# 8.1 Why pointers: Pass by pointer example

A challenging but powerful programming construct is something called a *pointer*. This section illustrates one example's beneficial usage of pointers, namely pass by pointer function parameters.

A function can only return one value. But consider a desired function that converts total inches into feet and inches, e.g., 95 inches would be converted to 7 feet and 11 inches. To effectively return two values, the function can be defined with two ***pass by pointer*** parameters, by putting a * before a parameter name, and & before the corresponding argument variable[parm].



The & passes the variable's memory address, known as a ***pointer***, rather than the variable's value. The * before the parameter name indicates the parameter is a pointer. The function's statements can update each argument variable's memory location, effectively "returning" a value. The technique is also known as ***pass by reference***, but the term pass by pointer avoids confusion with pass by reference parameters in C++ programs (which are different), and to more accurately describe this technique.

The following animation illustrates how pass by value does not work to return two values from a function.

**PARTICIPATION ACTIVITY** 8.1.1: Pointer example: Without pass by pointer parameters.

Start

```
#include <stdio.h>
#include <stdlib.h>

void ConvFeetInches(int totDist,
                    int inFeet, int inInches) {

   inFeet   = totDist / 12;
   inInches = totDist % 12;
   return;
}

int main(void) {
   int initMeasure = 45;
   int resFeet = 0;
   int resIn = 0;

   ConvFeetInches(initMeasure, resFeet, resIn);
   printf("%d feet %d inches\n", resFeet, resIn);

   return 0;
}
```

| | | |
|---|---|---|
| 90 | 45 | initMeasure |
| 91 | 0 | resFeet |
| 92 | 0 | resIn |
| 93 | | |
| 94 | | |
| 95 | | |
| 96 | | |
| 97 | | |

```
0 feet 0 inches
```

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

The following animation illustrates how pass by pointer effectively enables a function to return two values.

**PARTICIPATION ACTIVITY** 8.1.2: Pointer example: Pass by pointer parameters.

Start

```
#include <stdio.h>
#include <stdlib.h>

void ConvFeetInches(int totDist,
                    int* inFeet, int* inInches) {

   *inFeet   = totDist / 12;
   *inInches = totDist % 12;
   return;
}

int main(void) {
   int initMeasure = 45;
   int resFeet = 0;
   int resIn = 0;

   ConvFeetInches(initMeasure, &resFeet, &resIn);
   printf("%d feet %d inches\n", resFeet, resIn);

   return 0;
}
```

| | | |
|---|---|---|
| 90 | 45 | initMeasure |
| 91 | 3 | resFeet |
| 92 | 9 | resIn |
| 93 | | |
| 94 | | |
| 95 | | |
| 96 | | |
| 97 | | |

```
3 feet 9 inches
```



*Pass by pointer parameters should be used only when the output values are tightly related.* New programmers commonly create one function with two outputs (using pass by pointer) to reduce coding, where two functions would have been better. For example, defining two functions `int StepsToFeet(int baseSteps)` and `int StepsToCalories(int baseCalories)` is better than defining a single function `void StepsToFeetAndCalories(int baseSteps, int* feetTot, int* caloriesTot)`. Defining separate functions supports modular development, and enables use of the functions in an expression as in `if (StepsToFeet(baseSteps) < 100)`.

Good candidates for multiple pass by pointer parameters might include computing the number of each type of coin to give as change, whose function might be `void ComputeChange(int totCents, int* numQuarters, int* numDimes, int* numNickels, int* numI` Another example is converting from polar coordinates to Cartesian coordinates, whose function might be `void PolarToCartesian(int radialPol, int anglePol, int* xCar, int* yCar)`. In both situations, the two outputs are tightly related.



**PARTICIPATION ACTIVITY** 8.1.3: Calculating change.

Complete the program to compute the number of quarter, dime, nickel, and penny coins that equal total cents, using the fewest coins (i.e., using the most possible of a larger coin first).

Load default template... Run

```

#include <stdio.h>

void ComputeChange(int totCents) { // FIXME add four pass b
   printf("FIXME: Finish this function.\n");
```

```
    return;
}

int main(void) {

   int totalCents = 67;     // Total amount of change needed
   int quartersChange = 0; // Number of quarters used for c
   int dimesChange    = 0; // Number of dimes used for chan
   int nickelsChange  = 0; // Number of nickels used for ch
   int penniesChange  = 0; // Number of pennies used for ch

   ComputeChange(totalCents); // FIXME Add four pointer arg

   printf("Change for %d cents is:\n", totalCents);
   printf("  %d quarters\n", quartersChange);

```



**PARTICIPATION ACTIVITY** 8.1.4: Why pointer.

1) Complete the function declaration for MyFct with input parameters w and x, and "output" parameters y and z, in that order, all dealing with type int.

```
void MyFct(
____________________);
```

Check Show answer

2) Call a function MyFct that returns void, with four parameters, the last two being pointers. Call the function with argument variables a, b, c, and d, in that order, all being of type int.

____________________

Check Show answer



Exploring further:

- Pointers tutorial from msdn.microsoft.com

**CHALLENGE ACTIVITY** 8.1.1: Calling a function that has pass by pointer parameters.

Write a function call with arguments tensPlace, onesPlace, and userInt. Be sure to pass the first two arguments as pointers. Sample output for the given program:

```
tensPlace = 4, onesPlace = 1
```



```c
#include <stdio.h>

void SplitIntoTensOnes(int* tensDigit, int* onesDigit, int DecVal){
   *tensDigit = (DecVal / 10) % 10;
   *onesDigit = DecVal % 10;
   return;
}

int main(void) {
   int tensPlace = 0;
   int onesPlace = 0;
   int userInt = 0;

   userInt = 41;

   /* Your solution goes here  */

   printf("tensPlace = %d, onesPlace = %d\n", tensPlace, onesPlace);

   return 0;
}
```

Run

**CHALLENGE ACTIVITY** 8.1.2: Pass by pointer: Adjusting start/end times.

Define a function UpdateTimeWindow() with parameters timeStart, timeEnd, and offsetAmount. Each parameter is of type int. The function adds offsetAmount to each of the first two parameters. Make the first two parameters pass by pointer. Sample output for the given program:

```
timeStart = 3, timeEnd = 7
timeStart = 5, timeEnd = 9
```



```c
#include <stdio.h>

// Define void UpdateTimeWindow(...)

/* Your solution goes here  */

int main(void) {
```

```
    int timeStart = 0;
    int timeEnd = 0;
    int offsetAmount = 0;

    timeStart = 3;
    timeEnd = 7;
    offsetAmount = 2;
    printf("timeStart = %d, timeEnd = %d\n", timeStart, timeEnd);

    UpdateTimeWindow(&timeStart, &timeEnd, offsetAmount);
    printf("timeStart = %d, timeEnd = %d\n", timeStart, timeEnd);

    return 0;
}
```

Run



(*parm) Recall that the *parameter* is part of the function definition, while the *argument* is the item passed in a function call

# 8.2 Pointer basics

A ***pointer*** is a variable that contains a memory address, rather than containing data like most variables introduced earlier. The following program introduces pointers via example:

Figure 8.2.1: Introducing pointers via a simple example.

```
#include <stdio.h>

int main(void) {
   int usrInt = 0;    // User defined int value
   int* myPtr = NULL; // Pointer to the user defined int value

   // Prompt user for input
   printf("Enter any number: ");
   scanf("%d", &usrInt);

   // Output int value and location
   printf("We wrote your number into variable usrInt.₩n");
   printf("The content of usrInt is: %d.₩n", usrInt);
   printf("usrInt's memory address is: %p.₩n", (void*) &usrInt);
   printf("₩nWe can store that address into pointer variable myPtr.₩n");

   // Grab location storing user value
   myPtr = &usrInt;

   // Output pointer value and value pointed by pointer
   printf("The content of myPtr is: %p.₩n", (void*) myPtr);
   printf("The content of what myPtr points to is: %d.₩n", *myPtr);

   return 0;
}
```

```
Enter any number: 555
We wrote your number into variable usrInt.
The content of usrInt is: 555.
usrInt's memory address is: 0x7fff5fbff908.

We can store that address into pointer variable myPtr.
The content of myPtr is: 0x7fff5fbff908.
The content of what myPtr points to is: 555.
```



The example demonstrates key aspects of working with pointers:

- *Appending "*" after a data type* in a variable declaration declares a pointer variable, as in `int* myPtr;`. One might imagine that the programming language would have a type like "address" in addition to types like int, char, etc., but instead the language requires each pointer variable to indicate the type of data to which the address points. So valid pointer variable declarations are `int* myptr1;`, `char* myptr2;`, `double* myptr3`, and even `Seat* myptr4;` (where Seat is a struct type); all such variables will contain memory addresses.
- *Prepending "&" to any variable's name gets the variable's address*."&" is the reference operator that returns a pointer to a variable using the following form:

### Construct 8.2.1: Reference operator.

```
&variableName
```

- *Prepending "*" to a pointer variable's name in an expression gets the data to which the variable points*, as in `*myPtr1`, an act known as ***dereferencing*** a pointer variable."*" is the dereference operator that allows the program to access the value pointed to by the pointer using the form:

### Construct 8.2.2: Dereference operator.

```
*variableName
```



Observe the above program's output. For int variable usrInt, `printf("%p.₩n", (void*) &usrInt);` prints usrInt's memory address. printf() can be used to print the memory address stored within a pointer variable using the format specifier "%p". %p expects the data type void*, but `&usrInt` is the data type int*. So, `&usrInt` is type cast to the data type void*.

Notice that memory address is a large number 0x7fff5fbff908 in contrast to short memory addresses like 96 that have appeared in earlier animations. That large number is in hexadecimal or base 16 number, which you need not concern yourself with as you will not normally print or ever have to look at such memory addresses — the memory address is printed here just for illustration.

The statement `myPtr = &usrInt;` will thus set myPtr's contents to that large address. `printf("%p.₩n", (void*) myPtr);` will print myPtr's contents, which is that large address. `printf("%d.₩n", *myPtr);` will instead go to that address and then print that address' contents.

The "*" (asterisk) symbol is used in two ways related to pointers. One is to indicate that a variable is a pointer type, as in `int* myPtr;`. The other is to dereference a pointer variable, as in `printf("%d.₩n", *myPtr);`. Don't be confused by those two different uses; they have different meanings, both related to pointers.



The pointer was initialized to ***NULL***. In C, NULL is macro defined to represent that the pointer variable points to nothing. On most systems, NULL is defined as the value 0 because 0 is not a valid memory address.

The following animation illustrates pointers.

**PARTICIPATION ACTIVITY** 8.2.1: Simple pointer example.

Start

```
#include <stdio.h>

int main() {
   int usrInt = 0;     // User defined int value
   int* myPtr = NULL; // Pointer to an integer

   printf("Enter any number: ");
   scanf("%d", &usrInt);

   printf("We wrote your number into variable usrInt.\n");
   printf("The content of usrInt is: %d.\n", usrInt);
   printf("usrInt's memory address is: %p.\n", (void*) &usrInt);
   printf("We can store that address into ");
   printf("pointer variable myPtr.\n");

   myPtr = &usrInt;

   printf("The content of myPtr is: %p.\n", (void*) myPtr);
   printf("The content of what myPtr points to ");
   printf("is: %d.\n", *myPtr);


   return 0;
}
```

*Memory address*

| | | |
|---|---|---|
| 75 | | |
| 76 | 555 | usrInt |
| 77 | | |
| 78 | | |
| 79 | 76 | myPtr |

```
Enter any number:   555
We wrote your number into...
The content of userInt is:   555
usrInt's memory address is: 7(
We can store that address intc
      pointer variable myPtr.
The content of myPtr is: 76.
The content of what myPtr poi
      is: 555.
```



The "*" in a pointer variable declaration has some syntactical options. We wrote `int* myPtr;`. However, also allowed is `int *myPtr;`. Many programmers find the former option that groups the "int" and "*" more intuitive, suggesting myPtr is of type "integer pointer". On the other hand, note that `int* myPtr1, myPtr2;` does *not* declare two pointers, but rather declares pointer variable myPtr1 , and int variable myPtr2. For this reason, some programmers prefer the option that groups the "*" with the variable name, as in `int *myPtr1, *myPtr2;`. Our advice: to reduce errors, it may be good practice to only declare one pointer per line, using the "int*" option.

**PARTICIPATION ACTIVITY** 8.2.2: Using pointers.

The following provides an example (not useful other than for learning) of assigning the address of variable vehicleMpg to the pointer variable valPtr.



1. Run and observe that the two output statements produce the same output.
2. Modify the value assigned to *valPtr and run again.
3. Now uncomment the statement that assigns vehicleMpg. PREDICT whether both output statements will print the same output. Then run and observe the output; did you predict correctly?

Load default template... Run

```

#include <stdio.h>

int main(void) {
   double vehicleMpg = 0.0;
   double* valPtr = 0;

   valPtr = &vehicleMpg;

   *valPtr = 29.6; // Assigns the number to the variable
                   // POINTED TO by valPtr.

  // vehicleMpg = 40;   // Uncomment this later

   printf("Vehicle MPG = %lf\n", vehicleMpg);
   printf("Vehicle MPG = %lf\n", *valPtr);
   return 0;
}

```

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

**PARTICIPATION ACTIVITY** 8.2.3: Pointer basics.

Assume variable int `numStudents = 12` is at memory address 99, and `int* myPtr;` is at address 44. Answer "error" where appropriate.

1) What does `printf("%d", numStudents)` output?

Check Show answer

2) What does `printf("%p", (void*) &numStudents)` output?

Check Show answer



3) What does `printf("%d", *numStudents)` output?

Check Show answer

4) After `myPtr = &numStudents`, what does `printf("%d", *myPtr)` output?

Check Show answer

**CHALLENGE ACTIVITY** 8.2.1: Printing with pointers.

Assign numItems' address to numItemsPtr, then print the shown text followed by the value to which numItemsPtr points. End with newline.

```
Items: 99
```

```
#include <stdio.h>

int main(void) {
   int* numItemsPtr = 0;
   int numItems = 99;

   /* Your solution goes here  */

   return 0;
}
```

Run



# 8.3 The malloc and free functions

Sometimes memory should be allocated while a program is running and should persist independently of any particular function. The malloc() function carries out such memory allocation.

***malloc()*** is a function defined within the ***standard library***, which can be used by adding `#include <stdlib.h>` to the beginning of a program. The malloc() function is named for *m*emory *alloc*ation. malloc() allocates memory of a given number of bytes and returns a pointer (i.e., the address) to that allocated memory. A basic call to malloc() has the form:

Construct 8.3.1: Malloc function.

```
malloc(bytes)
```

malloc() takes a single argument specifying the number of bytes to allocate in memory. Thus, the programmer must determine the number of bytes needed to allocate space for the desired data type. But, as you may recall, the number of bytes for various data types (e.g., int) may vary across different computer systems. Fortunately, C provides a sizeof() function that returns the number of bytes for a given data type. For example, on a system where an int is 32 bits, `sizeof(int)` returns 4, because 32-bits is 4 bytes. Calls to malloc() are typically combined with sizeof() using the form:

Construct 8.3.2: Malloc and sizeof functions.

```
malloc(sizeof(type))
```



malloc() returns a pointer to allocated memory using a `void*`, referred to as a ***void pointer***. A void pointer is a special type of pointer that stores a memory address without referring to the type of variable stored at that memory location. The void pointer return type allows a single malloc() function to allocate memory for any data type. The pointer returned from malloc() can then be written into a particular pointer variable by casting the void pointer to the data type using the form:

Construct 8.3.3: Malloc return type.

```
pointerVariableName = (type*)malloc(sizeof(type));
```

The following animation illustrates using malloc() to allocate memory for an int. int is used for introduction; malloc() is more commonly used to allocate memory for struct types, arrays, and strings.

**PARTICIPATION ACTIVITY** 8.3.1: The malloc() operation.



Start

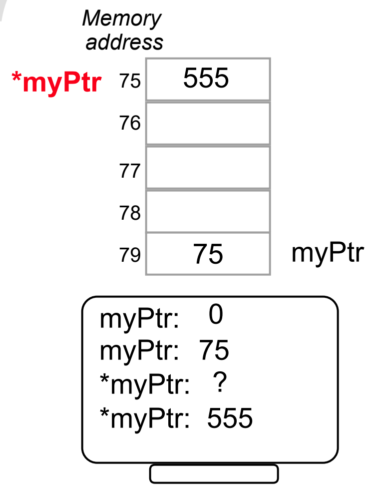
```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int* myPtr = NULL;
   printf("myPtr: %p\n", (void*) myPtr);
   // Next line would cause error because myPtr is null
   // printf("*myPtr: %d\n", *myPtr); // ERROR

   // sizeof() returns number of bytes for type
   // malloc() allocates specified number of bytes
   // cast void pointer to the desired pointer type
   myPtr = (int*) malloc(sizeof(int));
   printf("myPtr: %p\n", (void*) myPtr);
   printf("*myPtr: %d\n", *myPtr);

   *myPtr = 555;
   printf("*myPtr: %d\n", *myPtr);

   return 0;
}
```




The malloc() function returns NULL if the function failed to allocate memory. Such failure could happen if a program has used up all memory available to the program.

***free()*** is a function that deallocates a memory block pointed to by a given pointer, which must have been previously allocated by malloc(). In other words, free() does the opposite of malloc(). Deallocating memory using free() has the following form:

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

Construct 8.3.4: The free function.

```
free(pointerVariable);
```

After `free(pointerVariable);`, the program should not attempt to dereference pointerVariable, as pointerVariable points to a memory location that is no longer allocated for use by pointerVariable. Dereferencing a pointer whose memory has been deallocated is a common error, and may cause strange program behavior that is difficult to debug — if that memory had since been allocated to another variable, that variable's value could mysteriously change. Calling free with a pointer that wasn't previously allocated by malloc is also an error.

**PARTICIPATION ACTIVITY** 8.3.2: malloc and free.

1) Declare a variable named myValPointer as a pointer of type int, initializing the pointer to NULL in the declaration.

Check Show answer



2) Write a statement that allocates memory for a new double value using the pointer variable newInputPtr.

Check Show answer

3) Write a statement that deallocates memory for the pointer variable newInputPtr.

Check Show answer

Exploring further:

- malloc Reference Page from cplusplus.com
- More on malloc from msdn.microsoft.com
- free Reference Page from cplusplus.com
- More on free from msdn.microsoft.com



**CHALLENGE ACTIVITY** 8.3.1: Using malloc and pointers.

Write two statements that each use malloc to allocate an int location for each pointer. Sample output for given program:

```
numPtr1 = 44, numPtr2 = 99
```



```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int* numPtr1 = NULL;
   int* numPtr2 = NULL;

   /* Your solution goes here  */

   *numPtr1 = 44;
   *numPtr2 = 99;

   printf("numPtr1 = %d, numPtr2 = %d\n", *numPtr1, *numPtr2);

   free(numPtr1);
   free(numPtr2);

   return 0;
}
```

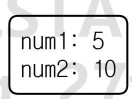
Run

# 8.4 Pointers with structs

The malloc() function is commonly used with struct types to allocate the appropriate block of memory for a variable of a struct type.

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

Figure 8.4.1: Using malloc() with a struct type.



```
#include <stdio.h>
#include <stdlib.h>

typedef struct myItem_struct {
   int num1;
   int num2;
} myItem;

void myItem_PrintNums(myItem* itemPtr) {
   if (itemPtr == NULL) return;

   printf("num1: %d\n", itemPtr->num1);
   printf("num2: %d\n", itemPtr->num2);

   return;
}

int main(void) {
   myItem* myItemPtr1 = NULL;

   myItemPtr1 = (myItem*)malloc(sizeof(myItem));

   myItemPtr1->num1 = 5;
   (*myItemPtr1).num2 = 10;

   myItem_PrintNums(myItemPtr1);

   return 0;
}
```



Accessing a struct's member variables by first dereferencing a pointer, as in `(*myItemPtr1).num2 = 5;`, is so common that the language includes a second ***member access operator*** with the form:

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

## Construct 8.4.1: Member access operator.

```
structPtr->memberName     // Equivalent to (*structPtr).memberName
```

The above program illustrates use of the member access operator: `myItemPtr1->num1 = 5;`.

**PARTICIPATION ACTIVITY** 8.4.1: The malloc, free, and -> operators.

Assuming the following is defined:

```
typedef struct Fruit_struct {
   // member variables
} Fruit;
```

1) Declare a variable named orange as a pointer of type Fruit.

Check Show answer

2) Write a statement that allocates memory for the new variable orange that points to class Fruit.

Check Show answer

3) For the variable orange, write a statement that assigns a member variable named hasSeeds to 1. Use the -> operator.

Check Show answer

4) Write a statement that deallocates memory pointed to by variable orange, which is a pointer of type Fruit.

Check Show answer

5) Assuming a struct Fruit has two int data members and an int is 32 bits, what number does sizeof(Fruit) return?

Check Show answer

**CHALLENGE ACTIVITY** 8.4.1: Struct pointers.

Write two statements to assign numApples with 10 and numOranges with 3. Sample output for given program:

```
Apples: 10
Oranges: 3
```



```c
#include <stdio.h>
#include <stdlib.h>

typedef struct bagContents_struct {
   int numApples;
   int numOranges;
} bagContents;

void bagContents_PrintBag(bagContents* itemPtr) {
   if (itemPtr == NULL) return;

   printf("Apples: %d\n", itemPtr->numApples);
   printf("Oranges: %d\n", itemPtr->numOranges);

   return;
}

int main(void) {
   bagContents* groceryPtr = NULL;

   groceryPtr = (bagContents*)malloc(sizeof(bagContents));
```

Run

## 8.5 String functions with pointers



The C string library, introduced elsewhere, contains several functions for working with C strings. This section describes the use of char pointers in such functions. Recall that the C string library must first be included via: `#include <string.h>`

String functions accept a char pointer for a string argument. That pointer is commonly a char array variable, or a string literal (each of which is essentially a pointer to the 0th element of a char array), but could also be an explicit char pointer. Example of such functions are strcmp(), strcpy(), and strchr(), introduced elsewhere.

Figure 8.5.1: String functions accept char pointers as arguments.

```
char string1[10] = "abcxyz";
char string2[10] = "xyz";
char newText[10] = "";
char* subStr = NULL;

if (strcmp(string1, string2) == 0) {      // abcxyz does not equal xyz
   ...
if (strcmp(&string1[3], "xyz") == 0) {   // xyz equals xyz
   ...
subStr = &string1[3];                     // Points to 'x' in string1
if (strcmp(subStr, string2) == 0) {       // xyz equals xyz
   ...
strcpy(newText, subStr);                  // newText is now "xyz"
```



Table 8.5.1: Some C string modification functions.

Given:

```
char orgName[100] = "The Dept. of Redundancy Dept.";
char newText[100] = "";
char* subString = NULL;
```

| | | |
|---|---|---|
| ***strchr()*** | `strchr(sourceStr, searchChar)`<br><br>Returns NULL if searchChar does not exist in sourceStr. Else, returns pointer to first occurrence. | `if (strchr(orgName, 'D') != NULL) { // 'D' exists in orgName?`<br>`   subString = strchr(orgName, 'D');     // Points to first 'D'`<br>`   strcpy(newText, subString);      // newText now "Dept. of Redundancy Dept."`<br>`}`<br>`if (strchr(orgName, 'Z') != NULL) { // 'Z' exists in orgName?`<br>`   ... // Doesn't exist, branch not taken`<br>`}` |
| ***strrchr()*** | `strrchr(sourceStr, searchChar)`<br><br>Returns NULL if searchChar does not exist in sourceStr. Else, returns pointer to LAST occurrence (searches in reverse, hence middle 'r' in name). | `if (strrchr(orgName, 'D') != NULL) { // 'D' exists in orgName?`<br>`   subString = strrchr(orgName);     // Points to last 'D'`<br>`   strcpy(newText, subString);       // newText now "Dept."`<br>`}` |
| ***strstr()*** | `strstr(str1, str2)`<br><br>Returns char* pointing to first occurrence of string str2 within string str1. Returns NULL if not found. | `subString = strstr(orgName, "Dept"); // Points to first 'D'`<br>`if (subString != NULL) {`<br>`   strcpy(newText, subString);       // newText now "Dept. of Redundancy Dept."`<br>`}` |

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

The following example carries out a simple censoring program, replacing an exclamation point by a period and "Boo" by "---" (assuming those items are somehow bad and should be censored):

Figure 8.5.2: String searching example.

```
#include <stdio.h>
#include <string.h>

int main(void) {
   const int MAX_USER_INPUT = 100;  // Max input size
   char userInput[MAX_USER_INPUT];  // User defined string
   char* stringPos = NULL;          // Index into string

   // Prompt user for input
   printf("Enter a line of text: ");
   fgets(userInput, MAX_USER_INPUT, stdin);

   // Locate exclamation point, replace with period
   stringPos = strchr(userInput, '!');

   if (stringPos != NULL) {
      *stringPos = '.';
   }

   // Locate "Boo" replace with "---"
   stringPos = strstr(userInput, "Boo");

   if (stringPos != NULL) {
      strncpy(stringPos, "---", 3);
   }

   // Output modified string
   printf("Censored: %s\n", userInput);

   return 0;
}
```

```
Enter a line of text: Hello!
Censored: Hello.

...

Enter a line of text: Boo hoo to you!
Censored: --- hoo to you.

...

Enter a line of text: Booo! Boooo!!!!
Censored: ---o. Boooo!!!!
```

Note above that only the first occurrence of "Boo" is replaced, as strstr() returns a pointer just to the first occurrence. (Additional code would be needed to delete all occurrences).



**PARTICIPATION ACTIVITY** 8.5.1: Modifying and searching strings.

1) Declare a char* variable named charPtr.

Check Show answer

2) Assuming `char* firstR;` is already declared, store in firstR a pointer to the first instance of an 'r' in the char* variable userInput.



Check Show answer

3) Assuming `char* lastR;` is already declared, store in lastR a pointer to the last instance of an 'r' in the char* variable userInput.

Check Show answer

4) Assuming `char* firstQuit;` is already declared, store in firstQuit a pointer to the first instance of "quit" in the char* variable userInput.

Check Show answer

**CHALLENGE ACTIVITY** 8.5.1: Find char in string.



Assign searchResult with a pointer to any instance of searchChar in personName. Hint: Use strchr().

```
#include <stdio.h>
#include <string.h>

int main(void) {
   char personName[100] = "Albert Johnson";
   char searchChar = 'J';
   char* searchResult = NULL;

```

```
   /* Your solution goes here  */

   if (searchResult != NULL) {
      printf("Character found.\n");
   }
   else {
      printf("Character not found.\n");
   }

   return 0;
}
```



Run

**CHALLENGE ACTIVITY** 8.5.2: Find string in string.

Assign movieResult with the first instance of The in movieTitle.

```
#include <stdio.h>
#include <string.h>

int main(void) {
   char movieTitle[100] = "The Lion King";
   char* movieResult = NULL;

   /* Your solution goes here  */

   printf("Movie title contains The? ");
   if (movieResult != NULL) {
      printf("Yes.\n");
   }
   else {
      printf("No.\n");
   }

   return 0;
}
```

Run



# 8.6 The malloc function for arrays and strings

Pointers are commonly used to allocate arrays just large enough to store a required number of data elements. Previously, the programmer had to declare arrays to have a fixed size, referred to as a ***statically allocated*** array. Statically allocated arrays may not use all the allocated memory due to the programmer not knowing the actual needed size before the program runs, thus creating larger-than-necessary arrays. Even then, the program might need a larger size.

Instead of statically allocating an array, malloc() can be used to allocate just enough memory for the array. Recall that arrays are stored in sequential memory locations. malloc() can be used to allocate a ***dynamically allocated*** array by determining the total number of bytes needed to store the desired number of elements, using the following form:



Construct 8.6.1: Dynamically allocated array.

```
pointerVariableName = (dataType*)malloc(numElements * sizeof(dataType))
```

When allocating an array, malloc() returns a pointer to memory location of the first element within the array. This memory location can be stored within a pointer variable declared as a pointer to the type of element within the array. For example, when dynamically allocating an array of integers, a pointer variable of integers "int*" is used. Notice then that an int* pointer can point to either a single integer or to an array of multiple characters. A programmer must carefully keep track of how each pointer variable is utilized within the program.

The following program illustrates how to dynamically allocate an arrays of integers.

Figure 8.6.1: Dynamically allocating an array of integers.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int* userVals = NULL; // No array yet
   int numVals = 0;
   int i = 0;

   printf("Enter number of integer values: ");
   scanf("%d", &numVals);

   userVals = (int*)malloc(numVals * sizeof(int));

   printf("Enter %d integer values...₩n", numVals);
   for (i = 0; i < numVals; ++i) {
      printf("Value: ");
      scanf("%d", &(userVals[i]));
   }

   printf("You entered: ");
   for (i = 0; i < numVals; ++i) {
      printf("%d ", userVals[i]);
   }
   printf("₩n");

   free(userVals);

   return 0;
}
```

```
Enter number of integer values: 8
Enter 8 integer values...
Value: 4
Value: 23
Value: 14
Value: 5
Value: 4
Value: 3
Value: 7
Value: 9
You entered: 4 23 14 5 4 3 7 9
```

**PARTICIPATION ACTIVITY** 8.6.1: Using malloc for arrays.

1) Write a malloc function call to allocate an array for 10 int variables.

```
int* itemList = NULL;
itemList = (int*)malloc(10 *
sizeof(          ));
```

Check Show answer



2) Write a malloc function call to allocate an array for 15 double variables.

```
double* priceList = NULL;
priceList = (          
)malloc(15 * sizeof(double));
```

Check Show answer

3) Using malloc, write a statement that allocates an array of 10 chars.

```
char* myStr = NULL;
myStr = (char*)malloc(
          );
```

Check Show answer

The following program creates a string for a simple greeting given a user entered name.

Figure 8.6.2: Concatenating strings using a statically allocated array.

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

```
Enter name: Bill
Hello Bill.
```

```
#include <stdio.h>
#include <string.h>

int main(void) {
   char nameArr[100] = "";       // User specified name
   char greetingArr[100] = "";   // Output greeting and name

   // Prompt user to enter a name
   printf("Enter name: ");
   fgets(nameArr, 100, stdin);

   // Eliminate end-of-line char
   nameArr[strlen(nameArr)-1] = '\0';

   // Modify string, hello + user specified name
   strcpy(greetingArr, "Hello ");
   strcat(greetingArr, nameArr);
   strcat(greetingArr, ".");

   // Output greeting and name
   printf("%s\n", greetingArr);

   return 0;
}
```



The above program declares two statically allocated arrays named nameArr and greetingArr. Each array can store a string with 0 to 99 characters — keeping in mind the space needed to store the null character at the end of the string. However, if the user enters the name "Bob", the resulting string stored within the greeting array only requires 11 characters — 6 characters for "Hello ", 3 characters for the name "Bob", 1 character for the ".", and 1 character for the null character. Thus, 88 characters are unused in the greeting array. Likewise, the program fails if the user enters a very long name.

The following program revises the earlier example by dynamically allocating a greeting array to be just large enough to store the entire greeting.

Figure 8.6.3: Concatenating strings using a dynamically allocated array.

```
Enter name: Julia
Hello Julia.

...

Enter name: John Smith
Hello John Smith.
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
   char nameArr[100] = "";   // User specified name
   char* greetingPtr = NULL; // Pointer to output greeting and name

   // Prompt user to enter a name
```

The nameArr array is used for reading the user input, so nameArr is statically allocated with a fixed size, defining a limit to the length of the name that the program can support.



In contrast, the size of the greeting is determined at runtime based upon the actual user-entered name length. `char* greetingPtr;` declares a char pointer variable named greetingPtr. Once the user has entered a name and the newline character at the end of the name string has been removed, `strlen(nameArr)` determines the number of characters within that name. In addition to the length of the string, 8 characters are needed for the greeting — 6 for the string "Hello ", 1 for the ".", and 1 for the end-of-string null character. Thus, the total number of bytes required for the greeting array can be determined using the expression `strlen(nameArr) + 8) * sizeof(char)`. For example, if the user enters the name "Julia", the total number of characters allocated for the greeting array will be 5 + 8 = 13.

The program can then create the greeting array by first copying the string "Hello " to the greeting array using the statement `strcpy(greetingPtr, "Hello ");`. The statement `strcat(greetingPtr, nameArr);` then appends the user-entered name to the end of the string stored within greetingPtr. Finally, `strcat(greetingPtr, ".");` appends a "." to the end of the string.

To create a dynamically allocated copy of a string, malloc() can be used to create a character array with a size equal to the number of characters within the source string plus one character for the null character. The following program illustrates. As the length returned by `strlen(nameArr)` does not include the null character required at the end of a string, `strlen(nameArr) + 1` is used to determine the number of characters required for the dynamically allocated string. A common error is to allocate only strlen chars for a copied string, forgetting the + 1 needed for the null character.

Figure 8.6.4: Creating a dynamically allocated copy of a string.

```
Enter name: Julia
Hello Julia

...

Enter name: John Smith
Hello John Smith
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
   char nameArr[50] = ""; // User specified name
   char* nameCopy = NULL; // Output greeting and name
```

**PARTICIPATION ACTIVITY** 8.6.2: Creating and modifying strings.



1) Given an existing string sentStart, complete the following statement to allocate a new string songVerse that is large enough to store the string sentStart plus seven additional characters.

```
songVerse = (char*)malloc((
                    ) *
sizeof(char));
```

Check Show answer

2) In a single statement, copy the string pointed to by the char* sentStart to the string pointed to by the char* songVerse.

Check Show answer

**CHALLENGE ACTIVITY** 8.6.1: Pointers for allocating a C string.



Use strlen(userStr) to allocate exactly enough memory for newStr to hold the string in userStr (Hint: do NOT just allocate a size of 100 chars).

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


int main(void) {
   char userStr[100] = "";
   char* newStr = NULL;

   strcpy(userStr, "Hello friend!");

   /* Your solution goes here  */
```

```
    strcpy(newStr, userStr);
    printf("%s\n", newStr);

    return 0;
}
```

Run



# 8.7 The realloc function

The ***realloc*** function re-allocates an original pointer's memory block to be the newly-specified size. realloc() can be used to both increase or decrease the size of dynamically allocated arrays. realloc() returns a pointer to the re-allocated memory. The pointer returned by realloc() may or may not differ from the original pointer. For example, if realloc() is used to increase a memory block but the function doesn't find enough available memory at the existing block's end, the function finds a large enough block of memory at another location in memory, and copies the existing block's contents to the new block.

In its most common form, the pointer returned from realloc() will be assigned to the same pointer provided as the input argument, using the form:

Construct 8.7.1: The realloc function.

```
pointerVariable = (type*)realloc(pointerVariable, numElements * sizeof(type))
```

The following program computes the average of several user-entered values stored within a dynamically allocated array. The array is reallocated each time the user specifies the number of integers values.

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

Figure 8.7.1: Dynamically reallocating the size of an array.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int* userVals = NULL; // No array yet
   int numVals = 0;
   int i = 0;
   char userInput = 'c';
   int userValsSum = 0;
   double userValsAvg = 0.0;

   while (userInput == 'c') {

      printf("Enter number of integer values: ");
      scanf(" %d", &numVals);

      if (userVals == NULL) {
         userVals = (int*)malloc(numVals * sizeof(int));
      }
      else {
         userVals = (int*)realloc(userVals, numVals * sizeof(int));
      }

      printf("Enter %d integer values...₩n", numVals);
      for (i = 0; i < numVals; ++i) {
         printf("Value: ");
         scanf("%d", &(userVals[i]));
      }

      // Calculate average
      userValsSum = 0;
      for (i = 0; i < numVals; ++i) {
         userValsSum = userValsSum + userVals[i];
      }
      userValsAvg = (double)userValsSum / (double)numVals;

      printf("Average = %lf₩n", userValsAvg);

      printf("₩nEnter 'c' to compute another average (any other key to quit): ");
      scanf(" %c", &userInput);
   }

   free(userVals);

   return 0;
}
```



```
Enter number of integer values: 3
Enter 3 integer values...
Value: 13
Value: 11
Value: 17
Average = 13.666667

Enter 'c' to compute another average (any other key to quit): c
Enter number of integer values: 7
Enter 7 integer values...
Value: 10
Value: 14
Value: 56
Value: 23
Value: 18
Value: 3
Value: 6
Average = 18.571429

Enter 'c' to compute another average (any other key to quit): q
```

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

PARTICIPATION ACTIVITY 8.7.1: realloc.

1) Given an int* pointer dynInputVals that points to an existing dynamically allocated array of integers, complete the following statement to reallocate the array to have 14 elements.

```
dynInputVals = (int*)realloc(
[          ], 14 *
sizeof(int));
```

Check Show answer



2) Given a double* pointer sensorVals that points to an existing dynamically allocated array of 200 elements, complete the following statement to reallocate the size of the array to have only 2 elements.

```
sensorVals =
(double*)realloc(sensorVals,
                         );
```

Check Show answer



# 8.8 Vector ADT

structs and pointers can be used to implement a computing concept known as an ***abstract data type (ADT)***, which is a data type whose creation and update are supported by specific well-defined operations. A key aspect of an ADT is that the internal implementation of the data and operations are hidden from the ADT user, a concept known as ***information hiding***, thus allowing the ADT user to be more productive by focusing on higher-level concepts, and also allowing the ADT developer to improve the internal implementation without requiring changes to programs using the ADT.

Programmers commonly refer to separating an ADT's *interface* from its *implementation*; the user of an ADT need only know the ADT's interface (functions declared within the ADT's header file) and not its implementation (function definitions and struct data members).

**PARTICIPATION ACTIVITY** 8.8.1: Abstract data types (ADT).

1) ADTs (Abstract data types) are meant to the hide values of variables from the user.

- True
- False

2) An ADT's interface is commonly separated from its implementation.

- True
- False



3) An ADT is a fixed data type, like an int or char.

- True
- False

A vector is an example of ADT that stores an ordered list of items (called elements) of a given data type, such as a vector of integers. Like an array, an individual element within the vector can be accessed using an index. However, unlike an array, a vector's size can be adjusted while a program is executing, an especially useful feature when the number of items that will be in the list is unknown at compile-time. To support such size adjustment, a vector ADT provides functions for common operations like creating the vector, inserting elements into the vector, removing elements from the vector, resizing the vector, and accessing elements at specific locations.

The following defines an ADT for a vector of integers — defining a new struct type named "vector" and the function prototypes for the vector's interface. These declarations are included in the header file for the ADT named "vector.h". Note_vector_ADT



Figure 8.8.1: struct and function prototypes for vector ADT.

```
#ifndef VECTOR_H
#define VECTOR_H

// struct and typedef declaration for Vector ADT
typedef struct vector_struct {
   int* elements;
   unsigned int size;
} vector;

// interface for accessing Vector ADT

// Initialize vector
void vector_create(vector* v, unsigned int vectorSize);

// Destroy vector
void vector_destroy(vector* v);

// Resize the size of the vector
void vector_resize(vector* v, unsigned int vectorSize);

// Return pointer to element at specified index
int* vector_at(vector* v, unsigned int index);

// Insert new value at specified index
void vector_insert(vector* v, unsigned int index, int value);

// Insert new value at end of vector
void vector_push_back(vector* v, int value);

// Erase (remove) value at specified index
void vector_erase(vector* v, unsigned int index);

// Return number of elements within vector
int vector_size(vector* v);

#endif
```

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

To use the vector, a program must first include the following: `#include "vector.h"`. Then, a variable for a vector can be declared and used as shown in the below animation. The definition creates a vector variable named v. Data members within the vector struct are not initialized automatically. The vector_create() function is used to initialize the vector given the specified size of the vector. The function call `vector_create(&v, 3)` initializes the vector by allocating memory for three elements, each of type int, and initializes each of those elements to the value 0.

The function call `vector_at(&v, 0)` returns a pointer to the int element stored at location 0 of the the vector v. A value can be written to this location by dereferencing the returned int pointer. For example, the statement `*vector_at(&v, 0) = 119;` assigns a value of 119 to the int element at location 0 of the vector c.

Notice that the first parameter for each of the vector's functions is a pointer to a vector ("vector*"). When calling each of these functions, this parameter will point to the specific vector on which the corresponding operation will be performed. Thus, a program can consist of multiple vectors, using the same set of the vector functions.

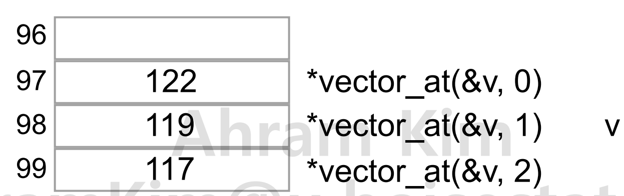
**PARTICIPATION ACTIVITY** 8.8.2: A vector declaration creates multiple variables in memory, each accessible using vector_at();.

Start

```
vector v;
vector_create(&v, 3);

*vector_at(&v, 0) = 122;
*vector_at(&v, 1) = 119;
*vector_at(&v, 2) = 117;

printf("%d", *vector_at(&v, 1));
```




```
119
```

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

The following summarizes the functions for the vector ADT defined above:

Table 8.8.1: Vector ADT functions.

| Function | Description | Example |
|---|---|---|
| /ector_create(vector* v, unsigned int vectorSize) | Initializes the vector pointed to by v with vectorSize number of elements. Each element with the vector is initialized to 0. | vector_create(&v, 2( |
| /ector_destroy(vector* v) | Destroys the vector by freeing all memory allocated within the vector. | vector_destroy(&v) |
| /ector_resize(vector* v, unsigned int vectorSize) | Resizes the vector with vectorSize number of elements. If the vector size increased, each new element within the vector is initialized to 0. | vector_resize(&v, 2! |
| /ector_at(vector* v, unsigned int index) | Returns a pointer to | x = vector_at(&v, 1) |


Ahram Kim AhramKim@u.boisestate.edu BOISESTATECS253Fall2017 Aug. 27th, 2017 18:15

| | | |
|---|---|---|
| | the element at the location index. | |
| ned int vector_size(vector* v) | Returns the vector's size — i.e. the number of elements within the vector. | if (vector_size(&v) |
| vector_push_back(vector* v, int value) | Inserts the value x to a new element at the end of the vector, increasing the vector size by 1. | vector_push_back(&v, adds "47" onto the end the vector. |
| vector_insert(vector* v, unsigned int index, int value) | Inserts the value x to the element indicated by position, making room by shifting over the elements at that position and higher, thus increasing | vector_insert(&v, 2, inserts "33" at vector_a 2). If the vector conten were (18, 26, 47, 52) be the insert, the contents the insert would be (18 33, 47, 52). (Recall tha vector indices start at 1, so position 2 is actu the 3rd position). |



Ahram Kim AhramKim@u.boisestate.edu BOISESTATECS253Fall2017 Aug. 27th, 2017 18:15

| | | |
|---|---|---|
| | the vector size by 1. | |
| vector_erase(vector* v, unsigned int index) | Removes the element from the indicated position. All elements at higher positions are shifted over to fill the gap left by the removed element. Thus, the vector size decreases by 1. | vector_erase(&v, 0) would erase element vector_at(&v, 0). If t vector contents were ( 26, 47, 52) before the e the contents after the would be (26, 47, 52). |



The definition of the vector's functions are implemented within a file named "vector.c" shown below. Notice from the vector's struct definition that the vector has a data member for a *dynamically allocated array* of integers and a data member for the size of the array. For ADTs, a programmer should not directly access the data members of the struct in order to adhere to the concept to information hiding.

Figure 8.8.2: Function definitions for vector ADT.

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

```
#include <stdio.h>
#include <stdlib.h>
#include "vector.h"

// Initialize vector with specified size
void vector_create(vector* v, unsigned int vectorSize) {
   int i = 0;

   if (v == NULL) return;

   v->elements = (int*)malloc(vectorSize * sizeof(int));;
   v->size = vectorSize;
   for (i = 0; i < v->size; ++i) {
      v->elements[i] = 0;
   }
}
```

```
// Destroy vector
void vector_destroy(vector* v) {
   if (v == NULL) return;

   free(v->elements);
   v->elements = NULL;
   v->size = 0;
}

// Resize the size of the vector
void vector_resize(vector* v, unsigned int vectorSize) {
   int oldSize = 0;
   int i = 0;
   if (v == NULL) return;

   oldSize = v->size;
   v->elements = (int*)realloc(v->elements, vectorSize * sizeof(int));
   v->size = vectorSize;
   for (i = oldSize; i < v->size; ++i) {
      v->elements[i] = 0;
   }
}

// Return pointer to element at specified index
int* vector_at(vector* v, unsigned int index) {
   if (v == NULL || index >= v->size) return NULL;

   return &(v->elements[index]);
}

// Insert new value at specified index
void vector_insert(vector* v, unsigned int index, int value) {
   int i = 0;

   if (v == NULL || index > v->size) return;

   vector_resize(v, v->size + 1);
   for (i = v->size - 1; i > index; --i) {
      v->elements[i] = v->elements[i-1];
   }
   v->elements[index] = value;
}

// Insert new value at end of vector
void vector_push_back(vector* v, int value) {
   vector_insert(v, v->size, value);
}

// Erase (remove) value at specified index
void vector_erase(vector* v, unsigned int index) {
   int i = 0;

   if (v == NULL || index >= v->size) return;

   for (i = index; i < v->size - 1; ++i) {
      v->elements[i] = v->elements[i+1];
   }
   vector_resize(v, v->size - 1);
}

// Return number of elements within vector
int vector_size(vector* v) {
   if (v == NULL) return -1;

   return v->size;
}
```

A common error when using ADT such as a vector is passing an incorrect pointer to the vector's functions. For example, if a NULL pointer is passed to the vector_create() function, the expression `v->elements` would attempt to dereference a NULL pointer. As a NULL ptr refers to an invalid memory address, this type of error will cause a program to terminate in an error referred to as a ***segmentation fault***, ***access violation***, or ***bad access***. The actual error name depends on the computer system you are using.

To avoid this common error, each of the functions for the vector ADT first checks if the vector pointer is NULL before trying to dereference that pointer. If the pointer is NULL, the function will either return immediately or return a value indicating an error condition. For example, the function call `vector_size(NULL)` returns the value -1, indicating an error, as a vector cannot have a negative number of elements.

The following animation illustrates `vector_insert()` and `vector_erase()`.



**PARTICIPATION ACTIVITY** 8.8.3: The vector_insert() and vector_erase() functions.

Start

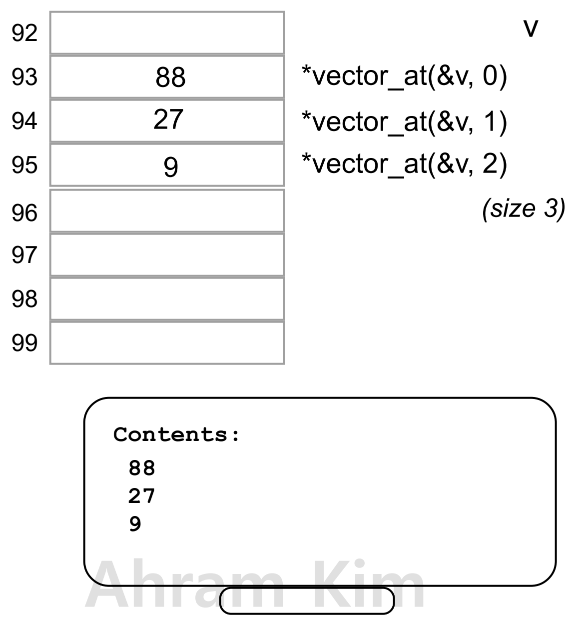
```c
#include <stdio.h>
#include "vector.h"

int main(void) {
   int i = 0;
   vector v;
   vector_create(&v, 4);

   *vector_at(&v, 0) = 27;
   *vector_at(&v, 1) = 44;
   *vector_at(&v, 2) = 9;
   *vector_at(&v, 3) = 17;
   vector_erase(&v, 1);
   vector_insert(&v, 0, 88);
   vector_erase(&v, 3);

   printf("Contents:\n");
   for (i = 0; i < vector_size(&v); ++i) {
      printf("%d\n", *vector_at(&v, i));
   }

   vector_destroy(&v);
   return 0;
}
```






The following modifies an earlier number smoothing program to use the vector ADT rather than directly using an array. The benefit is that the program can support an arbitrary number of user-entered integer numbers.

Figure 8.8.3: Number smoothing program using vector.

```
#include <stdio.h>
#include <stdlib.h>
#include "vector.h"

// Get number from user
void GetNums(vector* nums) {
   int numsSize = 0; // Vector size
   int i = 0;        // Loop index

   printf("Enter number of integers to be entered: ");
   scanf("%d", &numsSize);

   vector_resize(nums, numsSize);

   for (i = 0; i < vector_size(nums); ++i) {
      printf("%d: ", i + 1);
      scanf("%d", vector_at(nums, i));
   }
}

// Smooths by setting element to average of itself and next two
elements
void FilterNums(vector* nums) {
   int i = 0;

   for (i = 0; i < vector_size(nums) - 2; ++i) {
      *vector_at(nums, i) = (*vector_at(nums, i) +
                             *vector_at(nums, i + 1) +
                             *vector_at(nums, i + 2)) / 3;
   }

   *vector_at(nums, i) = (*vector_at(nums, i) +
                          *vector_at(nums, i + 1)) / 2;

   // Last element needs no averaging
}

// Print all elements within the vector
void PrintsNums(vector* nums) {
   int i = 0;

   printf("Numbers: ");
   for (i = 0; i < vector_size(nums); ++i) {
      printf("%d ", *vector_at(nums, i));
   }
   printf("\n");
}

int main(void) {
   vector nums;

   vector_create(&nums, 0);

   GetNums(&nums);
   PrintsNums(&nums);
   FilterNums(&nums);
   PrintsNums(&nums);

   vector_destroy(&nums);

   return 0;
}
```

```
Enter number of integers to be entered:
10
1: 10
2: 20
3: 30
4: 40
5: 50
6: 60
7: 70
8: 80
9: 90
10: 100
Numbers: 10 20 30 40 50 60 70 80 90 100
Numbers: 20 30 40 50 60 70 80 90 95 100
```

PARTICIPATION ACTIVITY 8.8.4: Vector declaration and use.

Write a single statement for each answer.

1) Declare a vector named vals.

Check Show answer

2) Initialize a vector vals to size 10 with all elements set to 0.



Check Show answer

3) Assign the value of the element held at index 8 of vector vals to an int variable

x.

Check Show answer

4) Write 555 into element at index 2 of vector vals.



Check Show answer

5) Store 777 into the second element of vector vals.

Check Show answer

6) Append the value 37 to the vector vals.

Check Show answer

7) Set the int variable sz to the size of the vector vals.

Check Show answer



8) Erase element 0 of vector vals.

Check Show answer

**CHALLENGE ACTIVITY** 8.8.1: Operator overloading.

Modify the existing vector's contents, by erasing 200, then inserting 100 and 102 in the shown locations. Use Vector ADT's erase() and insert() only. Sample output of below program:

```
100 101 102 103
```



```c
#include <stdio.h>
#include <stdlib.h>

// struct and typedef declaration for Vector ADT
typedef struct vector_struct {
  int* elements;
  unsigned int size;
} vector;

// Initialize vector with specified size
void vector_create(vector* v, unsigned int vectorSize) {
  int i = 0;

  if (v == NULL) return;

  v->elements = (int*)malloc(vectorSize * sizeof(int));
  v->size = vectorSize;
  for (i = 0; i < v->size; ++i) {
     v->elements[i] = 0;
  }
}
```

Run

(*Note_vector_ADT) Note to instructors: For the vector ADT we use different convention for naming the functions. All functions supporting the vector ADT begin with "vector_", which is intended to indicate the functions are associated with the "vector" type. The following portion of the function corresponds to operation being performed on the vector ADT. The names for these operations for the vector ADT closely match the names of corresponding operations on the vector class in C++.



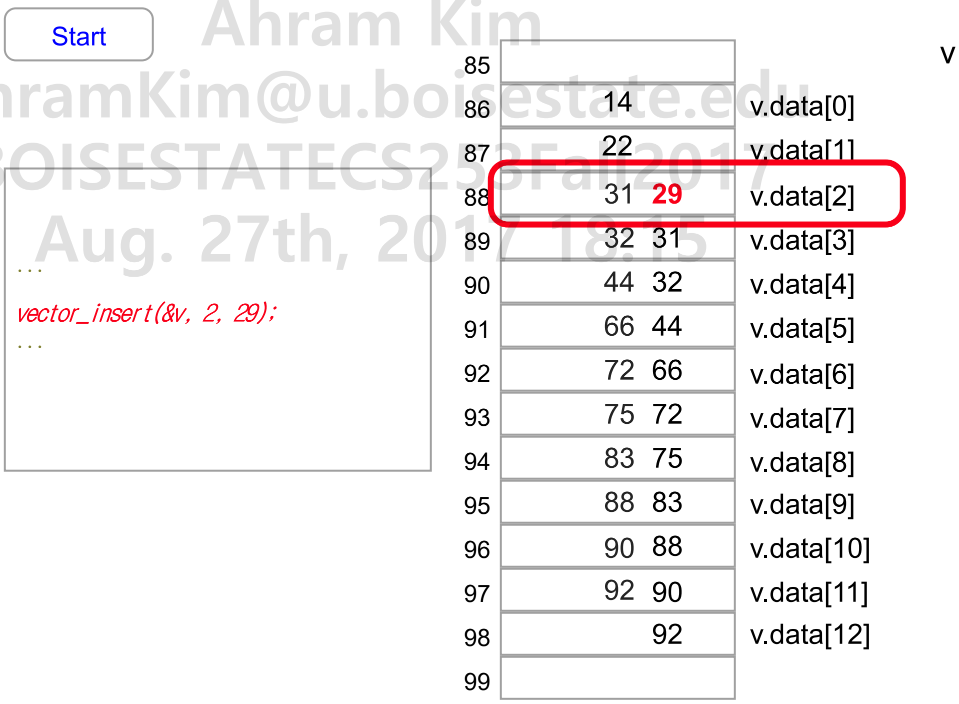
# 8.9 Why pointers: A list example

To further elaborate on the need for pointers, this section describes another of many situations where pointers are useful.

The vector ADT (or arrays) stores a list of items in contiguous memory locations, which enables immediate access to any element i of vector v by using `vector_at(&v, i)`. Recall that inserting an item within a vector requires making room by shifting higher-indexed items. Similarly, erasing an item requires shifting higher-indexed items to fill the gap. Each shift of an item from one element to another element requires a few processor instructions. This issue exposes the ***vector insert/erase performance***

***problem***. For vectors with thousands of elements, a single call to insert() or erase() can require thousands of instructions, so if a program does many inserts or erases on large vectors, the program may run very slowly. The following animation illustrates shifting during an insert.

PARTICIPATION ACTIVITY 8.9.1: Vector insert performance problem.




The following program can be used to demonstrate the issue. The user inputs a vector size vectorSize, and a number numOps of elements to insert. The program then carries out several tasks, namely it resizes the vector to size vectorSIze, writes an arbitrary value to all vectorSIze elements, does numOps push_backs, numOps inserts, and numOps erases.

Figure 8.9.1: Program illustrating how slow vector inserts and erases can be.

```c
#include <stdio.h>
#include <stdlib.h>
#include "vector.h"

int main(void) {
   vector tempValues;  // Dummy vector to demo vector ops
   int vectorSize = 0; // User defined vector size
   int numOps = 0;     // User defined number of inserts
   int i = 0;          // Loop index

   vector_create(&tempValues, 0);

   printf("Enter initial vector size: ");
   scanf("%d", &vectorSize);

   printf("Enter number of inserts: ");
   scanf("%d", &numOps);

   printf("  Resizing vector...");
   fflush(stdout);
   vector_resize(&tempValues, vectorSize);

   printf("done.\n");
   printf("  Writing to each element...");
   fflush(stdout);

   for (i = 0; i < vectorSize; ++i) {
      *vector_at(&tempValues, i) = 777; // Any value
   }

   printf("done.\n");
   printf("   Doing %d inserts at end...", numOps);
   fflush(stdout);

   for (i = 0; i < numOps; ++i) {
      vector_insert(&tempValues, vector_size(&tempValues), 888); // Any value
   }

   printf("done.\n");
   printf("   Doing %d inserts at beginning...", numOps);
   fflush(stdout);

   for (i = 0; i < numOps; ++i) {
      vector_insert(&tempValues, 0, 444);
   }

   printf("done.\n");
   printf("   Doing %d removes...", numOps);
   fflush(stdout);


   for (i = 0; i < numOps; ++i) {
      vector_erase(&tempValues, 0);
   }

   printf("done.\n");

   return 0;
}
```



Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

```
Enter initial vector size: 100000
Enter number of inserts: 40000
  Resizing vector...done.                     (fast)
  Writing to each element...done.             (fast)
   Doing 40000 inserts at end...done.         (fast)
   Doing 40000 inserts at beginning...done.   (SLOW)
   Doing 40000 removes...done.                (SLOW)
```

The video shows the program running for different vectorSize and numOps values; notice that for large vectorSize and numOps, the resize, writes, and numOps push_backs all run quickly, but the numOps inserts and numOps erases take a noticeably long time. The `fflush(stdout);` forces any characters written to stdout to be displayed on the screen before doing each task, lest the characters be held in the buffer until a task completes.



Video 8.9.1: Vector inserts.

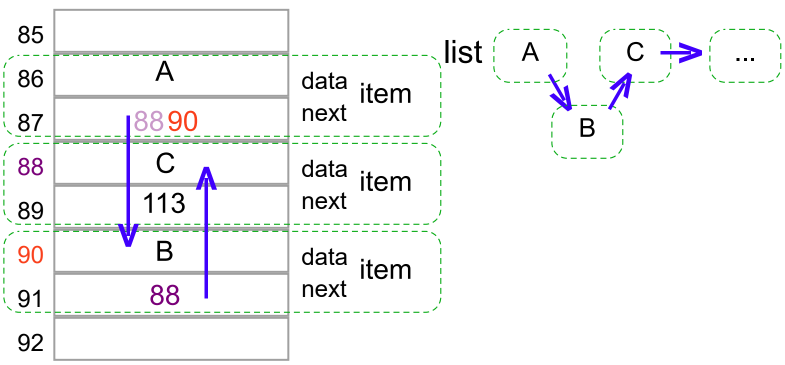
Programming example: Vector inserts

The vector_push_backs() are fast because they do not involve any shifting of elements, whereas each vector_insert() requires 500,000 elements to be shifted — one at a time. 7,500 inserts thus requires 3,750,000,000 (over 3 billion) shifts.

One way to make inserts or erases faster is to use a different approach for storing a list of items. The approach does not use contiguous memory locations. Instead, each item contains a "pointer" to the next item's location in memory, as well as, the data being stored. Thus, inserting a new item B between existing items A and C just requires changing A to point to B's memory location, and B to point to C's location, as shown in the following animation.

**PARTICIPATION ACTIVITY** 8.9.2: A list avoids the shifting problem.

Start




*Add new item B at location 90.*
*Change item A to point to 90.*
*Set item B to point to 88.*

*New list is (A, B, C, ...) -- no shifting of items after C*

A list whose items each point to the next item avoids the element shifting problem.

The animation begins with a list having some number of items, with the first two items being A and C. The first item has data A, and an address 88 pointing to the next item's location in memory, which just happens to be adjacent to the first item's location. That second item has data C, and an address 113 pointing to the next item (not shown). The animation shows a new item being created at memory location 90, having data B. To keep the list in sorted order, item B should go between A and C in the list. So item A's next pointer is changed to point to B's location of 90, and B's next pointer is set to point to 88.

A ***linked list*** is a list wherein each item contains not just data but also a pointer — a *link* — to the next item in the list. Comparing vectors and linked lists:

- *Vector*: Stores items in contiguous memory locations. Supports quick access to i'th element via `vector_at(&v, i)`, but may be slow for inserts or deletes on large lists due to necessary shifting of elements.
- *Linked list*: Stores each item anywhere in memory, with each item pointing to the next item in the list. Supports fast inserts or deletes, but access to i'th element may be slow as the list must be traversed from the first item to the i'th item. Also uses more memory due to storing a link for each item.



**PARTICIPATION ACTIVITY** 8.9.3: Vector performance.

1) Appending a new item to the end of a 1000 element vector requires how many elements to be shifted?

Check Show answer

2) Inserting a new item at the beginning of a 1000 element vector requires how many elements to be shifted?

Check Show answer

# 8.10 A first linked list



A common use of pointers is to create a list of items such that an item can be efficiently inserted somewhere in the middle of the list, without the shifting of later items as required for a vector. The following program illustrates how such a list can be created. A struct is defined to represent each list item, known as a ***list node***. A node is comprised of the data to be stored in each list item, in this case just one int, and a pointer to the next node in the list. A special node named head is created to represent the front of the list, after which regular items can be inserted.

Figure 8.10.1: A basic example to introduce linked lists.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct IntNode_struct {
   int dataVal;
   struct IntNode_struct* nextNodePtr;
} IntNode;

// Constructor
void IntNode_Create
(IntNode* thisNode, int dataInit, IntNode* nextLoc) {
   thisNode->dataVal = dataInit;
   thisNode->nextNodePtr = nextLoc;
   return;
}

/* Insert newNode after node.
 Before: thisNode -- next
 After:  thisNode -- newNode -- next
 */
void IntNode_InsertAfter
(IntNode* thisNode, IntNode* newNode) {
   IntNode* tmpNext = NULL;

   tmpNext = thisNode->nextNodePtr; // Remember next
   thisNode->nextNodePtr = newNode; // this -- new -- ?
   newNode->nextNodePtr = tmpNext;  // this -- new -- next
   return;
}

// Print dataVal
void IntNode_PrintNodeData(IntNode* thisNode) {
   printf("%d\n", thisNode->dataVal);
   return;
}

// Grab location pointed by nextNodePtr
IntNode* IntNode_GetNext(IntNode* thisNode) {
   return thisNode->nextNodePtr;
}

int main(void) {
   IntNode* headObj  = NULL; // Create intNode objects
   IntNode* nodeObj1 = NULL;
   IntNode* nodeObj2 = NULL;
   IntNode* nodeObj3 = NULL;
   IntNode* currObj  = NULL;

   // Front of nodes list
   headObj = (IntNode*)malloc(sizeof(IntNode));
   IntNode_Create(headObj, -1, NULL);

   // Insert nodes
   nodeObj1 = (IntNode*)malloc(sizeof(IntNode));
   IntNode_Create(nodeObj1, 555, NULL);
   IntNode_InsertAfter(headObj, nodeObj1);

   nodeObj2 = (IntNode*)malloc(sizeof(IntNode));
   IntNode_Create(nodeObj2, 999, NULL);
   IntNode_InsertAfter(nodeObj1, nodeObj2);

   nodeObj3 = (IntNode*)malloc(sizeof(IntNode));
   IntNode_Create(nodeObj3, 777, NULL);
   IntNode_InsertAfter(nodeObj1, nodeObj3);

   // Print linked list
   currObj = headObj;
   while (currObj != NULL) {
      IntNode_PrintNodeData(currObj);
```

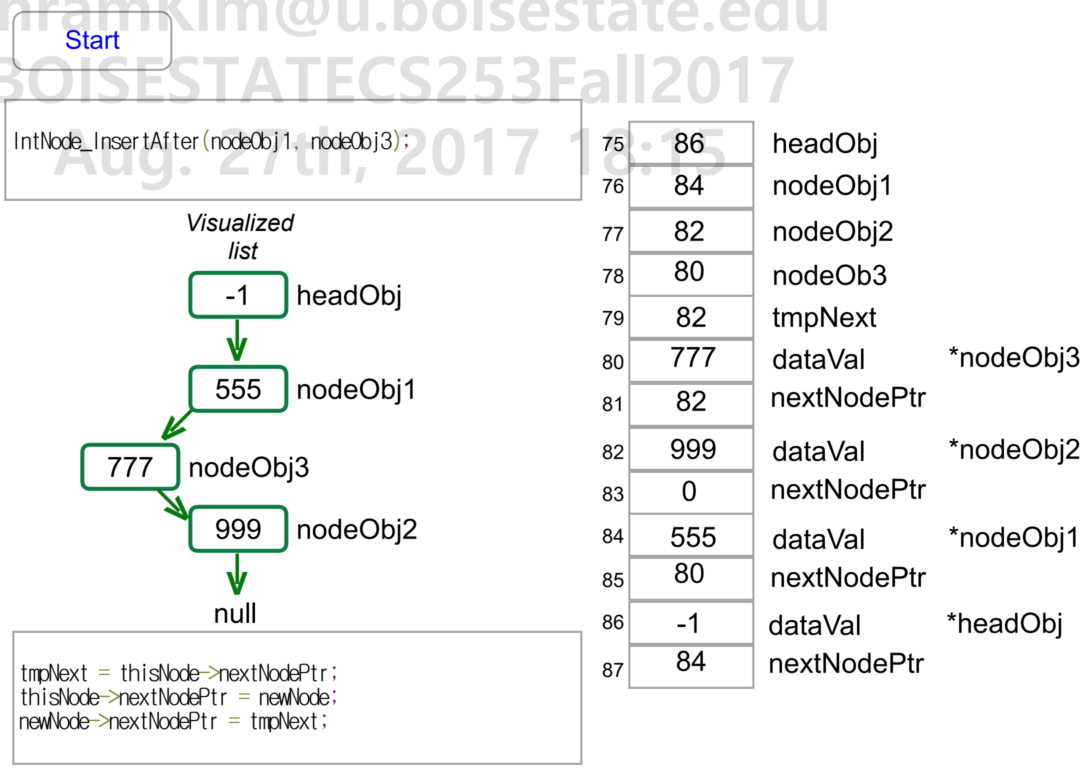
```
-1
555
777
999
```

```
        currObj = IntNode_GetNext(currObj);
    }

    return 0;
}
```

**PARTICIPATION ACTIVITY** 8.10.1: Inserting nodes into a basic linked list.






The most interesting part of the above program is the InsertAfter() function, which inserts a new node after a given node already in the list. The above animation illustrates.

**PARTICIPATION ACTIVITY** 8.10.2: A first linked list.

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

Some questions refer to the above linked list code and animation.

1) A linked list has what key advantage over a sequential storage approach like an array or vector?

- An item can be inserted somewhere in the middle of the list without having to shift all subsequent items.
- Uses less memory overall.

- Can store items other than int variables.

2) What is the purpose of a list's head node?

- Stores the first item in the list.
- Provides a pointer to the first item's node in the list, if such an item exists.
- Stores all the data of the list.



3) After the above list is done having items inserted, at what memory address is the last list item's node located?

- 80
- 82
- 84
- 86

4) After the above list has items inserted as above, if a fourth item was inserted at the front of the list, what would happen to the location of node1?

- Changes from 84 to 86.
- Changes from 84 to 82.
- Stays at 84.

In contrast to the above program that declares one variable for each item allocated by the malloc function, a program commonly declares just one or a few variables to manage a large number of items allocated using the malloc function. The following example replaces the above main() function, showing how just two pointer variables, currObj and lastObj, can manage 20 allocated items in the list.



Figure 8.10.2: Managing many new items using just a few pointer variables.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct IntNode_struct {
   int dataVal;
   struct IntNode_struct* nextNodePtr;
} IntNode;

// Constructor
void IntNode_Create
```

```
(IntNode* thisNode, int dataInit, IntNode* nextLoc) {
   thisNode->dataVal = dataInit;
   thisNode->nextNodePtr = nextLoc;
   return;
}

/* Insert newNode after node.
 Before: thisNode -- next
 After:  thisNode -- newNode -- next
 */
void IntNode_InsertAfter
(IntNode* thisNode, IntNode* newNode) {
   IntNode* tmpNext = NULL;

   tmpNext = thisNode->nextNodePtr; // Remember next
   thisNode->nextNodePtr = newNode;// this -- new -- ?
   newNode->nextNodePtr = tmpNext; // this -- new -- next
   return;
}

// Print dataVal
void IntNode_PrintNodeData(IntNode* thisNode) {
   printf("%d\n", thisNode->dataVal);
   return;
}

// Grab location pointed by nextNodePtr
IntNode* IntNode_GetNext(IntNode* thisNode) {
   return thisNode->nextNodePtr;
}

int main(void) {
   IntNode* headObj  = NULL; // Create intNode objects
   IntNode* currObj  = NULL;
   IntNode* lastObj  = NULL;
   int i             = 0;    // Loop index

   headObj = (IntNode*)malloc(sizeof(IntNode)); // Front of nodes list
   IntNode_Create(headObj, -1, NULL);
   lastObj = headObj;

   for (i = 0; i < 20; ++i) {                 // Append 20 rand nums
      currObj = (IntNode*)malloc(sizeof(IntNode));
      IntNode_Create(currObj, rand(), NULL);

      IntNode_InsertAfter(lastObj, currObj);    // Append curr
      lastObj = currObj;                        // Curr is the new last item
   }


   currObj = headObj;                           // Print the list

   while (currObj != NULL) {
      IntNode_PrintNodeData(currObj);
      currObj = IntNode_GetNext(currObj);
   }

   return 0;
}
```

```
-1
1481765933
1085377743
1270216262
1191391529
812669700
553475508
445349752
1344887256
730417256
1812158119
147699711
880268351
1889772843
686078705
2105754108
182546393
1949118330
220137366
1979932169
1089957932
```



**PARTICIPATION ACTIVITY** 8.10.3: Managing a linked list.

Finish the program so that it finds and prints the smallest value in the linked list.

Load default template... Run

```
#include <stdio.h>
#include <stdlib.h>

typedef struct IntNode_struct {
   int dataVal;
   struct IntNode_struct* nextNodePtr;
} IntNode;

// Constructor
void IntNode_Create
(IntNode* thisNode, int dataInit, IntNode* nextLoc) {
   thisNode->dataVal = dataInit;
   thisNode->nextNodePtr = nextLoc;
   return;
}

/* Insert newNode after node.
 Before: thisNode -- next
 After: thisNode -- newNode -- next
```



Normally, a linked list would be implemented as an ADT using a set interface functions and struct type declaration, such as IntList. Data members of that struct might include a pointer to the list head, the list size, and a pointer to the list tail (the last node in the list). Supporting functions might include InsertAfter (insert a new node after the given node), PushBack (insert a new node after the last node), PushFront (insert a new node at the front of the list, just after the head), DeleteNode (deletes the node from the list), etc.

**CHALLENGE ACTIVITY** 8.10.1: Linked list negative values counting.

Assign negativeCntr with the number of negative values in the linked list.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct IntNode_struct {
   int dataVal;
   struct IntNode_struct* nextNodePtr;
} IntNode;

// Constructor
void IntNode_Create(IntNode* thisNode, int dataInit, IntNode* nextLoc) {
   thisNode->dataVal = dataInit;
   thisNode->nextNodePtr = nextLoc;
}

/* Insert newNode after node.
 Before: thisNode -- next
 After: thisNode -- newNode -- next
*/
void IntNode_InsertAfter(IntNode* thisNode, IntNode* newNode) {
   IntNode* tmpNext = NULL;

```

Run



# 8.11 Memory regions: Heap/Stack

A program's memory usage typically includes four different regions:

- ***Code*** — The region where the program instructions are stored.
- ***Static memory*** — The region where global variables (variables declared outside any function) as well as static local variables (variables declared inside functions starting with the keyword "static") are allocated. The name "static" comes from these variables not changing (static means not changing); they are allocated once and last for the duration of a program's execution, their addresses staying the same.
- ***The stack*** — The region where a function's local variables are allocated during a function call. A function call adds local variables to the stack, and a return removes them, like adding and removing dishes from a pile; hence the term "stack." Because this memory is automatically allocated and deallocated, it is also called ***automatic memory***.
- ***The heap*** — The region where the malloc function allocates memory, and where the free function deallocates memory. The region is also called ***free store***

The following animation illustrates:

**PARTICIPATION ACTIVITY** 8.11.1: Use of the four regions of memory.

Start

```
#include <stdio.h>
#include <stdlib.h>

// Program is stored in code memory

int myGlobal = 33;    // In static memory

void MyFct() {
   int myLocal = 999; // On stack
   printf(" %d", myLocal);
   return;
}

int main(void) {
   int myInt = 555;   // On stack
   int* myPtr = 0;    // On stack

   myPtr = (int *)malloc(sizeof(int)); // In heap
   *myPtr = 222;
   printf("%d %d", *myPtr, myInt);
   free(myPtr);       // Deallocated from heap
```

| | | | |
|---|---|---|---|
| | | | ***Code mem*** |
| 1 | Add R1, #1, R2 | | |
| 2 | Sub R3, #1, R4 | | |
| 3 | Add R1, R3, R5 | | |
| 4 | Jmp 40 | | |
| | ... | | ***Static mem*** |
| 3000 | 33 | myGlobal | |
| 3001 | | | |
| | ... | | ***Stack*** |
| 3200 | 555 | myInt | |
| 3201 | 9400 | myPtr | |
| 3202 | 999 | myLocal | |
| 3203 | | | |
| | ... | | ***Heap*** |
| 9400 | 222 | (int) | |
| | | | |

```
    MyFct();          // Stack grows, then shrinks

    return 0;
}
```

9401
9402
...

PARTICIPATION ACTIVITY 8.11.2: Stack and heap definitions.



The heap | Code | Automatic memory | The stack | Free store | Static memory

A function's local variables are allocated in this region while a function is called.

The memory allocation and deallocation operators affect this region.

Global and static local variables are allocated in this region once for the duration of the program.

Another name for "The heap" because the programmer has explicit control of this memory.

Instructions are stored in this region.

Another name for "The stack" because the programmer does not explicitly control this memory.



Reset

# 8.12 Memory leaks

A program that allocates memory but then loses the ability to access that memory, typically due to failure to properly destroy/free dynamically allocated memory, is said to have a ***memory leak***. The program's available memory has portions leaking away and becoming unusable, much like a water pipe might have water leaking out and becoming unusable. A memory leak may cause a program to occupy more and more memory as the program runs. Such occupying of memory can slow program runtime, or worse can cause the program to fail if the memory becomes completely full and the program is unable to allocate additional memory. The following animation illustrates.

**PARTICIPATION ACTIVITY** 8.12.1: Memory leak can use up all available memory.



**Animation captions:**

1. Memory is allocated for newVal each loop iteration, but the loop does not deallocate memory once done using newVal, resulting in a memory leak.
2. Each loop iteration allocates more memory, eventually using up all available memory, causing the program to fail.

Failing to free allocated memory when done using that memory, resulting in a memory leak, is a common error. Many programs that are commonly left running for long periods, such as web browsers, suffer from known memory leak problems — just do a web search for "<your-favorite-browser> memory leak" and you'll likely find numerous hits.

Some programming languages, such as Java, use a mechanism called ***garbage collection*** wherein a program's executable includes automatic behavior that at various intervals finds all unreachable allocated memory locations (e.g., by comparing all reachable memory with all previously-allocated memory), and automatically freeing such unreachable memory. Some C/C++ implementations include garbage collection but those implementations are not standard. Garbage collection can reduce the impact of memory leaks, at the expense of runtime overhead. Computer scientists debate whether new programmers should learn to explicitly free memory versus letting garbage collection do the work.

**PARTICIPATION ACTIVITY** 8.12.2: Memory Leaks.

**Memory leak** **Garbage collection** **Unusable memory**

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

Memory locations that have been dynamically allocated but can no longer be used by a program.

Occurs when a program allocates memory but loses the ability to access that memory.

Automatic process of finding unreachable allocated memory locations freeing that unreachable memory.

Reset



# 8.13 C example: Employee list using vector ADT

**PARTICIPATION ACTIVITY** 8.13.1: Managing an employee list using a vector ADT.

The following program allows a user to add to and list entries from a vector ADT, which maintains a list of employees.

1. Run the program, and provide input to add three employees' names and related data. Then use the list option to display the list.
2. Modify the program to implement the deleteEntry function.
3. Run the program again and add, list, delete, and list again various entries.

Current file: main.c

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "vector_employee.h"

// Add an employee
void AddEmployee(vector *employees) {
   Employee theEmployee;

   printf("\nEnter the name to add: \n");
   fgets(theEmployee.name, 50, stdin);
   theEmployee.name[strlen(theEmployee.name) - 1] = '\0'; // Remove trailing newline

   printf("Enter %s's department: \n", theEmployee.name);
   fgets(theEmployee.department, 50, stdin);
   theEmployee.department[strlen(theEmployee.department) - 1] = '\0'; // Remove trailing newline

   printf("Enter %s's title: \n", theEmployee.name);
   fgets(theEmployee.title, 50, stdin);
   theEmployee.title[strlen(theEmployee.title) - 1] = '\0'; // Remove trailing newline

```

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

```
a
Rajeev Gupta
Sales
```

Run

Below is a solution to the above problem.

**PARTICIPATION ACTIVITY** 8.13.2: Managing an employee list using a vector ADT (solution).



Current file: **main.c**

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "vector_employee.h"

// Add an employee
void AddEmployee(vector *employees) {
   Employee theEmployee;

   printf("\nEnter the name to add: \n");
   fgets(theEmployee.name, 50, stdin);
   theEmployee.name[strlen(theEmployee.name) - 1] = '\0'; // Remove trailing newline

   printf("Enter %s's department: \n", theEmployee.name);
   fgets(theEmployee.department, 50, stdin);
   theEmployee.department[strlen(theEmployee.department) - 1] = '\0'; // Remove trailing newline

   printf("Enter %s's title: \n", theEmployee.name);
   fgets(theEmployee.title, 50, stdin);
   theEmployee.title[strlen(theEmployee.title) - 1] = '\0'; // Remove trailing newline

```

a
Rajeev Gupta
Sales

Run