The structure of C programs

Let's looks at the high-level structure of a short C program, \texttt{rot.c} (using ellipsis to omit some statements for now). At this stage it's not important what the program is supposed to do.

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

/* Compile this program as:
   gcc -std=c99 -Wall -Werror -pedantic -o rot rot.c */

#define ROT 13

static char rotate(char c)
{
    ....
    return c;
}

int main(int argc, char *argv[])
{
    // check the number of arguments
    if(argc != 2) {
        ....
        exit(EXIT_FAILURE);
    }
    else {
        ....
        exit(EXIT_SUCCESS);
    }
    return 0;
}
```

Of note in this example:

- Characters such as a space, tab, or newline, may appear almost anywhere - they are stripped out and ignored by the C compiler.

We use such \textit{whitespace} characters to provide a layout to our programs. While the exact layout is not important, using a consistent layout is very good practice.

- Keywords, in \textbf{bold}, mean very specific things to the C compiler.

- Lines commencing with a ‘#’ in \textcolor{blue}{blue} are processed by a separate program, named the C \textit{preprocessor}.

In practice, our program is provided as input to the preprocessor, and the preprocessor’s output is given to the C compiler.

- Lines in \textcolor{green}{green} are comments. They are ignored by the C compiler, and may contain (almost) any characters.

C99 provides two types of comments -
1. /* block comments */ and
2. // comments to the end of a line
The structure of C programs, continued

More to note:

- A variety of brackets are employed, in pairs, to group together items to be considered in the same way. Here:
  - angle brackets enclose a filename in a `#include` directive,
  - round brackets group items in arithmetic expressions and function calls,
  - square brackets enclose the index when access arrays (vectors and matrices...) of data, and
  - curly brackets group together sequences of one or more statements in C. We term a group of statements a block of statements.

- Functions in C, may be thought of as a block of statements to which we give a name. In our example, we have two functions - `rotate` and `main()`.

- When our programs are run by the operating system, the operating system always starts our program from `main()`. Thus, every complete C program requires a `main()` function.

The operating system passes some special information to our `main()` function, command-line arguments, and `main()` needs a special syntax to receive these.

- When our program finishes its execution, it returns some information to the operating system. Our example here exits by announcing either its failure or success.

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

#define ROT 13

static char rotate(char c)
{
    ....
    return c;
}

int main(int argc, char *argv[])
{
    // check the number of arguments
    if(argc != 2) {
        ....
        exit(EXIT_FAILURE);
    }
    else {
        ....
        exit(EXIT_SUCCESS);
    }
    return 0;
}
```
Compiling and linking our C programs

C programs are human-readable text files, that we term source-code files.

This makes them very easy to copy, read, and edit on different computers and different operating systems. C is often described as being portable at the source-code level.

Before we can run (execute) our C programs, we must translate, or compile, their source-code files to files that the operating system can better manage.

A program known as a compiler translates (compiles) source-code files into object-code files.

Finally, we translate or link one or more object-code files to produce an executable program, often termed a 'binary' or an 'exe' file. A program known as a linker performs this translation, also linking our object-code file(s) with standard libraries and (optionally) 3rd-party libraries.

Depending on how we invoke the compiler, sometimes we can 'move' straight from the source-code files to the executable program, all in one step.

In reality the compiler is 'silently' executing the linker program for us, and then removing any unwanted object-files.
Variables

Variables are locations in a computer's memory. A typical desktop or laptop computer will have 4GB of memory, or four billion addressable memory locations.

A typical C program will use 4 bytes to hold a single integer value, or 8 bytes to hold a single floating-point value.

Any variable can only hold a single value at any time - they do not maintain a history of past values they once had.

Naming our variables

To make programs more readable, we provide variables with simple names. We should carefully choose names to reflect the role of the variable in our programs.

- While variable names can be almost anything (but not the same as the keywords in C) there's a simple restriction on the permitted characters in a name -
  - they must commence with an alphabetic or the underscore character (_ A-Z a-z), and
  - be followed by zero or more alphabetic, underscore or digit characters (_ A-Z a-z 0-9).

- C variable names are case sensitive, thus:

  MYLIMIT, mylimit, Mylimit and MyLimit

  are four different variable names.

- Older C compilers may limit variable names to, say, 8 unique characters. Thus, for them,

  turn_reactor_coolant_on and turn_reactor_coolant_off

  are the same variable! Keep this in mind when writing portable code.

- While not required, it's preferred that variable names do not consist entirely of uppercase characters. We'll consistently use uppercase-only names for constants provided by the C preprocessor, or user-defined type names:

  MAXLENGTH, AVATAR, BUFSIZE, and ROT
Basic datatypes

Variables are declared to be of a certain data type, or just type.

We use different types to represent the permissible values that a program’s variable has.

For example, if we’re using a variable to just count things, we’ll use an integer variable to hold the count; if performing trigonometry on angles expressed in radians, we’ll use floating-point variables to hold values with both an integral and a fractional part.

C provides a number of standard, or base types to hold commonly required values, and later we’ll see how we can also define our own user-defined types to meet our needs.

Let’s look quickly at some of C’s base datatypes:

<table>
<thead>
<tr>
<th>typename</th>
<th>description, and an example of variable initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Boolean (truth values), which may only hold the values of either true or false</td>
</tr>
<tr>
<td>char</td>
<td>character values, to each hold a single values such as an alphabetic character, a digit character, a space, a tab...</td>
</tr>
<tr>
<td>int</td>
<td>integer values, negative, positive, and zero</td>
</tr>
<tr>
<td>float</td>
<td>floating point values, with a typical precision of 10 decimal digits (on our lab machines)</td>
</tr>
<tr>
<td>double</td>
<td>&quot;bigger&quot; floating point values, with a typical precision of 17 decimal digits (on our lab machines)</td>
</tr>
</tbody>
</table>

Some textbooks will (too quickly) discuss the actual storage size of these basic types, and discuss the ranges of permissible values. We’ll examine these later, but for now we’ll focus on using these basic types in their most obvious ways.

From where does the bool datatype get its name? - the 19th century mathematician and philosopher, George Boole.
The scope of variables

The scope of a variable describes the range of lines in which the variable may be used. Some textbooks may also term this the visibility or lexical range of a variable.

C has only 2 primary types of scope:

- **global scope** (sometimes termed file scope) in which variables are declared outside of all functions and statement blocks, and

- **block scope** in which variables are declared within a function or statement block.

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

static int count = 0;

int main(int argc, char *argv[])
{
    int nfound = 0;

    // check the number of arguments
    if(argc != 2) {
        int nerrors = 1;

        ....
        exit(EXIT_FAILURE);

    } else {
        int ntimes = 100;

        ....
        exit(EXIT_SUCCESS);

    } 

    return 0;
}
```

The variable `count` has **global scope**.

It is defined on line 6, and may be used anywhere from line 6 until the end of the file (until line 26).

The variable `nfound` has **block scope**.

It is defined on line 10, and may be used anywhere from line 10 until the end of the block in which it was defined (until line 26).

The variable `nerrors` has **block scope**.

It is defined on line 14, and may be used anywhere from line 14 until line 18.

The variable `ntimes` has **block scope**.

It is defined on line 20, and may be used anywhere from line 20 until line 24.

We could define a different variable named `nerrors` in the block of lines 20-24 - without problems.

We could define a different variable named `nfound` in the block of lines 20-24 - but this would be a very bad practice!
Flow of control in a C program

A program’s *control flow* describes how sequences of statements are executed.

- As we’ve seen, C programs commence their execution at their `main()` function, execute their statements, and exit (return the flow of control) to the operating system.

- It’s fairly obvious that statements *need* to be executed in a well-defined order, as we expect programs to always behave the same way (unless some randomness is introduced, as in computer games).

- Default flow of control executes each statement in order, top-to-bottom.

- Programs that only execute from top-to-bottom are pretty boring, and we need to control their flow with a variety of *conditional* statements and *loops*.

---

Conditional execution

Conditional statements first evaluate a Boolean condition and then, based on whether it’s true or false, execute other statements.

The most common form is:

```c
if(condition)
{
    // more statements;
    ..... 
}
else
{
    // more statements;
    ..... 
}
```

Sometimes, the else clause is omitted:

```c
if(condition)
{
    // more statements;
    ..... 
}
```

Often, the else clause provides further if statements:

```c
if(condition1)
{
    // more statements;
    ..... 
}
else if(condition2)
{
    // more statements;
    ..... 
}
else
{
    // more statements;
    ..... 
}
```

Of significance, and a very common cause of errors in C programs, is that pre ISO-C99 has no Boolean datatype. Whenever requiring the true and false constants (introduced in C99), we need to provide the line:

```c
#include <stdbool.h>
```
Flow of control in a C program - bounded loops

One of the most powerful features of computers, in general, is to perform thousands, millions, of repetitive tasks quickly.

(in fact, one of the first uses of computers in the 1940s was to calculate trigonometric tables for the firing of artillery shells).

C provides its for control statement to loop through a sequence of statements, a block of statements, a known number of times:

The most common form appears below, in which we introduce a loop control variable, i, to count how many times we go through the loop:

```c
// here, variable i holds the values 1,2,...10
for(int i = 1 ; i <= 10 ; i = i+1)
{
    // the above introduced a loop-control variable, i
    ....
    printf("loop number %i\n", i);
    ....
    // variable i is available down to here
}
// but variable i is not available from here
```

The loop control variable does not always have to be an integer:

```c
// here, variable ch holds each lowercase value
for(char ch = 'a' ; ch <= 'z' ; ch = ch+1)
{
    ....
    printf("loop using character '%c'\n", ch);
    ....
}
```

Notice that in both cases, above, we have introduced new variables, here i and ch, to specifically control the loop.

The variables may be used inside each loop, in the statement block, but then "disappear" once the block is finished (after its bottom curly bracket).

It's also possible to use any other variable as the loop control variable, even if defined outside of the for loop. In general, we'll try to avoid this practice - unless the value of the variable is required outside of the loop.
Flow of control in a C program - unbounded loops

The for loops that we've just seen should be used when we know, ahead of time, how many times we need to loop (i.e. 10 times, or over the range 'a'..'z').

Such loops are termed bounded loops and, unless we've made a silly coding error will always terminate after a fixed number of iterations.

There are also many occasions when we don't know, ahead of time, how many iterations may be required. Such occasions require unbounded loops.

C provides two types of unbounded loop:

The most common is the while loop, where zero or more iterations are made through the loop:

```c
#define NLOOPS 20
int i = 1;
int n = 0;
......
while(i <= NLOOPS)
{
    printf("iteration number %i\n", i);
    ......
    i = some_calculation_setting_i;
    n = n + 1;
}
printf("loop was traversed %i times\n", n);
```

Less common is the do....while loop, where at least one iteration is made through the loop:

```c
#define NLOOPS 20
int i = 1;
int n = 0;
......
do
{
    printf("iteration number %i\n", i);
    ......
    i = some_calculation_setting_i;
    n = n + 1;
} while(i <= NLOOPS);
printf("loop was traversed %i times\n", n);
```

Notice that in both cases we still use a variable, i, to control the number of iterations of each loop, and that the changing value of the variable is used to determine if the loop should "keep going".

However, the statements used to modify the control variable may appear almost anywhere in the loops. They provide flexibility, but can also be confusing when loops become several tens or hundreds of lines long.

Notice also that while, and do....while loops cannot introduce new variables to control their iterations, and so we have to use "more global" variables.
Writing loops within loops

There’s a number of occasions when we wish to loop a number of times (and so we use a `for` loop) and within that loop we wish to perform another loop. While a little confusing, this construct is often quite common. It is termed a *nested loop*.

```c
#define NROWS 6
#define NCOLS 4

for(int row = 1 ; row <= NROWS ; row = row+1) // the 'outer' loop
{
    for(int col = 1 ; col <= NCOLS ; col = col+1) // the 'inner' loop
    {
        printf("(%i,%i)  ", row, col); // print row and col as if "coordinates"
    }
    printf("\n"); // finish printing on this line
}
```

The resulting output will be:

```
(1,1)  (1,2)  (1,3)  (1,4)
(2,1)  (2,2)  (2,3)  (2,4)
(3,1)  (3,2)  (3,3)  (3,4)
(4,1)  (4,2)  (4,3)  (4,4)
(5,1)  (5,2)  (5,3)  (5,4)
(6,1)  (6,2)  (6,3)  (6,4)
```

Notice that we have two distinct loop-control variables, `row` and `col`.

Each time that the *inner loop* (`col`’s loop) starts, `col`’s value is initialized to 1, and advances to 4 (NCOLS).

As programs become more complex, we will see the need for, and write, all combinations of:

- `for` loops within `for` loops,
- `while` loops within `while` loops,
- `for` loops within `while` loops,
- and so on....
Changing the regular flow of control within loops

There are many occasions when the default flow of control in loops needs to be modified.

Sometimes we need to leave a loop early, using the `break` statement, possibly skipping some iterations and some statements:

```c
for(int i = 1; i <= 10; i = i+1)
{
    // Read an input character from the keyboard
    ....
    if(input_char == 'Q') // Should we quit?
        break;
    ....
    ....
} // Come here after the 'break'. i is unavailable
```

In the first example, we iterate through the loop at most 10 times, each time reading a line of input from the keyboard. If the user indicates they wish to quit, we `break` out of the bounded loop.

Sometimes we need to start the next iteration of a loop, even before executing all statements in the loop:

```c
for(char ch = 'a'; ch <= 'z'; ch = ch+1)
{
    if(ch == 'm') // skip over the character 'm'
        continue;
    ....
    ....
    statements that will never see ch == 'm'
    ....
} // Come here after the 'break'. i is unavailable
```

In the second example, we wish to perform some work for all lowercase characters, except 'm'. We use `continue` to ignore the following statements, and to start the next loop (with ch == 'n').
The equivalence of bounded and unbounded loops

We should now be able to see that the for, while, and do ... while control flow statements are each closely related.

To fully understand this, however, we need to accept (for now), that the three "pieces" of the for construct, are not always initialization, condition, modification.

More generally, the three pieces may be C expressions - for the moment we'll consider these as C statements which, if they produce a value, the value is often ignored.

The following loops are actually equivalent:

```c
for( expression1 ; expression2 ; expression3 )
{
    statement1;
    ....
}

expression1;
while(expression2)
{
    statement1;
    ....
    expression3;
}
```

In both cases, we're expecting expression2 to produce a Boolean value, either true or false, as we need that truth value to determine if our loops should "keep going".

You should think about these carefully, perhaps perform some experiments, to determine where control flow really goes when we introduce break and continue statements.
Some unusual loops you will encounter

As you read more C programs, you'll see some statements that look like `for` or `while` loops, but appear to have something missing. In fact, any (or all!) of the 3 "parts" of a `for` loop may be omitted.

For example, the following loop initially sets \( i \) to 1, and increments it each iteration, but it doesn't have a "middle" conditional test to see if the loop has finished. The missing condition constantly evaluates to `true`:

```
for(int i = 1 ; /* condition is missing */ ; i = i+1)
{
    .....  
    .....  
}
```

Some loops don't even have a loop-control variable, and don't test for their termination. This loop will run forever, until we interrupt or terminate the operating system process running the C program. We term these infinite loops:

```
// cryptic - avoid this mechanism
for( ; ; )
{
    .....  
    .....  
}
```

While we often see and write such loops, we don't usually want them to run forever!

We will typically use an enclosed condition and a `break` statement to terminate the loop, either based on some user input, or the state of some calculation.

```
#include <stdbool.h>

// clearer - use this mechanism
while( true )
{
    .....  
    .....  
}
```
Introducing functions

C is a procedural programming language, meaning that its primary synchronous control flow mechanism is the procedure call.

C names its procedures functions (in contrast, Java has a different mechanism - methods).

In Mathematics, we apply a function, such as the trigonometric function \( \cos \), to one of more values. The function performs an evaluation, and returns a result.

In many programming languages, including C, we call a function. We pass zero or more arguments to the function, the function's statements are executed (often involving the arguments), and a result is returned.

We've already seen the example of \( \text{main()} \) - the function that all C programs must have, which we might write in different ways:

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

int main(int argc, char *argv[]) {  
    // check the number of arguments
    if(argc != 2) {
        ....
        exit(EXIT_FAILURE);
    } else {
        ....
        exit(EXIT_SUCCESS);
    }
    return 0;
}
```

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

int main(int argc, char *argv[]) {  
    int result;
    // check the number of arguments
    if(argc != 2) {
        ....
        result = EXIT_FAILURE;
    } else {
        ....
        result = EXIT_SUCCESS;
    }
    return result;
}
```

The operating system calls \( \text{main()} \), passing to it some (command-line) arguments, \( \text{main()} \) executes some statements, and returns to the operating system a result - usually \( \text{EXIT_SUCCESS} \) or \( \text{EXIT_FAILURE} \).
Why do we require functions?

The need for, and use of, `main()` should be clear. However, there's 3 other primary motivations for using functions:

1. Functions allow us to group together statements that have a strongly related purpose - statements, in combination, performing a single task.

   We prefer to keep such statements together, providing them with a *name* (as for variables), so that we may refer to the statements, and call them, collectively.

   This provides both convenience and readability.

2. We often have sequences of statements that appear several times throughout larger programs.

   The repeated sequences may be identical, or very similar (differing only in a very few statements). We group together these similar statement sequences, into a named function, so that we may call the function more than once and have it perform similarly for each call.

   - Historically, we'd identify and group similar statements into functions to minimize the total memory required to hold the (repeated) statements.
   - Today, we use functions not just to save memory, but to enhance the robustness and readability of our code (both good *Software Engineering* techniques).

3. Functions provide a convenient mechanism to package and distribute code. We can distribute code that may be *called* by other people's code, without providing them with a complete program.

   We frequently use *libraries* for this purpose.
Where do we find functions?

1. We will write our own functions, in the same file as `main()`, to simplify our code and to make it easier to read.

2. Soon, we will write our own functions in other files, and call them from our main file.

3. Collections of related functions are termed **libraries** of functions.

   The most prominent example, that we've already seen, is C's **standard library** - a collection of frequently required functions that must be provided by a standards' conforming C compiler.

   In our programming, so far, we've already called **library functions** such as:

   ```c
   printf(), atoi(), and exit().
   ```

4. Similarly, there are many task-specific **3rd-party libraries**. They are not required to come with your C compiler, but may be downloaded or purchased.
The role of function `main()`

In general, even our small programs will have several functions.

*We will no longer place all of our statements in the `main()` function.*

`main()` should be constrained to:

- receive and check the program's command-line arguments,
- report errors detected with command-line arguments, and then call `exit(EXIT_FAILURE)`,
- call functions from `main()`, typically passing information requested and provided by the command-line arguments, and
- finally call `exit(EXIT_SUCCESS)` if all went well.
The *datatype* of a function

There are two distinct categories of functions in C:

1. functions whose role is to just perform a task, and to then return to the statement that called it.

   Such functions often have *side-effects*, such as performing some output, or modifying a global variable so that other statements may access that modified value.

   These functions don't return a specific value to the caller, are termed *void functions*, and we casually say that they "return a *void".

2. functions whose role is to calculate a value, and to return that value for use in the statements that called them. The single value returned will have a *type*, such as *int*, *char*, *bool*, or *float*.

   These functions may also have side-effects.

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

static void output(char ch, int n)
{
    for(int i=1 ; i<=n ; i=i+1)
    {
        printf("%c", ch);
    }
}

int main(int argc, char *argv[])
{
    output(' ', 19);
    output('*', 1);
    output('
', 1);
    return 0;
}
```

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

static float square(float x)
{
    return x * x;
}

int main(int argc, char *argv[]) {
    if(argc > 2) {
        float a, b, sum;
        a = atof(argv[1]);
        b = atof(argv[2]);
        sum = square(a) + square(b);
        printf("hypotenuse = %f\n", sqrt(sum));
    }
    return 0;
}
```

Note that this 2nd example uses the *sqrt()* function from the standard maths *library*.
We should thus compile it with: `gcc .... -lm`
Passing parameters to functions

The examples we've already seen show how parameters are passed to functions:

- a sequence of expressions are separated by commas, as in:
  
  ```c
  a = average3( 12 * 45, 238, x - 981 );
  ```

- each of these expressions has a datatype. In the above example, each of the expressions in an int.
- when the function is called, the expressions are evaluated, and the value of each expression is assigned to the parameters of the function:

  ```c
  float average3( int x, int y, int z )
  {
    return (x + y + z) / 3.0;
  }
  ```

- during the execution of the function, the parameters are local variables of the function.

  They have been initialized with the calling values (x = 12 * 45 ...), and the variables exist while the function is executing.

  They "disappear" when the function returns.

- Quite often, functions require no parameters to execute correctly. We declare such functions with:

  ```c
  void backup_files( void )
  {
    .....  
  }
  ```

  and we just call the functions without any parameters: backup_files();
**Very common mistakes with parameter passing**

Some common misunderstandings about how parameters work often result in incorrect code (even a number of textbooks make these mistakes!):

- The *order of evaluation* of parameters is not defined in C. For example, in the code:

```c
int square( int a )
{
    printf("calculating the square of %i\n", a);
    return a * a;
}

void sum( int x, int y )
{
    printf("sum = %i\n", x + y);
}
....
....
sum( square(3), square(4) );
```

are we hoping the output to be:

- calculating the square of 3  // the output on PowerPC Macs
  calculating the square of 4
  sum = 25

or

- calculating the square of 4  // the output on Intel Macs
  calculating the square of 3
  sum = 25

Do not assume that function parameters are evaluated left-to-right. The compiler will probably choose the *order of evaluation* which produces the most efficient code, and this will vary on different processor architectures.

(A common mistake is to place auto-incrementing of variables in parameters.)
Very common mistakes with parameter passing, continued

Another common mistake is to assume that function arguments and parameters must have the same names to work correctly.

Some novice programmers think that the matching of names is how the arguments are evaluated, and how arguments are bound to parameters. For example, consider the code:

```c
int sum3( int a, int b, int c )
{
    return a + b + c;
}
....
int a, b, c;
a = 1;
b = 4;
c = 9;
printf("%i\n", sum3(c, a, b));
```

Here, the arguments are not "shuffled" until the names match.

It is not the case that arguments must have the same names as the parameters they are bound to. Similarly, the names of variables passed as arguments are not used to "match" arguments to parameters.

If you ever get confused by this, remember that arithmetic expressions, such as $2\times 3 + 1$, do not have names, and yet they are still valid arguments to functions.
Very common mistakes with parameter passing, continued

While not an example of an error, you will sometimes see code such as:

```c
return x;
```

and sometimes:

```c
return(x);
```

While both are correct, the parentheses in the 2nd example are unnecessary.

`return` is not a function call, it is a statement, and so does not need parentheses around the returned value.

**However - at any point when writing code, use extra parentheses if they enhance the readability of your code.**
Functions receiving a variable number of arguments

To conclude our introduction to functions and parameter passing, we consider functions such as `printf()` which may receive a variable number of arguments!

We've carefully introduced the concepts that functions receive strongly typed parameters, that a fixed number of function arguments in the call are bound to the parameters, and that parameters are then considered as local variables.

But, consider the perfectly legal code:

```c
int i = 238;
float x = 1.6;

printf("i is %i, x is %f\n", i, x);
....
printf("this function call only has a single argument\n");
....
printf("x is %f, i is %i, and x is still %f\n", x, i, x);
```

In these cases, the first argument is always a string, but the number and datatype of the provided arguments keeps changing.

`printf()` is one of a small set of standard functions that permits this apparent inconsistency. It should be clear that the format specifiers of the first argument direct the expected type and number of the following arguments.

Fortunately, within the ISO-C99 specification, our gcc compiler is permitted to check our format strings, and warn us (at compile time) if the specifiers and arguments don't "match".

Another good justification for our use of "strong" compilation options:
```
gcc -std=c99 -Wall -Werror -pedantic ...
```

We'll see later in the unit how we can correctly write our own functions to accept a variable number of arguments.
Introducing arrays

As programs become more complex, we notice that they require more variables, and thus more variable *names* to hold all necessary values.

We could define:

```c
int x1, x2, x3, x4, x5..... ;
```

but referring to them in our programs will quickly become unwieldy, and their actual names *may* be trying to tell us something.

In particular, our variables are often related to one another - they hold data having a physical significance, and the data value held in one variable is related to the data in another variable.

For example, consider a 2-dimensional field, where each square metre of the field may be identified by its rows and column coordinates. We may record each square's altitude, or temperature, or its number of ants:

```
(0,0)  (1,0)  (2,0)  (3,0)  
(0,1)  (1,1)  (2,1)  (3,1)  
(0,2)  (1,2)  (2,2)  (3,2)  
```

Like most languages, C provides a simple data structure, termed an *array*, to store and access data where the data items themselves are closely related.

Depending on the context, different problem domains will describe different kinds of arrays with different names:

- 1-dimensional arrays are often termed *vectors*,
- 2-dimensional arrays are often termed *matrices* (as in our example, above),
- 3-dimensional arrays are often termed *volumes*, and so on.

We'll start with the simple 1-dimensional arrays.
1-dimensional arrays

C provides support for 1-dimensional arrays by allowing us to identify the required data using a single index into the array.

Syntactically, we use square-brackets to identify that the variable is an array, and use an expression inside the array to identify which "part" of it we're considering.

In all cases, an array is only a single variable, with one or more elements.

Consider the following code:

```
#define N 20
int myarray[ N ];
int evensum;
evensum = 0;
for(int i=0 ; i < N ; i=i+1) {
    myarray[ i ] = i * 2;
evensum = evensum + myarray[ i ];
}
```

What do we learn from this example?

- We declare our 1-dimensional arrays with square brackets, and indicate the maximum number of elements within those brackets.
- A fixed, known value (here \( N \), with the value 20) is used to specify the number of elements of the array.
- We access elements of the array by providing the array's name, and an integer index into the array.
- Elements of an array may be used in the same contexts as basic (scalar) variables. Here \( \text{myarray} \) is used on both the left-hand and right-hand sides of assignment statements.
- We may also pass array elements as arguments to functions, and return their values from functions.
- Array indices start "counting" from 0 (not from 1).
- Because our array consists of \( N \) integers, and indices begin at zero, the highest valid index is actually \( N-1 \).
Initializing 1-dimensional arrays

Like all variables, arrays should be *initialized* before we try to access their elements. We can:

- initialize the elements at *run-time*, by executing statements to assign values to the elements:

```c
#define N 5
int myarray[N];
....
for(int i=0 ; i < N ; i=i+1)
{
    myarray[i] = i;
}
```

- we may initialize the values at *compile-time*, by telling the compiler what values to initially store in the memory represented by the array. We use curly-brackets (braces) to provide the initial values:

```c
#define N 5
int myarray[N] = {0, 1, 2, 3, 4};
```

- or, we may initialize just a few values at *compile-time*, and have the compiler initialize the rest *with zeroes*:

```c
#define HUGE 10000
int myarray[HUGE] = {4, 5};
```
Strings are 1-dimensional arrays of characters

In contrast to some other programming languages, C does not have a basic datatype for strings. However, C compilers provide some basic support for strings by considering strings to simply be arrays of characters.

We've already seen this support when calling the `printf()` function:

```c
printf("I'm afraid I can't do that Dave\n");
```

The double quotation characters simply envelope the characters to be treated as a sequence of characters.

In addition, a standards' conforming C compiler is required to also provide a large number of string-handling functions in its standard C library. Examples include:

```c
int strlen( char string[] );               // to determine the length of a string
char *strcpy( char destination[], char source[] ); // to make a copy of a string
int strcmp( char str1[], char str2[] );     // to determine if two strings are equal
```

In reality these functions are not "really" managing strings as a basic datatype, but are just managing arrays of characters.
Initializing character arrays

As we've just seen with 1-dimensional arrays of integers, C also provides facility to initialize character arrays.

All of the following examples are valid:

```c
char greeting[5] = { 'h', 'e', 'l', 'l', 'o' };
char today[7] = "Tuesday";
char month[] = "August";
```

The 3rd of these is the most interesting. We have not specified the size of the array month ourselves, but have permitted the compiler to count and allocate the required size.
Strings are terminated by a special character

Unlike other arrays in C, the support for character arrays is extended by treating one character, the null byte, as having special significance. We may specify the null byte, as in the example:

```c
array[3] = '\0';
```

The null byte is used to indicate the end of a character sequence, and it exists at the end of all strings that are defined within double-quotes.

Inside the computer's memory we have:

```
hello
```

Of note, when dealing with strings:

- the string requires 6 bytes of memory to be stored correctly, but
- functions such as `strlen()`, which calculate the string's length, will report it as 5.

There is no inconsistency here - just something to watch out for.

Because the null byte has special significance, and because we may think of strings and character arrays as the same thing, we can manipulate the contents of strings by changing the array elements. Consider:

```
hello
world
```

If we execute the statement:

```c
array[5] = '\0';
```

the space between the two words is replaced by the null byte. The result is that the array still occupies 12 bytes of storage, but if we tried to print it out, we would only get `hello`. 

---

Copying strings

As strings are so important, the standard C library provides many functions to examine and manipulate strings. However, C provides no basic string datatype, so we often need to treat strings as array of characters.

Consider these implementations of functions to copy one string into another:

```c
// determine the string length, then use a bounded loop
void my_strcpy(char destination[], char source[])
{
    int length = strlen(source);
    for(int i = 0 ; i < length ; i = i+1)
    {
        destination[i] = source[i];
    }
    destination[length] = '\0';
}

// DO NOT WRITE STRING-PROCESSING LOOPS THIS WAY
void my_strcpy(char destination[], char source[])
{
    for(int i = 0 ; i < strlen(source) ; i = i+1)
    {
        destination[i] = source[i];
    }
    destination[i] = '\0';
}

// use an unbounded loop, copying until the null-byte
void my_strcpy(char destination[], char source[])
{
    int i = 0;
    while(source[i] != '\0')
    {
        destination[i] = source[i];
        i = i+1;
    }
    destination[i] = '\0';
}

// use an unbounded loop, copying until the null-byte
void my_strcpy(char destination[], char source[])
{
    int i = 0;
    do
    {
        destination[i] = source[i];
        i = i+1;
    } while(source[i-1] != '\0');
}
```

**NOTE** that the above function in the printed lecture notes incorrectly has `while(source[i] != '\0');`
Formatting our results into character arrays

There are many occasions when we wish our "output" to be written to a character array, rather than to the screen. Fortunately, we need to learn very little - we now call standard function `sprintf`, rather than `printf`, to perform our formatting.

```c
char chess_outcome[64];

if(winner == WHITE) {
    sprintf(chess_outcome, "WHITE had %i", nwhite_pieces);
} else {
    sprintf(chess_outcome, "BLACK had %i", nblack_pieces);
}
```

We must be careful, now, not to exceed the maximum length of the array receiving the formatted printing. Thus, we prefer functions which ensure that not too many characters are copied:

```c
char chess_outcome[64];

// format, at most, a known number of characters
if(winner == WHITE) {
    snprintf(chess_outcome, 64, "WHITE had %i", nwhite_pieces);
}

// OR, greatly preferred:
if(winner == WHITE) {
    snprintf(chess_outcome, sizeof(chess_outcome), "WHITE had %i", nwhite_pieces);
}
```
Pre- and post-, increment and decrement

To date, we've always employed the "traditional" mechanism of incrementing integer values, in both assignment statements and in for loops:

```c
int value = 0;
.....
value = value + 1;
.....
for(int i=0 ; i < MAXVALUE ; i=i+1)
{
    ....
}
```

C provides a shorthand notation for incrementing and decrementing scalar variables, by one:

```c
int value = 0;
char ch    = 'z';

++value;     // value is now 1
--ch;        // ch is now 'y'

.....
for(int i=0 ; i < MAXVALUE ; ++i)
{
    ....
}
for(char letter='a' ; letter <= 'z' ; ++letter)
{
    ....
}
```

The notation used above is always used to increment or decrement by one, and the 2 statements:

```c
++value ;     // pre-increment value
value++ ;     // post-increment value
```

produce the exact same result.
Pre- and post-, increment and decrement, continued

While pre- and post- incrementing (and decrementing) initially appears simple, we must be careful when using modified variables in expressions. Consider these results:

```c
int x = 0;
int y = 0;
int what = 0;

// ------------------- what --- X --- Y -
what  = ++x;       //  1  1  0
what  = y++;       //  0  1  1
what  = y++;       //  1  1  2
what  = ++y;       //  3  1  3
```

Shorthand arithmetic

A similar notation may be used to perform any standard arithmetic operation on a variable. For example, assuming the correct declarations:

```c
value += 2;       // equivalent to value = value + 2;
value -= y;       // equivalent to value = value - y;
total *= x;       // equivalent to total = total * x;
half /= 2;        // equivalent to half = half / 2;
poly  += x*1;     // equivalent to poly  = poly + (x*1);
```
A slow introduction to pointers in C

The ISO-C99 programming language has a very powerful feature, which we'll casually term "pointers in C" for now.

- If used correctly pointers can enable very fast, efficient, execution of C programs.

- If misunderstood or used incorrectly, pointers can make your programs do very strange, often incorrect things, and result in very hard to diagnose and debug programs.

The primary role of pointers - to allow a program (at run-time) to access its own memory - sounds like a useful feature, but is often described as a very dangerous feature.

There is much written about the power and expressiveness of C's pointers, and much (more recently) written about Java's lack of pointers.

More precisely, Java does have pointers, termed references, but the references to Java's objects are so consistently and carefully constrained at both compile and run-time, that very little can go wrong.
What are pointers?

We know that C has both "standard" variables and structures, also described as scalar variables. In addition, we've seen arrays of these variables.

Let's follow this simplified explanation:

- We understand that variables occupy memory locations (1 or more bytes) of a computer's memory.
- Each variable requires enough bytes to store the values the variable will need to hold. For example, on our CSSE labs' computers, an integer will require 4 bytes of memory.
- Similarly, an array of 100 integers, will require 400 bytes of contiguous memory.
- Computers have a large amount of memory, e.g. our lab computers have 4 gigabytes of memory (4GB), or a little over 4 billion bytes.
- Each of a computer's memory bytes is uniquely numbered, from 0 to some large value. Each such number is termed the byte's memory address.
- We often refer to memory locations as just addresses and the action of identifying an address as addressing.

With these points in mind, we can make 3 simple statements:

1. A pointer is a variable that holds the address of a memory location.
2. Pointers are variables that point to memory locations.
3. Pointers (usually) point to memory locations being used to hold variables' values/contents.
The & operator, the address-of operator, the ampersand operator

The punctuation character &, often pronounced as the address of operator, is used to find a variable's address.

For example, we’d pronounce this as:

```c
int total;
.... &total ....
```

"the address of total", and if the integer variable `total` was located at memory address 10,000 then the value of `&total` would be 10,000.

We can now introduce a variable named `p`, which is a pointer to an integer (pedantically, `p` is a variable used to store the address of a memory location that we expect to hold an integer value).

```c
int total;
int *p ;
p = &total ;
```

If the integer variable `total` was located at memory address 10,000 then the value of `p` would be 10,000.

If necessary (though rarely), we can print out the address of a variable, or the value of a pointer, by first casting it to something we can print, such as an unsigned integer, or to an "generic" pointer:

```c
int total;
int *p = &total ;
printf("address of variable is: \%lu\n", (unsigned long)&total );
printf(" value of pointer p is: \%lu\n", (unsigned long)p );
printf(" value of pointer p is: \%p\n", (void *)p );
```
**Dereferencing a pointer**

We now know that pointer variables may point to memory locations holding variables' values.

It should also be obvious that if the variable's value (contents) changes, then the pointer will *keep* pointing to the same variable, (which now holds the new value).

We can use C's concept of dereferencing a pointer to determine the value the pointer points to:

```c
int total;
int *p = &total;

total = 3;
printf("value of variable total is: %i\n", total);
printf("value pointed to by pointer p is: %i\n", *p);

++total; // increment the value that p points to
printf("value of variable total is: %i\n", total);
printf("value pointed to by pointer p is: %i\n", *p);
```

Even though the variable's value has changed (from 3 to 4), the pointer still points at the variable's location.

The pointer first pointed at an address containing 3, and the pointer kept pointing at an address (the same address) containing 4.
Dereferencing a pointer, continued

We now know that changing the value that a pointer points to does not change the pointer (good!).

Now we'd like to change the value that the pointer points to.

Similarly, this will not change the pointer itself.

```c
int total;
int *p = &total;
int bigger;

total = 8;

printf("value of variable total is: %i\n", total);
printf("value pointed to by pointer p is: %i\n", *p);

*p = *p + 2; // increment, by 2, the value that p points to

printf("value of variable total is: %i\n", total);
printf("value pointed to by pointer p is: %i\n", *p);

bigger = *p + 2; // just fetch the value that p points to

printf("value of variable total is: %i\n", total);
printf("value of variable bigger is: %i\n", bigger);
printf("value pointed to by pointer p is: %i\n", *p);
```
An array's name is an address

When finding the address of a scalar variable (or a structure), we precede the name by the address of operator, the ampersand:

```c
int total;
int *p = &total ;
```

However, when requiring the address of an array, we're really asking for the address of the first element of that array:

```c
#define N 10
int totals[N];
int *first = &totals[0]; // the first element of the array
int *second = &totals[1]; // the second second of the array
int *third  = &totals[2]; // the third third of the array
```

As we frequently use a pointer to traverse the elements of an array (see the following slides on pointer arithmetic), we observe the following equivalence:

```c
int *p = &totals[0] ;
```

and

```c
int *p = totals ;
```

That is: "an array's name is synonymous with the address of the first element of that array".
**Pointer Arithmetic**

Another facility in C is the use of *pointer arithmetic* with which we may *change a pointer's value* so that it points to *successive* memory locations.

We specify pointer arithmetic in the same way that we specify numeric arithmetic, using the symbols `++` and `- -` to request pre- and post- increment and decrement operators.

We generally use pointer arithmetic when accessing *successive* elements of arrays.

Consider the following example, which initializes all elements of an integer array:

```c
#define N 10
int totals[N];
int *p = totals; // p points to the first/leftmost element of totals

for(int i=0 ; i<N ; ++i)
{
    *p = 0; // set what p points to to zero
    ++p;    // advance/move pointer p "right" to point to the next integer
}

for(int i=0 ; i<N ; ++i)
{
    printf("value of totals[%i] is: %i\n", i, totals[i]);
}
```
How far does the pointer move?

It would make little sense to be able to "point anywhere" into memory, and so C automatically adjusts pointers (forwards and backwards) by values that are multiples of the size of the base types (or user-defined structures) to which the pointer points(!).

In our example:

```c
for(int i=0 ; i<N ; ++i)
{
    *p = 0;  // set what p points to to zero
    ++p;    // advance/move pointer p "right" to point to the next integer
}
```

*p will initially point to the location of the variable:

- `totals[0]`, then to
- `totals[1]`, then to
- `totals[2]` ...

Similarly, we can say that *p has the values:

- `&totals[0]`, then
- `&totals[1]`, then
- `&totals[2]` ...

Combining pointer arithmetic and dereferencing

With *great care*, we can also combine pointer arithmetic with dereferencing:

```c
#define N    10

int totals[N];
int *p = totals;

for(int i=0 ; i<N ; ++i)
{
    *p++ = 0; // set what p points to to zero, and then
    // advance p to point to the "next" integer
}

for(int i=0 ; i<N ; ++i)
{
    printf("value of totals[%i] is: %i\n", i, totals[i]);
}
```

In English, we read this as:

"set the contents of the location that the pointer p currently points to the value zero, and then increment the value of pointer p by the size of the variable that it points to." 😊.

Similarly we can employ pointer arithmetic in the control of *for* loops. Consider this excellent use of the preprocessor:

```c
int array[N];
int n;
int *a;

#define FOREACH_ARRAY_ELEMENT  for(n=0, a=array ; n<N ; ++n, ++a)

FOREACH_ARRAY_ELEMENT
{
    if(*a == 0)
    {
        ....
    }
}
```
Functions with pointer parameters

We know that pointers are simply variables.
We now use this fact to implement functions that receive pointers as parameters.

A pointer parameter will be initialized with an address when the function is called.

Consider two equivalent implementations of C's standard strlen function - the traditional approach is to employ a parameter that "looks like" an array; our new approach employs a pointer parameter:

```
int strlen_array( char array[] )
{
    int len = 0;
    while( array[len] != '\0' ) {
        ++len;
    }
    return len;
}
```

```
int strlen_pointer( char *strp )
{
    int len = 0;
    while( *strp != '\0' ) {
        ++len;
        ++strp;
    }
    return len;
}
```

During the execution of the function, any changes to the pointer will simply change what it points to.
In this example, strp traverses the null-byte terminated character array (a string) that was passed as an argument to the function.

We are not modifying the string that the pointer points to, we are simply accessing adjacent, contiguous, memory locations until we find the null-byte.
Returning a pointer from a function

Let's consider another example. Here we provide two equivalent implementations of C's standard `strcpy` function, which copies a string from its source, `src`, to a new destination, `dest`.

The C99 standards state that the `strcpy` function function must return a copy of its destination parameter.

In both cases, we are returning a copy of the `dest` parameter - that is, we are returning a pointer as the function's value.

We say that "the function's return-type is a pointer".
Returning a pointer from a function, continued

Consider the following functions to copy one string into another:

```c
char *strcpy_array( char dest[], char src[] ) // returns a pointer
{
    int i = 0;
    while( src[i] != '\0' ) {
        dest[ i ] = src[ i ];
        ++i;
    }
    dest[ i ] = '\0';

    return dest; // returns the original destination parameter
}

char *strcpy_pointer( char *dest, char *src ) // two pointer parameters
{
    char *destcopy = dest; // take a copy of the dest parameter
    while( *src != '\0' ) {
        *dest = *src;       // copy one character from src to dest
        ++src;
        ++dest;
    }
    *dest = '\0';

    return destcopy; // returns a copy of the original destination parameter
}
```

Note:

- In the array version, the function returns a pointer as its value. This further demonstrates the equivalence between array names (here, dest) and a pointer to the first element of that array.

- In the pointer version, we move the dest parameter after we have copied each character, and thus we must first save and then return a copy of the parameter’s original value.

- If very careful, we could reduce the loop of the pointer version to the statement `*dest++ = *src++;`
Passing pointers to functions

Consider a very simple function, whose role is to swap two integer values:

```c
#include <stdio.h>

static void swap(int i, int j)
{
    int temp;
    temp = i;
    i    = j;
    j    = temp;
}

int main(int argc, char *argv[])
{
    int a=3, b=5; // MULTIPLE DEFINITIONS AND INITIALIZATIONS
    printf("before a=%i, b=%i\n", a, b);
    swap(a, b); // ATTEMPT TO SWAP THE 2 INTEGERS
    printf("after  a=%i, b=%i\n", a, b);
    return 0;
}
```

Doh! What went wrong?

The "problem" occurs because we are not actually swapping the values contained in our variables a and b, but are (successfully) swapping copies of those values.
Passing pointers to functions, continued

Instead, we need to pass a 'reference' to the two integers to be interchanged.

We need to give the `swap()` function "access" to the variables `a` and `b`, so that `swap()` may modify those variables:

```c
#include <stdio.h>

static void swap(int *ip, int *jp)
{
    int temp;
    temp = *ip;       // swap's temp is now 3
    *ip = *jp;        // main's variable a is now 5
    *jp = temp;       // main's variable b is now 3
}

int main(int argc, char *argv[])
{
    int a=3, b=5;
    printf("before a=%i, b=%i\n", a, b);
    swap(&a, &b);    // pass pointers to our local variables
    printf("after  a=%i, b=%i\n", a, b);
    return 0;
}

before a=3, b=5
after  a=5, b=3
```

Much better! Of note:

- The function `swap()` is now dealing with the original variables, rather than new copies of their values.
- A function may permit another function to modify its variables, by passing `pointers to those variables`.
- The receiving function now modifies `what those pointers point to`. 
**Duplicating a string**

We know that:

- C considers null-byte terminated character arrays as strings, and
- the length of such strings is not determined by the array size, but by where the null-byte is.

So how could we take a duplicate copy, a clone, of a string? We could try:

```c
#include <string.h>

static char *my_strdup(char *str)
{
    char bigarray[SOME_HUGE_SIZE];
    strcpy(bigarray, str);  // WILL ENSURE THAT bigarray IS NULL-BYTE TERMINATED
    return bigarray;       // RETURN THE ADDRESS OF bigarray
}
```

But we’d instantly have two problems:

1. we’d never be able to know the largest array size required to copy the arbitrary string argument, and
2. we can’t return the address of any local variable. Once function *my_strdup()* returns, variable *bigarray* no longer exists, and so we can’t provide the caller with its address.
Allocating new memory

Let's first address the first of these problems - we do not know, until the function is called, how big the array should be.

It is often the case that we do not know, until we execute our programs how much memory we'll really need!

Instead of using a fixed sized array whose size may sometimes be too small, we must dynamically request some new memory at runtime to hold our desired result.

This is a fundamental (and initially confusing) concept of most programming languages - the ability to request from the operating system additional memory for our programs.

C99 provides a small collection of functions to support memory allocation.
The primary function we'll see is named malloc(), which is declared in the standard <stdlib.h> header file:

```
#include <stdlib.h>
extern void *malloc( size_t nbytes );
```

malloc() is a function (external to our programs) that returns a pointer. However, malloc() doesn't really know what it's returning a pointer to, it doesn't know if it's a pointer to an integer, or a pointer to a character, or even to one our own user-defined types.

For this reason, we use the generic pointer, pronounced "void star" or "void pointer".

It's a pointer to "something", and we only "know" what that is when we place an interpretation on the pointer.

malloc() needs to be informed of the amount of memory that it should allocate - the number of bytes we require.

We use the standard datatype size_t to hold an integer value that may be 0 or positive (we obviously can't request a negative amount of memory!).

We have used, but skipped over, the use of size_t before - it's the datatype of values returned by the sizeof operator, and the pedantically-correct type returned by the strlen() function.
Checking memory allocations

Of course, the memory in our computers is finite (even if it is several gigabytes!), and if we keep calling `malloc()` in our programs, we'll eventually exhaust available memory.

Note that a machine's operating system will probably not allocate all memory to a single program, anyway. There's a lot going on on a standard computer, and those other activities all require memory, too.

For programs that perform more than a few allocations, or even some potentially large allocations, we need to check the value returned by `malloc()` to determine if it succeeded:

```c
#include <stdlib.h>

int bytes_wanted = 1000000 * sizeof(int);
int *huge_array = malloc(bytes_wanted);
if (huge_array == NULL) // DID malloc FAIL?
{
    printf("Cannot allocate %i bytes of memory\n", bytes_wanted);
    exit(EXIT_FAILURE);
}
```

Strictly speaking, we should check all allocation requests to both `malloc()` and `calloc()`.
Duplicating a string, continued

We'll now use `malloc()` to dynamically allocate, at runtime, exactly the correct amount of memory that we need.

When duplicating a string, we need enough new bytes to hold every character of the string, including a null-byte to terminate the string.

This is 1 more than the value returned by `strlen`:

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

static char *my_strdup2(char *str)
{
    char *new = malloc(strlen(str) + 1);
    if(new != NULL)
    {
        strcpy(new, str);  // ENSURES THAT DUPLICATE WILL BE NUL-TERMINATED
    }
    return new;
}
```

Of note:
- we are not returning the address of a local variable from our function - we've solved both of our problems!
- we're returning a pointer to some additional memory given to us by the operating system.
- this memory does not "disappear" when the function returns, and so it's safe to provide this value (a pointer) to whoever called `my_strdup2`.
- the new memory provided by the operating system resides in a reserved (large) memory region termed the heap. We never access the heap directly (we leave that to `malloc()`) and just use (correctly) the space returned by `malloc()`.
## Allocating an array of integers

Let's quickly visit another example of `malloc()`. We'll allocate enough memory to hold an array of integers:

```c
#include <stdlib.h>

static int *randomints(int wanted) {
    int *array = malloc( wanted * sizeof(int) );
    if(array != NULL) {
        for(int i=0 ; i<wanted ; ++i) {
            array[i] = rand() % 100;
        }
    }
    return array;
}
```

Of note:

- `malloc()` is used here to allocate memory that we'll be treating as integers.
- `malloc()` does not know about our eventual use for the memory it returns.
- how much memory did we need?

We know how many integers we want, `wanted`, and we know the space occupied by each of them, `sizeof(int)`. We thus just multiply these two to determine how many bytes we ask `malloc()` for.
**Requesting that new memory be cleared**

In many situations we want our allocated memory to have a known value. The C99 standard library provides a single function to provide the most common case - clearing allocated memory:

```c
extern void *calloc( size_t nitems, size_t itemsize );
....
int *intarray = calloc(N, sizeof(int));
```

It's lost in C's history why `malloc()` and `calloc()` have different calling sequences.

To explain what is happening, here, we can even write our own version, if we are careful:

```c
#include <string.h>

static void *my_calloc( size_t nitems, size_t itemsize )
{
    int nbytes = nitems * itemsize;

    void *result = malloc( nbytes );
    if(result != NULL)
    {
        memset( result, 0, nbytes ); // SETS ALL BYTES IN result TO THE VALUE 0
    }
    return result;
}
....
int *intarray = my_calloc(N, sizeof(int));
```
Deallocating memory with `free`

In programs that:

- run for a long time
  (perhaps long-running server programs such as web-servers), or
- temporarily require a lot of memory, and then no longer require it,

we should deallocate the memory provided to us by `malloc()` and `calloc()`.

The C99 standard library provides an obvious function to perform this:

```c
extern void free( void *pointer );
```

Any pointer successfully returned by `malloc()` or `calloc()` may be freed.

Think of it as requesting that some of the allocated heap memory be given back to the operating system for re-use.

```c
#include <stdlib.h>

int *vector = randomints( 1000 );

if(vector != NULL)
{
    // USE THE vector
    ......
}
free( vector );
```

Note, there is no need for your programs to completely deallocate all of their allocated memory before they exit - the operating system will do that for you.
Reallocating previously allocated memory

We've already seen that it's often the case that we don't know our program's memory requirements until we run the program.

Even then, depending on the input given to our program, or the execution of our program, we often need to allocate more than our initial "guess".

The C99 standard library provides a function, named `realloc()` to grow (or rarely shrink) our previously allocate memory:

```c
extern void *realloc( void *oldpointer, size_t newsize );
```

We pass to `realloc()` a pointer than has previously been allocated by `malloc()`, `calloc()`, or (now) `realloc()`.

Most programs wish to extend the initially allocated memory:

```c
#include <stdlib.h>
int original;
int newsize;
int *array;
int *newarray;

array = malloc( original * sizeof(int) );
if(array == NULL) {
    // HANDLE THE ALLOCATION FAILURE
}
newarray = realloc( array, newsize * sizeof(int) );
if(newarray == NULL) {
    // HANDLE THE ALLOCATION FAILURE
}
```

Of note:
- If `realloc()` fails to allocate the revised size, it returns the `NULL` pointer.
- If successful, `realloc()` copies any old data into the newly allocated memory, and then deallocates the old memory.
- If the new request does not actually require new memory to be allocated, `realloc()` will usually return the same value of `oldpointer`.
- A request to `realloc()` with an "initial" address of NULL, is identical to just calling `malloc()`.

```c
#include <stdlib.h>
int nitems = 0;
int *items = NULL;

while( fgets(.....) != NULL )
{
    items = realloc( items, (nitems+1) * sizeof(items[0]) );
    if(items == NULL)
    {
        // HANDLE THE ALLOCATION FAILURE
    }
    ++nitems;
}
if(items != NULL)
{
    free( items );
}
```