Week 7

Intro to Computer Science - CSI 1430

Hello and welcome to the weekly resources for CSI 1430: Introduction to Computer Science! This week's resource will be covering the course material that will be covered in week 7 of the course.

Another light week in terms of content. Most of how programming is learned is taking a huge topic, learning it, and then doing a lot of practice with it. Last week, loops were introduced, and that can be one of the most important things to know in programming. So make sure you take time to break apart the major topics discussed last week in looping, and do lots of practice. **Just as a reminder:** These documents are intended for going over concepts that are discovered in class, as they are a concise and to the point review of the materials. But remember, these documents are not a replacement for lecture and they cannot cover everything, so make sure you get help on any specific concepts you struggle with as well as go over lecture notes, programs, and the textbook.

Reminder: If you have any questions about these study guides, group tutoring sessions, private 30 minutes tutoring appointments, the Baylor Tutoring Youtube channel or any tutoring services we offer, please visit our website https://baylor.edu/tutoring or call our drop in center during open business hours. Monday-Thursday 9am - 8pm on class days (254) 710-4135.

Keywords: *Arrays, Output Formatting*

TOPIC OF THE WEEK

Arrays

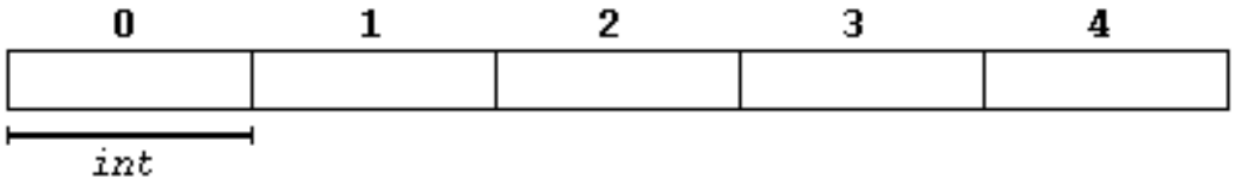
Say we want to make a program that takes in 100 numbers input by the user, store them all, and display them to the screen. To do this we could make 100 different variables to store the data, or we could invoke the use of an array! As we start to create programs that require the use of a lot of variables to store the information we need, we don't want to hardcode a bunch of variables of the same type. Rather, using an array we can group these variables together.

Arrays are one of the most commonly used structures in programming. An array is a collection of elements of the same type that are able to be referenced based on their location within the array. To explain how arrays work, let's take a look at a basic example.

Say we want to declare an array of 5 integers and call the array numList. To do this we can declare an array as such:

int numList[5];

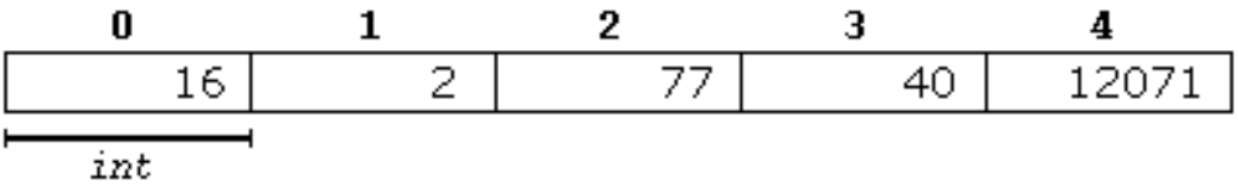
To conceptualize what a list is doing, look towards the image below. The array numList was declared to have 5 spaces. Each space is able to store a separate integer variable. Each variable is able to be put into the array based on a specific position in the array.



Adding data to an array is very similar to how to set a variable to a specific value. You must specify what location is being stored to on the left hand side of an equals, and the right hand side will provide the value of what should be stored in that data. For example, let's fill our array with some numbers:

numList[0] = 16;
numList[1] = 2;
numList[2] = 77
numList[3] = 40;
numList[4] = 12071;

As you can see, the location within the array that we want to store the data is being specified on the right, and the value that will be stored to it is on the left. After these statements are processed, the array should resemble the following:



Lastly, to access positions within the array, we can reference them by the position number that they were inserted into the array by. Referencing *numList[2]* in this example would yield 77.

HIGHLIGHTS

Output Formatting

When we are outputting in C++, we often will want to do manipulation of that more accurately or neatly presents the information that we are displaying. In these cases, we want to invoke the use of some formatting commands that can be used to make the numerical or even string data that we are presenting be displayed correctly.

Almost all of this manipulation can be achieved through the use of the library ***iomanip***. To use iomanip we have to include the library, since it is not prebuilt into C++. To achieve this, just #include <iomainp> just like you would include any other library such as *iostream* or *cmath*.

For numerical data, the most common ways of manipulating data include:

Manipulator	Description	Example
fixed	Use fixed-point notation. From <ios>	`// 12.340000` `cout << fixed << 12.34;`
scientific	Use scientific notation. From <ios>	`// 1.234000e+01` `cout << scientific << 12.34;`

setprecision(p)	If stream has not been manipulated to fixed or scientific:
	Sets max number of digits in number
		`// 12.3`
		`cout << setprecision(3) << 12.34;`
		
		`// 12.34`
		`cout << setprecision(5) << 12.34;`

	If stream has been manipulated to fixed or scientific:
	Sets max number of digits in fraction only (after the decimal point).
	From <iomanip>
		`// 12.3`
		`cout << fixed << setprecision(1) << 12.34;`
		
		`// 1.2e+01`
		`cout << scientific << setprecision(1) << 12.34;`

showpoint	Even if fraction is 0, show decimal point and trailing 0s.
	Opposite is noshowpoint.
	From <ios>
		`// 99`
		`cout << setprecision(3) << 99.0;`
		
		`// 99.0`
		`cout << setprecision(3) << showpoint << 99.0;`

For strings, the most common ways of manipulating data include:

Manipulator	Description	Example (for item "Amy")
setw(n)	Sets the number of characters for the next output item only (does not persist, in contrast to other manipulators). By default, the item will be right-aligned, and filled with spaces. From <iomanip>	For n=7: " Amy"
setfill(c)	Sets the fill to character c. From <iomanip>	For c='*': "****Amy"
left	Changes to left alignment. From <ios>	"Amy "
right	Changes back to right alignment. From <ios>	" Amy"

These manipulators get super important when we start to talk about things like money. Dollar amounts are limited to 2 decimal places, and therefore should also be expressed as such. However, some mathematical processes that happen with numbers can yield variable data that breaks this rule. As programmers, we must account for that to make sure the data we are presenting makes sense. In the example of money, we have to make sure we use setprecision to limit the decimal places to 2.

CHECK YOUR LEARNING

1. **Initialize an array named "example" that holds string types and has 12 positions.**
2. **Display a variable named dblVariable, but use iomanip to set the output to 4 decimal places with fixed-point notation. Assume the variable has already been declared and initialized.**
3. **All arrays start with position 1, and continue to however long you have declared the array to be. True or false?**

THINGS STUDENTS MAY STRUGGLE WITH

1. Note that arrays start at an index of 0. This is super important to remember due to making sure you dont cause issues within your program. If you initialize an array to have 10 spaces, then the range of positions within the array will be 0 to 9, NOT 1-10. Trying to access the 10 position will cause an error.
2. Take time to get used to the different *iomanip* manipulations… Sometimes they can not behave like you would like them to and need a little bit of playing around with to get them to format your data how you would like it to.
3. When talking about places after a decimal point, always use the fixed manipulator to precede the setprecision() manipulator. setprecision() by itself sets the number of digits to be printed before and after the decimal place.

Thanks for checking out these weekly resources! Don't forget to check out our website for group tutoring times, video tutorials and lots of other course resources: https://baylor.edu/tutoring. The answers to the 'Check Your Learning' questions are below.

Answers:

1. *string example[12];*
2. cout << fixed << setprecision(4) << dblVariable;
3. False, all arrays start with the 0 position.

Note: All tables were taken from the Baylor CS 1430 zyBook