C [exam][2007]

[Based On The Lectures Of Moti Freiman && Ofir Pele]

Collected, extended upon and written by: Aviad Pines && Lev Faerman.

.Contents

C Program Structure

C Files

C program divides into two types of files: the source files (.c) and the header files (.h). The header files contain only the prototypes of functions and structs. The .c files contain the functions and structs implementation. Structs implementation might be inserted into the header file itself, but function implementation **cannot** be written in the header files, since it's a sure way to get a linker error. Exceptions to that are inline functions and class member functions (see C++ sections).

C Compilation Process

The road from source code to an executable is: pre-process, compilation, and linkage:

1. Preprocessor

   Everything that has the # symbol belongs to the preprocessor. The preprocessor's job is to include header files, replace 'define' definitions thorough out the program, and to remove comments.

   The #include command simply tells the preprocessor to copy the entire content of the files preceding the include command to the .c file.

   The #define command does the same thing just inside the .c file itself – it matches the #define command and replace it with the definition entered – either a number, a char, or even a small function (called macro). Since the preprocessor does not validate any arguments or types, you can put everything inside the #define command, but it will probably lead to nasty hard to trace bugs.

2. Compilation

   Takes the code after the preprocessor is done and validates it for proper syntax and type checking. Even after the compilation process, undefined references are allowed – meaning that if we have only prototypes and no definition, the compiler will mark them as undefined references, which will have to result no later then the linkage process.

   To compile only, using the gcc:

   ```
   gcc -Wall -c <files.c>
   ```

   The final product will be .o file (object files)

3. Linkage

   The final stage is to link all the .o files into one executable file. All undefined references from the compilation process have to be defined here, either from user defined functions or from c library functions. Only one file has to contain a main function. If none exist, or more then one exists, a linker error will occur.

Types, Variables && Functions

Primitives

C primitive types: `short, int ,long, float, double, char.`

All the types can be signed (default) or unsigned, which means the type cannot have negative values, but it's positive value range is doubled. Unsigned ints are often used for size definition. C does not have

boolean variables, and goes by a simple rule: 0 means false, and any other number is true, thus the code while(1) will continue infinitely.

Functions

C is a one pass compiler, thus a function knows only the functions above it, unless we use forward declarations (functions prototypes). This also applies (In theory) to variables, that means that all local variables for a function should be allocated before any other code is written for the function.

prototype: `void noFoo();`

full function declaration: `int reallyNotFoo(int bar) {...}`

User Defined Types

enum

Stands for enumaration – a set of named constants.

Declaration:

```
enum notFoo {FOO_BAR, FOO_LOUNGE, PITY_THE_FOO};
```

Usage:

```
enum notFoo enum_var_name;
enum_var_name = FOO_LOUNGE;
```

Each element inside the enum has a value (starting with 0), so it is possible to iterate between the values using increment and decrement. Using the values of the enum is mostly considered bad programming.

struct

We use structs to define types which are not covered by, or are too complex to be conveniently represented by a collection of primitives. This is a primitive version of the Class concept. A struct is allocated as a single continuous memory block.

Declaring a struct:

```
struct <name> {
      <member definitions>
} ; // Do not forget the ;
```

Example:

```
struct Person {
      char *name;
      unsigned int age;
      unsigned int id;
      int bankBalance;
} ;
```

Note, that in order to create a variable of type “Person” we do not write:

```
Person dude;
```

But rather:

```
struct Person dude;
```

To define a pointer to our struct, we would have to write:

```
struct Person * ptr;
```

To allocate our struct on the heap, we would have to write: (see dynamic memory allocation for usage of malloc – page 12)

```
struct Person * dude = (struct Person *)malloc(sizeof(struct Person));
```

Note, that the sizeof command works on structs just as well as it works on primitive types, but the size of the struct may be bigger than the sum of the sizes of its members (since the data might be aligned)

Freeing the pointer would be done in the familiar manner: free(dude);

Accesing Struct Members

This is handled differently if we have to access through a pointer, or struct variable. In the case of a struct variable the access is as follows:

```
struct Person dude;
dude.age = 21;
```

In the case of a pointer to a struct, the same would look like this:

```
struct Person dude;
struct Person *dudeP = &dude;
(*dudeP).age = 21; // option one
dudeP->age = 31; // option two, same result
```

Structs Copying

A struct is forever passed by value, be it to a function, or in an assignment.
This means that a function receiving struct Person as a parameter will actually create a copy of the struct on its stack. This causes an issue with pointers, as in our example, the char *name pointer would simply be copied by its own value, that means that both structs (The original and the copy) will now point to the same place on the heap with their name pointers. Because of that, changing the name of the one of them, will also affect the other. This may not be what we wanted.

The above does not affect arrays. If a struct is passed to a function and it contains an array member, then the array will be copied by value. This is an exception to the pointer rule (Since

arrays are actually const pointers).

A possible solution to copying structs is the creation of a poor man's copy constructor (clone function) that will return a pointer to a true deep copy of the struct, copying all its members by value, including a full copy of pointed memory by a new malloc (Or copy the values into a second struct given as a parameter).
Note, that this solution still means that passing structs by value is not recommended, as the clone function simply will not be called when the struct is passed to the function. A more elegant solution is dealing with structs via pointers. Passing structs by value is acceptable if the struct only contains primitives and arrays – but no pointers.

typedef

Defines a synonym to the specified type declaration:

```
typedef <existing type name> <new type name>;
```

In the previous section, we had to write:

```
struct Person dude;
```

in order to define a variable of type “Person”. By typing:

```
typedef struct Person Person;
```

this could be shortened to:

```
Person dude;
```

In addition, the whole typedef can be rolled into the struct declaration, as follows:

```
typedef struct Person {
      [... members ...]
} Person;
```

Typedef in pointers to functions (See Pointers to functions – page 10)

To define a pointer to a specified function as a different type, the previous syntax is not required, as it would be extremely long. Instead, we write:

```
typedef int (*NotFooFunc)(int x ,int y);
```

and the type NotFooFunc will now refer to a function which receives two ints and returns an int.

bit fields

Bit fields are a special type of struct. They are used in cases where major savings in memory are required, and where using several ints (or chars) as boolean variables would consume too much memory. The definition is rather straightforward:

```
struct MyBitField {
      unsigned int bit0 : 1;
      unsigned int bit1 : 1;
```

```
        unsigned int bit2 : 1;
} ;
```

What we have just defined is a struct of three individual bits, each can be set to be either 0 or 1. We can also set the number of bits to another arbitrary number and receive a relevant number of bits to play with.

Usage:

```
struct MyBitField bits;
bits.bit0 = 1;
bits.bit1 = 0;
bits.bit2 = bits.bit0;
```

Note, even though we defined our bits of type unsigned int, this is really meaningless, since they are not really of that type, so the following line will not compile:
unsigned int err = bits.bit0;

Arrays

General

Unlike Java, arrays are not objects but simply a consecutive block of memory, and the name of the array variable is simply a pointer (discussed in page 9) to the first element in the array (arr[0]).
Declaration of an array:

```
int arr[5] = {1,2,3,4,5}
int arr[] = {1,2,3,4,5}
char[5]; //will contain junk until initialized.
```

Arrays are a constant pointer, which means we cannot move it using pointer arithmetic:

```
a = b;          // illegal
a++;            // illegal
int *c = a+1;   // legal
```

Important:

1. There are no boundary checks what so ever in arrays. Use asserts during debug to make sure you are within boundaries.
2. Arrays cannot be dynamically allocated on the stack by their standard [] syntax, which means that we cannot write arr[i] when i is a variable. The only way to dynamically allocate memory is on the heap using malloc, explained later on. This holds for multi dimensional arrays as well.

Arrays && Functions

Since an array is simply a pointer, passing it as an argument to a function is *exactly* the same as

passing a pointer.

```
void notFoo(int arr[5])
void notFoo(int arr[])
void notFoo(int *arr)
```

All the declarations above are the same.

Important: There is *NO WAY* of knowing the size of the array inside the function, even if you wrote the size (first example) – to know the size you have to send it as a different parameter to the function. Using the sizeof command inside the function will simply return sizeof(void*)

Pointers

General

Pointers are special type of variables which 'point' to other variables - they simply contain the memory addresses of other variables. Each pointer has a type which indicate the variable type which it points to (for type checking and pointer arithmetic). void pointers are a special kind of pointer and will be discussed later on.

Usage:

```
int i, j;
int *x;      // x points to an integer
i = 1;
x = &i       // x points to the address of i
j = *x       // j now has the same value as x
(*x) = 69;   // The value pointed to by x (That is, i) now contains 69.
```

Note: Pointers to pointers are possible (As are pointers to pointers to pointers, and so on). A very common usage is multi dimensional arrays, discussed later on. The syntax is:

```
<type to point to> ** <var_name>
```

and the result is a pointer to a pointer to a given type.

Pointer Arithmetic

Pointer arithmetic is done as regular integer arithmetic, they can have values added to them and subtracted from them (Also by the operators ++ and --). Since each type has a different size, and we know the type the pointer points to, C allows us to add (or subtract) numbers from it, and it will automatically move the pointer to the next required block of memory. This helps us, for example, in arrays. Since we know arrays are actually pointers (discussed later on), we can use the pointer arithmetic to advance through the array – instead of writing arr[i] we can write *(arr + i) and it will mean exactly the same thing, which is to advance the pointer i times the size of the type of arr.

```
int j[5], i;
for(i = 0; i < 5; ++i)
      *(j + i) = 0;
```

The effect of a ++ or -- operator on a pointer is similar. For example:

```
pi++; // Advances the pointer pi by 1 size of int.
```

To sum it all up: The effect of such an action is changing the region in the memory to which the pointer is pointing, but it does not change the value in the memory itself.

NULL Pointer

Unlike Java, where the NULL pointer has a special meaning, in C (And C++) NULL is simply a constant containing the value of zero (And hence the boolean value of false, which can be used in boolean statements). Since the memory address of 0 is not accessible to us, a pointer containing the value of zero is used as a NULL pointer. Note that if you do not include any header of the standard library, NULL will not be recognized and you'll have to define it by yourself (`#define NULL 0`).

Accessing a NULL pointer will compile but will lead to runtime error.

Void*

This pointer, most often seen as part of a generic function or struct, is a pointer to no concrete type. Like any other pointer, it contains a memory address it points to, but carries no type information. This makes the (void *) a dangerous type to use.

The void pointer cannot be dereferenced, an attempt to do so will result in an error. It must be explicitly cast to a concrete type before that (Which is where most bugs may result). Pointer arithmetic on void pointers works in multiples of a single byte of memory.

Pointers To Functions

Since functions are stored in the memory, we can define a pointer which will point into that section of the memory in which they are located, allowing us to write much more flexible code.

syntax: `<return type> (*<func_ptr_name>)(<parameters>)`

example: `int (*func)(int a, char* b)`

To use a pointer to a function, simply assign a function to it:

```
func  = &notFoo;
func = notFoo;
```

will declare that the pointer to a function named func is now pointing to the function notFoo, and to use the function through the pointer just write:

```
func(<parameters>)
```

Functions which receive pointers to other functions:

To write a function which receives a pointer to a function, simply write the name of the function it receives in the parameters list:

```
int rec_func(int he, int cmp(char a, long b));
```

```
int rec_func(int he, int (*cmp)(char a, long b));
```

both statements above declare the same thing: a function rec_func, which returns an int and receives an integer and a pointer to a function called cmp, which returns an int and receives a char and a long.

Multi Dimensional Arrays

General

Multi dimensional arrays are actually arrays in which each cell is an array of its own. As we've seen that arrays are actually pointers, multi dimensional arrays are actually pointers to pointers, that is int arr[5][4] is actually of type int**.

Ways to write:

```
int arr[5][4]     // a 2d array with 5 rows and 4 columns
int **arr         // can be allocated dynamically using malloc
int arr[][5]      // declare a pointer to an array of size 5
int (*a)[5]       // same.
```

MDA && Functions

Since sending an array to a function is exactly like sending a pointer, sending a multi-dimensional array to a function is actually sending a pointer to an array, meaning that only the first dimension (the first [ ]) can be left unknown. All other dimensions have to be known and will be known even inside the function using sizeof. If we are using more than two dimensions, all the dimensions aside from the first one have to be known.

Representations

Using a matrix form:

The standard and more intuitive way, simply hold an array in which each cell is an array of its own. Define it by `arr[rowNum][colNum]`. Creates rowNum memory blocks, each of size colNum. Can be accessed using `arr[i][j]`.

Row major ordering:

Define the matrix as a one dimensional array and arrange it into columns and rows using simple calculations. In this way all the memory is in one chunk, which makes iterator usage extremely easy, but the rest of the code is less coherent. The access to the cell in this way is much faster.

Memory Management

General

There are three places which C can allocate memory to: The stack, the static heap and the dynamic heap, each with a different use by the program, but with similar access for us.

Stack

The stack is a place in the memory which all local variables are stored during their lifetime. All functions (including main) allocate their locally declared variables and parameters which are given by value on the stack. Each function has its own region on the stack, and when the function ends, its stack is

automatically freed and all variables that were declared on it are destroyed.

Static Heap

The static heap is the place for global variables declared through out the program (in the global scope). Those variables exist through out the entire program and are destroyed when the program ends. The word 'static' by itself has other meanings which are discussed later on.

Dynamic Heap

When we need memory to be allocated dynamically during the program runtime we have to define it on a place called the dynamic heap. All memory on the dynamic heap is only allocated by the programmer, and has to be freed by the programmer as well. If the memory is allocated but not freed, it will cause memory leaks. To allocate memory on the dynamic heap we need to use the dreaded malloc command.

Dynamic Memory Allocation

There are few commands that allocate memory on the dynamic heap. The one we use the most is malloc, but there are a few more:

`calloc` `void *calloc(size_t nmemb, size_t size);`

Calloc() is used to allocate arrays of elements. It allocates memory for 'nmemb' elements, each of size 'size'. A void pointer is returned and again, must be explicitly cast before it can be used. Each element is set to zero.

`malloc` `void *malloc(size_t size);`

The malloc() function asks the operating system to allocate 'size' bytes on the dynamic heap and then returns a pointer to said memory. The pointer must be explicitly cast to a concrete type in order to be used. From that point onward, it may be accessed as any other pointer (But it must be freed before the program ends).

`realloc` `void *realloc(void *ptr, size_t size);`

Function changes the size of the memory object pointed to by *ptr* to the size specified by *size*. The contents of the object will remain unchanged up to the lesser of the new and old sizes. If the new size of the memory object would require movement of the object, the space for the previous instantiation of the object is freed. If the new size is larger, the contents of the newly allocated portion of the object are unspecified. If *size* is 0 and *ptr* is not a null pointer, the object pointed to is freed. If the space cannot be allocated, the object remains unchanged.

`free` **`void free(void *ptr);`**

Frees the allocated memory pointed by ptr from the heap.

The return value for malloc, calloc and realloc is void* to the newly allocated memory.
IMPORTANT: Using free on an unallocated memory or memory that was freed before will cause unknown problems (probably seg. fault or glibc)

Example: Dynamically allocating a two dimensional array using malloc:

```
// This code dynamically allocates memory for a square matrix of size
// arrSize * arrSize.

int **arr; // will point to the matrix
int i, arrSize = 10;
arr = (int**)malloc(sizeof(int*) * arrSize);
for(i = 0; i < arrSize; ++i)
      arr[i] = (int*)malloc(sizeof(int) * arrSize);

// now to free the array:

for(i = 0; i < arrSize; ++i)
      free(arr[i]);
free(arr);
```

Inter Module Scope Rules && Variable Attributes

Extern Variables && Functions

The extern keyword tells the compiler that a variable is declared in another source module. The linker then finds this actual declaration and sets up the extern variable to point to the correct location. If a variable is declared extern, and the linker finds no actual declaration of it, it will throw an "Unresolved external symbol" error. In the case of functions, the extern word is redundant since by importing the relevant header file, we already cause the linker to search for a suitable function.

Example:

```
extern int i;
```

This tells the linker to go and find a global variable of type int by the name of i in some other module you are linked with.

Another example:

#file.c

```
int num = 900;
```

#file2.c

```
extern int num;
void printNum(int n) {
      printf("%d", n);
```

```
}

int main() {
     printf(num);
}
```

the result will be: 900

Static Variables && Functions

A static variable is only accessible in its own source module (its own .c file), the same goes for a static function. If a second file 'externs' a static variable, and a fitting variable does exist in another module, this will result in a linker error.

Static Variables In A Function

A static variable in a function is allocated on the static heap and is initialized in the function only by the first call to the function. It must be initialized when it is declared, and will exist for the running time of the program. It can be changed by the function following its declaration and the value will be saved until the next time the function is called. This can be useful to implement counters (Such as malloc counters).

Example:

```
void fooCount() {
      static int num = 0;
      printf("This func was called to: %d times.\n", ++num);
}
```

The counter will increase every time the function is called.

Const Variables

Primitives

Any primitive type that is declared const cannot be changed through out the program. A const variable has to be initialized upon its declaration (unlike Java).

Pointers

Pointers has few const declaration: The golden rule is that a const protects the name to its left, unless there is nothing on the left, then it protects its right side.

```
const int* ptr              A pointer to a constant integer
                            *ptr = 1 //illegal
                            p++      //legal
                            p = NULL //legal

int const *ptr              Same

int* const ptr              A constant pointer to an integer
                            *ptr = 1 //legal
                            p++      //illegal
                            p = NULL //illegal


const int const *ptr        A constant pointer to a constant integer
                            *ptr = 1 //illegal
```

```
p++       //illegal
p = NULL //illegal
```

(As we mentioned before, an array is actually a const pointer to a type)

Structs

Defining a struct as const means that all of its fields are immutable. That is as implying a “const” qualifier to every one of its members. It's getting tricky though, when we discuss pointers and nested structs inside a const struct, for example:

```
struct Person {
      int age;
};

struct Student {
      struct Person p;
      char *name;
      int gradeAvg;
};

int main() {
      struct Person p1;
      p1.age = 24;

      struct Student stud;
      stud.p = p1;
      stud.name = (char*)malloc(sizeof(char) * 5);
      *stud.name = “Dude”;
      stud.gradeAvg = 65;
      const struct Student cstud = stud; // Initialized by copying each
                                         // value.

// The various commands we can try:
      cstud.p.age = 23;     // Illegal – The inner struct is also const.
      cstud.gradeAvg = 67   // Illegal – This is a const

      *cstud.name = “d00d”; // Legal – The pointer is const, the value
                            // inside is not. A pointer breaks the
                            // const chain. Note that this also changed
                            // the struct stud, since the two pointers
                            // point to the same place (copied by
```

```
                                // value).
        cstud.name++;           // Illegal, the pointer is const.
        return 0;
}
```

In essence, the entirety of the struct, and its members (and if the members are structs, then their members) behave as if they are declared as const. However, while a member pointer is itself a const, the value it points to is not. It will forever point to the same place in memory, but the value it points to may be changed freely (just like defining <type> const *<var_name>).

Sending Const Parameters To A Function

The const qualifier should be added onto any value which we do not wish to accidentally change. For example, the function:

```
void printInt(void *intSuspect);
```

Which receives a void pointer, and casts it to an int to print it, should in fact be defined as:

```
void printInt(const void const *intSuspect);
```

As we only wish to print the value, and not change the pointer, nor the value itself. This is crucial for proper coding.

C Strings

General

Unlike Java, where Strings are objects, in C strings are arrays of characters that must have '\0' as their last element (The char '\0' has the ASCII value of zero, identical to the NULL constant we have met earlier), usually identified as: char *str. Since they are arrays they accept the same [] operators that arrays work with. That is, the expression str[5] returns a single char. The char '\0' tells the different string manipulators where to end their work on the string. A string lacking the '\0' in its end will more often than not lead to illegal memory access and unspecified behavior (Due to an array index out of bounds situation).

Ways to define strings:

```
char str[7] = “Hello!”;
char str[] = “Hello!”;
char str[] = {'H','e','l','l','o'};
char *codeSegmentString = “Hello !!”;
```

The examples above are not identical. For example, the first two are the same (Notice that we included

the '\0' in the size, but we did not type it ourselves), but the third is not a string, as it is not terminated by a '\0'. However, if we did write the '\0' as an additional element, it would be a string. The last one isn't what it seems. The pointer codeSegmentString points to the section of the memory holding the program's code which is protected. If we attempt to change it, e.g:

```
codeSegmentString[3] = 'a';
```

we would get a segmentation fault.

Strings Manipulation Functions

strcmp `int strcmp(const char *s1, const char *s2);`

The function compares the two strings s1 and s2. It returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2. The compare is done lexicographically.

strdup `char *strdup(const char *s);`

The strdup() function returns a pointer to a new string which is a duplicate of the string s. Memory for the new string is obtained with malloc, and can be freed with free.

strlen `size_t strlen(const char *s);`

Function calculates the length of the string s, **not including the terminating `\0' character.**

strcpy `char *strcpy(char *dest, const char *src);`

Function copies the string pointed to by src (**including the terminating `\0' character**) to the array pointed to by dest. The strings may not overlap, and the destination string dest must be large enough to receive the copy. No boundary check is applied.

strcat `char *strcat(char *dest, const char *src);`

Function appends the src string to the dest string overwriting the `\0' character at the end of dest, and then adds a terminating `\0' character. The strings may not overlap, and the dest string must have enough space for the result. No boundary check is applied.

sscanf `int sscanf(const char *str, const char *format, ...);`

sscanf is identical to the scanf function, only it reads its input from a c-string pointed to by str.

atoi `int atoi( const char *str );`

The atoi() function converts *str* into an integer, and returns that integer. *str* should start with whitespace or some sort of number, and atoi() will stop reading from *str* as soon as a non-numerical character has been read.

atof && atol Identical to atoi, but converts to double && long respectively.

I / O

The common I/O functions in C are:

printf `printf(format string, arguments)`

Prints to the standard output stream the string with formatted arguments. Types of formatting that can be put in the string: %d – integer, %f – float, %lf – double, %l – long, %p – pointer (print address), %c – single character, %s – string (char*). To change floating point print accuracy: `printf("%.4lf\n", j);` will print a double variable j with 4 digits after the decimal point.

scanf `int scanf(format string, arguments)`

Reads input from the standard input stream into the arguments according to the format string. White space inside the format string can match any amount of white space in the input (including none) while any other character will be matched only to itself (and will not be stored inside any var). Scanning stops when a character does not match the format string or if a conversion cannot be made between the input and the arguments assigned. The return value of the function is the number of variables that were actually assigned.

putchar `int putchar(int c)`

Writes a single character to the standard output stream. Return value is the character that was written cast into an int, or EOF if error occurred.

getchar `int getchar()`

Returns a single character from the standard input stream cast into an int, or EOF at end of file or error.

gets `char *gets(char *s)`

Reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with '\0'. No check if we exceeded the size of the char array s.

File I/O

General

Just as we have different functions for general I/O, we also have functions for file I/O. These are also

located in stdio.h. First, we need to understand how C treats files: It describes file streams (And streams in general, as we will see shortly) as pointers to files with the alias: 'FILE *'. In addition, the standard output stream, standard input stream and the standard error stream (stdin, stdout, stderr) are also of the 'FILE *' type, and these functions work on them also, but since they are used so often, C has special functions made just for them as we've seen in the I/O section.

The functions for file I/O are generally identical in behavior to the general I/O functions, however, they receive an additional parameter of type 'FILE *' to indicate to which stream we wish to send data to (or receive from). The name of the function is usually the same as the standard I/O function with a 'f' prefix (fprintf, fscanf, etc').

Sometimes, especially when redirected input/output/errors are required, these functions will also be used with the stderr, stdout and stdin streams, allowing us to write flexible code.

Opening && Closing Files

To close and open files, we use the fopen and fclose functions. The fopen function receives a c-string which indicates the path of the file we want to open, and returns a FILE* variable which points to the specified file, or NULL if there was a problem, and errno global var is set with the proper error number. The fclose function receives a FILE* variable, and attempts to close it. Upon success, it returns 0, otherwise it returns EOF and errno is set to the proper error number.

Example of file I/O (taken from ex2):

```
int processAutomaton(Automaton *a, char* dataFilePath) {
	FILE* data;
	data = fopen(dataFilePath, "r");
	if(data == NULL) {
		printf("Problems opening file, aborting\n");
		exit(1);
	}
	parseFile(data, a); // parse the file into the automaton
	fclose(data);
	return testInput(a, 0, 0);
}
```

Libraries

General

Libraries are a collection of functions, user defined types, redefinition of existing types (Such as size_t,

which is actually an unsigned long, or an unsigned int) and/or global variables - linked in some form or concept to one another. They exist to provide generic useful functions and types for other programmers to use. For example: The GNU C standard libraries, the Math library or graphics libraries. A single library may be composed of many different object files.

Libraries are divided into two types: static libraries and shared (dynamic) libraries.

Types of Libraries

Static Libraries

These libraries are linked with your code during the link process. Their standard suffixes are: .a (unix), .lib (windows). The relevant code is copied into your executable as it is created. The executable may then run without any dependency on the static library.

Shared (Dynamic) Libraries

These libraries contain code which is not directly copied into your executable. This code is linked with the executable on-demand, during run-time, making your executable dependent on the presence of the library files on the machine. Their standard suffixes are: .so (unix) and .dll (windows).

Using && Creating Static Libraries

Syntax: `gcc object1.o object2.o -ldata -o prog`

This will compile object1.o and object2.o into the executable prog and link with the static library libdata.a. The compiler knows to add the 'lib' prefix and the '.a' suffix when it searches for the library with the relevant name. Using the library later would be done by including the relevant header into your code.

Another important compilation flag is the -I (capital i, not lower-case L) flag. This flag allows us to specify the location where our libraries are indeed located, if they do not sit in the same location where out code is. The previous example would look like this, if my library was located at /usr/libs/:

```
gcc -c object1.o object2.o -I /usr/libs/ -o prog
```

The relevant flag during linkage is -L.

Creating a Static Library

Creating a static library is done by compressing the files with the 'ar' command (It is similar to tar), by using the following syntax:

```
ar rcu <name_of_library> <names of object files>
```

After which we run the following command:

```
ranlib <name_of_library>
```

The ranlib command creates the library and the symbol table for the library, which is a sort of an index telling the linker what the library contains. A library can exist without a symbol tablle (you can use the result of the 'ar' command directly) but this will result in much longer link times, as the linker will have to search all the files in the library for the relevant code (libraries may well be thousands of files large).

Note: Dynamic library creation and usage was not discussed in class.

Linking Process

The link process differs between objects and libraries:

Objects

The whole object (the code) is linked to the executable, even functions which are not used !

Two different function implementations (for the same prototype) will cause a link error.

Libraries

Only code not found in the object files compiled from our .c files is linked.

In the case of two implementations for the same function prototype, the local (our) version is used.

Library Type Comparison

Static Library Pros:

Independent of the presence of library files. Will run even with them absent.

Less linking overhead during run-time demand (It is also easier to evaluate program efficiency and write faster code).

Shared Library Pros:

Smaller executable files.

No need to re-compile executables when libraries are changed (If the basic interfaces remain intact).

The same executable can run with different libraries.

Loading libraries dynamically, as they are demanded by the program, is possible.

Make

General

Make is not part of C syntax but is a different unix program by its own, which purpose is to update other programs in a smart way, thus updating only the parts of the program that were changed since last compilation.

Running Make

When pressing 'make' in the shell, it automatically looks for a file named 'makefile' or 'Makefile'. If we want to name our makefile with another name we need to use the syntax:

```
make -f <new_makefile_name>
```

Writing A Makefile

Explicit Rules

Syntax:

```
target : prerequisites
<tab>   command
<tab>   . . .
```

It is important that the <tab> will actually be a tab indent and not spaces indent or the make will not recognize it as a command.

target: the label that will be called upon to execute the command it holds.

prerequisites: The files or other labels this label depends upon.

command: can be any shell command available, such as gcc, clear, rm, etc'

The way make works:

Given a target (by the 'make' command or by other targets), finds the line in the make file containing the label of that target, and check if the prerequisites are up to date (this check is done recursively). If one or more of the prerequisites is newer than the target, run the command(s) in the body.

For example:

```
program: notFoo.o

notFoo.o: notFoo.c notFoo.h
      gcc -Wall -c notFoo.c
```

now, when the command 'make program", the program label will go to the label notFoo.o, since it is a prerequisite. If either notFoo.c or notFoo.h had been updated, it will run the compilation command again.

Implicit Rules

Explicit rules tells us exactly what files are we looking for, but what if we want to create a rule for all the .c files? In case like this we can use implicit rules, such as the following:

```
.c.o:
      gcc -Wall -c $< -o $@
```

Tells us to create .o file from .c files. Notice that there are some weird symbols there, these are few of makefile's automatic variables:

`$@` : File for which the match was made.

`$<` : The matched prerequisite (matched dependency)

`$*` : The match without the suffix

`$^` : The entire prerequisite list, including their full paths.

So if we have several files: test.o and test.c, test1.o and test1.c and test2.o along with test2.c, the result of this command will be: Find all .o files, make them depend on the .c files, check the dates of the .c files, if any of them are newer, then compile them into .o files and leave those which have not been modified alone.

The implicit rules are made with respect to the explicit rules, for example, if we have the above line, and also a specific rule for test.o, then the specific explicit rule will be used for test.o, but the implicit rule will be used for the remaining .o files. Implicit rules will only apply to files which have explicit rules for them.

Variable Definitions

We can define variables in makefiles, in order to make our lives easier. For example, if we wish the makefile to be able to run across several platforms, then we would like to create a variable which would hold the name of our compiler, so that we would not have to modify the entirety of the file if we moved to a different compiler:

```
COMPILER = gcc
```

And now if we wish to write a compile command then we will write:

```
<tab> $(COMPILER) <additional arguments and file names>
```

Comments

Comments in makefiles are denoted with # (Just as in C/C++).

Wild Cards

Wildcards are used to cover many files with a single command (prerequisite or target), for example, the wildcard * denotes any string:

```
clean:
<tab> rm *.o -r
```

The command 'make clean' will immediately run the rm command (as it has no prerequisites) and remove all files ending with the extension .o

The command 'make clean' in this case:

```
clean:
<tab> rm proj?.o -r
```

Will remove all files starting with 'proj' followed by any character and the extension .o. In contrast to the * wildcard, ? only represents a single character, even though any character will match it.

Automatic Dependencies

Thankfully, gcc (and g++) can resolve dependencies automatically, using the following syntax:

```
gcc -MM *.c >> <makefile_name>
```

This will read the dependencies of all .c files and enter them automatically into the specified makefile.

Special Targets

The target 'all' is the target automatically searched for if no specific target follows the make command, that is: 'make' equals 'make all'. The target 'clean' is commonly used to clean up the project directory, usually by removing all .o files.

Every target which is not specifically a file name (e.g clean and all) are written under a special qualifier called '.PHONY' which indicates to the make program, this is not really a file, but a label describing an action.

Debugging && Common Bugs

Pointer Arithmetic

Careless pointer arithmetic can easily cause Segmentation Faults (which are attempts by a program to access a memory region to which it was not allowed access by the operating system). Common causes of Segmentation Faults are buffer overruns (Writing too much into a string) and array index out of bounds situations. In addition, care should be taken to not lose allocated memory locations, as this can lead to memory leaks (See Memory Management).

malloc( ) && free( )

The greatest source of memory leaks are those specific two commands (And their similar counterparts, see dynamic memory allocation). Aside of the usual reminder to use valgrind to make sure no memory leaks are present, there are several common mistakes we should most likely avoid:

Returning A Pointer To A Local Variable:

The following code will result in unspecified behavior, as the temp variable is allocated on the stack rather than on the heap and the memory is freed when the function myStructCtor() terminates:

```
struct MyStruct {
	int _myNum;
}

struct MyStruct * myStructCtor() {
	struct MyStruct temp;
	temp._myNum = 0;
	return &temp;
}
int main() {
	struct MyStruct *ptr = myStructCtor();
	ptr->_myNum = 10;
	return 0;
}
```

Freeing Memory In The Wrong Order:

In the following example, we have fixed the previous error, and used malloc properly. We also decided to write a C-Constructor and a C-Destructor for our struct. Our mistake was in the Destructor:

```
typedef struct MyStruct {
	int _myStackNum;
	int *_myHeapNum;
} MyStruct ;

MyStruct * myStructCtor() {
	MyStruct *temp = (MyStruct *)malloc(sizeof(MyStruct));
	temp->_myStackNum = 0;
	temp->_myHeapNum = (int *)malloc(sizeof(int));
	*temp->_myHeapNum = 0;
}

void freeMyStruct(MyStruct *deleted) {
	free(deleted); // MEMORY LEAK !!
}
```

Our mistake is simple: Our constructor allocated more memory on the heap (To be used by the pointer _myHeapNum), but when we freed the memory, we freed only the struct itself, and not the memory allocated additionally for its use. Here is the fixed version of freeMyStruct:

```
void freeMyStruct(MyStruct *deleted) {
	free(deleted->_myHeapNum);
	free(deleted);
}
```

Using Uninitialized Pointers:

If we create a bunch of pointers (Say an array), then we should initialize all of them to be NULL. This is for a simple reason. Calling free() on a NULL pointer is legal and performs no operation. However, calling free() on a pointer to memory which was not allocated to us will lead to unspecified behavior (And often the dreaded “glibc detected” crash). This is likely to save us plenty of headache later on.

Asserts

Assert is a great tool for debugging our own code. However, it both increases running time (Additional function calls) and it is not elegant. Do use assert often in your code (Array bounds, expected input to a

function, and so forth), but do not forget to define: #define NDEBUG above the #include <assert.h> line when you compile your code for distribution. Also, assert is not to be used to check user input. It prints no informative message and aborts the program in an “ugly” way.

#ifndef, #ifdef, #else, #endif

These tags can be used to define sections of code which will only run if specific preprocessor constants are defined (or not defined) in the sourcecode. This can be highly useful to create code which has a lot of debugging information (printf(), redirected output, etc.), but is lightweight when compiled for distribution. In addition, the #ifndef tag should be used to envelop the entirety of every .h file, so that if a file includes a given header A.h and also includes the header B.h which by itself includes A.h the compiler will not run into multiple definition issues.

Example:

```
#ifdef NDEBUG
    #define printDEBUG(format, args ...)  ;   // notice the ;
#else
    #define printDEBUG(format, args ...)  printf(format, args)
#endif
```

Notice that if NDEBUG is defined, every printDEBUG will be replaced with `';'` , which will have no effect at all. Otherwise, it will be the same as printf.

Misc. Functions

```
void *memcpy(void *dest, const void
*src, size_t n);
```

The memcpy() function copies n bytes from memory area src to memory area dest. The memory areas may not overlap. Use memmove() if the memory areas do overlap (Not studied in class).

```
void qsort( void *base,
            size_t nmemb,
            size_t size,
            int(*cmp)(const void *,
                      const void *) );
```

The qsort() function sorts an array with nmemb elements of size size.

The base argument points to the start of the array. The contents of the array are sorted in ascending order according to a comparison function pointed to by compar, which is called with two arguments that point to the objects

being compared. The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined.

The qsort() function returns no value.