Chapter 1 (C version): Review of C Programming

Contents

Review of C Programming[1]

These notes present a concise review of the C programming language with examples, stored on Google Drive in file `c_examples.tar`. On Linux, extract the example C source files with the command `tar –xvf c_examples.tar`.

Perhaps the most classic text on C programming is:

> Kernighan, Brian W., and Dennis M. Ritchie. *The C programming language*. 2nd Edition. Englewood Cliffs: Prentice-Hall, 1988.

You do not need to purchase this text; there is a PDF version available online. The websites www.cplusplus.com and www.cprogramming.com also provide comprehensive tutorial information for C functions and syntax as well as C++ classes, functions, and syntax. This document is a concise review of C – if you go to www.google.com, www.cplusplus.com, etc. and type each function name (e.g., `printf`) you can find a detailed description of function input arguments and return values.

If you are an experienced C programmer, you will likely be asked to help other students with less experience in C, where you will learn by teaching rather than exposure to totally new material from these notes. Finally, please be aware that these notes cover C thus are compatible with the "original" language specification covered in [Kernighan and Ritchie] as well as the newer C99 language specification. The languages are largely identical but C99 includes extensions enabling features such as inline (locally scoped) variable declarations and variable-length arrays.

Introduction to C

The C language was developed for procedural programming. In procedural programming, a logical step-by-step sequence of calculations, e.g., historically described in a flow chart, is used to specify each function or procedure, and a single high-level main program executes a sequence of function calls. Code is then written to represent these steps, and data is defined as needed. Readability of procedural code is improved by careful definition and use/re-use of simple functions that are then composed. See the companion C++ review document for an introduction to object-oriented programming. Object-oriented programming has proven very useful to improve scalability and ability to debug code at the cost of modest overhead. Languages such as Java are inherently object-oriented.

[1] This is a companion document to the "Review of C++" tutorial posted. While C++ is most commonly used today it is not yet as fast or simple as C for embedded procedural programming. This document mirrors the C++ review document but only uses pure C syntax (thus files have a ".c" extension and can be compiled with "gcc" on Linux).

In C, one learns to write a special function called `main()` that serves as the entry point to all programs. The `main()` function in turn calls functions that initialize variables, execute the top-level procedure(s) until they complete/end. Under normal or exception conditions, `main()` will ultimately exit (or you can hit CTRL-C or `kill` the process (`ps`) from your Linux shell).

The basic structure of a C program, given a `.c` file extension, is shown below. The "#" indicates a preprocessor statement (e.g., `#include` cuts/pastes declaration/definition information into the code). In this case, the header (.h) file `stdio.h` is included – this file declares a basic set of standard (std) input/output (io) functions. The curly braces {} are used to group parts of code together – in the below example the block of code defining the `main()` function is grouped in curly braces. The `main` in this example has no input arguments, which would be specified as the parenthetical list including data type and variable name. The `main` function returns an integer data type (`int`) indicating whether the program succeeded (0) or exited with a non-zero error code. The double slash (//) denotes a comment from the // to the end of the current line; one can also use the block comment format denoted by an initial /* and a final */, e.g., `/* Everything between the start and end of the block is considered a "comment"… */` . Edit code using your favorite editor (e.g., `nano, pico, emacs, gedit, vi, vim` on Linux) – just be sure the editor recognizes C syntax so you will have help with color-coding key words, matching curly braces, and indenting code blocks properly.

In the below example, the function `scanf` reads data from the keyboard while `printf` is used to print data to the screen (`stdout`). In the `printf` function call, there are two data fields: the format string (a C string is enclosed in double quotes) followed by a list of any variables to print. Each variable to be printed is referenced in the format string by a format specifier given by % followed by a sequence of characters indicating the data type and how that many digits to use, etc. See the `printf` page on www.cplusplus.com for a complete list of format specifier options. The `scanf` function uses the same format specifier structure and options, but the list of variables to read must be passed in by memory address (&) rather than by value, as shown in the below example. If you inadvertently pass a variable memory address to printf you will just see bad data printed to the screen; if you pass a value (e.g., x) rather than a memory address (e.g., &x) into scanf the compiler may warn you, but not in all cases.

```
//  An initial example of C code (copy/paste as example1.c)
#include <stdio.h>     // Input/output function declarations
int main()
{ double x;  // A floating-point variable
  printf("Enter x:\n");
  scanf("%lf", &x);  // Read x from
  printf("The variable x is:  %lf  \n", x);
  return 0;
}
```


Send questions/comments to Prof. Atkins (ematkins@umich.edu)

If such a problem exists after the program compiles, your program will exit ungracefully as `scanf` will try to read data into an inappropriate memory address. Try removing the "&" before the `x` in `scanf` and recompile to see for yourself in the below example. "Segmentation faults" are important to recognize and remedy (never ignore them just because code might work sometimes!).

`Quick compiling notes:` On Linux, you can compile the above C code into an executable program with: `gcc example1.c` which will create an executable `a.out` which can be run from the command prompt with `./a.out` (the ./ specifies your current directory). If you want to name your executable something more intuitive than `a.out` do this with the `-o` compiler option, e.g., `gcc example1.c –o example1` Note that on Linux you need not use a particular file extension for an executable program but this file needs to have "execute" permissions `(chmod +x filename).` `gcc` generates executable files by default; you can view your file's permissions through the directory/file listing command `ls –l filename.` The first output first field will have a combination of permissions `"rwx"` (read, write, execute). A letter (r, w, x) indicates the file has that permission; a dash (-) indicates the file does not have that permission. The field `"rwx"` can also be treated as a three-bit binary number where decimal 7 (111 binary) is `"rwx",` decimal 4 (100 binary) is read only, etc.

C data types, operators, and math functions

C has three basic data types: the character (`char`), integer (`int`), and real or floating-point number (`float, double`). Characters and integers can be explicitly specified as `signed` or `unsigned`, and integers can be `short` or `long`. A character is one byte (8 bits); an unsigned character can represent the (base 10) values 0 to 255, while a signed character, using two's complement notation[2] for negative numbers, represents values from -128 to 127. A list of built-in math operators is given in Table 1. Order of precedence is listed, with () having top priority.

Table 1: Basic math operators in C (and C++)

1.	()	Parenthesis for grouping
2.	-	Unary minus to change a variable's sign (e.g., -x)
3.	*	Multiplication
4.	/	Division (note that one or both operands must be floating-point for the result to be floating-point, so 1.0/2 = 0.5 but 1/2 = 0 because the integer result is truncated)
5.	%	Modulus or integer remainder (e.g., 10/8 = 1, 10%8 = 2)
6.	+	Addition
7.	-	Subtraction
8.	=	Assignment

[2] You are again encouraged to consult an online source for detailed information on specific terms you may find unfamiliar, e.g., "two's complement.

Order of application for the increment (++) and decrement (--) operators depend on whether they precede or succeed the variable on which they operate. "Pre-increment" (e.g., ++x) increments the variable before it is used in any equation, while "post-increment" (e.g., x++) leaves the variable unchanged until after it is used where embedded in a computation. For example, suppose *x* is initialized through the statement `int x=3;`. The statement `y = 2 * ++x;` sets *y* to 8 while the statement `y = 2 * x++;` sets *y* to 6; *x* is incremented to 4 with both statements. "Shortcut" operators `+=, -=, *=, /=` also are defined for convenience and are used to reassign variable values. For example, `x = x – 52.5;` can be written `x -= 52.5;`

Basic math functions are accessed by `#include`'ing a header file `math.h` common to C and C++ that declares math functions and `#define`'s math constants such as `M_PI` (for π) and `NAN` (for not-a-number; commonly seen when a floating-point variable is not properly initialized or when dividing by zero). In C this header file is referenced as `<math.h>`. Basic math functions are reviewed below; they are listed with hyperlinks containing more information in www.cplusplus.com/reference/cmath. Note that floating-point absolute value requires `fabs(),` not `abs()` which is for integers. No single-symbol exponentiation operator is available in C++; instead use the `pow()` function. All trigonometric functions presume angles are specified in radians. There is NEVER cause to use the `atan()` function; instead use `atan2(y, x)` which uses the sign of the two input arguments to return an angle in the proper quadrant.

Table 2: Commonly-used math functions (valid in C and C++)

Trigonometric: `cos(), sin(), tan(), acos(), asin(), atan2()`
Hyperbolic: `cosh(), sinh(), tanh(), acosh(), asinh(), atanh()`
Exponential and logarithmic: `exp(), log(), log10(), log2(), logb()`
Power functions: `pow(), sqrt(), cbrt(), hypot()`
Error and gamma functions: `erf(), erfc(), tgamma(), lgamma()`
Rounding and remainder functions: `ceil(), floor(), fmod(), trunk(), round(), remquo()`

A suite of relational and logical operators are also available. These are listed below in Table 3. Note that C uses "side by side" symbols to represent operators that are not naturally indicated on a standard keyboard (e.g., >= for "greater-than-or-equal-to"). The exclamation (!) is used to indicate negation or "not" for both relational and logical operators. Precedence is given first to math operators, then to relational operators, then finally to logical operators; parentheses can still be used to manually specify precedence over any operator set. Table 4 shows bitwise operators available to C (and C++); bitwise operators are useful for embedded code in which individual bits need to be set (1), cleared (0), or accessed. Hexadecimal representation of numbers, indicated by a leading `0x` and digits `0-9, A-F` is convenient when accessing or manipulating individual variable bits. For example, to clear bit 0 in byte variable `x`, stored as an 8-bit `unsigned char`, one can use the following expression: `x = x & 0xFE`.

Table 3: Relational and Logical Operators for C and C++

Relational:	
>	Strictly greater than
<	Strictly less than
>=	Greater than or equal to
<=	Less than or equal to
!=	Not equal to
==	Equality test[3]
Logical (note that in C++ false = 0; true = any non-zero value):	
!	NOT (inversion)
&&	AND
||	OR

Table 4: Bitwise Operators in C and C++

~	bitwise NOT (flips each bit)
&	bitwise AND
|	bitwise OR
^	bitwise XOR (exclusive or)
<<	bit shift to the left (zeros are shifted into open bit fields by default)
>>	bit shift to the right

Below is an example C program that uses a simple math function. Note that with gcc you must "link" the code to the standard math library using `-lm` (`-l` indicates a library, `m` indicates math), e.g., `gcc example2.c -lm -o example2`.

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

int main()
{
      double x, y;
      x = M_PI/2;
      y = sin(x);
      printf("y = %lf\n", y);
      return 0;
}
```

[3] Be careful not to accidentally use assignment operator = for equality testing. This is a common error and it's hard to catch because no syntax error is introduced (so the code will compile and run).

Selection

Algorithms often require execution of code only when certain conditions are true. The selection primitives in C (and C++), `if/else` and `switch/case`, allow one to select and execute the appropriate code "branch" based on "tests", often using the relational and logical operators described above. Each test returns `TRUE` (non-zero) or `FALSE` (zero). The syntax is shown below for an `if/else` block of code. While `if/else` is sufficiently expressive to handle all selection code needs in C, some prefer to organize a series of tests of a single value in a `switch/case` block; please see an online source for more details of `switch/case`. Note the use of curly braces to define code blocks and the use of "nested ifs" in the `beer.c` example.

```
if (test1) {
  code1;   // Executed if test1 returns true
} else if (test2) {  // Note the space between else and if
  code2;  // Executed when test1 returns false and test2 returns true
} else {
  code3;  // Executed when test1 and test2 both return false
}
```

Below is program `beer.c` that is motivational to many young undergraduates…

```
#include <stdio.h>
int main()
{
  int b, btype;
  printf("Enter your favorite beverage:  (1=beer, 2=coke, 3=water):\n");
  scanf("%d", &b);

  if (b == 1) {
    printf("What kind would you like?  ");
    printf("We have Guinness (1) and Bud (2) on draft.\n");
    scanf("%d", &btype);
    if (btype == 1) {  // This selection block is nested, executing when b==1
      printf("Mmmmhh -- tasty!\n");
    } else if (btype == 2) {
      printf("Swill!  Bleach!!!!\n");
    } else {
      printf("We don't have that, you fool.\n");
    }
  } else if (b==2 || b==3) {
    printf("Boring...  Are you planning to drive home?\n");
  } else {
    printf("Have you been drinking too much already?\n");
    printf("I might suggest water next time...\n");
  }
  return 0;
}
```

Iteration

Most all programming languages offer iteration, enabling the same block of code to be executed as many times as is needed based on a test. In C (and C++) two iteration constructs are offered: `for` and `while`. Typically, for is used when an iteration variable (e.g., a loop counter) manages loop execution. The syntax is `for (initialization; test; iteration step) { code block }.` The initialization code is executed only once when the `for` statement is first reached. The `test` determines whether the code block should again execute or should be skipped. The `iteration step` specifies how the loop variable is modified between executions of code block.

`while` is typically preferred when the number of iterations is not pre-specified but instead based on the value(s) of variable(s) that are modified as part of the iterative code block. Two forms of the `while` iteration construct are available: `(1) while (test) { code block }` where the `test` is executed before the `code block` is ever executed; if the `test` initially returns false the `code block` is never executed, and (2) `do { code block} while (test);` where the `code block` is executed once before the `test` is evaluated.

A statement `if (test) break;` can be embedded in any iteration code block to allow the `code block` to terminate before it completes. There always must be some `test` that will eventually return `false` as iteration proceeds. Otherwise the code will be said to be stuck in an infinite loop, running until the machine reboots or you manually end the executing program, e.g., with `CTRL-C` at the Linux prompt. One can design iteration code blocks to depend on an intermediate `break` rather than a `test`. In such a design one could purposely set up a loop that appears intentionally infinite until examining the `break` condition, such as `for(;;) { code block with break condition }` or `while (1) { code block with break condition }.` Below are a series of example codes illustrating the use of iteration. In the first example, a single `for` block is shown that prints even numbers between 0 and 100. Note that this example also illustrates how C (and C++) allow one to declare variables locally embedded within code blocks, an alternative to declaring all local variables at the top of each function where it is used. While the code below shows grouping braces {}, these braces are optional for any code block such as that shown below containing only one [semicolon-terminated] statement.

```
for (int i=0; i<=100; i+=2) {
   printf("%d,", i);
}
```

Additional examples of iteration will be illustrated in subsequent code that requires additional concepts be first introduced, specifically arrays and functions/procedures.

Arrays and Pointers

A set of quantities with like data types can be stored as a group using the notion of an array. Conceptually, the array is similar to a vector used to group numbers in mathematics, although the array doesn't have any notion of magnitude and direction. An array can be used to group any valid C (or C++) data type. Array subscripts or indices are designated in square brackets [], and indices start with 0.[4] To statically declare an array, the code must indicate as a constant number how many elements the array will have. This size must be known when the programmer writes the code. An example `add_v1.c` that adds two three-dimensional vectors is shown below.

```
// add_v1.c:  Program will calculate z = x + y
#include <stdio.h>
int main()
{
  double x[3], y[3], z[3];
  int i;  // Loop variable (must declare at top of function in pure C)

  // Ask user to enter data for x and y
  printf("Enter x values:\n");
  for (i=0; i < 3; i++) {
    scanf("%lf", (x+i)); // Notation: pointer (x+i) is the same as &(x[i]).
  }
  printf("Enter y values:\n");
  for (i=0; i < 3; i++) {
    scanf("%lf", (y+i));
  }

  // Do math and print result to the screen
  printf("Result is:\n");
  for (i=0; i < 3; i++) {
    z[i] = x[i] + y[i];
    printf("%7.3lf\t", z[i]); // (7.3) format:  7 total print digits reserved
             // (including sign and .); 3 place fractional precision (1/1000)
  }
  printf("\n");
  return 0;
}
```

Arrays can also be dynamically-allocated, meaning that array size is determined at run-time. The standard method for allocating and deleting memory in C is to use the `malloc()` and `free()` functions to reserve and delete (free) a memory block, respectively. Memory must always be reserved but not yet deleted (freed) whenever it will still be used in read or write access operations. Dynamic memory allocation is very powerful but also dangerous as the most

[4] Be careful to remember differences between C/C++ and Matlab array syntax. C array size and indices are denoted by square brackets [] to distinguish them from function arguments () and the math operator (). Matlab array subscripts are denoted by parenthesis (). Also, C array subscripts or indices begin with 0 to indicate "how many slots from the beginning of the array to move" when accessing an array element; Matlab array subscripts start with 1 to be consistent with notation used for matrices and vectors (e.g., row 1, column 1 is the upper left matrix entry).

common cause of the memory "segmentation fault", i.e., improperly accessing random access memory (RAM) space, which causes your program to crash ungracefully. Memory allocation and access errors may be hard to find depending on where memory was overwritten. Tools such as `valgrind` are highly recommended to help check for memory errors and leaks (e.g., memory is repeatedly allocated but never freed, erroneously increasing program size in memory over time). Remember that dynamic memory allocation is just that – dynamic. Observed problems therefore may vary depending on the specific data input to the program at runtime.

Below is a version of the vector addition program shown above that instead uses dynamic memory allocation for the three arrays. This version of the program allows the user to input a dimension (size) of the arrays at runtime, a feature only possible using dynamic memory allocation.[5] Note the use of "pointer variables" instead of single variables or statically-sized arrays. The pointer variable is used to indicate that the variable will directly reference a memory address rather than a single value. To address a value for a pointer variable, one can either use a pointer "dereference" * or array subscript []. For example, suppose one declared a pointer to characters with `char *carray;` then initialized it to point to an array of `n` characters with `carray=(char *)malloc(n*sizeof(char));` To access the third character of the available `(n>=3),` one can either use the statement `carray[2]` or `*(carray + 2).` Similarly, to access the first character, one can use either `*carray` or `carray[0]`.

An important concept to learn and remember: C (and C++) do not automatically initialize a variable to a particular value, so there's no guarantee a declared variable will automatically have a value such as 0. For a pointer, this is a big deal, as codes often check whether a pointer is NULL (0) before using them. Remember to initialize any variable, including pointers, that you will test later. One of the most common causes of segmentation fault is use of a pointer that tests as OK because its value is non-zero but that actually has not been properly allocated, e.g., with the `malloc` command. The second most common cause of pointer errors is accessing a memory address, e.g., beyond the end of an array, that is not allocated to that particular array. Such access errors can cause segmentation faults, or they can cause other variables that just happen to be adjacent to the array in memory to "mysteriously" assume new, erroneous values that may not crash the program.

```
// add_v2.c
```

[5] Matlab uses dynamic memory allocation all the time; you just don't notice because memory management is hidden from the user. This is a feature from a code simplicity perspective. It does limit flexibility, however, and it also encourages bad practices such as not pre-allocating memory blocks when possible. In "old languages" that did not support dynamic memory allocation, e.g., Fortran 77, programmers would statically create large memory blocks then use parts of them as needed at runtime. While this strategy worked, it was inefficient in that much more memory was typically allocated than used, and if the maximum size wasn't sufficient the code would need to be modified, recompiled, and restarted.

```
//
#include <stdio.h>
#include <stdlib.h>  // Declares malloc() and free() C functions

int main()
{
  double *x, *y, *z;
  int size, i;

  printf("Enter size:\n");
  scanf("%d", &size);

  // Dynamically allocate memory for x, y, and z
  // (double *) type casts memory block to be a "double pointer"
  // Input to malloc() is the number of memory bytes to allocate.
  // sizeof() function returns the size of the input data type.
  x = (double *)malloc(size * sizeof(double));
  y = (double *)malloc(size * sizeof(double));
  z = (double *)malloc(size * sizeof(double));

  // Ask user to enter data for x and y
  printf("Enter x values:\n");
  for (i=0; i < size; i++) {
    scanf("%lf", x+i);
  }

  printf("Enter y values:\n");
  for (i=0; i < size; i++) {
    scanf("%lf", y+i);
  }

  // Do math and print result
  printf("Result is:\n");
  for (i=0; i < size; i++) {
    z[i] = x[i] + y[i];
    printf("%7.3lf\t", z[i]);
  }
  printf("\n");

  free(x);
  free(y);
  free(z);
  return 0;
}
```

Functions and Procedures

Good code is organized into a group of compact, intuitive functions and procedures. Each manipulates input arguments and/or global variables shared by all code. A function returns a single variable as its primary result, e.g., the function `sqrt()` in `y = sqrt(x);` A procedure produces side effects, e.g., writing data to a file, printing to the screen, sending a message but returns nothing (designated as a `void` return value in C and C++). This distinction between functions and procedures is conceptually important but the syntax for functions and procedures is identical, which also allows a programmer to write a hybrid function-procedure code block that returns a variable and has side effects. Such flexibility is essential to execute code with side effects as well as returning a status of execution (`int` indicating error, no error) as is done by the `main()` function. The syntax for a function in its most general form is:
`<return type> function_name(<argument list>).`

Suppose one manually defined a function `my_sqrt` to compute the square root of a number. In code using this new function, a declaration found near the top of the code, at least before the function is first used, is required: `double my_sqrt(double);` This declaration enables the C++ compiler, which first reads and processes code sequentially, as you would read a book, to understand the form of the `my_sqrt()` function so it can check that it is used with the correct set of input arguments and return type. Declarations of functions available in the implementation of the `my_sqrt` function can then appear before or after other functions, or in a separate file that is linked with any code using this function. The implementation of this example function would appear as:

```
double my_sqrt(double x)
{
    double y;
    // code to set y to the square root of x
    return y;
}
```

There are two ways to pass variables into a function or procedure in C:

1. Pass by value (shown in the example above)
2. Pass by memory address pointer (*)

C++ offers a third method known as “pass by reference” not discussed here because it is not available in C. In pass-by-value, each variable is copied into a local memory block used only by the function into which the variable is passed. This protocol eliminates the possibility that this variable could be inadvertently changed by the function since the function only uses a “copy” of the original. The drawback to copy-by-value is that processing and memory overhead are

required to copy the variable. This overhead is trivial when only a few values are copied but can become an issue when [repeatedly] passing large data structures by value.

Pass-by-pointer is denoted in the function declaration and implementation with a star (*), e.g., `double *x`, in the argument. An equivalent and acceptable representation is an empty array, e.g., `double x[]`. When passing by pointer, the variable references a memory address inside the function. To access the memory address, one uses the variable alone; to access an element of the memory block to which the variable points, one must use the pointer dereference operator *, e.g., *x = 3.14; or an array subscript, e.g., x[0] = 3.14; Arrays should be passed by pointer to the array, as shown in examples below. Pointers to single variables (one-dimensional arrays) can also be passed by memory. When passing by pointer, the values stored in the memory block referenced by the pointer variable CAN be changed in the function as a side effect of the function with new values available to the calling function. Although a function can also allocate memory for the pointer, e.g., `x=(double *)malloc(5*sizeof(double));`, the new memory address is NOT passed back to the calling function because the pointer itself was "passed by value"; any memory allocated locally in a function (to a local variable or a pointer argument) must be deleted/freed prior to returning from this function – otherwise the memory is "leaked" which can cause problems when the function is repeatedly called (thus building up the amount of memory leaked). If a function needs to locally allocate memory to a pointer variable but then make this memory available to the calling function, a "pointer-to-the-pointer" must be passed into the function, e.g., `double **y;`. Try using pointers with functions for yourself! Pointers offer a powerful capability but one that will cause compiler errors/warnings (if you're lucky) and segmentation faults (if you're not).

The next two examples illustrate code for the numerical algorithm bisection and an algorithm binary search to find an element in a discrete list of sorted values, respectively. More information on the bisection algorithm is available at http://en.wikipedia.org/wiki/Bisection_method, while more information on binary search, including different coding options (iterative vs. recursive) is available at http://en.wikipedia.org/wiki/Binary_search_algorithm. Complete programs are shown for each. These examples also illustrate use of C functions to organize code, including how arguments are passed into functions. Further content on functions and procedures is included below.

```
// bisect.c :  Example bisection implementation
// Don't forget to compile with the math library (-lm)
//
#include <stdio.h>
#include <math.h>
#include <assert.h> // Assert function (if (not true) exit)

// Function Declarations
// Note the passing of a function reference as the last argument to bisect()
double bisect(double xmin, double xmax, double tolerance, double f(double));
double myfunc(double x);

double bisect(double xmin, double xmax, double tolerance, double f(double))
{
  double x = xmin, y = xmax, midPoint;
  assert((f(xmin) * f(xmax)) <= 0.0);

  while ((y-x) > tolerance) {
    midPoint = 0.5 * (x + y);
    if (f(x) * f(midPoint) <=0.0)
      y = midPoint;
    else
      x = midPoint;
  }
  return (0.5*(x+y));
}

// Now define a main() program and a math function (myfunc) to be "bisected"
int main()
{
  double xmin=0.0, xmax=10.0;
  double answer;

  answer = bisect(xmin, xmax, 1.0E-6, myfunc);
  printf("The function zero is at x = %lf", answer);
  printf(", f(x) = %lf\n", myfunc(answer));

  return 0;
}

double myfunc(double x)
{
  double y;
  y = exp(x) - 10;
  return y;
}
```

```
// bsearch.c:  Example binary search implementation

#include <stdio.h>

#define n 100

// Declaration of binary search function
// int A[] and *A are identically interpreted by the C/C++ compiler

int bsearch(int A[], int length, int value);

int main()
{
  int A[n], i, elem;

  // Initialize array values
  printf("A = [");
  for (i=0;i<n;i++) {
    A[i] = 2*i;
    printf("%d\t", A[i]);
  }
  printf("]\n");

  // Find a specific number in the list
  printf("Enter element to find:\n");
  scanf("%d", &elem);

  /* Binary search:  O(log_2(n)) complexity */
  i = bsearch(A, n, elem);
  if (i >= 0) printf("Found at subscript %d\n", i);
  else        printf("Element was not found.\n");

  return 0;
}

// Returns integer subscript of where value was found in A[].
// Returns -1 if value was not found.
//
int bsearch(int A[], int length, int value)
{
  int low = 0, high = length -1 , mid;
  while (low <= high) {
    mid = (low + high) / 2;
    if (A[mid] > value) high = mid - 1;
    else if (A[mid] < value) low = mid + 1;
    else                     return mid;
  }
  return -1;
}
```

Multi-dimensional arrays can also be defined and used in C (and C++). The most commonly-used is the two-dimensional array or matrix, although an array can have three or more dimensions. In C (and C++) a 2-D matrix is declared and accessed by two array subscripts, e.g., `double A[3][2];` In this example matrix `A` has three rows and two columns. An array variable with values accessed by two subscripts is the same as a double pointer. If the variable is declared with constant numbers of rows and columns, it is statically allocated. If the variable is declared as a double pointer, e.g., `double **A;`, it can be dynamically allocated once the size is known. The sketch below shows how memory is organized for a two-dimensional matrix in C. Memory is allocated in two steps: first an array of single pointers (equal to number of matrix rows) is allocated, then memory is allocated for each column. Also below is example code to dynamically allocate memory for a matrix `A`.

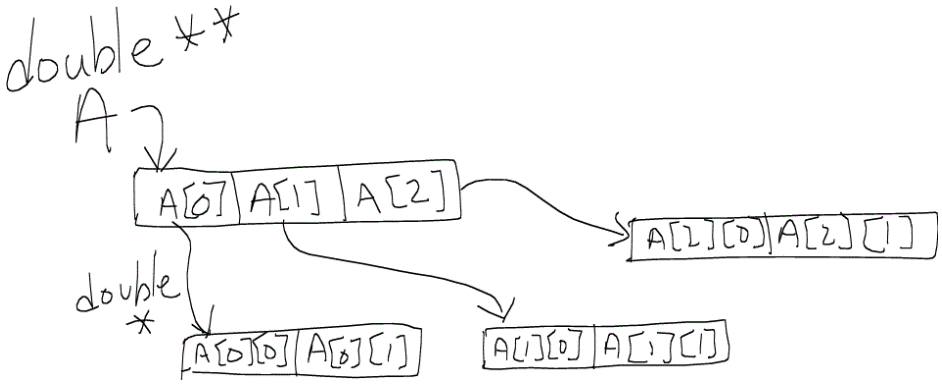


Figure 2-1: 2-D Array (Matrix) Representation in C (and C++)

```
double **A;  int i, nrows, ncols;
… // Code to determine integer numbers of rows (nrows) and columns (ncols)
A = (double **)malloc(nrows * sizeof(double *));
for (i=0; i<nrows; i++)
  A[i] = (double *)malloc(ncols * sizeof(double));
A[2][1] = 1.23;  // Sets the 3rd row, 2nd column of matrix A to value 1.23
```

While conceptually it is nice to have multi-dimensional arrays in C, they are not as efficient as single-dimensional arrays. First, row pointers take memory beyond what is necessary to store array values. Next, accessing each array value requires the computer to reference two values in memory: the row pointer first, then the appropriate value in the column memory block. A commonly-used alternative is the stacked array, which is a one-dimensional array in which the rows are "stacked" end-to-end in a single memory block. The code below revisits the example above using a stacked array rather than a 2-D matrix data representation.

```
double *A; int nrows, ncols;
… // Code to determine integer numbers of rows (nrows) and columns (ncols)
A = (double *)malloc(nrows * ncols * sizeof(double));
A[2*ncols + 1] = 1.23;  // Sets 3rd row, 2nd column of stacked array A to 1.23
```

Data Files

In C, pointer variables of type `FILE` are typically used to manage files; functions to handle files are declared in <stdio.h> so no additional header (.h) files need to be included in your code to handle C files. In any language, there are three steps to managing a file: (1) Open the file for reading ("r"), writing ("w"), or appending ("a"), (2) Execute code to read or write data from/to the file, (3) Close the file. Example C code to read data from a file `input.txt` and write the same data to file `output.txt` is shown. These examples deal with ASCII (text) file formats; one can also read/write data in binary formats that are not readable in text editors but that can be post-processed using the "fread/fwrite" and "read/write" functions (see Linux "man" pages and/or www.cplusplus.com for more information). Binary file formats only tend to be used when disk space and/or read/write speed are important factor(s). Underlying your code is the operating system which manages actual cycles of writing buffered data to disk. Note that it is possible for your code to write a substantial amount of data to a buffer which might be lost should the computer reboot or be powered down before data is actually written to disk. Use the `flush()` function in to force immediate write of data. Only use the flush functions when necessary – they can slow the computer as the operating system must immediately drop all other activities to write data to disk each time a `flush()` call is made.

```
/* myfile.c:  Simple example of file copying in C.
   This code also illustrates the use of the "block comment".
*/

#include <stdio.h>  // C input/output functions including FILEs
int main()
{
  FILE *infile, *outfile;
  char c;  // File is read char-by-char

  // Zero return value indicates file open error;
  // Error checks are included in the below code
  if ((infile = fopen("input.txt","r")) == NULL) {
    printf("Error opening input file.\n");
    return -1;
  }
  if ((outfile = fopen("output.txt","w")) == NULL) {
    printf("Error opening output file.\n");
    return -1;
  }

  while (!feof(infile)) {  // Read chars until end of file (EOF)
    fscanf(infile,"%c",&c);
    fprintf(outfile,"%c",c);
  }

  fclose(infile);
  fclose(outfile);

  return 0;
}
```

Data structures in C

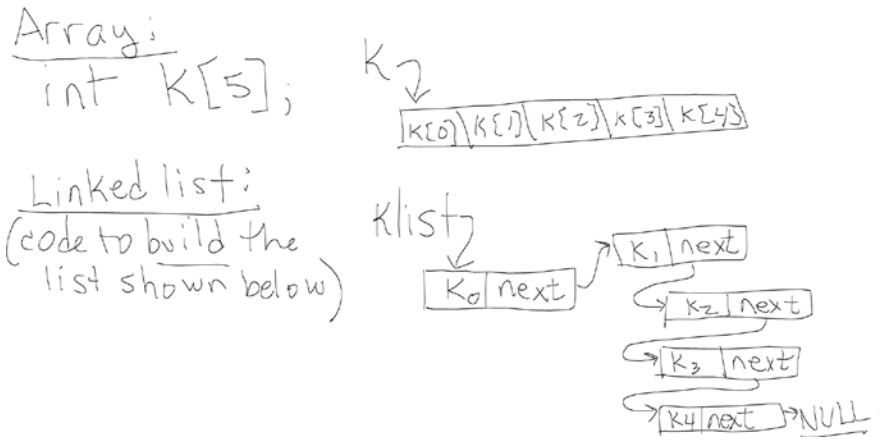
C offers the ability to group heterogeneous data types into structures using the `struct` keyword, as illustrated below. While the "struct" may have been the basis for object-oriented programming, in pure C only data can be grouped, while in C++ data and functions together form abstract objects known as "classes".

```
// Code excerpt showing definition of a data structure in C
struct student {
  char first_name[20], last_name[20];
  int studentID;
  double hw_grades[6], exam_grades[2], project_grade;
  char course_grade;
};
// Example use of the student struct
struct student s[15];  // Statically declares an array of 15 student structs
s[2].hw_grade[0] = 94;  // Assigns third student 94% for the first homework
```

Below are classic examples of how data structures can be utilized to improve data storage and access in C or any other procedural or object-oriented language including C++.

Linked Lists

Arrays are the most efficient way of representing multiple instances of a particular data type, and dynamic memory allocation offers a way to build and resize arrays as needed. However, in cases where data instances are incrementally added and/or deleted, overhead of managing memory and copying values can become substantial, particularly with large datasets. Linked lists offer an important alternative means of collecting multiple instances of data into a single collection of structures. In a linked list, a top-level "root" data structure (`struct`) is defined that contains the data type needing to be represented plus a "link" or "pointer" to any additional elements in the list. Below is a graphical representation of memory used for an array of five integers vs. a linked list containing five integer values. In the linked list, memory is allocated for each new element in the list as it is added, but the existing list elements need not be re-allocated or copied. It is critical to initialize linked list pointers to NULL, which then can be used as a test for the end of the linked list.



A C program to enter integers (one by one) in a dynamically-expanding array is shown below, followed by equivalent code using a linked list. The linked list implementation is more time-efficient because only one new `struct` must be allocated and initialized for each iteration. The below example illustrates incrementally building an array (which is very inefficient) followed by incrementally building and storing the same data in memory using a singly-linked list. Note that a list with a `*next` and a `*previous` pointer can be traversed forward or backward thus is called doubly-linked.

```
// badarray.c
//
// Array version:  user-entered data set of arbitrary length
// [Not efficient!!!]
// Note how array is recopied again and again;
// This is what programs such as Matlab do when you
// incrementally increase array/matrix size; the way around this
// is to preallocate the full array before you populate it if
// you use arrays and know their size in advance.  If you don't know
// array size in advance linked lists are way better than arrays as
//  a data structure.

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

int main()
{
  int *k, *kcopy;
  int blah, size=1, i;

  printf("Enter the first list element:\n");
  k = (int *)malloc(sizeof(int));
  scanf("%d", k); // k is already a pointer - no & required

  while (1) {
```

```
    printf("Enter next list element (-9999 to exit):\n");
    scanf("%d", &blah);
    if (blah == -9999) break;
    kcopy = k;
    k = (int *)malloc((size+1)*sizeof(int));
    for (i=0; i<size; i++)
      k[i] = kcopy[i];
    free(kcopy);
    k[size++] = blah;
  }

  printf("The final list:\n");
  for (i=0;i<size;i++)
    printf("%d, ", k[i]);
  printf("\n");
  return 0;
}
```

```
// llist.c
// Program illustrating use of an incrementally-constructed
// linked list to store user-entered data.
// Note that the only extra data copied each time the user enters
// a new number is a single linked list pointer *next.

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

struct intlist  // Linked list data structure
{
  int k;
  struct intlist *next;
};

int main()
{
  struct intlist klist, *kptr;
  int blah;

  printf("Enter the first list element:\n");
  scanf("%d", &(klist.k));
  klist.next = NULL;
  kptr = &klist;

  while (1) {
    printf("Enter next list element (-9999 to exit):\n");
    scanf("%d", &blah);
    if (blah == -9999) break;
    kptr->next = (struct intlist *)malloc(sizeof(struct intlist));
```

```
    kptr = kptr->next;
    kptr->k = blah;
    kptr->next = NULL;  // Marks end of linked list
  }

  printf("The final list:\n");
  kptr = &klist;
  while (kptr != NULL) {
    printf("%d, ", kptr->k);
    kptr = kptr->next;
  }
  printf("\n");

  return 0;
}
```

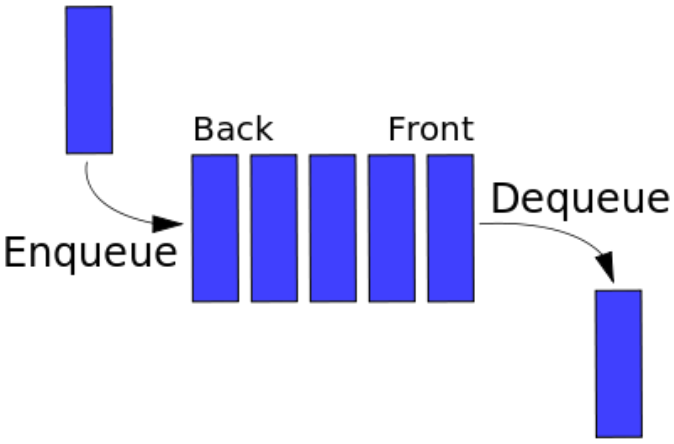
Stacks and Queues

Data describing a set of computing goals, tasks, or jobs (e.g., for a printer) typically arrive and dispatch asynchronously. In the real world, a motivating example is the line one follows through airport security. One "arrives" when entering this line, and is "dispatched" once airport security (TSA) indicates you are cleared to enter the gate area. Storing such data as a linked list is efficient since individual data can be added (upon arrival) or deleted (following dispatch or retrieval) without manipulating the remainder of the stored data.

Two basic orderings are possible when dealing with a series of asynchronously-arriving tasks:

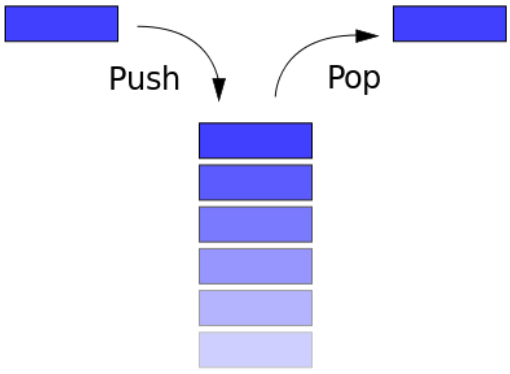
1. First-in-first-out (FIFO)
2. Last-in-first-out (LIFO)

The FIFO ordering is typically what we consider a "fair policy" for handling lines such as those found at airport security. Such lines are often called "queues". The data structure `queue` stores tasks such that they are dispatched in FIFO order. Two basic operations are possible: `enqueue` which places new tasks (one at a time) on the queue, and `dequeue`, which retrieves or dispatches single tasks. A graphic describing the `queue` data structure is shown below.



The `queue` Data Structure (http://en.wikipedia.org/wiki/Queue_(abstract_data_type))

Although a LIFO ordering seems unfair in that the job arriving last is actually retrieved first, it can be quite useful from an optimization standpoint. Consider a stack of papers on your desk. Perhaps the most useful one is on or near the top because you reference it frequently. If you don't clean your desk often, the bottom paper on this stack can be quite old and potentially useless. It is typically most efficient to indeed store the last-referenced paper on top of a stack of papers on your desk, as this paper will be easiest to find later. The data structure `stack` stores tasks such that they are dispatched in LIFO order. Two basic operations are possible: `push` which places new tasks, one at a time, on the stack, and `pop`, which retrieves or dispatches single tasks from the top of the stack. A graphic describing the `stack` data structure is shown below.



The Stack Data Structure (http://en.wikipedia.org/wiki/Stack_(abstract_data_type))

Example code for handling stacks and queues in C with a single "integer" as data content is shown below. Of course the data content (or payload) could be much more extensive than a single integer.

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

struct stack
{
  int element;  // Data "payload"
  struct stack* next;  // Linked list pointer
};

struct stack* push(struct stack* top,int key)
{
  struct stack* temp;
  if(top==NULL) {
    top=(struct stack *)malloc(sizeof(struct stack));
    top->element=key;
    top->next=NULL;
    return top;
  } else {
    temp=(struct stack *)malloc(sizeof(struct stack));
    temp->element=key;
    temp->next = top;
    return temp;
  }
}

struct stack* pop(struct stack* top)
{
  struct stack* temp;
  if(top==NULL)    {
    printf("\nit is impossible to pop an element - stack is empty ");
    return NULL;
  }

  printf("\nthe element popped from the stack is %d\n",
         top->element);
  temp = top->next;
  free(top);
  return temp;
}

void stack_display(struct stack* top) {
  printf("\nthe elements of stack are \n");
  if(top!=NULL) {
    while(top->next!=NULL) {
      printf("%d->", top->element);
```

```
      top=top->next;
    }
    printf("%d\n",top->element);
  }
  else
    printf("the stack is empty\n");
}

int main()
{
  int key,ch;
  struct stack *top=NULL;

  printf("\nchoose the operation\n");
  printf("\n1.push\t2.pop\t3.exit\n\n");
  scanf("%d", &ch);
  while(ch!=3) {
    switch(ch) {
    case 1:
      printf("\nenter the key to be inserted\n");
      scanf("%d", &key);
      top=push(top,key);
      stack_display(top);
      break;
    case 2:
      top=pop(top);
      stack_display(top);
      break;
    case 3:
      break;
    default:
      printf("\nenter correct choice\n");
      break;
    }
    printf("\n.........................\n");
    printf("\nchoose the operation\n");
    printf("\n1.push\t2.pop\t3.exit\n\n");
    scanf("%d", &ch);
    printf("\n.........................\n");
  }
  return 0;
}



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

```
struct queue  // C-style struct declaration (compatible with C++ or C)
{
  int element;
  struct queue* next;
};

struct queue* enqueue(struct queue* head,int key)
{
  struct queue* temp=head;
  if(head==NULL) {
    head=(struct queue *)malloc(sizeof(struct queue));
    head->element=key;
    head->next=NULL;
  } else {
    while (temp->next) temp = temp->next;
    temp->next=(struct queue *)malloc(sizeof(struct queue));
    temp->next->element=key;
    temp->next->next=NULL;
  }
  return head;
}

struct queue* dequeue(struct queue* head)
{
  struct queue* temp;
  if(head==NULL)    {
    printf("\nit is impossible to dequeue an element - queue is empty ");
    return NULL;
  }

  printf("\nthe element dequeued from the queue is %d\n", head->element);
  temp = head->next;
  free(head);
  return temp;
}

void queue_display(struct queue* head) {
  printf("\nthe elements of queue are \n");
  if(head!=NULL) {
    while(head->next!=NULL) {
      printf("%d->", head->element);
      head=head->next;
    }
    printf("%d\n", head->element);
  }
  else
    printf("the queue is empty\n");
}
```

```
int main()
{
  int key,ch;
  struct queue *head=NULL;

  printf("\nchoose the operation\n");
  printf("\n1.enqueue\t2.dequeue\t3.exit\n\n");
  scanf("%d", &ch);
  while(ch!=3) {
    switch(ch) {
    case 1:
      printf("\nenter the key to be inserted\n");
      scanf("%d", &key);
      head=enqueue(head,key);
      queue_display(head);
      break;
    case 2:
      head=dequeue(head);
      queue_display(head);
      break;
    case 3:
      break;
    default:
      printf("\nenter correct choice\n");
      break;
    }
    printf("\n.........................\n");
    printf("\nchoose the operation\n");
    printf("\n1.enqueue\t2.dequeue\t3.exit\n\n");
    scanf("%d", &ch);
    printf("\n.........................\n");
  }
  return 0;
}
```