C-Programming Cont'ed

Defining a function

- when you create a function, you specify the function header as the first line of the function definition
  - followed by a starting curly brace {
  - The executable code in between the starting and ending braces
  - The ending curly brace }
  - the block of code between braces following the function header is called the function body.

- the function header defines the name of the function
  - parameters (which specify the number and types of values that are passed to the function when it's called)
  - the type for the value that the function returns

- the function body contains the statements that are executed when the function is called
  - have access to any values that are passed as arguments to the function

```
Return_type  Function_name( Parameters - separated by commas )
{
   // Statements...
}
```

Defining a function

- the statements in the function body can be absent, but the braces must be present

- If there are no statements in the body of a function, the return type must be void, and the function will not do anything
  - defining a function with an empty body is often useful during the testing phase of a complicated program
  - allows you to run the program with only selected functions actually doing something
  - you can then add the detail for the function bodies step by step, testing at each stage, until the whole thing is implemented and fully tested

Naming functions

- the name of a function can be any legal name
  - not a reserved word (such as int, double, sizeof, and so on)
  - Is not the same name as another function in your program.
  - Is not the same name as any of the standard library functions
    - would prevent you from using the library function

- a legal name has the same form as that of a variable
  - a sequence of letters and digits
  - first character must be a letter
  - underline character counts as a letter

Function prototypes

- a function prototype is a statement that defines a function
  - defines its name, its return value type, and the type of each of its parameters
  - provides all the external specifications for the function

- you can write a prototype for a function exactly the same as the function header
  - only difference is that you add a semicolon at the end

```
void printMessage (void);
```

- a function prototype enables the compiler to generate the appropriate instructions at each point where you call the function
  - It also checks that you are using the function correctly in each invocation

- when you include a standard header file in a program, the header file adds the function prototypes for that library to your program
  - the header file stdio.h contains function prototypes for printf(), among others

Function prototypes (cont'd)

- generally appear at the beginning of a source file prior to the implementations of any functions or in a header file

- allows any of the functions in the file to call any function regardless of where you have placed the implementation of the functions

- parameter names do not have to be the same as those used in the function definition
  - not required to include the names of parameters in a function prototype

- its good practice to always include declarations for all of the functions in a program source file, regardless of where the are called
  - will help keep your programs more consistent in design
  - prevent any errors from occurring if, at any stage, you choose to call a function from another part of your program

Example

```
// #include & #define directives...

// Function prototypes
double Average(double data_values[], size_t count);
double Sum(double *x, size_t n);
size_t GetData(double*, size_t);

int main(void)
{
  // Code in main() ...
}

// Definitions/implementations for Average(), Sum() and GetData()...
```

Arguments and parameters

- a parameter is a variable in a function declaration and function definition/implementation
- when a function is called, the arguments are the data you pass into the functions parameters.
  - the actual value of a variable that gets passed to the function
- function parameters are defined within the function header
  - are placeholders for the arguments that need to be specified when the function is called
- the parameters for a function are a list of parameter names with their types
  - each parameter is separated by a comma
  - entire list of parameters is enclosed between the parentheses that follow the function name
- a function can have no parameters, in which case you should put void between the parentheses

Arguments and parameters (cont'd)

- parameters provide the means to pass data to a function
  - data passed from the calling function to the function that is called
- the names of the parameters are local to the function
  - they will assume the values of the arguments that are passed when the function is called
- the body of the function should use these parameters in its implementation
- a function body may have additional locally defined variables that are needed by the function's implementation
- when passing an array as an argument to a function
  - you must also pass an additional argument specifying the size of the array
  - the function has no means of knowing how many elements there are in the array

Example

- when the printf() function is called, you always supply one or more values as arguments
  - first value being the format string
  - the remaining values being any variables to displayed
- parameters greatly increase the usefulness and flexibility of a function
  - the printf() function displays whatever you tell it to display via the parameters and arguments passed
- It is a good idea to add comments before each of your own function definitions
  - help explain what the function does and how the arguments are to be used

The return statement

- the return statement provides the means of exiting from a function

return;

- this form of the return statement is used exclusively in a function where the return type has been declared as void
  - does not return a value
- the more general form of the return statement is:

return expression;

- this form of return statement must be used when the return value type for the function has been declared as some type other than void
- the value that is returned to the calling program is the value that results when expression is evaluated
  - should be of the return type specified for the function

Returning data

- a function that has statements in the function body but does not return a value must have the return type as void
  - will get an error message if you compile a program that contains a function with a void return type that tries to return a value
- a function that does not have a void return type must return a value of the specified return type
  - will get an error message from the compiler if return type is different than specified
- If expression results in a value that's a different type from the return type in the function header, the compiler will insert a conversion from the type of expression to the one required
  - If conversion is not possible then the compiler will produce an error message
- there can be more than one return statement in a function
  - each return statement must supply a value that is convertible to the type specified in the function header for the return value

Example

```
int multiplyTwoNumbers (int x, int y)
{
   int retult = x * y;
   return result;
}

int main (void)
{
   int result = 0;
   result = multiplyTwoNumbers (10, 20);

   printf("result is %d\n", result);
   return 0;
}
```

Local Variables

- variables defined inside a function are known as automatic local variables
  - they are automatically “created” each time the function is called
  - their values are local to the function
- the value of a local variable can only be accessed by the function in which the variable is defined
  - its value cannot be accessed by any other function
- if an initial value is given to a variable inside a function, that initial value is assigned to the variable each time the function is called
- can use the auto keyword to be more precise, but, not necessary, as the compiler adds this by default
- local variables are also applicable to any code where the variable is created in a block (loops, if statements)

Global Variables

- the opposite of a local variable
- global variables value can be accessed by any function in the program
- A global variable has the lifetime of the program
- global variables are declared outside of any function
  - does not belong to any particular function
- any function in the program can change the value of a global variable
- if there is a local variable declared in a function with the same name, then, within that function the local variable will mask the global variable
  - global variable is not accessible and prevent it from being accessed normally.

Example

```
int myglobal = 0;       // global variable

int main ()
{
   int myLocalMain = 0;       // local variable
  // can access my global and myLocal
  return 0;
}

void myFunction()
{
  int x;         // local variable
 // can access myGlobal and x, cannot access myLocal
}
```

Requirements

- we need to get some practice writing functions
  - better organized code
  - avoid duplication

- for this challenge you are to write three functions in a single program

- write a function which finds the greatest common divisor of two non-negative integer values and to return the result
  - gcd, takes two ints as parameters

- write a function to calculate the absolute value of a number
  - should take as a parameter a float and return a float
  - test this function with ints and floats

- write a function to compute the square root of a number
  - if a negative argument is passed then a message is display and -1.0 should be returned
  - should use the absoluteValue function as implemented in the above step

Strings

- we have learned all about the char data type
  - contains a single character.

- to assign a single character to a char variable, the character is enclosed within a pair of single quotation marks

```
plusSign = '+';
```

- you have also learned that there is a distinction made between the single quotation and double quotation marks
  - plusSign = "+"; // incorrect if plusSign is a char

- a string constant or string literal is a sequence of characters or symbols between a pair of double-quote characters
  - anything between a pair of double quotes is interpreted by the compiler as a string
  - includes any special characters and embedded spaces

Strings (cont'd)

- every time you have displayed a message using the printf() function, you have defined the message as a string constant

```
printf("This is a string.");
printf("This is on\ntwo lines!");
printf("For \" you write \\\".");
```

- understand the difference between single quotation and double quotation marks
  - both are used to create two different types of constants in C

- for the third example above, you must specify a double quote within a string as the escape sequence \"
  - the compiler will interpret an explicit double quote without a preceding backslash as a string delimiter

- also, you must also use the escape sequence \\ when you want to include a backslash in a string
  - a backslash in a string always signals the start of an escape sequence to the compiler

String in Memory

Null Character

- a special character with the code value 0 is added to the end of each string to mark where it ends
  - this character is known as the null character and you write it as \0
- a string is always terminated by a null character, so the length of a string is always one greater than the number of characters in the string
- don't confuse the null character with NULL
  - null character is a string terminator
  - NULL is a symbol that represents a memory address that doesn't reference anything
- you can add a \0 character to the end of a string explicitly
  - this will create two strings

Null Character Example

```
#include <stdio.h>

int main(void)
{
  printf("The character \0 is used to terminate a string.");
  return 0;
}
```

- If you compile and run this program, you'll get this output:
  - The character
  - only the first part of the string has been displayed
  - output ends after the first two words because the function stops outputting the string when it reaches the first null character
  - the second \0 at the end of the string will never be reached
- The first \0 that's found in a character sequence always marks the end of the string

Character Strings

- C has no special variable type for strings
  - this means there are no special operators in the language for processing strings
  - the standard library provides an extensive range of functions to handle strings

- strings in C are stored in an array of type char
  - characters in a string are stored in adjacent memory cells, one character per cell

- to declare a string in C, simply use the char type and the brackets to indicate the size

```
char myString[20];
```

- this variable can accommodate a string that contains up to 19 characters
  - you must allow one element for the termination character (null character)

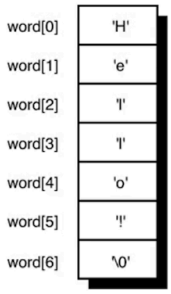
- when you specify the dimension of an array that you intend to use to store a string, it must be at least one greater than the number of characters in the string that you want to store
  - the compiler automatically adds \0 to the end of every string constant

Initializing a String

- You can initialize a string variable when you declare it

```
char  word [] = { 'H', 'e', 'l', 'l', 'o', '!' };
```

- to initialize a string, it is the same as any other array initialization
  - in the absence of a particular array size, the C compiler automatically computes the number of elements in the array
    - based upon the number of initializers
    - this statement reserves space in memory for exactly seven characters
    - automatically adds the null terminator



Taken from Programming in C,

Initializing a String (cont'd)

- you can specify the size of the string explicitly, just make sure you leave enough space for the terminating null character

```
char  word[7] = { "Hello!" };
```

- If the size specified is too small, then the compiler can't fit a terminating null character at the end of the array, and it doesn't put one there (and it doesn't complain about it either)

```
char  word[6] = { "Hello!" };
```

- So….., do not specify the size, let the compiler figure out, you can be sure it will be correct

- you can initialize just part of an array of elements of type char with a string

```
char str[40] = "To be";
```

- the compiler will initialize the first five elements, str[0] to str[4], with the characters of the string constant
  - str[5] will contain the null character, '\0'
  - space is allocated for all 40 elements of the array

Assigning a value to a string after initializing

• since you can not assign arrays in C, you can not assign strings either

• the following is an error:

```
char s[100]; // declare
s = "hello"; // initalize - DOESN'T WORK ('lvalue required' error)
```

• you are performing an assignment operation, and you cannot assign one array of characters to another array of characters like this
  • you have to use strncpy() to assign a value to a char array after it has been declared or initialized

• The below is perfectly valid

```
s[0] = 'h';
s[1] = 'e';
s[2] = 'l';
s[3] = 'l';
s[4] = 'o';
s[5] = '\0';
```

Displaying a string

•when you want to refer to a string stored in an array, you just use the array name by itself

•to display a string as output using the printf function, you do the following

```
printf("\nThe message is: %s", message);
```

•the %s format specifier is for outputting a null-terminated string

•the printf() function assumes when it encounters the %s format characters that the corresponding argument is a character string that is terminated by a null character

Inputting a string

•to input a string via the keyboard, use the scanf function

```
char input[10];
printf(“Please input your name:  ");
scanf(“%s”, input);
```

•the %s format specifier is for inputting string

•no need to use the & (address of operator) on a string

Testing if two strings are equal

- you cannot directly test two strings to see if they are equal with a statement such as

if (string1 == string2)

- the equality operator can only be applied to simple variable types, such as floats, ints, or chars
  - does not work on structures or arrays

- to determine if two strings are equal, you must explicitly compare the two character strings character by character
  - we will discuss as easier way with the strcmp function

- **Reminder::**
  - the string constant "x" is not the same as the character constant 'x'
  - 'x' is a basic type (char)
  - "x" is a derived type, an array of char
  - "x" really consists of two characters, 'x' and '\0', the null character

Example

```
#include <stdio.h>
int main(void)
{
  char str1[] = "To be or not to be";
  char str2[] = ",that is the question";
  unsigned int count = 0;              // Stores the string length

  while (str1[count] != '\0')          // Increment count till we reach the
    ++count;                           // terminating character.

  printf("The length of the string \"%s\" is %d characters.\n", str1, count);

  count = 0;                           // Reset count for next string
  while (str2[count] != '\0')          // Count characters in second string
    ++count;

  printf("The length of the string \"%s\" is %d characters.\n", str2, count);
  return 0;
}
```

Constant Strings

- sometimes you need to use a constant in a program

```
circumference = 3.14159 * diameter;
```

- the constant 3.14159 represents the world-famous constant pi (π)

- there are good reasons to use a symbolic constant instead of just typing in the number
  - a name tells you more than a number does

```
owed = 0.015 * housevalue;
owed = taxrate * housevalue;
```

- If you read through a long program, the meaning of the second version is plainer.

- suppose you have used a constant in several places, and it becomes necessary to change its value
  - you only need to alter the definition of the symbolic constant, rather than find and change every occurrence of the constant in the program

#define

- the preprocessor lets you define constants

```
#define TAXRATE 0.015
```

- when your program is compiled, the value 0.015 will be substituted everywhere you have used TAXRATE
  - compile-time substitution

- a defined name is not a variable
  - you cannot assign a value to it

- notice that the #define statement has a special syntax
  - no equal sign used to assign the value 0.015 to TAXRATE
  - no semicolon

#define (cont'd)

- #define statements can appear anywhere in a program
  - no such thing as a local define
  - most programmers group their #define statements at the beginning of the program (or inside an include file) where they can be quickly referenced and shared by more than one source file

- the #define statement helps to make programs more portable
  - it might be necessary to use constant values that are related to the particular computer on which the program is running

- the #define statement can be used for character and string constants

```
#define BEEP '\a'
#define TEE 'T'
#define ESC '\033'
#define OOPS "Now you have done it!"
```

const

- C90 added a second way to create symbolic constants
  - using the const keyword to convert a declaration for a variable into a declaration for a constant

```
const int MONTHS = 12;    // MONTHS a symbolic constant for 12
```

- const makes MONTHS into a read-only value
  - you can display MONTHS and use it in calculations
  - you cannot alter the value of MONTHS

- const is a newer approach and is more flexible than using #define
  - it lets you declare a type
  - it allows better control over which parts of a program can use the constant

- C has yet a third way to create symbolic constants
  - enums

const (cont'd)

- initializing a char array and declaring it as constant is a good way of handling standard messages

```
const char message[] = "The end of the world is nigh.";
```

- because you declare message as const, it's protected from being modified explicitly within the program
  - any attempt to do so will result in an error message from the compiler

- this technique for defining standard messages is particularly useful if they are used in many places within a program
  - prevents accidental modification of such constants in other parts of the program

String Functions

- you already know that a character string is a char array terminated with a null character (\0)
  - character strings are commonly used
  - C provides many functions specifically designed to work with strings

- some of the more commonly performed operations on character strings include
  - getting the length of a string
    - strlen
  - copying one character string to another
    - strcpy() and strncpy()
  - combining two character strings together (concatenation)
    - strcat() and strncat()
  - determining if two character strings are equal
    - strcmp() and strncmp()

Getting the length of a string

- the strlen() function finds the length of a string
  - returned as a size_t

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

int main(){
    char myString[] = "my string";

    printf("The length of this string is: %d", strlen(myString));

    return 0;
}
```

Copying strings

- since you can not assign arrays in C, you can not assign strings either

```
char s[100]; // declare
s = "hello"; // initalize - DOESN'T WORK ('lvalue required' error)
```

- you can use the strcpy() function to copy a string to an existing string
  - the string equivalent of the assignment operator

```
char src[50], dest[50];

strcpy(src,  "This is source");
strcpy(dest, "This is destination");
```

Copying strings (cont'd)

- the strcpy() function does not check to see whether the source string actually fits in the target string
  - safer way to copy strings is to use strncpy()

- strncpy() takes a third argument
  - the maximum number of characters to copy

```
char src[40];
char dest[12];

memset(dest, '\0', sizeof(dest));
strcpy(src, "Hello how are you doing");
strncpy(dest, src, 10);
```

String concatenation (cont'd)

- the strcat() function does not check to see whether the second string will fit in the first array
  - if you fail to allocate enough space for the first array, you will run into problems as excess characters overflow into adjacent memory locations

- use strncat() instead
  - takes a second argument indicating the maximum number of characters to add

- for example, strncat(bugs, addon, 13) will add the contents of the addon string to bugs, stopping when it reaches 13 additional characters or the null character, whichever comes first

```
char src[50], dest[50];

strcpy(src,  "This is source");
strcpy(dest, "This is destination");

strncat(dest, src, 15);

printf("Final destination string : |%s|", dest);
```

comparing Strings

- suppose you want to compare someone response to a stored string
  - cannot use ==, will only check to see if the string has the same address

- there is a function that compares string contents, not string addresses
  - it is the strcmp() (for string comparison) function
  - does not compare arrays, so it can be used to compare strings stored in arrays of different sizes
  - does not compare characters
    - you can use arguments such as "apples" and "A", but you cannot use character arguments, such as 'A'

- this function does for strings what relational operators do for numbers
  - it returns 0 if its two string arguments are the same and nonzero otherwise
  - if return value < 0 then it indicates str1 is less than str2
  - if return value > 0 then it indicates str2 is less than str1

comparing strings example

```
printf("strcmp(\"A\", \"A\") is ");
printf("%d\n", strcmp("A", "A"));

printf("strcmp(\"A\", \"B\") is ");
printf("%d\n", strcmp("A", "B"));

printf("strcmp(\"B\", \"A\") is ");
printf("%d\n", strcmp("B", "A"));

printf("strcmp(\"C\", \"A\") is ");
printf("%d\n", strcmp("C", "A"));

printf("strcmp(\"Z\", \"a\") is ");
printf("%d\n", strcmp("Z", "a"));

printf("strcmp(\"apples\", \"apple\") is ");
printf("%d\n", strcmp("apples", "apple"));
```

comparing strings (cont'd)

- the strcmp() function compares strings until it finds corresponding characters that differ
  - could take the search to the end of one of the strings
- the strncmp() function compares the strings until they differ or until it has compared a number of characters specified by a third argument
  - if you wanted to search for strings that begin with "astro", you could limit the search to the first five characters

```
if (strncmp("astronomy","astro", 5) == 0)
{
    printf("Found: astronomy");
}

if (strncmp("astounding","astro", 5) == 0)
{
    printf("Found: astounding");
}
```

Overview

- lets discuss some more string functions

- searching a string
  - the string.h header file declares several string-searching functions for finding a single character or a substring
    - strchr() and strstr()

- tokenizing a string
  - a token is a sequence of characters within a string that is bounded by a delimiter (space, comma, period, etc)
  - breaking a sentence into words is called *tokenizing*
  - strtok()

- analyzing strings
  - Islower(), isupper(), isalpha(), isdigit(), etc.

concept of a pointer

- we are going to discuss in detail, the concept of a pointer in an upcoming section
  - however, in order to understand some of these string functions, I want to give you a quick peek on this concept

- C provides a remarkably useful type of variable called a pointer
  - a variable that stores an address
  - its value is the address of another location in memory that can contain a value
  - we have used addresses in the past with the scanf() function

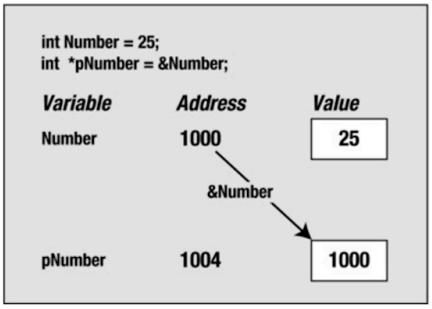
```
int Number = 25;
int *pNumber = &Number;
```

- above, we declared a variable, Number, with the value 25

- we declared a pointer, pNumber, which contains the address of Number
  - asterisk used in declaring a pointer

- to get the value of the variable pNumber, you can use the asterisk to dereference the pointer
  - *pNumber = 25
  - * is the dereference operator, and its effect is to access the data stored at the address specified by a pointer

pointer (cont'd)



(taken from Beginning C, Horton)

- the value of &Number is the address where Number is located
  - this value is used to initialize pNumber in the second statement

- many of the string functions return pointers
  - this is why I wanted to briefly mention them
  - do not worry if this concept does not sink in right now, we are going to cover points in a ton of detail in an upcoming section

Searching a string for a character

- the strchr() function searches a given string for a specified character
  - first argument to the function is the string to be searched (which will be the address of a char array)
  - second argument is the character that you are looking for

- the function will search the string starting at the beginning and return a pointer to the first position in the string where the character is found
  - the address of this position in memory
  - is of type char* described as the “pointer to char.”

- to store the value that’s returned, you must create a variable that can store the address of a character

- if the character is not found, the function returns a special value NULL
  - NULL is the equivalent of 0 for a pointer and represents a pointer that does not point to anything

strchr()

- you can use the strchr() function like this

```
char str[] = "The quick brown fox";    // The string to be searched
char ch = 'q';                         // The character we are looking for
char *pGot_char = NULL;                // Pointer initialized to NULL
pGot_char = strchr(str, ch);           // Stores address where ch is found
```

- the first argument to strchr() is the address of the first location to be searched
  - second argument is the character that is sought (ch, which is of type char)
  - expects its second argument to be of type int, so the compiler will convert the value of ch to this type
  - could just as well define ch as type int (int ch = 'q';)
  - pGot_char will point to the value (“quick brown fox”)

searching for a substring

- the strstr() function is probably the most useful of all the searching functions
  - searches one string for the first occurrence of a substring
  - returns a pointer to the position in the first string where the substring is found
  - if no match, returns NULL

- the first argument is the string that is to be searched
- the second argument is the substring you’re looking for

```
char text[] = "Every dog has his day";
char word[] = "dog";
char *pFound = NULL;
pFound = strstr(text, word);
```

- searches text for the first occurrence of the string stored in word
  - the string "dog" appears starting at the seventh character in text
  - pFound will be set to the address text + 6 (“dog has his day”)
  - search is case sensitive, "Dog“ will not be found

Tokenizing a string

- a token is a sequence of characters within a string that is bound by a delimiter
- a delimiter can be anything, but, should be unique to the string
  - spaces, commas, and a period are good examples
- breaking a sentence into words is called tokenizing
- the strtok() function is used for tokenizing a string
- It requires two arguments
  - string to be tokenized
  - a string containing all the possible delimiter characters

strtok example

```
int main () {
  char str[80] = “Hello how are you – my name is - jason";
  const char s[2] = "-";
  char *token;

  /* get the first token */
  token = strtok(str, s);

  /* walk through other tokens */
  while( token != NULL ) {
    printf( " %s\n", token );

    token = strtok(NULL, s);
  }

  return(0);
}
```

Analyzing strings

Function	Tests for
`islower()`	Lowercase letter
`isupper()`	Uppercase letter
`isalpha()`	Uppercase or lowercase letter
`isalnum()`	Uppercase or lowercase letter or a digit
`iscntrl()`	Control character
`isprint()`	Any printing character including space
`isgraph()`	Any printing character except space
`isdigit()`	Decimal digit (`'0'` to `'9'`)
`isxdigit()`	Hexadecimal digit (`'0'` to `'9'`, `'A'` to `'F'`, `'a'` to `'f'`)
`isblank()`	Standard blank characters (space, `'\t'`)
`isspace()`	Whitespace character (space, `'\n'`, `'\t'`, `'\v'`, `'\r'`, `'\f'`)
`ispunct()`	Printing character for which `isspace()` and `isalnum()` return `false`

- the argument to each of these functions is the character to be tested
- all these functions return a nonzero value of type int if the character is within the set that’s being tested for
- these return values convert to true and false, respectively, so you can use them as Boolean values.

Converting Strings

- it is very common to convert character case
  - to all upper case or all lower case

- the toupper() function converts from lowercase to uppercase
- the tolower() function converts from uppercase to lowercase

- both functions return either the converted character or the same character for characters that are already in the correct case or are not convertible such as punctuation characters

- this is how you convert a string to uppercase

```
for(int i = 0 ; (buf[i] = (char)toupper(buf[i])) != '\0' ; ++i);
```

- this loop will convert the entire string in the buf array to uppercase by stepping through the string one character at a ti
  - loop stops when it reaches the string termination character '\0'
  - the cast to type char is there because toupper() returns type int

Case conversion example

```
char text[100];          // Input buffer for string to be searched
char substring[40];      // Input buffer for string sought

printf("Enter the string to be searched (less than %d characters):\n", 100);
scanf("%s", text);

printf("\nEnter the string sought (less than %d characters):\n", 40);
scanf("%s", substring);

printf("\nFirst string entered:\n%s\n", text);
printf("Second string entered:\n%s\n", substring);

 // Convert both strings to uppercase.
 for(i = 0 ; (text[i] = (char)toupper(text[i])) != '\0' ; ++i);
 for(i = 0 ; (substring[i] = (char)toupper(substring[i])) != '\0' ; ++i);

 printf("The second string %s found in the first.\n", ((strstr(text, substring) == NULL) ? "was not" : "was"));
```

Converting Strings to numbers

the stdlib.h header file declares functions that you can use to convert a string to a numerical value

Function	Returns
`atof()`	A value of type `double` that is produced from the string argument. Infinity as a `double` value is recognized from the strings `"INF"` or `"INFINITY"` where any character can be in uppercase or lowercase and 'not a number' is recognized from the string `"NAN"` in uppercase or lowercase.
`atoi()`	A value of type `int` that is produced from the string argument.
`atol()`	A value of type `long` that is produced from the string argument.
`atoll()`	A value of type `long long` that is produced from the string argument.

Taken from Beginning C, Horton

For all four functions, leading whitespace is ignored

```
char value_str[] = "98.4";
double value = atof(value_str);
```

Converting Strings to Numbers

Function	Returns
`strtod()`	A value of type `double` is produced from the initial part of the string specified by the first argument. The second argument is a pointer to a variable, `ptr` say, of type `char*` in which the function will store the address of the first character following the substring that was converted to the `double` value. If no string was found that could be converted to type `double`, the variable `ptr` will contain the address passed as the first argument.
`strtof()`	A value of type `float`. In all other respects it works as `strtod()`.
`strtold()`	A value of type `long double`. In all other respects it works as `strtod()`.

Taken from Beginning C, Horton

Example

```
double value = 0;
char str[] = "3.5 2.5 1.26";     // The string to be converted
char *pstr = str;                // Pointer to the string to be converted
char *ptr = NULL;                // Pointer to character position after conversion

while(true)
{
  value = strtod(pstr, &ptr);    // Convert starting at pstr
  if(pstr == ptr)                // pstr stored if no conversion...
    break;                       // ...so we are done
  else
  {
    printf("  %f", value);       // Output the resultant value
    pstr = ptr;                  // Store start for next conversion
  }
}
```

Requirements

- In this challenge, you are going to write a program that tests your understanding of char arrays

- write a function to count the number of characters in a string (length)
  - cannot use the strlen library function
  - function should take a character array as a parameter
  - should return an int (the length)

- write a function to concatenate two character strings
  - cannot use the strcat library function
  - function should take 3 parameters
    - char result[]
    - const char str1[]
    - const char str2[]
    - can return void

- write a function that determines if two strings are equal
  - cannot use strcmp library function
  - function should take two const char arrays as parameters and return a Boolean of true if they are equal and false otherwise

Overview

- debugging is the process of finding and fixing errors in a program (usually logic errors, but, can also include compiler/syntax errors)
  - for syntax errors, understand what the compiler is telling you
  - always focus on fixing the first problem detected

- can range in complexity from fixing simple errors to collecting large amounts of data for analysis

- the ability to debug by a programmer is an essential skill (problem solving) that can save you tremendous amounts of time (and money)

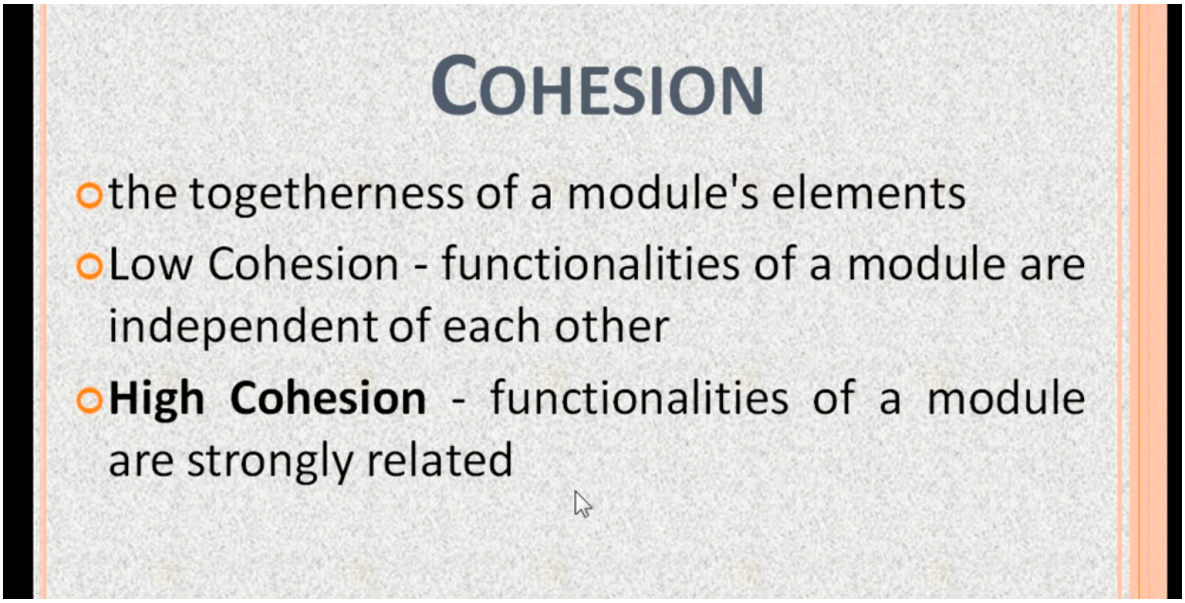
- maintenance phase is the most expensive phase of the software life cycle

- understand that bugs are unavoidable

Common Problems

- Logic Errors
- Syntax Errors
- Memory Corruption
- Performance / Scalability
- Lack of Cohesion
- Tight Coupling (dependencies)

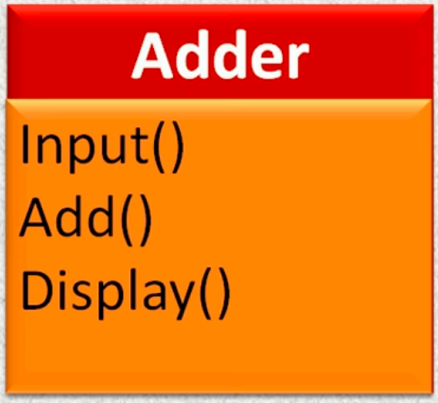
High Cohesion and low coupling



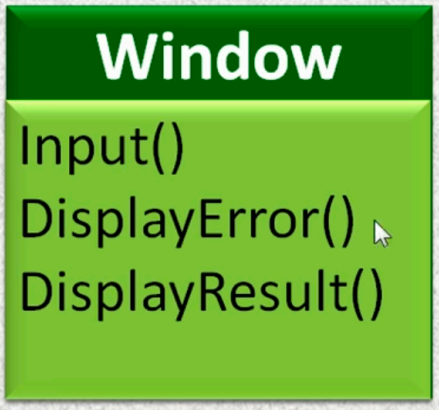
Cohesion

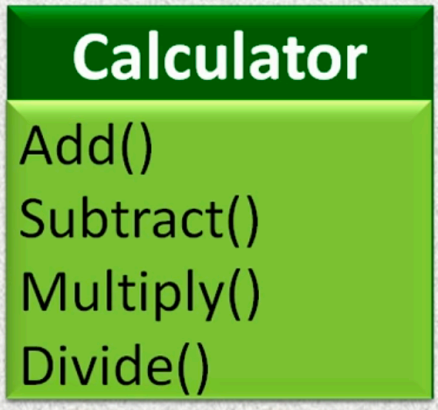
- Cohesion indicates level of logical connection of methods and attributes used in a class.
- In good software design, Classes should be highly cohesive.

Low Cohesion



High Cohesion





COUPLING

- dependence of a module on other modules
- Tight coupling (high coupling) - a module has many relationships with many other modules
- **Loose coupling** (low coupling) - a module has fewer relationships with other modules

Coupling

- Coupling indicates level of dependency between classes.
- In good software design classes should be loosely coupled to each other.

Debugging Process

- Understand the problem (sit down with tester, understand requirements)
- Reproduce the problem
  - Sometimes very difficult as problems can be intermittent or only happen in very rare circumstances
    - Parallel processes or threading problems
- Simplify the problem / Divide and conquer / isolate the source
  - Remove parts of the original test case
  - Comment out code / back out changes
  - Turn a large program into a lot of small programs (unit testing)

Debugging Process (cont'd)

- Identify origin of the problem (in the code)
  - Use Debugging Tools if necessary
- Solve the problem
  - Experience and practice
  - Sometimes includes redesign or refactor of code
- Test!, Test!, Test!

Techniques and Tools

- Tracing / using print statements
  - Output values of variables at certain points of a program
  - Show the flow of execution
  - Can help isolate the error
- Debuggers – monitor the execution of a program, stop it, restart it, set breakpoints and watch variables in memory
- Log Files – can be used for analysis, add "good" log statements to your code
- Monitoring Software – run-time analysis of memory usage, network traffic, thread and object information

Common Debugging Tools

- Exception Handling helps a great deal to identify catastrophic errors
- Static Analyzers – analyze source code for specific set of known problems
  - Semantic checker, does not analyze syntax
  - Can detect things like uninitialized variables, memory leaks, unreachable code, deadlocks or race conditions
- Test Suites – run a set of comprehensive system end-to-end tests
- Debugging the program after it has crashed
  - Analyze the call stack
  - Analyze memory dump (core file)

Preventing Errors

- write high quality code (follow good design principles and good programming practices)
- Unit Tests – automatically executed when compiling
  - Helps avoid regression
  - Finds errors in new code before it is delivered
  - TDD (Test Driven Development)
- Provide good documentation and proper planning (write down design on paper and utilize pseudocode)
- Work in Steps and constantly test after each step
  - Avoid too many changes at once
  - When making changes, apply them incrementally. Add one change, then test thoroughly before starting the next step
  - Helps reduce the possible sources of bugs, limits problem set

Overview

- A stack trace (call stack) is generated whenever your app crashes because of a fatal error
- A stack trace shows a list of the function calls that lead to the error
  - Includes the filenames and line numbers of the code that cause the exception or error to occur
  - Top of the stack contains the last call that caused the error (nested calls)
  - Bottom of the stack contains the first call that started the chain of calls to cause the error
  - You need to find the call in your application that is causing the crash

Common C Mistakes

- misplacing a semicolon

```
if ( j == 100 );
    j = 0;
```

- the value of j will always be set to 0 due to the misplaced semicolon after the closing parenthesis
  - semicolon is syntactically valid (it represents the null statement), and, therefore, no error is produced by the compiler
  - same type of mistake is frequently made in while and for loops
- confusing the operator = with the operator ==
  - usually made inside an if, while, or do statement
  - perfectly valid and has the effect of assigning 2 to a and then executing the printf() call
  - printf() function will always be called because the value of the expression contained in the if statement will always be nonzero

```
if ( a = 2 )
  printf ("Your turn.\n");
```

Common C Mistakes

- omitting prototype declarations

```
result = squareRoot (2);
```

- If squareRoot is defined later in the program, or in another file, and is not explicitly declared otherwise
  - compiler assumes that the function returns an int
  - always safest to include a prototype declaration for all functions that you call (either explicitly yourself or implicitly by including the correct header file in your program)

- failing to include the header file that includes the definition for a C-programming library function being used in the program

```
double answer = sqrt(value1);
```

- if this program does not #include the <math.h> file, this will generate an error that sqrt() is undefined

Common C Mistakes

- using the wrong bounds for an array

```
int  a[100], i, sum = 0;
  ...
for ( i = 1;  i <= 100;  ++i )
  sum += a[i];
```

- valid subscripts of an array range from 0 through the number of elements minus one
  - the preceding loop is incorrect because the last valid subscript of a is 99 and not 100
  - also probably intended to start with the first element of the array; therefore, i should have been initially set to 0

- forgetting to reserve an extra location in an array for the terminating null character of a string
  - when declaring character arrays they need to be large enough to contain the terminating null character
  - the character string "hello" would require six locations in a character array if you wanted to store a null at the end

Common C Mistakes

- confusing the operator → with the operator . when referencing structure members.
  - the operator . is used for structure variables
  - the operator → is used for structure pointer variables

- omitting the ampersand before nonpointer variables in a scanf() call

```
int  number;
  ...
scanf ("%i", number);
```

- all arguments appearing after the format string in a scanf() call must be pointers

Common C Mistakes

- using a pointer variable before it's initialized

```
char  *char_pointer;
*char_pointer = 'X';
```

- you can only apply the indirection operator to a pointer variable after you have set the variable pointing somewhere
  - char_pointer is never set pointing to anything, so the assignment is not meaningful

- omitting the break statement at the end of a case in a switch statement
  - if a break is not included at the end of a case, then execution continues into the next case

Common C Mistakes

- inserting a semicolon at the end of a preprocessor definition
  - usually happens because it becomes a matter of habit to end all statements with semicolons

```
#define    END_OF_DATA   999;
```

- leads to a syntax error if used in an expression such as

```
if ( value == END_OF_DATA )
  ...
```

- the compiler will see this statement after preprocessing

```
if ( value == 999; )
  ...
```

Common C Mistakes

- omitting a closing parenthesis or closing quotation marks on any statement

```
total_earning = (cash + (investments * inv_interest) + (savings * sav_interest);
printf("Your total money to date is %.2f, total_earning);
```

- the use of embedded parentheses to set apart each portion of the equation makes for a more readable line of code
  - however, there is always the possibility of missing a closing parenthesis (or in some occasions, adding one too many)

- the second line is missing a closing quotation mark for the string being sent to the printf() function

- both of these will generate a compiler error
  - sometimes the error will be identified as coming on a different line
  - depending on whether the compiler uses a parenthesis or quotation mark on a subsequent line to complete the expression which moves the missing character to a place later in the program

Overview

- it is sometimes very hard to understand what the compiler is complaining about
  - need to understand compiler errors in order to fix them
  - it is sometimes difficult to identify the true reason behind a compiler error

- the compiler makes decisions about how to translate the code that the programmer has not written in the code
  - is convenient because the programs can be written more succinctly (only expert programmers take advantage of this feature)

- you should use an option for the compiler to notify all cases where there are implicit decisions
  - this option is -Wall

Overview

- the compiler shows two types of problems
  - errors
    - a condition that prevents the creation of a final program
    - no executable is obtained until all the errors have been corrected
    - The first errors shown are the most reliable because the translation is finished but there are some errors that may derive from previous ones
    - Fix the first errors are first, it is recommended to compile again and see if other later errors also disappeared.

  - warnings
    - messages that the compiler shows about “special” situations in which an anomaly has been detected
    - non-fatal errors
    - the final executable program may be obtained with any number of warning

- compile always with the -Wall option and do not consider the program correct until all warnings have been eliminated

most common compiler messages

- `variable' undeclared (first use in this function)
  - this is one of the most common and easier to detect
  - the symbol shown at the beginning of the message is used but has not been declared

- warning: implicit declaration of function `...'
  - this warning appears when the compiler finds a function used in the code but no previous information has been given about it
  - need to declare a function prototype

- warning: control reaches end of non-void function
  - this warning appears when a function has been defined as returning a result but no return statement has been included to return this result
  - either the function is incorrectly defined or the statement is missing

most common compiler messages

- warning: unused variable `...'
  - this warning is printed by the compiler when a variable is declared but not used in the code
  - message disappears if the declaration is removed

- undefined reference to `...'
  - appears when there is a function invoked in the code that has not been defined anywhere
  - compiler is telling us that there is a reference to a function with no definition
  - check which function is missing and make sure its definition is compiled

- error: conflicting types for `...'
  - two definitions of a function prototype have been found
  - one is the prototype (the result type, name, parenthesis including the parameters, and a semicolon)
  - the other is the definition with the function body
  - the information in both places is not identical, and a conflict has been detected
  - the compiler shows you in which line the conflict appears and the previous definition that caused the contradiction

Indirection

- pointers are very similar to the concept of indirection that you employ in your everyday life
  - suppose you need to buy a new ink cartridge for your printer
  - all purchases are handled by the purchasing department
    - you call Joe in purchasing and ask him to order the new cartridge for you
    - Joe then calls the local supply store to order the cartridge
  - you are not ordering the cartridge directly from the supply store yourself (indirection)

- in programming languages, indirection is the ability to reference something using a name, reference, or container, instead of the value itself

- the most common form of indirection is the act of manipulating a value through its memory address

- a pointer provides an indirect means of accessing the value of a particular data item
  - a variable whose value is a memory address
  - its value is the address of another location in memory that can contain a value

Overview

- just as there are reasons why it makes sense to go through the purchasing department to order new cartridges (you don't have to know which particular store the cartridges are being ordered from)
  - there are good reasons why it makes sense to use pointers in C

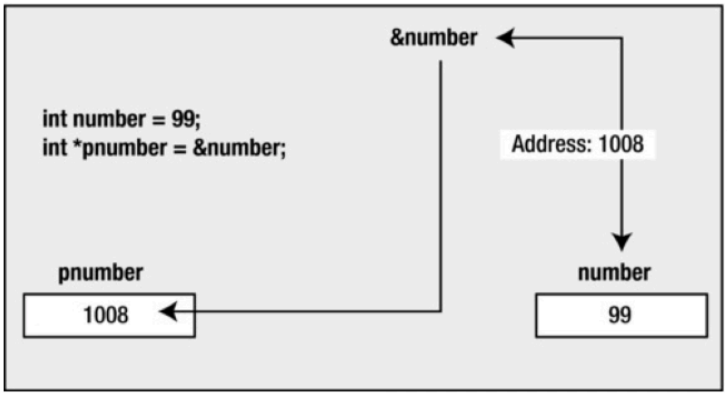
- using pointers in your program is one of the most powerful tools available in the C language

- pointers are also one of the most confusing concepts of the C language
  - it is important you get this concept figured out in the beginning and maintain a clear idea of what is happening as you dig deeper

- the compiler must know the type of data stored in the variable to which it points
  - need to know how much memory is occupied or how to handle the contents of the memory to which it points
  - every pointer will be associated with a specific variable type
  - it can be used only to point to variables of that type

- pointers of type "pointer to int" can point only to variables of type int

Overview (cont'd)



(taken from Beginning C, Horton)

- the value of &number is the address where number is located
  - this value is used to initialize pnumber in the second statement

Why use pointers?

- accessing data by means of only variables is very limiting
  - with pointers, you can access any location (you can treat any position of memory as a variable for example) and perform arithmetic with pointers

- pointers in C make it easier to use arrays and strings

- pointers allow you to refer to the same space in memory from multiple locations
  - means that you can update memory in one location and the change can be seen from another location in your program
  - can also save space by being able to share components in your data structures

- pointers allow functions to modify data passed to them as variables
  - pass by reference - passing arguments to function in way they can be changed by function

- can also be used to optimize a program to run faster or use less memory than it would otherwise

Why use pointers?

- pointers allow us to get multiple values from the function
  - a function can return only one value but by passing arguments as pointers we can get more than one values from the pointer

- with pointers dynamic memory can be created according to the program use
  - we can save memory from static (compile time) declarations

- pointers allow us to design and develop complex data structures like a stack, queue, or linked list

- pointers provide direct memory access.

Declaring pointers

- pointers are not declared like normal variables

```
pointer ptr;    // not the way to declare a pointer/
```

- it is not enough to say that a variable is a pointer
  - you also have to specify the kind of variable to which the pointer points
  - different variable types take up different amounts of storage
  - some pointer operations require knowledge of that storage size

- you declare a pointer to a variable of type int with:

```
int *pnumber;
```

- the type of the variable with the name pnumber is int*
  - can store the address of any variable of type int

```
int * pi;       // pi is a pointer to an integer variable
char * pc;      // pc is a pointer to a character variable
float * pf, * pg; // pf, pg are pointers to float variables
```

Declaring pointers (cont'd)

- the space between the * and the pointer name is optional
  - programmers use the space in a declaration and omit it when dereferencing a variable

- the value of a pointer is an address, and it is represented internally as an unsigned integer on most systems
  - however, you shouldn't think of a pointer as an integer type
  - things you can do with integers that you can not do with pointers, and vice versa
  - you can multiply one integer by another, but you can not multiply one pointer by another

- a pointer really is a new type, not an integer type
  - %p represents the format specifier for pointers

- the previous declarations creates the variable but does not initialize it
  - dangerous when not initialized
  - you should always initialize a pointer when you declare it

NULL Pointers

- you can initialize a pointer so that it does not point to anything:

```
int *pnumber = NULL;
```

- NULL is a constant that is defined in the standard library
  - is the equivalent of zero for a pointer

- NULL is a value that is guaranteed not to point to any location in memory
  - means that it implicitly prevents the accidental overwriting of memory by using a pointer that does not point to anything specific

- add an #include directive for stddef.h to your source file

Address of operator

- if you want to initialize your variable with the address of a variable you have already declared
  - use the address of operator, &

```
int number = 99;
int *pnumber = &number;
```

- the initial value of pnumber is the address of the variable number
  - the declaration of number must precede the declaration of the pointer that stores its address
  - compiler must have already allocated space and thus an address for number to use it to initialize pnumber

---

Be careful

- there is nothing special about the declaration of a pointer
  - can declare regular variables and pointers in the same statement

```
double value, *pVal, fnum;
```

- only the second variable, pVal, is a pointer

```
int *p, q;
```

- the above declares a pointer, p of type int*, and a variable, q, that is of type int
  - a common mistake to think that both p and q are pointers

- also, it is a good idea to use names beginning with p as pointer names

---

Accessing pointer values

- you use the indirection operator, *, to access the value of the variable pointed to by a pointer
  - also referred to as the dereference operator because you use it to "dereference" a pointer

```
int number = 15;
int *pointer = &number;
int result = 0;
```

- the pointer variable contains the address of the variable number
  - you can use this in an expression to calculate a new value for result

```
result = *pointer + 5;
```

- the expression *pointer will evaluate to the value stored at the address contained in the pointer
  - the value stored in number, 15, so result will be set to 15 + 5, which is 20

- the indirection operator, *, is also the symbol for multiplication, and it is used to specify pointer types
  - depending on where the asterisk appears, the compiler will understand whether it should interpret it as an indirection operator, as a multiplication sign, or as part of a type specification
  - context determines what it means in any instance

Example

```
int main (void)
{
    int   count = 10, x;
    int   *int_pointer;

    int_pointer = &count;
    x = *int_pointer;

    printf ("count = %i, x = %i\n", count, x);

    return 0;
}
```

Displaying a pointers value

- to output the address of a variable, you use the output format specifier %p
  - outputs a pointer value as a memory address in hexadecimal form

```
int number = 0;             // A variable of type int initialized to 0
int *pnumber = NULL;        // A pointer that can point to type int

number = 10;
pnumber = &number;
printf("pnumber's value: %p\n", pnumber);          // Output the value (an address)
```

- pointers occupy 8 bytes and the addresses have 16 hexadecimal digits
  - if a machine has a 64-bit operating system and my compiler supports 64-bit addresses
  - some compilers only support 32-bit addressing, in which case addresses will be 32-bit addresses

Displaying an address (cont'd)

```
printf("number's address: %p\n", &number);            // Output the address
printf("pnumber's address: %p\n", (void*)&pnumber);   // Output the address
```

- remember, a pointer itself has an address, just like any other variable
  - you use %p as the conversion specifier to display an address
  - you use the & (address of) operator to reference the address that the pnumber variable occupies
  - the cast to void* is to prevent a possible warning from the compiler
    - the %p specification expects the value to be some kind of pointer type, but the type of &pnumber is "pointer to pointer to int"

Displaying the number of bytes a pointer is using

- you use the sizeof operator to obtain the number of bytes a pointer occupies
  - on my machine this shows that a pointer occupies 8 bytes
  - a memory address on my machine is 64 bits
- you may get a compiler warning when using sizeof this way
  - size_t is an implementation-defined integer type
  - to prevent the warning, you could cast the argument to type int like this:

```
printf("pnumber's size: %d bytes\n", (int)sizeof(pnumber)); // Output the size
```

Overview

- C offers several basic operations you can perform on pointers
- you can assign an address to a pointer
  - assigned value can be an array name, a variable preceded by address operator (&), or another second pointer.
- you can dereference a pointer
  - the * operator gives the value stored in the pointed-to location
- you can take a pointer address
  - the & operator tells you where the pointer itself is stored
- you can perform pointer arithmetic
  - use the + operator to add an integer to a pointer or a pointer to an integer (integer is multiplied by the number of bytes in the pointed-to type and added to the original address)
  - Increment a pointer by one (useful in arrays when moving to the next element)
  - use the - operator to subtract an integer from a pointer (integer is multiplied by the number of bytes in the pointed-to type and subtracted from the original address
  - decrementing a pointer by one (useful in arrays when going back to the previous element)

Overview

- you can find the difference between two pointers
  - you do this for two pointers to elements that are in the same array to find out how far apart the elements are
- you can use the relational operators to compare the values of two pointers
  - pointers must be the same type
- remember, there are two forms of subtraction
  - you can subtract one pointer from another to get an integer
  - you can subtract an integer from a pointer and get a pointer
- be careful when incrementing or decrementing pointers and causing an array "out of bounds" error
  - computer does not keep track of whether a pointer still points to an array element

pointers used in expressions

- the value referenced by a pointer can be used in an arithmetic expressions
  - if a variable is defined to be of type "pointer to integer" then it is evaluated using the rules of integer arithmetic

```
int number = 0;             // A variable of type int initialized to 0
int *pnumber = NULL;        // A pointer that can point to type int
number = 10;
pnumber = &number;          // Store the address of number in pnumber
*pnumber += 25;
```

- increments the value of the number variable by 25
- * indicates you are accessing the contents to which the variable called pnumber is pointing to

- if a pointer points to a variable x
  - that pointer has been defined to be a pointer to the same data type as is x
  - use of *pointer in an expression is identical to the use of x in the same expression