C-Programming Cont'd part 2

pointers in expressions (cont'd)

- a variable defined as a "pointer to int" can store the address of any variable of type int

```
int value = 999;
pnumber = &value;
*pnumber +=  25;
```

- the statement will operate with the new variable, value
  - the new contents of value will be 1024

- a pointer can contain the address of any variable of the appropriate type
  - you can use one pointer variable to change the values of many different variables
  - as long as they are of a type compatible with the pointer type

when receiving Input

- when we have used scanf() to input values, we have used the & operator to obtain the address of a variable
  - on the variable that is to store the input (second argument)

- when you have a pointer that already contains an address, you can use the pointer name as an argument for scanf()

```
int value = 0;
int *pvalue = &value;                 // Set pointer to refer to value

printf ("Input an integer: ");
scanf(" %d", pvalue);                 // Read into value via the pointer

printf("You entered %d.\n", value);       // Output the value entered
```

Testing for NULL

- there is one rule you should burn into your memory
  - do not dereference an uninitialized pointer

```
int * pt;  // an uninitialized pointer
*pt = 5;   // a terrible error
```

- second line means store the value 5 in the location to which pt points
  - pt has a random value, there is no knowing where the 5 will be placed

- It might go somewhere harmless, it might overwrite data or code, or it might cause the program to crash

- creating a pointer only allocates memory to store the pointer itself
  - it does not allocate memory to store data
  - before you use a pointer, it should be assigned a memory location that has already been allocated
    - assign the address of an existing variable to the pointer
    - or you can use the malloc() function to allocate memory first

Testing for NULL (cont'd)

- we already know that when declaring a pointer that does not point to anything, we should initialize it to NULL

```
int *pvalue = NULL;
```

- NULL is a special symbol in C that represents the pointer equivalent to 0 with ordinary numbers
  - the below also sets a pointer to null using 0

```
int *pvalue = 0;
```

- because NULL is the equivalent of zero, if you want to test whether pvalue is NULL, you can do this:
  - or you can do it explicitly by using == NULL

```
if(!pvalue)  .....
```

- you want to check for NULL before you dereference a pointer
  - often when pointers are passed to functions

Keep practicing

- stay with me
  - pointers can be confusing :)
    - you can work with addresses
    - you can work with values
    - you can work with pointers
    - you can work with variables

- sometimes it is hard to work out what exactly is going on

- the best thing to understand this concept of a pointer is to keep writing short programs that use pointers
  - getting values using pointers
  - changing values using pointers
  - printing addresses, etc.

- this is the only way to really get confident about using pointers, practice!!!!

Overview

- when we use the const modifier on a variable or an array it tells the compiler that the contents of the variable/array will not be changed by the program

- with pointers, we have to consider two things when using the const modifier
  - whether the pointer will be changed
  - whether the value that the pointer points to will be changed

- you can use the const keyword when you declare a pointer to indicate that the value pointed to must not be changed

```
long value = 9999L;
const long *pvalue = &value;          // defines a pointer to a constant
```

- you have declared the value pointed to by pvalue to be const
  - the compiler will check for any statements that attempt to modify the value pointed to by pvalue and flag such statements as an error

- the following statement will now result in an error message from the compiler

- `*pvalue = 8888L;          // Error - attempt to change const location`

pointers to constants

- you can still modify value (you have only applied const to the pointer)

```
value = 7777L;
```

- the value pointed to has changed, but you did not use the pointer to make the change

- the pointer itself is not constant, so you can still change what it points to:

```
long number = 8888L;
pvalue = &number;          // OK - changing the address in pvalue
```

- will change the address stored in pvalue to point to number
  - still cannot use the pointer to change the value that is stored
  - you can change the address stored in the pointer as much as you like
  - using the pointer to change the value pointed to is not allowed, even after you have changed the address stored in the pointer.

constant pointers

- you might also want to ensure that the address stored in a pointer cannot be changed

- you can do this by using the const keyword in the declaration of the pointer

```
int count = 43;
int *const pcount = &count;      // Defines a constant pointer
```

- the above ensures that a pointer always points to the same thing
  - indicates that the address stored must not be changed
  - compiler will check that you do not inadvertently attempt to change what the pointer points to elsewhere in your code

```
int item = 34;
pcount = &item;          // Error - attempt to change a constant pointer
```

- it is all about where you place the const keyword, either before the type or after the type
  - `const int * .....          // value can not be changed`
  - `int *const  .....          // pointer address cannot change`

constant pointers (cont'd)

- you can still change the value that pcount points to using pcount

```
*pcount = 345;              // OK - changes the value of count
```

- references the value stored in count through the pointer and changes its value to 345

- you can create a constant pointer that points to a value that is also constant:

```
int item = 25;
const int *const pitem = &item;
```

- the pitem is a constant pointer to a constant so everything is fixed
  - cannot change the address stored in pitem
  - cannot use pitem to modify what it points to

- you can still change the value of item directly
  - if you wanted to make everything not change, you could specify item as const as well

Overview

- the type name void means absence of any type
- a pointer of type void* can contain the address of a data item of any type
- void* is often used as a parameter type or return value type with functions that deal with data in a type-independent way
- any kind of pointer can be passed around as a value of type void*
  - the void pointer does not know what type of object it is pointing to, so, it cannot be dereferenced directly
  - the void pointer must first be explicitly cast to another pointer type before it is dereferenced
- the address of a variable of type int can be stored in a pointer variable of type void*
- when you want to access the integer value at the address stored in the void* pointer, you must first cast the pointer to type int*

Example

```
int i = 10;
float f = 2.34;
char ch = 'k';

void *vptr;

vptr = &i;
printf("Value of i = %d\n", *(int *)vptr);

vptr = &f;
printf("Value of f = %.2f\n", *(float *)vptr);

vptr = &ch;
printf("Value of ch = %c\n", *(char *)vptr);
```

Overview

- an array is a collection of objects of the same type that you can refer to using a single name
- a pointer is a variable that has as its value a memory address that can reference another variable or constant of a given type
  - you can use a pointer to hold the address of different variables at different times (must be same type)
- arrays and pointers seem quite different, but, they are very closely related and can sometimes be used interchangeably
- one of the most common uses of pointers in C is as pointers to arrays
- the main reasons for using pointers to arrays are ones of notational convenience and of program efficiency
- pointers to arrays generally result in code that uses less memory and executes faster

Arrays and Pointers

- if you have an array of 100 integers

```
int values[100];
```

- you can define a pointer called valuesPtr, which can be used to access the integers contained in this array

```
int  *valuesPtr;
```

- when you define a pointer that is used to point to the elements of an array, you do not designate the pointer as type "pointer to array"
  - you designate the pointer as pointing to the type of element that is contained in the array
- to set valuesPtr to point to the first element in the values array, you write

```
valuesPtr = values;
```

- the address operator is not used
  - the C compiler treats the appearance of an array name without a subscript as a pointer to the array
  - specifying values without a subscript has the effect of producing a pointer to the first element of values

Arrays and Pointers

- an equivalent way of producing a pointer to the start of values is to apply the address operator to the first element of the array

```
valuesPtr = &values[0];
```

- So, you can use the above example or the one on the previous slide

```
valuesPtr = values;
```

- either one is fine and a matter of programmer preference

pointer arithmetic

- the real power of using pointers to arrays comes into play when you want to sequence through the elements of an array

*valuesPtr // can be used to access the first integer of the values array, that is, values[0]

- to reference values[3] through the valuesPtr variable, you can add 3 to valuesPtr and then apply the indirection operator

*(valuesPtr + 3)

- the expression, *(valuesPtr + i) can be used to access the value contained in values[i]
  - to set values[10] to 27, you could do the following

values[10] = 27;

- or, using valuesPtr, you could

*(valuesPtr + 10) = 27;

pointer arithmetic (cont'd)

- to set valuesPtr to point to the second element of the values array, you can apply the address operator to values[1] and assign the result to valuesPtr

valuesPtr = &values[1];

- If valuesPtr points to values[0], you can set it to point to values[1] by simply adding 1 to the value of valuesPtr

valuesPtr += 1;

- this is a perfectly valid expression in C and can be used for pointers to any data type

pointer arithmetic (cont'd)

- the increment and decrement operators ++ and -- are particularly useful when dealing with pointers

- using the increment operator on a pointer has the same effect as adding one to the pointer
- using the decrement operator has the same effect as subtracting one from the pointer

++valuesPtr;

- sets valuesPtr pointing to the next integer in the values array (values[1])

--textPtr;

- sets valuesPtr pointing to the previous integer in the values array, assuming that valuesPtr was not pointing to the beginning of the values array

Example

```
int  arraySum (int  array[], const int  n)
{
    int  sum = 0, *ptr;
    int  * const arrayEnd = array + n;

    for ( ptr = array;  ptr < arrayEnd;  ++ptr )
        sum += *ptr;

    return sum;
}

void main (void)
{
    int  arraySum (int  array[], const int  n);
    int  values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };

    printf ("The sum is %i\n", arraySum (values, 10));
}
```

Example (cont'd)

- to pass an array to a function, you simply specify the name of the array
- to produce a pointer to an array, you need only specify the name of the array

- this implies that in the call to the arraySum() function, what was passed to the function was actually a pointer to the array values
  - explains why you are able to change the elements of an array from within a function

- so, you might wonder why the formal parameter inside the function is not declared to be a pointer

```
int  arraySum (int  *array, const int  n)
```

- the above is perfectly valid
  - pointers and arrays are intimately related in C
  - this is why you can declare array to be of type "array of ints" inside the arraySum function or to be of type "pointer to int."

- if you are going to be using index numbers to reference the elements of an array that is passed to a function, declare the corresponding formal parameter to be an array
  - more correctly reflects the use of the array by the function

Example with pointer notation

```
int  arraySum (int  *array, const int  n)
{
    int  sum = 0;
    int  * const arrayEnd = array + n;

    for (  ; array < arrayEnd;  ++array )
        sum += *array;

    return sum;
}

void main (void)
{
    int  arraySum (int  *array, const int  n);
    int  values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };

    printf ("The sum is %i\n", arraySum (values, 10));
}
```

Summary

```
int urn[3];
int * ptr1, * ptr2;
```

Valid	Invalid
`ptr1++;`	`urn++;`
`ptr2 = ptr1 + 2;`	`ptr2 = ptr2 + ptr1;`
`ptr2 = urn + 1;`	`ptr2 = urn * ptr1;`

Taken from C Primer Plus, Prata

Passing Array address vs passing pointer

```c
#include <stdio.h>

int ArraySum(int array[]
, int count);

int main(int argc, char **argv)
{
    //printf("hello world\n");
    int values[]={1,2,3,4,5,6,7,8,9,10};
    int count=10;

    printf("Sum of array element is %d\n", ArraySum(values,count));
    return 0;
}
// using array
int ArraySum(int array[], int count){
    int * const ArrayEnd=array+ count;
    int sum=0, *ptr;
    for(ptr=array ;ptr < ArrayEnd; ptr++ ){
        sum+=*ptr;
    }
    return sum;
}


//passing pointer
int ArraySum(int *array, int count){
    int * const ArrayEnd=array+ count;
    int sum=0;
    for( ;array < ArrayEnd; array++ ){
        sum+=*array;
    }
    return sum;
}
```

Example (array parameter vs char * parameter)

```
void copyString (char  to[], char  from[])
{
    int  i;

    for ( i = 0;  from[i] != '\0';  ++i )
        to[i] = from[i];

    to[i] = '\0';
}

void copyString (char  *to, char  *from)
{
    for (  ;  *from != '\0';  ++from, ++to )
        *to = *from;
    *to = '\0';
}
```


Example (array parameter vs char * parameter)

```
void copyString (char  to[], char  from[])
{
    int  i;

    for ( i = 0;  from[i] != '\0';  ++i )
        to[i] = from[i];

    to[i] = '\0';
}

void copyString (char  *to, char  *from)
{
    for (  ;  *from != '\0';  ++from, ++to )
        *to = *from;
    *to = '\0';
}
```


char arrays as pointers

- if you have an array of characters called text, you could similarly define a pointer to be used to point to elements in text

```
char  *textPtr;
```

- if textPtr is set pointing to the beginning of an array of chars called text

```
++textPtr;
```

- the above sets textPtr pointing to the next character in text, which is text[1]

```
--textPtr;
```

- the above sets textPtr pointing to the previous character in text, assuming that textPtr was not pointing to the beginning of text prior to the execution of this statement

Example optimized

```
void  copyString (char  *to, char  *from)
{
    while ( *from )    // the null character is equal to the value 0, so will jump out then
        *to++ = *from++;

    *to = '\0';
}

int main (void)
{
   char  string1[] = "A string to be copied.";
   char  string2[50];

    copyString (string2, string1);
    printf ("%s\n", string2);
}
```

Requirements

- In this challenge, you are going to write a program that tests your understanding of pointer arithmetic and the const modifier

- write a function that calculates the length of a string
  - the function should take as a parameter a const char pointer
  - the function can only determine the length of the string using pointer arithmetic
    - incrementation operator (++pointer) to get to the end of the string
  - you are required to use a while loop using the value of the pointer to exit
  - the function should subtract two pointers (one pointing to the end of the string and one pointing to the beginning of the string)
  - the function should return an int that is the length of the string passed into the function

```c
#include <stdio.h>

int stringLength(char inputString[]);

int main(int argc, char **argv)
{
    //printf("hello world\n");
    char name[]="Srikanth";

    printf("Length of string is %d\n",stringLength(name));
    return 0;
}

int stringLength(const char *firstCharacterptr){

    printf("%p\n",firstCharacterptr);
    const char *stringPtr=firstCharacterptr;

    while(*stringPtr){
        *stringPtr++;
        printf("%p\n",stringPtr);
    }

    return stringPtr-firstCharacterptr;

}
```

pass by value

- there are a few different ways you can pass data to a function
  - pass by value
  - pass by reference

- pass by value is when a function copies the actual value of an argument into the formal parameter of the function
  - changes made to the parameter inside the function have no effect on the argument

- C programming uses call by value to pass arguments
  - means the code within a function cannot alter the arguments used to call the function
  - there are no changes in the values, though they had been changed inside the function

Example, pass by value

```
/* function definition to swap the values */
void swap(int x, int y) {
   int temp;

   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */

   return;
}
```

Passing data using copies of pointers

- pointers and functions get along quite well together
  - you can pass a pointer as an argument to a function and you can also have a function return a pointer as its result

- pass by reference copies the address of an argument into the formal parameter
  - the address is used to access the actual argument used in the call
  - means the changes made to the parameter affect the passed argument

- to pass a value by reference, argument pointers are passed to the functions just like any other value
  - you need to declare the function parameters as pointer types
  - changes inside the function are reflected outside the function as well
  - unlike call by value where the changes do not reflect outside the function

Example using pointers to pass data

```
/* function definition to swap the values */
void swap(int *x, int *y) {

   int temp;
   temp = *x;    /* save the value at address x */
   *x = *y;      /* put y into x */
   *y = temp;    /* put temp into y */

   return;
}
```

Summary of syntax

- you can communicate two kinds of information about a variable to a function

function1(x);

- you transmit the value of x and the function must be declared with the same type as x

int function1(int num)

function2(&x);

- you transmit the address of x and requires the function definition to include a pointer to the correct type

int function2(int * ptr)

const pointer parameters

- you can qualify a function parameter using the const keyword
  - indicates that the function will treat the argument that is passed for this parameter as a constant
  - only useful when the parameter is a pointer

- you apply the const keyword to a parameter that is a pointer to specify that a function will not change the value to which the argument points

```
bool SendMessage(const char* pmessage)
{
  // Code to send the message
  return true;
}
```

- the type of the parameter, pmessage, is a pointer to a const char.
  - it is the char value that's const, not its address.
  - you could specify the pointer itself as const too, but this makes little sense because the address is passed by value
    - you cannot change the original pointer in the calling function

- the compiler knows that an argument that is a pointer to constant data will be safe

- If you pass a pointer to constant data as the argument for a parameter then the parameter must be a use the above

returning pointers from a function

- returning a pointer from a function is a particularly powerful capability
  - it provides a way for you to return not just a single value, but a whole set of values

- you would have to declare a function returning a pointer

```
int * myFunction() {
   .
   .
   .
}
```

- be careful though, there are specific hazards related to returning a pointer
  - use local variables to avoid interfering with the variable that the argument points to

Overview

- whenever you define a variable in C, the compiler automatically allocates the correct amount of storage for you based on the data type

- it is frequently desirable to be able to dynamically allocate storage while a program is running

- if you have a program that is designed to read in a set of data from a file into an array in memory, you have three choices
  - define the array to contain the maximum number of possible elements at compile time
  - use a variable-length array to dimension the size of the array at runtime
  - allocate the array dynamically using one of C's memory allocation routines

Dynamic memory allocation

- with the first approach, you have to define your array to contain the maximum number of elements that would be read into the array

```
Int dataArray [1000];
```

- the data file cannot contain more that 1000 elements, if it does, your program will not work
  - If it is larger that 1000 you must go back to the program, change the size to be larger and recompile it
  - no matter what value you select, you always have the chance of running into the same problem again in the future

- using the dynamic memory allocation functions, you can get storage as you need it
  - this approach enables you to allocate memory as the program is executing

Dynamic memory allocation

- dynamic memory allocation depends on the concept of a pointer and provides a strong incentive to use pointers in your code
- dynamic memory allocation allows memory for storing data to be allocated dynamically when your program executes
  - allocating memory dynamically is possible only because you have pointers available
- the majority of production programs will use dynamic memory allocation
- allocating data dynamically allows you to create pointers at runtime that are just large enough to hold the amount of data you require for the task

Heap vs. Stack

- dynamic memory allocation reserves space in a memory area called the heap
- the stack is another place where memory is allocated
  - function arguments and local variables in a function are stored here
  - when the execution of a function ends, the space allocated to store arguments and local variables is freed
- the memory in the heap is different in that it is controlled by you
  - when you allocate memory on the heap, it is up to you to keep track of when the memory you have allocated is no longer required
  - you must free the space you have allocated to allow it to be reused

malloc

- the simplest standard library function that allocates memory at runtime is called malloc()
  - need to include the stdlib.h header file
  - you specify the number of bytes of memory that you want allocated as the argument
  - returns the address of the first byte of memory that it allocated
  - because you get an address returned, a pointer is the only place to put it

```
int *pNumber = (int*)malloc(100);
```

- in the above, you have requested 100 bytes of memory and assigned the address of this memory block to pNumber
  - pNumber will point to the first int location at the beginning of the 100 bytes that were allocated.
  - can hold 25 int values on my computer, because they require 4 bytes each
  - assumes that type int requires 4 bytes
- it would be better to remove the assumption that ints are 4 bytes

```
int *pNumber = (int*)malloc(25*sizeof(int));
```

- the argument to malloc() above is clearly indicating that sufficient bytes for accommodating 25 values of type int should be made available
- also notice the cast (int*), which converts the address returned by the function to the type pointer to int
  - malloc returns a pointer of type pointer to void, so you have to cast

Malloc pointer

malloc (cont'd)

- you can request any number of bytes

- if the memory that you requested can not be allocated for any reason
  - malloc() returns a pointer with the value NULL
  - It is always a good idea to check any dynamic memory request immediately using an if statement to make sure the memory is actually there before you try to use it

```
int *pNumber = (int*)malloc(25*sizeof(int));
if(!pNumber)
{
  // code to deal with memory allocation failure . . .
}
```

- you can at least display a message and terminate the program
  - much better than allowing the program to continue and crash when it uses a NULL address to store something

releasing memory

- when you allocate memory dynamically, you should always release the memory when it is no longer required

- memory that you allocate on the heap will be automatically released when your program ends
  - better to explicitly release the memory when you are done with it, even if it's just before you exit from the program

- a memory leak occurs when you allocate some memory dynamically and you do not retain the reference to it, so you are unable to release the memory
  - often occurs within a loop
  - because you do not release the memory when it is no longer required, your program consumes more and more of the available memory on each loop iteration and eventually may occupy it all

- to free memory that you have allocated dynamically, you must still have access to the address that references the block of memory

releasing memory (cont'd)

- to release the memory for a block of dynamically allocated memory whose address you have stored in a pointer

```
free(pNumber);
pNumber = NULL;
```

- the free() function has a formal parameter of type void*
  - you can pass a pointer of any type as the argument

- as long as pNumber contains the address that was returned when the memory was allocated, the entire block of memory will be freed for further use

- you should always set the pointer to NULL after the memory that it points to has been freed

calloc

- the calloc() function offers a couple of advantages over malloc()
  - it allocates memory as a number of elements of a given size
  - it initializes the memory that is allocated so that all bytes are zero

- calloc() function requires two argument values
  - number of data items for which space is required
  - size of each data item

- is declared in the stdlib.h header

```
int *pNumber = (int*) calloc(75, sizeof(int));
```

- the return value will be NULL if it was not possible to allocate the memory requested
  - very similar to using malloc(), but the big plus is that you know the memory area will be initialized to 0

realloc

- the realloc() function enables you to reuse or extend memory that you previously allocated using malloc() or calloc()

- expects two argument values
  - a pointer containing an address that was previously returned by a call to malloc(), calloc()
  - the size in bytes of the new memory that you want allocated

- allocates the amount of memory you specify by the second argument
  - transfers the contents of the previously allocated memory referenced by the pointer that you supply as the first argument to the newly allocated memory
  - returns a void* pointer to the new memory or NULL if the operation fails for some reason

- the most important feature of this operation is that realloc() preserves the contents of the original memory area

Example

```
int main () {
   char *str;

   /* Initial memory allocation */
   str = (char *) malloc(15);
   strcpy(str, “jason");
   printf("String = %s,  Address = %u\n", str, str);

   /* Reallocating memory */
   str = (char *) realloc(str, 25);
   strcat(str, ".com");
   printf("String = %s,  Address = %u\n", str, str);

   free(str);

   return(0);
}
```

guidelines

- avoid allocating lots of small amounts of memory
  - allocating memory on the heap carries some overhead with it
  - allocating many small blocks of memory will carry much more overhead than allocating fewer larger blocks

- only hang on to the memory as long as you need it
  - as soon as you are finished with a block of memory on the heap, release the memory

- always ensure that you provide for releasing memory that you have allocated
  - decide where in your code you will release the memory when you write the code that allocates it

- make sure you do not inadvertently overwrite the address of memory you have allocated on the heap before you have released it
  - will cause a memory leak
  - be especially careful when allocating memory within a loop

Requirements

- In this challenge, you are going to write a program that tests your understanding of dynamic memory allocation

- write a program that allows a user to input a text string. The program will print the text that was inputted. The program will utilize dynamic memory allocation.

- the user can enter the limit of the string they are entering
  - you can use this limit when invoking malloc

- the program should create a char pointer only, no character arrays

- be sure to release the memory that was allocated