1. Data Types

Arrays

Arrays are declared using the syntax `type array_name[array_size];`.

When an array of a certain length is declared, the compiler allocates a contiguous block of memory for it. The size of an array cannot be changed after declaration.

In most contexts, the array name decays to a pointer to the first item of an array. As a result, passing an array to a function actually passes a pointer instead of a copy of the array.

In formal parameters, `int arr[]` and `int arr[123]` are equivalent to `int *a`. However, this syntax is only valid in formal parameters and should be avoided.

`x[y]` is defined as `*((x) + (y))`. Since `name` is a pointer, `name[index]` is equivalent to `*(name + index)`. The address of the element at index `i` of the array `array` is `array + (sizeof(array[0]) * i)`.

The size of an array is not stored. The statement `int array[4] = {0, 1, 2, 3};` declares and initialises an array containing 4 values; however, the statement `array[4] = 5;` won't cause an error and will simply access and update the next address, which could be storing another variable or a garbage value. As a result, it is possible to accidentally and unknowingly change the value of a variable, or to access garbage values.

When initialising arrays, if the initialiser list is shorter than the array, then the leftover elements will be initialised to `0`. For example, `int array[4] = {0, 1};` initialises the elements at indices `2` and `3` to `0`. The expressions in the initialiser must be constant, variables cannot be used.

When an array name is assigned to a pointer variable, the variable points to the first element of the array.

Dynamic Allocation of Arrays

When declaring arrays, the size needs to be known at compile time and variables cannot be used. However, dynamic memory allocation of arrays is possible using the function `malloc`. The syntax is as follows`type *array = malloc(length * sizeof(type));`. Note, `sizof(array)` will return the size of the pointer.

It is useful when returning an array from a function. Since local variables are deallocated when the function terminates, but memory allocated using `malloc` isn't, functions can return pointers to memory allocated using `malloc`.

Read more about `malloc` in 3. Dynamic Memory Allocation.

Designator

Designators are a C99 feature. In a designated initialiser, each value is labelled with the member it initialises. Since the values are labelled, they don't need to be in order.

As with regular initialisers, the leftover members will be initialised to `0`. The expressions must also be constant.

If the length of the array is not mentioned, the compiler will use the largest designator to determine it.

Designators increase the readability and make it easier to check for errors.

```
int array_1[5] = { [1] = -3, [3] = 4 };
int array_2[5][5] = { [1][2] = -3, [3][0] = 4 };
int array_3 = {[5][1] = -1};
```

Compound Literal

Another feature introduced in C99, compound literals are used to create unnamed objects with initialised values. They are particularly useful when passing function parameters. Compound literals may contain arbitrary expressions, not just constants.

```
(int [5]) {1, 2, 3, 4, 5};
(int []) {1, 2, 3};
```

Pointers

Pointers are declared using the syntax `type *pointer_name;`

Pointers are variables that store the address of another variable.

The `*` operator dereferences a pointer and the `&` operator returns the address of a variable. The `[]` operator dereferences the address at an offset from the address stored in the pointer.

Logical and Arithmetic Operations

If `pt` is a pointer to a variable of type `type` and `n` is a variable of an integer type, then `pt + n` is equivalent to `pt + sizeof(type) * n` and `pt - n` is equivalent to `pt - sizeof(type) * n`.

Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a sub-object at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space. The constant `0` compares equal to null pointers.

If two pointers are subtracted, both should point to elements of the same array object, or one past the last element of the array object. The result is the difference of the subscripts of the two array elements. The size of the result is implementation-defined, and its type (a signed integer type) is `ptrdiff_t` defined in the `<stddef.h>` header.

The operators `>`, `>=`, `<`, and `<=` convert the pointers to unsigned integers for comparison. Pointer comparison is defined for pointers pointing within the same object in memory.

Null Pointers

A null pointer can be obtained by casting, implicitly or explicitly, the constant `0` to a pointer type. A null pointer is guaranteed to not compare equal to a pointer to any actual value. Hence, it can be used as a signal value. `NULL` can be used instead of `0` for the purpose of clarity. The `NULL` macro is defined in `locale.h`, `stddef.h`, `stdio.h`, `stdlib.h`, `string.h`, and `time.h`.

Note, a null pointer may not necessarily point to the address `0x0`. However, casting the constant `0` of an integer type to a pointer type is guaranteed to result in a null pointer.

All non-null pointers test true. Thus, `if(ptr == NULL)` is equivalent to `if(!ptr)` where `ptr` is a pointer variable.

Pointers to Pointers

Since pointers are variables, there can be pointers to pointers.

Pointers to pointers are commonly used when passing pointers as arguments to functions and are required when the function has to change where the pointer points.

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

void badDoStuff(int *localPointer) {
        localPointer = malloc(sizeof(int));
        *localPointer = 10;
}

void goodDoStuff(int **pointerPointer) {
        *pointerPointer = malloc(sizeof(int));
        **pointerPointer = 10;
}

int main() {
        int *pointer;
        printf("Bad Function - %d\n", (badDoStuff(pointer),
*pointer));
        printf("Good Function - %d\n", (goodDoStuff(&pointer),
*pointer));
        free(pointer);
        return 0;
}
```

Here, the bad function reassigns the local variable `localPointer` to point at the pointer returned by `malloc`, hence not affecting the variable `pointer`. However, since the good function accepts a pointer to the variable `pointer`, it can dereference the local variable `pointerPointer` once and then directly manipulate the variable `pointer`.

Read more about `malloc` in [3. Dynamic Memory Allocation]()

Pointers to Functions

Pointers to functions are often passed as parameters to other functions. They are useful as they allow us to write functions to be as general as possible.

When a function name isn't followed by parentheses, the C compiler produces a pointer to the function instead of generating code for a function call.

We can also declare arrays of function pointers

```c
// function headers
int sum(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
int div(int a, int b);

// Array declaration
int (*func[4]) (int, int) = {sum, sub, mul, div};

// Calling functions at an index
for(int i = 0; i < 4; i++) printf("%d", (*func[i]) (4, 2));
```

Structures

Structures and structure variables are declared using the syntax

```c
/* Variables of a type of struct can be declared in two ways, as
 * shown below. The list of variable names before the semicolon is
 * optional and variables can be declared using the usual syntax
 * Optionally, structs can be initialised at declaration (struct_3)
**/

struct tag {
        type member_1;
        type member_2;
} struct_1, struct_2[10], struct_3 = { value_1, value_2 };

struct tag struct_4;
```

Note, struct tags are not recognised without the keyword `struct`, and hence won't clash with variables or typedefs of the same name. Structs need not have a tag if all the variables are declared along with the struct.

When struct variables are initialised, the values in the initialiser must be in the same order as the members of the struct. The expressions in the initialiser must be constant (variables cannot be used). The initialiser does not need to contain values for all members. Leftover members will be initialised to `0`.

Structs are useful for aggregating data that is not of the same type and when the data has no relation to numeric indices.

The word `struct` is required whenever a variable of a structure type is declared.

Memory is allocated in the order that the members are declared in a struct. Padding might be added to align the data in memory. As a result, the order of declaration must be carefully determined.

The `->` (arrow) operator can be used to access the members of a struct through a pointer to it. The syntax is `structPointer->member`. The arrow operator is defined as `(*structPointer).member`. The parenthesis are necessary since the `.` operator has a higher precedence than the `*` operator, parenthesis need to be used when accessing a member through a pointer to a struct.

The `=` (assignment) operator can be used to copy the values of the members of a struct into another struct. `struct tag struct_5 = struct_3;`

Unlike arrays, a copy of the struct is passed to a function and returned by it. As such, it may introduce a large overhead, particularly with large structs. Passing a pointer to a struct should be considered instead.

Structs can also be self-referential.

```
struct node {
        int root;
        struct node *left;
        struct node *right;
};
```

Designator

```
struct demo {
        int member_1;
        int member_2;
} struct_1 = { .member_1 = 1, .member_2 = 2 };

struct demo struct_2 = { .member_2 = 1, .member_1 = 2 };
```

Compound Literal

`(struct demo) {1, 2};`

Unions

Like structs, unions contain one or more members. However, the amount of memory allocated is equal to the size of the largest member. All the members are stored at the same address. As a result, altering a member alters all members, which could corrupt them. Hence, accessing members other than the one most recently modified results in undefined behaviour.

Unions and union variables can be declared and initialised using the syntax

```
union {
        type_1 variable_1;
        ...
} u_1, u_2 = { value };
```

However, unlike structs, the initialiser can be used to initialise only the first member. The expression in the initialiser must be constant.

The `->` (arrow) and `.` (dot) operators can be used to access the members of a union similarly to structs.

The `=` (assignment) operator copies unions.

Designator

Designated initialisers can be used to specify which member of a union to initialise.

```
union {
        type_1 variable_1;
        type_2 variable_2;
        ...
} u_1 = { .variable_1 = value_1 }, u_2 = { .variable_2 = value_2 };
```

Enumeration

Enumerated type is a type which can have limited set of values that are enumerated by the programmer. Each value must have a name. These names are called enumeration constants.

Unlike the members of a structure or union, however, the names of enumeration constants must be different from other identifiers declared in the enclosing scope.

```
enum BOOL { FALSE = 0, TRUE = 1 } bool_1, bool_2, ...;
enum BOOL bool_3;
```

When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant. The first enumeration constant has the value `0` by default.

Combining Structures, Unions, and Enumerations

```
typedef struct {
        enum { INT, DOUBLE } type;
        union {
                int i;
                double d;
        } data;
} Number;

Number number = { .type = INT, .data.i = 1 };
number.data.d = 2.0;
number.type = DOUBLE;
```

In this way, structures, unions and enumerations can be combined to keep track of which member of a union was last modified.

`typedef`

`typedef` is used for creating aliases of data types.

```
typedef struct tag {
        ...
} tag;

tag struct_1;
struct tag struct_2;
```

Typedefs make the code easier to modify. Since the data types of related variables can be changed by just updating the typedef as opposed to having to carefully change the data type everywhere the variables have been used. This is especially useful since the size of data types varies between computers.

`sizeof`

`sizeof` returns the the size of its operand. `sizeof(char)` is always `1 byte` however the size of other data types is machine dependent. `sizeof` can be used to calculate the length of an array as follows `sizeof(array)/sizeof(dataType);` or `sizeof(array)/sizeof(*array);`. However, this fails if the array variable is cast to a pointer type.

The type of a `sizeof` statement is `size_t`. The type `size_t` is unsigned and its size is platform dependent.

In C89, it is safest to cast it to `unsigned long` when printing, as it is the largest of the unsigned types and hence guaranteed to be large enough.

```
printf("Size of int: %lu\n", (unsigned long) sizeof (int));
```

In C99, `size_t` can be printed directly.

```
printf ("Size of int: %zu\n", sizeof (int));
```

Note, `sizeof` is a compile time operator and hence cannot be used if the size isn't known at compile time, such as using it on an array passed as a parameter to a function.

---

3. Dynamic Memory Allocation

Memory allocation

When memory is dynamically allocated, it is allocated in heap memory. Such memory has to be allocated and deallocated explicitly by the programmer, as opposed to stack memory which is temporary and is automatically deallocated when a function returns.

`malloc`

The prototype for `malloc` is `void * malloc(size_t size);`

`malloc` allocates `size` bytes of memory and returns a `void` pointer to the allocated memory.

In C89, a `void` pointer needs to be explicitly cast to the appropriate type. In C99, a `void` pointer can be cast implicitly.

If `malloc` can't locate a large enough block of memory, it returns a null pointer and sets errno to `ENOMEM`.

Attempting to access memory through a null pointer results in undefined behaviour.

Memory Leak

Not deallocating memory allocated by `malloc` can lead to memory leaks.

```
#include <stdlib.h>

int allocate() {
        int *pt;
        if ((pt = (int *) malloc(sizeof(int))) == NULL) return(-1);
        return(0);
}

int main() {
        return(allocate());
}
```

Here, the pointer `pt` is stored in the stack of `void allocate()` but it points to memory stored in the heap. Since stack memory is deallocated after a function terminated, the pointer `pt` will be deallocated. However, the memory assigned by `malloc` is still allocated but cannot be accessed anymore since the pointer to is lost.

If enough memory is leaked, the program will run out of memory and encounter an out of memory error, or ENOMEM.

`free`

The prototype for `free` is `void free(void *ptr);`

`free` is used for deallocating allocations that were created via the preceding allocation functions. `free` deallocates the memory allocation pointed to by `ptr`. `free` has no return value.

If `ptr` is a NULL pointer, no operation is performed.

The memory leak in the previous example can now be fixed as follows

```
int allocate() {
        int *pt;
        if ((pt = (int *) malloc(sizeof(int))) == NULL) return(-1);
        free(pt);
        return(0);
}

int main() {
        return(allocate());
}
```

`free` does not reset the pointer or value at the address the pointer points to, it simply indicates to the memory management system that the memory can be allocated again. Pointers that point to `free`'d memory are called dangling pointers. Pointers should be set to null after being freed to avoid dereferencing deallocated memory.

Each call to `malloc` needs to be `free`'d individually. In nested data structures, care should be taken to determine the order in which `free` is called so as to not use a dangling pointer.

```
int **pointers = malloc(sizeof(int *) * 2);
// The first element is a pointer to an int
pointers[0] = malloc(sizeof(int));
```

```
// The second element is a pointer to an array of ints
pointers[1] = malloc(sizeof(int) * 3);
free(pointers[0]);
free(pointers[1]);
free(pointers);
```

`calloc`

The prototype for `calloc` is `void * calloc(size_t count, size_t size);`

`calloc` contiguously allocates enough space for count objects that are size bytes of memory each and returns a pointer to the allocated memory.

The allocated memory is filled with bytes of value `0`.

If there is an error, `calloc` returns a null pointer and sets errno to `ENOMEM`.

`realloc`

The prototype for `realloc` is `void * realloc(void *ptr, size_t size);`

`realloc` tries to change the size of the allocation pointed to by `ptr` to `size`, and returns `ptr`.

If there is not enough room to enlarge the memory allocation pointed to by `ptr`, `realloc` creates a new allocation, copies as much of the old data pointed to by `ptr` as will fit to the new allocation, frees the old allocation, and returns a pointer to the allocated memory.

If `ptr` is NULL, `realloc` is identical to a call to `malloc` for `size` bytes.

If `size` is `0` and `ptr` is not NULL, a new, minimum sized object is allocated and the original object is freed.

When extending a region allocated with `calloc`, `realloc` does not guarantee that the additional memory is also `0`-filled.

Passing a pointer to `realloc` that is not NULL and didn't come from a previous call to `malloc`, `calloc`, or `realloc` results in undefined behaviour.

If `realloc` can't locate a large enough block of memory, it returns a NULL pointer and sets errno to `ENOMEM`. The input pointer is still valid if reallocation failed.

---

4. Strings

Strings are `char` arrays with a special null terminator to mark the end of the text. It is `\0`.

String literals are strings that cannot be changed. They are initialised as - `char *text = "hello";`

String variables are initialised as - `char text[] = "hello";` The size of the `char` array when a string is initialised like this is equal to one more than the number of characters. This additional index is for the null terminator. If the size of the string is explicitly stated and the initialisation list is smaller than this stated size, then the remaining indices are initialised to the null terminator. If the initialiser list is the same length as the stated size, then the null terminator isn't added.

When a string literal is used to initialise a char array, the string literal is first stored in the data section of memory, and then copied to the variable to initialise it. The size of a `char` is guaranteed by the standard to be one byte.

`string.h` contains all the C string functions.

Command Line Arguments

Command line arguments are passed to the `main` function as an array of strings. `argv[0]` holds the program's name. `argv[argc]` is guaranteed to equal `NULL`.

Converting Strings to Integers

The library function `strtol` from `stdlib.h` can be used to parse strings as long.

The prototype of `strtol` is `long strtol(const char *restrict str, char **restrict endptr, int base);`

`strtol` converts the value in the string `str` to an integer of the base `base`. If `endptr` is not null, `strtol` stores the address of the first non-digit character in it.

The string may begin with an arbitrary amount of white space followed by a single optional `+` or `-` sign. If base is `0` or `16`, the string may then include a `0x` prefix, and the number will be read in base `16`; otherwise, a `0` base is taken as `10` unless the next character is `0`, in which case it is taken as `8`.

If no conversion could be performed, `0` is returned and the global variable `errno` is set to `EINVAL`. If an overflow or underflow occurs, `errno` is set to `ERANGE` and the function return value is clamped.

Useful Standard Library Functions

`strlen`

The prototype for `strlen` is `size_t strlen(const char *s);`

`strlen` computes the length of the string `s`. It returns the number of characters that precede the null terminator.

`strcpy` and `strncpy`

The prototype for `strcpy` is `char *strcpy(char * dst, const char * src);`

`strcpy` copies the string `src` to `dst` (including the terminating '\0' character.)

The prototype for `strncpy` is `char *strncpy(char * dst, const char * src, size_t len);`

`strncpy` copies at most len characters from `src` into `dst`. If `src` is less than len characters long, the remainder of `dst` is filled with `\0` characters. Otherwise, `dst` is not null terminated.

The source and destination strings should not overlap, as the behaviour is undefined. The input `dst` need not be null terminated as it will be overwritten anyways. For safety, the null terminator should be manually added to the end of `dst`.

`strcpy` and `strncpy` return `dst`.

Safe Usage

```
strncpy(str1, str2, sizeof(str1));
str1[sizeof(str1) - 1] = '\0';
```

`strcat` and `strncat`

The prototype for `strcat` is `char *strcat(char *restrict s1, const char *restrict s2);`

The prototype for `strncat` is `char *strncat(char *restrict s1, const char *restrict s2, size_t n);`

`strcat` and `strncat` append a copy of the null-terminated string `s2` to the end of the null-terminated string `s1`, then add a terminating `\0`. The string `s1` must have sufficient space to hold the result. `strncat` appends not more than n characters from `s2`, and then adds a terminating `\0`.

The source and destination strings should not overlap, as the behaviour is undefined.

`strcat` and `strncat` return the pointer `s1`.

`strdup` and `strndup`

The prototype for `strdup` is `char *strdup(const char *s1);`

`strdup` allocates sufficient memory for a copy of the string `s1`, does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to `free`.

The prototype for `strndup` is `char *strndup(const char *s1, size_t n);`

`strndup` copies at most n characters from the string `s1` and always null terminates the copied string.

If insufficient memory is available, `NULL` is returned and `errno` is set to `ENOMEM`.

`strchr` and `strstr`

The prototype for `strchr` is `char *strchr(const char *s, int c);`

`strchr` locates the first occurrence of `c` (converted to a char) in the string pointed to by `s`. The terminating null terminator is considered to be part of the string; therefore if `c` is `\0`, the functions locate the terminating `\0`. It returns a pointer to the located character, or NULL if the character does not appear in the string.

The prototype for `strstr` is `char *strstr(const char *haystack, const char *needle);`

`strstr` locates the first occurrence of the null-terminated string `needle` in the null-terminated string `haystack`. If `needle` is an empty string, `haystack` is returned; if `needle` occurs nowhere in haystack, `NULL` is returned; otherwise a pointer to the first character of the first occurrence of `needle` is returned.

`strcmp` and `strncmp`

The prototype of `strcmp` is `int strcmp(const char *s1, const char *s2);`

The prototype of `strncmp` is `int strncmp(const char *s1, const char *s2, size_t n);`

The `strcmp` and `strncmp` lexicographically compare the null terminated strings `s1` and `s2`.

`strncmp` compares not more than n characters. Because `strncmp` is designed for comparing strings rather than binary data, characters that appear after a '\0' character are not compared. They return an integer greater than, equal to, or less than `0`, according as the string `s1` is greater than, equal to, or less than the string `s2`. The comparison is done using unsigned characters, so that `\200` is greater than `\0`.

---

5. Compilation (Incomplete)

Makefiles

Makefiles consist of a sequence of rules specifying the target, i.e. the file to be constructed, the dependencies, and the recipe, i.e. the list of commands required to create the target. The recipe is executed if either the target doesn't exist ot one or more of the dependencies are newer than the target.

```
target: dependency_1 dependeny_2...
        recipe
```

The command `make` is used to run the Makefile. If no command line arguments are passed, only the first rule will be evaluated. However, if a command line argument is passed, then it will evaluate the rule for that target. If the rule contains dependencies and the dependencies are also targets in the Makefile, then those rules will be evaluated first.

Makefiles also support name wildcards and variables.

More information can be found here

6. Files and Streams

Streams

An input stream is a source of data that provides input to our program, and an output stream is a destination for data output by the program. The streams standard input, standard output, and standard error are automatically open when a program runs. Streams can also refer to disk files, network connections, and computer devices. Streams can be redirected to change the input or output locations. This is feature of the operating system.

Files

The unix system calls to manage files are `open`, `read`, `write`, `close`. However, these are OS-dependent. For this reason, the `stdio` library is used for file access in C. `FILE` is an abstract data type declared in `stdio.h`. It is portable to non-unix operating systems and supports buffering.

Buffering

It is slow to perform a system call for little pieces of data. The `stdio` library reads a large amount of data even if has not been called for yet, and saves the excess data in a buffer until it is called for later. Similarly, when writing data, it saves a large amount of data. This is so that the system calls need to be called less often.

Library Functions for Streams

`fopen` and `fclose`

The prototype of `fopen` is `FILE *fopen(const char * restrict path, const char * restrict mode);`

`fopen` opens the file whose name is the string pointed to by `path` and associates a stream with it. `mode` points to a string beginning with one of the following letters - `"r"` for reading (fails if file does not exist), `"w"` for writing (truncates a file to zero length if it exists and creates it otherwise), and `"a"` for appending (creates a file if it does not exist; writes always take place at the end of the file regardless of `fseek` calls). An optional `“+”` following any of the aforementioned letters opens the file for both reading and writing. `mode` can also include the letter `“b”` after either the `“+”` or the first letter. Any created files will have mode `S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH`

Upon successful completion `fopen` returns a FILE pointer. Otherwise, `NULL` is returned and `errno` is set to indicate the error.

The prototype of `fclose` is `int fclose(FILE *stream);`

`fclose` dissociates the named stream from its underlying file or set of functions. If the stream was being used for output, any buffered data is written first, using `fflush`.

`fread` and `fwrite`

The prototype of `fread` is `size_t fread(void *restrict ptr, size_t size, size_t nitems, FILE *restrict stream);`

The prototype of `fwrite` is `size_t fwrite(const void *restrict ptr, size_t size, size_t nitems, FILE *restrict stream);`

`fread` reads `nitems` objects, each `size` bytes long, from the stream pointed to by `stream`, storing them at the location given by `ptr`.

`fwrite` writes `nitems` objects, each `size` bytes long, to the stream pointed to by `stream`, obtaining them from the location given by `ptr`.

`fread` and `fwrite` advance the file position indicator for the stream by the number of bytes read or written. They return the number of objects read or written. If an error occurs, or the end-of- file is reached, the return value is a short object count (or zero).

`fread` does not distinguish between end-of-file and error. `fwrite` returns a value less than `nitems` only if a write error has occurred.

`fprintf`, `fscanf`, and `sscanf`

`fgets`

The prototype of `fgets` is `char * fgets(char * restrict str, int size, FILE * restrict stream);`

`fgets` reads at most one less than the number of characters specified by `size` from the given stream `stream` and stores them in the string `str`. Reading stops when a newline character is found, at end-of-file or error. The newline, if any, is retained. If any characters are read and there is no error, a `‘\0’` character is appended to end the string.

Upon successful completion, `fgets` returns a pointer to the string. If end-of-file occurs before any characters are read, it returns `NULL` and the buffer contents remain unchanged. If an error occurs, it return `NULL` and the buffer contents are indeterminate.

```
char str[50]; char *p;
fgets(str, 50, stdin);
if((p = strchr(str, '\n'))) *p = '\0';
```

`fseek`, `ftell`, and `rewind`

The prototype of `fseek` is `int fseek(FILE *stream, long offset, int whence);`

The `fseek` function sets the file position indicator for the stream pointed to by `stream`. The new position, measured in bytes, is obtained by adding offset bytes to the position specified by `whence`. If whence is set to `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively.

`fseek` returns the value `0` if successful; otherwise the value `-1` is returned and the `errno` is set to indicate the error.

The prototype of `ftell` is `long ftell(FILE *stream);`

`ftell` obtains the current value of the file position indicator for the stream pointed to by stream.

Upon successful completion, `ftell` returns the current offset. Otherwise, `-1` is returned and `errno` is set to indicate the error.

The prototype of `rewind` is `void rewind(FILE *stream);`

`rewind` sets the file position indicator for the stream pointed to by `stream` to the beginning of the file. It is equivalent to `fseek(stream, 0L, SEEK_SET);`.

Since `rewind` does not return a value, an application wishing to detect errors should clear `errno`, then call `rewind`, and if `errno` is non-zero, assume an error has occurred.

`fflush`

The prototype of `fflush` is `int fflush(FILE *stream);`

`fflush` synchronises the state of the given stream in light of buffered I/O. For output or update streams it writes all buffered data via the stream's underlying write function. For input streams it seeks to the current file position indicator via the stream's underlying seek function. The open status of the stream is unaffected.

If the stream argument is `NULL`, `fflush` flushes all open streams.

Upon successful completion `0` is returned. Otherwise, `EOF` is returned and the global variable errno is set to indicate the error.

`getchar` and `getc`

`getchar`

- Reads one byte from the standard input.
- Returns an `int` from `0` to the maximum unsigned byte value.
- Returns `-1` to indicate end of file.
- Syntax - `getchar();`

`getc`

- Identical to `getchar` except accepts a pointer to the file to read from.
- Syntax - `getc(FILE*);`

`putchar`

- Outputs one byte to the standard output.

- Returns the byte outputted on success.
- Returns `EOF` on failure
- Syntax - `putchar(byte);`

`putc`

- Identical to `putchar` except accepts a pointer to the file to output to.
- Syntax - `putc(byte, FILE*);`

`fprintf`

- Identical to `printf` except accepts a pointer to the file to output to.
- Returns the number of characters written on success.
- Returns a negative number on failure.
- Syntax - `fprintf(FILE*, "...");`

`fscanf`

- Identical to `scanf` except accepts a pointer to the file to read.
- Returns the number of input items successfully matched and assigned on success.
- Returns `0` of failure.
- Syntax - `fscanf(FILE*, "...");`

`sscanf`

- Identical to `scanf` except reads a string instead of the standard input.
- Syntax - `sscanf(str, "...");`

---

7. Processes

Terminology

- Program - Programs are the executable instructions of a program; either source code or compiled code.
- Process - Processes are the running instance of a program. A process includes the machine code and information about the current state of the process such as current values of variables.

Memory Structure

- Program Code
- Globals - Holds the values of global variables.
- Heap - Holds the values of `malloc`'d variables .
- Stack - Holds the values of current local variables of the function(s) being executed.
- Operating System
    - Process Control Block - A data structure that stores additional information about the state of a process
        - PID - Process ID (a unique identifier of the process)
        - PPID - Parent PID (PID of the parent process)
        - PGID - Process Group ID - Unique ID of the process group
        - SID - Session ID
        - Program Counter - Identifies the next instruction to be executed
        - Stack Pointer - Identifies the top of the stack
        - Open File Table
        - Signal Table - Pointer to code to handle a signal
        - Other things that the OS manages

Process States

The number of CPUs determines how many processes can execute instructions at a time. The scheduler determines which process should be run and when.

- Running State - Processes that are executing instructions simultaneously.
- Ready State - Processes that are ready and would be executing instructions if a CPU were available.
- Blocked State - Processes that are waiting for an event to occur. They are in a blocked or sleeping state.

Creating Processes

Processes can be created using the `fork` system call. This is done by duplicating the process calling `fork`. The operating system duplicates a process by copying its address space, data, and process control block. The child process is almost identical to its parent, differing only in its PID, its Parent PID, and the return value of `fork`.

Upon successful completion, `fork` returns a value of `0` to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of `-1` is returned to the parent process, no child process is created, and the global variable `errno` is set to indicate the error.

The child process has the same value for the program counter as the original process, so when the child process runs, it starts executing just after fork returns. However, when the operating system is finished creating the new process, control can be returned to either the parent *or* the child.

The parent and child processes are now completely separate processes and don't share memory.

Refer to `man fork` for more details.

Parent-Child Relationship and Termination

A parent process can terminate before its child process(es).

The `wait` system call can be used to suspend the execution of the parent process until one of its child processes has terminated. It has one parameter (`int *stat_loc`), a pointer to an `int`. On return from a successful `wait` call, the parameter contains termination information about the process that exited.

The termination information consists of two parts

- The lowest 8 bits indicate whether the child process terminated normally, or whether it terminated because it received a signal. If it terminated due to a signal, the lower 8 bits indicate which signal.
- The next `8` bits contain the exit status.

Macros defined in `sys/wait.h` can be used to extract this information

- `WIFEXITED(status)` - True if the process terminated normally by a call to `exit`.
- `WEXITSTATUS(status)` - If `WIFEXITED(status)` is true, evaluates to the low-order 8 bits of the argument passed to `exit` by the child.
- `WIFSIGNALED(status)` - True if the process terminated due to receipt of a signal.
- `WTERMSIG(status)` - If `WIFSIGNALED(status)` is true, evaluates to the number of the signal that caused the termination of the process.

If `wait` returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

The `wait` system call will fail and return immediately if the calling process has no existing unwaited-for child processes and `errno` will be set to `ECHILD`.

Refer to `man 2 wait` for more details.

Zombie and Orphan Processes

A zombie process is a process that has terminated but still has an entry in the process table. The operating system can't delete the process control block of a terminated process until it knows it is safe to clean it up. Hence, it is only deleted once `wait` is called and its termination status is collected.

An orphan process is a process that is still executing but whose parent process has terminated. The parent PID of an orphan process is `1`, which is the PID of the `init` process. This is because orphan processes are adopted by the `init` process. The `init` process calls `wait` in a loop, hence terminating the processes it adopts.

If the parent process of a child process terminates without calling `wait` and collecting the child's termination status, the child process becomes an orphan. It is then adopted by the `init` process, which calls `wait` and terminates it.

Executing Programs - The `execl` and `execv` family

Explanation

The `execv` and `execl` family of functions replaces the current process image with a new process image. The functions are front-ends for the function `execve`.

The new process is constructed from an ordinary file, whose name is pointed to by `path`, called the new process file. This file is either an executable object file, or a file of data for an interpreter. The system tells whether a file contains a program by looking at its first two bytes or so. If these contain a special value called the magic number, then the system treats the file as a program.

The PID and the parent PID of the process remain the same.

Usage

The `v` stands for vector and `l` for list since these functions accept the arguments as variadic array and an array of strings respectively.

The prototype for `execl` is `int execl(const char *path, const char *arg0, ..., /*, NULL */);`

The prototype for `execv` is `int execv(const char *path, char *const argv[]);`

The initial argument for these functions is the pathname of a file which is to be executed.

The `const char *arg0` and subsequent ellipses in the `execl` can be thought of as `arg0`, `arg1`, ..., `argn`. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments must be terminated by a `NULL` pointer.

The `execv` provides an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the file name associated with the file being executed. The array of pointers must be terminated by a `NULL` pointer.

If any of the `exec` functions returns, an error will have occurred. The return value is `-1`, and the global variable errno will be set to indicate the error.

Commonalities between the caller and the new process

File descriptors open in the calling process image remain open in the new process image, except for those for which the close-on-exec flag is set. Signals set to be ignored in the calling process are set to be ignored in the new process. Signals which are set to be caught in the calling process image are set to default action in the new process image. Blocked signals remain blocked regardless of changes to the signal action. The real user ID, real group ID and other group IDs of the new process image remain the same as the calling process image.

The new process also inherits the following attributes from the calling process - process ID, parent process ID, process group ID, working directory, root directory, control terminal, signal mask

Refer to `man 3 exec` and `man execve` for more details

Terminating Processes

`exit` is used to terminate a process, although a process will also stop when it runs out of program by reaching the end of the main function, or when main executes a return statement.

The single, integer argument to exit is called the process' exit status. It is normally used to indicate the success or failure of the task performed by the process. By convention a process returns zero on normal termination, some non-zero value if something has gone wrong.

A call to `exit` flushes all open output streams closes all open file descriptors. `exit` will also call any programmer-defined exit handling routines and perform what are generally described as clean-up actions. A programmer can also set at least 32 exit handling routines with the `atexit` function. Each of the exit handling functions recorded by `atexit` will be called on exit in the reverse order to which they were set.

`exit` does nothing to prevent bottomless recursion should a function registered using `atexit` itself call `exit`. Such functions must call `_exit` instead.

Files

Every process expects three files to be open, those on file descriptors `0`, `1` and `2`. By default, these correspond to the standard input, standard output, and standard error.

We can redirect where processes take their input from or output to by opening files on file descriptors `0` and `1` respectively.

A successful `open` always gives us the lowest available file descriptor number. If file descriptors `0`, `1`, and `2` are open, and we call `open` successfully, it will give us file descriptor `3`. However, if we close file descriptor `1` before calling `open`, so that the open file descriptors are now only `0` and `2`, then a successful `open` will return `1`, because that's the lowest available file descriptor number. So, the file will be open on file descriptor `1`.

Files kept open across a call to fork remain intimately connected in child and parent. This is because the read-write pointer for each file is shared between the child and the parent. This is possible because the read-write pointer is maintained by the system; it is not embedded explicitly within the process itself. Consequently, when a child process moves forward in a file, the parent process will also find itself at the new position.

Refer to [8. Pipes and Unbuffered IO]

Sample Code

`fork` and `wait`

```
int main() {
        int result;
        int i, j;

        printf("[%d] Original Process\n", getpid());

        for(i = 0; i < 5; i++){
                if((result = fork()) == -1) {  // Fork failed
                        perror("wait");
                        exit(0);  // Terminate parent process
                } else if(result == 0) {  // Child process
                        for(j = 0; j < 5; j++) {
                                printf("[%d] Child %d, %d\n",
getpid(), i, j);
                                usleep(100);  // Suspend thread
execution
                        }
                        exit(0);  // Terminate child process
                }
        }

        for(i = 0; i < 5; i++) {
                pid_t pid;
                int status;
                if( (pid = wait(&status)) == -1 ) {
                        perror("wait");
                } else {
                        printf("Child %d terminated with %d\n", pid,
status);
                }
        }

        printf("[%d] Parent about to terminate\n", getpid());
        return(0);
}
```

`execl`

```
int main()
{
    execl("/bin/cat", "cat", "a wacky file name illustrating that
each of these is an argument without reparsing", (char *)NULL);
    perror("/bin/cat");
    return(1);
}
```

`execv`

```
int main()
{
    char *x[3];

    x[0] = "cat";
    x[1] = "a wacky file name illustrating that each of these is an
argument without parsing";
    x[2] = NULL;
    execve("/bin/cat", x);
```

```
        perror("/bin/cat");
        return(1);
    }
```

`fork`, `exec`, and `wait`

```
int main()
{
    int x = fork();
    if (x == -1) {
        perror("fork");
        return(1);
    } else if (x == 0) {
        /* child */
        execl("/bin/ls", "ls", (char *)NULL);
        perror("/bin/ls");
        return(1);
    } else {
        /* parent */
        int status, pid;
        pid = wait(&status);
        printf("pid %d exit status %d\n", pid, WEXITSTATUS(status));
        return(0);
    }
}
```

Approximating `system`

```
int docommand(char *command) {
        pid_t pid;
        if((pid = fork()) < 0) {
                perror("fork");
                return(-1);
        } else if (pid == 0) {
                execl("/bin/sh", "sh", "-c", command, (char *) 0);
                perror("execl");
                exit(1);
        } else {
                wait((int *) 0);
                return(0);
        }
}

int main() {
        char command[3] = "ls\0";
        return(docommand(command));
}
```

---

8. Pipes and Unbuffered IO

Pipes consist of two file descriptors, a file descriptor to write to the pipe and another to read from it. These two together form an unbuffered communication channel that can be used for inter-process communication.

The `pipe` system call can be used to form a pipe.

When all the write ends of the pipe are closed, `EOF` will be sent to the pipe and calling `read` on it will return `0`.

Data in a pipe can be read only once. If multiple processes call `read` on the read end of the same pipe, then the operating system determines which process reads what data.

A forked process inherits the parent's open file descriptors. When a process terminates, all its open file descriptors are closed.

A `read` call to an empty pipe and a `write` call to a full pipe will block.

A file descriptor is an index into the process' open file descriptor table. This table stores pointers to data structures containing information about open files.

Useful System Calls

`open` and `close`

The prototype of `open` is `int open(const char *path, int oflag, ...);`

The file name specified by `path` is opened for reading and/or writing, as specified by the argument `oflag`; the file descriptor is returned to the calling process. The `oflag` argument may indicate that the file is to be created if it does not exist (by specifying the `O_CREAT` flag). In this case, `open` requires an additional argument `mode_t mode`.

The flags specified for the `oflag` argument must include exactly one o the following file access modes:

- `O_RDONLY` - open for reading only
- `O_WRONLY` - open for writing only
- `O_RDWR` - open for reading and writing

In addition any combination of the following values can be or'ed in `oflag`:

- `O_APPEND` - append on each write
- `O_CREAT` - create file if it does not exist
- `O_TRUNC` - truncate size to `0`

If successful, `open` returns a non-negative integer, termed a file descriptor. It returns `-1` on failure. The file pointer (used to mark the current position within the file) is set to the beginning of the file. When a new file is created, it is given the group of the directory which contains it.

The prototype of `close` is `int close(int fildes);`

`close` deletes a descriptor from the per-process object reference table. If this is the last reference to the underlying object, the object will be deactivated. For example, on the last close of a file the current seek pointer associated with the file is lost; on the last close of a socket associated naming information and queued data are discarded

Upon successful completion, a value of `0` is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

`dup2`

The prototype of `dup2` is `int dup2(int fildes, int fildes2);`

`dup2` duplicates an existing object descriptor and returns its value to the calling process. If both are equal, then just `dup2` returns the second file descriptor. Otherwise, it duplicates the first file descriptor. If the second file descriptor is already in use, it is first deallocated as if a `close` call had been done first. However, it does not close the first file descriptor.

Upon successful completion, the new file descriptor is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

```
// Redirecting a file to standard input
int fd = open("file", O_RDONLY);
dup2(fd, STDIN_FILENO);
close(fd);
```

`read`

The prototype of `read` is `ssize_tread(int fildes, void *buf, size_t nbyte);`

`read` attempts to read `nbyte` bytes of data from the object referenced by the descriptor `fildes` into the buffer pointed to by `buf`.

On objects capable of seeking, the `read` starts at a position given by the pointer associated with `fildes`. Upon return from `read`, the pointer is incremented by the number of bytes actually read. Objects that are not capable of seeking always read from the current position. The value of the pointer associated with such an object is undefined.

The system guarantees to read the number of bytes requested if the descriptor references a normal file that has that many bytes left before the end-of-file, but in no other case.

`read` will fail if the parameter `nbyte` exceeds `INT_MAX`, and it will not attempt a partial read.

Upon successful completion, `read` returns the number of bytes actually read and placed in the buffer. Upon reading end-of-file, `0` is returned. Otherwise, a `-1` is returned and the global variable errno is set to indicate the error. `ssize_t` is signed `size_t`.

`write`

The prototype of `write` is `ssize_t write(int fildes, const void *buf, size_t nbyte);`

`write` attempts to write `nbyte` bytes of data to the object referenced by the descriptor `fildes` from the buffer pointed to by `buf`.

On objects capable of seeking, `write` starts at a position given by the pointer associated with `fildes`. Upon return from `write`, the pointer is incremented by the number of bytes which were written. Objects that are not capable of seeking always write from the current position. The value of the pointer associated with such an object is undefined.

When using non-blocking I/O on objects, such as sockets (refer to 9. Sockets), that are subject to flow control, `write` and may write fewer bytes than requested.

`write` will fail if the parameter `nbyte` exceeds `INT_MAX`, and it will not attempt a partial write.

Upon successful completion the number of bytes which were written is returned. Otherwise, a `-1` is returned and the global variable errno is set to indicate the error.

`pipe`

The prototype of `pipe` is `int pipe(int fildes[2]);`

`pipe` creates a pipe (an object that allows unidirectional data flow) and allocates a pair of file descriptors. Memory for the `fildes` array needs to be allocated before `pipe` is called. The first descriptor connects to the read end of the pipe; the second connects to the write end. Data written to `fildes[1]` appears on (i.e., can be read from) `fildes[0]`.

The pipe persists until all of its associated descriptors are closed. A pipe whose read or write end has been closed is considered widowed. Writing on such a pipe causes the writing process to receive a `SIGPIPE` signal. Widowing a pipe is the only way to deliver end-of-file to a reader: after the reader consumes any buffered data, reading a widowed pipe returns a zero count.

On successful creation of the pipe, `0` is returned. Otherwise, `-1` is returned and `errno` is set to indicate the error.

`select`, `FD_SET`, `FD_ISSET`, `FD_CLR`, `FD_ZERO`

The prototype of `select` is `int select(int nfds, fd_set *restrict readfds, fd_set *restrict writefds, fd_set *restrict errorfds, struct timeval *restrict timeout);`

`select` examines the I/O descriptor sets whose addresses are passed in `readfds`, `writefds`, and `errorfds` to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first `nfds` descriptors are checked in each set; i.e., the descriptors from `0` through `nfds-1` in the descriptor sets are examined.

On return, `select` replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. `select` returns the total number of ready descriptors in all the sets.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets:

- `FD_ZERO(&fdset)` initialises a descriptor set `fdset` to the null set.
- `FD_SET(fd, &fdset)` includes a particular descriptor `fd` in `fdset`.
- `FD_CLR(fd, &fdset)` removes `fd` from `fdset`.
- `FD_ISSET(fd, &fdset)` is non-zero if `fd` is a member of `fdset`, `0` otherwise.

The behaviour of these macros is undefined if a descriptor value is less than zero or greater than or equal to `FD_SETSIZE`, which is normally at least equal to the maximum number of descriptors supported by the system.

If timeout is not a null pointer, it specifies a maximum interval to wait for the selection to complete. If timeout is a null pointer, the select blocks indefinitely. To effect a poll, the timeout argument should be not be a null pointer, but it should point to a zero-valued `timeval` structure.

For example, a timeout of `1` second can be set using

```
struct timeval timeout = { .tv_sec = 1, .tv_usec = 0 };
select(nfds, &read_fds, &write_fds, &errorfds, &timeout);
```

Any of `readfds`, `writefds`, and `errorfds` may be given as null pointers if no descriptors are of interest.

`select` returns the number of ready descriptors that are contained in the descriptor sets, or `-1` if an error occurred. If the time limit expires, `select` returns `0`. If `select` returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified and `errno` will be set to indicate the error.

9. Sockets

Sockets can be used to communicate between processes on different computer. The IP Address of a machine is the address of a machine that can be used to send messages to it over the internet. The port specifies which process should receive that message. A packet contains the address and the data. Port numbers range from `0` to `65535`. However, the ports `0` to `1023` are reserved for well-known services. The ports `1024` to `49151` should be registered for use for public services. Other ports can be freely used. `127.0.0.1` is the local host address which refers to the local machine.

Webpages are typically served at port 80 and secure web pages at port 443.

Stream sockets are built on the TCP Protocol and guarantee that no data will be lost and that the order in which the data is sent will be maintained.

System Calls and Library Functions

`socket`

The prototype of `socket` is `int socket(int domain, int type, int protocol);`

`socket` creates an endpoint for communication and returns a descriptor. `domain` defines the communication domain used for communication. This can be set to either of `PF_INET` or `AF_INET`. `type` defines the socket type. This can be set to `SOCK_STREAM` for stream sockets. `protocol` defines the protocol family used. This can be set to `0` to indicate that the default protocol, which is TCP, should be used.

A stream socket must be in a connected state before any data may be sent or received on it. Once connected, data may be transferred using `read` and `write` system calls. When a session has been completed, `close` may be called.

`-1` is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

`struct sockaddr_in`

```
struct sockaddr_in {
        short sin_family;
        u_short sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];
};
```

`sin_family` should be set to `AF_INET` to b consistent with our `socket` call configurations. `sin_port` should store the port of the process. `htons` needs to be used to make sure the byte order is correct. `sin_addr.in_addr` should be set to `INADDR_ANY` so that the socket can accept connections from any address. The `sin_zero` field pads out the struct so that it has the same size as the `sockaddr` struct. `memset` should be used to set these bytes to `0`.

`bind`

The prototype of `bind` is `int bind(int socket, const struct sockaddr *address, socklen_t address_len);`

`bind` assigns a name to an unnamed socket. When a socket is created with `socket` it exists in a name space (address family) but has no name assigned. `bind` requests that address be assigned to the socket. Binding a name in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed (using `unlink`).

`socket` is the file descriptor of the socket. `address` is the generic struct for all address families. The struct `sockaddr_in`needs to be used for the TCP Protocol. `addres_len` is the length of the address being passed. This should be `sizeof(struct sockaddr_in)`.

`0` is returned on success and `-1` is returned on failure and `errno` is set.

`listen`

The prototype of `listen` is `int listen(int socket, int backlog);`

`listen` specifies the willingness to accept incoming connections and a queue limit for incoming connections. `socket` defines the file descriptor of the socket and `backlog` defines the maximum length for the queue of pending connections. If a connection request arrives with the queue full, the client may receive an error. Alternatively, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

`listen` applies only to sockets of type `SOCK_STREAM`.

`0` is returned on success and `-1` is returned on failure and `errno` is set.

`accept`

The prototype of `accept` is `int accept(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);`

`socket` is a socket that has been created with `socket`, bound to an address with `bind`, and is listening for connections after a `listen`. `accept` extracts the first connection request on the queue of pending connections, creates a new socket with the same properties of socket, and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, `accept` blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, `accept` returns an error. The accepted socket may not be used to accept more connections. The original socket socket, remains open.

`address` is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. Only the `sin_family` parameter needs to be set.

`address_len` is a value-result parameter; it should initially contain the amount of space pointed to by address; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with `SOCK_STREAM`.

It is possible to `select` a socket for the purposes of doing an `accept` by selecting it for read.

`listen` returns `-1` on error and sets `errno`. Otherwise, it returns the file descriptor for the accepted socket.

`connect` and `getaddrinfo`

The prototype of `getaddrinfo` is `int getaddrinfo(const char *hostname, const char *servname, const struct addrinfo *hints, struct addrinfo **res);`

`getaddrinfo` is used to get a list of IP addresses and port numbers for host `hostname` and service `servname`. `hostname` can be set to the string representing the name of the machine on which the code is running. The parameters `servname` and `hints` can be set to `NULL`. `res` is a linked list of structs contains information about the valid addresses. The `sin_addr` field of the `ai_addr` field of `res` can be used to initialise the `sin_addr` field of the `sockaddr_in` struct. `freeaddrinfo` should be called to deallocate the memory used by this linked list.

`0` is retuned on success and a non-zero error code on failure.

The prototype of `connect` is `int connect(int socket, const struct sockaddr *address, socklen_t address_len);`

`socket` is the file descriptor of the socket. `address` is the socket to which `connect` attempts to connect to. `address_len` is the size of the address.

`0` is returned on success and `-1` is returned on failure and `errno` is set.

`htons`, `htonl`, `ntohl`

The prototype for `htons` is `uint16_t htons(uint16_t hostshort);`
The prototype for `htonl` is `uint32_t htonl(uint32_t hostlong);`
The prototype for `ntohl` is `uint32_t ntohl(uint32_t netlong);`

These routines convert 16 bit, 32 bit, and 64 bit quantities between network byte order and host byte order. Network byte order is big endian, or most significant byte first. On machines which have a byte order which is the same as the network order, routines are defined as null macros.

Sample Code

Client

```
int main() {
    // create socket
    int soc = socket(AF_INET, SOCK_STREAM, 0);
    if (soc == -1) {
        perror("client: socket");
        exit(1);
    }

    //initialize server address
    struct sockaddr_in server;
    server.sin_family = AF_INET;
    server.sin_port = htons(54321);
    memset(&server.sin_zero, 0, 8);

    struct addrinfo *ai;
    char * hostname = "teach.cs.toronto.edu";

    /* this call declares memory and populates ailist */
    getaddrinfo(hostname, NULL, NULL, &ai);
    server.sin_addr = ((struct sockaddr_in *) ai->ai_addr)->sin_addr;

    // free the memory that was allocated by getaddrinfo for this
list
    freeaddrinfo(ai);

    int ret = connect(soc, (struct sockaddr *)&server, sizeof(struct
sockaddr_in));
    if (ret == -1) {
        perror("client: connect");
        exit(1);
    }

    printf("Connect returned %d\n", ret);

    char buf[10];
    read(soc, buf, 7);
    buf[7] = '\0';
    printf("I read %s\n", buf);

    write(soc, "0123456789", 10);
    return 0;
}
```

Server

```
int main() {
    // create socket
    int listen_soc = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_soc == -1) {
        perror("server: socket");
        exit(1);
    }

    //initialize server address
    struct sockaddr_in server;
    server.sin_family = AF_INET;
    server.sin_port = htons(54321);
    memset(&server.sin_zero, 0, 8);
    server.sin_addr.s_addr = INADDR_ANY;

    // bind socket to an address
    if (bind(listen_soc, (struct sockaddr *) &server, sizeof(struct
sockaddr_in)) == -1) {
      perror("server: bind");
      close(listen_soc);
      exit(1);
    }

    // Set up a queue in the kernel to hold pending connections.
    if (listen(listen_soc, 5) < 0) {
        // listen failed
        perror("listen");
        exit(1);
    }

    struct sockaddr_in client_addr;
    unsigned int client_len = sizeof(struct sockaddr_in);
    client_addr.sin_family = AF_INET;

    int client_socket = accept(listen_soc, (struct sockaddr
*)&client_addr, &client_len);
    if (client_socket == -1) {
        perror("accept");
        return -1;
    }

    write(client_socket, "hello\r\n", 7);

    char line[10];
    read(client_socket, line, 10);
    /* before we can use line in a printf statement, ensure it is a
string */
    line[9] = '\0';
    printf("I read %s\n", line);
    return 0;
}
```

Miscellaneous

The `,` (Comma) Operator

The `,` (comma) operator is a binary operator. It first evaluates the its left operand, discards its value, and then evaluates its right operand. The value of the right operand is the value of the overall expression.

```
int x = 5;
while(--x, x >= 0) {
        // Do something
}
```

Command-line Arguments

There is an alternate allowable definition of main() in C and C++
`int main(int argc, char **argv)`
`argc` stands for argument count and stores the number of arguments in `argv`. `argv` stands for argument vector and is the array of arguments.
`argv[0]` is the name of the program. Hence, `argc` is one more than the number of command line arguments.
Note, the parameter name `argc` and `argv` are not mandatory, but are standard.

This is equivalent to the `echo` command

```
#include <stdio.h>

int main(int argc, char **argv) {
    for (argc--, argv++; argc > 0; argc--, argv++)
        printf("%s%c", *argv, (argc == 1) ? '\n' : ' ');
    return(0);
}
```

Bitwise Operations

Binary can be represented in C with a preceding `0b`.
Octal can be represented in C with a preceding `0`.

The `&` (bitwise and) operator performs the `&&` operation on each pair of corresponding bits of the operands.
`0b00001111 & 0b00111100` evaluates to `0b00001100`

The `|` (bitwise inclusive or) operator performs the `||` operation on each pair of corresponding bits of the operands.
`0b00001111 | 0b00111100` evaluates to `0b00111111`

The `^` (bitwise exclusive or) operator returns `0` if both operands are either `0` or `1` and returns `1` if exactly one operand is `1`. If the first operand is `1`, it returns the negation of the second operand. If the first operand is `0`, it returns the second operand without negation.
`0b00001111 ^ 0b00111100` evaluates to `0b00110011`

The `~` (bitwise complement) operator negates every bit in the operand.
`~0b00001111` evaluates to `0b11110000`

The `<<` (bitwise shift left) and `>>` (bitwise shift right) operators is a binary operator. The first operand is the value to shift and the second operand is the number of places to shift the value by. The `<<` and `>>` operators work like multiplication and integer division by `2`. The bitwise shift operators have lower precedence than the arithmetic operators.
`0b11001100 << 2` evaluates to `0b00110000`

The statement `(num |= (1 << k));` changes the `k`th bit of `num` to 1.
The statement `(num &= ~(1 << k));` changes the `k`th bit of `num` to 0.
The expression `(num & (1 << k))` checks if the `k`th bit of `num` is `1`.
The expression `(num ^= (1 << k))` toggles the `k`th bit of num.
The expression `(num = (num & ~(1 << k)) | (~num & (1 << k)))` toggles the `k`th bit of num.

Miscellaneous

`extern` can be used when declaring functions and variables in a file other than the one they are defined in.

Functions and variables can be declared multiple times however they can only be defined once in a scope.

The compiler only allocates memory when a variable is defined, not when it is declared.

Declaration `extern int i;`
Declaration, and definition `int i;`
Declaration, definition and initialisation `int i = 0;`

Variables declared in the body of a function belong exclusively to that function; they can't be examined or modified by other functions. In C89, variable declarations must come first, before all statements in the body of a function. In C99, variable declarations and statements can be mixed, as long as each variable is declared prior to the first statement that uses the variable.

In C99, the keyword `static` can be used to specify the minimum expected length of an array argument. This may help the compiler optimise the code better.

```
void function(int array[static 5]) {}
```

The compiler flag `-Werror=vla` can be used to generate an error for using variable length arrays.

Vim mapping `map <F8> :!gcc -Wall % -o %< && ./%< <CR>`

Use the `cdecl` program to understand casts and declarations.