CS 161 Final Study Guide
Winter 2017

defining a function

contains a *name, parameter list, body, return type.*

ex.

```
double findSomething(double a, double b){
        return a * b;
}
```

void functions - doesn't return value.

ex.

```
void setMain(int a){
        x = a;
}
```

calling a function

When called, it branches into function and executes statements in body.

ex.

```
void dispMess(){
        cout << "Hello" << endl;
}

int main(){
        dispMess();
        return 0;
}
```

(Will print out "Hello" from the function call.)

ex.

```
void deeper(){
        cout << "TESTING!" << endl;
}

void deep(){
        deeper()
{

int main(){
        deep();
        return 0;
}
```

(Will return "TESTING!" from calling a function that calls another function.)

Checkpoint

6.1 Is the following a function header or a function call? calcTotal();

6.2 Is the following a function header or a function call? void showResults()

6.3 What will the output of the following program be if the user enters 10?

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

void func1()
{
	cout << "Able was I\n";
}
void func2()
{
	cout << "I saw Elba\n";
}
int main()
{
	int input;
	cout << "Enter a number: ";
	cin >> input;
	if (input < 10)
	{
		func1();
		func2();
	}
	else
	{
		func2();
		func1();
	}
	return 0;
}
```

6.4
The following program skeleton determines whether a person qualifies for a credit card. To qualify, the person must have worked on his or her current job for at least two years and make at least $17,000 per year. Finish the program by writing the definitions of the functions qualify and noQualify. The function qualify should explain that the applicant qualifies for the card and that the annual interest rate is 12 percent. The function noQualify should explain that the applicant does not qualify for the card and give a general explanation why.

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

// You must write definitions for the two functions qualify
// and noQualify.

int main()
{
	double salary;
	int years;

	cout << "This program will determine if you qualify\n";
	cout << "for our credit card.\n";
	cout << "What is your annual salary? ";
	cin >> salary;
	cout << "How many years have you worked at your ";
	cout << "current job? ";
	cin >> years;

	if (salary >= 17000.0 && years >= 2)
		qualify();
	else
		noQualify();
	return 0;
}
```

function prototypes

A function prototype eliminates the need to place a function definition before all calls to the function.

- *Function prototypes are also known as function declarations*
- *WARNING! Must either place the function definition or the function prototype ahead of all calls to the function. Otherwise program will not compile.*

ex.

```
//Function Prototype
void dispMess();

int main(){
        dispMess();
        return 0;
}

void dispMess(){
        cout << "Hello" << endl;
}
```

passing parameters (by value)

When an argument is passed into a parameter by value, only a copy of the argument's value is passed. Changes to parameter do not affect the original argument. (Pointers will do this.)
-Main purpose is to hold data that is passed, it receives a copy of the data.

ex.

```
//Function prototype
void callParam(int aValue);

int main()
{
        int number = 1;
        callParam(number);
        return 0;
}

void callParam(int dummy)
{
dummy = 14;
cout << "dummy is " << dummy;
}
```

(Will output "dummy is 14". Value of number is still 1, passed into function as 1, by then changed to 14 within the function. number remains unchanged in main.)

Checkpoint

6.5 Indicate which of the following is the function prototype, the function header, and the function call:

```
void showNum(double num)
void showNum(double);
showNum(45.67);
```

6.6 Write a function named timesTen. The function should have an integer parameter named number. When timesTen is called, it should display the product of number times 10. (Note: just write the function. Do not write a complete program.)

6.7 Write a function prototype for the timesTen function you wrote in question 6.6.

6.8 What is the output of the following program?

```
#include <iostream>
using namespace std;
void showDouble(int value); // Function prototype
int main()
{
        int num;
        for (num = 0; num < 10; num++)
        showDouble(num);
        return 0;
}

// Definition of function showDouble
void showDouble(int value)
{
        cout << value << " " << (value * 2) << endl;
}
```

6.9 What is the output of the following program?

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

void func1(double, int); // Function prototype

int main()
{
        int x = 0;
        double y = 1.5;
        cout << x << " " << y << endl;
        func1(y, x);
        cout << x << " " << y << endl;
        return 0;
}

void func1(double a, int b)
{
        cout << a << " " << b << endl;
        a = 0.0;
        b = 10;
        cout << a << " " << b << endl;
}
```

6.10

The following program skeleton asks for the number of hours you've worked and your hourly pay rate. It then calculates and displays your wages. The function showDollars, which you are to write, formats the output of the wages.

```
#include <iostream>
#include <iomanip>
using namespace std;

void showDollars(double pay); // Function prototype

int main()
{
        double payRate, hoursWorked, wages;
        cout << "How many hours have you worked? "
        cin >> hoursWorked;
        cout << "What is your hourly pay rate? ";
        cin >> payRate;
        wages = hoursWorked * payRate;
        showDollars(wages);
        return 0;
```

```
}

// Write the definition of the showDollars function here.
// It should have one double parameter and display the message
// "Your wages are $" followed by the value of the parameter.
```

Checkpoint

6.16 What is the difference between a static local variable and a global variable?

6.17 What is the output of the following program?

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

void myFunc(); // Function prototype

int main()
{
	int var = 100;
	cout << var << endl;
	myFunc();
	cout << var << endl;
	return 0;
}

// Definition of function myFunc
void myFunc()
{
	int var = 50;
	cout << var << endl;
}
```

6.18 What is the output of the following program?

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

void showVar(); // Function prototype

int main()
{
	for (int count = 0; count < 10; count++)
	showVar();
	return 0;
}

// Definition of function showVar
void showVar()
{
	static int var = 10;
	cout << var << endl;
	var++;
}
```

passing parameters (by reference)

When wanted to change the value in the calling function, we can do this by making the parameter a reference variable.

- Instead of having its own memory location, it uses the memory location of another variable. This changes the data within that variable.
- Calling is an ampersand **&**

ex.

```
// Function prototype. The parameter is a reference variable.
 void doubleNum(int &refVar);

 int main()
{
 int value = 4;

	cout << "In main, value is " << value << endl;
	cout << "Now calling doubleNum..." << endl;
	doubleNum(value);
	cout << "Now back in main, value is " << value << endl;
	return 0;
}
```

(cont.)

```
void doubleNum (int &refVar)
{
        refVar *= 2;
}
```

(this results in the original value, 4, to be changed in main to 8.)

NOTE: Only variables may be passed by reference. If you attempt to pass a non-variable argument, such as a literal, a constant, or an expression, into a reference parameter, it will cause an error.

When to Pass Arguments by Reference and When to Pass Arguments by Value;

- When an argument is a constant, it must be passed by value. Only variables can be passed by reference.
- When a variable passed as an argument should not have its value changed, it should be passed by value. This protects it from being altered.
- When exactly one value needs to be "sent back" from a function to the calling routine, it should generally be returned with a return statement rather than through a reference parameter.
- When two or more variables passed as arguments to a function need to have their values changed by that function, they should be passed by reference.
- When a copy of an argument cannot reasonably or correctly be made, such as when the argument is a file stream object, it must be passed by reference.
-

Here are three common instances when reference parameters are used.

- When data values being input in a function need to be known by the calling function.
- When a function must change existing values in the calling function.
- When a file stream object is passed to a function.

Checkpoint

6.19 What kinds of values may be specified as default arguments?

6.20 Write the prototype and header for a function called compute. The function should have three parameters: an int, a double, and a long (not necessarily in that order). The int parameter should have a default argument of 5, and the long parameter should have a default argument of 65536. The double parameter should not have a default argument.

6.21 Write the prototype and header for a function called calculate. The function should have three parameters: an int, a reference to a double, and a long (not necessarily in that order.) Only the int parameter should have a default argument, which is 47.

6.22 What is the output of the following program?

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

void test(int = 2, int = 4, int = 6);

int main()
{
	test();
	test(6);
	test(3, 9);
	test(1, 5, 7);
	return 0;
}

void test (int first, int second, int third)
{
	first += 3;
	second += 6;
	third += 9;
	cout << first << " " << second << " " << third << endl;
}
```

6.23
The following program asks the user to enter two numbers. What is the output of the program if the user enters 12 and 14?

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

void func1(int &, int &);
void func2(int &, int &, int &);
void func3(int, int, int);

int main()
{
	int x = 0, y = 0, z = 0;
	cout << x << " " << y << z << endl;
	func1(x, y);
	cout << x << " " << y << z << endl;
	func2(x, y, z);
	cout << x << " " << y << z << endl;
	func3(x, y, z);
	cout << x << " " << y << z << endl;
```

```
	return 0;
}

void func1(int &a, int &b)
{
	cout << "Enter two numbers: ";
	cin >> a >> b;
}

void func2(int &a, int &b, int &c)
{
	b++;
	c--;
	a = b + c;
}

void func3(int a, int b, int c)
{
	a = b - c;
}
```

Chapter 6 Misc Notes:

- Remember that local variables are not automatically initialized as global variables are, Programmer must handle this.
- A function's default arguments should be assigned in the earliest occurrence of the function name. This will usually be the function prototype. However, if a function does not have a prototype, default arguments may be specified in the function header. The showArea function could be defined as follows:

```
void showArea(double length = 20.0, double width = 10.0)
{
double area = length * width;
cout << "The area is " << area << endl;
}
```

- The exit() function unconditionally shuts down your program. Because it bypasses a program's normal logical flow, you should use it with caution.

<u>defining classes</u>

In C++, the class is the construct primarily used to create objects. Main concept in OOP.

ex.

```
class ClassName              // Class declaration begins with
{                            // the key word class and a name.

        Declarations for class member variables
        and member functions go here.

};                           // Notice the required semicolon.
```

- NOTE: If a program statement outside a class attempts to access a private member, a compiler error will result. Later you will learn how outside functions may be given special permission to access private class members.

creating and using objects

Objects are instances of a class. They are created with a definition statement after the class has been declared.

- Accessing object members are done by calling the class object then with a .functionName.

  ex.

  ```
  cout << city.length() << endl;
  circle1.setRadius(1.0);          // This sets circle1's radius to 1.0
  circle2.setRadius(2.5);                  // This sets circle2's radius to 2.5
  ```

- Inline functions
  - Can be called within the class if it is short

    ex.

    ```
    (Within the class file)
    void setRadius(double)
    { radius = 0; }
    ```
- When calling a function outside the class, we must use the *scope resolution operator*. (::)

  ex.

  ```
  void Circle::setRadius(double r)
  { radius = r;
  }

  double Circle::getArea()
  { return 3.14 * pow(radius, 2);
  }
  ```

  WARNING!

  The class name and scope resolution operator are an extension of the function name. When a function is defined outside the class declaration, these must be present and must be located immediately before the function name in the function header.

Checkpoint

7.1 Which of the following shows the correct use of the scope resolution operator in a member function definition?

A) InvItem::void setOnHand(int units)
B) void InvItem::setOnHand(int units)

7.2 An object's private member variables can be accessed from outside the object by

A) public member functions
B) any function
C) the dot operator
D) the scope resolution operator

7.3 Assuming that soap is an instance of the Inventory class, which of the following is a valid call to the setOnHand member function?

A) setOnHand(20);
B) soap::setOnHand(20);
C) soap.setOnHand(20);
D) Inventory.setOnHand(20);

7.4 Complete the following code skeleton to declare a class called Date. The class should contain member variables and functions to store and retrieve the month, day, and year components of a date.

```
class Date
{
        private:
        public:
}
```

constructors and destructors

A constructor is a member function that is automatically called when a class object is created.

- Looks like a regular function, except its the same name as its class. ie Circle() and no return type.
- Overloading constructors - Circle(), Circle(int a);
    - Circle() is a default constructor.

A destructor is a member function that is automatically called when an object is destroyed.

- Like constructors, destructors have no return type.
- Destructors cannot accept arguments, so they never have a parameter list.
- Because destructors cannot accept arguments, there can only be one destructor.

Checkpoint

7.5 Briefly describe the purpose of a constructor.

7.6 Constructor functions have the same name as the

A) class
B) class instance
C) program
D) none of the above

7.7 A constructor that requires no arguments is called

A) a default constructor
B) an inline constructor
C) a null constructor
D) none of the above

7.8 Assume the following is a constructor:

```
ClassAct::ClassAct(int x)
{
        item = x;
}
```

Define a ClassAct object called sally that passes the value 25 to the constructor.

7.9 True or false: Like any C++ function, a constructor may be overloaded, providing each constructor has a unique parameter list.

private and public

Private member functions may only be called from a function that is a member of the same class. Public members functions are all the functions that the program may access directly.

ex.

```
class SimpleStat
{
private:
        int largest; // The largest number received so far
        bool isNewLargest(int); // This is a private class function
public:
        SimpleStat(); // Default constructor
        bool addNumber(int);
        double getAverage();
        int getLargest() { return largest; }
        int getCount() { return count; }
};
```

Checkpoint

7.15 A private class member function can be called by

A) any other function
B) only public functions in the same class
C) only private functions in the same class
D) any function in the same class

7.16 When an object is passed to a function, a copy of it is made if the object is

A) passed by value
B) passed by reference
C) passed by constant reference
D) any of the above

7.17 If a function receives an object as an argument and needs to change the object's member data, the object should be

A) passed by value
B) passed by reference
C) passed by constant reference
D) none of the above

7.18 True or false: Objects can be passed to functions, but they cannot be returned by functions.

7.19 True or false: When an object is passed to a function,

arrays

An array allows you to store and work with multiple values of the same data type.

- holds multiple values

  ex.

  int hours[6]; // holds 6 values ie: { 1, 2, 3, 4, 5, 6 } within the hours array.

accessing and modifying arrays

The individual elements of an array are assigned unique subscripts. These subscripts are used to access the elements.

- A subscript is used as an index to pinpoint a specific element within an array

NOTE: Subscript numbering in C++ always starts at zero. The subscript of the last element in an array is one less than the total number of elements in the array. This means that in the array the element hours[6] does not exist. The last element in the array is hours[5].

NOTE: The expression hours[0] is pronounced “hours sub zero.” You would read this assignment statement as “hours sub zero is assigned twenty.”

NOTE: Because values have not been assigned to the other elements of the array, question marks are used to indicate that the contents of those elements are unknown. If an array holding numeric values is defined globally, all of its elements are initialized to zero by default. Local arrays however, have no default initialization value.

NOTE: It is important to understand the difference between the array size declarator and a subscript. The number inside the brackets in an array definition is the size declarator. It specifies how many elements the array holds. The number inside the brackets in an assignment statement or any statement that works with the contents of an array is a subscript. It specifies which element is being accessed.

- Inputting into arrays

  ex.

```
const int NUM_EMPLOYEES = 6;
int hours[NUM_EMPLOYEES];                    // Holds hours worked for 6 employees

// Input the hours worked by each employee
cout << "Enter the hours worked by " << NUM_EMPLOYEES
<< " employees: ";
        cin >> hours[0];
        cin >> hours[1];
        cin >> hours[2];
        cin >> hours[3];
        cin >> hours[4];
        cin >> hours[5];
```

(this stores all int inputs into the int array ‘hours’)

Checkpoint

8.1 Define the following arrays:

A) empNum, a 100-element array of ints
B) payRate, a 25-element array of doubles
C) miles, a 14-element array of longs
D) stateCapital, a 50-element array of string objects.
E) lightYears, a 1,000-element array of doubles

8.2 What's wrong with the following array definitions?

```
int readings[-1];
double measurements[4.5];
int size;
string name[size];
```

8.3 What would the valid subscript values be in a four-element array of doubles?

8.4 What is the difference between an array's size declarator and a subscript?

8.5 What is "array bounds checking"? Does C++ perform it?

8.6 What is the output of the following code?

```
int values[5], count;

for (count = 0; count < 5; count++)
        values[count] = count + 1;
for (count = 0; count < 5; count++)
        cout << values[count] << endl;
```

8.7 Complete the following program skeleton so it will have a 10-element array of int values called fish. When completed, the program should ask how many fish were caught by fishermen 1 through 10, and store this information in the array. Then it should display the data.

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

int main ()
{
const int NUM_MEN = 10;

// Define an array named fish that can hold 10 int values.
// You must finish this program so it works as described above.

return 0;
}
```

NOTE: You must specify an initialization list if you leave out the size declarator.
Otherwise, C++ doesn't know how large to make the array.

passing arrays as arguments to functions

Individual elements of arrays and entire arrays can both be passed as arguments to functions.

- Single elements of arrays can be passed as functions like any other variable.

```
// This program demonstrates that an array element
// can be passed to a function like any other variable.
#include <iostream>
using namespace std;

void showValue(int); // Function prototype

int main()
{
const int ARRAY_SIZE = 8;
int collection[ARRAY_SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};

for (int index = 0; index < ARRAY_SIZE; index++)
        showValue(collection[index]);
cout << endl;
return 0;
}

void showValue(int num)
{
```

- Using it as a function input:

```
void showValues (arrayType nums, int size)
{
        for (int index = 0; index < size; index++)
        cout << nums[index] << " ";
        cout << endl;
}
```

NOTE: In C++ when a regular variable is passed to a function and an & precedes its name, it means that the function is receiving a reference to the memory address where a variable is stored. An array name, however, is already a memory address. That is, instead of holding a value, it holds the starting address of where the array is located in memory. Therefore, an & should not be used with it.

- Sometimes you need to prevent another function from modifying the array. Using const prevents this from happening.

```
void showValues(const arrayType, int)                    // Function prototype
void showValues(const arrayType nums, int size)          // Function header
```

Checkpoint

8.15 Write a typedef statement that makes the name TenInts an alias for an array that holds 10 integers.

8.16 When an array name is passed to a function, what is actually being passed?

8.17 What is the output of the following program? (You may need to consult the ASCII table in Appendix A.)

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

// Function prototypes
void fillArray(char [], int)
void showArray(const char [], int)

int main ()
{
	char prodCode[8] = {'0', '0', '0', '0', '0', '0', '0', '0'};
	fillArray(prodCode,8);
	showArray(prodCode,8);
	return 0;
}

// Definition of function fillArray (Hint: 65 is the ASCII code for 'A'.)
void fillArray(char arr[], int size)
{
	char code = 65;
	for (int k = 0; k < size; code++, k++)
	arr[k] = code;
}

// Definition of function showArray
void showArray(const char codes[], int size)
{
	for (int k = 0; k < size; k++)
	cout << codes[k];
}
```

8.18 The following program skeleton, when completed, will ask the user to enter 10 integers, which are stored in an array. The function avgArray, which you must write, should calculate and return the average of the numbers entered.

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

// Write the avgArray function prototype here.
// It should have a const array parameter.

int main()
{
	const int SIZE = 10;
	int userNums[SIZE];

	cout << "Enter 10 numbers: ";
		for (int count = 0; count < SIZE; count++)
		{
			cout << "#" << (count + 1) << " ";
			cin >> userNums[count];
		}
	cout << "The average of those numbers is ";
	cout << avgArray(userNums, SIZE) << endl;
	return 0;
}
// Write the avgArray function here.
```

multidimensional arrays

A two-dimensional array is like several identical arrays put together. It is useful for storing multiple sets of data.

```
// Nested loops are used to fill the array with quarterly
// sales figures for each division and to display the data

for (div = 0; div < NUM_DIVS; div++)
{
        for (qtr = 0; qtr < NUM_QTRS; qtr++)
{

cout << "Division " << (div + 1) << ", Quarter " << (qtr + 1) << ": $";
datafile >> sales[div][qtr];
cout << sales[div][qtr] << endl;
}
cout << endl;
```

(this line uses a nested loop to write the numbers into a file. A double array is used.)

- Double arrays are just like arrays but have two layers.
- array[ ][ ] is the call. ie. myArray[2][4]
    - This array would have a value like this:

      myArray[2][4] = { {2,3,1,3}, {1,4,5,2] }

      int hours[3][2] = {{8, 5}, {7, 9}, {6, 3}};

      The same statement could also be written as

      ```
      int hours[3][2] = {{8, 5},
                         {7, 9},
                         {6, 3}};
      ```

- The array can be passed as shown.
  void showArray(const int array[][NUM_COLS], int numRows)

- Summing all data from a 2D array
- ```
  for (div = 0; div < NUM_DIVS; div++)
  {
          for (qtr = 0; qtr < NUM_QTRS; qtr++)
          totalSales += sales[div][qtr];
  }
  ```
  ```

-Summing the rows of a 2D array

```
const int NUM_STUDENTS = 3;                          // Number of students
const int NUM_SCORES = 5;                            // Number of test scores
double total;                                        // Accumulator
double average; // Holds a given student's average
double scores[NUM_STUDENTS][NUM_SCORES] = {{88, 97, 79, 86, 94},
                                          {86, 91, 78, 79, 84},
                                          {82, 73, 77, 82, 89}};

// Sum each student's test scores so his or her
// average can be calculated and displayed

for (int row = 0; row < NUM_STUDENTS; row ++)
{
        // Reset accumulator to 0 for this student
        total = 0;

        // Sum a row
        for (int col = 0; col < NUM_SCORES; col++)
                total += scores[row][col];

        // Compute and display the average for this student
        average = total / NUM_SCORES;
        cout << "Score average for student " << (row + 1) << " is " << average << endl;
}
```

- Summing the columns of a 2D array.

```
const int NUM_STUDENTS = 3;                          // Number of students
const int NUM_SCORES = 5;                            // Number of test scores
double total;                                        // Accumulator
double average; // Holds average score on a given test
double scores[NUM_STUDENTS][NUM_SCORES] = {{88, 97, 79, 86, 94},
                                          {86, 91, 78, 79, 84},
                                          {82, 73, 77, 82, 89}};
// Calculate and display the class
// average for each test
for (int col = 0; col < NUM_SCORES; col++)
{
        // Reset accumulator to 0 for this test
        total = 0;
        // Sum a column
        for (int row = 0; row < NUM_STUDENTS; row++)
                total += scores[row][col];
```

```
        // Compute and display the class average for this test
        average = total / NUM_STUDENTS;
        cout << "Class average for test " << (col + 1) << " is " << average << endl;
}
```

Array with 3 or more dimensions
The third dimension becomes most of a Zaxis. 2D is a XY, while a single array is just an X.

- Written like
    - double seat[3][5][8]

Checkpoint

8.19 Define a two-dimensional array of ints named grades. It should have 30 rows and 10 columns.

8.20 How many elements are in the following array?
double sales[6][4];

8.21 Write a statement that assigns the value 56893.12 to the first column of the first row of the sales array defined in question 8.20.

8.22 Write a statement that displays the contents of the last column of the last row of the sales array defined in question 8.20.

8.23 Define a two-dimensional array named settings large enough to hold the table of information below. Initialize the array with the values in the table. 8.24 Fill in the empty table below so it shows the contents of the following array:

```
int table[3][4] = {{2, 3}, {7, 9, 2}, {1}};
```

8.25 Write a function called displayArray7. The function should accept a two dimensional array as an argument and display its contents on the screen. The function should work with any of the following arrays:

```
int hours[5][7];
int stamps[8][7];
int autos[12][7];
```

8.26 A DVD rental store keeps DVDs on 50 racks with 10 shelves each. Each shelf holds 25 DVDs. Define a 3D array to represent this storage system.

C-style strings

C-strings provide another way to store and work with strings.

- Here is a statement that defines word to be an array of characters that will hold a C-string and initializes it to "Hello".

```
char word[10] = "Hello";
```

- The only difference is the [10] that follows the name of the variable. This is called a size declarator.
- Because one space must be reserved for the null terminator, word can only hold a string of up to nine characters.

```
// This program uses the strcpy function to copy one C-string to another.
#include <iostream>
using namespace std;

int main()
{
	const int SIZE = 12;
	char name1[SIZE],
	name2[SIZE];

	strcpy(name1, "Sebastian");
	cout << "name1 now holds the string " << name1 << endl;

	strcpy(name2, name1);
	cout << "name2 now also holds the string " << name2 << endl;

	return 0;
}
```

- Keeping Track of a How Much a C-String Can Hold

```
char word[5];
cin >> setw(5) >> word;
char word[5];
cin.width(5);
cin >> word;
```

- There are three important points to remember about the way cin handles field widths:
  - The field width only pertains to the very next item entered by the user.
  - To leave space for the '\0' character, the maximum number of characters read and stored will be one less than the size specified.
  - If cin comes to a whitespace character before reading the specified number of characters, it will stop reading.

Checkpoint

3.26 Will the following string literal fit in the space allocated for name? Why or why not?

```
char name[4] = "John";
```

3.27 If a program contains the definition string name; indicate whether each of the following lettered program statements is legal or illegal.

A) cin >> name;
B) cin.getline(name, 20);
C) cout << name;
D) name = "John";

3.28 If a program contains the definition char name[20]; indicate whether each of the following lettered program statements is legal or illegal.

A) cin >> name;
B) cin.getline(name, 20);
C) cout << name;
D) name = "John";

pointer, & address operator

Every variable is assigned a memory location whose address can be retrieved using the address operator &. The address of a memory location is called a pointer.

- Each byte used has a memory location and unique address.
- & is an address operator that can be used to retrieve the actual address or memory location. Even address are typically used as they are much faster at being accessed than odd.
- The address are typically stored as hexadecimal; ie 3x483 or something to that nature.

pointer variables, using pointers

A pointer variable is a variable that holds addresses of memory locations.

- They are like any other data values, memory address, or pointer values, they can be stored in variables of the appropriate type. They are called pointer variables but simply called pointers.

  ex.

  ```
  int *ptr;
  ```

- This indicates that the pointer variable can only be used to, or point to, or hold address of an integer.
- can also be declared as int* ptr

  ex.

  ```
  int main()
  ```

```
{
        int x = 25;             //int variable
        int *ptr;               //Pointer variable, can point to int

        ptr = &x;               //stores the address of x in ptr. (hexademal)
        cout << "The value of x is " << x << endl;
        cout << "The address of x is " << ptr <, endl;
        return 0;
}
```

(will return x is 25, then address is a hexadecimal) Pointers allow us to directly access the information instead of copying, almost like a referenced variable.

- you can use a pointer indirectly to access AND modify the variable that it is pointed to.
- When the indirection operator is placed in front of a pointer variable name, it dereferences the pointer.
- Since we are working directly with that address, we are changing that value. *ptr = x is literally now x.

NOTE: So far you've seen three different uses of the asterisk in C++:

• As the multiplication operator, in statements such as
*distance = speed * time;*
• In the definition of a pointer variable, such as
*int *ptr;*
• As the indirection operator, in statements such as
**ptr = 100;*

Arrays and pointers

Array names can be used as pointer constants, and pointers can be used as array names.

- Array name is really just a pointer.
  ex.

```
int main()
{
        short numbers[] = {10, 20, 30, 40, 50};

        cout << "The first element of the array is ";
        cout << *numbers << endl;
        return 0;
}
```

outputs: The first element of the array is 10

- Parenthesis are IMPORTANT when dealing with pointers.
- Using mathematical statements do not work like regular variables.

ex.

*numbers + 1 = value of numbers + 1
while
*(numbers + 1) adds to address, so moves down the “array”

NOTE: The parentheses are critical when adding values to pointers. The * operator has precedence over the + operator, so the expression *numbers + 1 is not equivalent to *(numbers + 1). The expression *numbers + 1 adds one to the contents of the first element of the array, while *(numbers + 1) adds one to the address in numbers, then dereferences it.

WARNING! Remember that C++ performs no bounds checking with arrays. When stepping through an array with a pointer, it’s possible to give the pointer an address outside of the array.

- When using an array’s address, you do not need to use the address operator, that would be incorrect. You can use it to to get the address of an individual element in an array.
-
- The only difference between array names and pointer variables is that you cannot change the address an array name points to. For example, given the following definitions:

  double readings[20], totals[20];
  double *dptr;

- These statements are legal:

  dptr = readings; // Make dptr point to readings
  dptr = totals; // Make dptr point to totals

- But these are illegal:

  readings = totals; // ILLEGAL! Cannot change readings
  totals = dptr; // ILLEGAL! Cannot change totals

- Array names are pointer constants. You can’t make them point to anything but the array they represent.

Initializing Pointers

Pointers may be initialized with the address of an existing object.

ex.

int myValue;
*int *pint = &myValue;*

- In most computers, memory address is 0 is inaccessible, so making a variable 0 means the value cannot be anything. Typically can be defined as NULL, which has a value of 0.
-

Checkpoint

10.1 Write a statement that displays the address of the variable count.

10.2 Write a statement defining a variable dPtr. The variable should be a pointer to a double.

10.3 List three uses of the * symbol in C++.

10.4 What is the output of the following program?

```
#include <iostream>
using namespace std;
int main()
{
        int x = 50, y = 60, z = 70;
        int *ptr;
        cout << x << " " << y << " " << z << endl;
        ptr = &x;
        *ptr *= 10;
        ptr = &y;
        *ptr *= 5;
        ptr = &z;
        *ptr *= 2;
        cout << x << " " << y << " " << z << endl;
        return 0;
}
```

10.5 Rewrite the following loop so it uses pointer notation (with the indirection operator) instead of subscript notation.

```
for (int x = 0; x < 100; x++)
cout << array[x] << endl;
```

10.6 Assume ptr is a pointer to an int and holds the address 12000. On a system with 4-byte integers, what address will be in ptr after the following statement?

```
ptr += 10;
```

10.7 Assume pint is a pointer variable. For each of the following statements, determine whether the statement is valid or invalid. For those that are invalid, explain why.

A) pint++;
B) --pint;
C) pint /= 2;
D) pint *= 4;
E) pint += x; // Assume x is an int.

10.8 For each of the following variable definitions, determine whether the statement is valid or invalid. For those that are invalid, explain why.

A)

```
int ivar;
int *iptr = &ivar;
```

B)

```
int ivar, *iptr = &ivar;
```

C)

```
float fvar;
int *iptr = &fvar;
```

D) `int nums[50], *iptr = nums;`

E)

```
int *iptr = &ivar;
int ivar;
```

Comparing Pointers

C++'s relational operators may be used to compare pointer values.

- Pointers may be compared by using any of C++'s relational operators:

  `>  <  ==  !=  >=  <=`

NOTE: Comparing two pointers is not the same as comparing the values the two pointers point to. For example, the following if statement compares the addresses stored in the pointer variables ptr1 and ptr2:

```
if (ptr1 < ptr2)
```

The following statement, however, compares the values that ptr1 and ptr2 point to:

```
if (*ptr1 < *ptr2)
```

- Pointers and arrays are good, since pointers will not go beyond the boundaries of the array

Pointers as Function Parameters

A pointer can be used as a function parameter. It gives the function access to the original argument, much like a reference parameter does.

- Almost like passing an argument by reference, which is much easier.
- Don't have to deal with dereferencing of other mechanics.
- Pointers are better at dealing with string values.

ex.

```
int main()
{
        int number;

        // Call getNumber and pass the address of number
        getNumber(&number);

        // Call doubleValue and pass the address of number
        doubleValue(&number);

        // Display the value in number
        cout << "That value doubled is " << number << endl;
        return 0;
}

void getNumber(int *input)
{
cout << "Enter an integer number: ";
cin >> *input;
}

void doubleValue(int *val)
{
        *val *= 2;
}
```

Enter an integer number: 10[Enter] *That value doubled is 20*

WARNING!

It's critical that the indirection operator be used in the previous statement. Without it, cin would store the value entered by the user in input, as if the value were an address. If this happens, input will no longer point to the number variable in function main. Subsequent use of the pointer will result in erroneous, if not disastrous, results.

NOTE:

The two previous statements could be combined into the following statement:

```
sum += *array++;
```

The * operator will first dereference array, then the ++ operator will increment the address in array.

Pointers to Constants and Constant Pointers

A pointer to a constant may not be used to change the value it points to; a constant pointer may not be changed after it has been initialized.

ex.

```
const int SIZE = 6;
const double payRates[SIZE] = { 18.55, 17.45,
                                12.85, 14.97,
                                10.35, 18.89 };
```

- payRates is an array of const doubles, this means each element in array is a const double.
- the compiler will not be able to change the contents.
- If we want to pass into a pointer parameter, it must be declared const double as well.
-

```
void displayPayRates(const double *rates, int size)
{
    // Set numeric output formatting
    cout << setprecision(2) << fixed << showpoint;

    // Display all the pay rates
    for (int count = 0; count < size; count++)
    {
        cout << "Pay rate for employee " << (count + 1) << " is $" << *(rates +
        count) << endl;
    }
}
```

- A const pointer can also receive address of a non-constant item.

NOTE:

When writing a function that uses a pointer parameter, and the function is not intended to change the data the parameter points to, it is always a good idea to make the parameter a pointer to const. Not only will this protect you from writing code in the function that accidentally changes the argument, but the function will be able to accept the addresses of both constant and non-constant arguments.

- Here is the difference between a pointer to const and a const pointer:

    - A pointer to const points to a constant item. The data that the pointer points to cannot change, but the pointer itself can change.
    - With a const pointer, it is the pointer itself that is constant. Once the pointer is initialized with an address, it cannot point to anything else.

- The following code shows an example of a const pointer.

```
int value = 22;
int * const ptr = &value;
```

Constant Pointers to Constants

You can also have constant pointers to constants.

ex.

```
int value = 22;
const int * const ptr = &value;
```

- In this, ptr is a const pointer to a const int.

dynamic memory allocation

Variables may be created and destroyed while a program is running.

- When allocating data, it fills up a memory location. If it is not removed, evertually the computer will run out of space.
- Using pointers is the only way to address this issue.

  ex.

```
int *iptr;
iptr = new int[100];
for (int count = 0; count < 100; count++)
    iptr[count] = 1;
```

- This will write a large amount of data that will fill the storage.
- Every call to new allocates storage from a special area of the programs memory called the heap. Eventually this will become depleted and cause any additional calls to new fail.
- C++ will throw bad_alloc as a warning
- an Exception is a mechanism for notifying the program that something has gone wrong.
- This issue can be solved by removing the block of memory the “new” has created.

  ex.

```
delete iptr;
iptr = 0;
// Must set to 0 or NULL

in arrays:
    delete [ ] iptr;
    iptr = 0;

//Must include brackets between delete and pointer.
```

WARNING! Only use pointers with delete that were previously used with new. If you use a pointer with delete that does not reference dynamically allocated memory, unexpected problems could result!

Dangling Pointers and Memory Leaks

A pointer is said to be dangling if it is pointing to a memory location that has been freed by a call to delete.

- Can be difficult to trace.
- avoided by setting pointers to 0 once they are deleted.

- A memory leak occurs when you forget to delete.

Returning Pointers from Functions

Functions can return pointers, but you must be sure the item the pointer references still exists.

- For example, if the function is passedthe value 4, it returns an array whose elements are 1, 4, 9, and 16.
  ex.

```
int *squares(int n)
{
        // Allocate an array of size n
        int *sqarray = new int[n];
        // Fill the array with squares
        for (int k = 0; k < n; k++)
        sqarray[k] = (k+1) * (k+1);
        // Return base address of allocated array
        return sqarray;
}
```

- A function can safely return a pointer to dynamically allocated storage that has not yet been deleted.
  ex.

```
int *errSquares(int n)
{
        // Assume n is less than 100, use local array
        int array[100];

        // Fill the array with squares
        for (int k = 0; k < n; k++)
        array[k] = (k+1) * (k+1);

        // Return base address of local array
        return array;
}
```

  - A call such as
    int * arr = errSquares(5);
  - Will return
    cout << arr[0];

NOTE: Storage for a static local variable is not deallocated upon return, so a function returning a pointer to such a variable will not trigger the kind of error we are talking about here. Such a function, however, may cause other types of errors whose discussion is beyond the scope of this book.

Checkpoint

10.9 Assuming array is an array of ints, which of the following program segments will display "True" and which will display "False"?

A)

```
if (array < &array[1])
cout << "True";
else
cout << "False";
```

B)

```
 if (&array[4] < &array[1])
cout << "True";
else
cout << "False";
```

C)

```
if (array != &array[2])
cout << "True";
else
cout << "False";
```

D)

```
if (array != &array[0])
cout << "True";
else
cout << "False";
```

10.10

Give an example of the proper way to call the following function in order to negate the variable int num = 7;

```
void makeNegative(int *val)
{
        if (*val > 0)
        *val = -(*val);
}
```

10.11 Complete the following program skeleton. When finished, the program should ask the user for a length (in inches), convert that value to centimeters, and display the result. You are to write the function convert. (Note: 1 inch = 2.54 cm. Do not modify function main.)

```
#include <iostream>
#include <iomanip>
using namespace std;

// Write your function prototype here.


int main()
{
        double measurement;
        cout << "Enter a length in inches, and I will convert\n";
        cout << "it to centimeters: ";
        cin >> measurement;
        convert(&measurement);
        cout << setprecision(4);
        cout << fixed << showpoint;
        cout << "Value in centimeters: " << measurement << endl;
        return 0;
}

//
// Write the function convert here.
//
```

10.12
Look at the following array definition:

```
const int numbers[SIZE] = { 18, 17, 12, 14 };
```

Suppose we want to pass the array to the function processArray in the following manner:

```
processArray(numbers, SIZE);
```

Which of the following function headers is the correct one for the processArray function?

A) void processArray(const int *array, int size)
B) void processArray(int * const array, int size)

10.13 Assume ip is a pointer to an int. Write a statement that will dynamically allocate an integer variable and store its address in ip, then write a statement that will free the memory allocated in the statement you just wrote.

10.14 Assume ip is a pointer to an int. Write a statement that will dynamically allocate an array of 500 integers and store its address in ip, then write a statement that willfree the memory allocated in the statement you just wrote.

10.15 What is a null pointer?

10.16 Give an example of a function that correctly returns a pointer.

10.17 Give an example of a function that incorrectly returns a pointer.

Pointers to Class objects

Pointers and dynamic memory allocation can be used with class objects and structures.

you can declare a pointer to Rectangle and create a Rectangle object by writing

```
Rectangle *pRect;          // Pointer to Rectangle
Rectangle rect;            // Rectangle object
```

and you can assign the address of rect to pRect as follows:

```
pRect = &rect;
```

- Accessing members of the class by

```
(*pRect).width
```

- Shortly you can use the SPO or structure pointer operator. ->

```
pRect->width = 10;
pRect->height = 20;
```

- You can set a single object like this

```
pRect = new Rectangle;
pRect->width = 10;
pRect->height = 3;
        or
pRect = new Rectangle(10, 30);
```

- Pointers to class variables can be passed to functions as parameters as well.
- The function receiving the pointer can use it to access or modify members.

Table 10-1
Dereferencing Pointers to Structures

Expression	Description
s->m	s is a pointer to a structure variable or class object, and m is a member. This expression accesses the m member of the structure or class object pointed to by s.
*a.p	a is a structure variable or class object and p, a pointer, is a member of a. This expression accesses the value pointed to by a.p.
(*s).m	s is a pointer to a structure variable or class object, and m is a member. The * operator dereferences s, causing the expression to access the m member of the object
*s.	This expression is the same as s->m.
*s->p	s is a pointer to a structure variable or class object and p, a pointer, is a member of the object pointed to by s. This expression accesses the value pointed to by s->p.
*(*s).p	s is a pointer to a structure variable or class object and p, a pointer, is a member of the object pointed to by s. This expression accesses the value pointed to by (*s).p. This expression is the same as *s->p.