set (non-descending)
  - In each step
    - Compare the search value with the value of the middle element of the array
    - If there's a match, return the index
    - Otherwise, if the search value is less than the middle element of the array, repeat in the left half of the array. If the search value is greater than the middle element of the array, repeat in the right half of the array
    - If remaining array is empty, then the item was not found
  - What is the worst case cost of this search for n elements?

36

36

log n



Binary Search Implementation

- Practice - given the function signature(s):

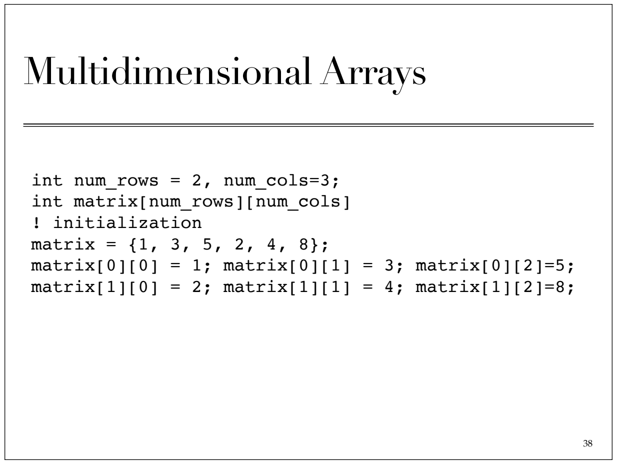
```
int binarySearch(vector<int> array, int key);
int binarySearch(const vector<int> & array, int key);
```

  - Return the index of the key if found
  - Return -1 otherwise
- Let's start with a driver… https://gist.github.com/jrob8577/02ef9d1d9c114bef42bc

37

37

Quick question - what's the difference in the two signatures?
Overview of driver - let's talk about what this is doing before we write the binary search algorithm
An untested solution at https://gist.github.com/jrob8577/c77dccb353d3ac5b620b

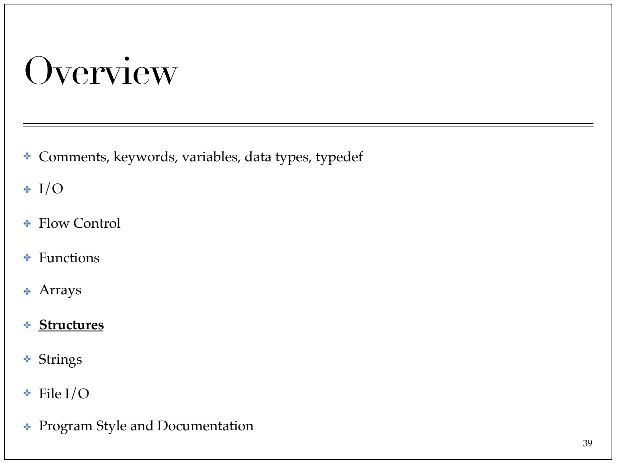


Multidimensional Arrays

```
int num_rows = 2, num_cols=3;
int matrix[num_rows][num_cols]
! initialization
matrix = {1, 3, 5, 2, 4, 8};
matrix[0][0] = 1; matrix[0][1] = 3; matrix[0][2]=5;
matrix[1][0] = 2; matrix[1][1] = 4; matrix[1][2]=8;
```

38

38



Overview

- Comments, keywords, variables, data types, typedef
- I/O
- Flow Control
- Functions
- Arrays
- **Structures**
- Strings
- File I/O
- Program Style and Documentation

39

39

40

Struct

- Contains multiple data members
  - Logically related
  - Can be of different data types
  - Example - Person: firstName, lastName, ssn, age, salary
- Is like a class but with all its data members defaulted to public
  - Is not strictly needed
  - Still used frequently to define data types that consist of mostly data where direct access is more convenient
  - Example - the data types of nodes in a linked list

41

Struct Syntax

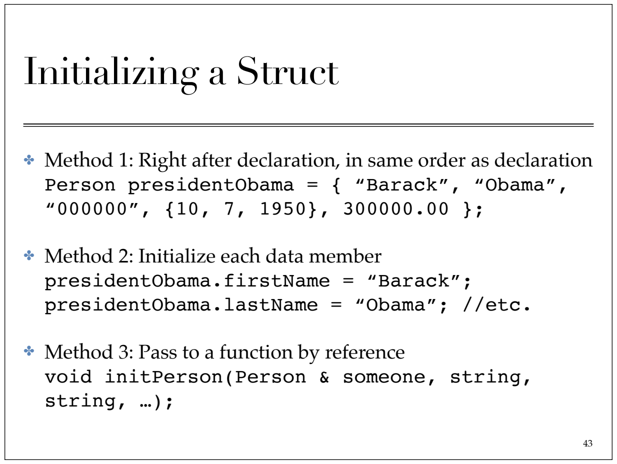
- Don’t forget semicolon!
- Note the composition of types (Person includes Date)

```
struct Date          struct Person
{                    {
  char day;            string firstName; //first name
  char month;          string lastName; //last name
  int year;            string ssn; //social number
};                     Date birthDay;
                       double salary;
                     };
```

42

Using Structs

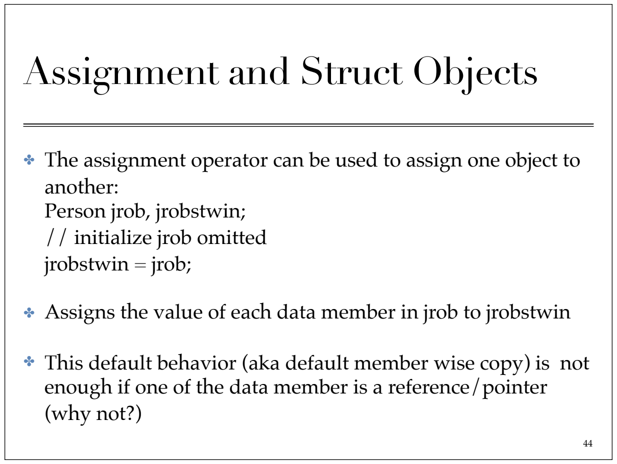
- Structure definition is generally placed outside any function definition
  - This makes the structure available to all code that follows the structure definition
- To declare objects of type `Person`:
  `Person presidentObama, jrob;`
- Each `Person` object contains the same set of data members, with different values



Initializing a Struct

- Method 1: Right after declaration, in same order as declaration
```
Person presidentObama = { “Barack”, “Obama”,
“000000”, {10, 7, 1950}, 300000.00 };
```
- Method 2: Initialize each data member
```
presidentObama.firstName = “Barack”;
presidentObama.lastName = “Obama”; //etc.
```
- Method 3: Pass to a function by reference
```
void initPerson(Person & someone, string,
string, …);
```

43

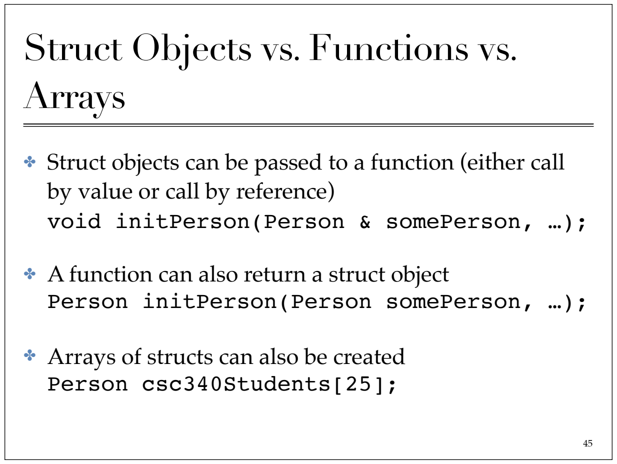


Assignment and Struct Objects

- The assignment operator can be used to assign one object to another:
  Person jrob, jrobstwin;
  // initialize jrob omitted
  jrobstwin = jrob;
- Assigns the value of each data member in jrob to jrobstwin
- This default behavior (aka default member wise copy) is not enough if one of the data member is a reference/pointer (why not?)

44

Because it copies the memory location of the member, meaning the two objects would share a reference to that member (and both be able to mutate that shared reference)



Struct Objects vs. Functions vs. Arrays

- Struct objects can be passed to a function (either call by value or call by reference)
```
void initPerson(Person & somePerson, …);
```
- A function can also return a struct object
```
Person initPerson(Person somePerson, …);
```
- Arrays of structs can also be created
```
Person csc340Students[25];
```

45

46

Overview

- Comments, keywords, variables, data types, typedef
- I/O
- Flow Control
- Functions
- Arrays
- Structures
- **Strings**
- File I/O
- Program Style and Documentation

47

The Standard string Class

- String is an array of characters
- The `string` class is defined in the string library and the names are in the standard namespace

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

48

Assignment of Strings

- Variables of type string can be assigned with the = operator

```
string s1, s2, s3;
s3 = s2;
```

- Quoted strings are type cast to type string

```
string s1 = “Hello World!”;
```

Using + With strings

- Variable of type string can be concatenated with the + operator

```
string s1, s2, s3;
s3 = s1 + s2;
```

- If s3 is not large enough to contain s1 + s2, more space is allocated

string Constructors

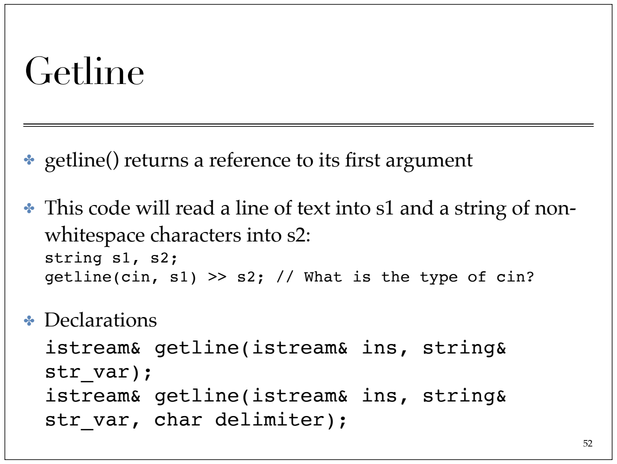
- The default string constructor initializes the string to the empty string
- Another string constructor takes a C-string argument
- Examples

```
string phrase; //empty string
string noun(“ants”);
```

I/O With Class string

- Basics covered in earlier I/O section
- One addition, get line can stop reading at a character specified in the argument list. Example shows stop reading when a ‘?’ is read:

```
string line;
cout << “Enter some input: \n”;
getline(cin, line, ‘?’);
```



Getline

- getline() returns a reference to its first argument
- This code will read a line of text into s1 and a string of non-whitespace characters into s2:

```
string s1, s2;
getline(cin, s1) >> s2; // What is the type of cin?
```

- Declarations

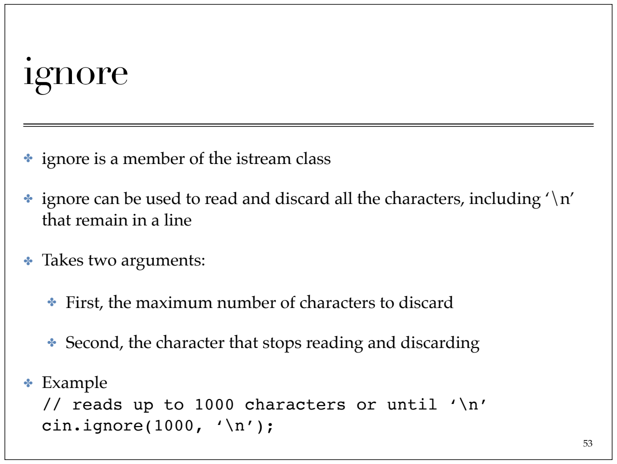
```
istream& getline(istream& ins, string&
str_var);
istream& getline(istream& ins, string&
str_var, char delimiter);
```

52

52

Returning a reference is important so we can “chain” calls…

These declarations will be important later when we start overloading operators!!



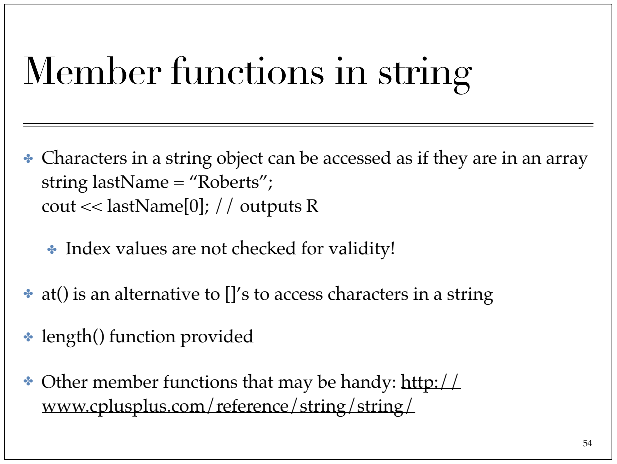
ignore

- ignore is a member of the istream class
- ignore can be used to read and discard all the characters, including ‘\n’ that remain in a line
- Takes two arguments:
  - First, the maximum number of characters to discard
  - Second, the character that stops reading and discarding
- Example

```
// reads up to 1000 characters or until ‘\n’
cin.ignore(1000, ‘\n’);
```

53

53



Member functions in string

- Characters in a string object can be accessed as if they are in an array
  string lastName = “Roberts”;
  cout << lastName[0]; // outputs R
  - Index values are not checked for validity!
- at() is an alternative to []’s to access characters in a string
- length() function provided
- Other member functions that may be handy: http://www.cplusplus.com/reference/string/string/

54

54

55

Overview

- Comments, keywords, variables, data types, typedef
- I/O
- Flow Control
- Functions
- Arrays
- Structures
- Strings
- **File I/O**
- Program Style and Documentation

56

Streams

- A flow of characters
- Input stream
  - Flows into program
  - We've discussed the standard input stream (cin) that comes from the keyboard
  - May also come from file
- Output stream
  - Flows out of program
  - We've discussed the standard output stream (cout) that goes to screen
  - May also go to file

57

Streams Usage

- Programmers can define other streams
- Used similarly to cin and cout

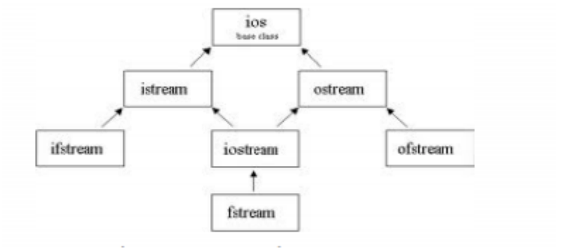
58

I/O in Java and C++

- Java I/O
  - A large library that is housed mostly in package java.io
  - Makes use of inheritance by defining four abstract classes: InputStream, OutputStream, Reader, and Writer
  - They can be extended to cover different sources of data such as files and arrays
- C++ I/O
  - Is considerably smaller than Java
  - Also uses inheritance

59

C++ I/O Hierarchy Simplified



60

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

61

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

62

Declaring Streams

- Streams declared like any other typed 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

63

Checking File Open Succeeded

- File opens could fail
  - input file doesn’t exist
  - No write permissions to output file
  - Unexpected results
- Member function fail()
  - Place call to fail() to check stream operation success
```
inStream.open(“stuff.txt”);
if(inStream.fail())
{
  cout << “File open failed.\n”;
  exit(1);
}
```

Streams Usage

- Once declared (and successfully opened), use normally

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

ofstream outStream;
outStream.open(“outfile.txt”);
outStream << “oneNumber = “ << oneNumber; //etc.
```

Checking End of File (EOF)

- Typical approach for reading a file: Use loop to process file until end
- Two ways to test for end of file
- Method 1: member function eof()
  - Reads each character until file ends
  - Returns bool (i.e. end of file reached)

```
inStream.get(next);
while( !inStream.eof() )
{
  cout << next;
  inStream.get(next);
}
```

Checking End of File (EOF)

- Method 2: read (extraction) operations returns a boolean value (inStream >> next) / / true or false returned
  - Returns true if read was successful
  - Returns false if attempted to read beyond end of file

```
double next, sum = 0;
while(inStream >> next)
  sum += next;
cout << “The sum is “ << sum << endl;
```

67

File Flush

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

67

68

Closing Files

- Files should be closed
  - When program completed getting input or sending output
  - Disconnects stream from file
    ```
    inStream.close();
    outStream.close();
    ```
  - Takes no arguments
- Files automatically close when program ends

68

69

File I/O Example

- https://gist.github.com/jrob8577/99d9820d9494bf44619c
- Let's parse the program
- Note that there is no command line output

69

Appending to a File

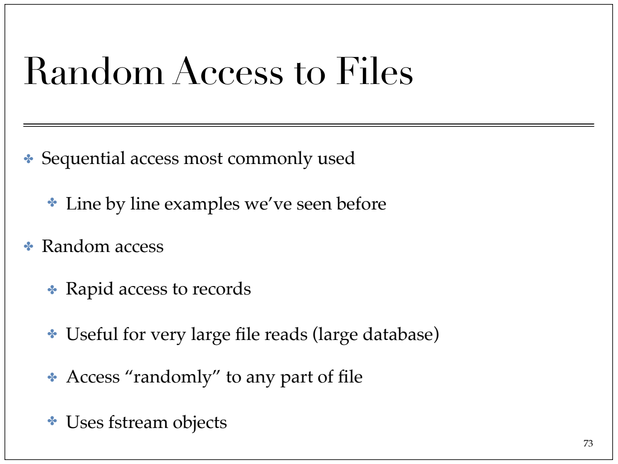
- Standard open operation begins with an empty file
  - If the file exists, the contents are lost
- Open for append:
  ```
  ofstream outStream;
  outStream.open(“important.txt”, ios::app);
  ```
  - If file doesn’t exist, creates it
  - If file exists, appends to end
  - `ios::app` argument is class ios defined constant (in <iostream> library, std namespace)

Character I/O with files

- All cin and cout character I/O same for files
- Member functions work the same:
  - get, get line
  - put, putback
  - peek, ignore

Output Member Functions

- Additional member functions:
  ```
  outStream.setf(ios::fixed);
  outStream.setf(ios::showpoint);
  outStream.precision(2);
  ```
- Member function `precision(x)` writes decimals with x digits after decimal
- Member function `setf()` sets many different output flags to affect how output is written

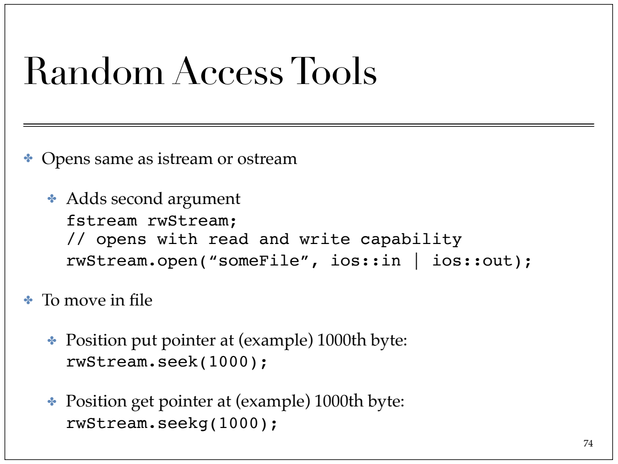


73

Random Access to Files

- Sequential access most commonly used
  - Line by line examples we've seen before
- Random access
  - Rapid access to records
  - Useful for very large file reads (large database)
  - Access "randomly" to any part of file
  - Uses fstream objects

73



74

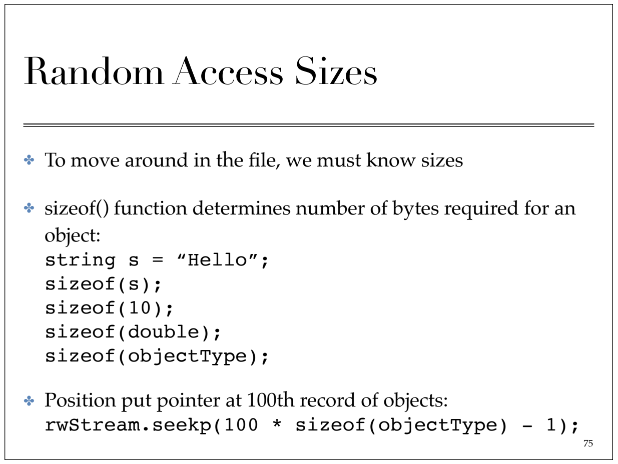
Random Access Tools

- Opens same as istream or ostream
  - Adds second argument

```
fstream rwStream;
// opens with read and write capability
rwStream.open("someFile", ios::in | ios::out);
```

- To move in file
  - Position put pointer at (example) 1000th byte:
    `rwStream.seek(1000);`
  - Position get pointer at (example) 1000th byte:
    `rwStream.seekg(1000);`

74



75

Random Access Sizes

- To move around in the file, we must know sizes
- sizeof() function determines number of bytes required for an object:

```
string s = "Hello";
sizeof(s);
sizeof(10);
sizeof(double);
sizeof(objectType);
```

- Position put pointer at 100th record of objects:
  `rwStream.seekp(100 * sizeof(objectType) - 1);`

75

Overview

- Comments, keywords, variables, data types, typedef
- I/O
- Flow Control
- Functions
- Arrays
- Structures
- Strings
- File I/O
- **Program Style and Documentation**

Program Style

- A program written with attention to style
  - easy to read
  - easy to correct
  - easy to maintain

Indenting

- Items considered a group should look like a group
  - Skip lines between logical groups of statements
  - Indent statements within statements
    ```
    if( x == 0 )
      statement;
    ```
- Braces {} create groups
  - Indent within braces to make group clear
  - Braces placed on separate lines are easier to locate

Comments

- // is the symbol for a single line comment
  - Comments are explanatory notes for the programmer
  - All text on the line following // is ignored by the compiler

    ```
    // calculate regular wages
    gross_pay = rate * hours;
    ```
- /* and */ enclose multiple line comments

  ```
  /* This is a multiple
       line comment.  Note the indentation
       I used…
  */
  ```