C for Java Programmers

I have to say thanks to Jason Maasson from Frije Universiteit Amsterdam for his excellent script and slides. I've translated most of the slides and added some graphics and text.I would also like to thank Marshall Brain, founder of "www.howstuffworks.com" for his wonderful article on "How C Programming Works" which explains also the c-runtime environment.And last but not least Steven Simpson from Lancaster University for pointing out the differences between both languages on a few excellent pages.Look at the resource section at the end for links.

Introduction

Goals

1. **Learn how the C language differs from Java**
2. **Learn to create C programs with the standard C library**
3. **Understand the necessary C libraries for networking etc.**
4. **Learn the necessary tools: compiler, linker, assembler, archiver, debugger**
5. **Understand the C-runtime environment: memory areas, stack handling, calling frame**
6. **Understand why C has problems with buffer overflows and memory management**

Roadmap

1. C and its time - a short history
2. Main differences to Java
3. Language types and keywords
4. Functions
5. Pointers
6. The Standard C library
7. Tools: Preprocessor, C-Compiler, Linker, Archiver, Debugger
8. The C Runtime (Memory areas, stack etc.)

C and its time

In the seventies C was created by Kernighan and Ritchie - the fathers of Unix. They derived C from a previous language "B" and used it to write the first versions of Unix on PDP11 machines. Those machines were slow compared to our standards today and system programming was usually done in assembler. C had the following revolutionary features:

1. **A high-level language suitable for system programming**
2. **A portable language that could run on many different systems**
3. **A FAST language with little overhead**
4. **An easy to learn language with a small set of keywords and little restrictions**

The C syntax is used in many newer languages like C++ and Java. But those languages are object orientied in nature and the similarities with C fooled many programmers into believing that it was only a small step from C to OO.

The power of C

Modern languages try to restict side-effects as much as possible. Object state should be changed by going through interfaces (methods) and nothing else. C statements can access any part of the programs data directly because memory addresses are available. This means that a simple C statement can change data without overhead - but also without protection.

Some C compilers allow insertion of assembly language statements right in the middle of C code to modify registers or use special functions from the CPUs instruction set.

C strings and other data structures are NOT bounds checked at runtime. This is of course faster than with bounds checking

The C runtime creates little overhead. No garbage collection e.g.

C types are defined with "best size per CPU" and not with an absolute length. This way an "int" integer type can be 16 bit, 32 bit or 64 bit, whatever suits your hardware best. Interoperability across machines becomes of course more difficult.

The weaknesses of C

Memory management is left to the programmer. Releasing memory to early leads at runtime to strange bugs caused by the program overwriting its own data or heap foilgroups

Functions without bounds checking are causing so called buffer-overflow errors which can be used to take over machines. C has buffer-overflows practically built in - as a glance at the weekly security reports e.g. from cert.org shows.

How C is different from Java

Differences between C and Java

1. **C is a procedural language, Java is object oriented**
2. **C's main structuring element are functions in files. Java organizes code in classes and packages, thereby providing better namespaces.**
3. **C code gets compiled into machine code for a specific CPU. Java code gets compiled into bytecode which is interpreted on a virtual machine (JVM). Java code can get compiled into machine code as well (just-in-time compilers etc.)**
4. **C does not have exceptions. Errors are handled through return codes.**
5. **C types like structures (objects without methods) can be allocated on the stack. Java needs to allocate all non-primitives on the heap using expensive memory management functions (e.g. "new").**
6. **Arrays and strings in C are not bounds-checked. It is the programmers responsibility to stay within the allocated bounds**
7. **C lets programmers access memory addresses directly through the use of POINTERS. Pointers are variables which contain a memory address. Text (code), data, heap and stack areas are all within a programmers reach. Java lets only the virtual machine access a programs stack, e.g. to perform security checks.**
8. **Java primitive types are fixed in length. C types can vary per machine. This lets C take maximum use of hardware specifics (e.g. 16 bit register size vs. 32 bit register size). It also creates portability problems.**

Differences between C and Java (Continued)

9. **C has a preprocessor and include files. The preprocess works like a macro processor which substitutes macros in the program file.**

Example Program in C

In most examples we will use the latest C standard (C99) which brings the C syntax even closer to C++ and Java . File simple.c:

```
#include <stdio.h>   // this imports a definition file (a header).

double value;    // some variable definition on global scope

/* this is another form of comment. Here we use it to describe both
   parameters of main: argc is the number of
   commandline parameters given to the program.
  **argv is an array of pointers pointing to the parameter values.
   The program can use it to access its parameters.  */

int main(int argc, char **argv)   // note the return type of main: 0 means ok
{
    int local=0;
    value = 0.42;       // value is defined outside of this block

    // a function call into the standard C library.
    // Note that the number of parameters is variable!!

    printf("local = %d value = %f\n", local, value);
    return 0;           // tells whoever started the program that all is well
}
```

Compiling a C program

`gcc simple.c`gcc is the gnu c-compiler. It will translate simple.c into an executable. The default name for the executable is a.out.`gcc -o simple simple.c` will create the executable with the name `simple`. Per default the C-runtime library is also linked during the translation process. `./a.out` or `./simple` will produce the expected output: `local = 0 value = 0.420000`

Keywords of the C-language

Table 1. There are 32 keywords

unsigned	**continue**	**typedef**	**volatile**
const	**break**	**default**	**do**
double	**else**	**enum**	**extern**
float	**for**	**goto**	**if**
int	**long**	**register**	**return**
short	**signed**	**sizeof**	**static**
struct	**switch**	**case**	**union**
auto	**void**	**char**	**while**

The philosophy behind was to keep the language clear and simple. How many keywords and API calls do you really use/understand? The windows system call interface e.g. has more than 600 functions!!!

Types

C Language Types

A type defines the size of the memory area that a variable of this type would occupy. It also defines how a variable of this type can be used - which operations are legal on this type. And last but not least does a type define a name as a reference to it. C types are either primitives (int, char, long) or composite (struct, union). The composite types are built from primitive types.

Example 1. Structs in C

```
struct controller {
                        unsigned char output;
                        unsigned char input;
                        unsigned int control;
                };
```

Note

The actual size of a variable in memory is also determined by the so-called "alignment": Rules on which memory addresses certain types can start. This leads sometimes to "padding" bytes - empty areas within or between types. Compiler arguments control the alignment.

C Language Types (Continued)

Size of C types

The size of c types differs between machines and compilers. Typical sizes of an "int" are:

- **16 bits on Palm or Lego brick**
- **32 bits on a typical Intel/AMD PC with Pentium or Athlon**
- **64 bits on Alpha workstations or PCs with AMDs hammer cpu**

The "natural" size is simply the width of a CPU register.

C and Java types

Table 2. size differences

char	16 bits	8 bits
short	16 bits	16 bits
int	32 bits	16,32 or 64 bits
long	64 bits	32 or 64 bits
float	32 bits	32 bits
double	64 bits	64 bits
boolean	1 bit	1 bit or int
byte	8 bits	use char
long long	N/A	64 bits
long double	N/A	80,96 or 128 bits

C used to have no boolean types but the C99 standard introduced this type.

Note

With the `sizeof` keyword one can get the size of a C type in a certain runtime environment.

C and Java types (Continued)

```
int size = sizeof(int);
```

would return either 2, 4 or 8 (bytes).

Unsigned types

- ```
  int i1;          // range -2,147,483,684 to 2,147,483,684

  signed int i2;   // range -2,147,483,684 to 2,147,483,684

  unsigned int i2; // range 0 to 4,294,967,296
  ```

Java does not know unsigned types, just like ADA. In a mixed language environment (e.g. C operating system, ADA applications) this can cause trouble. Also watch out for sign extension when you shift a variable to the right (division). A signed variable will introduce new "1" bits in the most significant position when it is shifted to the right.

Example 2. Shifting signed types

```
signed char foo = 0xFF;     // 11111111
                        signed char bar = foo >> 4; // 11111111


unsigned char foo = 0xFF;     // 11111111
                        unsigned char bar = foo >> 4; // 00001111
```
```

Unsigned types (Continued)

Integer used as boolean

Older C code did not have a boolean type. Instead, integers were used. Comparisions for true or false were represented as integer comparisons: 0 meant false, >0 meant true.

Example 3. Integers substitutes for boolean

```
int x = 100;
if (0)  { // not reached }
if (42)  { // reached }
if (x==4)  { // not reached }
if (x)  { //  reached }
while (x--)  { //  100 times reached }


if (x=100)  { // reached but most likely a BUG }
```

Boolean Operators with Integer types

```
1 && 0  == 0
1 && -3 == 1
7 || 25 == 1
!34     == 0
```

Creating an alias for a C-type

The typedef keyword allows the definition of an alias name for a known type. The most important reason to do so is to achieve better portability and evolution of C programs. Remember: The same C type (e.g. int) can have different sizes on different hardware or compilers. To make a program easily adjustable to different type sizes one defines new types with a program-specific meaning. If the program is ported to new hardware with different int size, only the typedef file is adjusted and the program is recompiled on the new platform.

```
file myTypeDefs.h:
   typedef char byte;
   typedef int INT32;
   typedef short INT16;
   typedef long INT32;

used in myProgram.c:

   #include myTypeDefs.h

   byte b = 0xff;  // only the programs own type definitions
   INT32 i = 12345; // are used.
```

Using own types for portability

Suppose the previous program is ported to a platform where int types are only 16 bit wide and only long types have 32 bits. With all our types definitions collected in one place: myTypeDefs.h, it is easy to make the adjustments and recompile:

```
file myTypeDefs.h:          changed to:
   typedef char byte;       typdef char byte;
   typedef int INT32;       typedef int INT16;  // now 16 bit
   typedef short INT16;     typedef short INT16;
   typedef long INT32;      typedef long  INT32; // use long not int
```

No need to change the program besides recompiling it. The typedefs make sure that a variable that is supposed to be 32 bit wide will be created with the proper type: int on the first platform, long on the second platform.

Using own types for program evolution

Suppose you are using a char type for a variable. Later on you discover that it should better be a long type. But the knowledge about the "char" type is spread all over your program, e.g. by using "extern char ...". If you change the type of your variable you must change all these places in your software which know that the variable was a "char". Using a typedef and defining an alias for "char" with a program-specific meaning solves this problem.

Another advantage is that the MEANING of a variable can be much better communicated.

```
typedef char* FileName;

char* c = "ConfigFile";   // generic type name
FileName f = "ConfigFile"; // specific type name
```

Don't overdue this feature. A new type name is nice but the user (programmer) does not automatically know how it needs to be used. The usage of a char* is clear.

Pointer types

C allows a programmer direct access to memory locations via memory addresses. A pointer type declares that a variable of its type contains an address pointing to some address in memory where a certain variable is stored. The size of a pointer type is implementation dependent but usually just int. (Yes, the size of a pointer itself can be larger than what it points to).

```
int i = 5;

int* iPointer;  // a variable of "pointer to int" type is declared

iPointer = &i;  // the pointer variable contains now the address of i

*iPointer = 10; // the content of i now changed to 10

printf("content of i %d :" i); // gives 10
```

The ampersand operator & represents the address of a variable. And a pointer is never just a pointer alone. It is always a "pointer to" some type. The asterisk operator takes the content of the pointer - a memory address and goes to this address to retrieve what is stored there. Pointers are very powerful as we will see but lets say the *pointer = 10; statement would be in a different file. Then suddenly i would change and nobody would know why.

How pointers work

Why pointers?

One of the major advantages of pointers is simply performance. Pointers save a lot of copies. A piece of code can access some complex and large structures through a pointer reference. That's the good news. The bad news is that with a pointer reference any piece of code can manipulate the referenced variables and those errors are very hard to find. It also makes life for a garbage collector extremely hard: How should the collector know when a variable is no longer used? An example:

```
// allocate memory on the heap to store an Address type
Address* addressPointer = (Address*) malloc(sizeof struct Address);

fillAddress(addressPointer); // get Address filled in.
shareAddress(addressPointer); // let others use Address

free(addressPointer);   // give memory back to heap
```

So far so good but how does the programmer know when to call "free" to release the memory? What if the function shareAddress has stored the pointer somewhere and tries to use it later? Once "free" is called the memory can be re-used and the results of accessing it with an old pointer are simply undefined. The program can crash quickly or subtle logic problems can appear.

Variables in C

Note

Variables in programming languages need to be destinguised with respect to “visibility” and “lifetime”. The term “scope” combines both and leaves it sometimes unclear what was meant.

visibility	**The places where a variable that has been defined somewhere can be used. A variable may not be visible for certain sections of code and therefore not be usable there. But this does NOT imply that the variable is no longer used at all and could be collected.**
lifetime	**Only when there is NO code for which the variable is visible can it be safely collected. This is e.g. the job of a garbage collector.Automatic variables (stack variables) are visible ONLY within the function that defines them. C as well as Java are so called "block-oriented" languages. A variable defined in an outer block is visible inside but not vice versa.**

Visibility

```
int foo = 1;

function bar(void)
{
   int foo = 2;       // not visible outside of this function
   int glob = 2;       // not visible outside of this function
   for (int foo=3;.....)   {

   }
}

function foobar(void)
{
   int temp =foo;   // temp = 1
   for (..;.;.)   {
        temp=2      // temp is visible in embedded blocks
   }
   temp = glob;     // error: glob is not visible in this block
}
```

Lifetime

```
int foo = 1;           // globally visible. Lives until end of pr

function bar(void)
{
   int temp = 2;        // not visible outside of this function
   return;
}                       // exactly here does the lifetime of the
                        // temp variable end. ALL memory within t
                        // bar has been allocated on the stack.
                        // At return the stack is rolled back to
                        // it was BEFORE the function was called
```

Global and Local Variables

C knows two basic types of variables with different scope:

1. **Global variables are defined outside of functions on module level (except static, see below)jjjjjjjj**
2. **Local variables are defined within functions**

```
int global;        // outside function

void someFunction(void)
{
  int local;     // within function
}
```

Global Variables

They can be used through tree different modifiers:

const	**defines a constant variable on global scope. The compiler will pack those variables together with executable code into the text segment which is read-only protected by the operating system.**
extern	**an extern declaration makes a global variable defined in another file accessible in the file with the extern declaration. The c-module (file) declares that it will use the external variable in its code. After compilation it is the linkers responsibility to find the file with the global definition of the variable and "link" it together with the referencing code.**
static	**a variable declared as "static" is visible only within the file with the definition. All functions in this file can use the variable. The variable is NOT a stack (automatic) variable and will NOT cease to exist after a function call. (Same behavior as in Java)**

Examples with global variables

The following code is split in two files. One file contains definitions of global variables. The other file uses some of them. Note that the "static" variable from fileOne.c is NOT the same as the "static" variable in fileTwo.c

```
/* fileOne.c: */

    int global     = 2;       // defines a global integer variable
    const int FOUR = 4;       // a constant variable (text segment)
    static int privat = 5;    // all functions in fileOne can use
                              // Not visible outside of file



/* fileTwo.c: */

   extern int global;         // declares that fileTwo.c wants to use the
                              // variable defined in fileOne.c
   static int private = 7;    // A new variable only visible within fileTwo.c

   void function(void)
          {
            int local = global * privat;  // local variable == 2*7
          }
```

Examples with global variables (Continued)

Local Variables

They can be used only within functions. Older C compilers will only accept definitions of local variables BEFORE the first line of code (at the begin of a function). This has been changed with the latest C99 standard. The following modifiers are available:

register	**Gives the compiler a hint that this variable will be frequently used in the code (e.g. counter in a loop) and that it should be put into a CPU register (cached) for faster access time. Nowadays compilers do their own performance analysis and will detect the best register use anyway.**
static	**The variable is declared within a function and would therefore disappear when the end of the function is reached (a stack or automatic variable). But the static keyword make the compiler allocate the variable space NOT on the stack but in persistent memory and the variable will NOT lose its contents between function calls**
volatile	**A compiler may put a variable into a register or other cache for better access performance. This is OK if there is only one way that this variable can change and the compiler controls it. This is not always true: shared memory e.g. lets variables change from 2 or more processes. The compiler does not know that "behind his back" somebody changes the variable and would still use the stale copy (cached version) of the variable. The volatile keyword forces the compiler to always read the variable fresh from memory.**

Local Variables (Continued)

Volatile Keyword

Examples with local variables

```
void function(void)
 {
    int local = 0;          // local (automatic) variable allocated on stack of fu
    static int foo = 0;     // declaration and initialization of foo
    foo++;                  // every time the function is called foo would
                            // be incremented
 }                          // end of function. "local" would disappear. foo stays

 function();                // first call to function, foo is 1, local is 0
 function();                // second call, foo is 2, local is 0
```

Arrays and Strings

Java and C differ significantly with respect to arrays and strings. In general Java arrays and strings are of referential type while in C both are merely blocks of memory with a name. The indirection used by Java gives more control at the price of lower performance. From a safety point of view C arrays and strings are dangerous because a violation of array limits usually results in a runtime error possibly much later than the violation itself.

Java Arrays

1. A Java array is a reference to an array object. This implies that the array can be created without knowledge of the array size.
2. A Java array reference can only be allocated on the heap (persistent). An array can therefore be allocated within a method and returned to the caller without becoming invalid.
3. A Java array can be assigned to another array reference. The original array memory is then collected vy the garbage collector.
4. An exception is thrown if the array is accessed with an index lower or higher than the array definition. This is a runtime check performed by the virtual machine. No adjacent memory is affected by such an overwrite error.

```
int [] a1, a2;   // no size given
a1 = new int[8]; // the array memory allocated on heap

int [] a3 = {1,2,3,4};  // direct initialization
a2 = a3;    // a2 and a3 now reference the same memory
a2[10] = 5;  // ArrayOutOfBoundsException thrown
```

C Arrays

1. **A C array is the name of a piece of memory. Specifically it is a name for the starting address of this block of memory.**
2. **A C array can be allocated on the heap (persistent) or on the stack (temporary). A stack array can therefore be allocated within a method and returned to the caller. This results in the receiver of the array address getting an invalid heap address - without noticing it.**
3. **A C array cannot be assigned to another array because it is no reference. But the contents of one array can be copied into another - hopefully respecting the array limits.**
4. **No exception is thrown if the array is accessed with an index lower or higher than the array definition. No runtime check is performed by the C runtime system. Adjacent memory is simply overwritten and can result in spurious bugs. This approach trades safety for speed.**

Array Examples

```
int [] a1   // Error, no size given
int a2[];   // Error, no size given
a1 = a2; // Error, not a reference

int a4 [2][2];       // ok, size is known
int  a3 [] = {1,2,3,4};  // direct initialization
a3[10] = 5;  // memory 11 integers away from the array start
             // is silently overwritten
```

Array Lifetimes in Java

```
File.java:
  int [] function() {
         int [] myArray = new int[10];  // allocates myArray on HEAP
                                        // do some processing with array
         return myArray;         // the REFERENCE is copied back to caller
  }

  int [] someArray = function();  // gets a reference to memory on HEAP
                                  // all is well!
```

When the call to function() returns the local variable myArray becomes invalid. It will be collected by the garbage collector because it is no longer reachable from anywhere in this program. But the VALUE of myArray has been copied back to the caller and stored in the array variable someArray. This means that the memory allocated in function() is STILL REACHABLE from within your program via someArray. It will not be collected (yet). The caller has received a reference to valid memory.

Array Lifetimes in C

```
File.c:
  int [] function() {
         int myArray[10];        // allocates 10 integers on STACK
                                          // do some processing with array
         return myArray;         // the startaddress of the memory
                                 // ON THE STACK is copied back to caller
  }

  int someArray[] = function();  // gets a stack address back (where myArray
                                  // USED TO START). This memory is NO LONGER
                                  // allocated to function() or caller and can be
                                  // re-used anytime.
```

When the call to function() returns becomes the local variable myArray AND the associated array memory invalid. But the address of this memory has been copied back to the caller and stored in the array variable someArray. This means that the memory allocated in function() is STILL REACHABLE from someArray. Unfortunately the memory is no longer allocated and can be re-used during the next function call. The caller has received a reference to invalid memory.

It is possible in C to allocate the array memory on the HEAP just like in Java. We will compare the performance of stack vs. heap allocation below. For now just look at a correct array allocation:

Array Lifetimes in C (Continued)

```
File.c:
  int [] function() {
     int myArray[] = malloc(sizeof(int)*10); // allocates 10 integers on HEAP
                             // do some processing with array
     return myArray;         // the startaddress of the memory
                             // ON THE HEAP is copied back to caller
  }

  int someArray[] = function();  // gets a HEAP address back (where myArray
                                 // starts). This memory is still
                                 // allocated but ownership has been
                                 // transferred SILENTLY to someArray.
```

Note

Note that the OWNERSHIP of the memory changed. The caller MUST release the allocated memory when it is no longer used. Otherwise a "memory leak" happens and the piece of memory from the heap is lost to the program. It stays allocated but if someArray is thrown away without a "free(somearray);" call then nobody can reach the memory anymore. Unlike Java C has no garbage collector looking out for those orphans and collecting them. Memory management is the programmers responsibility.

Strings in Java

Strings are full-blown objects in Java. Some of their characteristics:

1. **Strings are objects and are copied by reference. (Unlike Java primitive types which are copied by value like integers)**
2. **Java Strings are IMMUTABLE. Once initialized they cannot be changed. This has the big advantage that the Java VM can share equal strings by making the string references point to the same memory location. Many applications (like e.g. parsers and scanners) use a lot of the same string tokens. They would be allocated only once in Java. Construction takes a bit longer as the string has to be hashed to generate the memory address.**
3. **Strings are complex objects and keep metadata about the characters contained, e.g. length. This allows strings to have gaps and the string length can be determined quickly by looking at the length field. C needs to run through the string to detect its length as we will see.**

Strings in C

C Strings are simply an array of "char" type elements. There is no length field.

1. **A C string must be terminated by a "\0" nul character. If you want a string of 10 characters you need to allocate 11 characters to make room for the nul character.**
2. **String manipulation functions rely on the final nul character and will overwrite memory mercilessly if it is missing.**
3. **Like java strings they are copied by reference. A string reference can be used to manipulate the string in any way.**
4. **Like arrays strings can be allocated on the stack which makes allocation/deallocation extremely fast.**
5. **The same memory management and responsibility rules as for arrays apply. This results in many programmers creating local copies of strings "just to make sure" that they own the memory behind.**

String manipulation functions in C

Many functions exist to read, write or copy strings from and to different sources and destinations. The following lists some popular functions.

```
int strlen(char  s[]);  // returns the lenght of string s
char* strcopy(char dest[], char source[]);  // copies  the source string into the d
                                // Note that dest must be at least as big as source
                                // adjacent memory will be overwritten. If dest is l
                                // we will have a memory leak.

 int strlen(char * s);  // same as above but using pointers (see below)
```

String handling in C is best explained by showing the implementation of strlen functions:

```
int strlen(char s[])  {
    int x = 0;
    while (s[x] != '\0')  // not string end yet
       x++;               // increment index
    return x;
}
```

String examples

With pointers those examples would look a litte different: the array declarations would be replaced with "char *". The rest is the same.

```
char name0[6];
name0[0] = 'J';
name0[0] = 'a';
name0[0] = 's';
name0[0] = 'o';
name0[0] = 'n';
name0[0] = '\0';  // don't forget to terminate the string

// shows nicely that C strings are actually arrays:
char name1[] = { 'J', 'a', 's', 'o', 'n', '\0' };
char name2[6] = "Jason";  // you can specify the array length or
char name3[] = "Jason";  // leave it to the compiler
// leave it to the compiler to avoid memory leaks.
char name4[100] = "This is not 100 characters long: waste of memory";
// BTW: whenever you use string literals like these examples
// the strings will end up in non-writeable text segment
```

Complex objects: Enumerations, Structures and Unions

C does not have classes but it allows the creation of complex datastructures. We will see how these datastructures evolved into the classes of C++ - the object-oriented extension of C.

Enumerations

Enumerations in C are types which are restricted to certain constants. Variables of an enumeration type can only be set with values from the enumeration.

```
enum workdays { monday, tuesday, wednesday, thursday, friday };

enum workdays today;

today = tuesday;
today = friday;
today = sunday;  // Error at compile time.

enum weekend { saturday = 1, sunday = 2 }; // one can assign any value to enum c
int foo = monday;  // the enum type is not necessary.
```

In Java the following construct is used to create constants:

```
interface Date  {
   public static final String MONDAY = "MONDAY";
}

String someDay = Date.MONDAY; // uppercase is only a convention
```

A small note on software changes and types

C, C++ and Java are strongly typed languages. Types are used so that a compiler can at compile-time check whether certain assignments make sense. Enums are type definitions - otherwise the compiler could not check the validity of an assignment to an enum declared variable. This is convenient and prevents stupid bugs. But the downside of this is that those enum type definitions become public visible - they are part of what is called in C++ or Java the INTERFACE of a class or module. And the consequence is that if you add another constant to an enum definition, this IS AN INTERFACE CHANGE and usually requires that ALL source code that knows (depends from) this interface needs to get recompiled. If you expect frequent additions to your enum types better use simple strings. This trades compile time security against runtime flexibility.

C Structures

C does not have classes like C++ or Java. But it lets us define datastructures which could be called "classes without methods". In Java this would be represented by a class with no methods (also no constructor) and public members.

```
struct point {
       int x;
       int y;
    };

struct point foo = { 0, 0};  // struct is part of the name
struct point bar;
       bar.x = 1;
       bar.y = 2;

struct point foobar;
     foobar = (struct point) { 4, 4};  // anonymous struct in C99
```

The same assignments using pointers

```
struct point {
        int x;
        int y;
    };

struct point bar;
struct point* barPointer = &bar; // barPointer points to bar
        barPointer->x = 1;       // -> instead of .
        barPointer->y = 2;
```

From struct to class: structures with functions

Object-Orientation means keeping data and functions together. C++ has something between C structs and full-blown classes: structs with functions. Here functions are added to the struct definition. those functions operate only on the struct members. The biggest difference to real classes is that these structs with functions are not polymorphic.

C Unions

A union is a datastructure that can hold different data types but NOT AT THE SAME TIME. A union will contain either an instance of a type X OR an instance of a type Y. It is the programmers responsibility to make sure that on retrieval only the same type is read that was stored in the union before.

A union is therefore a polymorphic type but unlike Java or C++ the polymorphism is not taken care of by the language runtime. It is the programmers job.

```
union number {  // number does NOT contain ALL
    char c;     // of i,c,f and d. Only ONE at
    int  i;     // at time
   float f;
  double d;
};

union number n;
int j;

n.i = 10;
j = n.i;  // n.d, n.f, n.c all would have been possible
          // but wrong
```

Making unions safe

This example shows how a programmer can use an extra variable which stores the type of the latest write access to make sure that the proper read access is done.

```
union number n;
enum { CHAR, INT, FLOAT, DOUBLE } accessType;
n.i = 10;
accessType = INT;

// on retrieval:
switch (accessType)  {
case CHAR:
   // get n.c
   break;
case INT:
   // get n.i
   break;
case FLOAT:
   // get n.f
   break;
case DOUBLE:
   // get n.d
   break;
}
```

In many years of kernel and other c-hacking I have found little use for unions....

Making unions safe (Continued)

Unions compared to regular structs simply save some memory because the allocate enough memory for the largest type defined within the union - in the above case "double" and use this space for every type in the union.

C Functions

In a procedural language functions are the only way to structure code. To really understand C one must understand how a function call works: What kind of role does the stack play? How are parameters and return values transported (reference vs. value). This will take us down into some assembly language as well.

With such a small set of keywords like in C or Java the functions are what makes those languages usable. Without functions C could not even read anything from a keyboard or show anything on a screen.

Function Declarations

There are three ways to declare a function in C:

1. **A function with no parameters is declared like this:**

```
int foo(void) { /* body */}  // note that void is needed
```

2. **A function with a fixed number of parameters is declared like this:**

```
void bar(int p1, double p0) { /* body */}
```

3. **And last but not least a function with variable parameter lists:**

```
double func(int p1, int p2, ...) { /* body */}
```

Where do funtions live?

At runtime	**During program execution functions live in the text segment of a program which is immutable. Each function has an address just like a data variable but in a different segment. The function name is only an alias for programmers. Internally functions are known only by their address.**
At compile time	**When code is compiled (see below) the code statements are usually collected in archives (libraries). If a program needs a function, the proper library is "linked" to the program and it learns about the address of the function. The address is at runtime used to call the function.**

Funcion Prototypes

In Java a class needs to be known before it can be used. Java uses the "import" statement for this purpose. Importing a class allows the compiler to match the intended use against the class definition and prevent stupid bugs already at compile time.

C functions need to be declared as well before they can be used. Because functions are usually called from a different location than their own file this declaration is best put in a place where everybody can get to it: a header file:

```
in Example.h:
  int max(int, int);  // simply declares max as taking two integer
                      // parameter and returning an integer

in Example.c:
  void example(void) {
      int result = max(4,5);
}
  int max(int one, int two) {
    return (one < two? two: one);
}
```

Using max from a different module (file):

```
#include "Example.h"          // "imports" the declaration
```

Funcion Prototypes (Continued)

```
  void someFunction(void) {
    int result = max(200,100);
      .....
}
```

While this seems almost a no brainer one has to remember that C initially did not have function prototypes

The main function

Unlike Java the main function is a special function in C. It is the first programmer defined function that is called (we will see that there is a lot going on in a program before main is actually called). Its job is to transport command line parameters into the program and to return a result value to whoever started the program. That is why main returns an integer value.

```
int main(void) { /* body */ }  // no command line args
int main(int argc, char** argv) { /* body */ } // with args
```

The expression "char argv" stands for a pointer to pointer to characters. We know now that a string is a pointer to characters. So we have a pointer to string(s). And those strings are the command line arguments. Another way to express this would be char* argv[] which is an array of pointers to characters (strings).**

Why is parameter handling easier in Java?

```
static void main(String [] argv) {
       for (int i=0;i<argv.length;i++)
         String arg = argv[i];..
```

The reason is simply that an array in Java has a length that can be queried. A C array's length must be kept in a separate variable: in this case "argc", the argument count. The main functions body needs first to check argc and then it knows how many arguments are there. The first argument string btw. is always the programs name.

But now another question comes up: How do those command line arguments get into the program? If main is a function than somebody must have set up the arguments in a way that they look to main just like regular parameters. The answer ist: the C-runtime system, together with the operating system, perform this trick. And the same goes for environment variables like "path" or "home".

Example of command line parameters

```
file example.c:
   int main(int argc, char **argv)
   {
      int i;
      for (i=0; i<argc;i++) {
         printf("%s\n", argv[i];) // note the switch from
                                  // pointer to array view.
      }
      return 0;
   }

   compile and run:

   gcc example.c

   ./a.out one two three

   a.out
   one
   two
   three
```

The printf family of functions

Printf is the main print function in C. It has many variants like fprintf (prints into a file) or sprintf (prints into a string) as well as its reverse functions (scanf etc.) which take an input and dissemble it into different parts. The syntax of all those functions is pretty similiar. First comes a so called format string and then a variable number of parameters.

```
int printf(const char *format, .....);
```

"const char* format" is the format string. It tells the printf function how many parameters and of which type follow. Now we can guess how the code for printf might look (symbolicly):

```
int printf(const char *format, .....) {
   1. read the format string and look for
      special characters
   2. List those special characters and
      get the parameters for which they stand in
   3. Display every parameter in the requested
      format.
   4. Tell caller how many parameters were
      processed. }
```

Printf Format String

The following meta-characters can show up in a format string. They stand for parameters in the order of their appearance. They can be qualified e.g. with the number of digits which should be printed.

- **%d % signed int**
- **%u % unsigned int**
- **%x % hexadecimal unsigned int**
- **%c % character**
- **%f % double and float**
- **%s % string (array of characters)**
- **%% % To print a %**

Printf Examples

```
printf("Hello World");
int value = 5;
printf("Hello %d world\n", value);

char c = 'a';
printf("val = %d c = %c\n", value, c); // format string tells
                                       // printf to first expect
                                       // an integer and then a char
char[] world_str = "world";
printf("Hello %s\n", world_str);
```

Can you guess what happens if this piece of code is executed?

```
char[] world_str = "world";
printf("Hello %s\n");  // no second argument!
```

The format string tells printf to look for a string parameter which is not there - does printf recognize the mistake? Or will it just take something for a string pointer? The answer requires some knowledge about how c-functions work.

How to learn about functions

The best online source for c-functions are the man pages on a Unix system. `man - S 3 printf` will print a lot of information on your screen:

```
PRINTF(3) Linux Programmers Manual PRINTF(3) Name printf, fprintf, sprintf,
snprintf, vprintf, vfprintf, vsprintf, vsnprintf, - formatted output conversion
SYNOPSIS #include <stdio.h> int printf(const char *FORMAT [, ARG, ...]); int
fprintf(FILE *FD, const char *FORMAT [, ARG, ...]); int sprintf(char *STR,
const char *FORMAT [, ARG, ...]); int saprintf(char **STRP, const char *FORMAT
[, ARG, ...]); int snprintf(char *STR, size_t SIZE, const char *FORMAT [, ARG,
...]); DESCRIPTION `printf' accepts a series of arguments, applies to each a
format speci- fier from `*FORMAT', and writes the formatted data to `stdout',
termi- nated with a null character. The behavior of `printf' is undefined if
there are not enough arguments for the format. `printf' returns when
.............. :
```

Resources

The resources cover freely available information as well as excellent books right to the topic. All entries are commented to let you know what a paper or book is all about. I also expect participants to use this literature in case of questions.

Open Source Information on C programming

The following information is freely available and taken together is an excellent introduction to the subject.

1. **Jason Maassen, C for Java Programmers. A complete introduction to C for Java people. I have used and extended his slides for this lecture but you should read the background information here as well.60+ pages. Good if you need to prepare for a test or need some more information about a feature from my slides. Find his C course with other materials here: http://www.cs.vu.nl/~jason/course.html [http://www.cs.vu.nl/~jason/course.html].**
2. **Steven Simpson, Learning C from Java. An experienced Java programmer will get the most from this short paper focussing on the differences. Excellent. h/ http://www.comp.lancs.ac.uk/computing/users/ss/java2c/diffs.htm [http://www.comp.lancs.ac.uk/computing/users/ss/java2c/diffs.html].**
3. **Marshal Brain, How C programming works. Another excellent paper from www.howstuffworks.com. This really explains complicated memory problems using pointers, how array overwriting can happen etc. And many useful pointers to other computing related topics like memory organization, operating systems etc. www.howstuffworks.com**
4. **Allen Downey, How to think like a computer scientist [python|java|C++]. Excellent introduction to those programming languages. Requires no advance programming skills. http://www.ibiblio.org/obp/thinkCS/ [http://www.ibiblio.org/obp/thinkCS/]**

Open Source Information on C programming

5. **C von A bis Z, Juergen Wolf. Open book in German. Find it via C/Linux/Win32 Tutorial (deutsch) [http://www.pronix.de/C/index.shtml]**

Books

I always try to have all recommended books available in our library. Also take a look at my special section there where I collect books which should be present at all times.

1. **Kernighan/Ritchie, Programming in C. The bible of c-programming from the inventors. A classic text. Very short- compare this to nowadays documentation bloat.**
2. **The C pocket reference. A short book covering the latest developments in C. In my library.**
3. **The C Puzzle Book. Alan R. Feuer. A very short and nice book full of examples with C.**