(The Very) Basics of C Programming
(With some C++)

Nuno Miguel Cardanha Paulino (nmcp@fe.up.pt)
Professor Auxiliar Convidado, DEI, FEUP

February 2020

This document is a condensed overview on basics of C programming, and on some concepts about compilation in general. The point of this document is to introduce you to some concepts that might not be familiar if your prior programming experience was not C/C++ or a compiled language.

This document is a work in progress! So check back for future versions.

Contents

1 C Compilation Flow

Unlike interpreted languages (e.g., Python and MATLAB) C/C++ programs are compiled into executable files. Compilation is the process of transforming one or more source code files into a single output file which can be executed on a specific target environment (e.g. Windows, Ubuntu, etc). Depending on the environment, compilation details vary (e.g. different compilers and libraries), but the general process remains the same. The following figure illustrates this:

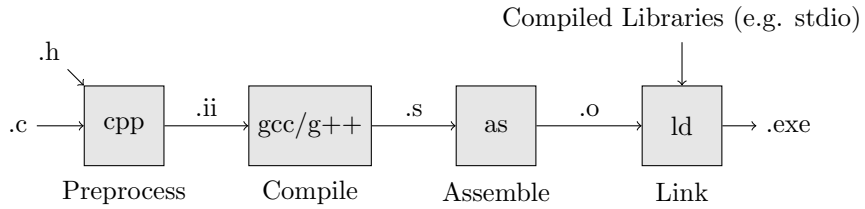

Figure 1: C/C++ Compilation Flow

In C there are two types of source files: *.c* files, which are most of the source code, and *.h* files, which are called headers. Source files and header files are joined together by the preprocessor, *compiled* into several *.s* files, which are then *assembled* into *.o* (object) files, and then *linked* into the final executable.

The *.s* files are are assembly files, which contain assembly instructions. The assembly code is specific to the processor (or processor family) you are coding for, so at this stage, the code is no longer independent of processor details, i.e., it is not high-level source code. The assembler transforms these instructions into binary code, which is what the processor reads from memory when executing. The final stage is to *link* all the object files together. Linking is the process of combining all the information, such as variable and function names, and their location in memory. A call to *gcc/g++*, from the command line or from an IDE like Visual Studio, looks similar to this:

```
gcc main.c file1.c file2.c -o myexecutable
```

This single call performs **all** of the steps shown in Figure 1. The compiler receives a list of source files, and the *-o* option which specifies the name of the output executable. Other options, called *flags*, can be passed to the compiler. See Section 16.

2 About Headers

The headers are included by the source files with the directive:

```
#include<headerfile.h>
```

Where *headerfile* is the name of the file. Directives which start with the cardinal symbol (#) are processed by the C Pre-processor (*cpp*), as shown in Figure 1. Another example of this are *macros*, explained in Section 3.

An included file is basically copied, by *cpp*, into the file that is including it. This means that you can (and must) define things such as constants, macros, and function prototypes in a **single** file, and include that information into another, or multiple, *.c* files. But why must a source file necessarily include header files? So that *symbols* in other *compilation units* are made visible to that file.

A *symbol* is any variable name, or function name. A *compilation unit* is any file that the compilation flow transforms from a *.c* into a *.o* file. But why are contents of files not "visible" to each other? Don't you pass all the files to the compiler? Yes, but files are compiled one at a time, so each file needs to know the name and type of other symbols, in other files, that it needs to use.

2.1 Basic Header Usage

Header files are (very) typically included at the top of source files. Although its possible to include a *.c* file, that's not good practice (remember, the "only" thing *include* does is copy-paste). If you want functions defined in *file1.c* to be used in *file2.c*, then you should create a *file1.h*, and include that file in *file2.c*. Specifically, if you want a function in *file1.c* to be visible to others, then *file1.h* should contain only its **prototype**. See Section 12 for more.

2.2 Header Guards

Header guards are a technique that make use of the preprocessor to prevent an include file from being included multiple times into the same *.c* file. This happens, for example, when you include *file1.h* and *file2.h* into *file3.c*, and *file1.h* itself also includes *file2.h*. Header guards prevent symbols from being defined multiple times in the same compilation unit. They are written like this:

```
#ifndef _MYINCFILE_
#define _MYINCFILE_

// All your header content goes here
    // (function prototypes,
    // extern variables, typedefs, etc)

extern int aVariable;
    // I am a global variable declared somewhere else

int myfunction(int a);
    // I am a function prototype
    // I have a head but no body
#endif
```

The *#ifndef*, *#define*, and *#endif* are preprocessor directives. During the first phase shown in Figure 1, the preprocessor evaluates if a macro called

*_MYINCFILE_* exists. If not, the content in the *#ifndef* is processed, i.e., its copied by *cpp* into the *.c* file including the header, and the *_MYINCFILE_* macro is defined. If this *if* is evaluated again, the content inside will not be copied by *cpp* again. See Section 3 for more information on macros.

You might find code that uses this single line at the beginning of the file:

```
#pragma once
```

The effect is the same as the header guards, but *#pragma once* is not supported by all compilers.

2.3 Where does the compiler look for header files?

It varies. The compiler will try to find library headers (like *stdio.h* and *math.h*) in directories that are different between operating systems. But since it is already configured to look for libraries in the proper place, you only need to worry about the location of your own headers. If you have all *.c* files and *.h* files in the same directory, the preprocessor will be able to find them, otherwise, you must specify the search directory for headers when calling the compiler:

```
gcc myfile.c -I./inc1/ -I./inc2/
```

The *-I* flag tells the compiler to look for headers in *inc1*, and *inc2*, which are subfolders inside the folder (i.e. working directory) that you have called *gcc* from. See Section 16 for other compilation flags.

---

3 Macros

3.1 Using Macros to Define Constants

As previously mentioned, macros are preprocessor directives. A macro is essentially the definition of a symbolic name for value or expression. For example:

```
#define VALUE 20

int a = VALUE;
int b = a + VALUE:
int c = function1(b, VALUE);
```

When the preprocessor replaces a macro name with its content, this is called **expansion**. Defining constants with macros allows you to keep the code cleaner and to more easily interpret what your expressions are calculating. Also, if you need to change *VALUE* across the entire code (which could be hundreds of files and thousands of lines), you only have to change the macro definition. You can even change macro values when you call the compiler, for instance:

```
gcc main.c -DVALUE=15
```

For this compilation, *VALUE* will be 15. You can use this to generate many executables with different parameters without changing the source code. One

useful macro (which you will likely use) is defined in *math.h*, and defines the value of π to many decimal places, with the name *M_PI*.

3.2 Using Macros to Define Expressions

Another use for macros are expressions. As you may have concluded, the preprocessor basically replaces every occurrence of the macro name with what you have defined. This means you can define the macro as anything. For example:

```
#define BIT(a) (1 << a)

int a = BIT(2);
    // the third bit of "a" will be 1,
    // the others will be 0
```

The macro shown above accepts one argument, and before compilation, it is replaced with the expression using that argument. This macro is just a convenient and readable way of setting a single bit of a variable to 1. Macros are usually named with all capital letters, to distinguish them from function calls. Also note that the contents of the macro are enclosed in parenthesis, which is to prevent the following type of error:

```
#define BIT(a) 1 << a
    // I forgot the parenthesis!

int a = BIT(2) * 2;
    // This macro will expand like this:
    // int a = 1 << a * 2;
        // this means "a" will be multiplied BEFORE
        // the shift, due to operator precedence
```

---

4 Basics of Variables and Data Types

4.1 Declaring Variables

Programming is all about manipulating data. Variables are basically names given to positions in memory where you data is stored. In *C*, variables must have a type (more on types in Section 4.2), and must have unique names. Variables can be declared anywhere in *C*: inside the *main* function, any other function, or outside functions (i.e., global variables):

```
// file starts here
#include<stdio.h>
    // include whatever you need

int a = 4;
    // A global variable

int funcA(int var1) {
    int result = var1 * a;
        // A local variable
```

```
    return result;
}

int main() {
    int result = funcA(8);
}
// file ends here
```

Notice on how the *integer* variable in *main* has the same name as the variable declared inside *funcA*. Doesn't this violate the rule of unique names? No, because the variables are declared in different **scopes**. Names must be unique only inside the same scope, and each function (both *main* and *funcA*) define their own scope. However, declaring a variable called *a* (of any type), inside either function would **not** be allowed, since *a* is declared on the **global** scope.

To be precise, it's not the function that defines a scope, it's the curly brackets ({}). You can define scopes by enclosing any block of code inside curly brackets. For example:

```
// file starts here
#include<stdio.h>
    // include whatever you need

int main() {

    // one scope
    {
        int a = 2;
        // (...) more code here...
    }

    // another scope
    {
        int a = 4;
        // (...) more code here...
    }
}
// file ends here
```

Its also possible to define scopes inside scopes, and inner scopes will have access too all names of their parents, but not vice-versa.

4.2 Data Types of Variables

The (basic) types of variables defined in C include:

- char (8 bit)
- short (16 bit)
- int (at least 16 bit)
- long (at least 32 bit)
- long long (at least 64 bit)
- float (a 32 bit representation of real numbers)

- double (a 64 bit representation of real numbers)

As you have noticed, some types have an undefined number of bits. This is because what the compiler does with these types depends on the specific processor you are compiling the code for. Why is this important? Because the number of bits you use to store a value determines how large (or how small) the represented value can be. For 8 bits, you can represent values between 0–255, or between -127–128, if you are using a signed value. Variables are *signed* by default, if you want an *unsigned* variable, you declare it with that modifier:

```
unsigned char a = 0; // I go from 0 to 255
```

The maximum representable value for an unsigned variable is computed with $2^n - 1$, where n is the number of bits. The number of bits of a datatype is also called its *bitwidth*. To make sure that you have the exact size you want, its typical to include *stdint.h*. This library defines variable types (with *typedef*, see Section 6.6), which ensure a specific bit length, and if the variable is signed or unsigned. These types include:

- uint8_t (unsigned *int* with 8 bit)
- uint16_t (unsigned *int* with 16 bit)
- int32_t (signed *int* with 32 bit)
- etc.

Note that *stdint.h* only defines *integer* types, i.e., no real (*float*/*double*) numbers.

4.3 Implicit and Explicit Type Casting

To understand type casting, and data types in general, you have to understand that variables are only names to positions in memory that contain a certain number of bits. So what happens when you assign a variable to another variable? It depends on their types. For example:

```
char a = 12;
char b = a; // "a" is the same type as "b"
            // the value of "a" is simply
            // copied to the location of "b"
```

This is the most straightforward example. The two other cases are: the assignment of a variable to another with a smaller bitwidth, and vice versa. The behaviour also depends on whether the variables are signed or not:

```
// 32bit Signed to 8bit signed
int a = 1024;
char b = a;
    // An int is 4 bytes, and a char is 1 byte
    // During the assignment, only 1 byte is copied
    // Since the value a is holding exceeds 1 byte,
    // the actual value of "a" is lost
```

```
// 8bit Signed to 32bit signed (source is positive)
char c = 56;
int d = c;
    // This is the opposite situation,
    // and "c" is correctly copied to "d"

// 8bit Signed to 32bit signed (source is negative)
char e = -12;
int f = e;
    // "e" is one byte, with the value 0xF4
    // (in binary: 11110100)
    // Since both variables are signed,
    // the compiler performs "sign extension",
    // so instead of copying a single byte to "f":
    // 0x000000F4 (this would be a positive number),
    // the most significant bit of "e" are extended
    // to all bytes of "f": 0xFFFFFFF4

// 8bit Unsigned to 32bit signed
unsigned char g = 24;
int h = g; // This works

// 8bit Signed to 8bit unsigned
unsigned char i = -24;
int j = i;
    // i = 0xE8 (in hexadecimal)
    // The one byte of "i" is copied
    // but sign extension is not performed
    // This means that "j" will hold a value
    // equal to 232 (0x000000E8)
```

All these conversions (type casts) are **implicit**, meaning that the compiler determines what conversion to perform, if it can. If you want, you can call *gcc/g++* with the *-Wconversion* flag, and warnings will be printed for all conversions that may fail.

The other type of cast is the explicit cast:

```
int a = 1024;
unsigned int b = (unsigned int) a;
```

The type specified between parenthesis defines the target type. In this example, it is not required (or very useful) since the compiler knows the source and destination types. But explicit casts are useful for intermediate calculations:

```
char a = 32;
char b = 64;
char c = (char) ( ((int) a * (int) b) >> 8 );
    // 32 * 64 = 1388, which exceeds a single byte!
    // 1388 = 0x056c
    // By shifting 0x056c to the right by 8 bits
    // we want to save the most significant byte,
    // 0x05, into "c"
```

Without casting a and b to *int*, the intermediate result of their calculation would be a single byte. In other words, the result **overflows**, and the second byte is lost. Since we want to save that byte to c, by shifting right by 8 bits (and discarding $0x6c$), the result of c would instead be zero (0x00). By telling the

compiler to use intermediate, unnamed, *int* variables to store the calculations, we get the behaviour we want.

Explicit casts are also necessary when using dynamic memory allocation, and *void* pointers. See Sections 7.1 and 7.2. Finally, you may ask why we simply don't use the *double* type for every variable, since it its 64 bit wide can can represent real numbers. The reasons are the amount of memory required, and computing efficiency. Typically, calculations performed on *integers* are faster than *float/double*.

4.4 Typed vs. Untyped Languages

Unlike Python, PhP, or MATLAB, the C/C++ language (as well as many other compiled languages) are strongly typed. This means that explicit types are given to variables when they are declared, and that a variable cannot change type. In other words, the following is **not** valid C/C++:

```
int aVariable = 2;
aVariable = "a string";
```

The identifier *aVariable* is used to refer to a specific location in memory where a *integer* datum is stored. The name *aVariable* refers only to this, and cannot be reused for other purposes. Specifically, C is statically typed, meaning that *you* define the type of a variable before compilation, and that it does not change. Although the most recent revisions of C support automatic variables, they are still statically typed. Example of an automatic variable declaration:

```
auto variableX = 4.5;
// The compiler can determine that this is a "float"
```

4.5 Variables in Header Files?

Never. See Section 2.1. In headers you should define macros, function prototypes, and make variable **names** visible by using the *extern* modifier (see Section 13). Two things can happen if you define variables in headers, depending on whether or not your headers have *header guards* (which they should):

- If you don't: If you define a variable in a header file, and then include that header in multiple *.c* files, then the same variable name (i.e., *symbol*) will be defined multiple times. It is easy to generate these errors if you do this, especially if you include several headers, and those headers also include each other.
- If you do: The header guards will prevent the headers from being included multiple times into the same compilation unit (*.c* file), which seems to solve the problem. But during linking, each *.o* will have effectively **different** variables with the **same name**, and linking will fail due to duplicate symbols.

5 Operators

Operators in *C/C++* are similar to operators in other languages you may know. Operators accept one operands (unary), two operands (binary), or three operands (ternary). The latter case is explained in Section 8. Operators can also be classified by function, for instance, as *logical*, *arithmetic*, or *bitwise*. The following code summarizes all operators:

```
// the assignment operator (=)
int a = 1, b = 2, c = 3;

// arithmetic
a = b + c; // addition, can cause overflow
a = b - c; // subtraction, can cause underflow
a = b / c; // division, can cause undefined
           // behaviour due to division by zero
a = b * c; // multiplication, can cause overflow
a = b % c; // remainder operation (integers only)
a++; // increment by 1
a--; // decrement by 1
b = a++; // assign "a" to "b", then increment "a"
b = ++a; // first increase, then assign

// short forms arithmetic
a += 2; // same as "a = a + 2";

// comparison (i.e., relational)
a = (b > c); // returns "0" or "1", depending on comparison
a = (b < c);
a = (b == c); // equality
a = (b != c); // inequality
a = (b <= c);
a = (b >= c);

// logical
a = (b == c) && (b < c); // "a" is "1" if both conditions are "1"
a = (b == c) || (b < c); // "a" is "1" if either condition is "1"
a = !(b > c); // negation of the condition result

// bitwise operators (bit by bit operations)
a = 1; // in binary --> 0001
b = 2; // in binary --> 0010
c = 3; // in binary --> 0011
a = a & b; // logical AND, 0000 = 0001 & 0010;
a = a | b; // logical OR, 0011 = 0001 & 0010;
a = ~a; // logical NOT, 1110 = ~0001;
a = b ^ c; // logical XOR, 0001 = 0010 ^ 0011;
a = a << 1; // shift left, 0010 = 0001 << 1;
a = c >> 1; // shift right 0001 = 0011 >> 1;

// shifts in C are signed operations
short d = -20; // in binary --> 1111111111101100 (16 bits)
d = d >> 2; // "1"s are shifted in, sign is retained
            // d = 1111111111111011

// ternary operator
c = (a == b) ? 2 : 4;
```

```
// see Section 8.4
```

See also section Section 4.2 for details on results of operations in function of the bitwidth of the operands.

---

6 Arrays, Structs, and Unions

In the last section we saw the basics of variables. This section covers advanced topics like (static) arrays of variables, *structs*, and how we can attribute custom names (i.e., aliases) to data types.

6.1 (Static) Arrays

Arrays are a way to define a sequence of variables of the same type, under a single name. You may declare arrays of any variable type, by specifying that type, the array name, and its size:

```
int a[20];
    // Twenty ints
```

Each value in the array is commonly called an *element*. The array in the example is a **static array**, meaning that is has a fixed size, which cannot be changed after declaration. Notice how the array is declared, but unlike variable declarations we saw before, its values are not **initialized**. There are multiple ways to initialize array values:

```
int a[5] = {0}; // everything is zero
int b[5] = {1,2,3,4,5}; // each element has the specified value
int c[5] = {[2]=4}; // second element is "4", others are zero
```

To access specific elements of arrays, we use the index operator ([]), like this:

```
int a[5]; // Five elements
a[0] = 2; // which go from index "0"
a[4] = 6; // to index "4"

int i = 2;
a[i] = 4; // Using integer variables (int, short, char)
          // as indexes, is also allowed
```

There is no maximum size to a static array, in terms of what the language itself allows, but in practice, you will have limited memory. Trying to use very large static arrays is the kind of "error" (or bad coding practice) that will only manifest during compilation (it may fail), or execution (performance will be poor). It is also possible to declare arrays of multiple dimensions, but this document will not cover that. Instead, see Section 7.2 for information of arrays of variable size.

6.2 Structs

The *C* language allows you to define custom data structures. *Classes* are more sophisticated than *structs*, and are only supported in *C++*, which this document does not discuss. The main difference between the two is that *structs* can only contain *members*, and *Classes* can contain *members* and *methods*. A *member* is any variable of any data type, a *method* is a reference to function of any kind. This is an example *C struct*:

```
struct aName {
    int a, b;
    char byte1;
    int data[20];
};
```

This **defines** a *struct* of type *struct aName*, which contains all the variables declared within: three *ints*, one *char*, and 20 more *ints* in an array. Understand that *structs* are basically variables of a custom type. So we can declare a variable of this type, and access its *members* (or *elements*):

```
struct aName s1; // Declare a variable of type "struct aName"
s1.a = 20;       // Members of structs can be accessed with "."
s1.data[2] = 4;
```

We can even do this:

```
struct anotherTypeName {
    struct aName a1;
    int avalue;
};
```

That is, a *struct* definition may contain a declaration of another type of *struct*. However, it **can not** contain a declaration of a *struct* variable of its own type (although it can contain a pointer to a *struct* of its own type, see Section 7.1).

You **can**[1] however, do this:

```
struct aName a1; // Declare one struct
struct aName a2; // another one;

a1.a = 2;    // Put some values in struct "a1"
a2 = a1;     // This WORKS
```

It is up to the compiler how to copy the members of the structure. This is not usually an issue, but different compilers might generate the same *struct* with a different layout for the bytes that it contains. Copying by assignment (i.e., the = operator) will let the compiler handle this on its own. An alternative is to use *memcpy*:

```
struct aName a1;
// put stuff in a1

struct aName a2;
```

[1] Note: versions of this document previous to v0.55 claimed incorrectly that structure copying through assignment was not possible

```
memcpy(&a2, &a1, sizeof(struct aName));
    // this will copy the struct byte by byte
```

However, either method results only in a **shallow copy!**. This is not a problem for the structures show above, but if your *structs* contain pointers to arrays allocated on the *heap* (see Section 7.2), then the compiler only copies the pointers themselves. In other words, you **are not** creating a copy of the arrays they point to. For example:

```
struct bName {
    int avalue;
    char arr[20];
    int *p;
};
// this struct contains 4 + 20 + 4 bytes

int main() {

    struct bName b1;
    b1.avalue = 2;
    memset(&(b1.arr), 0, sizeof(char) * 20));
        // fill "arr" with zeros

    strcpy(&(b1.arr), "Hello!\n");
        // put some text in the array

    b1.p = (int *) malloc(10 * sizeof(int));
    // some code to put values in b1.p[1], etc

    struct bName b2;
    b2 = b1;
        // This WORKS
        // BUT, its only a "shallow" copy

        // this "works", "avalue" is copied,
        // and "arr" is copied byte by byte
        // BUT b1.p and b2.p are pointing to the SAME array
        // since only the value of the pointer itself is copied
}
```

The assignment technically works, depending on what the desired effect is! The value of *avalue* is copied, the *arr* array is copied byte by byte, since its bytes are inside the *struct* itself, but *b1.p* and *b2.p* are pointing to the **same array** since only the value of the pointer itself is copied.

If we want to copy everything, including *heap* allocated data associated to the *struct*, we need a **deep copy**:

```
// assume we have the code initializing "b1"
// from the previous example here...

// now we declare "b2"
struct bName b2;
b2 = b1;
    // we can still use the assignment, WITH CAUTION
    // knowing that the copy isn't deep, yet
```

```
// now we must allocate a new array for b2
b1.p = (int *) malloc(10 * sizeof(int));

// and copy the array
memcpy(&(b2.p), &(b1.p), sizeof(int) * 10));

// rest of the program goes here...

// at the end (or when we don't
// need them anymore), we must free both
free(b1.p);
free(b2.p);
```

Code becomes much better structured if you implement functionalities like this in functions, see Section 12. Naturally, for different types of *structs*, you will need different deep copy functions, depending on what they contain.

Finally, you can also declare arrays of *structs*:

```
struct aName a[20]; // Twenty structs
a[0].a = 2;
```

6.3 Unions

While *structs* allow you to group any number of variables into a named sequence, *unions* allow you to reserve a block of memory of a given maximum size, which you can access in several ways. A declaration of a union is identical to the declaration of a *struct*:

```
struct mstruct {
    int aVar, bVar;
    char cVar[8];
};

union munion {
    int aVar, bVar;
    char cVar[8];
};
```

The important different is that, while the above *struct* occupies 16 bytes, (two *ints* and 8 *char*), the *union* only occupies **eight bytes**. The *struct* actualy contains two integers, and either characters, which you can access with the respective variables names. With the *union*, you can use those variable names to access the bytes in the union in specific ways. For example:

```
union munion u1;
u1.aVar = 4;
    // this sets the first 4 bytes
    // of the union to = 00 00 00 04

u1.cVar[3] = 5;
    // this sets the fourth byte to 5.
    // Now the first 4 bytes look like:
    // 05 00 00 04
```

This feature of unions is especially useful to save memory in embedded systems (with small memory capacity), and to give you more control over writting to specific bytes via named identifiers.

Remember however that each named identifier is aligned to the beginning of the union:

```
union bunion {
    int wVar;
    char byte1, byte2, byte3, byte4;
};

union bunion b1;
b1.byte1 = 0x4;
b1.byte2 = 0x20;
    // if you try to access
    // each byte of wVar
    // with a union like this
    // it WON'T work
```

Each *char* in the *union* of the above example refers to the first byte of the reserved memory, unlike the previous example where an **array** of *chars* makes it so that each array index refers to a specific byte. Alternatively, you can declare a *struct* inside a *union*.

6.4 Unions with Structs, and Vice-Versa

To expand on the previous example, you can have differently named identifiers for your bytes if you ensure that each *char* variable refers to a different byte in the *union*, like so:

```
union dunion {
    int aVar;
    struct {
        char byte1;
        char byte2;
        char byte3;
        char byte4;
    };
};

// now this is possible
union dunion u1;
u1.byte2 = 0x40;
// makes aVar = 0x00004000;
```

By using *unions* inside *structs*, you can construct data types whose bytes you can treat differently, depending on context. For you instance, you could declare a generic vector, or binary tree of nodes, of the following *struct* type:

```
enum datatype = {INT, FLOAT};

struct cstruct {
    union {
        int data;
        float data;
```

```
    };
    // other fields...
    enum datatype dt;
};

struct cstruct c1;

// some code.. (e.g., function call)
    if(c1.dt == INT)
        c1.data = 2;
// more code...
```

The *union* allows you to store different types of data, while saving memory, and encapsulating it in a parent data type.

In all these examples, the internal *unions/structs* are defined inside their parent, and are **anonymous**. That is, they are nested definitions, and the type of structure defined has no **type name**. They also have no instance name, but we may do so if we wish:

```
struct cstruct {
    union { // still no type name
        int data;
        float data;
    } uName; // instance name
    // other fields...
    enum datatype dt;
};

struct cstruct c1;
c1.uName.data = 2;
    // now the fields of the union must be
    // accessed under the name of the union instance
```

6.5 Using Bitfields

Variables which are parts of *structs* or *unions* can have a specific number of bits, using the *width* field in the declaration of the variable. With this, you can modify the examples of the previous sections (Section 6.3, Section 6.4) to manipulate data with even further detail, and achieve some additional memory savings.

```
struct dstruct {
    int aVar : 1; // 1 bit
    int bVar : 2; // 2 bits
};
```

The number of bits of each variable cannot exceed the number of the type specifier (in this example, the 32 bits available to an integer, for both cases). The compiler packs the bits in sequence into the *struct*, and rounds up the size of the total size of *struct* to the largest storage type declared within, in this case, an *int*. That is, the above *struct* occupies 4 bytes, since that is enough to store the 3 required bits, and since we've specified *int* for the variable types. If both variables were of type *char*, the *struct* would require 1 byte.

When assigning values to the variables, the bit width is taken into account, and so is signedness:

```
struct dstruct d1;
d1.aVar = 1; // this is fine, aVar can be 0 or 1
d1.bVar = 4; // bVar will be ZERO, since "4" requires 3 bits
             // i.e. there is overflow

d1.bVar = 3;
    // "11" in binary, ocuppying both bits

cout << d1.bVar << endl;
    // will produce "-1"
```

Finally, we can apply bit fields to access individual bits of a variable by combining them with what we know about *structs* and *unions*:

```
union eunion {
    char aByte;
    struct {
        char b1 : 1;
        char b2 : 1;
        char b3 : 1;
        char b4 : 1;
    };
    // named access to
    // the first 4 bits
    // of "aByte"
};

union eunion a1;
a1.b1 = 1; // first bit
```

6.6 Defining Types

The *typedef* keyword allows you to give new names to data types. This is how the types explained in section 4.2, e.g., *uint8_t*, are defined. Usually, *typedef* is used to give *structs* shorter names. For example:

```
// Define and typedef the struct
typedef struct aName {
    int a, b;
    char byte1;
    int data[20];
}; aName_t;

// Declare the struct
aName_t a1;
```

7 Where do Variables go? What is Memory?

When you declare variables, their data is placed in memory. From the point of view of the *C* programmer (i.e., you), what happens exactly at the level of

the RAM and operating system doesn't matter. The memory in a *C* program (and virtually all other languages) can be modeled as table like the example in Table 1.

Addresses	Content (32 bit)
0x00	
0x04 (a)	2
0x08 (b)	4
(...)	
0x20 (c[0])	1
0x24 (c[1])	2
0x28 (c[2])	3
0x2c (c[3])	5
0x30 (c[4])	8
(...)	
0xff	

Table 1: A simplified look at the memory

On the left-hand side are the memory addresses, which we use to lookup specific contents in memory, on the right-hand size. The number of rows of the table is the memory size, and the range of addresses you can use to look up all the rows is called the **memory space**. Notice how the addresses increment by 0x04 from row to row. This is because this table represents the data in memory as pieces of 32 bit each, which corresponds to 4 bytes. It is useful to think this way, since most data types you will use in *C* are 4 bytes wide.

It is the compiler than chooses where to place data in memory when it is declared, so the memory positions shown are just examples, but this code would place those elements into memory:

```
int a = 2;
int b = 4;
int c[5] = {1,2,3,5,8};
```

Notice that elements of arrays are placed into memory **sequentially**, and its because of that we can **iterate** over arrays using loops (see Section 9). Think of declaring a variable as basically giving a specific memory address a name. We can then use that name to read/write to that position. You can figure out the address of a variable with the **dereferencing** operator &:

```
int x = 2;  // Assume "x" is in memory position 0x08;
int y = (int) &x; // "y" is now equal to 0x08;
```

7.1 What are Pointers? Referencing and Dereferencing

Pointers are extremely important in *C*. Most of the difficulties you will have with *C* programming will most likely come from not understanding pointers

and memory. Pointers are closely connected to the concepts of memory addresses, **referencing**, and **dereferencing**. The previous section closed with a dereferencing example where the address of a variable is placed into another variable. You'll notice how we had to cast the result of $\&x$ to *int*. This is because the dereferencing operator returns **addresses**, and addresses can only be held by special data types, called **pointers**:

```
int x = 2;  // Assume "x" is in memory position 0x08;
int *y = &x; // "y" is now equal to 0x08;
```

By prefixing the variable name with an asterisk, that variable becomes a pointer. It should only be used to hold addresses of variables which have the same type. In terms of memory, pointers are nothing more than variables whose **value is a memory address**. That address can be, but not necessarily, the address of another variable. The contents of the memory for the example above would look something like what is shown in Table 2.

Addresses	Content (32 bit)
0x00	
(...)	
0x20 (a)	4
0x24 (b)	0x20
(...)	
0xff	

Table 2: A pointer variable holding the address of another variable

In Table 1, we saw an array declared in memory. When you declare a static array, the name of the array is actually a pointer type variable. You may have noticed that to access the elements of the array we used the [] operator. This is a **referencing** operation. Referencing is when you take a pointer, and use its value (an address), to read/write the value at that memory address. For example:

```
int a[5];
int *b = a; // This is valid

// doing this
int x = a[0];

// or this, is the same
int y = b[0];
```

Notice how the & was not applied to a when initializing b. This is because a is of type *int **, i.e., *int* pointer. These examples have used the square bracket to perform referencing, but the actual operator is the asterisk symbol, *:

```
int a = 5;  // Assume "a" is at address 0x20
int *b = &a; // Save the address
```

```
// assign the value at that address to another variable
int c = *b;
    // * means "point to this address"
```

Note how the asterisk means different things when declaring the pointer, and when initializing *c*. In the first case, it indicates that the variable is a pointer, in the second case, the asterisk is an operator. In other words, this means that when you access elements of an array, this is happening:

```
int a[5];

// this
int b = a[1];

// is the same as this
int c = *(a + 1);
    // we take the base address,
    // add one position, and reference
```

When we take the name of the array, which is in fact a pointer to the start of the array, and add or subtract values to it, we are performing what is called **pointer arithmetic**.

7.2 Dynamic Memory Allocation

There are two ways in which memory may be declared in *C/C++*. So far we have seen static declaration of memory. For example:

```
int main() {
    int var1;
    int a[5];    // both variables are
                 // allocated statically on the "stack"
}
```

The *stack* is the memory region onto which variables, scalars or arrays, declared as shown above are placed. The allocation of variables to the stack is done at *compile time*. To be precise, every scope (i.e., blocks of code contained within curly brackets) has its own stack. In functions, for example, it is used to store the arguments passed to the function, and the local stack from which functions are called is used to store the location to which to return when the function ends.

Regardless, the greatest implication you will encounter is that arrays allocated on the stack must be of a known fixed size. This means that the following code, **isn't** valid:

```
int main() {
    int var1;
    cin >> var1;
    int a[var1];    // the compiler doesn't know the value of var1
}
```

An easy solution to handle arrays of arbitrary size, is to enforce a maximum size, and reserve that much memory statically:

```
int main() {
    int a[1000];
        // I'm sure I won't need more
}
```

But this is far from being an efficient solution. To handle arbitrary memory requirements, we resort to the *heap*. The heap is the memory region onto which memory can be dynamically allocated. In *C/C++* the **dynamic allocation** of memory is handled by the *malloc* family of functions (whose definitions are found in the *stdlib.h* header).

```
int main() {
    int size;
    cin >> size;
    int a* = (int *) malloc(size * sizeof(int));

    // more code using "a"...

    free(a);
    return 0;
}
```

In the example above, the user inputs, **at runtime**, the size desired for the array of integers with the name *a*. The *malloc* function (memory allocate) reserves a region in the heap with the number of bytes specified (10 times the number of bytes of an integer), and returns a pointer to the start of that memory region. The memory region is **contiguous**, just like a static array.

Since *malloc* returns a *void **, we must static cast the address to our target variable. Additionally, *malloc* does **not** initialize memory, meaning that the allocated region will contain random data. To ensure that the allocated region is set to zeros, you can use *calloc* instead.

Finally, we **free** the memory, which means that the space it used to occupy on the heap can be reused for another allocation. Other languages, such as Java, manage the heap automatically, and clean up unreachable memory for re-utilization. This process is called **garbage collection**, but in *C/C++* the heap is managed entirely by the user code.

A critical difference between allocation on the stack and on the heap, is that memory allocated on the heap **survives** the scope its declared in. For example:

```
int* giveme(int nelem) {
    int a[20];
        // whatever I do with this array,
        // it disappears once the function ends

    int b* = (int *) malloc(size * sizeof(nelem));
    // now I put some values in b

    return b;
}
```

The memory allocated survives, but the *int b** point does not, since it is itself a variable allocated on the local function stack. In order to not "lose" the location of the allocated memory, we must store/return the address. If we lose

all references we may have to a region in memory allocated onto the heap, we suffer from what is called a **memory leak**.

```
int main() {
    int a* = (int *) malloc(10 * sizeof(int));
    // some more code...

    a = (int *) malloc(5 * sizeof(int));
        // another array

    free(a);
        // array nr 2 is free, but nr 1
        // is lost (leaked), it cannot be freed anymore
}
```

Finally, we may also allocate *structs*, or array of pointers, to construct multi-dimensional arrays:

```
struct mstruct {
    int a;
    char b;
};

int main() {

    // array of arbitrary structs
    struct mstruct *p =
        (struct mstruct*) malloc(10 * sizeof(mstruct));

    // 10 pointers to ints, allocated in a static array
    int *a[10];

    for(int i = 0; i < 10; i++)
        a[i] = (int *) malloc(5, sizeof(int));

    // now I have a 2D array of size 10x5
    a[0][2] = 5;
        // etc

    // at the end, I must free everything
    free(p);
    for(int i = 0; i < 10; i++
        free(a[i]);

    return 0;
}
```

---

8 Conditional Constructs

Conditional constructs allow you to verify the truth or falsehood of a specific statement, and to execute code based on the result of that verification. Common conditional constructs include the *if-else* clause, the *switch-case*, and the ternary

operator. The termination conditions of loops are derivatives of these types (see Section 9).

8.1 *if-else*

Almost all programming languages you may have used employ, at least, the *if-else* clause.

```
int a = 5;

// the comparison operator, ==, returns either "0" or "1"
if(a == 5) {
    a = a + 1;

} else {
    a = a + 2;
}
```

All code within a pair of brackets after an *if* statement will execute if the condition of the *if* evaluates to a value **different from zero**. This means that, in the following example, the code **will** execute:

```
int a = 5;

// the code in the if will execute
if(a) {
    a = a + 1;
}

a = 0;
// and now it wont
if(a) {
    a = a + 1;
}
```

Notice also that an *if* does not require an *else* to accompany it, but an *else* must always be preceded by an *if*. Additionally, the code above is equal to the following:

```
int a = 5;
if(a)
    a = a + 1;
```

That is, the *if* brackets may be omitted (from either the *if* or the *else*) if the conditional code to execute consists only of one *statement*. However, it is usually good practice to always use the brackets for clarity.

8.2 *Chaining "if-elses"*

Any statement that you cloud place outside an *if-else* may be placed within (i.e., anything except declarations of functions). This means that an *if*, or *else*, may contain any number of *if-elses*:

```
int a = 5, b = 3;
if(a) {
```

```
        a = a + 1;
        if((b / a) < 1) {
            b = b + 1;
        } else {
            b = b - 1;
        }

    } else {
        // ... more code
    }
```

Be careful however when writing code that ends up being a sequence of deeply nested *if-else* clauses. This makes the code much more likely to fail, and difficult to debug and read. If you really need too, code like the following is more useful and easy to read:

```
if(a) {
    // .. code

} else if (b) {
    // ... more code

} else if (c) {
    // ... more code

} else if //...
```

This type of sequence of *if-elses* differs from the first in that the conditions are independent. That is, there are no *ifs* inside *ifs*, only after *elses*. This means that all conditions are checked in sequence, and that there is an implicit **priority** to the conditions. For example, if the first condition (*"a != 0"*) is true, then the code inside will execute, and no other *ifs* will be evaluated. If all conditions are **mutually exclusive**, then the order of the *ifs* is irrelevant. Otherwise, you must be careful in how you order state the conditions, based on the behaviour you want.

8.3 *switch-case*

The *switch-case* clause is similar to a sequence of *if-else* in that one of multiple blocks of code may be executed based on the value of a given expression (or variable):

```
int a = func();
    // assume "a" will have
    // some random value

switch (a) {
    case 1:
      // code
      break;

    case 2:
      // more code
      break;
```

```
    case 3:
    case 4:
        // much more code
        break;

    default:
      // code for all other values of "a"
      break;
}
```

A *switch-case* allows you to determine which code should execute based on the possible values of the *expression* placed inside the *switch* statement. The example uses a variable, but this expression could be the return of a function, or a comparison (although that would only evaluate to either 0 or 1). The *case* statements mark the start of the block of code which will execute if the expression evaluates to the value specified in the *case*. To end the block of code, a *break* statement is required. Without a *break*, execution will continue, crossing over from one *case* to another. This is not usually the behaviour you want, but it allows you to specify the same block of code for two (or more) different cases, like the example shows. The *default* case captures all values which are not listed by other cases.

8.4 Ternary Operator

The ternary operator is basically an *if-else* in a single line, with a single statement. It allows you to do short conditional attributions, like this:

```
int a = (b == 2) ? 2 : 4;
    // if the condition is true, "a" is g
    // given the value 2, otherwise
    // it is given the value 4
```

The code to the left of the colon character (*:*) will execute if the condition is true, and the code to the right will execute if the condition is false. The code may include function calls, or even another ternary operator:

```
int a = (b == 2) ? funcA(2) : 4;
int b = (a == 2) ? ((c == 4) ? 1 : 2) : 6;
```

---

9 Loops

Loops are code structures that allow you to iterate over data, and execute a block of code repeatedly, typically under the control over an iteration counter and/or other conditions. Types of loops include: the *for* loop, the *while* loop, and the *do-while* loop.

Loops are composed of their header (where conditions for execution are determined), and the body, which contains all code to execute. A loop body may contain any statement that is valid within a function body, i.e., function calls, declaration of variables, or other loops.

9.1 *for* Loop

A *for* loop allows you to use its header to specify certain starting conditions for variables, stopping conditions, and increment/decrement/control operations to execute **after** the body:

```
int a[5] = {1, 2, 3, 4, 5};

int sum = 0;
for(int i = 0; i < 5; i++) {
    sum = sum + a[i];
}
```

The header is the line containing the *for* keyword. Within the parenthesis are three statements separated by semicolons. Using the first statement, you can (typically) declare and set the initial values of one or more variables. In this case, we've declared and set the **iterator** variable *i*. The second statement **must** evaluate to either true or false ($!= 0$ or 0), and is used to terminate the loop when **false**. The third statement includes code which executes after the second statement is evaluated, but before the loop body. It is not obligatory to define all three statements. For instance:

```
int a[5] = {1, 2, 3, 4, 5};

for(int i = 0; ; i++) {
    if(a[i] == 4)
        break;
}
```

In the example above, the second statement is omitted (you must still provide the separation via semicolons). This means there are no stopping conditions, and we must exit the loop through other means (in this case a *break*). Note that if no value in array *a* was equal to 4, this loop would continue indefinitely (i.e., this is a very academic example).

As was mentioned before, each of the three header statements may be a list of statements. For example:

```
int a[5] = {1, 2, 3, 4, 5};

// two initialization statements
int sum;
for(sum = 0, int i = 0; i < 5; i++) {
    sum = sum + a[i];
}

// a compound condition
for(int i = 0; i < 5 && sum < 6; i++) {
    sum = sum + a[i];
}

// two increment/decrement statements
for(int i = 0; i < 5 && sum < 6; i++, sum = sum/2) {
    sum = sum + a[i];
}
```

9.2 *while*, and *do-while* loops

A *while* loop's header contains only the evaluation of a condition. The loop will execute while that condition is true. Alternatively, just like the *for* loop, you may exit from a *while* or loop at any time with a *break* statement.

```
int i = 0;
int sum = 0;
int a[5] = {1, 2, 3, 4, 5};

while(i < 5) {
    sum = sum + a[i];
    i++;
}
```

A *do-while* loop is similar to a *while* loop, but is useful in situations where you want the loop body to unconditionally execute at least once:

```
int i = 0;
int value = 0;
int a[5];

// the condition to execute the body again
// is evaluated after the first execution
do {
    cin >> value;
    a[i] = value;
    i++;
} while(value != 0 && i < 5);
```

In the example above, a *do-while* loop is used to read inputs from the terminal, while the input from the user is different than 0 **and** the number of values read is fewer than 5.

9.3 Nested Loops

When a loop is written as part of the body of another loop, it is referred to as a *nested loop*. You may nest loops indefinitely, but this quickly becomes confusing and inefficient. You can also nest different types of loops. This example shows a single level of nesting:

```
int a[5] = {1, 2, 3, 4, 5};
int b[5] = {2, 2, 3, 4};

int prodsum = 0;
for(int i = 0; i < 5; i++) {
    for(int j = 0; j < 4; j++) {
        prodsum = prodsum + a[i] * b[j];
    }
}
```

In this situation, the first loop is commonly referred to as the *outer loop*, and the second as the *inner loop*. For greater levels of nesting (i.e., more loops inside loops), the first loop is the *outermost loop*, and the most nested loop is the *innermost loop* (there may be several loops at the same nesting level).

In the example, notice how the iterator of the inner loop is named *j*, and not *j*. This is because the inner loop is inside the scope of the outer loop, and therefore a variable with the name of *i* already exists. On the other hand, the variable *j* only exists in the scope of the inner loop.

When using the *break* statement in nested loops, remember that the break only exits the loop scope it is executed in. That is, you can not exit the outer loop by executing a break on the inner loop. If you find yourself having difficulties controlling the execution flow of your program in a situation with many levels of nesting (including loops, but also *if-elses*, etc), consider restructuring your code.

---

10 Input/Ouput

Check back later!

---

11 State Machines

A state machine is a construct that allows you to implement a functionality based on states, and on transitions between those states, based on specific conditions. The following figure shows an example state machine with four states, and the conditions that trigger transitions between the states:

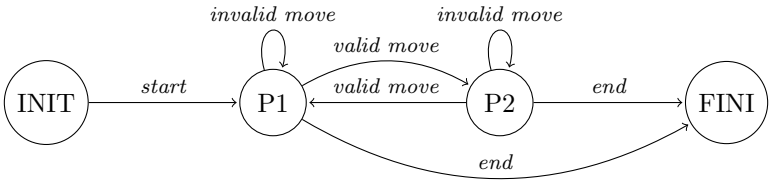

Figure 2: Example Finite-State Machine

The example illustrates a scenario where a generic game is played between two players. A given starting condition changes the state to *P1*, where the first player must make his move. Depending on whether the move is valid or invalid (for a given set of game rules), the state either transitions to *P2*, or remains in *P1*. The same logic applies for the remaining states. A *C* implementation for the above machine is as follows (contains some pseudo-code):

```
enum state {INIT, P1, P2, FINI};

int main() {
    enum state st = INIT;
    while(1) {
        switch(st) {
```

```
                case INIT;
                    // some code...
                    if(<condition is met>)
                        st = P1;
                    break;
                case P1:
                    // get player 1 move
                    if(<valid move> && <game not ended>)
                        st = P2;
                    else if(<valid move> && <game ended>)
                        st = FINI
                    break;
                case P2:
                    // same code as P1, or similar
                    break;
                case FINI:
                    // output relevant information
                    st = INIT; // restart the game
                    // alternatively, code a condition
                    // to terminate the "while(1)"
                    break;
            }
        }
        return 0;
    }
```

12 Functions

Functions in *C* are blocks of reusable code, useful for containing functionalities that you want to use repeatedly, and to help you create more readable and organized code, since it helps you identify the purpose of a specific block of code. This is a (very simple) function definition:

```
// A function is declared, and defined
int aFunctionForAddition(int a, int b) {
    return a + b;
}

// The function is called
int result = aFunctionForAddition(20, 10);
// result will be "30"
```

A function is defined by: the data type of the variable it **returns**, in this case an *int*, by its name (which in *C* must be unique), and by a list of **arguments** separated by commas. There is no limit to the number of arguments, and they may all be of different data types.

The above example **declares** and **defines** the function at the same type, because it states the prototype of the function (i.e., the head), and then defines the code inside the function (i.e., the body). Like we saw in Section 2.2, its possible (and correct) to declare only the function prototype in header files, but you can also do it in *.c* files. Why is this useful? Because the compiler reads

the file line by line. If you try to call a function in line 10 of your code, which is only declared and defined in line 20, that will produce and error. This could be resolved by "swaping" the position of the functions, but this solution has two problems: 1) for large code, you constantly have to keep track of where functions are called, and 2) it does not solve the problem of two functions calling each other. Instead, we can use **forward declaration**:

```
// A function is declared
int funcA(int a, int b);
    // Now the compiler know's I exist,
    // and what I look like

// Another function uses funcA
int funcB(int a) {
    return funcA(a, a);
}

// funcA is defined
int funcA(int a, int b) {
    return a + b;
}
```

Combine this with what you know about headers (Section 2.1), and you learn that when you have a header *file1.h* where you have declared all the prototypes for all the functions in *file1.c*, and then include *file1.h* in *file1.c* (at the top), what you have done is forward declare all the functions.

More elaborate things can be done with functions, especially if you resort to pointers, see Section 12.2.

12.1 Recursive Function Calls

Functions may call themselves, which is another way to implement loops. A typical recursion example in *C* is sum all numbers up to a specified number *"n"*:

```
// Forward declare!
int calcSum(int n);

// Recursive function
int calcSum(int n) {
    if (n == 0)
        return n;
    else
        return n + calcSum(n - 1);
}
```

In this example, the function calls itself with a modified input argument, and stops when that argument is 0. The result will be a sum which equals too $n+(n-1)+(n-2)+(...)+0$. There must always be a condition to terminate the recursion, otherwise the function will call itself "infinitely", and eventually the program will crash, due to **stack overflow**.

12.2 Passing Arguments by Value, Reference, or Address

In the previous section we have seen arguments being passed to functions by **value**. That is, functions were declared like this:

```
// a function declaration were arguments are passed by value
void func(int a, int b, int c);
```

This means that when the function is called, the arguments you pass to the function are **copied** to the function's local scope. That is, the variables are copied to local variables, and all modifications to their value will not manifest themselves back in the scope from where the function was called. For example:

```
void func(int a, int b, int c) {
    a = b  + c;
}

int main() {
    int a = 2;
    int b = 4;
    int c = 5;

    func(a, b, c);
    // after this call, "a" is still equal to 2
}
```

The other two methods to passing arguments to functions are **references** and **pointers**, which are functionally similar. The following function receives a value by reference:

```
// a function declaration an argument is passed by reference
int func(int &a);
```

Passing by reference is indicated by the ampersand character (&). (Note: do not confuse this with the dereferencing operator used to retrieve the addresses of variables, see Section 7.1). When passing an argument by reference, the function receives (as the name indicates) a reference to the variable, meaning that all operations performed on that variable within the function, *will be* reflected on the original variable in the calling scope. For example:

```
int func(int &a) {
    a = a * 2;
}

int main() {
    int a = 2;
    func(a);
    // after this call, "a" is "4"
}
```

Passing arguments as pointers, i.e., passing variable addresses, is similar:

```
// a function declaration where a
// variable's address is passed
// to a pointer variable local to the function
int func(int *a) {
   *a = *a * 2;
```

```
}

int main() {
    int a = 2;
    func(&a);
    // after this call, "a" is "4"
}
```

Notice how the function prototype now receives a **pointer** to a variable, and not a reference of a variable. This means you must pass to the function the address of a variable, which you can retrieve with the dereferencing operator &. Although similar, there are some differences between passing by reference or address (i.e., pointer).

For example: references cannot be *NULL*, while pointers can (i.e., their value is 0, meaning they are not a valid pointer); references are read only, whereas you can modify a pointer after passing it to a function; you can employ pointer arithmetic over a passed address to iterate over arrays (i.e., *a++* or *a[2]* are valid). These differences are due to the fact that when passing by pointer, you are essentially passing a copy of a variable's address, as a value to a local variable within the function which is a pointer to the type of address passed.

12.3 Variadic Functions

You may have noticed that functions such as *printf* accept a variable number of arguments. For example:

```
int a = 5, b = 2;
char arr[20] = "Hello!\n";
printf("Some numbers and a string: %d, %d, %s", a, b, arr);
    // 3 inputs (4, counting the formating string)

printf("More: %d, %d\n", a, b);
    // 2 inputs
```

What kind of function prototype does *printf* have to be able to accept a variable number of inputs? These functions are called **variadic functions**, and you may write them like this:

```
#include <stdarg.h>

// the ellipses (...) indicate a list of
// arguments of unknown type and size
int func(int n, ...) {

    int sum = 0;
    va_list args;
    va_start(args, n); // variable list starts after "n"
    for(int i = 0; i < n; i++) {
        int nextvalue = va_arg(args, int);
        sum += nextvalue;
    }
    va_end(args);
    return sum;
}
```

The function does not know what are the types of variables that are passed, so it is up to you to properly cast the variable types. The *printf* function does this through the format specifiers in its first argument, such as *"%d"* and *"%c"*.

---

13 Some Modifiers/Qualifiers

Modifiers are keywords typically placed before declarations of variables or functions which alter their *properties*. In *C* these keywords include: *extern*, *const*, *static*, *volatile*, and *register*.

13.1 Extern

Check back later!

13.2 Const

When a variable is declared as *const*, this means that any attempts to write a new value to it will result in a compilation error:

```
// a constant variable
const int a = 5;

a = 6;
    // This will generate a compile error
```

This modifier allows for the compiler to make optimizations by assuming a certain value will never change, and prevents you from committing errors, especially when passing variables references, or pointers, to functions:

```
// a function which receives a read-only pointer
void func(int *const a, int b) {
    *a = *a + 4;
        // this is fine

    a = &b;
        // this is not
}
```

In the example above, the *const* modifier prevents you from accidentally the value of *a*. In other words, *a* will always point to the same variable (since it receives that variable's address). You can however change **the value of the variable to which *a* points**. The opposite is also possible:

```
// a function which a pointer to a read-only variable
void func(const int *a, int b) {
    *a = *a + 4;
        // this not fine

    a = &b;
        // this is
}
```

To summarize:

```
const int a;     // a read-only variable
int const *b;    // pointer to read-only variable
const int *b2;   // this is equivalent to the above
int *const c;    // read-only pointer
                 // (i.e, c always points to the same address)
const int const *d; // read-only pointer to ead-only variable
```

13.3 Static

The *static* keyword can be applied to both variables, and functions. Then declared within a function, a static variable will **retain** its value, even after the function ends. You may think of it as a state variable of the function:

```
int acc(int a) {
    static int accumulator = 0;
        // initialized once at compile-time

    accumulator += a;
        // will hold its value throughout function calls
    return accumulator;
}
```

In other words, a static variable is permanent within its scope. When the *static* keyword is used on a global variable, or on a function declaration, it makes those symbols visible **only** within that scope.

```
static char var1;

static int func(int a) {
    return a * a;
}
```

This means that you may have two global variables with the same name, each in a separate *.c/.cpp* file, and no errors will occur. However, the actual purpose of static variables and functions is that you let the compiler know that they will only be used within that single file, which allows for more aggressive optimizations during compilation.

13.4 Volatile

You will use this keyword rarely, but in some occasions you may write code which interacts with peripheral devices. Those devices may write to positions in memory, which you access via a variable. However, the compiler is not aware of this, and the variable may **appear** to be either written by your code, but never read, or vice-versa. It is therefore useless, and the compiler will optimize the code by removing it. You can prevent this by declaring it as *volatile*:

```
volatile int var1;
    // This variable is modified outside the code
```

14 Classes

Check back later!

---

15 Templates

Check back later!

---

16 Compilation Flags

Check back later!

Compilation flags, or options, are parameters passed to the *gcc/g++* compiler that control the code generation process. The *gcc* compiler alone supports a very large number of options, which you can view (some of them) by calling *gcc* from the command line as such:

```
gcc --help=common
```

The ones you should most commonly use are the optimization level options, the warning display options, specification of *include* paths, and links to libraries:

```
gcc main.c -Wall -O2 -I./inc/ -lm
```

The above command compiles a file named *main.c* file. The *-Wall* options instructs the compiler to output *all* warnings. The *-O* option is the optimization level. The arguments for this option are:

```
-O0 : optimization level 0
-O1 : optimization level 1
-O2 : optimization level 2 // most commonly used level
-O3 : optimization level 3 // sometimes breaks the code
-Os : optimization for size
    // generates the smallest binary possible;
    // execution might be slower, but -Os
    // can be useful for embedded systems
    // with small memory
```

The *include* option specifies where the compiler may look for header files (you can specify this option multiple times). Finally, the *-l* option tells the linker to link your code with a pre-compiled dynamic library (a *.a* file in unix, and a *.dll* file in Windows) named *"m"* (which is the math library associated with the *math.h* include).

When compiling from an IDE (e.g., Visual Studio or similar) you typically don't directly access the terminal, so have to control the compilation options by configuring the project in the IDE.

---

17 Common Compilation Errors

Check back later!

---

18 Indentation

Check back later!

Errata

- 31/03/2020: In Section 6.2, it was incorrectly claimed, in versions prior to v0.55, that copying *structs* through direct assignment (i.e., =) was not possible. Posterior versions explain the differences between shallow and deep copying.