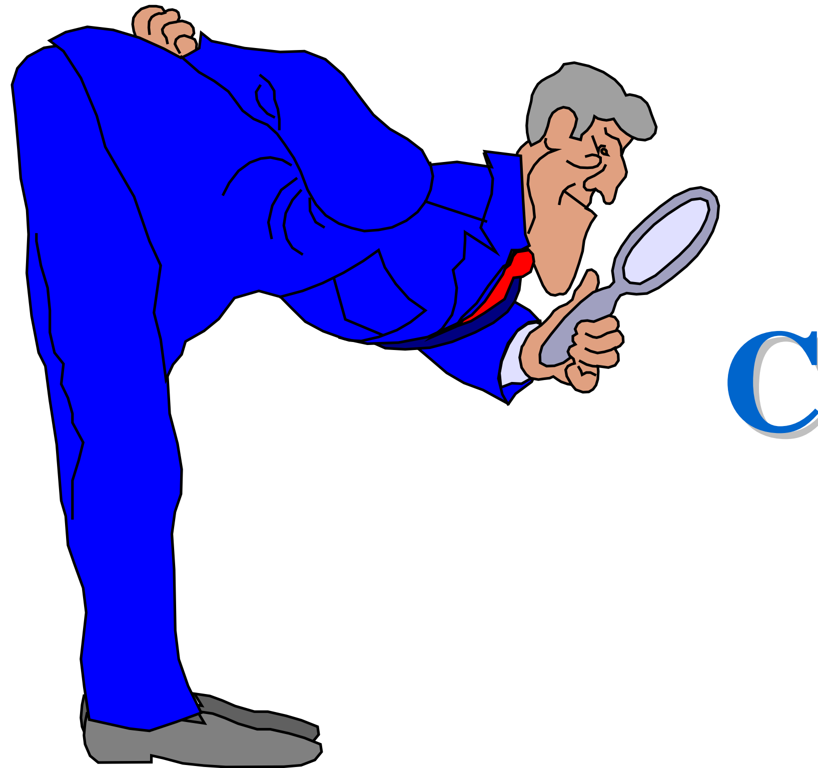
The C Language



Maurizio Martignano

Copyright © 2005-2006

C - Introduction

- In this section the C language will be presented.
- Not all the syntactical details of the language will be described, once again the focus will be on semantics.
- A step-by-step approach, from simple topics to more complex ones, will be used.
- Having some knowledge and/or experience on C helps in better understanding C++.
- The previous section on Principles is a pre-requisite for understanding this section on C.

Maurizio Martignano
Copyright © 2005-2006

C - Program Structure

First C program (hello0.c)

```
#include <stdio.h>

void main(void) {
  printf("Hello world!\n");
}
```

- Let’s have a look to the first C program which is always mentioned in literature as starting point: the “Hello world” program.
- Let’s also have a look to some other versions of the program.

...

Maurizio Martignano
Copyright © 2005-2006

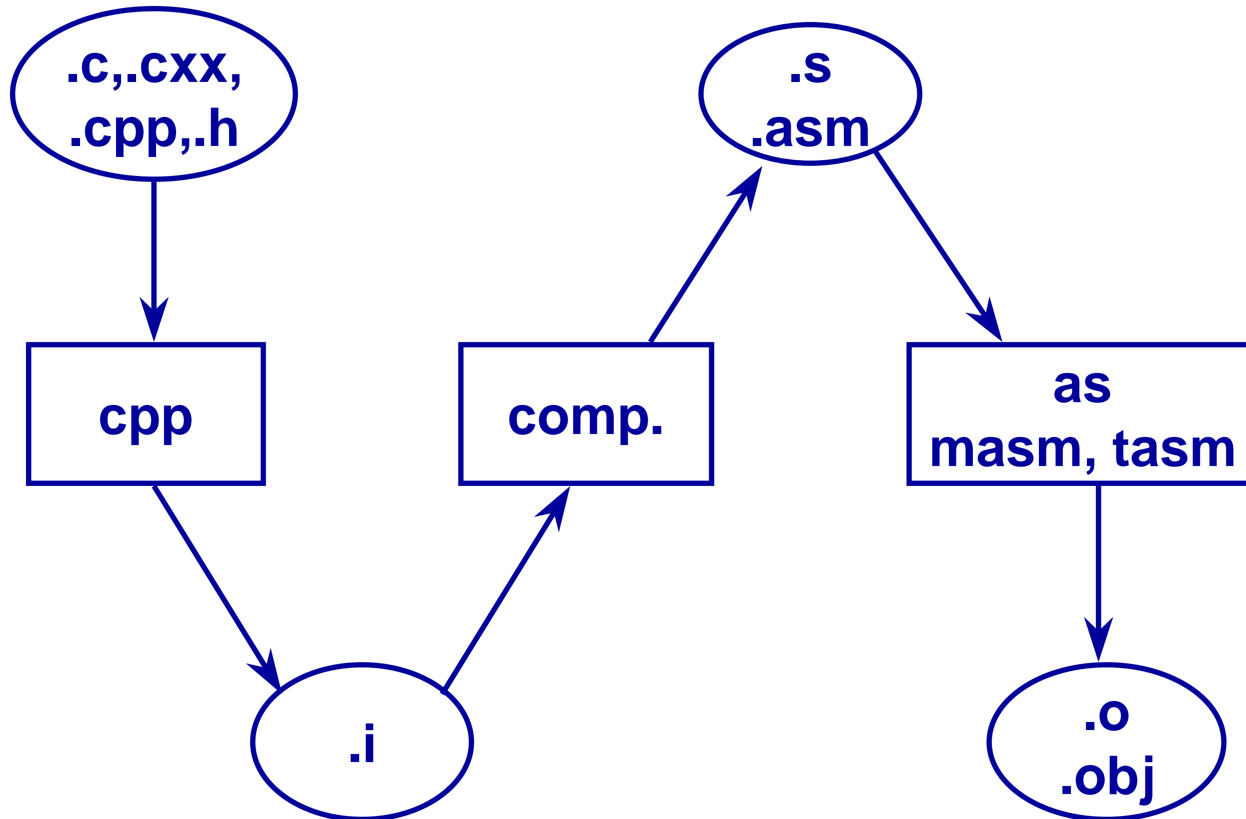
C - Program Structure (cont)

- Every C program has the following structure:
  - Include Directives
  - Types Definitions
  - Variables Declarations
  - Functions Prototypes
  - Functions Bodies
- Preprocessor Directives can appear in every portion of the program.
- One of the function must be the "main" function, I.e the starting point of the program.

  BTW: Who does call the "main" when the program starts? To whom the "main" returns its integer value?

Maurizio Martignano
Copyright © 2005-2006

C - Compiler Steps



Copyright © 2005-2006

C - Preprocessor Directives - #include

- Syntax

```
#include <header_name>
#include "header_name"
```

- Description
  - The #include directive pulls in other named files, known as include files, header files, or headers, into the source code.
  - The difference between the <header_name> and "header_name" formats lies in the searching algorithm employed in trying to locate the include file:
    - » when **“header_name”** is used the file is looked for in the **current directory**;
    - » when **<header_name>** is used the file is looked for in the directories belonging to the **include path**.

Maurizio Martignano
Copyright © 2005-2006

C - Preprocessor Directives - #define

- Syntax

```
#define macro_identifier <token_sequence>
```

- Description
  - The #define directive defines a macro. Macros provide a mechanism for token replacement with or without a set of formal, function-like parameters.
  - Each occurrence of macro_identifier in your source code following this control line will be replaced in the original position with the possibly empty token_sequence. Such replacements are known as macro expansions. The token sequence is sometimes called the body of the macro.
  - An empty token sequence results in the removal of each affected macro identifier from the source code.

Maurizio Martignano
Copyright © 2005-2006

C - Preprocessor Directives - #define (cont)

- Syntax

```
#define macro_identifier(<arg_list>)
  token_sequence
```

- Description
  - Note there can be no white space between the macro identifier and the (. The optional arg_list is a sequence of identifiers separated by commas. Each comma-delimited identifier plays the role of a formal argument or placeholder.
  - The syntax is identical to that of a function call; indeed, many standard library C "functions" are implemented as macros. However, there are some important semantic differences, side effects, and **potential pitfalls**.

Maurizio Martignano
Copyright © 2005-2006

C - Preprocessor Directives - #define (cont)

- The optional actual_arg_list must contain the same number of comma-delimited token sequences, known as actual arguments, as found in the formal arg_list of the #define line: There must be an actual argument for each formal argument. An error will be reported if the number of arguments in the two lists is different.
- (See the example program macro.c).

Maurizio Martignano
Copyright © 2005-2006

C - Preprocessor Directives - #if...

- Syntax

```
#if constant-expression-1
<section-1>
<#elif constant-expression-2 newline section-2>

<#elif constant-expression-n newline section-n>
<#else <newline> final-section>
#endif
```


Maurizio Martignano
Copyright © 2005-2006

C - Preprocessor Directives - #if… (cont)

- Description

  - The conditional directives #if, #elif, #else, and #endif work like the normal C conditional operators. If the constant-expression-1 (subject to macro expansion) evaluates to nonzero (true), the lines of code (possibly empty) represented by section-1, whether preprocessor command lines or normal source lines, are preprocessed and, as appropriate, passed to the compiler. Otherwise, if constant-expression-1 evaluates to zero (false), section-1 is ignored (no macro expansion and no compilation).

Maurizio Martignano
Copyright © 2005-2006

C - Preprocessor Directives - #if… (cont)

- In the true case, after section-1 has been preprocessed, control passes to the matching #endif (which ends this conditional sequence) and continues with next-section. In the false case, control passes to the next #elif line (if any) where constant-expression-2 is evaluated. If true, section-2 is processed, after which control moves on to the matching #endif. Otherwise, if constant-expression-2 is false, control passes to the next #elif, and so on, until either #else or #endif is reached. The optional #else is used as an alternative condition for which all previous tests have proved false. The #endif ends the conditional sequence.

Maurizio Martignano
Copyright © 2005-2006

C - Preprocessor Directives - #if… (cont)

- The processed section can contain further conditional clauses, nested to any depth; each #if must be matched with a closing #endif.
- The net result of the preceding scenario is that only one section (possibly empty) is passed on for further processing. The bypassed sections are relevant only for keeping track of any nested conditionals, so that each #if can be matched with its correct #endif.
- The constant expressions to be tested must evaluate to a constant integral value.

Maurizio Martignano
Copyright © 2005-2006

C - Preprocessor Directives - #ifdef...

- Syntax

```
#ifdef identifier
#ifndef identifier
```

- Description
  - The #ifdef and #ifndef conditional directives let you test whether an identifier is currently defined or not; that is, whether a previous #define command has been processed for that identifier and is still in force.
  - The expression "#ifdef identifier" has exactly the same effect as "#if 1" if identifier is currently defined, and the same effect as "#if 0" if identifier is currently undefined.

Maurizio Martignano
Copyright © 2005-2006

C - Preprocessor Directives - #ifdef… (cont)

- #ifndef tests true for the "not-defined" condition, so the expression "#ifndef identifier" has exactly the same effect as "#if 0" if identifier is currently defined, and the same effect as "#if 1" if identifier is currently undefined.
- The syntax thereafter follows that of the #if, #elif, #else, and #endif.
- An identifier defined as NULL is considered to be defined.

Maurizio Martignano
Copyright © 2005-2006

C - Preprocessor Directives - #undef

- Syntax

```
#undef identifier
```

- Description
  - The #undef directive deletes the effects of the previous corresponding #define (if any).

Maurizio Martignano
Copyright © 2005-2006

C - Variables Declaration

- Syntax

```
[<memory type>] [<type modifier>] <typename>
  identifier;
```

- Examples

int x;

static int y;

register int z;

unsigned char a;

int * pi;

Maurizio Martignano
Copyright © 2005-2006

C - Variables Declaration (cont)

- The <memory type> can be one of the following keywords:
  - **auto**: the variable is stored in the activation record of a function;
  - **register**: either the variable is stored in a register or the compiler tries to optimize its usage; this clause applies only to automatic variable;
  - **static**: the variable is stored in the static area, I.e it “stays alive” for all the program execution;
  - **volatile**: the variable belongs to a memory location/area not completely under the control of the program (see example vola.c).

Maurizio Martignano
Copyright © 2005-2006

C - Variables Declaration (cont)

- All variables declared **inside a functions** are implicitly assumed to be **auto**matic.
- All variables declared **outside the functions** are implicitly assumed to be **static** (see the example benv.c).
- The <type modifier> can be :
  - **signed**: the variable is a signed char or integer type;
  - **unsigned**: the variable is an unsigned char or integer type;
  - **short**: it applies to integer types and means:
    - » `sizeof(short int) <= sizeof(int)`
  - **long**: it applies to integer and double types, it means:
    - » `sizeof(int) <= sizeof(long int)`
    - » `sizeof(double) <= sizeof(long double)`

  (see example sizes.c).

Maurizio Martignano
Copyright © 2005-2006

C - Variables Declaration (cont)

- The <type name> can be either:
  - one of the **basic types** or
  - one **complex type** (I.e. a type built upon basic types or other derived types).
- The C basic types are:
  - **char**: character type;
  - **int**: integer type;
  - **float**: single precision floating point type;
  - **double**: double precision floating point type;
  - **void**: null type for pure procedures, empty arguments list and pointers to objects of unknown type.

Maurizio Martignano
Copyright © 2005-2006

C - Variables Declaration - Arrays (cont)

- Syntax

```
<type name> declarator [<constant-expression>]
```

- Description
  - The above statement declares an array composed of elements of <type names >. An array consists of a contiguous region of storage exactly large enough to hold all of its elements.
  - If an expression is given in an array declarator, it must evaluate to a positive constant integer. The value is the number of elements in the array. Each of the elements of an array is numbered from 0 through the number of elements minus one.

Maurizio Martignano
Copyright © 2005-2006

C - Variables Declaration - Arrays (cont)

- Multidimensional arrays are constructed by declaring arrays of array type. The following example shows the declaration of a single and multidimensional array.

- Examples

```
char my_string[100];
int my_matrix[10][10];
```


Maurizio Martignano
Copyright © 2005-2006

C - Variables Declaration - Structures (cont)

- Syntax

```
struct [<struct type name>] {
   [<type> <variable-name[, variable-name,
   ...]>] ;
   ...
} [<structure variables>] ;
```

- Description
  - Use a struct to group variables into a single record.
  - <struct type name> : an optional tag name that refers to the structure type.
  - <structure variables> : the data definitions, also optional.

Maurizio Martignano
Copyright © 2005-2006

C - Variables Declaration - Structures (cont)

- Though both <struct type name> and <structure variables> are optional, one of the two must appear.
- You define elements in the record by naming a <type>, followed by one or more <variable-name> (separated by commas).
- Separate different variable types by a semicolon.
- To access elements in a structure, use a record selector (.).
- To declare additional variables of the same type, use the keyword struct followed by the <struct type name>, followed by the variable names.

Maurizio Martignano
Copyright © 2005-2006

C - Variables Declaration - Structures (cont)

- Examples

```
struct _s_complex {
  float real;
  float ima;
} x, y, z;

struct _s_complex {
  float real;
  float ima;
};
struct _s_complex x, y,
  z;
```

```
struct _s_node {
  char info[10+1];
  _s_node * left;
  _s_node * right;
};

struct _s_node bt_node;
```


Maurizio Martignano
Copyright © 2005-2006

C - Variables Declaration - Structures (cont)

- In structures it is possible to use the following expression

  type-specifier <bitfield-id> : width;

  where type-specifier is char, unsigned char, int, or unsigned int. If the bit field identifier is omitted, the number of bits specified in width is allocated, but the field is not accessible. This lets you match bit patterns in, say, hardware registers where some bits are unused.
- Example:

```
struct mystruct {
  int          i : 2;
  unsigned     j : 5;
  int            : 4;
  int          k : 1;
  unsigned     m : 4;
} a, b, c;
```


Maurizio Martignano
Copyright © 2005-2006

C - Variables Declarations - Unions (cont)

- Syntax

```
union [<union type name>] {
  <type> <variable names> ;
  ...
} [<union variables>] ;
```

- Description
  - Use unions to define variables that share storage space.
  - The compiler allocates enough storage to accommodate the largest element in the union.
  - Writing into one element overwrites the other(s).

Maurizio Martignano
Copyright © 2005-2006

C - Variables Declarations - Unions (cont)

- Use the record selector (.) to access elements of a union .

- Example

```
union _u_numbers {
  short int short_integers[100];
  unsigned char bytes[200]
};

union _u_numbers my_numbers;
```


Maurizio Martignano
Copyright © 2005-2006

C - Variables Declarations - Enums (cont)

- Syntax

```
enum [<type_tag>] {<constant_name> [=
  <value>], ...} [var_list];
```

  - <type_tag> is an optional type tag that names the set.
  - <constant_name> is the name of a constant that can optionally be assigned the value of <value>. These are also called enumeration constants.
  - <value> must be an integer. If <value> is missing, it is assumed to be: <prev> + 1 where <prev> is the value of the previous integer constant in the list. For the first integer constant in the list, the default value is 0.
  - <var_list> is an optional variable list that assigns variables to the enum type.

Maurizio Martignano
Copyright © 2005-2006

C - Variables Declarations - Enums (cont)

- Description
  - Use the enum keyword to define a set of constants of type int, called an enumeration data type.
  - An enumeration data type provides mnemonic identifiers for a set of integer values.
  - Enums are always interpreted as ints if the range of values permits, but if they are not ints the value gets promoted to an int in expressions. Depending on the values of the enumerators, identifiers in an enumerator list are implicitly of type signed char, unsigned char, or int.

Maurizio Martignano
Copyright © 2005-2006

C - Variables Declarations - Enums (cont)

- In the absence of a <value> the first enumerator is assigned the value of zero. Any subsequent names without initialisers will then increase by one. <value> can be any expression yielding a positive or negative integer value (after possible integer promotions). These values are usually unique, but duplicates are legal.
- Enumeration tags share the same name space as structure and union tags. Enumerators share the same name space as ordinary variable identifiers.

- Examples:
  - See the program enum.c.

Maurizio Martignano
Copyright © 2005-2006

C - Variables Declarations - Constants (cont)

- Syntax

```
const <variable name> [ = <value> ] ;
```

- Description
  - Use the const modifier to make a variable value unmodifiable.
- Example
  - See the program const.c

Maurizio Martignano
Copyright © 2005-2006

C - Types Definition

- Syntax

```
typedef <type definition> <identifier> ;
```

- Description
  - Use the typedef keyword to assign the symbol name <identifier> to the data type definition <type definition>.
- Example
  - See the program bytes.c

Maurizio Martignano
Copyright © 2005-2006

C - Typedef vrs. Variable Declarations

- A type definition is just a renaming of the type.
- No memory allocation takes place during a type definition.
- Memory allocation occurs only at variables declaration time (see example bytes.c).

Maurizio Martignano
Copyright © 2005-2006

C - Expressions - Statement

- Any expression followed by a semicolon forms an expression statement:

  <expression>;
- The compiler (C RTS) executes an expression statement by evaluating the expression. All side effects from this evaluation are completed before the next statement is executed. Most expression statements are assignment statements or function calls
- The null statement is a special case, consisting of a single semicolon (;). The null statement does nothing, and is therefore useful in situations where the C RTS syntax expects a statement but your program does not need one.

Maurizio Martignano
Copyright © 2005-2006

C - Expressions - Assignment (cont)

- Syntax

  unary-expr assignment-op assignment-expr

- Remarks
  - The assignment operators are:

    ```
    =      *=     /=     %=     +=     -=
    <<=    >>=    &=     ^=     |=
    ```

  - The = operator is the only simple assignment operator, the others are compound assignment operators.
  - In the expression E1 = E2, E1 must be a modifiable lvalue. The assignment expression itself is not an lvalue.

Maurizio Martignano
Copyright © 2005-2006

C - Expressions - Assignment (cont)

- The expression

  E1 op= E2

- has the same effect as

  E1 = E1 op E2

- except the lvalue E1 is evaluated only once. For example, E1 += E2 is the same as E1 = E1 + E2.
- Note: Spaces separating compound operators (+<space>=) will generate errors.

Maurizio Martignano
Copyright © 2005-2006

C - Expressions - Arithmetic Operators (cont)

- Syntax

```
+ cast-expression
- cast-expression
add-expression + multiplicative-expression
add-expression - multiplicative-expression
multiplicative-expr * cast-expr
multiplicative-expr / cast-expr
multiplicative-expr % cast-expr
postfix-expression ++        (postincrement)
++ unary-expression          (preincrement)
postfix-expression --        (postdecrement)
-- unary-expression          (predecrement)
```


Maurizio Martignano
Copyright © 2005-2006

C - Expressions - Arithmetic Operators (cont)

- Description
  - Use the arithmetic operators to perform mathematical computations.
  - The unary expressions of + and **-** assign a positive or negative value to the cast-expression.
  - \+ (addition), **-** (subtraction), * (multiplication), and **/** (division) perform their basic algebraic arithmetic on all data types, integer and floating point.
  - **%** (modulus operator) returns the remainder of integer division and cannot be used with floating points.

Maurizio Martignano
Copyright © 2005-2006

C - Expressions - Arithmetic Operators (cont)

- ++ (increment) adds one to the value of the expression. Postincrement adds one to the value of the expression after it evaluates; while preincrement adds one before it evaluates.
- **--** (decrement) subtracts one from the value of the expression. Postdecrement subtracts one from the value of the expression after it evaluates; while predecrement subtracts one before it evaluates.

Maurizio Martignano
Copyright © 2005-2006

C - Expressions - Logical Operators (cont)

- Syntax
  - logical-AND-expr && inclusive-OR-expression
  - logical-OR-expr || logical-AND-expression
  - ! cast-expression
- Remarks
  - Operands in a logical expression must be of scalar type.
  - **&&** logical AND; returns true only if both expressions evaluate to be nonzero, otherwise returns false. If the first expression evaluates to false, the second expression is not evaluated.
  - || logical OR; returns true if either of the expressions evaluate to be nonzero, otherwise returns false. If the first expression evaluates to true, the second expression is not evaluated.

Maurizio Martignano
Copyright © 2005-2006

C - Expressions - Logical Operators (cont)

– **!** logical negation; returns true if the entire expression evaluates to be nonzero, otherwise returns false. The expression !E is equivalent to (0 = = E).

Maurizio Martignano
Copyright © 2005-2006

C - Expressions - Bitwise Operators (cont)

- Syntax
  - AND-expression **&** equality-expression
  - exclusive-OR-expr **^** AND-expression
  - inclusive-OR-expr **|** exclusive-OR-expression
  - ~cast-expression
  - shift-expression **<<** additive-expression
  - shift-expression **>>** additive-expression
- Remarks
  - Use the bitwise operators to modify the individual bits rather than the number.

Maurizio Martignano
Copyright © 2005-2006

C - Expressions - Bitwise Operators (cont)

- **&** bitwise AND; compares two bits and generates a 1 result if both bits are 1, otherwise it returns 0.
- **|** bitwise inclusive OR; compares two bits and generates a 1 result if either or both bits are 1, otherwise it returns 0.
- **^** bitwise exclusive OR; compares two bits and generates a 1 result if the bits are complementary, otherwise it returns 0.
- **~** bitwise complement; inverts each bit. ~ is used to create destructors.
- **>>** bitwise shift right; moves the bits to the right, discards the far right bit and assigns the left most bit to 0.
- **<<** bitwise shift left; moves the bits to the left, it discards the far left bit and assigns the right most bit to 0.

Copyright © 2005-2006

C - Control Structures - if

- Syntax

```
if ( <condition> ) {
  <statements1>
}
if ( <condition> )  {
  <statements1>
} else {
  <statements2>
}
```

- Description
  - Use if to implement a conditional statement.

Maurizio Martignano
Copyright © 2005-2006

C - Control Structures - if (cont)

- The condition statement must convert to an integer expression in C and a bool type in C++.
- When <condition> evaluates to a non zero value, <statements1> execute.
- If <condition> is zero, <statements2> execute.
- The else keyword is optional, but no statements can come between an if statement and an else.

Maurizio Martignano
Copyright © 2005-2006

C - Control Structures - switch (cont)

- Syntax

```
switch ( <switch variable> ) {
  case <constant expression> : <statements>;
                               [break;]
  ...
  default :                    <statements>;
}
```

- Description
  - Use the switch statement to pass control to a case which matches the <switch variable>. At which point the statements following the matching case evaluate.
  - If no case satisfies the condition the default case evaluates.
  - To avoid evaluating any other cases and relinquish control from the switch, terminate each case with break.

Maurizio Martignano
Copyright © 2005-2006

C - Control Structures - for (cont)

- Syntax

```
for ( [<initialization>] ; [<condition>] ; [<increment>]
   ) {
  <statements>
}
```

- Description
  - The for statement implements an iterative loop.
  - <statements> are executed repeatedly UNTIL the value of <condition> is false.
  - Before the first iteration of the loop, <initialization> initializes variables for the loop.
  - After each iteration of the loop, <increments> increments a loop counter. Consequently, j++ is functionally the same as ++j.
  - All the expressions are optional. If <condition> is left out, it is assumed to be always true.

Maurizio Martignano
Copyright © 2005-2006

C - Control Structures - while (cont)

- Syntax

```
while ( <condition> ) {
  <statements>
}
```

- Description
  - Use the while keyword to conditionally iterate a statement.
  - <statements> execute repeatedly until the value of <condition> is false. If no condition is specified, the while clause is equivalent to while(true).
  - The test takes place before <statements> execute. Thus, if <condition> evaluates to false on the first pass, the loop does not execute.

Maurizio Martignano
Copyright © 2005-2006

C - Control Structures - do (cont)

- Syntax

```
do {
   <statements>
} while ( <condition> );
```

- Description
  - The do statement executes until the condition becomes false.
  - <statements> are executed repeatedly as long as the value of <condition> remains true.
  - Since the condition tests after each the loop executes the <statement>, the loop will execute at least once.

Maurizio Martignano
Copyright © 2005-2006

C - Control Structures - goto (cont)

- Syntax

```
goto <identifier> ;
```

- Description
  - Use the goto statement to transfer control to the location of a local label specified by <identifier>.
  - Labels are always terminated by a colon.

Maurizio Martignano
Copyright © 2005-2006

C - Functions

- **C does not** clearly **distinguish** between **functions** and **procedures**:
  - **procedures** are a particular type of functions whose **return type** is **void**;
  - **functions** can be used in a **call statement** or in an **expression**.
- Example

```
rec_fact(6);
x = rec_fact(6);
```


Maurizio Martignano
Copyright © 2005-2006

C - Functions (cont)

- Syntax

```
<type name> identifier (<parameter declarator list>);
```

[prototype]

```
<type name> identifier (<parameter declarator list>) {
  <body>
```

[actual function]

```
}
```

- Remarks
  - The prototype of a function must appear in the code before any use of that function. If this is not the case, the compiler will issue some warning messages and will make default (incorrect) assumptions on it.
  - **In C a function is uniquely identified by its name**, I.e. two different functions cannot have the same name.

Maurizio Martignano
Copyright © 2005-2006

C - Functions (cont)

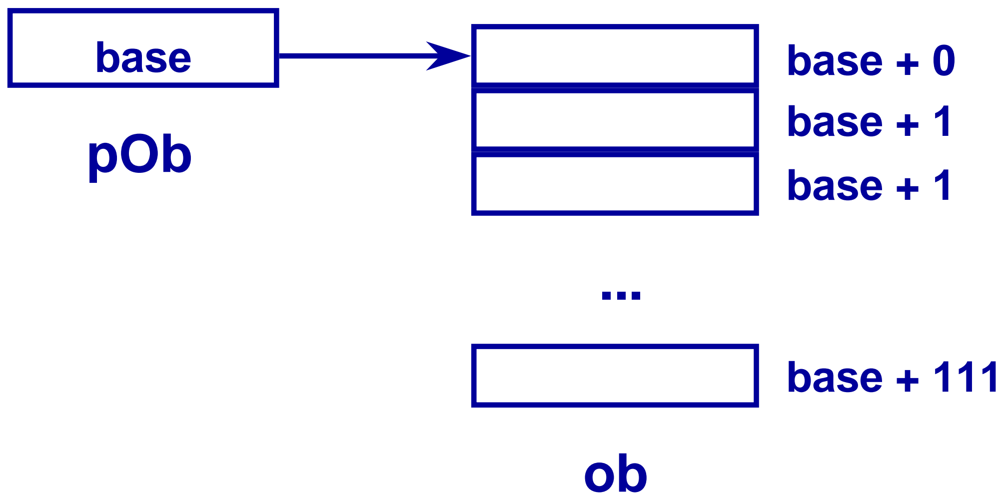
- If you use the const modifier with a pointer parameter in a function's parameter list, the function cannot modify the variable that the pointer points to. For example,

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

  printf is prevented from modifying the format string (see the cofunc.c example).
- The notation **...** at the end of the parameter list identifies a function with a variable parameter list (check the program sum.c to see how to define and use a function with variable parameter list).

Maurizio Martignano
Copyright © 2005-2006

C - Pointers (at last!)



- A pointer is a reference to a memory area containing a value, it is its address (I.e. l-value).
- Example

```
typedef _s_big {
   char info[100];
   long x, y, z
} BigObject;

BigObject ob;
BigObject * pOb;
```


Maurizio Martignano
Copyright © 2005-2006

C - Pointers (cont)

- Remarks
  - A pointer must be declared as pointing to some particular type, even if that type is void (which really means a pointer to anything).
  - If <type name> is any predefined or user-defined type, including void, the declaration

    ```
    <type name> *ptr;   /* Uninitialized pointer */
    ```

    declares ptr to be of type "pointer to <type name>." All the scoping, duration, and visibility rules apply to the ptr object just declared.

Maurizio Martignano
Copyright © 2005-2006

C - Pointers (cont)

- A null pointer value is an address that is guaranteed to be different from any valid pointer in use in a program. Assigning the integer constant 0 to a pointer assigns a null pointer value to it.
- The mnemonic NULL (defined in the standard library header files, such as stdio.h) can be used for legibility. All pointers can be successfully tested for equality or inequality to NULL.
- The pointer type "pointer to void" must not be confused with the null pointer. The declaration

  ```
  void *vptr;
  ```

  declares that vptr is a generic pointer capable of being assigned to by any "pointer to type" value, including null, without complaint.

Maurizio Martignano
Copyright © 2005-2006

C - Pointers (cont)

- NOTE: **pointers must refer to valid (I.e. properly allocated) memory areas.**
- **Pointers** are the **only method** to **access** and **use** memory areas in the **heap** (see example malloc.c).
- Though the C parameter passing mechanism is **call-by-value**, by passing to a function the pointer to an object it is possible for that function to modify the object.

Maurizio Martignano
Copyright © 2005-2006

C - Pointers & Arrays (cont)

- Given the following piece of code

```
int vect[100];
int *pv;

pv = &vect[0];  /* I.e pv = vect */
```

- The following holds true (see example minimal.c):

```
pv              ~       vect
*pv             ~       vect[0]
*(pv + 1)       ~       vect[1]
*(pv + i)       ~       vect[i]
&(*(pv + i))    ~       &vect[i]
pv + i          ~       &vect[i]
```


Maurizio Martignano
Copyright © 2005-2006

C - Pointers & Structures (cont)

- Given a structure and a pointer to it

```
struct struct_identifier {
  <type name> field_identifier;
  ...
};
struct struct_identifier * ptr;
```

- the two following expressions share the same semantics

```
(*ptr).field_identifier   ~      ptr->field_identifier
```


Maurizio Martignano
Copyright © 2005-2006

C - Pointers & Functions

- Syntax

```
<type name> (*identifier)( <parameter list> );
```

- Description
  - A pointer to a function is best thought of as an address, usually in a code segment, where that function's executable code is stored; that is, the address to which control is transferred when that function is called. The size and disposition of your code segments is determined by the memory model in force, which in turn dictates the size of the function pointers needed to call your functions (see the example callbck.c).
  - A pointer to a function has a type called "pointer to function returning type," where type is the function's return type.

Maurizio Martignano
Copyright © 2005-2006

C - Memory Allocation Problems

- Most of the memory allocation problems derive from a misuse of pointers.
- Let's have a look to some examples…
  - 1. Exercise 7 (dangling pointers)
  - 2. Example pexa0.c (pointers and integers arithmetic)
  - 3. Example pexa1.c (failure in pointer initialisation)
  - 4. Example pexa2.c (misuse of pointers)

Maurizio Martignano
Copyright © 2005-2006

C - Interrupt/Exception Handling

- It is very important to distinguish between:
  - **interrupt** - an **asynchronous** event, often generated by the HW or the user interaction;
  - **exception/fault/trap** - a **synchronous** event, often generated by SW.
- Interrupt and faults can be caught in C using the **signal** mechanism.
- Once a signal is caught, it is possible to perform some (recovery) action via the **setjmp/longjmp** mechanism.

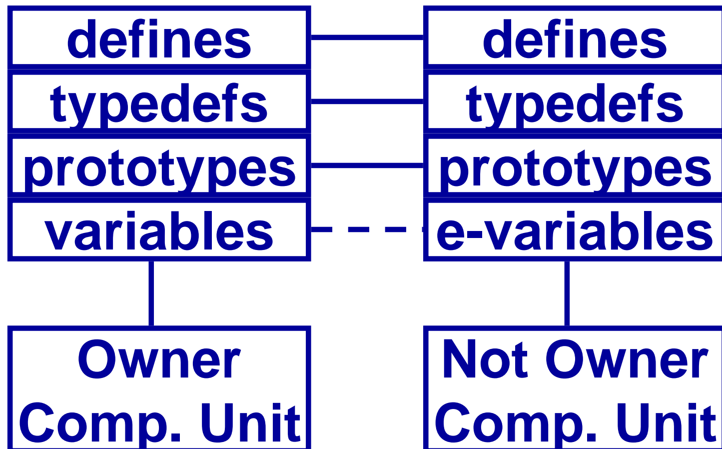
Maurizio Martignano
Copyright © 2005-2006

C - Interrupt/Exception Handling (cont)

- A **longjmp** is a "long goto" which crosses the functions boundaries.
- From a given function it is possible to longjump only to other functions which are already present in the stack. This is like "unwinding" the stack (see example signal.h).
- Memory allocated in the heap which is only accessible via pointers stored in the activation records of the skipped (longjumped) functions is lost.

Maurizio Martignano
Copyright © 2005-2006

C - Partitioning a Program into Modules



- When **more compilation units are used** to build a program it is important to devise a **proper structure of the header files** (*.h) so that all the compilation units (modules) interface properly.
- Each static data item can only be “owned” by a single compilation unit; the other compilation units can only refer to it as **extern** data item.

Maurizio Martignano
Copyright © 2005-2006

C - Partitioning a Program into Modules (c.)

- Defines, typedefs and prototypes can be repeated safely in each module because they do not refer to any memory allocation.
- Variables can only be allocated in a single module, their owner. In all the other modules their declaration has to be prefixed with the keyword **extern**.
- The keyword **static** can be used to confine a variable/function inside a single module (see the example files mod1.c and mod2.c).
- Conditional compilation should be used to prevent multiple inclusion.

Maurizio Martignano
Copyright © 2005-2006

C - Declaration vrs. Definition

- From the C++ Standard (it'll be also valid for C9X):
  - A declaration introduces names into a translation unit or redeclares names introduced by previous declarations. A declaration specifies the interpretation and attributes of these names.
  - A declaration is a definition unless it declares a function without specifying the function's body, it contains the **extern** specifier (…) or it is a **typedef** declaration (…).
- In simpler words:
  - in a **definition some memory allocation** takes place;
  - in a **pure declaration no memory allocation** takes place.

Maurizio Martignano
Copyright © 2005-2006

C - C9X

- C9X differs from C89 in the following areas:
  - Environment
  - Preprocessor
  - Syntax
  - Semantics
  - Library Annexes

  [taken from Thomas Wolf's http://lglwww.epfl.ch/~wolf/c/c9x_changes.html]
- Environment
  - Most of the translation limits (5.2.4.1) have been increased, e.g. 63 [31] significant intial characters for an internal identifier, 31 [6] significant initial characters for an external identifier, 4095 [509] characters in a logical source line, etc...

Maurizio Martignano
Copyright © 2005-2006

C - C9X (cont)

- Preprocessor
  - The #pragma directive has three reserved forms, all starting with the pp-token STDC right after "pragma". These are used to specify certain characteristics of the floating point support to comply with IEC 559.
  - The _Pragma unary operator allows the construction of pragmas through macro expansion.
  - Predefined macro __STDC_VERSION__ has now the value 199901L. (In C94, it's value was 199409L, C89 didn't have it at all.)
  - There are two conditionally defined macros, __STDC_IEC_559__ and __STDC_IEC_559_COMPLEX__, indicating IEC 559 conformance for floating point and complex arithmetic, respectively. If defined, they're defined to the decimal constant 1. A third conditionally defined macro called __STDC_ISO_10646__ shall indicate that wchar_t is in accordance with ISO/IEC 10646. If defined, this macro has a value of the form yyyymmL.

Maurizio Martignano
Copyright © 2005-2006

C - C9X (cont)

- Macro expansion: empty arguments are explicitly allowed. (In C89, this resulted in undefined behavior.) Stringification (the # operator) of an empty argument yields the empty string, concatenation (##) of an empty argument with a non-empty argument produces the non-empty argument, and concatenation of two empty arguments produces nothing at all.

- Function-like macros with variable arguments, uses the ellipsis (...) notation. For replacement, the variable arguments (including the separating commas) are "collected" into one single extra argument that can be referenced as __VA_ARGS__ within the macro's replacement list. __VA_ARGS__ may occur *only* within the replacement list of a function-like macro having a variable argument list. It's possible to have *only* variable arguments, as in

```
#define My_Macro(...) __VA_ARGS__
```


Maurizio Martignano
Copyright © 2005-2006

C - C9X (cont)

- The #line directive allows the specification of a line number up to 2**31-1. (In C89, the limit was 2**15-1, i.e. 32767.)
- The syntax of preprocessing numbers has been changed to allow for the new binary exponents present in hexadecimal floating point constants.
- Line-comments (starting with the pp-token "//" and extending up to the end of the line). As with normal comments, it's not possible to construct a comment as the result of macro replacement

Maurizio Martignano
Copyright © 2005-2006

C - C9X (cont)

- Syntax
  - New keywords: restrict, inline, _Complex, _Imaginary, _Bool.
  - Within a compound statement ("block"), declarations and statements can be freely mixed.
  - Digraph tokens (<: :> <% %> %: %:%:, synonym to [ ] { } # ##, are part of the language.
  - Array declarations may have a '*' between the square brackets (used for variable arrays in parameter lists).
  - In a for-loop, the first expression may be a declaration, with a scope encompassing only the loop.

```
for (decl; pred; inc)
     stmt;
```

is equivalent to:

```
{ decl;
        for (; pred; inc)
           stmt;
     }
```


Maurizio Martignano
Copyright © 2005-2006

C - C9X (cont)

– Compound literals (anonymous aggregates) can be created using the notation

```
( type-name ) { initializer-list }
```

(possibly with a trailing comma before the closing brace). Compound literals are primary expressions.

– Initializers (and anonymous aggregates) have a named notation for initializing members. For array elements, the element is designated by `[const-expression]`, for struct and union members using a dot notation `.member-name`. E.g.,

```
struct {int a[3], b;} w[] =
        { [0].a = {1}, [1].a = 2 };
```

or

```
struct {int a, b, c, d;} s =
        { .a = 1, .c = 3, 4, .b = 5};
```

Note: the '4' in the above initializer list initializes s.d.

Maurizio Martignano
Copyright © 2005-2006

C - C9X (cont)

- As usual, global data is by default set to zero (or to NULL in the case of pointers). If an initializer is present, any members not explicitly set also are zeroed out.
- Notation for "universal characters":
  universal-character-name:
  \u *hex-quad*
  \U *hex-quad hex-quad*
  hex-quad:
  hexadecimal-digit hexadecimal-digit
  hexadecimal-digit hexadecimal-digit
  Note that universal characters may appear even in the midst of an identifier! (An implementation is allowed to do some name mangling if the linker cannot deal with universal characters.)
- Notation for hexadecimal floating point constants with binary exponent, i.e., the exponent is given as a decimal power of two.

Maurizio Martignano
Copyright © 2005-2006

C - C9X (cont)

- New suffix "LL" or "ll" (and "ULL" and "ull", of course) for constants of the new long long types.

- **Semantics**
  - Floating point arithmetic defined such that it can comply with the IEC 559 standard ("Binary floating-point arithmetic"), also known as IEEE 754 (and IEEE 854).
  - New type long long (signed and unsigned), at least 64 bits wide.
  - New identifier `__func__`, which is declared implicitly if used within a function asstatic const char __func__[] = "*function-name*";

    ```
    static const char __func__[] = "function-name";
    ```

    where *function-name* is the unadorned name of the function the identifier is used in. (Provides a means to obtain the name of the current function, similar to the __FILE__ macro. It's a variable instead of a macro because the preprocessor doesn't know about functions.)

Maurizio Martignano
Copyright © 2005-2006

C - C9X (cont)

- Initializers for auto aggregates can be non-constant expressions.
- The integer division and modulus operators are defined to perform truncation towards zero. (In C89, it was implementation-defined whether truncation was done towards zero or -infinity. This is (obviously) important only if one or both operands are negative. Consider:

  -22 / 7 = -3 truncation towards zero
  -22 % 7 = -1
  -22 / 7 = -4 truncation towards –infinity
  -22 % 7 = 6

  Both satisfy the required equation (a/b)*b + a%b == a. The second has the advantage that the modulus is always positive -- but they decided on the other (more Fortran-like, less Pascal-like) variant...)
- Type specifiers: new combinations added for:
  - `Bool`
  - `float _Complex, double _Complex, long double _Complex`
  - `signed and unsigned long long int.`
- Note: These type specifiers may occur in any order.

Maurizio Martignano
Copyright © 2005-2006

C - C9X (cont)

- The implementation of the complex types is defined by the standard (6.2.5(13)) to use cartesian coordinates (real and imaginary part), i.e. forbids an implementation using polar coordinates (distance from [0,0] and an angle). Furthermore, the same paragraph also specifies that a complex type has the same alignment requirements as an array of two elements of the corresponding floating types, the first must be the real part and the second the imaginary part.
- Objects of the new boolean type `_Bool` may have one of the two values zero or one.
- In a declaration, there must be at least one type specifier, i.e., the default to int has been thrown out. E.g., the declaration

  ```
  f();
  ```

  was equivalent to `int f();` in C89, but is illegal in C9X.

Maurizio Martignano
Copyright © 2005-2006

C - C9X (cont)

– Structs: the last member may have an incomplete array type. (This is a way to codify the well-known "struct hack" that was widely used and in practice worked on nearly every compiler.) The idea is illustrated by the following piece of code:

```
 struct s {int n; double d[];};
struct s *p1, *p2;
size_t sz; sz = sizeof (struct s);
                // sz == offsetof (struct s, d)
p1 = malloc (sizeof (struct s) + 8 * sizeof (double));
p2 = malloc (sizeof (struct s) + 5 * sizeof (double));
/* p1 behaves now as if it had been declared as
       struct {int n; double d[8];} *p1;
    p2 behaves now as if it had been declared as
       struct {int n; double d[5];} *p2;
*/
```

Note that the specification as given in the Committee Draft implies that there be no padding before the variable last member, or, if there is, that it be included in `sizeof (struct s)`.

Maurizio Martignano
Copyright © 2005-2006

C - C9X (cont)

- Type qualifiers are idempotent, i.e., if a type qualifier appears several times (either directly or indirectly through typedefs) in a type specification, it's treated as if it appeared only once. E.g . `const const int i;` is equivalent to `const int i`;.
- There's a new type qualifier, called `restrict`. It's intended to be used only for pointer types (6.5.3(2)). Its semantics is that two restrict-qualified pointers cannot be aliases of the same object. A restricted pointer and a non-restricted point *can* be aliases, though.
- There's a new function specifier `inline`, giving the compiler a hint that such a function should be inlined.
- A compiler must parse and accept both `restrict` and `inline`, but is free to ignore the hints given by them.

Maurizio Martignano
Copyright © 2005-2006

C - C9X (cont)

- There are variable-length arrays, whose size depends not upon a constant expression but on a computed value. Variable-length arrays must not be global or members of a struct or union. Multi-dimensional variable-length arrays are allowed.
- The `goto` statement is not allowed to jump *into* the scope of a variable-length array. Jumps *within* such a scope are allowed.

◆ Library

- New `<stdbool.h>`, containg a typedef for bool and macros for true and false.
- The `<iso646.h>` header, containing alternatives for difficult character sequences.
- `<errno.h>` contains a new predefined macro EILSEQ. Used to report errors in wide-character conversion.

Maurizio Martignano
Copyright © 2005-2006

C - C9X (cont)

- New `<inttypes.h>`, giving typedefs specifying integer types with
  - exactly *n* bits
  - at least *n* bits
  - the fastest (whatever that means) type having at least *n* bits

  where *n* in [8, 16, 32, 64]. Also defines for each of these types macros expanding to the correct format specifiers for the printf and scanf families, as well as macros expanding to the correct suffixes for constants (e.g., UINT64_C (0x123) might expand to 0x123ULL) and for the maximum and minimum values of these types.
- New file `<fenv.h>`, providing access to the floating point state. (To conform to IEC 559.)
- `<math.h>` contains some new low-level functions (e.g. is_nan or copysign) as well as some configuration macros and a pragma to comply with IEC 559. Also contains new high-level functions, e.g. gamma.
- `<complex.h>` provides mathematical functions for the new complex types.

Maurizio Martignano
Copyright © 2005-2006

C - C9X (cont)

- Annexes
  - Annexes C and D (informative) detail the model of sequence points.
  - New annex F (normative) details the IEC 559 floating point model and its support in C.
  - Annex G (informative) describes IEC 559 conformant complex arithmetic.
  - Annex H (informative) describes to what extent C conforms to the ISO/IEC 10967-1 standard on language-independent arithmetic.
  - Annex I (normative) defines the ranges of legal values for universal character names in the source character set.

Maurizio Martignano
Copyright © 2005-2006

C - C9X (cont)

- `<tgmath.h>` stands for "type-generic math" and defines some macros that automagically call the right function from `<math.h>` or from `<complex.h>` depending upon the type of their arguments.
- `<stdarg.h>` has a new function va_copy to copy a variable argument list.
- The file model of `<stdio.h>` has been extended to cover also files with multi-byte or wide characters. There are some additional functions, most notably snprintf (like sprintf, but allows the programmer to specify the length of the result buffer) and a vscanf family (in analogy to vprintf).
- `<stdlib.h>` has a few new routines for conversions of long long, e.g. atoll.
- `<time.h>` has a new type struct tmx, which is like struct tm but contains a few more fields dealing with leap seconds. There are also a few new routines operating on this new structure.
- `<wctype.h>` contains a lot of wide-character handling functions, including formatted I/O and numeric conversions.

Maurizio Martignano
Copyright © 2005-2006

C - Summary

- Program Structure
- Preprocessor Directives
- Variable Declarations
- Types Definitions
- Expressions
- Control Structures
- Functions
- Pointers
- Memory Allocation Problems
- Interrupt/Exception Handling
- Partitioning a program into modules.
- Declaration vrs. Definition
- C9X

Maurizio Martignano
Copyright © 2005-2006

C - Exercises

- 1. Modify the macro.c program so that the macro MIN is converted to a normal function (e.g. `int my_min(int a, int b)`).
- 2. Modify the original macro.c program replacing the line

  ```
  c = MIN(a, b);
  ```

  with

  ```
  c = MIN(a++, b);
  ```

  and comment the results.
- 3. Modify the format strings in the sizes.c program so that the output is properly aligned.
- 4. Modify the original version of the program enum.c replacing the line

  ```
  day = sat;
  ```

  with

  ```
  day = 100;
  ```

  and comment the results.

Maurizio Martignano
Copyright © 2005-2006

C - Exercises (cont)

- 5. Modify the logic in the function move_disks in program hanoi.c so that only one statement of the type

  printf("Move disk %d from %c to %c\n", disks, from, to);

  is needed. The output of the program must not change.
- 6. Modify the program swap.c by using pointers. This time the function swap has to work.
- 7. Remove the keyword `static` in the declaration of `buf[2+1]` in the program enum.c and comment the results.
- 8. Use the swap function of exercise 6 to write a program which reads, sort and print out an array of integers.
- 9. Modify the files mod1.c and mod2.c so that they use an external header file (mod.h). Clearly identify in this header file the different sections (I.e. defines, typedefs, prototypes, variables). Use the conditional compilation mechanism to prevent multiple inclusion.

Maurizio Martignano
Copyright © 2005-2006

C - Further Reading

- 1. Kernighan, Ritchie, "The C Programming Language, 2/E - ANSI", PTR Prentice Hall, 1988, ISBN 0.13-110362.8.
- 2. Herbert Schildt, “C: The Complete Reference, 2/E”, Osborne McGraw-Hill, 1993, ISBN 0-07-881538-X.

Maurizio Martignano
Copyright © 2005-2006