Lecture 2 – Data Structures

C++ Primer

1

Lecture Outline

- The C++ Environment
- Preprocessor Directives and Macros
- C++ Control Statements
- Primitive Data Types and Class Data Types
- Objects, Pointers, and References
- Functions
- Arrays and C Strings
- The `string` Class
- Input/Output Using Streams

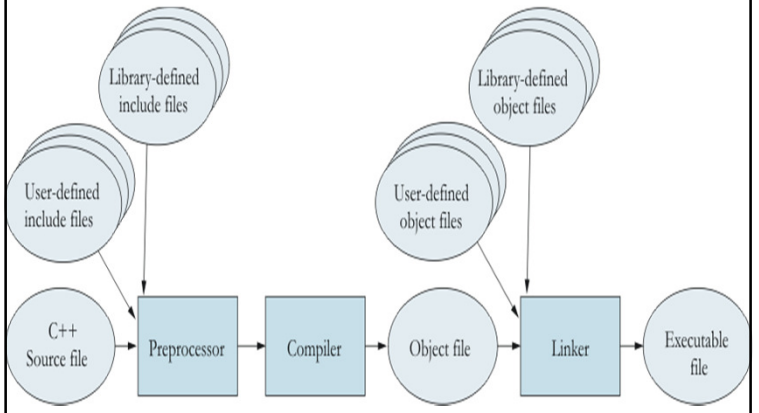
2

Compiling and Linking

- A *C++ program* consists of *one or more source files*.
- *Source files* contain *function and class declarations and definitions.*
  - *Files* that contain *only declarations* are incorporated into the source files that need them when they are compiled.
    - Thus they are called ***include files***.
  - *Files* that contain *definitions* are translated by the *compiler* into an intermediate form called ***object files***.
  - *One or more object files* are combined with to form the *executable file* by the *linker*.

3

Compiling and Linking



4

A Simple Program

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
  cout << "Enter your name\n";
  string name;
  getline(cin, name);
  cout << "Hello " << name
       << " - welcome to C++\n";
  return 0;
}
```

5

The `#include` Directive

- The first two lines:

```
#include <iostream>
#include <string>
```

  incorporate the *declarations* of the `iostream` and `string` *libraries* into the source code.

- If your program is going to use a member of the standard library, the appropriate *header file must be included at the beginning of the source code file.*

6

The `using` Statement

- The line

```
using namespace std;
```

  tells the compiler to make *all names in the predefined namespace **std** available*.

- The *C++ standard library* is defined within this namespace.
- Incorporating the statement

```
using namespace std;
```

  is an easy way to get access to the standard library.
  - But, it can lead to complications in larger programs.

7

The using declaration

- Instead of incorporating all names from a namespace into your program
  - It is a better approach to *incorporate only the names you are going to use.*
  - This is done with individual using declarations.

```
using std::cin;
using std::cout;
using std::string;
using std::getline;
```

8

The function main

- *Each program must include a main function.*
- This function is defined as follows:

```
int main()
{
  …
}
```

where the code for the function appears between the `{` and the `}`.

9

The stream insertion operator

- The statement:

```
cout << "Enter your name\n";
```

inserts the string into the *standard output stream*.

- The result is that it is displayed on the console.

10

The getline function

- The statement

```
getline(cin, name);
```

reads the characters from the input stream (keyboard) until a *new line character* is entered.

- The resulting string is stored in the `string` **`name`**.

11

The insertion operator again

- The statement:

```
cout << "Hello " << name << " - welcome to C++\n";
```

outputs three strings to the console:

```
Hello
```

the entered line

```
- welcome to C++
```

- If the characters `John Doe` were entered, the result would be

```
Hello John Doe - welcome to C++
```

12

Compiling and Executing

- The command to compile is dependent upon the compiler and operating system.
- For the ***gcc*** compiler (popular on Linux) the command would be:

  g++ -o HelloWorld HelloWorld.cpp
- For the *Microsoft compiler* the command would be:

  cl /EHsc HelloWorld.cpp
- To execute the program you would then issue the command

  HelloWorld

13

The Preprocessor

- *The compiler (effectively) makes several passes through the source program.*
- The *first* of these passes is done by what's known as the *preprocessor.*
  1. *Replace trigraphs* with their *equivalent*
  2. Splice *long lines into a single line.*
  3. *Remove comments* and replace by a *single space*.
  4. Split the *input file into tokens*
  5. Carry out *preprocessing directives*
  6. Expand *macros*
- Note that the *preprocessor is inherited from the C programming language.*

14

Trigraphs

- A C++ program uses the following characters:

  a b c d e f g h i j k l m n o p q r s t u v w x y z
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  0 1 2 3 4 5 6 7 8 9
  _ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ = , \ " '
- Not all of these are available in all countries of the world.
  - In Denmark the characters Æ æ Ø ø Å å are in place of [ ] { } | and \

15

Trigraphs (2)

- Trigraphs provide a way to write a C or C++ program using the minimum common set of characters.

Trigraph	Equivalent
??=	#
??/	\
??'	^
??(	[
??)	]
??!	|
??>	{
??>	}
??-	~

16

Alternate Tokens

- In addition to trigraphs, the C++ language provides for *alternate keywords or diagraphs for some operator* symbols:

Alternate	Equivalent	Alternate	Equivalent	Alternate	Equivalent
<%	{	and	&&	and_eq	%=
%>	}	bitor	|	or_eq	|=
<:	[	or	||	xor_eq	^=
:>	]	xor	^	not	!
%:	#	compl	~	not_eq	!=
%:%:	##	bitand	&		

17

Splicing Long Lines

- If a *line ends* with the character \ (or the trigraph sequence ??/)
  - Then the following line is appended to this line and the result is considered a single line.

18

Comments

- There are *two types* of comments:
- **Form 1:**
  - A comment begins with the characters /* and ends with the characters */
- **Form 2:**
  - A comment begins with the characters // and ends at the end of the current line
- All characters of a comment are replaced by a single space by the preprocessor.
- Note that a */* that follows a // is ignored*.
- A *// that appears after a /* is also ignored*.
- A comment that begins with a */* is terminated by the first */* that is encountered.

19

Macros

- Macros are defined by the forms:

  `#define` *macro-name macro-definition*

  `#define` *macro-name*`(`*parameters*`)` *macro-definition*
  - The definition ends at the end of the current line.
  - Macros requiring longer lines use long-line splicing.
- Examples:

```
#define NULL 0
#define MAX(x, y) ((x) > (y) ? (x) : (y))
```

- Within the program, where ever a macro appears, it is replaced by its definition.

20

Macro Expansion

- Given the previous examples.

```
if (ptr == NULL)
```

  - appears to the compiler as

```
if (ptr == 0)
```

  - and

```
c = max(a, b);
```

  - appears to the compiler as

```
c = ((a) > (b) ? (a) : (b));
```

Example:

```
#define SQR(x)    x*x
  c = SQR(a+b)
```

  - **Result?**

21

The # operator

- Placing a **#** *before a macro parameter* results in that parameter being *replaced by a string literal*.
- Example:

```
#define KW_assert(x) \
if (!(x)) {\
   std::cerr << "Assertion " << #x << " failed\n"\
             << "Line " << __LINE__ << "in file "\
              << __FILE__ << "\n";\
   exit(1);\
}
```

22

The # operator

- Given the previous definition
  - The statement:
    - KW_assert(length > 5)
  - Will appear to the compiler as:

```
if (!(length > 5) {\
  std::cerr << "Assertion " << "length > 5" << "failed\n"\
    << "Line " << line number << "in file "
    << file name << "\n";
  exit(1);
}
```

23

The __LINE__ and __FILE__ macros

- The *macros* __LINE__ and __FILE__ are *pre-defined.*
- They are replaced by the *line number* and *file name* of the *current line* in the *current file*.
- Note that these macros *begin and end* with *two underscore* **(_)** characters.

24

Conditional Compilation

- Forms:

```
#ifdef macro-name
code to be compiled if macro-name is defined
#else
code to be compiled if macro-name is not defined
#endif
```

or

```
#ifndef macro-name
code to be compiled if macro-name is not defined
#else
code to be compiled if macro-name is defined
#endif
```

25

Using Conditional Compilation

- Some *functions* are defined to be *used by both C and C++* programs.
- If a C/C++ compiler is *compiling* a program as a *C++ program*, then the macro *__cplusplus* is defined. (Note the two _ chars).
- Then the function would be declared as follows:

```
#ifdef __cplusplus
extern "C" {
#endif

function declaration

#ifdef __cplusplus
}
#endif
```

26

Preventing Multiple Includes

- A *header* file *may be included by another header* file.
- The user of the header file may not know this and may include a *duplicate*.
- This may lead to a *compile error*.
- To prevent this, *each include file should be structured as* follows:

```
#ifndef unique-name
#define unique-name
...
#endif
```

- Generally *unique-name* is *related to the file name*.
  - Example **myfile.h** would use the name *MYFILE_H_*

27

More on #include directive

- The *#include* directive has *two forms:*

```
#include <header>
```

  - is reserved for *standard library headers*.

```
#include "file-name"
```

  - is used for *user-defined include files*.
- The *convention* is that *user-defined* include files will end with the *extension* `.h`.
- Note that the *standard* library headers *do not end with* `.h`.

28

Compound Statements

- A sequence of statements between a *{ and }* character is called a *compound statement*.
- Syntactically, a compound statement *can be used in place of a single statement.*

29

Control Structure	Purpose	Syntax

`if ... else`	Used to write a decision with condition selecting the alternative to be executed.	`if (`*condition*`) {`
		`  . . .`
		`} else {`
		`  . . .`
		`}`

`switch`	Used to write a decision with scalar values (integers) that select the alternative to be executed.	`switch (selector) {`
		`case label:` *statements*`;`
		`  break;`
		`case label:` *statements*`;`
		`  break;`
		`  . . .`
		`default:` *statements*`;`
		`}`

`while`	Used to write a loop that specifies the repetition condition in the loop header. The condition is tested before each loop iteration	`while (condition) {`
		`  . . .`
		`}`

30

Control Structure	Purpose	Syntax

`do . . . while`	Used to write a loop that executes at least once. The repetition *condition* is at the end of the loop.	`do {`
		`  . . .`
		`} while (`*condition*`);`

`for`	Used to write a loop that specifies the *initialization*, repletion *condition*, and *update* steps in the loop header.	`for (`*initialization*`;`
		`  `*condition*`;`
		`  `*update*`) {`
		`  . . .`
		`}`

31

Using braces and indentation

- There are several *coding styles*.
- The one used in this text is:
  - Place a *{ on the same line* as the condition for an **`if`**, **`while`**, or **`for`** statement.
  - *Indent each line* of the controlled compound statement.
  - Place the closing *} on its own line*, indented at the same level as the if, while, or for.
  - For *else* conditions, use the form:

    } else {

32

Nested If Statements

- If there are *multiple alternatives* being selected,
  - the **`if`** that appears within an **`else`** part should be *on the same line* as the **`else`**.
  - Example:

```
if (operator == '+') {
  result = x + y;
  add_op++;
} else if (operator == '-') {
  result = x - y;
  subtract_op++;
}
```

33

Primitive Data Types

Data Type	Range of Values (Intel x86)
`short`[1]	-32,768 through 32,787
`unsigned short`	0 through 65,535
`int`	-2,147,483,648 through 2,147,483,647
`unsigned int`, `size_t`	0 through 4,294,967,295
`long`	-2,147,483,648 through 2,147,483,647
`unsigned long`	0 through 4,294,967,295
`float`	Approximately $\pm 10^{-38}$ to 10^{38} with 7 digits of precision
`double`	Approximately $\pm 10^{-308}$ to 10^{308} with 15 digits of precision
`long double`[1]	Approximately $\pm 10^{-4932}$ to 10^{4932} with 18 digits of precision
`char`	The 7-bit ASCII characters
`signed char`	-128 through 127
`unsigned char`	0 through 255
`wchar_t`	The Unicode characters
`bool`	**`true`** or **`false`**

[1] With the Microsoft compiler **`long double`** is the same as **`double`**.

34

The 7-bit ASCII Characters

	0	1	2	3	4	5	6	7
0	Null		Space	0	@	P	`	p
1			!	1	A	Q	a	q
2			"	2	B	R	b	r
3			#	3	C	S	c	s
4			$	4	D	T	d	t
5			%	5	E	U	e	u
6			&	6	F	V	f	v
7	Bell		'	7	G	W	g	w
8	Backspace		(	8	H	X	h	x
9	Tab		)	9	I	Y	i	y
A	Line Feed		*	:	J	Z	j	z
B		Escape	+	;	K	[	k	{
C	Form Feed		,	<	L	\	l	|
D	Return		-	=	M	]	m	}
E			.	>	N	^	n	~
F			/	?	O	_	o	delete

35

Numeric Constants

- `1234` is an `int`
- `1234U` or `1234u` is an `unsigned int`
- `1234L` or `1234l` is a `long`
- `1234UL` or `1234ul` is an `unsigned long`
- `1.234` is a `double`
- `1.234F` or `1.234f` is a `float`
- `1.234L` or `1.234l` is a `long double`.

36

Character Constants

- The form *'c'* is a *character constant*.
- The form *'\xhh'* is a character constant, where **hh** is a *hexadecimal* digit, and hh is between **00** and **7F**.
- The form *'\x'* where *\x* is one of the following is a character constant.

37

Escape Sequences

- The form *\x* is called an escape sequence, it represents one of the following special characters:

\a	bell	\\	backslash
\b	backspace	\?	question mark
\f	formfeed	\'	single quote
\n	newline	\"	double quote
\r	carrige return	*ooo*	octal number
\t	horizontal tab	\x*hh*	hexadidecimal number
\v	vertical tab		

38

String Constants

- The form "*sequence of characters*"

  where sequence of characters does not include '"' is called a *string constant*.
- Note *escape sequences may appear* in the sequence of characters.
- String constants are stored in the computer as *arrays of characters followed by a '\0'.*

39

Operator Precidence

Rank	Operator	Operation	Associativity
1	[]	Array subscript	Left
	()	Function call	
	.	Member access	
	->	Member access	
	++ --	Postfix increment or decrement	
2	++ --	Prefix increment or decrement	
	*	Pointer de-referencing operator	
	&	Address of operator	
	+ -	Unary plus or minus	
	!	Complement	
	~	Bitwise complement	
	new	Object creation	
3	* / %	Multiply, divide, remainder	
4	+ -	Addition, Subtraction	
5	<<	Shift left	
	>>	Shift right	

40

Operator Precidence (2)

Rank	Operator	Operation	Associativity
6	< <=	Less than, Less than or equal	
	> >=	Greater than, Greater than or equal	
7	==	Equal to	
	!	Not equal to	
8	&	Bitwise and	
9	^	Exclusive or	
10	|	Bitwise or	
11	&&	Logical and	
12	||	Logical or	
13	?:	Conditional	
14	=	Assignment	Right
	*= /= &= += -= <<= >>= &= |=	Compound Assignment	

41

Increment and Decrement

- **Prefix:**

  ++x

  x is replaced by x+1, and the value is x+1

  --x

  x is replaced by x-1, and the value is x-1

- **Postfix**

  x++

  x is replaced by x+1, but the value is x

  x--

  x is replaced by x-1, but the value is x

42

Prefix and Postfix Increment (2)

- Assume that ***i*** has the value ***3***.
- Then

  z = ++i;

  would result in both **z** and **i** having the value **4**.
- But

  z = i++;

  would result in z having the value **3** and **i** the value **4**.

43

Automatic Type Conversion

- If the *operands are of different types*, the following rules apply:
  - If either operand is *long double*, convert the other to *long double.*
  - If either operand is *double*, convert the other to *double*.
  - If either operand is *float*, convert the other to *float*.
  - Convert *char* and *short* to *int*
  - If either operand is *long*, convert the other to *long*.

44

Explicit Type Conversion

- An expression of one primitive type can be converted to another primitive type using the form:

  new-type (expression)

- Example
  - If ***i*** is an `int` and ***y*** a *`double`*

    ```
    i = int(y);
    ```

    will convert ***y*** to an ***int*** and store int into **i**.
  - The statement:

    ```
    i = y;
    ```

    will do the same thing, but may result in a *warning message*.

45

C-Style Casts

- You can also perform explicit conversion using the form:

  (new-type) expression

  This form is inherited from the ***C*** programming language and its use is *discouraged in C++* programs.
- *C++ has other type conversion operators* (also called cast operators) for *conversion among user-defined (i.e. Class)* types.
  - These are discussed later, when they are used.

46

The Conditional Operator

- Form:

  boolean-expression ? value1 : value2

  If the *boolean-exression* is true, then the result is *value1* otherwise it is *value2*.
- In most cases the *same effect* can be achieved using the **`if`** statement.

47

Objects, Pointers, References

- An *object* is an area of computer *memory containing data of some kind.*
- The kind of data is determined by the object's *type*.
- A type may be either
  - A *primitive type.*
  - A *user-defined (class) type.*
- For *class types*
  - *Objects may be contained within other objects.*

48

Object Declaration

- Form:

  type-name name;

  type-name name = initial-value;

  type-name name(*argument-list*);

- Example

```
int i;
string s = "Hello";
double x = 5.5;
double y(6.7);
point p(x, y);
```

49

Object Lifetimes

- *Objects are created when they are declared.*
- Objects declared within the *scope of a function* are *destroyed* when the *function is exited.*
- Objects declared in a *block (between { and })* are *destroyed* when the *block is exited*.
- Objects declared *outside* the scope of a *function* (called *global objects*)
  - Are *created before main is called*
  - Are *destroyed after main exits*
- Objects created using the **new** operator ***must be destroyed*** using the **delete** operator.

50

Pointers

- A *pointer is an object* that *refers to another object*.
- A pointer object contains the *memory address* of the object it points to.
- Example:

```
double x = 5.1234;
double *px = &x;
```

51

Pointer Declaration

- Form

  type-name pointer-variable*;

  pointer-variable = &object;

  *type-name *pointer-variable = &object*;

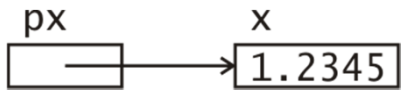
52

The *dereferencing* Operator

- The unary operator * is the *dereferencing* operator.
  - It *converts a pointer to the value pointed to*.
- Example:

```
*px = 1.2345;
```

Results in



53

*T** v versus *T *v*

- Using the form

```
double* px;
```

clearly states that px is of type *pointer-to-***double**.

- Using the form

```
double *px;
```

states that the expression *px is of type **double**, thus px must be a *pointer-to-***double**.

54

Multiple Variables in one Declaration

- The declaration:

```
double* px, py;
```

declares that px is a *pointer-to-double*, but **py** is a *double*.

- To declare *multiple pointer* variables in one declaration:

```
double *px, *py;
```

55

The *NULL* pointer

- The *null* pointer is a pointer value that *points to nothing*.
- Internally the *value* of the *null pointer* is *implementation defined*.
- The literal constant **0** is converted to a *null pointer*.
- Null pointers are converted to **false** when used in *boolean expressions*, and n*on-null* pointers are converted to **true**.
- The macro NULL is defined in <cstddef> as:

```
#define NULL 0
```

- Future versions of C++ will have a reserved-word for the null pointer literal.

56

The *new* operator

- *Pointers* are not generally *initialized* using the address-of operator *(&)*.
- The new operator will create an instance of an object and return the address.

```
double* px = new double;
*px = 5.1234;
```

- `Wrapper classes in Java are immutable.`

57

The *delete* operator

- All objects that are dynamically created using the *new* operator *must be destroyed* using the *delete* operator.

```
delete pointer-variable;
```

- Note that *pointer-variable must have been initialized* by *the new operator.*
- Attempts to use *delete* on some *other pointer* value will probably cause a *run-time error*.

58

References

- Declaring a reference variable is a way to give an object an *(alternate) name*.
- Example:

```
double& rx = x;
```

both x and rx refer to the *same object*.

Note that rx itself is *not an object*. Applying the address operator to *rx* and *x* give the same result.

```
&rx == &x
```

- **References are constants (can't be reassigned to reference other objects) and must be initialized when declared.**

59

Functions

- A function is a *collection of statements* that perform some action and/or compute some value.

```
char min_char(char ch1, char ch2) {
  if (ch1 <= ch2)
    return ch1;
  else
    return ch2;
}
```

60

Function Definition

- Form:

  return-type function-name(parameter list) {

  function body

  }

- The parameter list is either empty, or a comma-separated list of the form:

  type-name parameter-name

- *Function definitions are generally placed in their own file*, or *related function* definitions may be grouped into a single file.

61

Function Declaration

- *To use a function* within a source file it *must be declared*.
- Form *(function prototyping):*

  return-type function-name(parameter list);

  Within the parameter list, only the types of the parameters are required.

```
char min_char(char, char);
```

62

Call by *reference* vs. *value*

- By *default*, functions are *called by value*.
  - A *copy of the arguments* are made and stored into objects corresponding to the *parameters*.
  - Any *changes* made to the *parameter* values *do not affect the original* argument objects.
- If a *parameter* is declared to be a *reference type*, then:
  - The *parameter variable is bound to the argument value*.
  - Any *change* made to the *parameter* value is made to the *original* argument object.

63

Example of *call by reference*

```
void swap(int& x, int& y) {
  int temp = x;
  x = y;
  y = temp;
}
```

- The statement:

```
swap(i, j);
```

will result in the values stored in ***i*** *and* ***j*** *to be exchanged.*

64

Call by *const* reference

- *Class types may occupy several storage locations in memory (static, stack, heap).*
- *Passing a class type object by value is inefficient.*
- By declaring the parameter to be a *const reference*, function *can access the value of the argument, but not change it.*

65

Example of *const* reference

```
int count_occurences(char c, const string& s)
  {
  int count = 0;
  for (int i = 0; i < s.size(); i++) {
    if (c == s[i]) count++;
  }
  return count;
}
```

66

Operator Functions

- C++ allows *class types to behave the same as the primitive types.*
- You can define operators such as +, -, * etc. to operate.
- Example, if *s1* and *s2* are *strings*

  `s1 + s2`

  represents the string consisting of *s1* followed by *s2*.
- The *name of the operator functions* is the form `operator@` where @ represents the *operator*.
- Example:

  `operator+`

  is the *+ operator*.

67

Arrays and C Strings

- An *array is an object*.
- The elements of an array are all of the *same type*.
- The elements of an array are *accessed by an index* applied to the subscript operator.

  array-name[*index*]

68

Declaring an array

- Form:

  type-name *array-name*[*size*];

  type-name *array-name*[] = {*initialization list*};

- Examples:

```
int scores[5];

string names[] = {"Sally", "Jill", "Hal", "Rick"};
```

69

Pointers and Arrays

- C++ performs *automatic conversion between array types and pointer types.*
- *Array name is a constant pointer to the base address of the array.*
- The expression:
  `students[0]`
  and
  `*students`
  are equivalent.
- The expression:
  `a[i]`
  is equivalent to
  `*(a + i)`
  and
  `&a[i]`
  to
  `(a + i)`

70

Dynamically Allocated Arrays

- The *new[]* operator can be used to allocate an *array*.
- Form:

  `new` *type-name*`[`*size*`]`

  will *allocate space* for size objects of type type-name are *return a pointer to the first object.*
- A declaration of the form:

  pointer-variable = `new` *type-name*`[`size`]`;

  will initialize *pointer-variable* to point to the *dynamically allocated array*.
  - The pointer-variable *can then be used like an array variable*.

71

The *delete[]* operator

- All dynamically allocated arrays *must be destroyed* using the *delete[]* operator.
- Form

  `delete[]` *pointer-variable*;

  Note that *pointer-variable* must have been initialized using the *new[]* operator.

72

Arrays as function arguments

- *Arrays are passed as pointers* to function's parameters..
- Function *parameters* may be *declared* either *as pointers or arrays*,
  - but the two are *equivalent*.
- Example:

```
int find(int x[], int n, int target);
int find(int* x, int n, int target);
```

are *equivalent*.

- You can *call* this function with either an *array or a pointer*:

```
int loc = find(scores, 10, 50);
int loc = find(scores + 5, 5, 50);
```

73

C-Strings

- The C programming language uses an *array of char* values *terminated* with the *null character ('\0').*
- Thus the constant

```
char* str = "hello"
```

is stored as:

h	e	l	l	o	\0

char a = str[0] ;

74

The *string* class

- The `string` class is defined in the header `<string>`
- Using the string class allows us to manipulate string objects similar to objects of the primitive types.
- Example:

```
string s1, s2;
s1 = "hello";
s2 = s1 + " world";
```

75

The *string* class

Function	Behavior
Constructors/Destructor	
`string()`	Constructs a default, empty string.
`string(const string& str)` `string(const string& str, size_t pos, size_t n)`	Makes a copy of a string. The second form makes a copy of a substring of the parameter starting at position pos and n characters long.
`string(const char*)`	Constructs a string from a null-terminated array of characters.
`~string()`	Destroys a string.
Assignment Operators	

`string& operator=(const string& str)`
`string& operator=(const char*)`
`string& operator=(const char)`
	Assigns one string to another; assigns a C string to a string; assigns a single character to a string.

Query Operators	
`size_t size()` `size_t length()`	Returns the current size of this string.

76

The *string* class (2)

Function	Behavior
Element Access	

`char& operator[](size_t index)`
`const char& operator[](size_t index) const`
`char& at(size_t index)`
`const char& at(size_t index) const`
	Returns a reference to a character at the specified index of this string. The function `operator[]` allows the use of a string as if it was an array. The function `at` validates the index and indicates an error if it is out of range. The `const` forms are automatically called when the expression is on the right-hand side of an assignment, and the other form is automatically called when the expression is on the left-hand side of an assignment.

const function can not change it's implicit arguments *(this->)*

77

The *string* class (3)

Function	Behavior
Concatenation	

`string& operator+=(const string&)`
`string& operator+=(const char*)`
`string& operator+=(const char)`
	Appends the argument string to this string.

`string operator+(const string&, const& string)`
`string operator+(const string&, const char*)`
`string operator+(const string&, const char)`
`string operator+(const char*, const string&)`
`string operator+(const char, const string&)`
	Creates a new string that is the concatenation of two strings. One of the operands must be a `string` object, but the other one can be a null-terminated character array or a single character.

Search	

`size_t find(const string& str, size_t pos) const`
`size_t find(const char* s, size_t pos) const`
`size_t find(const char c, size_t pos) const`
	Returns the index of where the target first occurs in this string, starting the search at pos. If the target is not found the value `string::npos` is returned.

78

The *string* class (4)

Function	Behavior
Search	

`size_t rfind(const string& str, size_t pos) const`
`size_t rfind(const char* s, size_t pos) const`
`size_t rfind(const char, size_t pos) const`
	Returns the index of where the target last occurs in this string, starting the search at pos. If the target is not found, the value `string::npos` is returned

`size_t find_first_of(const string& str, size_t pos = 0) const`
`size_t find_first_of(const char* s, size_t pos = 0) const`
`size_t find_first_of(const char* s, size_t pos, size_t n) const`
`size_t find_first_of(char c, size_t pos) const`
	Returns the index of where any character in the target first occurs in this string, starting the search at `pos`. If such a character is not found, the value `string::npos` is returned. If the parameter `pos` is not specified, 0 is the default. The parameter `n` indicates the length of the character array pointed to by `s`. If it is omitted, the character array `s` is null terminated.

The *string* class (5)

Function	Behavior
Search	

`size_t find_first_not_of(const string& str, size_t pos = 0) const`
`size_t find_first_not_of(const char* s, size_t pos = 0);`
`size_t find_first_not_of(const char* s, size_t pos, size_t n)const`
`find first_not_of(char c, size_t pos = 0) const`
	Returns the index of where any character that is not in the target first occurs in this string, starting the search at `pos`. If such a character is not found, the value `string::npos` is returned. If the parameter `pos` is not specified, 0 is the default. The parameter `n` indicates the length of the character array pointed to by s. If it is omitted, the character array s is null terminated.

80

The *string* class (6)

Function	Behavior
Substring	
`string substr(size_t pos, size_t n = string::npos) const`	Returns a copy of the substring of this string starting at `pos` that is `n` characters long. If the parameter `n` is omitted, the characters starting at `pos` through the end of the string are used.
Comparisons	
`size_t compare(const string& other)`	Returns 0 if other is equal to this string, –1 if this string is less than the other string, and +1 if this string is greater than the other string.

`bool operator==(const string&, const string&)`
`bool operator!=(const string&, const string&)`
`bool operator<(const string&, const string&)`
`bool operator<=(const string&, const string&)`
`bool operator>=(const string&, const string&)`
`bool operator>(const string&, const string&)`
	All of the infix comparisons are defined.

81

The *string* class (7)

Function	Behavior
Conversion to C String	
`const char* c_str()`	Returns a pointer to a null-terminated array of characters that contain the same data as this string.
`const char* data()`	Returns a pointer to an array of characters that contain the data contained in this string. This array is at least `size()` characters long and is not necessarily null-terminated.

82

Using *find* versus *find_first_of*

- The function

  ```
  size_t find(const string& str);
  ```

  searches the string to which it is applied for the *substring* *str*.
- The function

  ```
  size_t find_first_of(const string& str);
  ```

  treats *str* as a *set of characters*, and will *find* the first occurrence of an *member* of *str* within the string to which it is applied.

83

Splitting a string into tokens

- We can use *find_first_of* and *find_first_not_of* to split a string into *tokens.*
- A *token* is a subset of a string that is separated by *characters* designated as *delimiters*.
  - For example a word in *English* is delimited by the *space character*, or by a *punctuation* character **(. , ! ?).**
- Assume that the string *line* has the value

  ```
  string line = "Look! Look!" ;
  ```

  the function call

  ```
  size_t start = line.find_first_not_of(".,!? ");
  ```

  will set *start* to *0*.
- The function call

  ```
  size_t end = line.find_first_of(".,!? ", start);
  ```

  will set `end` to 4, the index of the *first !* after position start.
- The function call

  ```
  string word = line.substr(start, end - start);
  ```

  will then set `word` to *"Look".*

84

Input/Output Streams

- An *input stream* is a *sequence of characters*.
  - They may be from the *keyboard*, a *file*, or some *other* data source (e.g. a network socket).
- An *output stream* is a *sequence of characters*.
  - They may be *written* to the *console*, a *file*, or some *other* data source (e.g. a network socket).

85

The *<iostream>* header

- The header `<iostream>` *declares* the following *pre-defined streams* as global variables:

```
istream cin;  //input from standard input
ostream cout; //output to standard output
ostream cerr; //output to the standard error
```

- *Standard input* is generally from the *keyboard*, but may be assigned to be from a *file*.
- *Standard output* and standard *error* are generally to the *console*, but may be assigned to a *file*.

86

The *istream* class

- The *istream* class performs *input from input streams*.
- It defines the extraction operator (>>) for the *primitive* types and the *string* class.

Type of operand	Processing
`char`	The first non-space character is read.
`string`	Starting with the first non-space character, characters are read up to the next space.

`int`
`short`
`long`
	If the first non-space character is a digit (or + or -), characters are read until a non-digit is encountered. The sequence of digits is then converted to an integer value of the specified type.

`float`
`double`
`long double`
	If the first non-space character is a digit (or + or -), characters are read as long as they match the syntax of a floating-point literal. The sequence of characters is then converted to a floating-point value of the specified type.

87

Status Reporting Functions

Member Function	Behavior
`bool eof() const`	Returns **`true`** if there is no more data available from the input stream, and there was an attempt to read past the end.
`bool fail() const`	Returns **`true`** if the input data did not match the expected format, or if there is an unrecoverable error.
`bool bad() const`	Returns **`true`** if there is an unrecoverable error.
`bool operator!() const`	Returns `fail()`. This function allows the `istream` variable to be used directly as a logical variable.
`operator void*() const`	Returns a null pointer if `fail()` is `true`, otherwise returns a non-null pointer. This function allows the use of an `istream` variable as a logical variable.

88

Reading all input from a *stream*

```
int n = 0;
int sum = 0;
int i;
while (cin >> i) {
  n++;
  sum += i;
}
if (cin.eof()) {
  cout << "End of file reached\n";
  cout << "You entered " << n << numbers\n";
  cout << "The sum is " << sum << endl;
} else if (cin.bad()) {
  cout << "Unrecoverable i/o error\n";
} else {
  cout << "The last entry was not a valid number\n";
}
```

89

The *ostream* class

- The *ostream* class provides output to an output stream.
- It defines the *insertion operator (<<)* for *primitive* types and the *string* class.

Type of operand	Processing
`char`	The character is output.
`string`	The sequence of characters in the string is output.

`int`
`short`
`long`
	The integer value is converted to decimal and the characters are output. Leading zeros are not output unless the value is zero, in which case a single 0 is output. If the value is negative, the output is preceded by a minus sign.

`float`
`double`
`long double`
	The floating-point value is converted to a decimal representation and output. By default a maximum of six digits is output. If the absolute value is between 10^{-4} and 10^{6}, the output is in fixed format; otherwise it is in scientific format.

90

Formatting Manipulators in *<iostream>*

Manipulator	Default	Behavior
`noshowpoint`	yes	If a floating-point value is a whole number, the decimal point is not shown.
`showpoint`	no	The decimal point is always shown for floating-point output.
`skipws`	yes	Sets the format flag so that on input white space (space, newline, or tab) characters are skipped.
`noskipws`	no	Sets the format flag so that in input white space (space, newline, or tab) characters are read.
`right`	yes	On output, the value is right-justified.
`left`	no	On output, the value is left-justified.
`dec`	yes	The input/output is in base 10.
`hex`	no	The input/output is in base 16.
`fixed`	no	Floating-point output is in fixed format
`scientific`	no	Floating-point output is in scientific format.
`ws`	no	On input, whitespace is skipped. This is a one-time operation and does not clear the format flag.
`endl`	no	On output, a newline character is written and the output buffer is flushed.

91

I/O Manipulators in *<iomanip>*

Manipulator	Behavior
`setw(size_t)`	Sets the minimum width of the next output. After this the minimum width is reset to the default value of 0.
`setprecision(size_t)`	Sets the precision. Depending on the output format, the precision is either the total number of digits (scientific) or the number of fraction digits (fixed). The default is 6.
`setfill(char)`	Sets the fill character. The default is the space.
`resetiosflags(ios_base::fmtflags)`	Clears the format flags set in the parameter.
`setiosflags(ios_base::fmtflags)`	Sets the format flags set in the parameter.

92

Floating-point output format

- The *default* floating-point format is called *general*.
- If you set either *fixed* or *scientific*, then to *get back to general* format you must use the manipulator call:

```
resetiosflags(ios_base::fixed | ios_base::scientific)
```

Format	Example	Description
Fixed	`123.456789`	Output is of the form `ddd.ffffff` where the number of digits following the decimal point is specified by the precision.
Scientific	`1.2345678e+002`	Output is of the form `d.fffff±ennn` where the number of digits following the decimal point is controlled by the value of precision. (On some systems only two digits for the exponent are displayed.)

General	`1.23456e+006`
	`1234567`
	`123.4567`
	`1.234567e-005`
		A combination of fixed and scientific. If the absolute value is between 10^{-4} and 10^{6}, output is in fixed format; otherwise it is in scientific format. The number of significant digits is controlled by the value of precision.

93

File Streams

- The header <fstream> defines the classes

`ifstream` An `istream` associated with a file

`ofstream` An `ostream` associated with a file

94

Constructors and the open function

Function	Behavior
`ifstream()`	Constructs an `ifstream` that is not associated with a file.
`ifstream(const char* file_name, ios_base::openmode mode = ios_base::in)`	Constructs an `ifstream` that is associated with the named file. By default, the file is opened for input.
`ofstream()`	Constructs an `ofstream` that is not associated with a file.
`ofstream(const char* file_name, ios_base::openmode mode = ios_base::out)`	Constructs an `ofstream` that is associated with the named file. By default, the file is opened for output.
`void open(const char* file_name, ios_base::openmode)`	Associated an `ifstream` or and `ofstream` with the named file and sets the openmode to the specified value.

95

Openmode Flags

openmode	Meaning
`in`	The file is opened for input.
`out`	The file is opened for output.
`binary`	No translation is made between internal and external character representation.
`trunc`	The existing file is discarded and a new file is written. This is the default and applies only to output.
`app`	Data is appended to the existing file. Applies only to output.

96

String Streams

- Defined in the header <sstream>
- *Associates* an *istream* or *ostream* with a *string object*.

Constructor	Behavior
`explicit istringstream(const string&)`	Constructs an `istringstream` to extract from the given string.
`explicit ostringstream(string&)`	Constructs an `ostringstream` to insert into the given string.
`ostringstream()`	Constructs an `ostringstream` to insert into an internal string.
Member Function	**Result**
`string str() const`	Returns a copy of the string that is the source or destination of the `istringstream` or `ostringstream`.

Explicit – constructor is *not* to be used as a *conversion constructor*

97

Using an *istringstream*

- Assume that the string `person_data` contains:

  `Doe, John 5/15/65`

- We want to split this into `family_name`, `given_name`, `month`, `day`, and `year`.

```
istringstream in(person_data);
in >> family_name >> given_name;
in >> month;            // Read the month
in >> c;                // Skip the / character
in >> day;              // Read the day
in >> c;                // Skip the / character
in >> year;             // Read the year
```

98

Using an *ostringstream*

- We want to *construct* the string `person_data` from the *component* values.

```
ostringstream out;
out << family_name << ", " << given_name << " "
    << month << "/" << day << "/" << year;
string person_data = out.str();
```

99