Just enough C for 243

C is quirky, flawed, and an enormous success.
— Dennis Ritchie, co-designer of C

A C program is like a fast dance on a newly waxed dance floor by people carrying razors.
— Waldi Ravens, NetBSD developer

This guide summarizes the C programming language as used in EEE243 *Applied Programming* at the Royal Military College of Canada. It's more of a quick reference than a tutorial, it deliberately leaves things out, and it occasionally over-simplifies, but it's enough to get you started.

The guide occasionally refers to Joseph H. Silverman's *C Reference Card*, from `www.math.brown.edu/~jhs/`. The *Card* provides more breadth but less explanation. For further information, see the course text or the Wikipedia articles on C and its syntax.

In this guide C code is written `like this`. Anything written *like this* in a code sample would need to be replaced with real C code in a program.

Contents

C program structure

A C program will have a *main program file* and will almost always include one or more *modules* or *libraries.*

main program file

Exactly one file in a program (whose name ends in `.c`) must have a function called `main`, which is the initial point of execution. The normal structure of a C program's main file is:

```
includes of header files required by this program
other preprocessor directives

global declarations, including prototypes for
   functions defined later in this file

int main(void) {
   local declarations
   statements
   return EXIT_SUCCESS;
}

function definitions
```

modules or libraries

There are two kinds of modules: those you and your team write as part of your program, and those provided to you. Some modules are part of the standard C distribution; others come from vendors or other third parties. A C module consists of a *header file*, whose name ends in `.h`, and an *implementation file* whose name ends in `.c`. The first part of the name must match. The implementation file may be pre-compiled and distributed in object code form.

header files

A header file contains declarations of functions and constants that are intended to be reused across multiple programs. It is always distributed in source code form. A header file is brought into your program using the `#include` directive.

module implementation files

Module implementation files provide the definitions of all functions declared in their corresponding header files. Every module is compiled separately; the resulting *object files* are combined by the *linker* to produce your program. Third party modules are often provided in pre-compiled form.

Preprocessor directives

Preprocessor directives run prior to compilation and modify the text of the program itself.

#include

The `#include` directive requires a file name as an argument. This will always be a C header file. The preprocessor inserts the contents of the header file at the point where the `#include` occurs. There are two forms:

`#include <` *filename.h* `>`	look for a file called *filename.h* in the system-defined library directories
`#include "` *filename.h* `"`	look for a file called *filename.h* in the same directory as the current file

#define

The `#define` directive is used for *replacement text*, typically *constants* (it can also define *replacement macros*, which we will not use in the course). For example

```
#define MAX_HEADROOM 2.3
```

will cause every occurrence of the characters `MAX_HEADROOM` in the program text to be replaced with the characters `2.3`

#ifdef and #ifndef

The `#ifdef` and `#ifndef` directives allow for conditional compilation depending on whether a symbol has been defined using `#define`. A typical use of `#ifndef` is in header files, to prevent a header's contents from being included multiple times (which could result in multiple declarations of the same function). This is how the pattern would be used in a header file named `schroedinger.h`

```
#ifndef SCHROEDINGER_H_
#define SCHROEDINGER_H_
      code to be included exactly once
#endif
```

If the symbol `SCHROEDINGER_H_` is not defined, then define it and include everything up to `#endif`. If this header is included a second time, `SCHROEDINGER_H_` will be defined, so the body of the `#ifndef` will be skipped. This assumes the symbol `SCHROEDINGER_H_` is globally unique in the program.

Basic types

The C programming model is very close to raw hardware. Values are stored as bit sequences. Types define the number of bits a value occupies in memory and how those bits should be interpreted.

In C, the bit size (and therefore range and precision) of the primitive types depends on both the underlying hardware and the compiler implementation. Srsly.

integer types (including bool and char)

`int`	signed integer, usually 16 or 32 bits — the default type for an integer literal
`long`	signed integer, usually longer than an `int`
`char`	a single ASCII character, normally 8 bits; yes, `char` is a kind of integer
`bool`	a truth value, either `1` (meaning `true`) or `0` (meaning `false`). `bool`, `true` and `false` are not actually part of the C language and are defined in the library `stdbool.h`

unsigned integers

A signed integer of *n* bits can represent numbers from -2^{n-1} to $2^{n-1}-1$. So if an `int` is 16 bits long, a signed `int` can represent values from -32768 to 32767. If negative values are not required, the `unsigned` keyword can be added to the declaration to give us a larger positive range. A 16 bit `unsigned int` permits us to store values from 0 to 65535.

floating point types

`float`	signed floating point number, normally IEEE 754 single-precision (32 bits)
`double`	signed floating point number, normally IEEE 754 double-precision (64 bits) — the default type for a floating point literal

void

The C type `void` has two meanings. When used as the return type of a function, `void` means that the function does not return a value. A `void` pointer is a pointer to a value of undefined type, so a void pointer is a "universal pointer."

Arrays

A C *array* is a fixed-size region of memory that stores a sequence of values of some base type. The base type may be either a primitive type or a *derived type* (array, pointer, `enum` or `struct`). All elements must have the same type. Array types are indicated by square brackets `[]`. See **literal arrays**, **variable and parameter declarations**, and **array index expression**.

Pointers

A *pointer* represents the location in memory at which a value is stored. Pointers are themselves values: a pointer is a special integer type whose size is equal to the hardware address length.

For examples of how to declare a pointer type, see **Variable and parameter declarations**. For examples of how to use pointers, see **pointer operators** and **pointer arithmetic**.

NULL

`NULL` is a `void` pointer to memory location zero. This should not be confused with the null character `'\0'` (see **escape sequence character literals**). Both have value zero, but `'\0'` is a `char` (one byte wide) and `NULL` is a pointer (typically eight bytes wide).

Values and literal representations

numbers

`42`	integer (exactly 42) of type `int`
`42L`	integer of type `long`
`42LU`	integer of type `unsigned long`
`42.7`	floating point number (approximately 42.7) of type `double`
`42.7F`	floating point number of type `float`
`4.275e2`	exponential notation: equivalent to `427.5`

truth values

Include the library `stdbool.h` in your program to use the literal values `true` and `false` and the type `bool`.

In C anything that evaluates to zero is considered false, and anything non-zero is considered true.

characters

`'c'`	single quotes denote characters; only characters in the ASCII range may be used

escape sequence character literals

The following "escape sequences" may be used in character or string literals. There are others; these are the ones you will need most frequently.

`\n`	new line (in output, advances to the next line)
`\"`	double quote `"`
`\'`	single quote `'`
`\\`	backslash `\`
`\0`	the null character, used to terminate strings; ASCII value `0`. Not to be confused with `NULL`
`%%`	a literal `%` character, needed only in `printf` format strings — elsewhere, a simple `%` is fine

array literals

Array literals are written with the array values enclosed in curly braces and separated by commas and may be used *only* in array declaration expressions. Array literals may be nested.

`{ 1, 42, -3 }`	a three-element array of `int`
`{ { 1, 42, -3 }, { 12, 99, 14 } }`	a two-element array, each element of which is a three element `int` array
`{ 1, 42, -3, 12, 99, 14 }`	because C stores arrays in row-major order, this is exactly equivalent to the previous expression

character strings

C uses null-terminated arrays of characters to represent strings. Literal strings are written with double quotes.

`"Hello"`	equvalent to the character array `{ 'H', 'e', 'l', 'l', 'o', '\0' }` (note the termimating null character)

Identifers (names)

Identifiers in C are used to name variables, functions, function parameters, and types declared by `typedef`, `struct`, and `enum`.

A name must begin with a letter or an underscore, which may be followed by any combination of letters, digits, and underscores. Uppercase and lowercase are different, so `foo` and `FOO` and `Foo` are three different names. Names may be any length; however, only the first 31 characters are guaranteed to be significant.

C forbids use of these *reserved words* for names: `auto`, `break`, `case`, `char`, `const`, `continue`, `default`, `do`, `double`, `else`, `entry`, `enum`, `extern`, `float`, `for`, `goto`, `if`, `int`, `long`, `register`, `return`, `short`, `signed`, `sizeof`, `static`, `struct`, `switch`, `typedef`, `union`, `unsigned`, `void`, `volatile`, and `while`.

Variable and parameter declarations

All variables and function parameters (see **Function declarations**) in C programs must be *declared* before use. A declaration reserves space in memory for the variable or parameter, sufficient to hold a value of the declared type. Variable declarations may include initializations (see **assignment**) but parameter declarations may not. Here are some examples:

`int x`	the variable called `x` is of type `int`
`int x = 5`	`x` is of type int and is intialized to 5
`int a[4]`	`a` is an array of four `int` values. See the section on **array and string expressions**.
`int a[]`	`a` is an array of `int`, length unknown. If this is a variable declaration, `a` must also be initialized here in order to determine its size; if `a` is a function parameter, this is equivalent to `int *a`
`int *p`	`p` is a pointer to a value of type `int`. See the sections on **Pointers**, **pointer operations**, and **pointer arithmetic**
`int *p[10]`	`p` is an array of 10 pointers to `int` values
`int (*p)[10]`	`p` is a pointer to an array of 10 `int` values (this is different from the previous declaration)
`char *s`	`s` is a pointer to a value of type `char`. This is the type of C string values.
`Bork b`	`b` is of type `Bork`. Since `Bork` is not a standard C type, it must have been defined previously in the program via a `typedef`; see the section on **typedef**.

const

The prefix `const` may be used to declare that a variable or function parameter is a constant value. If a declared constant is a variable, it must be initialized at the point of declaration.

```
const int x = 37
```

`x` is an integer constant with the value 37. Any attempt to modify the value of `x` will be flagged as illegal by the compiler.

static

See the section on **static variables** in the **Functions** section.

enum definition and use

The `enum` keyword can be used to define an *enumerated type,* a type that has a fixed set of named values. The syntax is

`enum` *tag* `{` *values* `};`

where *tag* is an identifier and *values* is a comma-separated list of identifiers. For example:

```
enum AccountKind { SAVINGS, CHEQUING, CREDIT };
```

defines a new type, `enum AccountKind`, with the three legal values

To declare a variable of an enumerated type we use the full type name, in this case `enum Account`. For example:

```
enum AccountKind new_account = CREDIT;
```

declares a variable `new_account` to be of type `enum AccountKind` and gives it an initial value of `CREDIT`

Each of the values in an enumerated type is treated as an integer, starting from zero. So, in our example, `CREDIT` is actually the integer 2. Always use the value names when working with `enum` types, never the integer equivalents — otherwise you defeat the purpose of having them.

struct definition and use

Structures allow us to group related values into a named unit. The syntax is

`struct` *tag* `{` *fields* `};`

where *tag* is an identifier and the *field-list* is a semicolon-separated list of fields, each of which takes the form of a variable declaration, and may be of any defined type, including `struct` or enumerated types. For example:

```
  struct Client {
    char name[30];
    int client_number;
    enum AccountKind account;
  };
```

defines a new type, `struct Client`, with three fields called `name`, `client_number`, and `account`. Note the trailing semicolon.

To declare a variable of a `struct` type we use the full type name, in this case `struct Client`. For example

```
struct Client bob;
```

declares a variable `bob` to be of type `struct Client`

We can assign all values of a `struct` at once, but *only* on initialization.

```
struct Client bob = {"Bob Smith", 25, SAVINGS};
```

Anywhere else, we must access each field separately, using *dot notation*, like this

```
bob.account = CHEQUING;
```

typedef definition and use

The `typedef` keyword can be used to give an alternate name to a type. The syntax is

`typedef` *type new-name* `;`

The *type* can be any C type and the *new-name* can be any valid C identifer. For example:

```
  typedef int Pounds;
```

defines the new type name `Pounds`, which has the same meaning as `int` and can be used anywhere `int` can be used

```
  typedef struct Client BankClient;
```

defines a new type name `BankClient` with the same meaning as `struct Client`

Types defined via `typedef` can be used anywhere a type is expected.

It is common to combine `typedef` with `enum` or `struct` definition. The syntax is the same, with the `enum` or `struct` definition appearing as the *type*. In this case, the *tag* is optional. For example

```
    typedef struct ClientTag {
        char name[30];
        int client_number;
        enum AccountKind account;
    } Client ;
```

Expressions

In C, an expression is anything that has a value (possibly void) when evaluated. Expressions may include variables, literal values, operators, and function calls. Expressions are *evaluated left to right,* according to *operator precedence rules,* and ultimately result in a single value.

operators and precedence

For a list of operators and their precedence, see the *C Reference Card.* Normal arithmetic operators have their usual precedence, so the expression `6+3*-7` evaluates to `-15` and is equivalent to `6+(3*(-7))`. However, C operator precedence is *tricky.* If you're in doubt, add parentheses `(...)` to enforce order, but try not to overdo it. Explanations of some of the less common and less obvious operators are provided below.

assignment (=)

Modifies the value of a variable.

var = *expression*

Evaluates *expression* and replaces the contents of *var* with the result. *var* must be a declared variable or an array index expression and it must be possible to coerce the type of *expression* to match the declared type of *var.* The value of an assignment is the value of its right-hand side.

compound assignment (+=, *= ...)

Compound assignment is a short form for the frequent case of updating a variable by performing some operation on its current value. C provides compound assignment forms for the operators +, -, *, /, and `%` as well as the bitwise operators.

`x *= 3 + 5` is equivalent to `x = x * (3 + 5)`

modulus

The modulus operator `%` applies to integers only and evaluates to the remainder after integer division.

`27 % 7`	6, the remainder after dividing 27 by 7

integer division

Dividing two integers gives an integer result, which is truncated, not rounded. So `2/3` evaluates to `0`. Floating point division works as expected and can be forced by making either the dividend or the divisor floating point, so `2.0/3` is approximately 0.6666667.

increment and decrement (++, --)

C provides operators to increment (++) and decrement (`--`) variables by one. If the increment/decrement appears to the left of the variable in an expression, the operation is performed before the expression is evaluated. If it appears to the right, the operation is performed afterwards. If `x`, `y`, and `z` are integer variables, the code in the two columns below is equivalent:

```
y = x++;        y = x;
                x = x + 1;

z = ++x;        x = x + 1;
                z = x;
```

comparison

The C comparison operators evaluate to `1` if the condition is true and `0` if the condition is false. Be careful not to mistakenly use = (assignment) when you mean == (equality comparison).

==	is equal to (`a == b` is true if `a` is less than `b`)
!=	is not equal to
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to

Since the comparison operators have precedence after the arithmetic operators, `5 < 4 + 3` is true, since the `4 + 3` is evaluated first.

Comparisons operate on two values at a time. So, perhaps surprisingly, `5 > 4 > 3` evaluates to `0` (false) since `5 > 4` evaluates to `1` and `1 > 3` is false. If you need to compare multiple values you must use the Boolean operators.

Boolean operators (&&, ||, !)

The Boolean operators support complex truth value expressions. Watch out for the `&` and `|` bitwise operators! Using `&` when you mean `&&` is a frequent cause of error.

!	logical NOT or "negation." `!a` is true if `a` is not true; `!a` is false if `a` is not false
&&	logical AND. `a && b` is true if `a` is true and `b` is also true
`||`	logical OR. `a || b` is true if `a` is true or if `b` is true, including the case where both are true

The correct way to write the (legal but misleading) expression `5 > 4 > 3` is `5 > 4 && 4 > 3` which means "5 is greater than 4 AND 4 is greater than three" – which evaluates to `1` (true).

The `&&` and `||` operators are both "short circuiting," which means that they stop evaluating as soon as the truth value of the whole expression can be known. So in `a && b`, if `a` is false then `b` will never be evaluated. Similarly, in `a || b`, if `a` is true then `b` will never be evaluated. This is important for expressions that cause side effects when evaluated.

array index expression

The individual elements of arrays (including strings) can be accessed by *indexing*, which is written *expression*`[`*index*`]` where *expression* must evaluate to a pointer. (An array name is a constant pointer to the first value of the array.)

The index of the first element in an array is `0`. If the index refers to a position that doesn't exist in the matrix, C does not report an error but happily returns the value stored at the indicated memory location.

In the following examples, assume the initialization

```
int v[3] =  { 4, 5, 6 };
char *s = "abcde";
int m[2][3] =  { { 7, 8, 9 }, { 10, 11, 12 } };
```

`v[1]`	is the second element of `v`, which is `5`.
`v[4]`	whatever happens to be located in memory four elements past the start of `v`, which is generally garbage since `v` has no fifth element. `imdb.com/title/tt0119116/`
`s[1]`	the second element of `s`, which is `'b'`.
`m[0]`	a pointer to the first element of `m`, which is the array `{ 7, 8, 9 }`
`m[0][2]`	the third element of the first element of `m`, which is `9`
`m[0][4]`	*probably* the fifth element of the first element of `m`, which is `11` (actually part of the second element of `m`)

pointer operators (address and dereference)

A pointer represents the location in memory at which a value is stored. A pointer is also a value stored in memory: it is a special kind of `unsigned` integer, often the same size as a `long`.

There are two operators used with pointers: the *address* operator `&`, and the *dereference* operator `*`.

If `x` is a variable, `&x` is "the address at which `x` is stored."

If `p` is a pointer variable, then `*p` is "the value stored at the address pointed to by `p`, interpreted according to the base type of `p`." **Warning**: if `p` is `NULL` or points to other memory not controlled by the program, attempting to evaluate `*p` will cause the program to crash!

In the example below assume that on this machine, `int` are four bytes (32 bits) wide and pointers are eight bytes (64 bits) wide.

`int x = 5;`	`x` is declared as an `int` and initialized to hold the value 5. Assume that `x` is stored in the four bytes starting at memory location 100.
`int *p;`	`p` is a declared as a pointer to `int`. Assume that `p` is stored in the eight bytes starting at memory location 104.
`int **pp;`	`pp` is a pointer to a pointer to `int`. Assume that `pp` is stored in the eight bytes starting at memory location 112.
`p = &x;`	`p` is assigned the value `100`, the address of `x`
`pp = &p;`	`pp` is assigned the value `104`, the address of p

After this initialization, the expressions below will have the values and types indicated:

`x`	`5 (int)`
`p`	`100` (pointer to `int` or `int *`), the address of `x`
`*p`	`5 (int)` the value at the location pointed to by `p`, interpreted as an `int`
`pp`	`104` (pointer to pointer to `int` or `int **`), the address of `p`
`*pp`	`100` (pointer to `int` or `int *`), the value at the location pointed to by `pp`, interpreted as a pointer to `int`
`**pp`	`5 (int)`, the value, interpreted as an `int`, stored at the value (interpreted as a pointer) that is pointed to by `pp`. Whew!

pointer arithmetic

It is legal to add an integer to a pointer and to subtract an integer from a pointer. The integer is multiplied by the width of the pointer's base type before being added. For example, if `p` is a pointer to `int` with the value `100`, and the `int` type is four bytes wide, then `p+1` has the value `104` and `p-3` has the value `88`.

function call expression

The value of a called function is its returned value, which will match the function's declared return type (possibly `void`). When a function is called, the caller must provide actual values that are type-compatible with each of the function's declared parameters.

expr `(` *actual1*`,` *actual2*`,` ...`)`

expr will normally be a function name, but can be any expression that evaluates to a function. Evaluates *actual1*, *actual2*, etc, in order, then causess the function the be executed, substituting the values of the actuals for the function's parameters.

The C standard library provides a large number of built-in functions; see the *C Reference Card* for a partial list.

ternary conditional expression (a ? b : c)

condition `?` *expression1* `:` *expression2*

Evaluates *condition.* If *condition* is true, evaluates *expression1*, which becomes the expression's value. Otherwise, evaluates *expression2*, and that becomes the value.

For example, assuming `x`, `y` and `z` are all declared as `int`...

```
x = y > z ? y : z ;
```

is equivalent to

```
if (y > z) {
  x = y;
} else {
  x = z;
}
```

Statements

expression statements

Any expression can be made into a statement by appending a semicolon. The course coding standard requires one expression statement per line; however, the compiler will allow multiple statements on the same line.

All of the following are legal C statements, assuming the variables and functions shown are appropriately declared. However, the last statement has no effect and will be flagged with a warning by most compilers.

```
a += 3;
printf("%d\n", a);
a * 97;
```

block statement (sequence)

An opening curly brace, a series of statements, and a closing curly brace is considered a single statement. The contained statements are executed in order.

```
{
  statement
  statement statement
}
```

if and if-else statements (selection)

Allow different statements to be executed depending on the truth value of a given *condition.*

```
if ( condition )
  statement
```

If *condition* is `true`, execute *statement.* Otherwise, skip *statement.*

```
if (condition)
  true-statement
else
  false-statement
```

If *condition* is `true`, execute *true-statement.* Otherwise, execute *false-statement.*

The course coding standard requires that the contained *statement*(s) in an `if` are **block statements** with opening and closing curly braces. For example (assume all variables are declared)

```
if (balance >= request) {
  balance = balance - request;
  printf("Here is your money\n");
} else {
  printf("You don't have that much\n");
}
```

switch statement (selection)

A generalized `if`. In an `if`, the condition expression is interpreted as a truth value and there are only two possible branches: true and false. In a `switch`, the expression may be any **integer type** and there can be one branch for each possible value of the type. The general form of `switch` is:

```
switch ( expression ) {
  case label1 :
    series of statements
    break;
  case label2 :
    series of statements
    break;
  (other cases as needed)
 default :
    series of statements
}
```

The *expression* must evaluate to an integer type. Each of the `case` *labels* must be a constant integer different from all other *labels.* Inside each `case` we can have a *series of statements* — which is **weird**, because in all other C control structures the body must consist of a single statement (which can be a block statement using curly braces). Each `case` will normally end with a `break`.

When execution arrives at a `switch`, the *expression* is evaluated. Then each *label* is compared against the *expression's* value. If there is a match, the *statements* inside the matching `case` are executed; if there is no match and the optional `default` case is present, the `default` *statements* are executed.

If the *statements* end with `break`, control transfers to the point just after the `switch` statement's closing curly brace.

If the *statements* do not end with a `break`, control "falls through" to the *statements* in the next `case`, and execution proceeds until either a `break` is encountered or the `switch` ends. **Leaving out `break` is almost always an error!**

Here is an example. Assume `input` has type `char.`

```
switch (input) {
case 'a':
    printf("handle a ");
case 'b':
    printf("handle b ");
    break;
default:
    printf("handle all others\n");
}
```

If `input` has the value `'a'`, this will display "handle a handle b", since `case 'a'` contains no `break`.

while statement (loop)

Allows a *statement* (in this course, a block statement) to be executed repeatedly, as long as a *condition* remains true.

```
while (condition)
  statement
```

If *condition* is false, skip *statement.* If *condition* is true, execute *statement* once, then re-evaluate *condition* and proceed as indicated above, repeatedly executing *statement* then re-evaluating *condition* until *condition* becomes false.

do-while statement (loop)

Operates just like the `while` statement, except that the *condition* is checked after the first execution of contained *statement*, not before.

```
do
  statement
while (condition);
```

Execute *statement.* Evaluate *condition*, and if it is true, execute *statement* again and re-evaluate *condition.* Repeat this process until *condition* evaluates to false. Note the closing semicolon after the *condition.*

for statement (loop)

Allows a *statement* (in this course, a block statement) to be executed repeatedly as long as a *condition* remains true. Provides an *initialization* and a post-statement *action.*

```
for ( initialization ; condition ; action )
  statement
```

Execute *initialization.* Evaluate *condition*, and if it is true, execute *statement,* execute *action*, and re-evaluate *condition.* Repeat this process until *condition* evaluates to false.

Example:

```
int sum = 0;
int prod = 1;
for (int val = 1; val < 10; val += 2) {
   sum += val;
```

```
    prod *= val;
  }
```

After executing this code, sum will have the value 25, which is the sum of the numbers 1, 3, 5, 7 and 9, and prod will have the value 945, the product of those numbers.

break

Executing a `break` statement causes control to exit from the nearest enclosing loop (`while`, `do-while` or `for`) and proceed to the next statement following the loop. It is also used with a slightly different meaning in the `switch` statement.

continue

Executing a `continue` statement causes execution to skip over any remaining statements in the nearest enclosing loop's body, exactly as if control had passed to the point immediately prior to the loop body's closing curly brace. In `while` and `do-while` loops this causes the loop condition to be re-tested. In `for` loops this causes the loop action to be executed, then the condition to be tested.

Functions

A *function* is a computational element that takes zero or more *parameters* and optionally returns a value. Function types are indicated using parentheses `( )`.

function prototype (declaration)

All functions in C programs must be *declared* before use. A function declaration, called a prototype, states that a function with a given name, parameter types and return type exists, but does not define the function. A **function definition** implicitly declares a function, but it is normally preferable to separate declaration from definition. A function prototype has the form:

return-type function-name `(` *parameter-list* `);`

The *return-type* may be any C type. If the function returns no value, the return type is `void`. The *function-name* must be a legal identifer. The *parameter-list* is a comma-separated list of parameters, each of which has the form of a variable declaration. Here are some examples:

`int next(void);`	`next` is a function that takes no parameters (`void`) and returns an `int`
`int multiply(int x, int y);`	`multiply` is a function that takes two `int` parameters called `x` and `y` and returns an `int`
`int multiply(int, int);`	a legal but discouraged equivalent to the previous declaration; don't do this
`void output(char *s);`	`output` is a function that takes a pointer to a `char` (i.e., a string) and doesn't return a value
`void output(const char *s);`	as above, but any attempt in the body of `output` to modify the string pointed to by `s` will be flagged as illegal by the compiler; guarantees to callers that the function `output` will not modify the string pointed to by `s`

function definition

A function definition provides the code that implements a function. It has the form

return-type function-name `(` *parameter-list* `) {`
  local variable declarations
  statements
`}`

If the function has been previously declared, the first line must match the prototype, including the number, order, and types of parameters. It is legal to use different parameter names in a function prototype and its definition, or to omit the names entirely in the prototype, but this is bad practice and not acceptable in this course.

Any variables declared in the function are *local:* visible only within the body of the function. Each time the function is called, local variables are re-allocated and re-initialized. The only exception are **static variables**, see below.

The function executes until either a `return` statement is encountered (see below) or the end of the function body is reached.

return statement

The `return` statement causes control to return from a function to the place from which it was called. There may be multiple `return` statements in a function.

If the function's declared return type is `void`, then the statement must have the form

```
return;
```

It is not necessary to provide a `return` statement at the end of a `void` function.

If the function's return type is anything other than `void` then the form

`return` *value* `;`

must be used, and *value* must be an expression whose type is compatible with the declared return type.

static variables

The prefix `static` may be used to declare that a local variable in a function retains its value for the duration of the program, rather than the duration of the function call. Static variables are initialized at compile time. The following function will return `101` the first time it is called, `102` the second time, `103` the third time, and so on.

```
int next_serial(void) {
  static int counter = 100;
  counter++;
  return counter;
}
```

Comments

Comments are text embedded in your code intended for human readers rather than the compiler. C provides single-line comments and block comments.

Where two slashes together (`//`) appear in your C code anywhere outside a character string, the slashes and the remainder of the line on which they appear are a comment.

```
  // This is a comment. The last line of this
  // example has code followed by a comment.
  auqlue = 42;  // The Adams number.
```

Block comments start with the characters `/*` and end with the characters `*/` Everything between is a comment.

```
  /* This is a comment
  which
    continues down
  here */
```

Memory management

sizeof

The `sizeof` operator can be applied to any type or declared variable and will evaluate to the number of bytes required to store an object of that type. `sizeof` operates at *compile time*, so it cannot determine the size of dynamically dimensioned objects. In the following, assume an `int` occupies four bytes.

`sizeof(int)`	evaluates to 4
`int x;`	
`sizeof(x)`	also 4
`int y[5];`	
`sizeof(y)`	evaluates to `20` (five elements by four bytes) since the size of `y` was specified at compile time

malloc

The `malloc` ("memory allocate") function is used to request memory dynamically at runtime. `malloc` takes one parameter, which is the size of memory to request, in bytes. It returns a `void` pointer to the memory allocated, or `NULL` if the allocation failed, e.g, there is no more memory available. The allocated memory is not zeroed or otherwise initialized.

The following example requests memory suitable for the `Client` type from the **typedef definition and use** section.

```
  Client *new_client;
  new_client = malloc(sizeof(Client));
  if (new_client != NULL) {
       success, initialize fields
  } else {
       failure, handle out of memory error
  }
```

calloc

The `calloc` ("contiguous allocate") function is similar to `malloc` but used where space is required for an array. It takes the number of elements and the size of a single element as parameters and returns a `void` pointer. For example, the following requests space for an array of 40 `int` values.

```
int *new_array = calloc(40, sizeof(int));
```

Like `malloc`, `calloc` will return `NULL` on failure. Unlike `malloc`, if `calloc` succeeds it does set all bits in the allocated memory to zero.

free

Memory allocated by `malloc` and `calloc` must be explicitly released using `free` when it is no longer required. Free takes a pointer to previously-allocated memory as a parameter. Failure to `free` unneeded memory is called a *memory leak* and can result in performance problems and program crashes.

For example, to release the memory allocated with `calloc` above, we would call `free(new_array)`

printf and sprintf

The `printf` and `sprintf` functions (found in `stdio.h`) combine a format specification string with (optional) values to produce a result string. `printf` displays its result on the console; `sprintf` puts its result in a character array buffer, which you must provide.

```
  printf(format, v1, v2, …)

  sprintf(buffer, format, v1, v2, …)
```

For example,

```
int n = 2;
double bal = 291.57923;
printf("In %d accounts\nyou have $%0.2f\n", n, bal);
```

will display

```
In 2 accounts
you have $291.58
```

The `format` string can include three kinds of things: literal characters, interpreted character combinations (also known as "escaped characters"), and value placeholders.

literal characters

Literal characters are passed through to the result unchanged. In the example's format string the literal characters are `"In "`, `" accounts"` and `"you have $"`.

escape sequences

See the section on **escape sequence character literals**, which have special meanings in format strings. Here, the only escape sequence is `\n` (used twice) meaning "go to the next line."

value placeholders (conversion specifiers)

Value placeholders (also known as conversion specifiers) specify where the provided values `v1`, `v2`, etc, are to appear in the result string as well as how they should be displayed. Values are inserted into placeholders in left-to-right order and there must be exactly one placeholder for each value.

In the example, the placeholders are `%d` and `%0.2f`, which match the provided values `n` and `bal`. The `%d` means "display as an integer." Similarly, `%f` means "display as a floating point number". The `0.2` in `%0.2f` means "as wide as necessary, with two digits after the decimal."

For a summary of the available C conversion specifiers, see the *C Reference Card* section on Input/Output, particularly Codes for Formatted I/O.

Copyright and use