

Introduction to C programming

3 lectures

9, 15, 22 september

(wednesday at 11:00-12:45 Q105)

dr. Jason Maassen

email: jason@cs.vu.nl

www.cs.vu.nl/~jason

Purpose of these lectures

- Crash course in C
- Related to Computer Networks Practical work.
 - ★ `www.cs.vu.nl/~cn`
 - ★ lectures on 8, 15, 22 september
(13:30-15:15 Q112)

Purpose of these lectures

- Crash course in C
 - ★ Emphasis on the differences between Java and C.
 - * most of you know Java
 - ★ Some practical examples
 - ★ Stuff related to C-programming
 - * compiler, preprocessor, debugging, makefiles, etc.

Literature

- Reader (C for Java programmers)
- Books
 - ★ The C Programming Language (Kernighan, Ritchie)
 - ★ The Complete Reference (Schildt)
 - ★ The C Puzzle Book (Feuer)
 - ★ Many others available....
- Ask amazon or google

Overview

- Very short history of C
- Overview of the differences between C and Java
- The C language (keywords, types, functions, etc.)
- Preprocessor
- Next lecture: pointers and memory management
- Last lecture: compiling, linking, makefiles, debugging

History of C

- Created in the mid-70's as a follow up to the 'B' and 'BCPL' languages
- Standardized between 1983 and 1988 (ANSI C)
- Syntax is the basis for other languages like C++, Objective-C, Java en C#

History of C

- Development of C is related to development of Unix
- Unix initially programmed using assembly language
 - ★ assembly instructions are very low-level
 - ★ assembly instructions are machine specific
- To make Unix more portable a high-level and portable language was needed
- Result was C

History of C

- Compared to assembly, C is
 - ★ high level (C has functions, while and for loops, etc.)
 - ★ portable (only C compiler must be ported for each platform, all applications stay the same)

Major differences between C and Java (1)

- C is **procedural** not object oriented
 - ★ C has no objects, interfaces or packages
 - ★ a programs only consists of functions and data
- C is **compiled** not intepreted
 - ★ translated directly to assembly language
 - ★ faster, less portable and **very hard** to debug.

Major differences between C and Java (2)

- C has **array bounds, null pointer or cast checks**
 - ★ you have to detect and handle all problems yourself
- C has **no garbage collector**
 - ★ you have to do all of the memory management yourself

Major differences between C and Java (3)

- C has **pointers**
 - ★ similar to references but ...
 - ★ ... they can be use in calculations (pointer arithmetic)
 - ★ allows you to use the **location** of data in computations (not just the value)
 - ★ useful, powerful and a **debugging nightmare!**

Major differences between C and Java (4)

- Compared to Java, C is a **low-level** language
 - ★ you can and must do everything yourself
 - ★ suitable for low-level software like device-drivers, communication libraries, operating systems, etc.

Keywords

- There are 32 keywords in C:

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

Keywords

- There are 32 keywords in C:

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

Keywords

- There are 32 keywords in C:

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

Keywords

- There are 32 keywords in C:

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

Program Structure (1)

- A Java program consists of
 - ★ several classes, each in different file.
 - ★ a main method in of these classes.
 - ★ external class libraries (jar files)

Program Structure (2)

- A C program consists of
 - ★ several functions, in one or more files.
 - ★ a main function in of these files.
 - ★ possibly some header files.

 - ★ external libraries and their header files

Example program

```
#include <stdio.h>

double global;

/* This is a comment */

int main(int argc, char** argv)
{
    int local = 0;
    global = 0.42;

    for (local=0;local<argc;local++)
    {
        printf("argument[%d] = %s\n", local, argv[local]);
    }

    printf("global = %f\n", global);
    return 0;
}
```

Example program

```
→ #include <stdio.h>

double global;

/* This is a comment */

int main(int argc, char** argv)
{
    int local = 0;
    global = 0.42;

    for (local=0;local<argc;local++)
    {
        printf("argument[%d] = %s\n", local, argv[local]);
    }

    printf("global = %f\n", global);
    return 0;
}
```

Example program

```
#include <stdio.h>

-> double global;

/* This is a comment */

int main(int argc, char** argv)
{
    int local = 0;
    global = 0.42;

    for (local=0;local<argc;local++)
    {
        printf("argument[%d] = %s\n", local, argv[local]);
    }

    printf("global = %f\n", global);
    return 0;
}
```

Example program

```
#include <stdio.h>

double global;

/* This is a comment */

-> int main(int argc, char** argv)
{
    int local = 0;
    global = 0.42;

    for (local=0;local<argc;local++)
    {
        printf("argument[%d] = %s\n", local, argv[local]);
    }

    printf("global = %f\n", global);
    return 0;
}
```

Main function (1)

- *main* is special (like in Java)
- Always returns an int
- Syntax:

```
int main(int argc, char** argv) { /* body */ }  
int main(void) { /* body */ }
```

Main function (2)

- The first form gets the command line arguments

| | |
|--------------------------|---|
| <code>int argc</code> | number of parameters |
| <code>char **argv</code> | an array of 'strings' (or: 'char arrays') |

- This is the same as

```
static void main(String [] args)
```

Example program

```
#include <stdio.h>

double global;

/* This is a comment */

int main(int argc, char** argv)
{
    int local = 0;
    global = 0.42;

    for (local=0;local<argc;local++)
    {
->        printf("argument[%d] = %s\n", local, argv[local]);
    }

->    printf("global = %f\n", global);
    return 0;
}
```

Compiling

- Compile this program like this:

```
gcc -Wall exampleMain.c
```

- The result is an executable file **a.out**:

```
./a.out hello world
```

```
argument[0] = ./a.out
```

```
argument[1] = hello
```

```
argument[2] = world
```

```
global = 0.420000
```

Example program

```
#include <stdio.h>

-> double global;

    /* This is a comment */

-> int main(int argc, char** argv)
{
->     int local = 0;
    global = 0.42;

    for (local=0;local<argc;local++)
    {
        printf("argument[%d] = %s\n", local, argv[local]);
    }

    printf("global = %f\n", global);
    return 0;
}
```

Basic Types (1)

- In C the size of the basic types is not defined
- Instead, the size depends on
 - ★ which target platform is compiled for
 - ★ which C compiler is used

Basic Types (2)

- An **int**, for example:
 - ★ is always **32 bits** in Java
 - ★ is the **natural size** for a processor in C
 - 16 bits** on a palm, **32 bits** on a PC, **64 bits** on an Alpha or Sparc Workstation
- The minimum size is defined

Basic Types (3)

| type | C (official) | C (pc+gcc) | Java |
|-------------|--------------------------|------------|------|
| char | 8 | 8 | 16 |
| short | ≥ 16 and \leq int | 16 | 16 |
| int | ≥ 16 | 32 | 32 |
| long | ≥ 32 | 32 | 64 |
| float | undefined | 32 | 32 |
| double | undefined | 64 | 64 |
| boolean | (use int) | (use int) | ? |
| byte | (use char) | (use char) | 8 |
| long long | unofficial type | 64 | - |
| long double | undefined | 96 | - |

Basic Types (4)

- **sizeof** determines the size of a type (in bytes)

```
#include <stdio.h>
int main(void)
{
    printf("char          = %d\n", sizeof(char)*8);
    printf("short         = %d\n", sizeof(short)*8);
    printf("int            = %d\n", sizeof(int)*8);
    printf("long             = %d\n", sizeof(long)*8);
    printf("long long        = %d\n", sizeof(long long)*8);
    printf("float            = %d\n", sizeof(float)*8);
    printf("double           = %d\n", sizeof(double)*8);
    printf("long double     = %d\n", sizeof(long double)*8);
    return 0;
}
```

Basic Types (5)

- C also supports **unsigned** types:

```
int i1; /* range -2,147,483,648 to 2,147,483,647 */
signed int i2; /* range -2,147,483,648 to 2,147,483,647 */
unsigned int i3; /* range 0 to 4,294,967,295 */
```

- Not for floats or doubles.

Booleans (1)

- C has no 'boolean' type, use 'int' instead
0 equals 'false', everything else is 'true':

```
if (0)          { /* not executed */ }  
if (42)         { /* executed   */ }  
  
int x = 100;  
if (x == 4)    { /* not executed */ }  
if (x)         { /* executed   */ }  
  
while (x--)    { /* repeated until x equals 0 */ }
```

Booleans (2)

- The 'boolean' operators now work in 'int' values:

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

Mistakes

- Missing an '=':

```
int y = 0;
if (y == 1) { /* not executed */ }
if (y = 1) { /* executed, result is 1 */ }
```

- Wrong operator:

```
int x = 0;
int y = 1;
if (x && y) { /* not executed */ }
if (x & y) { /* executed, result is 1 */ }
```

Mistakes

- No initialization:

```
int y;  
if (y)      { /* who knows what will happen ??? */ }  
while (y--) { /* same here */ }
```

Variables (1)

- C has two kinds of variables:
 - ★ local (declared inside of a function)
 - ★ global (declared outside of a function)

```
int global;  
void function(void)  
{  
    int local;  
}
```

Global Variables (1)

- Global variables have **file scope**
 - ★ they can (only) be used by all functions in a file
- Large C program's are typically split into multiple files
 - ★ each contains it's own global variables and functions
 - ★ their global variables can not be used in other files
 - ★ ...but their names may still clash!

Global Variables (2)

```
/* file1.c */  
  
int global = 2;  
int private = 5;
```

```
/* file2.c */  
#include <stdio.h>  
  
int private = 7;  
  
int main(void)  
{  
    printf("%d\n", global * private);  
}
```

Global Variables (2)

```
/* file1.c */
-> int global = 2;
   int private = 5;

->

/* file2.c */
#include <stdio.h>

int private = 7;

int main(void)
{
    printf("%d\n", global * private);
}
```

- Compile time error!

Global Variables (2)

```
/* file1.c */  
  
int global = 2;  
→ int private = 5;
```

```
/* file2.c */  
#include <stdio.h>  
  
int private = 7;  
  
int main(void)  
{  
    printf("%d\n", global * private);  
}
```

- Link time error!

Global Variables (3)

- To solve these problems, the following modifiers can be used:

| | |
|--------|--|
| extern | it is declared in another file, but can be used in this file |
| static | it can only be seen and used in this file |

Global Variables (4)

```
/* file1.c */
```

```
int global = 2;  
int private = 5;
```

```
/* file2.c */
```

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

```
int private = 7;
```

```
int main(void)
```

```
{
```

```
    printf("%d\n",  
           global * private);
```

```
}
```

Global Variables (4)

```
/* file1.c */  
  
-> int global = 2;  
   int private = 5;
```

```
/* file2.c */  
#include <stdio.h>  
  
extern int global;  
int private = 7;  
  
int main(void)  
{  
    printf("%d\n",  
           global * private);  
}
```

Global Variables (4)

```
/* file1.c */  
  
int global = 2;  
-> static int private = 5;
```

```
/* file2.c */  
#include <stdio.h>  
  
extern int global;  
static int private = 7;  
  
int main(void)  
{  
    printf("%d\n",  
           global * private);  
}
```

Local Variables (1)

- Can only be used inside a function (like in Java)
- Declaration **must** be done in beginning of function

```
/* ok */  
void funct1(void)  
{  
    int a, b;  
    a = 0;  
    b = 1;  
}
```

```
/* ok */  
void funct2(void)  
{  
    int a = 0;  
    int b = 1;  
}
```

```
/* wrong */  
void funct3(void)  
{  
    int a = 0;  
    a++;  
    int b = 1;  
}
```

Local Variables (2)

- Also in for loops !

```
/* ok */
void function1(void)
{
    int a, b;
    a = 10;
->   for (b=0;b<a;b++) {
        }
}
```

```
/* wrong */
void function2(void)
{
    int a = 0;
    a = 10;
    for (int b=0;b<a;b++) {
        }
}
```

Local variables (3)

- Modifiers:

| | |
|----------|--|
| auto | does nothing |
| register | try to put the variable in a register |
| volatile | never put the variable in a register |
| static | variable exist after function (sort of global) |

Arrays in Java

- Java has dynamic arrays
 - ★ arrays are *references* (created with *new*)
 - ★ size is determined at **run-time**
 - ★ array can be replaced (because it is a reference)

```
int [] a1, a2;  
a1 = new int[8];
```

```
int [] a3 = { 1, 2, 3 };  
a2 = a3;  
a2[10] = 5; /* throws an exception */
```

Arrays in C (1)

- C has static arrays
 - ★ *not references* but chunk of memory with a name
 - ★ size determined at **compile-time**
 - * size must be clear from the declaration
 - * size cannot be changed
 - ★ arrays cannot be assigned
 - ★ local arrays are only valid inside the function
 - * cannot be returned as a result value

Arrays in C (2)

```
int a1[5];           /* OK */
int a2[] = { 1, 2, 3 }; /* OK */
int a3[4][5];       /* OK */

int [] a4;          /* PARSE ERROR */
int a5[];           /* ERROR */

int [] function(void) { /* PARSE ERROR */
    int array[5];
    return array;
}

void foo(int a[]) { /* OK */
    a[3] = 0;
}

void bar(void) {
    int a[5];
    foo(a);           /* a is passed 'by-reference' */
    a = a5;           /* ERROR */
}
```

Arrays in C (3)

```
#include <stdio.h>
int main(void) {
    int i;

    int a1[] = { 1, 2, 3, 4, 5 };
    int a2[] = { 1, 2, 3, 4, 5 };

    a2[10] = 42; /* BUG */

    for (i=0;i<5;i++) {
        printf("a1[%d] = %d\n", i, a1[i]);
    }
}
```

Strings

- In C has no real strings, only arrays of 'char'.
 - ★ so strings have the same limitations as arrays
 - ★ strings must end with a special character
nul ('`\0`')
 - ★ string functions can be used by including *string.h*.

Strings Examples

```
char name0[6];  
name0[0] = 'J';  
name0[1] = 'a';  
name0[2] = 's';  
name0[3] = 'o';  
name0[4] = 'n';  
name0[5] = '\\0';
```

```
char name1[] = { 'J', 'a', 's', 'o', 'n', '\\0' };
```

```
char name2[6] = "Jason";
```

```
char name3[] = "Jason";
```

```
char name4[100] = "Not 100 characters long!";
```

Strings Examples

```
char name0[6];  
name0[0] = 'J';  
name0[1] = 'a';  
name0[2] = 's';  
name0[3] = 'o';  
name0[4] = 'n';  
→ name0[5] = '\0';
```

```
char name1[] = { 'J', 'a', 's', 'o', 'n', '\0' };
```

```
char name2[6] = "Jason";
```

```
char name3[] = "Jason";
```

```
char name4[100] = "Not 100 characters long!";
```

Strings Examples

```
char name0[6];  
name0[0] = 'J';  
name0[1] = 'a';  
name0[2] = 's';  
name0[3] = 'o';  
name0[4] = 'n';  
name0[5] = '\\0';
```

→ **char** name1[] = { 'J', 'a', 's', 'o', 'n', '\\0' };

```
char name2[6] = "Jason";
```

```
char name3[] = "Jason";
```

```
char name4[100] = "Not 100 characters long!";
```

Strings Examples

```
char name0[6];  
name0[0] = 'J';  
name0[1] = 'a';  
name0[2] = 's';  
name0[3] = 'o';  
name0[4] = 'n';  
name0[5] = '\\0';
```

```
char name1[] = { 'J', 'a', 's', 'o', 'n', '\\0' };
```

```
→ char name2[6] = "Jason";
```

```
char name3[] = "Jason";
```

```
char name4[100] = "Not 100 characters long!";
```

Strings Examples

```
char name0[6];  
name0[0] = 'J';  
name0[1] = 'a';  
name0[2] = 's';  
name0[3] = 'o';  
name0[4] = 'n';  
name0[5] = '\\0';
```

```
char name1[] = { 'J', 'a', 's', 'o', 'n', '\\0' };
```

```
char name2[6] = "Jason";
```

```
→ char name3[] = "Jason";
```

```
char name4[100] = "Not 100 characters long!";
```

Strings Examples

```
char name0[6];  
name0[0] = 'J';  
name0[1] = 'a';  
name0[2] = 's';  
name0[3] = 'o';  
name0[4] = 'n';  
name0[5] = '\\0';
```

```
char name1[] = { 'J', 'a', 's', 'o', 'n', '\\0' };
```

```
char name2[6] = "Jason";
```

```
char name3[] = "Jason";
```

```
→ char name4[100] = "Not 100 characters long!";
```

Enumerations

- *enum* is used to create a number of related constants

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

```
enum workdays today;
```

```
today = tuesday;
```

```
today = friday;
```

```
enum weekend {saturday = 10, sunday = 20};
```

Structures

- A *struct* can be used to 'combine variables'
- Fields can be accessed using a *record selector* ('.')
- Structs are passed 'by-value'

```
struct ComplexNumber {  
    double real, imag;  
};  
  
struct ComplexNumber num;  
num.real = 2.5;  
num.imag = 0.3;
```

```
struct Parameters {  
    struct ComplexNumber complex;  
    double value;  
} param;  
  
param.complex.real = 3.6;  
struct Parameters wow[5];  
wow[3].complex.real = 3.6;
```

Unions

- Look similar to structs but behave differently
- All fields use a **single** memory location
- Unions are passed 'by-value'

```
union MyUnion {  
    int i_value;  
    double d_value;  
};
```

```
union MyUnion u;  
u.i_value = 6;  
u.d_value = 5.4;
```

```
union Value {  
    struct ComplexNumber complex;  
    double normal;  
};
```

```
union Value val;  
val.complex.real = 7.9;  
val.normal = 9.8;
```

Defining Types

- Using *typedef* new types can be defined
- Syntax: *typedef* type name

```
typedef char byte;  
byte b = 123;
```

```
typedef struct ComplexNumber complex;  
complex var;  
var.real = 5.9;  
var.imag = 0.1;
```

Functions (1)

- In C, functions can be defined in three ways:

```
int foo(void) { /* body */ }  
void bar(int p1, double p0) { /* body */ }  
double func(int p1, int p2, ...) { /* body */ }
```

- The last function has a variable number of parameters
 - ★ `printf` is an example of such a function

Printf

- *printf* is the standard print function in C.
- Many types exist (*fprintf*, *sprintf*, etc.)

- Syntax:

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

- 'const char *format' is the *format string*.
- has variable number of parameters!!

Printf Examples

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

```
int main(void) {  
    int val = 5;  
    char c = 'a';  
    char str[] = "world";  
  
    printf("Hello world\n");  
    printf("Hello %d World\n", val);  
    printf("%d %c World\n", val, c);  
    printf("Hello %s\n", str);  
    return 0;  
}
```

```
Hello world  
Hello 5 World  
5 a World  
Hello world
```

Printf (*format string*)

| | |
|----|--------------------------|
| %d | signed int |
| %u | unsigned int |
| %x | hexadecimal unsigned int |
| %c | character |
| %f | double and float |
| %s | string |
| %% | to print a % |

Man pages

- *man pages* give more information on C functions:

```
man -S 3 printf
```

```
PRINTF(3)
```

```
Linux Programmer's Manual
```

```
PRINTF(3)
```

```
NAME
```

```
printf, fprintf, sprintf, snprintf, vprintf, vfprintf,  
vsprintf, vsnprintf - formatted output conversion
```

```
etc.
```

Function Prototypes (1)

- A function must be defined **before** it can be used
- This can be done using a *function prototype*

```
void example(void) {  
    int result = max(5, 8);    /* ERROR */  
}  
  
int max(int one, int two) {  
    return (one < two ? two : one);  
}
```

Function Prototypes (1)

- A function must be defined **before** it can be used
- This can be done using a *function prototype*

```
int max(int, int); /* prototype */
void example(void) {
    int result = max(5, 8); /* OK */
}
int max(int one, int two) {
    return (one < two ? two : one);
}
```

Function Prototypes (2)

- Sometimes useful to solve cyclic dependencies
- Essential when applications are split into multiple files!
 - ★ must specify the prototype of functions in 'other' files.
- Problem: may get out of control.

Function Prototypes (2)

- Example:
 - ★ application split into 10 files
 - ★ each uses 4 functions from every other file.
 - ★ need 36 function prototypes per file!
 - ★ same for type definitions!
- Solution: use header files and preprocessor
 - ★ requires 9 `#include`'s

Header files

- File that only contains
 - ★ type definitions
 - ★ function prototypes
 - ★ other preprocessor directives
- Can be used 'export' types and function from a C-file
- Other files can then 'import' the header using the `#include` preprocessor directive.

Preprocessor (1)

- Preprocessor is first step of compilation.
- Compiling a file consists of three steps:
 1. *Preprocessor*
Processes all 'preprocessor directives'
 2. *Compiler*
Translates the file to machine language (*file.o*)
 3. *Linker*
Combines binary files and libraries into executable

Preprocessor (2)

- Started automatically by the compiler (cpp).
- Reads and reacts to *preprocessor directives*.
 - ★ commands starting with #
 - ★ comments (`/* */`)
- output of preprocessor is then compiled

Preprocessor (`#include`)

- `#include` includes a header file
- The header files will be **copied** into the program
- Be **very careful** with repeating `#includes`!
 - ★ repeated type definitions not allowed!
- Syntax:

```
#include <filename>    /* FOR 'GLOBAL' INCLUDES */  
#include "filename"   /* FOR 'LOCAL' INCLUDES */
```

Preprocessor (#include Example)

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

```
int main(void) {  
    printf("Hello world\n");  
    return 0;  
}
```

```
|  
-----> int printf(const char *__format  
          int sprintf(char *__s, const ch  
          /* ... + many other prototypes  
  
          void main(void) {  
              printf("Hello world\n");  
              return 0;  
          }
```

Header files (standard libs)

- There are a large number of standard header files:

| | |
|-----------------------|--|
| <code>stdio.h</code> | Input/output functions |
| <code>stdlib.h</code> | Some standard functions and macros |
| <code>math.h</code> | Mathematical functions |
| <code>stdarg.h</code> | Functions to use a variable number of parameters |
| <code>string.h</code> | Functions to manipulate strings |
| <code>time.h</code> | Functions related to time |

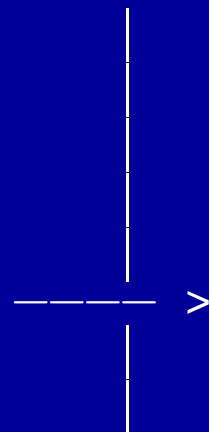
- have a look at `'/usr/include'` and `'/usr/lib'`

Preprocessor (#define)

- #define defines *constants*

```
#define LOOPS 100
```

```
void function(void) {  
    int i, j;  
    for (i=0;i<LOOPS;i++) {  
        ...  
    }  
}
```



```
void function(void) {  
    int i, j;  
    for (i=0;i<100;i++) {  
        ....  
    }  
}
```

Preprocessor (#ifdef en #if)

- #ifdef and #if can be used to conditionally compile

```
#define DEBUG
#define TEST 0

void function(void)
{
#ifdef DEBUG
    printf("first\n");
#endif
#if TEST
    printf("second\n");
#endif
}
```

----->

```
void function(void)
{
    printf("first\n");
}
```

Header files

- Header files should always look like this:

```
#ifndef SOME_UNIQUE_NAME
#define SOME_UNIQUE_NAME

/* type definitions go here */
/* function prototypes go here */

#endif /* SOME_UNIQUE_NAME */
```

Overview (1)

- C is a procedural language suited for low-level software
 - ★ compiled not interpreted
 - ★ hardly anything is checked
 - * your program will crash if you make a mistake

Overview (2)

- Basic C syntax is similar to Java, but
 - ★ size of basic types is not fixed
 - ★ C has global variables
 - ★ no booleans, bytes or strings
 - ★ arrays behave differently
 - ★ also has enums, structs etc.
- Preprocessor is not part of language, but frequently used.

More info

- Reader
- Books
 - ★ The C Programming Language, by Kernighan and Ritchie
- Man pages
- Ask amazon or google

Next week

Pointers and Memory Management!

`jason@cs.vu.nl`

`www.cs.vu.nl/~jason`

`www.cs.vu.nl/~cn`