# Chapter 5 (5-11)

- data saved to files sustains
- data stored in vars and control properties stored temporarily in RAM
- Three-step file process
  - ↳ open file
  - ↳ process file
  - ↳ close file
- two file types
  - ↳ text: ASCII or Unicode
  - ↳ binary
- sequential vs direct access
  - ↳ random-access file / direct access file
- file stream object associated w/ specific file
- two ways to open files
  - ↳ .open()
  - ↳ ifstream input_file("...txt");
- closing files
  - ↳ temp data stored in file buffer. Closing file saves data in buffer to file.
  - ↳ some OS's limit number of open files
  - ↳ .close()
- read position
  - ↳ initialized to first byte of file
- >> stream operator automatically converts text encoded numeric data to numeric data type.
  - ↳ also works as boolean to detect if data read or not
    - ↳ while(input_file >> number)
- detect successful file open
  - ↳ if(input_file) if successfully opened
  - ↳ if(input_file.fail()) if failed to opened
- string literals are stored in memory as null terminated strings, not string objects
  - ↳ .c_str()

# Chapter 7

- data types and storage reqs
  - ↳ bool (1 byte)
  - ↳ char (1 byte)
  - ↳ short (2 bytes)
  - ↳ int (4 bytes) (assumes: signed int)
    - ↳ unsigned (4 bytes)
  - ↳ long (4 bytes)
  - ↳ long long (8 bytes)
  - ↳ float (4 bytes)
  - ↳ double (8 bytes)
  - ↳ long double (8 - 16 bytes)
  - ↳ string (variable)

built-in primitive data types

- array size declarator
  - ↳ must be constant or literal
- subscript numbers can be stored as variables

- if array defined globally, all elements initialized to zero
  - ↳ locally, no default init. value
    - ↳ instead, will contain garbage data like all other local vars!

- initialization list
- static_cast <datatype> var
- partial initialization, set rest to 0
  - ↳ can't initialize elements randomly, must be in sequence from beginning
- while (count < ARRAY_SIZE && inputFile >> numbers [count])
- range-based for loop
  - ↳ range variable
  - ↳ for (dataType rangeVariable : array)
    statement;
    - ↳ datatype must match data type of array elements
- references vs. pointers
  - ↳ &val
  - ↳ val (actual value)
  - ↳ val (after declaring *val = nullPtr
  - ↳ val * (dereference from address)
- increments
  - ↳ ++score[2] pre-increment
  - ↳ score[2]++ post-increment
- void showValues (int nums[], int size)
  - ↳ pass array via fxn
- vector <int> array (10, 2)
  - ↳ declare size of array
  - ↳ initialized to value of 2
- vector <int> numbers {10, 20, 30, 40}

- Vector method functions
  - ↳ .at(element)
  - ↳ .push_back()
  - ↳ .pop_back()
  - ↳ .reverse()
  - ↳ .swap()
  - ↳ .size()
  - ↳ .resize()
  - ↳ .clear()
  - ↳ .empty()

---

# Chapter 12

- .get(), .put() → optional delimiter
- getline(file, str, '\n')
- Program type: Filter
- .write()
  - ↳ fileObject.write(address, size)
    (&letter, sizeof(letter))
- .read()
  - ↳ fileObject.read(address, size)
    (&letter, sizeof(letter))
- reinterpret_cast<datatype>(value)

```
int x = 65;
char *ptr = nullptr;
```

  - ptr=reinterpret_cast<char*>(&x)
  - fileObject.write(reinterpret_cast<char*>(&x), sizeof(x))
- for structures, always open in ios::binary mode due to mix of data structure types
- sequential vs. random file access
- .seekp(), .seekg()
  - ↳ ios::beg ↳ must use file.clear() else eof flag will prevent functionality
  - ↳ ios::end
  - ↳ ios::cur
  - ↳ file.seekp(20L, ios::beg)
  - ↳ file.seekp(-20L, ios::end)
- .tellg(), .tellp()
  - ↳ helps w/ determining number of bytes in file

| | |
|---|---|
| ios::app | |
| ios::ate | |
| ios::binary | |
| ios::in | |
| ios::out | |
| ios::trunc | |
| ios::eofbit | .eof() |
| ios::failbit | .fail() |
| ios::hardfail | |
| ios::badbit | .bad() |
| ios::goodbit | .good() |
| | .clear() |

---

# Chapter 13

- objects
  - ↳ data + functions
    - attributes + member fxns
    - (data) (procedures)
- encapsulation
- data hiding
- object reusability
- instance of class, instantiation of a class
- access specifiers
- Member functions
  - ↳ void getWidth() const;
- :: → scope resolution operator
- accessor + mutator
  - (getter) (setter)
- object's state
  - ↳ data in object at a given moment
- stale data
- → operator with pointers!

v1

```
class Rectangle...
Rectangle rect_1
Rectangle *rectptr = nullptr
rectptr = new Rectangle
rectptr = &rect_1
rectptr -> setLength(12)
rectptr -> setWidth(7)
delete rectptr
rectptr = nullptr
   or rectPtr
```

v2

```
#include <memory>
unique_ptr<Rectangle> rect_ptr(new Rectangle)
Rectangle rect_2
rect_ptr = &rect_2
rect_ptr -> set_length(8.3)
rect_ptr -> set_width(12.6)
rect_ptr = nullptr
```

→ alt way:

```
auto rect_ptr = make_unique<Rectangle>()
```

- class specification file
  - ↳ contains class declaration
- class implementation file
  - ↳ stores member function defs

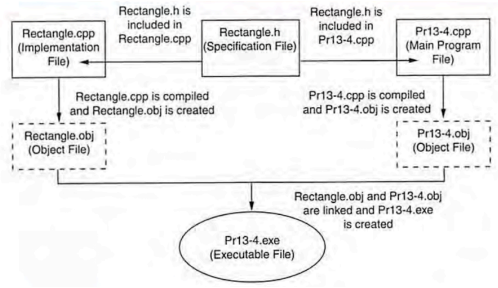
- #ifndef is an include guard
  - ↳ if not defined
- #define (defines macro)
- #endif

} pre-processor directives

## Rectangle Example:

- Rectangle.h
- Rectangle.cpp
- main.cpp

create object files




- inline function: double getArea() const { return width * length }
- inline expansion
  - ↳ improved performance (generally)

**Contents of Rectangle.h (Version 1)**

```
1  // Specification file for the Rectangle class.
2  #ifndef RECTANGLE_H
3  #define RECTANGLE_H
4
5  // Rectangle class declaration.
6
7  class Rectangle
8  {
9     private:
10       double width;
11       double length;
12    public:
13       void setWidth(double);
14       void setLength(double);
15       double getWidth() const;
16       double getLength() const;
17       double getArea() const;
18 };
19
20 #endif
```

- stack
- constructors not executed by explicit fxn calls
  - ↳ initializes object's attributes
- member initialization lists

```
Rectangle()          vs.   Rectangle(): width(0.0), length(0.0) { }
{ width = 0.0
  length = 0.0 }
```

initialization take place before body of constructor executes

- inline initialization
- default constructor
  - ↳ Rectangle *rect_ptr = nullptr (constructor not executed)
    rect_ptr = new Rectangle (constructor executes)
- as long as constructor can be called w/ no arguments, it's the default constructor (can have parameters w/ default arguments)
- destructors

Aside:

```
private:
    char *name;    // The name
    char *phone;   // The phone number
public:
    // Constructor
    ContactInfo(char *n, char *p)
    { // Allocate just enough memory for the name and phone number.
        name = new char[strlen(n) + 1];
        phone = new char[strlen(p) + 1];

        // Copy the name and phone number to the allocated memory.
        strcpy(name, n);
        strcpy(phone, p); }
```

↳ common approach to dynamically allocating memory for a C-string. strlen() only tallies up chars BEFORE the null terminator '\0'. Must add 1 to account for it.

↳ just use strings!

- Dynamically create new class object
  - ↳ ContactInfo *ptr = nullptr
    ptr = new ContactInfo("Armand", "123-4567")
- constructor delegation

```
class Contact {
    string name, email, phone;
public:
    Contact() : Contact("", "", "") { }   // delegate to main constructor

    Contact(string n, string e, string p)
        : name(n), email(e), phone(p) { }
};
```

**Contents of Rectangle.cpp (Version 1)**

```
// Implementation file for the Rectangle class.
#include "Rectangle.h"   // Needed for the Rectangle class
#include <iostream>      // Needed for cout
#include <cstdlib>       // Needed for the exit function
using namespace std;

//***********************************************************
// setWidth sets the value of the member variable width.    *
//***********************************************************

void Rectangle::setWidth(double w)
{
    if (w >= 0)
        width = w;
    else
    {
        cout << "Invalid width\n";
        exit(EXIT_FAILURE);
    }
}

//***********************************************************
// setLength sets the value of the member variable length.  *
//***********************************************************

void Rectangle::setLength(double len)
{
    if (len >= 0)
        length = len;
    else
    {
        cout << "Invalid length\n";
        exit(EXIT_FAILURE);
    }
}

//***********************************************************
// getWidth returns the value in the member variable width. *
//***********************************************************

double Rectangle::getWidth() const
{
    return width;
}

//***********************************************************
// getLength returns the value in the member variable length. *
//***********************************************************

double Rectangle::getLength() const
{
    return length;
}

//***********************************************************
// getArea returns the product of width times length.       *
//***********************************************************

double Rectangle::getArea() const
{
    return width * length;
}
```

**Program 13-5**

```
// This program uses the Rectangle class, which is declared in
// the Rectangle.h file. The member Rectangle class's member
// functions are defined in the Rectangle.cpp file. This program
// should be compiled with those files in a project.
#include <iostream>
#include "Rectangle.h" // Needed for Rectangle class
using namespace std;

int main()
{
    Rectangle box;     // Define an instance of the Rectangle class
    double rectWidth;  // Local variable for width
    double rectLength; // Local variable for length

    // Get the rectangle's width and length from the user.
    cout << "This program will calculate the area of a\n";
    cout << "rectangle. What is the width? ";
    cin >> rectWidth;
    cout << "What is the length? ";
    cin >> rectLength;
```

(program continues)

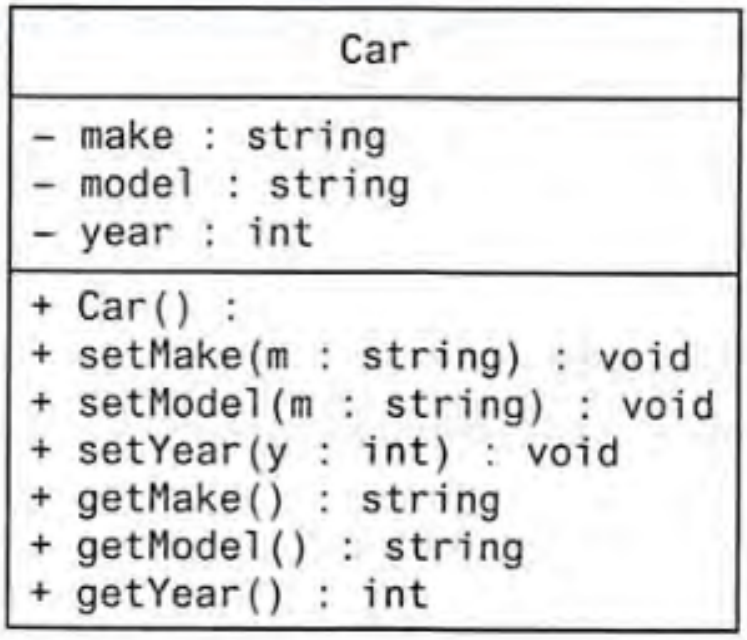


**Program 13-5** *(continued)*

```
    // Store the width and length of the rectangle
    // in the box object.
    box.setWidth(rectWidth);
    box.setLength(rectLength);

    // Display the rectangle's data.
    cout << "Here is the rectangle's data:\n";
    cout << "Width: " << box.getWidth() << endl;
    cout << "Length: " << box.getLength() << endl;
    cout << "Area: " << box.getArea() << endl;
    return 0;
}
```

# UML Diagrams




## STL packages, methods, functions:

### Ch. 12

<fstream> : ifstream, ofstream, fstream

↳ flags : ios:: in, out, ate, app, trunc, binary

↳ ios:: in | ios:: out preserves contents

Error flags/fxns → recoverable → unrecoverable

↳ ::eofbit, failbit, hardfail, badbit, goodbit

↳ .eof(), .fail(), .bad(), .good(), .clear()

Fxns, methods

↳ >>, <<

↳ get(), put()

↳ in.get(ch)
or ch = in.get()

↳ obj = getline(dataFile, str, '\n')

↳ seekg(), seekp()

↳ ios:: beg, end, cur)

↳ seekg(-10L, ios:: end)

↳ tellg(), tellp()

↳ .peek() reads next char w/o moving

Binary Files

↳ .write(&letter, sizeof(data))

↳ .read(&letter, sizeof(data))

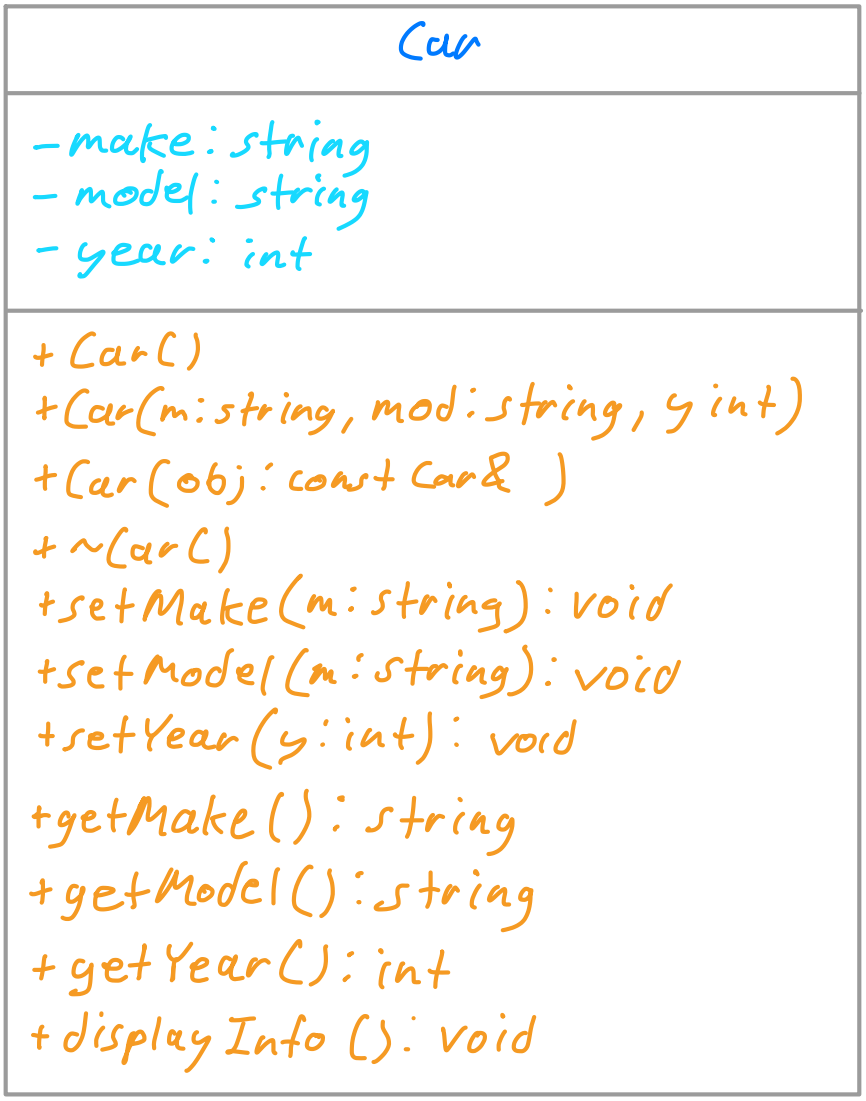
↳ If not char

↳ int x

↳ char *ptr = nullptr;

↳ ptr = reinterpret_cast<char *>(&x)

↳ file.write(reinterpret_cast<char *>(&x), sizeof(x))


Car
- make: string
- model: string
- year: int
+ Car()
+ Car(m: string, mod: string, y int)
+ Car(obj: const Car& )
+ ~Car()
+ setMake(m: string): void
+ setModel(m: string): void
+ setYear(y: int): void
+ getMake(): string
+ getModel(): string
+ getYear(): int
+ displayInfo(): void

# Chapter 13a

1. Class can only have 1 default constructor (True)
2. A private member function is useful for tasks that are internal to the class but it is not directly called by statements outside the class. (True)
3. Whereas object-oriented programming centers on the object, procedural programming centers on functions. (True)
4. Class objects can be defined prior to the class declaration. (False)
5. A destructor function can have zero to many parameters. (False)
6. In-place member initialization no longer is available in C++11. (False)
7. Examples of access specifiers are private and public.
8. When an object is defined without an argument list for its constructor, the compiler automatically calls the object's default constructor. (True)
9. While a class's member functions may be overloaded, the constructor cannot be overloaded. (False)
10. You must declare all data members of a class before you declare member functions. (False)
11. You can use the technique known as a member initialization list to initialize members of a class. (True)
12. The constructor fxn may not accept arguments (False)
13. Destructors free memory allocated by object (True)
14. When using smart pointers to dynamically allocate objects in C++ 11, it is unnecessary to delete the dynamically allocated objects because the smart pointer will automatically delete them. (True)
15. If you do not declare a destructor function, the compiler will furnish one automatically. (True)
16. More than one constructor may be defined for class (True)
17. You must use the **`private`** access specification for all data members of a class. (False)
18. Constructor functions are often used to allocate memory that will be needed by the object. (True)
19. In object-oriented programming, the object encapsulates both the data and the functions that operate on the data. (True)
20. More than one destructor function may be defined for a class. (False)

# Chapter 7 (1 legit wrong)

1. Assume `array1` and `array2` are arrays. To assign the contents of `array2` to `array1`, use `array1 = array2;` (False)
2. If you leave out the size declarator in an array definition, you must furnish an initialization list. (True)
3. An array with no elements is illegal in C++. (True)
4. C++ limits the number of array dimensions to two. (False)
5. If an array is partially initialized, the uninitialized elements will be set to zero. (True)
6. An array initialization must be all on one line. (False)
7. To initialize a `vector<int>` named `n` with the values 10 and 20, use `vector<int> n = {10, 20};` (True) glitched→
8. Given `int numbers[] = {99, 87, 66, 55, 101}; cout << numbers[3];`, the output is 55. (True)
9. By using the same subscript, you can build relationships between data stored in two or more arrays. (True)
10. `int scores[25];` is a valid C++ array definition. (True)
11. A `vector` object automatically expands in size to accommodate the items stored in it. (True)
12. An initialization list can be used to specify the starting values of an array. (True)
13. An individual array element can be processed like any other type of C++ variable. (True)
14. A two-dimensional array can be viewed as rows and columns. (True)
15. An array of `string` objects that holds five names is declared as `string names[5];` (True) glitched
16. A statement that correctly defines a `vector` object for holding integers is `vector<int> v;` (True) →
17. When writing functions that accept multi-dimensional arrays, all but the first dimension must be explicitly stated. (True)
18. There is no known way to pass a two-dimensional array to a function. (False)
19. An element of a two-dimensional array is referred to by the row subscript followed by the column subscript. (True)
20. An array's size declarator must be a constant integer expression with a value greater than zero. (True)
21. Given `int scores[] = {83, 62, 77, 97, 86};`, the value 77 is stored in `scores[2];` (True)
22. Arrays must be sized at the time they are declared. (None of these → True statement)
23. In C++11, the range-based `for` loop is best used when you need the element subscript. (False) or executed?
24. When you pass an array as an argument to a function, the function can modify the contents of the array. (True)
25. Each individual element of an array can be accessed by the array name and subscript. (True)
26. An array can store a group of values, but the values must be the same data type. (True)
27. A two-dimensional array of characters can contain strings of the same length, different lengths, or uninitialized elements. (All of these)
28. `int values[1000];` declares an array with 1000 elements. (True)
29. A two-dimensional array can have elements of one data type. (True)
30. An array can easily be stepped through by using a `for` loop. (True)
31. Storing data past an array's boundaries guarantees a compiler error. (False)
32. `int sizes[10];` is a valid C++ array definition. (True)

```
class Rectangle
{
  private:
    double width;
    double length;
    double height;
  public:
    void setWidth;
    void setLength;
    void setHeight;
    double getWidth() const;
    double getLength() const;
    double getArea() const;
    double getVolume() const;
};
```

set private

"gets" private

public members

public vs private

↳ can be accessed outside of class by only functs that are members of class

- can be listed in any order
- can appear multiple times in class
- if not specified, private
- const: specifies func will not Δ

```
int Rectangle::setWidth(double w)
{
    width = w;
}
```

- mutator: member func that stores value in private.
  setter!
- accessor: retrieve values from private.
  getter!

```
Rectangle r; ← object
r.setWidth(5.2);
cout << r.getWidth();
```

- will fail if dot operator tries to access private func

* Try making calculator w/ class

Reference: 13-1 to review

* Do all ch. 13 programs
* typing.com – 60wpm ↳ send rectangle output
  * ↳ write rectangle program w/ volume

- programming for midterm, final, project
* Do program challenges in back of textbook

State Data

↳ best to calc value of data w/in member fun
rather than store in var

Pointer

```
*ptr = nullptr;
```

```
rPtr = &otherRectangle;
rPtr->setLength(12.5);
cout << rPtr->getLength <<
```

- Dynamic Allocation

```
Rectangle *rectPtr = nullptr;
rectPtr = new Rectangle; } dynamically allocate memory
rectPtr->setWidth(10.0); } store values
rectPtr->setLength(15.0); }
delete rectPtr; } delete object from memory
rectPtr = nullptr; }
```

Went from ASCII to Unicode
- ↳ 1 byte
- ↳ 2 bytes
  - ↳ first 8 bits are all 0s for ASCII conversion
  - ↳ 1 byte

Private Members
- ↳ public functs define class's public interface
- ↳ data can only be accessed publicly

Classes = Libraries

- Separating Specification from Implementation
  - ↳ header for midterm
- Inline Member Functs
  - ↳ inline: in class declaration
  - ↳ after the class declaration

```
#ifndef RECTANGLE_H
#define RECTANGLE_H
  ...
#endif
```

- Constructors
  - ↳ constructor funct name is class name
  - ↳ no return type

public:

Rectangle(); constructor

- constructor initializes values

* Input validation code

* Do Program 13-7
  - ↳ include height, volume

In-Place Initialization

Default Constructors

- ↳ C++ autoconstructs one
- ↳ default constructor takes no arguments
- ↳ simple instantiation of class calls default constructor
  - ↳ Rectangle r

↳ Pass args to constructor

- ↳ can have >1 constructor
- ↳ Rectangle r(10, 5);
- ↳ Rectangle(double =0, double 0);
- ↳ when can requires args, no default const.

## Destructors

- ↳ 1 that does all destruction
- ↳ ~Rectangle
  - ↳ No return type; no args
  - ↳ one per class; cannot be overloaded
  - ↳ if cons allocates dyn mem, dest releases it

## Contents of InventoryItem.h

## Cons, Dests, Dyn Allocation

```
Rectangle *r = new Rectangle (10, 20)
```

↳ deletion?

## Overloading Constructors

```
Rectangle();
Rectangle(double);
Rectangle(double, double);
```

## Programming parameters

- ↳ validation
- ↳ comments

## Constructor Delegation

↳ In C++11, can use one constructor to fill another constructor

Square (int p = 0) will not compile

Member Funct Overloading

Private Member Functs

↳ can only be called by another member funct

## Arrays of Objects

InventoryItem inventory[40];

- initializer vs constructor

* constructors you can always overload

## Accessing Objects in an Array

inventory[2].setUnits(30);

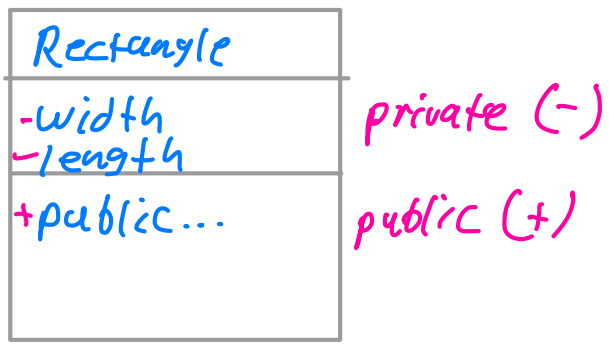
cout<< ...

| Hex | Dec |
|---|---|
| 0 | 0 |
| ⋮ | ⋮ |
| F | 9 |

cout << "Inventory Item" << setw(8)

* Add subtotals and totals

## The Unified Modeling Language



UML

+ setWidth(w: double) : void

* Written in Midterm + Final

HW1

↳ RectEagle

↳ Inventory ——→ add 5 objects

↳ ContactInfo

# Lec 3: 7: Arrays and Vectors

Sept. 20, 2025

cplusplus.com/reference/vector/vector/

## Arrays

↳ values stored in adjacent memory locations

↳ [] operator

Int is data type of array elements

↳ name

↳ size declarator

↳ size of array (func)

↳ number of elements · _____

↳ eg) tests[5] is an array of 20 bytes, each int is 4 bytes

↳ eg) long double is 80 bytes assuming 8 bytes for a long double

↳ 64 bits (8 bits to byte)

## Const

↳ const are fixed

↳ used to declare array size

## Accessing Array Elems

↳ each element is assigned a unique subscript

↳ cout << array //illegal

* Program 7-1, due

Initializing Array

```
const int ARRAY_SIZE = 5;
...
for (int count = 0; count < ARRAY_SIZE; count++;
    numbers[count] = 99;
```

Global Array = all elements initialize to 0

Local Array = all elements uninitialized

↳ initialize ourselves

```
int tests[SIZE] = {79, 82, 91, 77, 84};
```

↳ can't initialize beyond size

Partial Initialization

↳ remainder will be set to 0

```
int quizzes[] = {12, 17, 23, 24}
```

↳ open, memory decided by computer

No bounds checking in C++

↳ will grab junk in memory

Off-by-One Errors

# The Range-base for loop

```
int numbers[] = {1, 2, 3};
for (int val : numbers)
    cout << val << endl;
```

↳ cannot modify array this way

```
for (int &val : numbers)
```

↳ modifies array

```
for (auto &val : numbers)
```

↳ don't know datatype

```
{
  cout <
  cin >
}
```

If you need element, use for loop (general statement)

```
newTests = tests; //won't work
for (i = 0; i < ARRAY_SIZE; i++)
    newTests[i] = tests[i]
```

↳ copy

```
char fName[] = "Henry";
cout << fName << endl;
```

* String is a class

↳ computer converts to string

```
for (int val : numbers)
```

* ranged based

When using accumulator, initialize to 0.

sum += test[num] = sum = sum + test

Parallel arrays

↳ two or more arrays that contain data

* Program 7-15

Arrays as Funct Argument

showScores(test, Array)

* Proggram 7-17

2D Arrays:

```
const int ROWS = 4, COLS = 3;
int exams[ROWS][COLS];
```

exams[2][2] = third row + column

*Program 7-21

```
int exams[ROWS][COLS] = { {84, 78},
                          {92, 97};
```

getExams(exams, 2);

7.9: Arrays w/ three or more Dimensions

When used as param, specify all but first void ... (short [][43][5]);

7.11 Vector - Intro to the STL vector

Vector library ↳ Standard Template Library

library
^
classes
^
functions
^
data

vector<int> scores;

↳ vector is a class

↳ it's open []

vector<int> scores open

vector<int> scores(30,0) defined

* If running program, do C++ 20

scores.size()

class.function

scores.clear();

while(!scores.empty())...

↳ while not empty

*Program 7-25

*Read how vector is programmed

↳ c++ std vector reading

* Mid Oct 18 or 20 or 11

* Final Dec 20

Q1 pg. 812 13-11

Q2 pg. 458 7-15, 7-23

pg. 434 pg. 457

Q3 pg. 460 7.18. Tic Tac Toe

# Lec 4: 9: Pointers

Sept. 27, 2025

## Variable Addresses

↳ Addresses in hexadecimal: 0-F

```
cout << &num;
```

## Pointer Variables

Pointer variable: var that holds address

↳ points to data

```
showValues(numbers, SIZE)
```
↳ array

```
void getOrder(int &donuts)
{
   cout << "...."
   cin >> donuts;
}

int donutsr = 0;
```

Pointers are more low-level than arrays and reference vars.

Arrays are a contiguous block of memory.

```
int *intptr;
```

intptr can hold address of int

```
int * intptr
int* intptr
```
} all the same

```
int *intptr;
intptr = &num;
*intptr = num;
```

int ptr w/o * is the memory address.

```
int *ptr = nullptr;
```

↳ key C++ term

↳ we say empty, really 0.

Program 9-2

stores address of var in pointer

## Indirection Operator

Program 9-3

## Arrays + Pointers

```
int vals[] = {4, 7, 11};
```

↳ array is memory address

```
cout << *vals
```

↳ 4, first element

```
int *valptr = vals;
cout << valptr[1];
```

Program 9-5

```
int vals[] = {4, 7, 11}, *valptr;
valptr = vals;
valptr + 1
```

↳ means (address in valptr) + (1 * size of an int)

## Array Access

ptr and array are memory

## Program 9-7

## Pointer Arithmetic

```
int vals[]
valptr* = vals
```

↳ can increment, decrement (++, --)

↳ *(valptr + 2)

↳ valptr = vals; valptr += 2;

↳ valptr - val; (pointer from pointer)

## Program 9-9

## Initializing Pointers

```
int num, *numptr = &num;
int val[3], *valptr = val;
```

illegal

```
double cost;
int *ptr = &cost;
if (!ptr)
```

test w/ auto

## Comparing Pointers

```
if (ptr1 == ptr2) // addresses
if (ptr1* == ptr2*) // values
```

When you define array, also pointer

## Pointers as Func. Parameters

## 9-11

## Pointers to Constants

```
void displayPayRates...
int* const ptr = &value
```

points to ↑ ↳ cannot point to anything else.


## Dynamic Memory Allocation

```
double *dptr = nullptr;
dptr = new double;
```

raad, random access memory

```
delete fptr;
delete [] fptr; for array
```

9-14

## Returning Pointers from Functions

```
int* newNum();
```

9-15

## Smart Pointers

↳ don't have to worry about delete

unique_ptr
shared_ptr
weak_ptr
} functions?

```
#include <memory>
unique_ptr<int> ptr(new int);
```

9-17

HW

pg. 484 7-2

pg. 553 9-1

pg. 554 9-5

pg. 554 9-9

pg. 555 9-13

# Lec 5: Advanced File Operations

Oct 4, 2025

Nov 29 - no classes
Midterm: Oct 18 or 25 will be midterm
Final: Dec. 20

## Files

- ↳ set of data stored
- ↳ read from; write to files

## fstream

- ↳ ifstream (input file stream)
- ↳ ofstream
- ↳ fstream
- ↳ >> read from
- ↳ << write to
- ↳ eof (end of file)
  - ↳ boolean value

## open call

dFile.open ...

## File output formatting

setw(x), showprecision(x)

- ↳ requires iomanip
  - ↳ input/output manipulator

12-3

12-5*

- ↳ pass objects by reference

More Detailed Error Testing

12-6*

- getline function: reads input including
- '\n'
  - ↳ newline or return

Get one char
↳ .get(Lettergrade.csv)

Multiple Files

12-12*

Binary Files
↳ contains unformatted, non-ASCII data
↳ inFile.open("nums.dat", ios::in | ios::out)

Record w/ structures

Random-Access

*object.function

.dat = bin file

HW:
Eg
- 9-10, ~
- ~~12-5~~,
- 12-8, ✓
- 12-12
- 12-21
  ↳ check course materials for file: inventory.dat

Midterm
- Chapter 6 menu driven
- Two Programs

How to Think like a Computer Scientist

PC:
12-1
↳ create our own data files

HW: 12-10*
↳ file decrypter filter

# Lec 6: Ch. 14: More about Classes

Oct 11, 2025

## Instance and Static Members

- static variable: one var shared among all objects of class.
- static member function: used to access...

### 14-1

static can only access other static vars

Modified version of Tree.h

↳ line 12

### 14.2 Friends of Classes

Friend: a fxn or class that is not member of class, but has access to private members of the class

↳ friend can emerge from other classes

↳ friend is keyword

```
friend void setAVal(intVal8, int)
friend void SomeClass::setNum(int num)
```

↳ use public, can this be done private?

## Membershipwise Assignment

```
Rectangle r2 = r1;
```

### 14.5*

### 14.9 Copy Constructors

↳ default copy constructor copies field-to-field

### 14.5 Operator Overloading

↳ operator+ to overload + operator

↳ operator = to overload = operator

↳ this C++ keyword

↳ predefined pointer available to a class's member fxn

Notes on Overloaded Operators
↳ cannot change operands of operator

pg. 837 in textbook onward

*Version 3 of Student test scores
↳ v1 is not overloaded
↳ STUDENTSTESTSCORE.h

14-6

keyword : using
↳ look up!

Aggregation
↳ supports the modeling of 'has a' relationship btwn classes

14.10 Rvalue References + Temp Values
↳ lvalues persist beyond statement
↳ rvalues are temporary, cannot be accessed beyond statement
↳ use to move data outside class

14-7*

Midterm: Oct. 25 *

6-10*
↳ part of exam
↳ mean driven

In-class HW:
12-3
14-6
14-7
12-11

Out-of class HW:
6-10

# Lec 7:

Oct 18, 2025 File questions

Must draw the header file ↳ Comments on every line

* ↳ Draw the UML digram

## Questions

↳ 14-11 Notepad++

↳ 14-13 Bring paper and pen

| Rectangle |
| --- |
| - width : double<br>- length : double |
| + setWidth(w : double) : void<br>+ setLength(len : double) : void<br>+ getWidth() : double<br>+ getLength() : double<br>+ getArea() : double |

**FeetInches**

- feet : int
- inches : int
- simplify() : void

+ FeetInches(f:int=0, i:int=0)
+ FeetInches(obj : FeetInches)
+ multiply(obj : FeetInches) : FeetInches
+ setFeet(f: int)
+ setInches(i : int, simplify() : void)
+ getFeet(feet : constant int)
+ getInches(inches : constant int)
+

# Chapter 13

**Question 1** 6.25 / 6.25 points

The constructor function may not accept arguments.

- ○ True
- ✓ ◉ False

**Question 2** 6.25 / 6.25 points

Members of the class object are accessed with the

- ✓ ◉ dot operator
- ○ `cin` object
- ○ extraction operator
- ○ stream insertion operator

- ○ None of these

**Question 3** 6.25 / 6.25 points

A class is a(n) _____ that is defined by the programmer.

- ✓ ◉ data type
- ○ function
- ○ method
- ○ attribute
- ○ None of these

**Question 4** 6.25 / 6.25 points

Objects are created from abstract data types that encapsulate _____ and _____ together.

- ○ numbers, characters
- ✓ ◉ data, functions
- ○ addresses, pointers
- ○ integers, floating-point numbers
- ○ None of these

**Question 5** 6.25 / 6.25 points

When a constructor function accepts no arguments, or does NOT have to accept arguments because of default arguments, it is called a(n)

- ○ empty constructor
- ✓ ◉ default constructor
- ○ stand-alone function
- ○ arbitrator function

- ○ None of these

**Question 6** 6.25 / 6.25 points

If you do NOT declare an access specification, the default for members of a class is

- ○ `inline`
- ✓ ◉ `private`
- ○ `public`
- ○ `global`
- ○ None of these

**Question 7** 6.25 / 6.25 points

Where are class declarations usually stored?

- ○ on separate disk volumes
- ✓ ◉ in their own header files
- ○ in .cpp files, along with function definitions
- ○ under pseudonyms
- ○ None of these

**Question 8** 6.25 / 6.25 points

A C++ class is similar to a(n)

- ○ inline function
- ○ header file
- ○ library function
- ✓ ◉ structure

- ○ None of these

**Question 9** 6.25 / 6.25 points

Examples of access specifiers are the key words

- ○ **`near`** and **`far`**
- ○ **`opened`** and **`closed`**
- ✓ ◉ **`private`** and **`public`**
- ○ **`table`** and **`row`**
- ○ None of these

**Question 10** 6.25 / 6.25 points

The destructor function's return type is

- ○ **`int`**
- ○ **`float`**
- ○ **`char`**
- ✓ ◉ Nothing; destructors have no return type
- ○ None of these

**Question 11** 6.25 / 6.25 points

In OOP terminology, an object's member variables are often called its _____ and its member functions can be referred to as its behaviors or its _____.

- ○ values, morals
- ○ data, activities
- ○ attributes, activities
- ✓ ◉ attributes, methods

- ○ None of these

**Question 12** 6.25 / 6.25 points

Class objects can be defined prior to the class declaration.

- ○ True
- ✓ ◉ False

**Question 13** 6.25 / 6.25 points

How many default constructors can a class have?

- ✓ ◉ only one
- ○ two or more
- ○ only two
- ○ any number
- ○ None of these

**Question 14** 6.25 / 6.25 points

Which of the following is a directive used to create an "include guard" that allows a program to be conditionally compiled to prevent a header file from accidentally being included more than once?

- ○ `#include`
- ○ `#guard`
- ✓ ◉ `#ifndef`
- ○ `#endif`
- ○ None of these

**Question 15** 6.25 / 6.25 points

The type of member function that may be called from a statement outside the class is

- ✔ ◉ **`public`**
- ○ **`private`**
- ○ undeclared
- ○ **`global`**
- ○ None of these

**Question 16** 6.25 / 6.25 points

In a procedural program you typically have _____ stored in a collection of variables and a set of _____ that perform operations on the data.

- ○ numbers, arguments
- ○ parameters, arguments
- ○ strings, operators
- ✔ ◉ data, functions
- ○ None of these

Done

# Chapter 9

**Question 1** 4 / 4 points

If a variable uses more than one byte of memory, for pointer purposes its address is
If a variable uses more than one byte of memory, for pointer purposes its address is
Question options:

the address of the last byte of storage
the address of the last byte of storage

the average of all the addresses used to store that variable
the average of all the addresses used to store that variable

the address of the first byte of storage
the address of the first byte of storage

the address of the second byte of storage
the address of the second byte of storage

None of these
None of these

**Question 2** 4 / 4 points

C++ does not perform array bounds checking, making it possible for you to assign a pointer the address of an element out of the boundaries of an array.
C++ does not perform array bounds checking, making it possible for you to assign a pointer the address of an element out of the boundaries of an array.
Question options:

True

False

**Question 3** 4 / 4 points

A pointer can be used as a function argument, giving the function access to the original argument.

A pointer can be used as a function argument, giving the function access to the original argument.
Question options:

True

False

**Question 4** 4 / 4 points

After the code shown executes, which of the following statements is TRUE?

```
int numbers[] = {0, 1, 2, 3, 4};
int *ptr = numbers;
ptr++;
```

After the code shown executes, which of the following statements is TRUE?

```
int numbers[] = {0, 1, 2, 3, 4};
int *ptr = numbers;
ptr++;
```

Question options:

**ptr** will hold the address of **numbers[0]**

**ptr** will hold the address of **numbers[0]**

**ptr** will hold the address of the second byte within the element **numbers[0]**

**ptr** will hold the address of the second byte within the element **numbers[0]**

**ptr** will hold the address of **numbers[1]**

**ptr** will hold the address of **numbers[1]**

this code will not compile

this code will not compile

**Question 5** 4 / 4 points

Not all arithmetic operations can be performed on pointers. For example, you cannot _____ or _____ pointers.

Not all arithmetic operations can be performed on pointers. For example, you cannot _____ or _____ pointers.

Question options:

multiply, divide
multiply, divide

**+=, -=**
**+=, -=**

add, subtract
add, subtract

increment, decrement
increment, decrement

None of these
None of these

**Question 6** 4 / 4 points

A function may return a pointer but the programmer must ensure that the pointer

A function may return a pointer but the programmer must ensure that the pointer

Question options:

still points to a valid object after the function ends
still points to a valid object after the function ends

has not been assigned an address
has not been assigned an address

was received as a parameter by the function
was received as a parameter by the function

has not previously been returned by another function
has not previously been returned by another function

None of these
None of these

**Question 7** 4 / 4 points

A pointer variable is designed to store
A pointer variable is designed to store
Question options:

any legal C++ value
any legal C++ value

only floating-point values
only floating-point values

an integer
an integer

a memory address
a memory address

None of these
None of these

**Question 8** 4 / 4 points

Select all that apply. Of the following, which statements have the same meaning?
Select all that apply. Of the following, which statements have the same meaning?
Question options:

```
int *ptr = nullptr;
```

```
int *ptr = nullptr;
```


```
*int ptr = nullptr;
```

```
*int ptr = nullptr;
```


```
int ptr* = nullptr;
```

```
int ptr* = nullptr;
```


```
int* ptr = nullptr;
```

```
int* ptr = nullptr;
```


```
int ptr = nullptr;
```

```
int ptr = nullptr;
```


**Question 9** 4 / 4 points

A pointer variable may be initialized with
A pointer variable may be initialized with
Question options:

any nonzero integer value
any nonzero integer value

a valid address in the computer's memory
a valid address in the computer's memory

an address less than zero
an address less than zero

any nonzero number
any nonzero number

None of these
None of these

**Question 10** 4 / 4 points

Dynamic memory allocation occurs
Dynamic memory allocation occurs
Question options:

when a new variable is created by the compiler
when a new variable is created by the compiler

when a new variable is created at runtime
when a new variable is created at runtime

when a pointer fails to dereference the right variable
when a pointer fails to dereference the right variable

when a pointer is assigned an incorrect address
when a pointer is assigned an incorrect address

None of these
None of these

**Question 11** 4 / 4 points

If you are using an older computer that does NOT support the C++11 standard, you should initialize pointers with
If you are using an older computer that does NOT support the C++11 standard, you should initialize pointers with
Question options:

the integer **0** or the value **NULL**
the integer **0** or the value **NULL**

the null terminator **'\0'**
the null terminator **'\0'**

a nonzero value
a nonzero value

Any of these
Any of these

None of these
None of these

**Question 12** 4 / 4 points

Assuming **ptr** is a pointer variable, what will the following statement output?
**cout << *ptr;**
Assuming **ptr** is a pointer variable, what will the following statement output?
**cout << *ptr;**
Question options:

the value stored in the variable whose address is contained in **ptr**
the value stored in the variable whose address is contained in **ptr**

the string **"*ptr"**
the string **"*ptr"**

the address of the variable whose address is stored in **ptr**
the address of the variable whose address is stored in **ptr**

the address of the variable stored in **`ptr`**
the address of the variable stored in **`ptr`**

None of these
None of these

**Question 13** 0 / 4 points

In C++11, the **`nullptr`** keyword was introduced to represent the address **`0`**.
In C++11, the **`nullptr`** keyword was introduced to represent the address **`0`**.

Question options:

True

False

**Question 14** 4 / 4 points

The _____ and _____ operators can be used to increment or decrement a pointer variable.
The _____ and _____ operators can be used to increment or decrement a pointer variable.

Question options:

addition, subtraction
addition, subtraction

**`++, --`**
**`++, --`**

modulus, division
modulus, division

All of these
All of these

None of these
None of these

**Question 15** 4 / 4 points

Every byte in the computer's memory is assigned a unique
Every byte in the computer's memory is assigned a unique

Question options:

pointer
pointer

address
address

dynamic allocation
dynamic allocation

name
name

None of these
None of these

**Question 16** 4 / 4 points

The _____, also known as the address operator, returns the memory address of a variable.
The _____, also known as the address operator, returns the memory address of a variable.

Question options:

asterisk (***** )
asterisk (***** )

ampersand ( **&** )
ampersand ( **&** )

percent sign ( **%** )
percent sign ( **%** )

exclamation point ( **!** )
exclamation point ( **!** )

None of these
None of these

**Question 17** 4 / 4 points

Select all that apply. Select as many of the following options that make this sentence TRUE:
The contents of pointer variables may be changed with mathematical statements that perform

Select all that apply. Select as many of the following options that make this sentence TRUE:
The contents of pointer variables may be changed with mathematical statements that perform
Question options:

multiplication
multiplication

division
division

modulus
modulus

subtraction
subtraction

addition
addition

**Question 18** Glitched 0 / 4 points

Assuming **`myValues`** is an array of **`int`** values and **`index`** is an **`int`** variable, both of the following statements do the same thing.

Assuming **`myValues`** is an array of **`int`** values and **`index`** is an **`int`** variable, both of the following statements do the same thing.

Question options:

True

False

**Question 19** 4 / 4 points

It is legal to subtract a pointer variable from another pointer variable.

It is legal to subtract a pointer variable from another pointer variable.
Question options:

True

False

**Question 20** 4 / 4 points

In the following statement, what does **`int`** mean?

```
int *ptr = nullptr;
```

In the following statement, what does **`int`** mean?

```
int *ptr = nullptr;
```

Question options:

The variable named **`*ptr`** will store an integer value.
The variable named **`*ptr`** will store an integer value.

The variable named **`*ptr`** will store an asterisk and an integer value
The variable named **`*ptr`** will store an asterisk and an integer value

**`ptr`** is a pointer variable and will store the address of an integer variable.
**`ptr`** is a pointer variable and will store the address of an integer variable.

The variable named **`*ptr`** will store the value in **`nullptr`**.
The variable named **`*ptr`** will store the value in **`nullptr`**.

None of these
None of these

**Question 21** 4 / 4 points

An array name is a pointer constant because the address stored in it cannot be changed at runtime.
An array name is a pointer constant because the address stored in it cannot be changed at runtime.

Question options:

True

False

**Question 22** 4 / 4 points

The following statement _____.

```
cin >> *num3;
```

The following statement _____.

```
cin >> *num3;
```

Question options:

stores the keyboard input in the variable **num3**

stores the keyboard input in the variable **num3**

stores the keyboard input into the pointer **num3**

stores the keyboard input into the pointer **num3**

is illegal in C++

is illegal in C++

stores the keyboard input into the variable pointed to by **num3**

stores the keyboard input into the variable pointed to by **num3**

None of these

None of these

**Question 23** 4 / 4 points

In C++11 you can use smart pointers to dynamically allocate memory and not worry about deleting the memory when you are finished using it.

In C++11 you can use smart pointers to dynamically allocate memory and not worry about deleting the memory when you are finished using it.

Question options:

True

False

**Question 24** 4 / 4 points

The ampersand (`&`) is used to dereference a pointer variable in C++.
The ampersand (`&`) is used to dereference a pointer variable in C++.
Question options:

True

False

**Question 25** 4 / 4 points

In C++11, the _____ keyword was introduced to represent address `0`.
In C++11, the _____ keyword was introduced to represent address `0`.
Question options:

**`nullptr`**
**`nullptr`**

**`NULL`**
**`NULL`**

**`weak_ptr`**
**`weak_ptr`**

**`shared_ptr`**
**`shared_ptr`**

None of these
None of these

# Chapter 12

**Question 1** 4 / 4 points

The **`setprecision`** manipulator cannot be used to format data written to a file.

The **`setprecision`** manipulator cannot be used to format data written to a file.

True

True

False

False

**Question 2** 4 / 4 points

Data stored _____ disappears once the program stops running or the computer is powered down.

Data stored _____ disappears once the program stops running or the computer is powered down.

on a CD

on a CD

in RAM

in RAM

on a flash drive

on a flash drive

on the disk drive

on the disk drive

None of these

None of these

**Question 3** 4 / 4 points

File output may be formatted the same way as console screen output.

File output may be formatted the same way as console screen output.

True

True

False

False

**Question 4** 4 / 4 points

In order, the three-step process of using a file in a C++ program involves

In order, the three-step process of using a file in a C++ program involves

insert a disk, open the file, remove the disk

insert a disk, open the file, remove the disk

create the file contents, close the file, name the file

create the file contents, close the file, name the file

open the file, read/write/save data, close the file

open the file, read/write/save data, close the file

name the file, open the file, delete the file

name the file, open the file, delete the file

None of these

None of these

**Question 5** 4 / 4 points

When you store data in a variable, it is automatically saved in a file.

When you store data in a variable, it is automatically saved in a file.

True

True

False

False

**Question 6** 4 / 4 points

The **`ios::out`** flag causes the file's existing contents to be deleted if the file already exists.

The **`ios::out`** flag causes the file's existing contents to be deleted if the file already exists.

True

True

False

False

**Question 7** 0 / 4 points

If a file already exists, you can open it with the flags **`ios::in | ios::out`** to preserve its contents.

If a file already exists, you can open it with the flags **`ios::in | ios::out`** to preserve its contents.

True

True

False

False

**Question 8** 4 / 4 points

An alternative to using the **`open`** member function is to use the file stream object declaration itself to open the file. For example:

```
fstream DataFile("names.dat", ios::in | ios::out);
```

An alternative to using the **`open`** member function is to use the file stream object declaration itself to open the file. For example:

```
fstream DataFile("names.dat", ios::in | ios::out);
```

True

True

False

False

**Question 9** 4 / 4 points

When a file is opened, the file stream object's "read position" is

When a file is opened, the file stream object's "read position" is

at the end of the file

at the end of the file

at the beginning of the file

at the beginning of the file

nonexistent until the programmer declares it

nonexistent until the programmer declares it

in the middle of the file

in the middle of the file

None of these

None of these

**Question 10** 4 / 4 points

Which of the following access flags, when used by itself, causes a file's contents to be deleted if the file already exists?

Which of the following access flags, when used by itself, causes a file's contents to be deleted if the file already exists?

`ios::app`

`ios::app`

`ios::in`

`ios::in`

`ios::out`

`ios::out`

Any of these

Any of these

None of these

None of these

**Question 11** 4 / 4 points

To write to a file, you use the **`file_write`** function.

To write to a file, you use the **`file_write`** function.

True

True

False

False

**Question 12** 4 / 4 points

When data is read from a file, it is automatically stored in a variable.

When data is read from a file, it is automatically stored in a variable.

True

True

False

False

**Question 13** 4 / 4 points

When used by itself, the **`ios::app`** flag causes the file's existing contents to be deleted if the file already exists.

When used by itself, the **`ios::app`** flag causes the file's existing contents to be deleted if the file already exists.

True

True

False

False

**Question 14** 4 / 4 points

What is TRUE about the following statement?

```
out.open("values.dat", ios::app);
```

What is TRUE about the following statement?

```
out.open("values.dat", ios::app);
```

If the file already exists, its contents are preserved and all output is written to the end of the file.

If the file already exists, its contents are preserved and all output is written to the end of the file.

If the file already exists, it should be replaced with a new copy of **`values.dat`**.

If the file already exists, it should be replaced with a new copy of **`values.dat`**.

If the file already exists, it can be opened but not modified.

If the file already exists, it can be opened but not modified.

None of these

None of these

**Question 15** Question this 0 / 4 points

All stream objects have _____ which indicate the position of the stream.

All stream objects have _____ which indicate the position of the stream.

error state bits

error state bits

condition statements

condition statements

markers

markers

intrinsic error messages

intrinsic error messages

← my answer

None of these

None of these

**Question 16** 4 / 4 points

By default, files are opened in binary mode.

By default, files are opened in binary mode.

True

True

False

False

**Question 17** 4 / 4 points

Outside of a C++ program, a file is identified by its _____while inside a C++ program, a file is identified by a(n) _____.

Outside of a C++ program, a file is identified by its _____while inside a C++ program, a file is identified by a(n) _____.

file number, file name

file number, file name

file name, file number

file name, file number

name, address

name, address

name, file stream object

name, file stream object

None of these

None of these

**Question 18** 4 / 4 points

To set up a file to perform I/O you must declare

To set up a file to perform I/O you must declare

at least one variable, the contents of which will be written to the file

at least one variable, the contents of which will be written to the file

one or more file stream objects

one or more file stream objects

a **`string`** object to store the file contents

a **`string`** object to store the file contents

All of these

All of these

None of these

None of these

**Question 19** 4 / 4 points

Closing a file causes any unsaved information still held in the file buffer to be

Closing a file causes any unsaved information still held in the file buffer to be

saved to the file

saved to the file

deleted

deleted

retained in the buffer for safekeeping

retained in the buffer for safekeeping

duplicated

duplicated

None of these

None of these

**Question 20** 4 / 4 points

Only one file stream object can be declared per C++ program.

Only one file stream object can be declared per C++ program.

True

True

False

False

**Question 21** 4 / 4 points

The **`ios::hardfail`** bit is set when an unrecoverable error occurs.

The **`ios::hardfail`** bit is set when an unrecoverable error occurs.

True

True

False

False

**Question 22** 4 / 4 points

The end-of-file marker is automatically written

The end-of-file marker is automatically written

when a file is opened with **`ios::eof`**

when a file is opened with **`ios::eof`**

when a file is opened with **`ios::app`**

when a file is opened with **`ios::app`**

when a file is closed

when a file is closed

when the program ends

when the program ends

None of these

None of these

**Question 23** 4 / 4 points

When passing a file stream object to a function, you should always pass it by reference.

When passing a file stream object to a function, you should always pass it by reference.

True

True

False

False

**Question 24** 4 / 4 points

The _____ marker is the character that marks the end of a file and is automatically written when the file is closed.

The _____ marker is the character that marks the end of a file and is automatically written when the file is closed.

End Of File (**EOF**)

End Of File (**EOF**)

No More Data (**NMD**)

No More Data (**NMD**)

Data Stream Close (**DSC**)

Data Stream Close (**DSC**)

Data Read Stop (**DRS**)

Data Read Stop (**DRS**)

None of these

None of these

**Question 25** Question this 0 / 4 points

Which of the following data types can be used to create files and read information from them into memory?

Which of the following data types can be used to create files and read information from them into memory?

**ofstream**

**ofstream**

**istream**

**istream**

**ifstream**

**ifstream**

**instream**

**instream**

None of these

None of these

← my answer, only fstream can do this