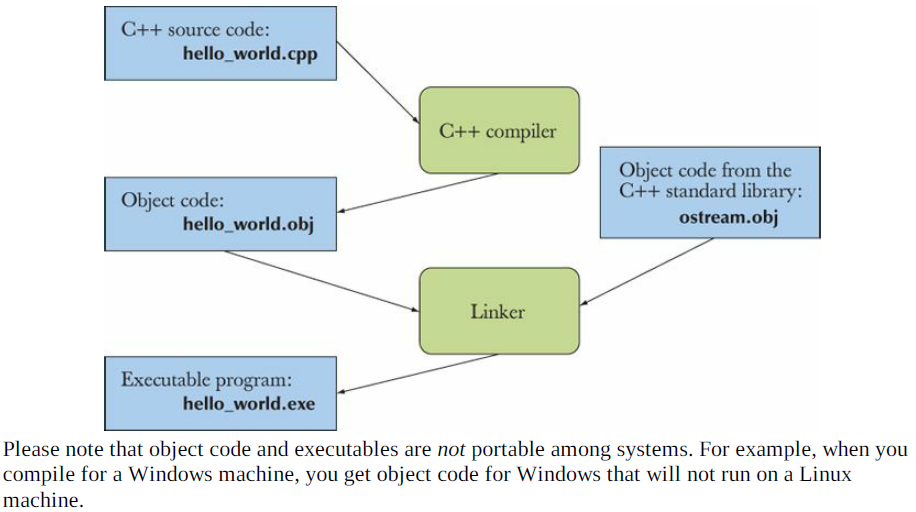
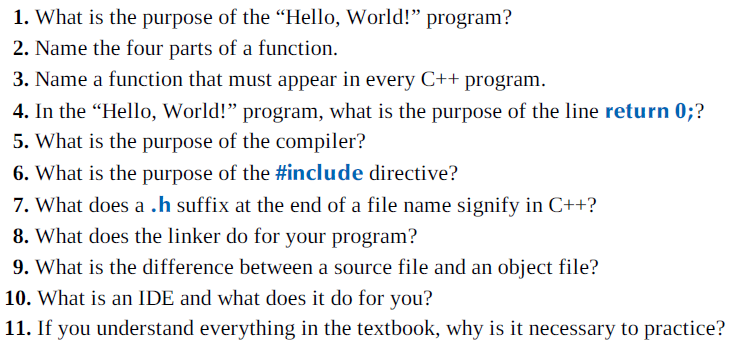
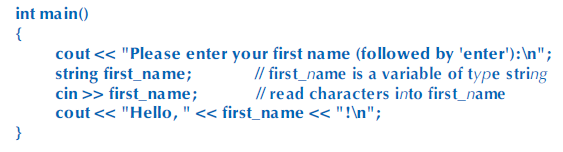
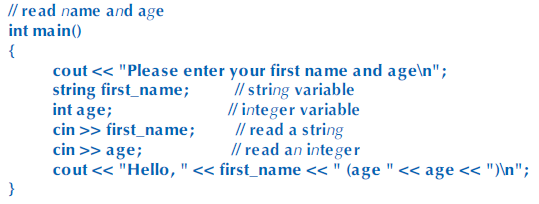
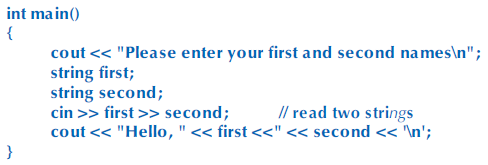
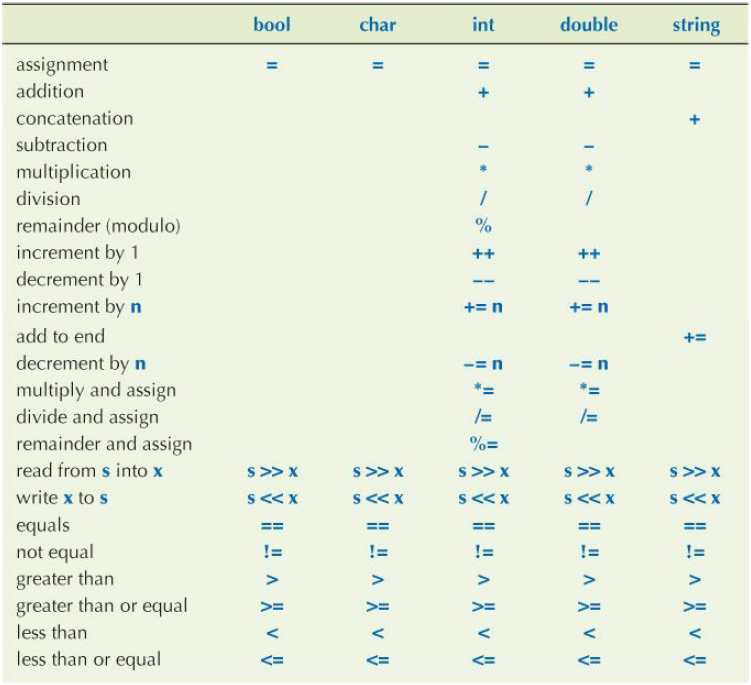
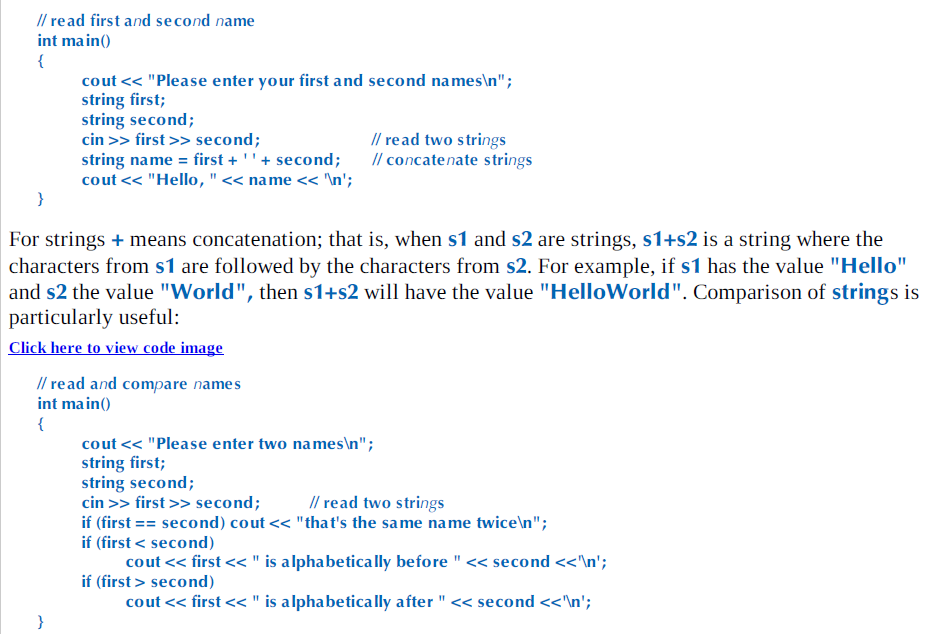
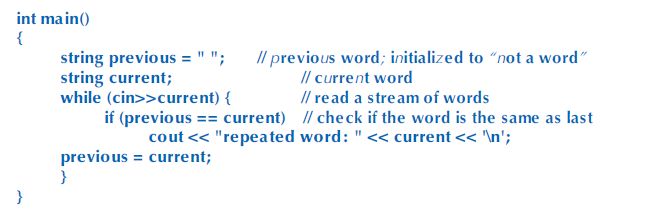
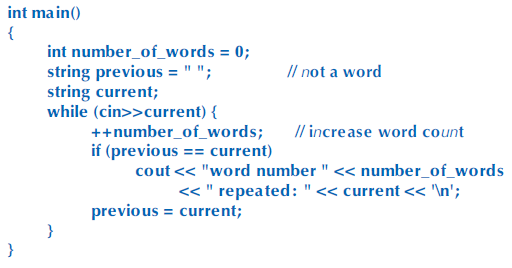
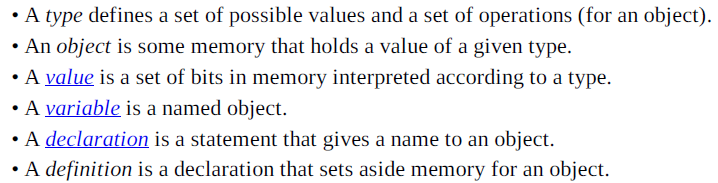
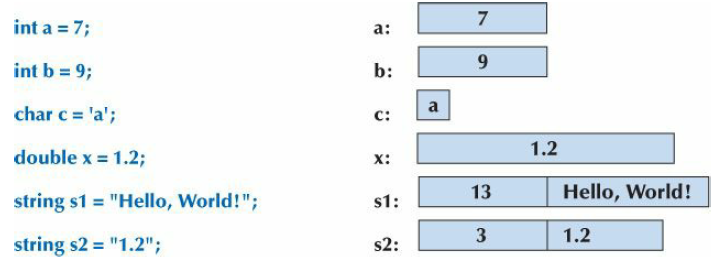
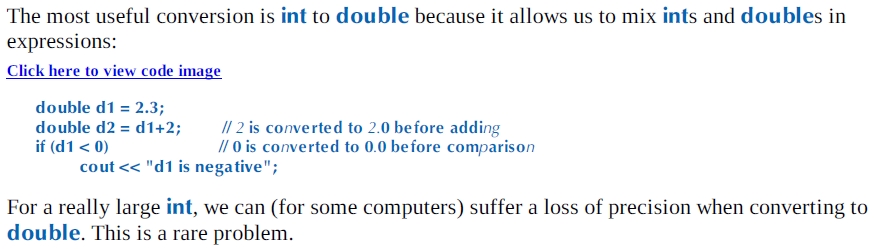
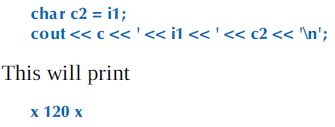
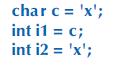
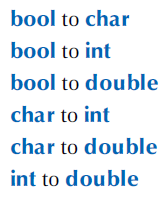
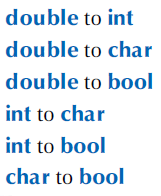
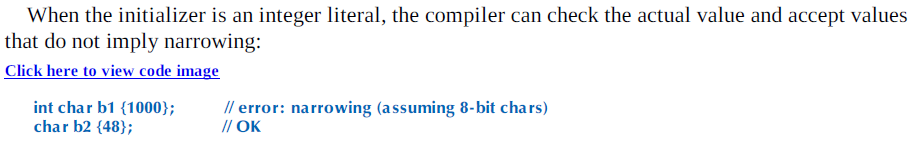
**Chapter 2**C++, String literals are delimited by double quotes.  
\n is special character indicating a new line, cursor moves to the next line.  
cout refers to a standard character output stream, << is output operator.  
#include “std\_lib\_facilities.h” is #include directive, instructs computer to include the C++ standard library.  
This library enables standard C++ stream I/O facilities available.  
File with suffix .h is called header file, header contains definition of terms, such as cout.  
Computer execution of program always starts with the main function.  
function is named sequence of instruction for the computer to execute in the order they are written.  
Function has 4 parts  
1. Return type  
2. Name  
3. Parameter list (parentheses)  
4. Function body (curly braces)  
Statement specifies an action and is not an #include directive or some other preprocessor directive.  
  
C++ is compiled language, translation is done by the compiler.  
Source Code -> C++ Compiler -> Object code (aka Machine code)  
  
Program usually consists of several separate parts, these separate parts must be compiled and the resulting object codes must be linked together to form an executable program. The program that links such parts together is called linker.  
  
  
  
Errors found by compiler are called compile-time errors.  
Errors found by the linker are called link-time errors.  
Errors not found until program is run are called run-time errors or logic errors.  
  
IDE (Interactive Development Environment or Integrated DE) usually includes an editor, color coding, debug, etc.  
Debugging is activity of finding errors in program and removing them.  
  
keep\_window\_open(); <- is needed on some Windows machine to prevent them from closing before reading the output, this is feature of Windows not C++.  
Difference between “\n” and endl; is that endl; flushes out the buffer.  
  
Writing to a file may be “expensive” if the code were to write one byte at a time, so common way is to store the data you are writing into a temporary buffer, postponing the write and writing in a large block, performance is improved.  
  
Flushing the buffer is the act of transferring the data from the buffer to the file, outputting everything in the buffer to the file.  
  
  
  
**Terms**  
  
// << C++ comment compiler compile-time error cout executable function header IDE #include library linker main() object code output program source code statement.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
**Chapter 3**To read something, to take in an input, we need a place in memory, this is an object.  
**Object** is a region of memory with a type that specifies what kind of information can be placed in it.  
**Named object** is called variable, think of object as a box which you can put a value in.  
  
First line (**cout**) is called prompt, it prompts the user to take an action.  
The next lines define a variable of type string called first name, read input from keyboard and write out greeting.  
  
**string** first name; -> sets aside an area of memory for holding a string of characters and gives it the name first name.  
  
Statement that introduces a new name into a program and sets aside memory for variable is called definition.  
  
Cin refers to standard input stream defined in the standard library.  
  
newline will not be the part of the string stored in memory.  
  
 Will give out   
We can do repetition of cout but this is discouraged because repetition provides opportunity for errors, so combine multiple operations into one (rather than 3 cout, 1 cout like in the example above.”  
  
The place we store data are called objects, to access object we need name, a named object is called variable and has specific data type that determines what goes inside the object and which operations can be applied, the data items we put into variables are called values.  
  
  
Common five data type: int, double, char, string, bool. (Each types has its own style of literals…see A2)  
  
Double is short for double precision floating point, floating point is the computer’s approximation to the mathematical concept of real number.   
  
Input operation (>>) is sensitive to type, it reads according to the type of variables you read into.  
  
  
If Carlos 22 is typed, Carlos will first go into first\_name and 22 will go into age.   
It will not store Carlos 22 into first\_name, because reading of strings is terminated by whitespace.  
  
Whitespace are space, newline, tabs. Whitespace is ignired by >>.  
  
If 22 Carlos is typed, 22 will be read into first\_name and since Carlos is not integer, it will not be read by age. The output will give out 22 for first\_name and some random number for age because age did not have an initial value, and you get some “garbage” value that exists in the part of memory you were executing.  
  
String read by using >> is terminated white space, so it reads a single word. There are more ways to read multiple words.  
  
Type of a variable determines what operations can be applied to it and what they mean.  
  
Above are basic operators, many other operations are represented as named functions, in case of **sqrt()** which is from standard library to get the square root. Square root function is not defined for int, so if int is passed as parameter, it will be converted to double inexplicably.  
  
  
  
Assignment is operator that gives a variable a new value (=).  
 **string a =”alpha”; a is alpha  
a = “beta”; a is beta  
string b = a; b is beta  
b = a + “gamma”; a is beta, b is betagamma.  
a = a + “delta”; a is betadelta, b is betagamma.**Initialization (gives variable its initial value)  
Assignment (gives variable a new value)  
Both uses the (=) operator.  
  
Both are logically different, initialization always start with type specification and assignment does not.   
Initialization always finds the variable empty, assignment must clear out the old value from the variable before putting in the new value.  
  
  
  
  
  
  
  
  
**Assignment / Initialization Example**First, string previous is set to none, since it only contains whitespace and >> operator does not read whitespace.  
  
Second, String current is declared  
  
While(cin>>current) works as long as cin>>current succeeds, meaning as long as there are characters to read on the standard input. (Ctrl+Z terminates the program)  
  
if previous == current, we print out the repeated word  
  
previous = current;, sets the current string to previous for the next loop.  
  
First run, since previous is empty string, it will not trigger if statement and will go directly to previous = current where the while loop starts again with loaded previous string and new current string from cin.  
  
When cin >> datatype and user input datatype does not match, the “test” will fail and skip further operations..search more.  
  
Keep in mind that **She** and **she** is not the “same word”, because of capitalization.  
  
Some composite assignment operators are…  
  
  
  
**a oper = b** means **a = a oper b**This increases word count when cin>>current is true, so when user input is entered.  
  
  
In a C++ program, name starts with a letter and contains only letters, digits, and underscores. They are case-sensitive  
C++ Language reserves some names as keywords that cannot be used.  
  
Choose names that will help you understand the code.  
  
Some variables are used conventionally (x and I … etc), do not use overly long names.  
  
Use underscores to separate words in an identifier, do not use all capital letters, that is convention for macro.  
  
Use initial capital letter for types we define, such as Square and Graph. This helps distinguish between user and standard library because standard does not use capital letter.  
  
  
  
  
Above is how we can think of object as a box which we can put values of a given type.  
  
Representation of a string is bit more complicated, string keeps track of the number of characters it holds.  
Double x stores 1.2, where String s2 stores 1, . , and 2.  
  
Every type if of the same size, holds up to  
Int is 4 bytes.  
Bool & Char is 1 byte  
Double is 8 bytes.  
Each type may be differ depending on the content it holds.  
  
Meaning of bits in memory is completely dependent on the type used to access it, think of it as unit notation.  
12.5 means nothing, but 12.5 dollars and 12.5 galloon mean different things.  
For example, 01111000 means 120 in Int but ‘x’ in char.  
  
  
  
Every object is given a type when it is defined, a program is type-safe when objects are used only according to the rules for their type. For example, it is not type-safe to use a variable before it has been initialized. This can give hardware error when uninitialized variable is used.  
  
The ideal is never to use language features that the compiler cannot prove to be safe :Static type safety, but this is restrictive and using unsafe-type leaves code checking to the programmer, not to the compiler.  
  
  
**Safe conversions** **Unsafe conversions**Unsafe conversion means, the value can be implicitly turned into a value of another type that does not equal the original value. This happens when you are trying to put in large byte types into smaller byte types. This is called “Narrowing”  
  
  
C++11 introduced an initialization notation {}, that outlaws narrowing conversions. These should be used in some variable.  
This notation is called universal and uniform initialization.  
   
  
  
**Review Questions**  
1. What is meant by the term prompt?

2. Which operator do you use to read into a variable?

3. If you want the user to input an integer value into your program for a variable named number,

what are two lines of code you could write to ask the user to do it and to input the value into your

program?

4. What is \n called and what purpose does it serve?

5. What terminates input into a string?

6. What terminates input into an integer?

7. How would you write

cout << "Hello, ";

cout << first\_name;

cout << "!\n";

as a single line of code?

8. What is an object?

9. What is a literal?

10. What kinds of literals are there?

11. What is a variable?

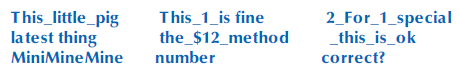
12. What are typical sizes for a char, an int, and a double?

13. What measures do we use for the size of small entities in memory, such as ints and strings?

14. What is the difference between = and ==?

15. What is a definition?

16. What is an initialization and how does it differ from an assignment?

17. What is string concatenation and how do you make it work in C++?  
18. Which of the following are legal names in C++? If a name is not legal, why not?  


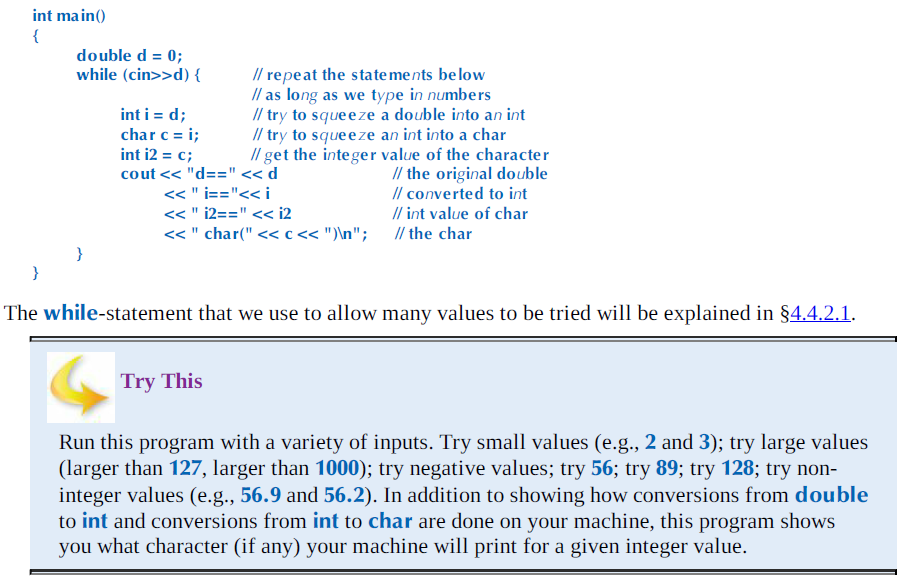
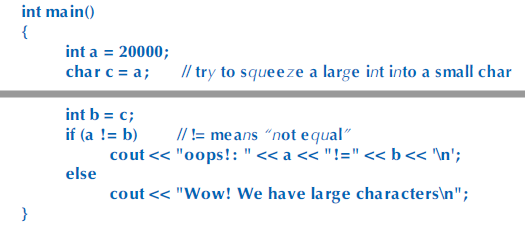
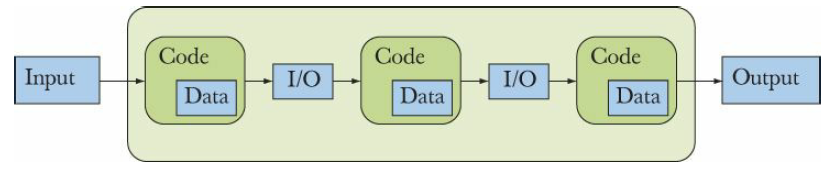
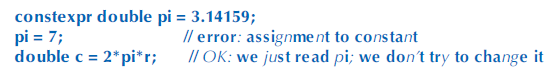
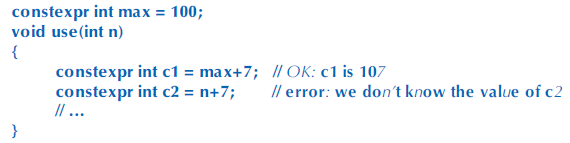
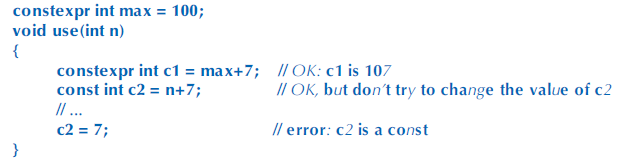
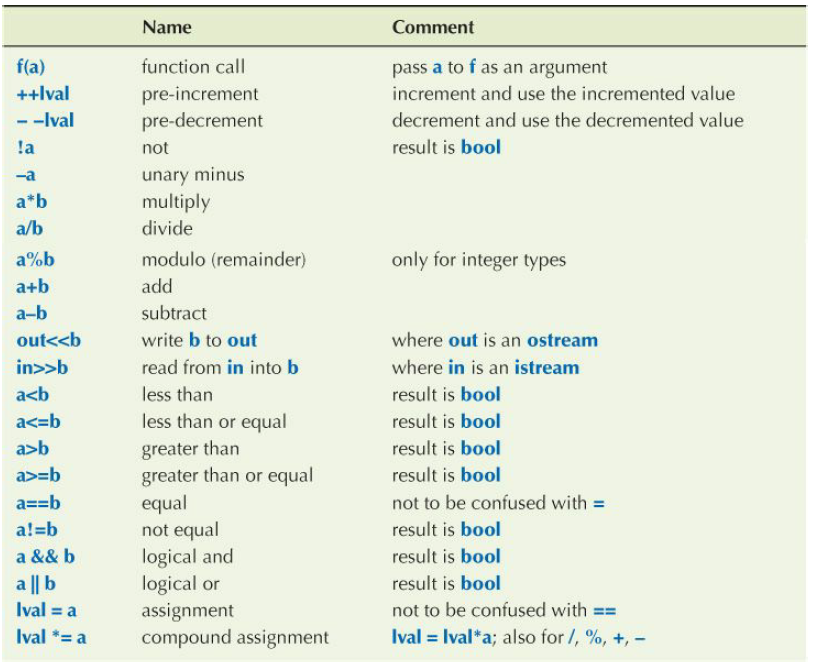
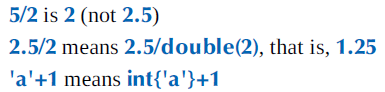
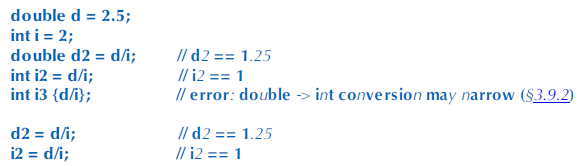
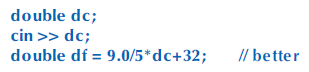
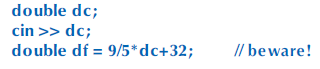
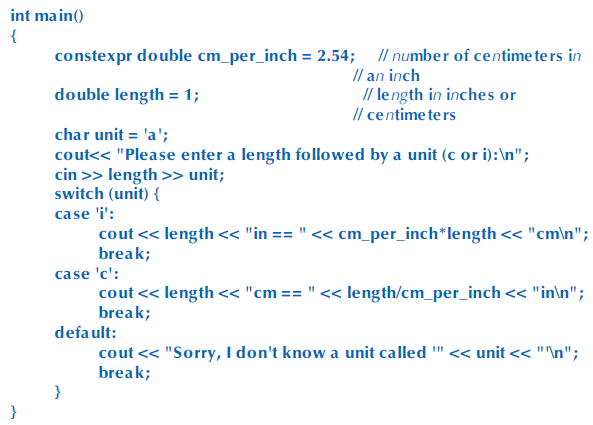
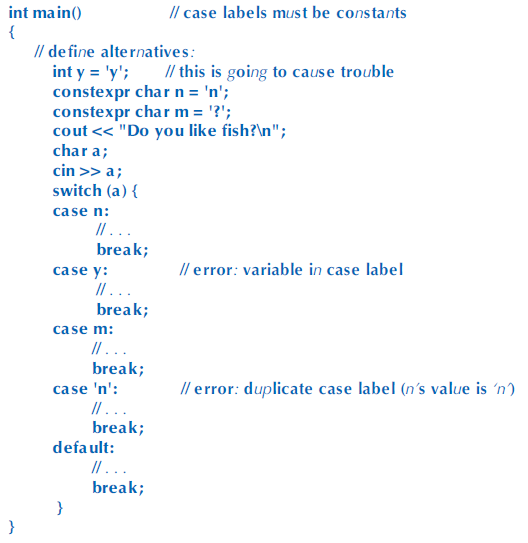
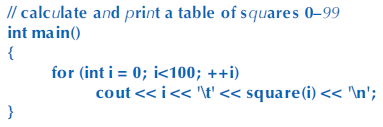
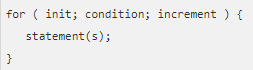
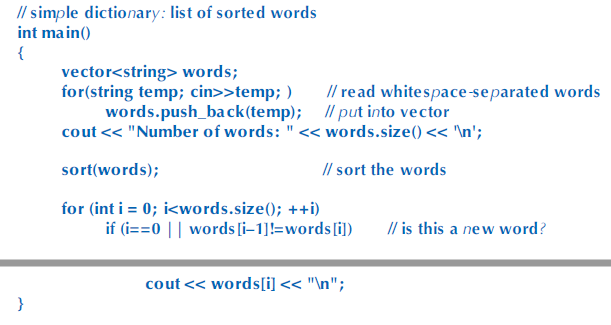
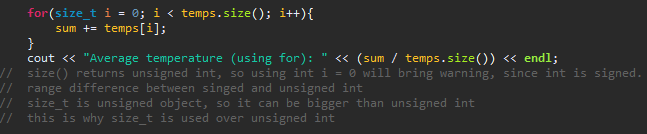
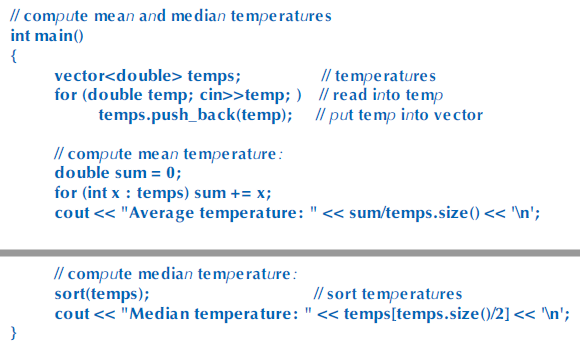
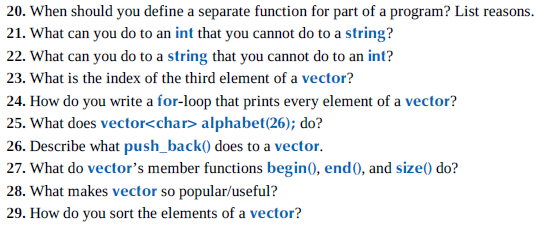
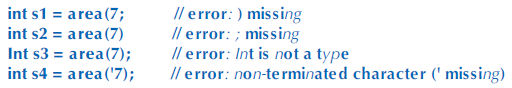
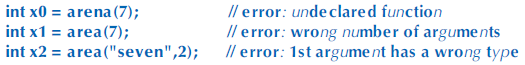
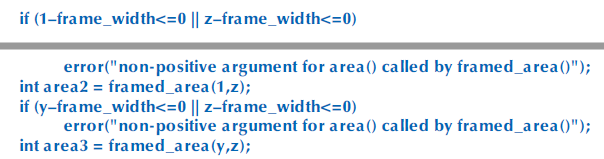
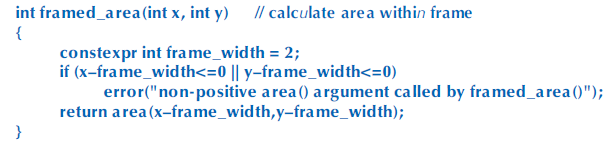
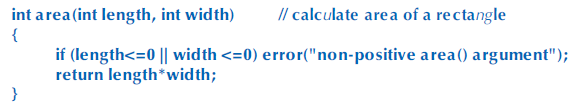
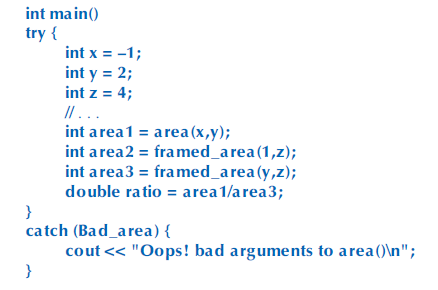
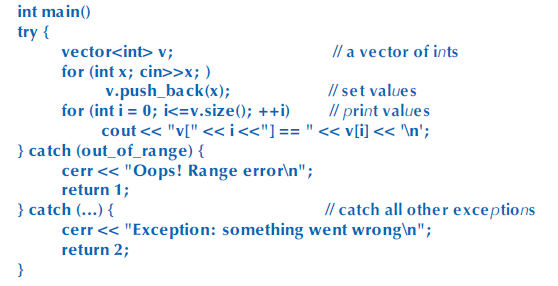
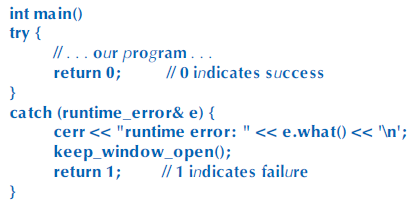
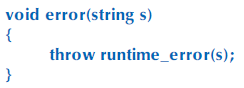
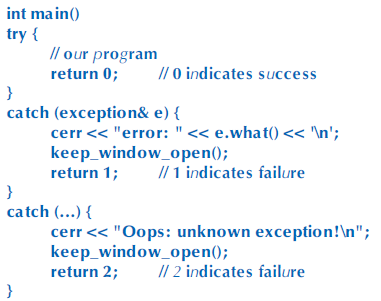
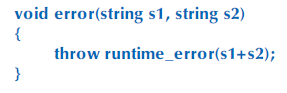
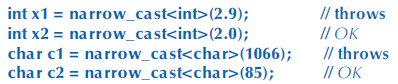
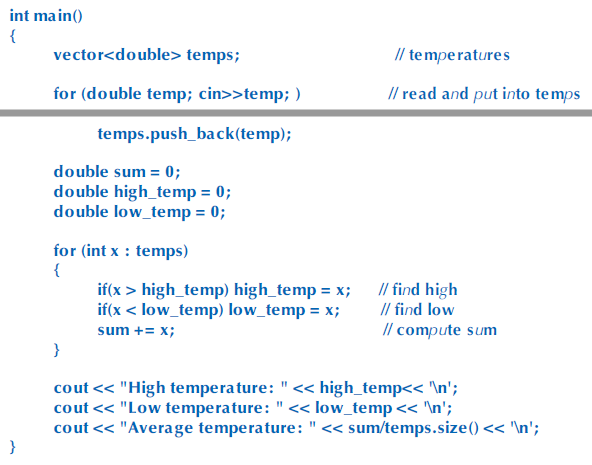
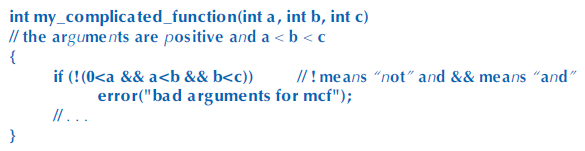
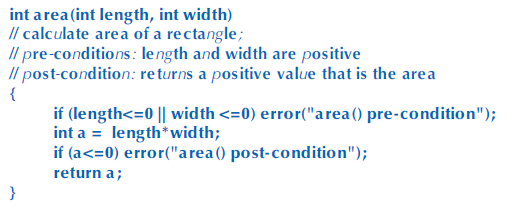
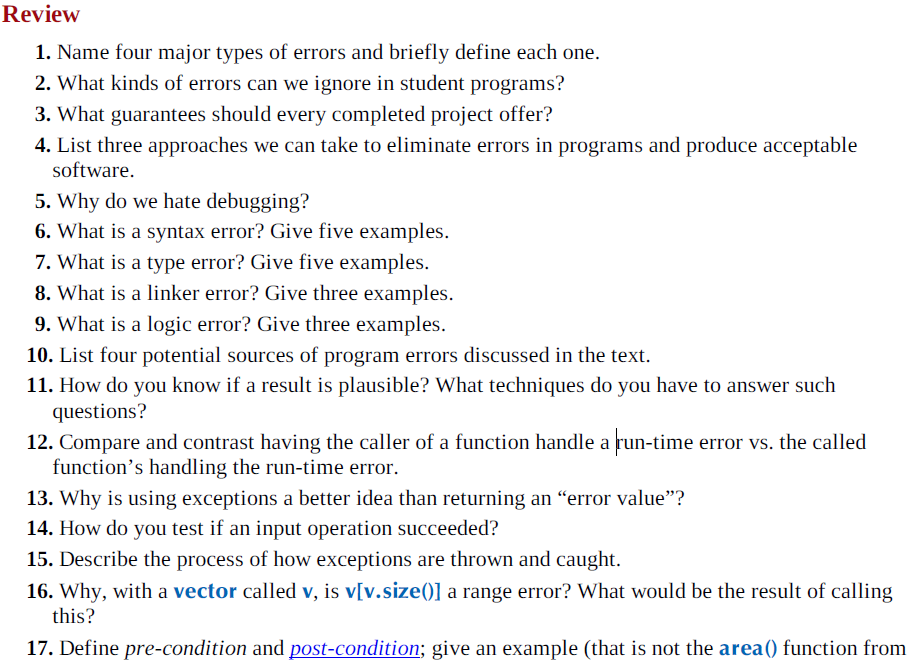
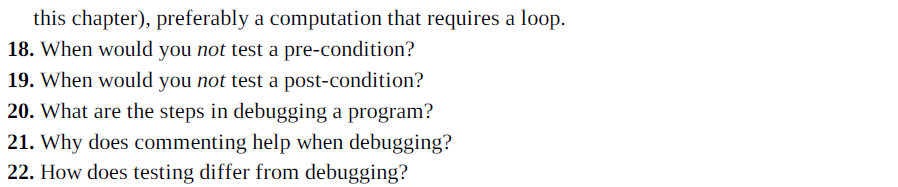
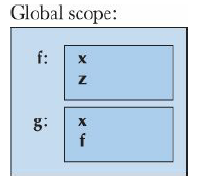
19. Give five examples of legal names that you shouldn’t use because they are likely to cause

confusion.

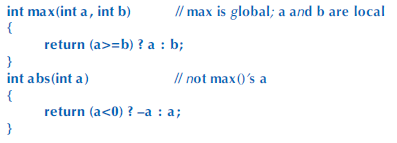
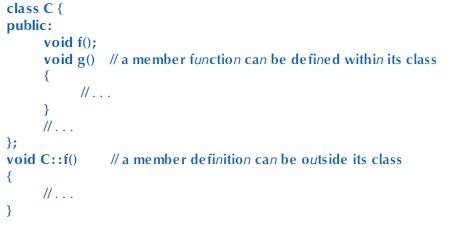
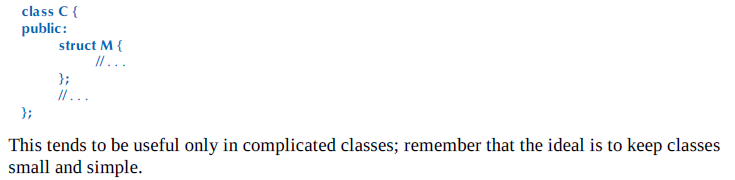
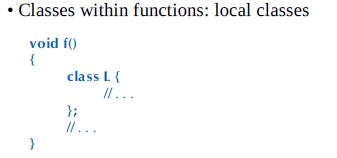
20. What are some good rules for choosing names?

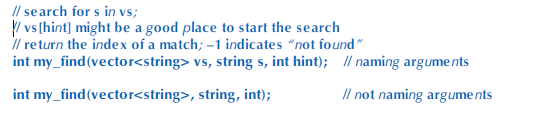
21. What is type safety and why is it important?

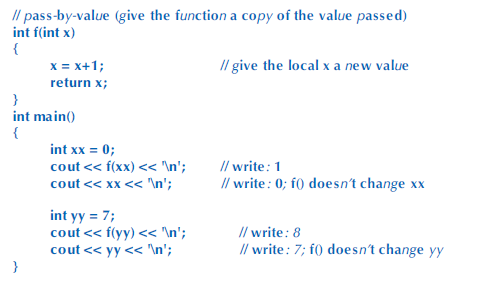
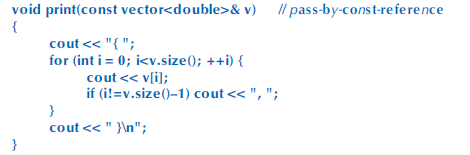
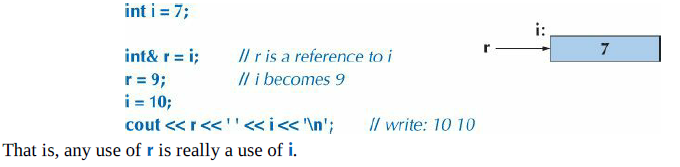
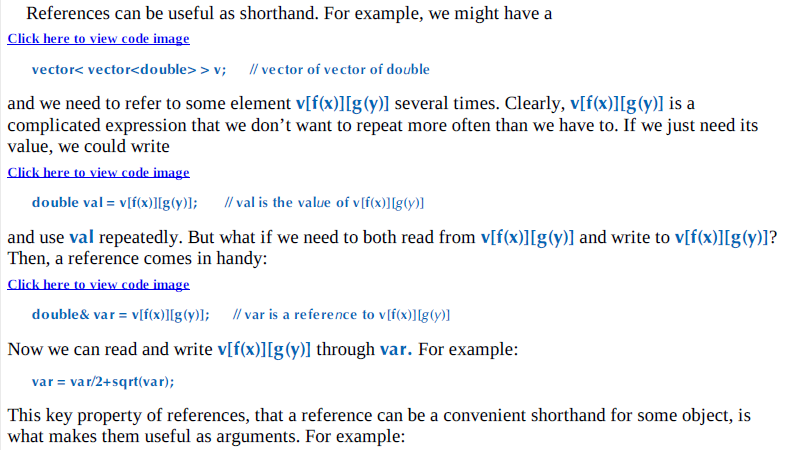
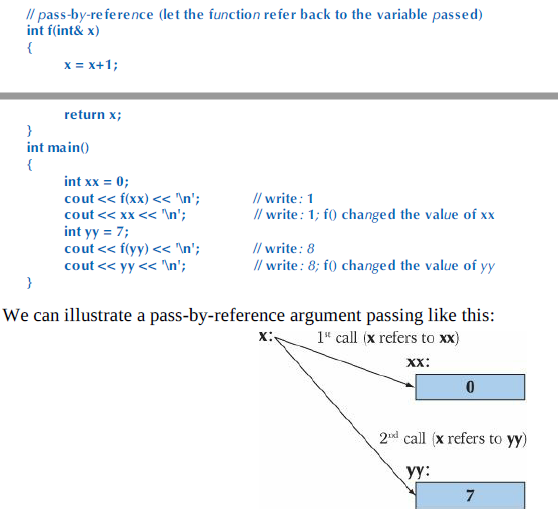
22. Why can conversion from double to int be a bad thing?

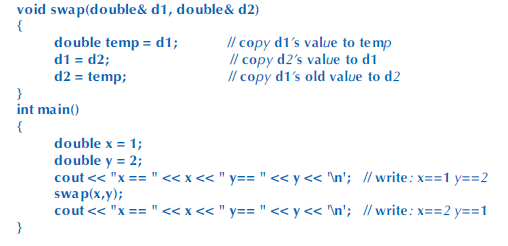
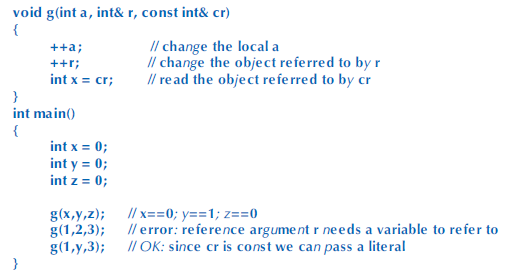
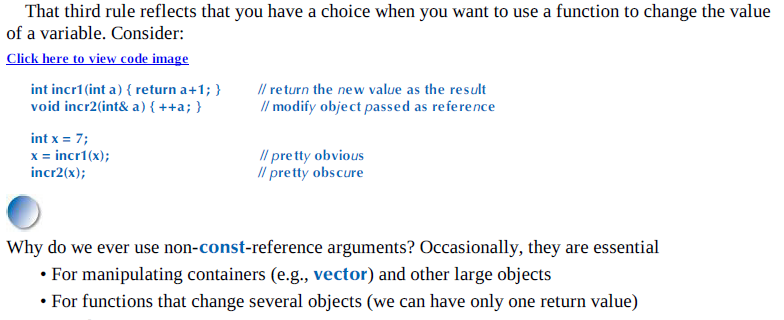
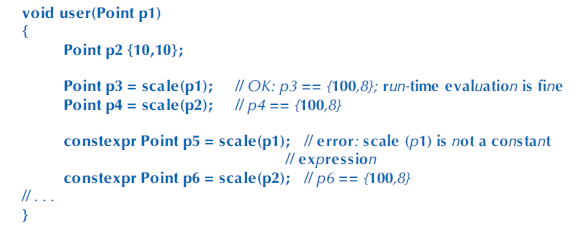
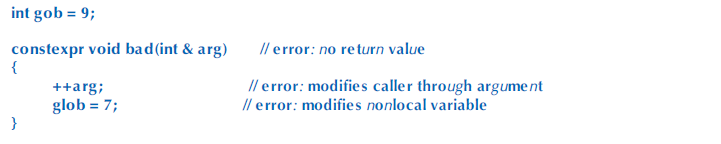
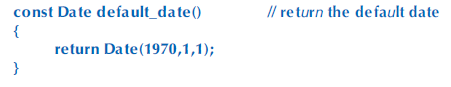
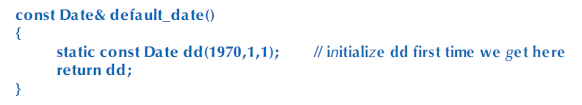
23. Define a rule to help decide if a conversion from one type to another is safe or unsafe.  
  
**Terms**assignment cin concatenation conversion declaration decrement definition  
increment initialization name narrowing object operation operator  
type typesafety value variable  
  
**Extra Amendments**  
  
   
**Chapter 4**  
All program ever does is compute… it takes input and produces an output. Input & Outputs are really infinite.  
In Programming point of view, the most important categories are to/from another program and to/from other parts of a program.   
  
In this case, output from one part of the code is input for the next part. “Parts of program” could be a function producing result from a set of input argument to a function performing action on physical object such as drawing line.  
  
Inputs to a part of a program are often called arguments and outputs are often called results.  
By computation, we mean act of producing some outputs based on some inputs.  
  
Main tool for organizing a program and for organizing out thoughts as we program is to break up big computation into many little ones.  
  
1. **Abstraction:** (**Selective Ignorance):** Hide details that we don’t need to use a facility behind a convenient and general interface. For example, rather than details of how to sort a phone book, we just call the sort algorithm from C++ standard library, All we need to know is how to call that algorithm. Another example is the way we use computer memory, through accessing typed and named variables such as vectors and maps.  
2. **Divide and conquer**: Large problem and divide it into several little ones. If we need to build dictionary, we separate that job into three, reading data, sorting data, outputting data.  
  
Emphasis on structure and organization, efficiency and simplicity of code is important. Take libraries into account when coding, it can crucially affect the way functionality is distributed into different parts of program.  
  
**An Expression** computes a value from a number of operands, the simplest expression is literal value.  
  
Names of variable are also expressions, a variable represents the object of which it is the name.  
Note the difference between **length = 99;**, where LHS is referring to object named length and RHS is referring to the value of the object named length.  
  
In the case of **int length = 20; int width = 40; int area = length \* width;**, literals 20 and 40 are used to initialize variables, and in area calculation, length and width refers to the value found in the object named length and width.  
  
C++ gives symbolic notion for constant variable, a named object that cannot be changed once it has been initialized.  
  
It is preferred to use constants with descriptive name instead of literals, these are called “Magic Constants”  
  
  
  
  
**Constexpr** must be given a value that is known at compile time, **Const** doesn’t have to be known at compile time.  
  
  
**Const** variable existed before **Constexpr**. **Const** is variable that are not constant expressions (value not known at compile) but do not change values after initialization.  
  
**Prefer** ++a over a = a+1, **Prefer** a \*= scale over a = a \* scale, the latter seems like an typo.  
  
**If Operator** has operand of type **Double,** we use floating-point arithmetic.  
  
  
The Notation **type(value)** and **type{value}** means conversion. The latter prevents narrowing (bracket) but the former does not prevent narrowing  
  
  
Expression computes a value from a set of operands using operator.  
Some example of statements (expression statement a = a+b; and declaration statement a = 7;)  
Statements are to have an effect (for example, 1+2 is meaningless, a = 1+2 is meaningful.)  
Thus, expression statements are typically assignments, I/O statements, function calls.  
Wrong use of semicolon can produce empty statements (statement doing nothing), but shows no compiler error.  
  
**Selection – if & switch**if statement – easy..just be sure that if you are doing 2 choice, say ==I and ==c, you need 3 if, two for I and C and else.  
if you only do == I and else, c and everything else will be in one else statement.  
  
  
**Switch**(x), x should be integer, char, or enumeration….it cannot be a string.  
Values in the case label must be constant expressions, variables cannot be used in case label.  
  
To select based on **String**, you have to use if statement or a **map** (chapter 21)  
  
  
  
  
  
**Iteration  
While – statement** contains **loop or control variable (**variable to keep track of how many times its been through the loop**)**, initializer for the loop variable, termination criterion, and the body of the loop.  
  
Sequence of statements delimited by curly braces are called block or compound statement. Empty block is useful for expressing that nothing is to be done.  
  
**For-statement**  **(FOR LOOP INCREMENT CAN BE EMPTY)  
Function** is named sequence of statements, function can return a result.  
Syntax of Function definition is -> type identifier (parameter list) function body  
Parameters could be empty and we can return a result of void (return nothing)  
  
Function defined to make the computation logically separate, make the program text clearer, makes it possible to use the function in more than one place in program and eases testing.  
Function definition – the actual body of the function  
Function declaration – Use of the function without the body.  
  
**Vector** – sequence of elements that can be accessed by an index.  
vector<type> name = { …..}; Example, **vector<int> v = {5,7,9,4,6,8};**  
Vector can also be defined without specifying the element values, **like vector<int> v(6);**  
**size()** function gives the ability to access elements of a vector without referring to an element outside of the range.  
  
**Range – for – loop Traditional – for - loop**  
 **push\_back()** adds new element to a vector at the last index.  
This is a member function of a vector and must be called using dot notation.  
**NAMEOFVECTOR.push\_back(element)  
  
Vector Numeric Example**Very similar… note the if statement for checking multiple  
  
  
  
   
   
**Terms**Abstraction begin() computation conditional statement declaration definition else end() expression for  
function increment input iteration loop lvalue memberfunction output push\_back() repetition rvalue  
selection size() sort() statement switch-statement vector while-statement  
  
  
  
  
  
  
  
  
**Chapter 5**Errors are unavoidable, we must organize software to minimize error, eliminate most error through debugging and testing, and make sure that remaining errors are not serious.  
  
**Sources of Errors**Poor Specification – if not specific about what programs should do, unlikely to examine every possible input.  
Incomplete Programs  
Unexpected arguments – If some is given an argument that does not handle certain type. Sqrt(-1.2)  
Unexpected input – Inputting string to an integer.  
Unexpected state – Examples of states are lists, vectors…if such data is incomplete or wrong.  
Logical Errors – When code is not what it is supposed to do.  
  
Before generating code, compiler analyzes code to detect syntax and type errors.  
  
**Compiler Errors** - First, Syntax error, than type error.  
  
Syntax errors are not always easy to report in a way that programmer can easily understand, it tends to be cryptic.  
  
Type errors report mismatches between types you declared (or did not declare) for variables, functions, etc.  
  
  
  
  
Some “Non – Errors”  
  
 **Link – time errors**Program consists of several separately compiled parts called translation units.  
Every function in a program must be declared with exactly same type in every translation unit in which it is used.  
Header file is used to ensure that, every function must be defined exactly once in program.  
If violated, link time errors occur.  
  
Two Possible Error – Not defined or multiply defined.  
  
The definition of area() must have exactly the same type (both return type and argument types) used in the file…  
  
Functions with the same name but different types will not match and will be ignored.  
  
Misspelled function name is not linker error, it will be compiler since compile – time errors are found earlier than link-time errors.  
  
This linkage rules for function holds for all other entities of a program, such as variables and types. There has to be ONE definition of entity with given name, but there can be many declarations and all have to agree exactly on its type.  
  
  
**Run – time errors**Errors that pass compile and linker, usually hard to detect.  
For example, ratio may look innocent on code, but when ran it may provide with division by 0, which causes runtime.  
This also gives out cryptic message to the USERS of the program who are uneducated.  
  
**Solution 1 : Caller deals with errors**This is hard to read, error-prone, and has to be done for each time we are calling functions  
  
**Solution 2: Callee deals with errors**Argument checking code is in one place (inside function)  
  
**Few key points**Some functions are in library and cannot be changed, maybe it is owned by someone else.  
Called function doesn’t know what to do in case of error, or doesn’t know where it was called from.  
Performance issue, checking implemented in function ups the size and performance.  
  
**Error Reporting**Returning certain value when an error is found (-1, false… etc)  
But, this makes test required for both called function and caller, caller can forget to test, and many functions do not have an extra return value to indicate error.  
  
**Exception**Mechanism to deal with errors, separate detection of an error from handling of an error.  
When a function finds an error, it does not return normally, it throws an exception indicating what went wrong. Any direct or indirect caller can catch the exception, a function expresses interest in exceptions by using a try block listing kinds of exceptions it wants to handle in the catch parts of the try block.  
If no caller catches exception, the program terminates.  
  
  
  
Main Function does not know what threw, and area() does not know what catches, this separation is important.  
  
  
**Ranged Error**Errors that pertain to the range, for example, vector’s indices going beyond what exists. The vector function throws exception that case.  
  
  
  
  
  
  
**Bad Input**condition !cin means previous operation on cin failed.  
runtime\_error provides message string like error().  
  
e.what() extracts error message from runtime\_error.  
cerr is cout for error, it is not optimized so more resilient to error and can be diverted to different target, such as files.  
  
Range Errors is not runtime error, it is compiler error, however both are thrown as exceptions. To deal with both,  
  
  
Typically, two pieces of information is desired when using error(), so we just concatenate the strings.  
  
If exception is not catched, it will get default system error, “uncaught exception”  
  
**Narrowing errors**< > brackets are used to specify a type, called template arguments. Narrow\_cast used when converting a value we are not sure if it is going to fit, so throws exception.  
  
**Logic Error Example**If none of the inputs are lower than zero, the lowest temperature will be 0….  
and if none are higher than zero, the highest temperature will be 0…..  
“Magic Constants” can be set to -1000 and 1000, but they are not so good in programming style….  
  
**Debugging**Make clear the name, purpose, assumptions, formatting, variable name, layout, break code into small functions, avoid complicated code (nested loop), use library  
  
Often, problem is not seen because too much code is being executed, you can temporarily put cerr output statements to see what’s going on  
  
Insert statements that check invariants (conditions that should always hold)  
  
**Pre-Conditions**Precondition comments and precondition checks.  
Some reasons for not pre-conditioning is… Nobody would give bad arguments (in real world, no), It would slow down coding (premature optimization), it is too complicated to check (sort dictionary).  
Overall, writing preconditions are benefit, summarizes and focuses what the purpose of function is.  
  
**Post-Conditions**Same thing on return value.  
  
**Testing**There is no “last bug”, in addition to debugging, systematic testing is also needed.  
Some are done in million test cases and cannot be done by humans, people write tools to approach testing.  
  
  
  
**Terms**argument error assertion catch compile-time error container debugging error exception invariant link-time error logic-error pre/post condition range-error requirement run-time error syntax error testing throw type-error  
  
  
  
  
  
  
**Chapter 8**Declaration is statement that introduces a name into a scope, specifying a type for what is named or specifying an initializer.  
  
Most declarations are found in headers, declaration defines how something can be used, it defines the interface of a function, variable or class, we do not have to know the details of it, we just have to use it.  
  
Definitions are declarations that fully specifies the entity declared.  
  
All definitions are declaration, but some declarations are not definitions. [ int a; vs. int a = 7;]  
  
Definition specifies exactly what a name refers to and it sets aside memory for that variable, so it cannot be defined twice.  
  
Declaration that is not definition is simply an interface that does not allocate memory, so you can declare something endlessly as long as the type declarations are consistent.  
  
**extern** keyboard is used to state that declaration is not a definition.  
  
For a variable, a declaration supplies the type but only definition supplies the object (memory), for a function, declaration provides type (parameter and return), but only definition supplies the function body.  
  
The rule is “Declaration before use”  
  
**Kinds of declaration –** Variables, Constants, Functions, Namespaces, Types, Templates  
  
Declaration of variable specifies name, type, and optionally an initializer.  
Constants have same declaration syntax, but they have **const** as part of their type and **requires** initializer  
  
Initialize variables to save yourself from making bug or error.  
  
{ } initializer syntax is preferred, since it is the most explicit.  
  
For **string** and **vectors**, they come with a default value, vector is empty and string is empty as well. This mechanism is called **default constructor**.   
  
**Header** is collection of declarations, definitions of functions and variables.   
managing declarations of facilities defined “elsewhere” in C++, typically a file of .h  
#include logically happens before compiler, this is preprocessing.  
A header is included in many source files, this means that header should only contain declarations that can be duplicated in several files (such as function declarations, class definitions, definitions of numeric constants)  
  
**Scope** is region of program text.  
  
**Global Scope** : Area of text ouside any other scope  
**Namespace Scope** - names cope nested in the global scope or in another namespace  
**Class scope** - area of text within a class  
**Local scope** - between { } braces of a block or in a function argument list  
**Statement scope** - example, in for statement.  
  
Main purpose is to keep names local, so it doesn't interfere with names declared elsewhere  
  


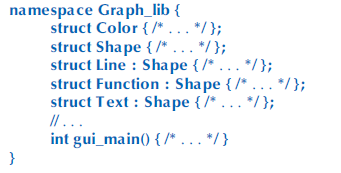
Case is most efficient when done with single case, for example  
  
case ‘0’ : case ‘2’ : case ‘4’ ….  
case ‘1’ : case ‘3’ : case ‘5’ ….  
  
is very inefficient and if should be used for this.  
  
If **break;** is not used, the case will fall down into the next case and produce two output…

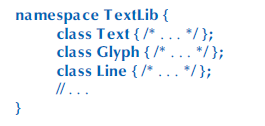
In this example, x is different in both f and g.  
x are local to each f and g.  
  
  
  
Another example, ? : operator is arithmetic if.  
  
The larger the scope of a name is, the longer and more descriptive it should be.  
  
**Note that most c++ constructs that define scopes nest:  
  
Functions within classes: member functions  
**This is most common and useful case.  
  
**Classes within classes : member classes (nested class)  
  
  
Classes within functions : local classes  
  
**Avoid this case, your function is too long **Functions within functions : local functions (nested function)**-This is illegal in C++, will not work

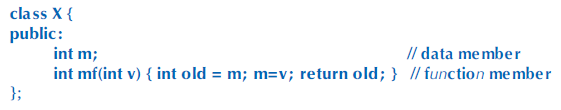
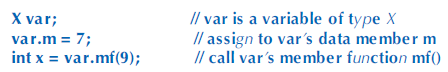
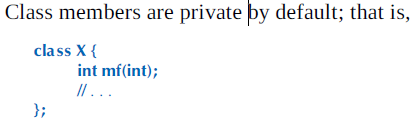
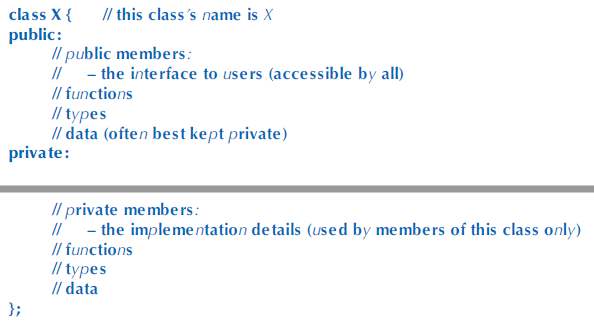
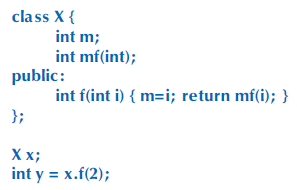
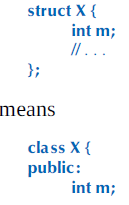
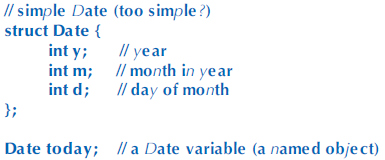
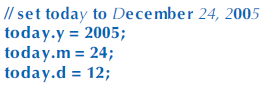
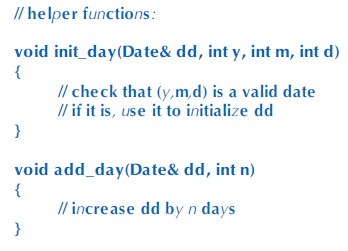
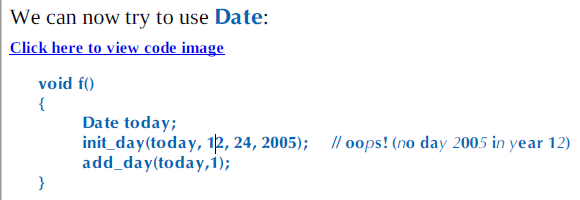
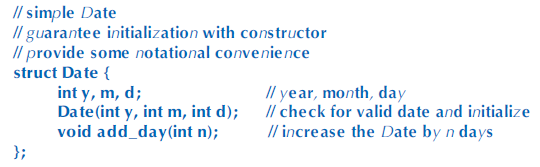
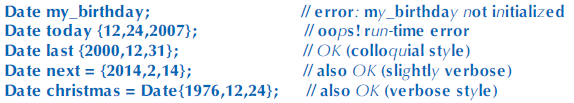
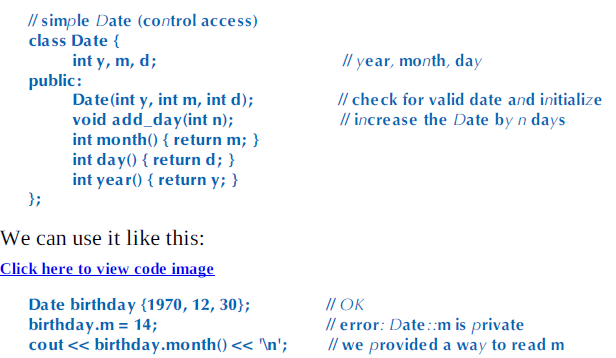
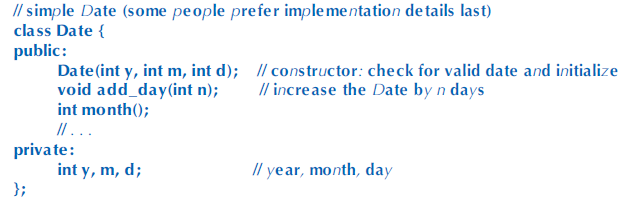
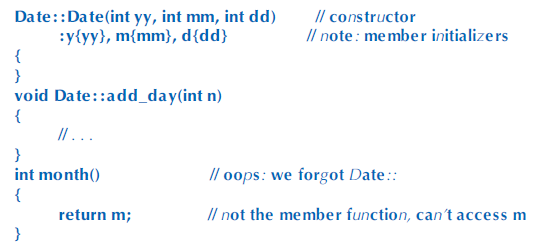
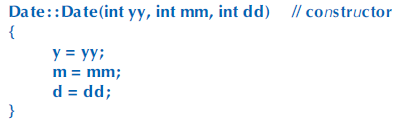
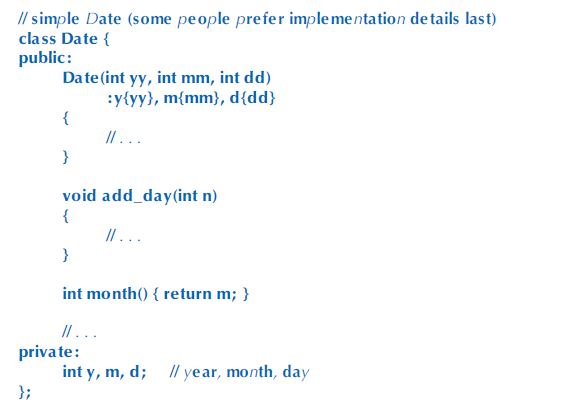
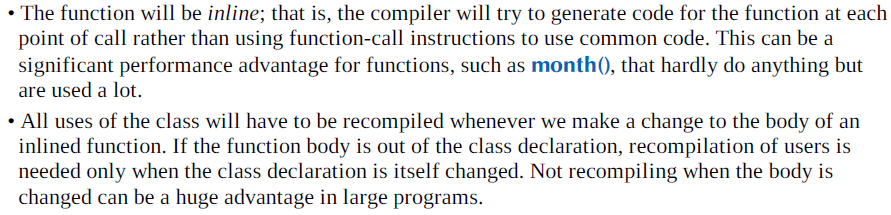
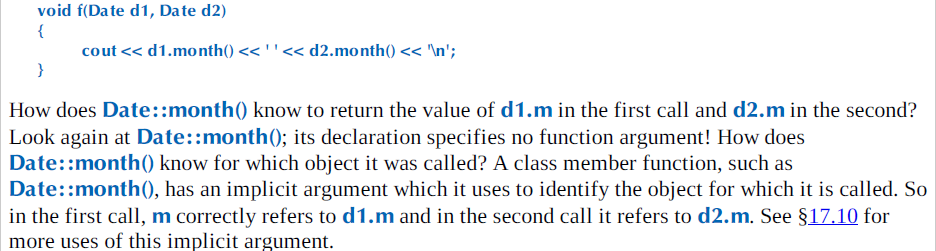
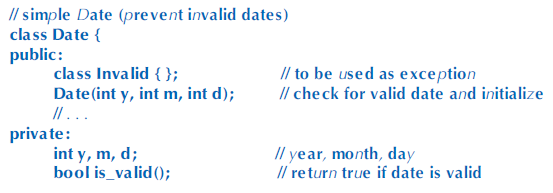
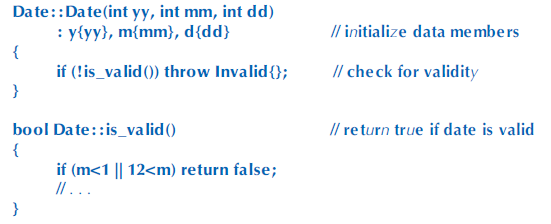
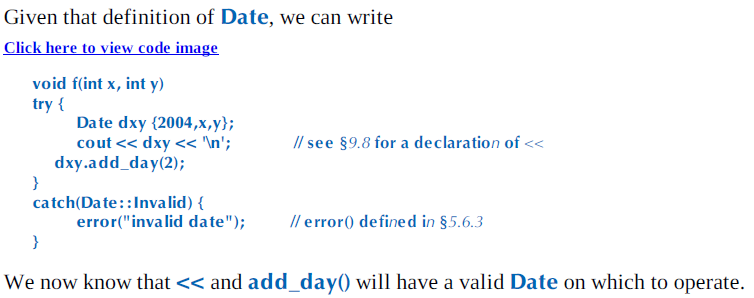
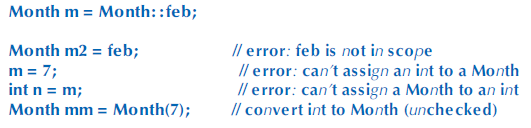
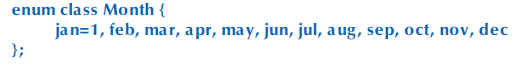
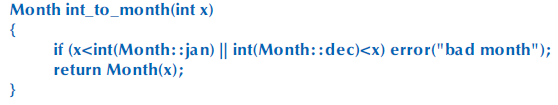
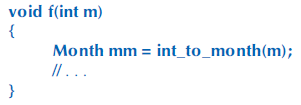
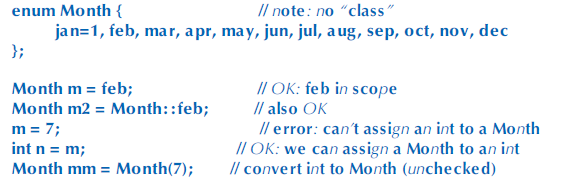
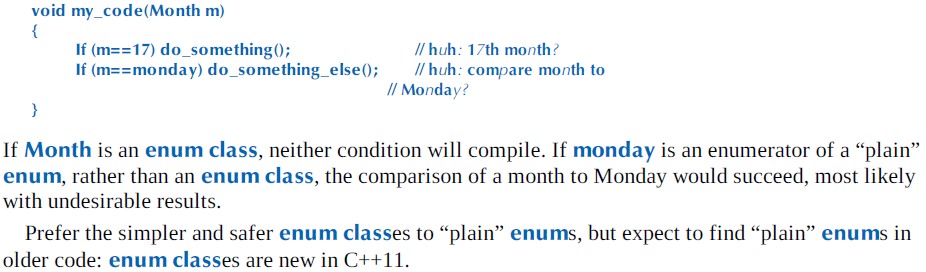
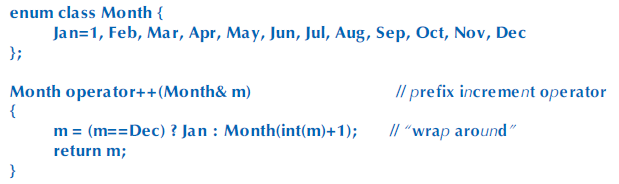
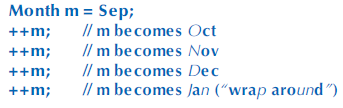
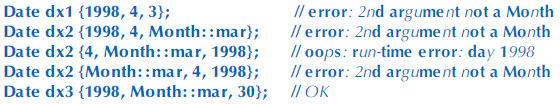
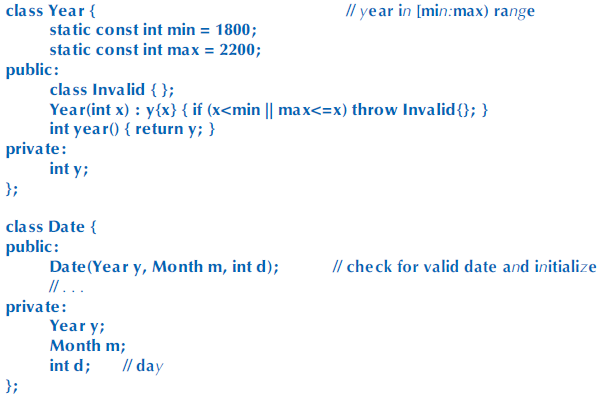
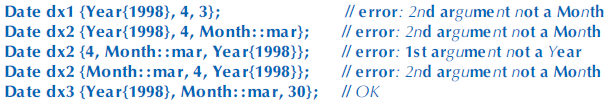
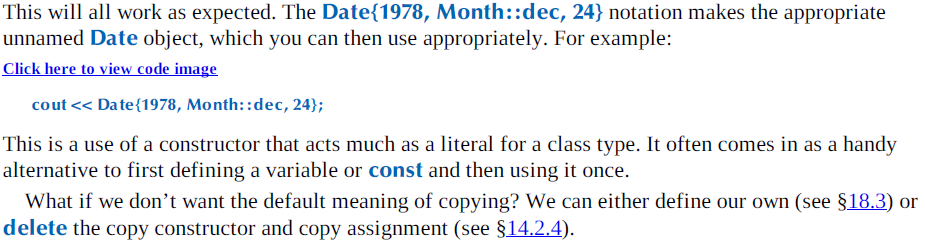
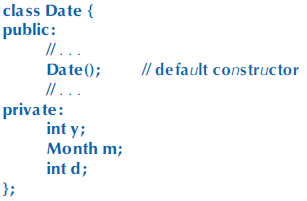
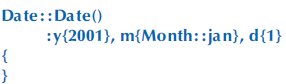
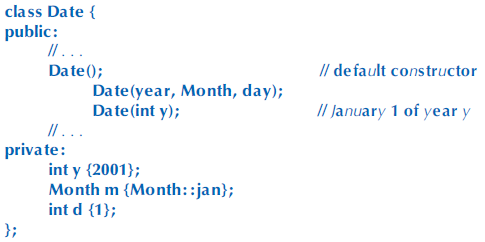
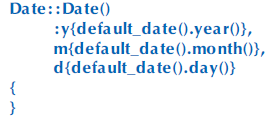
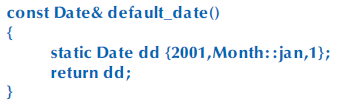
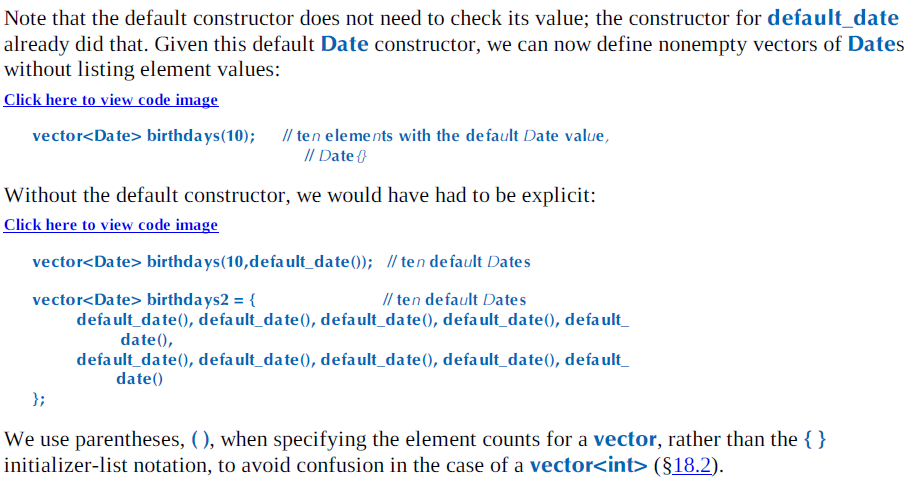
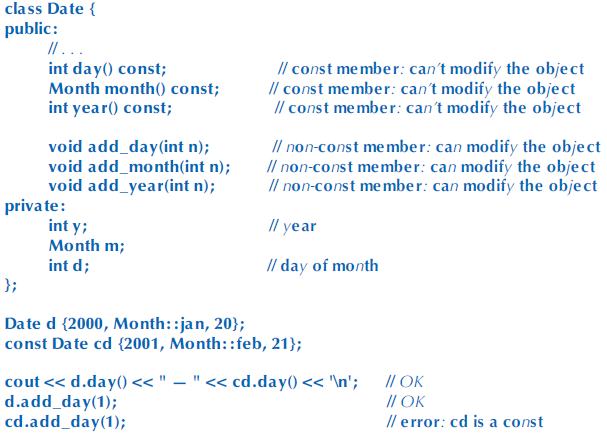
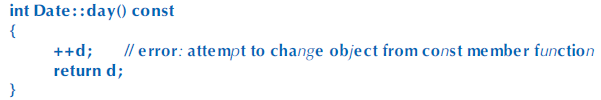
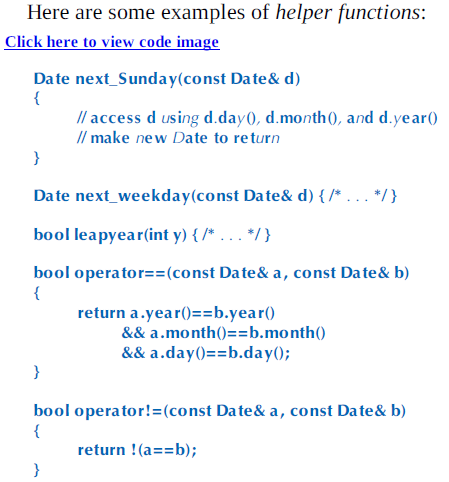
Functions are way to represent actions and computations,   
it is way to organize code made out of these primitives  
  
Functions same, can be declared / defined and its difference holds same  
  
Use () to have no parameters, use void to give no return type.  
  
  
You can name the parameters or not  
  
Declarations, parameter names are logically not necessary  
In Definitions, we name all the parameters.  
  
In Definition, you can leave parameter unnamed if you don't want to use it  
(Since other classes may use the parameter, just declare the parameter but don't use it in body)  
  
For return type, be sure that return statement or error() for every possible ways.  
For example, x>0 and x<0 leaves x==0 empty, so bug occur when return for x==0.  
  
Returning on main() is success when it is 0, even if it "falls through".  
  
In void functions, we can use return without value to cause a return.  
  
Return ends the function at the time it is read, so same as break. Kind of.  
  
you can return void function in void function.

**Pass by Value - gives function a copy of the value you use as parameter  
  
  
  
  
Pass by Const Reference - gives function address of the parameter  
**& means reference and const stops print() from modifying its argument.  
print() refers back to the parameter through reference.  
  
  
  
  
**Pass by reference - Ability to modify that given parameter  
  
  
  
  
  
Refering back to the original example,  
  
**

Pass by reference enables function to operate directly on any object to which we pass a reference.  
  
Another example is swapping two values.  
  
  
C++ provides swap() on standard library so  
  
  
In the last case for const pass by reference, the compiler generates 3.  
These compiler generated objects are called temporary objects.  
  
**Rule of Thumb**Pass by value to pass small objects  
Pass by const ref to pass large object that doesn't need modification  
Return a result rather than modifying an object through reference argument  
Use pass by reference only when have to.  
  
  
  
It is usually best to avoid functions that modify several objects,  
There are alternatives such as returning a class object holding several values.  
If it is using pass by ref, it assumes the function will modify the parameter.  
  
  
  
  
  
  
  
**Constexpr functions**Constexpr function is evaluated by compiler if it is given constant expressions as parameters.  
  
constexpr must be simple so compiler can evaluate it.  
It must have body consisting of single return statement.  
In c++14, it expanded to simple loop.  
constexpr must not have side effects, it may not change value of variables outside of its own body, except those it is assigned to or uses to initialize.  
  
  
**Global variables** are initialized before the first statement of main() is executed.  
They exist until program terminates.  
They are constructed in the order which they are defined and destroyed in reverse order.  
  
variables are constructed every time a loop is ran in the order they were written.  
they live until end of the loop and is destroyed in reverse order.  
  
  
  
Global initializations logically take place before the code in main() is executed.  
  
Global should be avoided, parts of program may not be known to programmers.  
So, it may yield different result depending on which file is initialized first.  
  
  
If we need to use global variable with complicated initializer...  
  
But we are not sure if this is used before being initialized... So,  
  
This constructs at the time function is ran, but can be expensive.  
  
Static local variable constructs the first time its function is called.  
const reference is on return type, no copying and no changing value.  
You should not return reference on local value (no static const) since it refers to garbage value at the end.

We use blocks to organize code within a function  
We use classes to organize functions, data, and types into a type.  
  
They both allow us to define number of entities without worrying about clashes  
They both give name to refer to what we have defined.  
  
To organize classes, functions, data, and types into identifiable name without defining a type, we use grouping mechanism of namespace.  
  


  
To avoid clash, we can do Graph\_lib::Text and TextLib::Text  
This is called fully quantified name, using :: operator.  
  
**using declaration**using std::string // string means std::string  
using std::cout // cout means std::cout  
  
**using directive**using namespace std; // make names from std directly accessible  
so look into std if you don't find declaration for a name in this scope.  
  
Using directive is discouraged except for very common namespace like std.  
Problem is that you will lose track of which names come from where, and may get clashes.  
  
Putting using directive in header file make users can't avoid it, its bad practice.  
  
**Header files** are actual files - stored in the file system, referenced by file name, and #include'd in other files (at least, in C/C++ or other languages using the M4 macro preprocessor). Header files typically group pieces of code that are all interdependent parts of the same specific item together. For instance, a game might have a header file for all of its graphics rendering.

**Namespaces, on the other hand**, are an element of the programming language - they don't exist as a file system object, but rather as a designation within code telling the compiler that certain things are within that namespace. Namespaces typically group interfaces (functions, classes/structs, types) of similar (but not necessarily interdependent) items. For instance, the std namespace in C++ contains all of the Standard Library functions and classes.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
**Chapter 9**  
Built in types, such as char int, and doubles are provided in C++ language, it is built in if the compiler knows how to represent objects of the type and which operation can be done on it without being told by declarations.  
  
**User – Defined – Types (UDTs)** are not built in, they are supplied by programmers.(Standard Library)  
  
**class** (UDT) that specifies how objects of its type are represented, how those objects can be created, how they are used, and how they can be destroyed. If you think of something as a separate entity, it is likely that you should define a class to represent that thing in your program.  
  
Class is building block for large programs.  
  
**Class** is composed of built in types, UDT, and functions.  
Members are parts used to define the class, a class has 0 or more members.  
  
  
Data members define the representation of object of the class  
Function members provide operation on such objects  
  
Dot operators are used to access members.  
  
Note that in mf(), m refers to var.m from the previous line.  
  
**Interface** is part of the class’s declaration that its users access directly.  
**Implementation** is part of class’s declaration that its users access only indirectly through interface.  
public interface is identified by the label **public:**  
implementation is identified by label **private:  
This is typical class declaration**  
   
  
So, X x; int y = x.mf(); will not work, because mf is private, since class X is private by default.  
  
User cannot directly refer to a private member, instead we have to go through a public function that can use it.  
  
Private and public is used to distinct between interface (user’s view) and implementation details (implementer’s view of class), for something that is data, this distinction doesn’t make sense, so there is simplified notation for class that has no private implementation details.  
   
Struct is primarily used for data structures where the members can take any value; that is we can’t define any meaningful invariant. They are also public by default.  
  
**As an example,** we are creating a Date UDT, using struct.  
   
However, this is error-prone, because y m d can be any integer.  
To avoid this, we provide some helper functions   
   
Whenever we define a type, we want some operations for it. Which kinds will vary.  
So initialization function is done for Dates and it checks on the Validity of Dates. However, if we fail to use the function it is useless, for example  
  
**Member functions –** functions declared as members of the class within the class body  
  
Member function with same name as its class is called **constructor**, it is used for construction of objects in the class.  
  
When months or days are changed directly, it is prone to error. As long as we leave representation of Date accessible to everybody, somebody will mess it up. So, we can make it inaccessible (private) to users, like this.  
  
notion of “**valid Date”** is important, so we tried to design our type so that values are guaranteed to be valid, we hide representation, provide constructor, design all member functions to expect valid values and leave only valid values behind when they return.  
  
Value of an object is called **state**, so idea of valid value is referred as **valid state** of an object.  
  
**invariant** something that must be always true at a given point (or points) of a program; typically used to describe the state (set of values) of an object or the state of a loop before entry into the repeated statement.  
  
If we can’t think of good invariant, we are dealing with plain data, so use a **struct.  
  
Date Class Reorganized**By convention, public interface is first, because it is what most people are interested in.  
  
When we are defining member functions outside its class, we need to say which class it is a member of.  
This is done by **class\_name :: member\_name notation** **:y{yy}, m{mm}, d{dd}** notion is how we initialize members, its called member initializer lists. This is more direct.  
  
The first like defines and the notation initializes it at the same time, it is different from below, that this may be used before being initialized.  
  
  
  
  
**Member function defined in the class definition**Class declaration is larger and messier, this is not preferred. Large functions are not defined in class declaration.  
month() can refer to m even though m is defined below month ().  
A member can refer to a function or data member of its class independently of where in the class that other member is declared.   
  
**Writing definition of member function within class definition has three effects**   
Generally, do not put function bodies in the class declaration unless you know it does performance boost from inlining tiny functions. Large functions, 5 or more lines do not benefit from inlining and makes class declaration harder to read.  
  
**To find invalid date,** the obvious place is to look where we first construct a Date.  
  
We put testing of validity into a separate is\_valid() function because checking for validity is logically distinct from initialization and we might want to have several constructors.  
  
Note that we can have private functions as well as private data.  
  
   
  
  
  
  
**enum (enumeration)** is a very simple UDT, specifying set of values as symbolic constants.  
  
The body of enumeration is simply a list of its enumerators, the **class** in **enum class** means that the enumerators are in the scope of the enumeration. So we use :: notation.  
  
Specific value can be set for an enumerator, or the compiler can give each enumerator the value of previous plus one.  
  
If enumerator is not initialized, it will start with 0.  
  
  
A constructor for an enumeration cannot be defined to check initializer values, but a simple function can be written  
  
  
  
  
Enumeration is used whenever we need a set of related named integer constants, this happens when we try to represent sets of alternatives (up, down, yes, no, maybe, on, off) or distinct values (red, blue, green, yellow, maroon, crimson, black)  
  
**There are plain enumerations** that differ from scoped enumerations by exporting their enumerations to the scope of the enumeration.  
  
  
plain **enum**s are less strict than enum classes, but they are prone to pollution. This can lead to unsuspected error, for example **dec** for December can clash with **dec for decimal** in <iostream>  
  
  
  
**Operator Overloading –** define C++ operators for class or enumeration operands.  
  
  
  
You can only overload existing operators, you cannot define existing operators. Also, you can define operators within their conventional number of operands, so you cannot us unary <=, because the number of operand does not match.  
  
Overloaded operator must have at least one UDT as operand.  
  
**Class Interface**Keep interfaces complete  
Keep interfaces minimal  
Provide constructors  
Support copying (or prohibit)  
Use types to provide good argument checking  
Identify non modifying member functions  
Free all resources in the destructor  
  
  
  
  
  
  
  
  
We defined that constructor for Date, we used three int as the argument.  
This may cause problems such as illegal int in inappropriate places.  
We use a month type to avoid this error.  
  
  
If month and day are swapped, the compiler will detect it. It is symbolic and easier to read and less error-prone.  
We use :: after name of a class, enumeration, or namespaces. We use dot operator after an object name.  
  
We can also implement year to catch the swap error of the day and the year.  
  
  
  
However, this does not fix   
**static const** is used for min and max, static to make sure that there is just one copy of the value in the program, rather than one per object of the class. Since initializer is a constant expression, we could have used **constexpr**.  
  
**Copying** – if you copy a class as an initializer or right hand side assignment, all its members are copied.  
  
  
  
  
**Default Constructors**For a type, T T{} is the notation for the default value, as defined by the default constructor.  
  
Without default constructors, the result will be undefined or to random addresses, nothing can be established.  
For types, it is better to define constructor that gives meaning to the creation of an object without an explicit initializer.  
  
   
**Instead** of placing default value for members in the constructor, we can place them on the members themselves.  
Because Date(int) does not explicitly initialize the month or day, specified initializers (Month::jan and 1) are implicitly used  
  
An initializer for class member specified as part of the member declaration is called in-class initializer.  
**Alternative way is to use constant** **Static** used to get variable (dd) created once at the first time default\_date() is called, and we define a default constructor.  
There is no need to check validity, since default\_date already is validated.  
  
  
  
**const member functions** **const** right after argument list in a member function declaration indicate that the function can be called for **const** object.  
Once we declare this, the compiler cannot modify the object, so   
  
 **Members and “Helper Functions”**  
A function that can be implemented as a free standing function (non-member) should be implemented outside the class, so bug in that function cannot directly corrupt the data in class object.  
  
If the representation changes, only the functions that directly access that representation need to be rewritten.  
  
If we decide that integer representing number of days since January 1, 1900 is much better representation than (y,m,d). Only the member functions would have to be changed.  
  
  


Today is not initialized for cout.  
tomorrow is constructed by hand.  
  
cout uninitialized Date will produce garbage output  
Incrementing day by adding 1 will yield invalid date in last day of the month.

This is difference between  
int x; ….. x=2;  
AND  
int x {2};

Helper functions are called convenience functions, auxiliary functions.  
  
Helper function is non-member functions  
  
Helper function is design concept, often take argument of the classes they are helpers of, usually.  
  
Often, namespaces are used to identify a group of helper functions.