

# Chapter 4: C Syntax II

## (Declarations, Procedures)

### References:

- Kernighan/Ritchie: The C Programming Language, 2nd Ed. Prentice-Hall, 1988.
- Harbison/Steele Jr.: C — A Reference Manual, 4th Ed. Prentice-Hall, 1995.
- Kammerer: Von Pascal zu Assembler (in German), 2. Aufl, Vieweg 2001.
- Link: Assembler Programmierung (in German), 9. Aufl. Franzis, 2000.
- Müller: Assembler Referenz (in German), 2. Aufl. Franzis, 2000.
- Güting/Erwig: Übersetzerbau (in German), Springer, 1999.
- Aho/Sethi/Ullman: Compilers — Principles, Techniques, and Tools. Addison-Wesley, 1986.
- Wirth: Grundlagen und Techniken des Compilerbaus (in German). Addison-Wesley, 1996.
- Online Documentation of Microsoft Visual C++ 6.0 (Standard Edition).

# Overview

1. Procedures/Functions: Call and Return

2. Procedures/Functions: Parameters

3. Global and Local Variables

4. Declaration Syntax

# Functions/Procedures (1)

- The input file to a C compiler (“translation unit”, “source file”) consists of a sequence of declarations (types, variables) and function definitions.

All statements in C must appear inside function definitions.

- Functions are named pieces of program code. In C, “function” and “procedure” are synonyms.

Other names are subprograms, subroutines, methods.

They are the main structuring mechanism for the program code.

- Functions can call each other, i.e. request to execute the program code (body) of a function.

# Functions/Procedures (2)

- Example for a function definition:

```
(1)  int square(int n)
(2)  {
(3)      return n * n;
(4)  }
```

- Each function has
  - ◇ a name, e.g. `square`,
  - ◇ a result type, e.g. `int`,
  - ◇ a parameter list, e.g. `int n`,
  - ◇ a body (block: declarations and statements).

# Functions/Procedures (3)

- Function names in a source file must be unique: there cannot be two functions with the same name.

In contrast, C++ permits overloading, i.e. there can be two functions/methods with the same name but different parameter types.

- Exactly one of the functions of the program is called “`main`”. Execution begins in this function.
- Other functions are only executed if they are called (directly or indirectly) from `main`.
- Functions are called by using them in expressions, e.g. `m = square(5);`

## Functions/Procedures (4)

- The **return type** can be **void**. That means that the function does not return a value.

It is possible not to specify a result type in the function definition. Then C assumes **"int"**.

- A function cannot return an array or a function.

However, it can return a pointer, also a pointer to a function.

- A function can return structures and unions.

Structures/Unions are copied upon return. One probably should avoid large structures.

# Functions/Procedures (5)

- Functions should be declared before they are called, so that the compiler knows parameter and result types (and can report possible errors in the call).
- However, C permits that functions are called without previous declaration.

The reason is historical: In the first version of C, forward declarations (see below) did not contain parameter types, the compiler itself did not check arguments.

- Then the compiler does not check the types of the arguments and assumes the result type `int`.

But modern compilers will print a warning in this case.

# Functions/Procedures (6)

- One can first write a (forward) declaration for a function (without the body), and define it later:

```
(1)  #include <stdio.h>
(2)  int square(int n);
(3)
(4)  int main()
(5)  { printf("The square of 5 is: %d\n",
(6)          square(5));
(7)    return(0);
(8)  }
(9)
(10) int square(int n) { return n * n; }
```

# Functions/Procedures (7)

- Some authors make a careful distinction between:
  - ◇ The **declaration** of a function or a variable: It specifies the type, but does not actually produce machine instructions (for functions) or reserve memory (for variables).
  - ◇ The **definition** of a function/variable: It includes the declaration but it specifies also the body of the function / reserves memory for the variable.
- However, many people do not make this distinction and always use the word “declaration”.

# Functions/Procedures (8)

- Of course, a (forward) declaration and the later definition of a function must agree in the result type and in the number and types of the parameters.
- However, it is not necessary that the parameter names agree.

Maybe some compilers print warnings if the parameter names do not agree.

- Therefore, parameter names are not required in forward declarations. E.g. one can declare `square` as

```
int square(int);
```

# Functions/Procedures (9)

- `void` can also be specified as parameter list.

This means that the function has no parameters.

In function definitions (that include a body), “`()`” and “`(void)`” are equivalent. In declarations without a body (see below) they are not: “`()`” is an old-style declaration. In the first version of C (as defined in the Kernighan/Ritchie published in 1978), function declarations did not specify parameters and the compiler did not check the types of parameters. E.g. the forward declaration for `square` was written as: “`int square();`”. There was an additional tool called `lint` that collected the parameter information from the actual function definitions and did parameter type checking. In order to be compatible to this old version of C, the parameter list “`()`” means that the compiler will not check the arguments in the call. One must write “`(void)`” to make clear that there are no parameters.

# Functions/Procedures (10)

Exercise: What errors/warnings will the compiler print?

```
(1)  int p(void);  
(2)  int q(char c);  
(3)  
(4)  void f(int n)  
(5)      { p(n, 1); }  
(6)  p()  
(7)      { int i = f(5); }  
(8)  int q(char x)  
(9)      { return x; }  
(10) int main()  
(11)     { int j = g(1); return j; }
```

# Library Functions (1)

- C comes with a “standard library” that contains a large number of useful functions, especially functions for interfacing with the operating system.

E.g. for input/output.

- Library functions are functions that were written by somebody else, but that can be called as if you had defined them yourself.

They are written for the most part in C, but small parts may be written in assembler (a symbolic form of machine code).

## Library Functions (2)

- Even library functions like `printf` should be declared before they are called. This is done by including “header files” like `stdio.h`.

The effect of `#include <stdio.h>` is that the compiler reads the specified file at this point (just as if one had copied the contents of `stdio.h` into the main file).

- It is possible to look into these header files to see the declaration of the library functions.

Under UNIX, the standard location for include files is `/usr/include`. The Microsoft Visual C++ include files are stored e.g. in `C:\Program Files\Microsoft Visual Studio\VC98\Include`.

## Library Functions (3)

- It is legal but confusing to declare functions with the same name as library functions (explicitly declared functions take precedence).

This can have unexpected effects. E.g. I once wrote a function “`read`” that called the library function “`getc`”. Unfortunately, “`getc`” normally calls another library function called “`read`”. However, my function “`read`” was called instead. This violates the programming language principle “It should not be necessary to know things one does not use.” It might be that more modern systems avoid such problems by resolving references in libraries before they are used, but one cannot rely on that.

# Library Functions (4)

- Library functions exist already in precompiled form (i.e. as machine code).

The header file contains only the declaration of the library function, but not the definition (i.e. not the procedure body).

- References to library functions are resolved not by the compiler itself, but by a second program called the “**linker**”.

As long as the C program consists of a single source file, the compiler will usually automatically call the linker, so the distinction is not very important at the moment. We will later consider programs consisting of several source files.

## Library Functions (5)

- When one calls functions that are not defined the final error message comes from the linker (“unresolved references to external symbols”).

The compiler may only print a warning if the function is not declared when it is called.

- Since some compilers slightly modify function names, the error message of the linker may show a different function name than was actually called.

Most C compilers add an underscore in front of all external symbols. In C++, the “name decoration” is much more involved (because of classes and overloading).

# Procedure Call & Return (1)

- The compiler generates a piece of machine code for each function.
- When one function  $f$  calls another function  $g$ , the machine code for  $f$  performs a “goto” to the beginning of the machine code for  $g$ .

In C, “goto” is permitted only within a function, one cannot jump to a statement in another function. But in machine code, one can continue execution with any address in memory.

- At the end of  $g$  (or when a “return” is executed), it must jump back to  $f$ , more specifically, to the instruction following the function call.

## Procedure Call & Return (2)

- Since the same function can be called from several places, the return address is not fixed.
- Therefore, the caller  $f$  stores the return address in memory before it jumps to the beginning of  $g$ .
- When  $g$  is finished, it jumps to address stored in that memory location.
- It is possible that  $g$  calls another function  $h$ . Thus, not always the same memory location can be used.

## Procedure Call & Return (3)

- It might seem possible that this point that for each function, there is only one memory address that contains the return address for that function.
- But it is even possible that a function calls itself:

```
(1)  int factorial(int n)
(2)  {
(3)      if(n <= 1)
(4)          return 1;
(5)      else
(6)          return n * factorial(n-1);
(7)  }
```

# Procedure Call & Return (4)

- Functions/Procedures that directly or indirectly call themselves are called **recursive**.

“**factorial**” above is a recursive function. However,  $n!$  can also easily be computed iteratively (with a loop), and iterative formulations are often slightly faster than recursive ones (because of the overhead for the procedure call). Some compilers even transform tail recursions (where the recursive call is more or less the last thing the function does) into loops. But functions that work on tree structures can often be written elegantly as recursive functions, whereas iterative solutions for such functions are quite difficult (although possible).

- E.g. **factorial(2)** calls **factorial(1)**. When that is done, it returns to **factorial(2)** which was only temporarily suspended.

# Procedure Call Stack (1)

- Because several “incarnations” of the same procedure can exist at the same time, the compiler creates an array to manage return addresses.
- This array is called the “**procedure call stack**” or simply “the stack”.
- There is also a pointer (index) that points to the next free position in the array (“**stackpointer**”).

The stackpointer is usually stored in a register within the CPU (very fast memory). E.g. Intel processors have a special register SP for the stack pointer. The register SS (“stack segment”) defines the position of the array.

## Procedure Call Stack (2)

- When a function  $f$  calls a function  $g$ , it
  - ◇ stores the return address (some position in  $f$ ) at the next free array position
    - I.e. the position to which the stackpointer points.
  - ◇ then increments the stackpointer,
  - ◇ and finally jumps to the beginning of  $g$ .
- When a function (e.g.  $g$ ) is finished, it
  - ◇ decrements the stackpointer, and
  - ◇ jumps to the address stored in the array position to which the stackpointer points.

# Procedure Call Stack (3)

**Exercise:** Simulate this program (especially the stack):

```
(1)  #include <stdio.h>
(2)  void p(void);
(3)  void q(void);
(4)
(5)  int main()
(6)  {
(7)      putchar('P', stdout); /* printf("P"); */
(8)      p();
(9)      putchar('S', stdout); /* printf("S"); */
(10)     q();
(11)     return(0);
(12) }
```

# Procedure Call Stack (4)

```
(13) void p(void)
(14) {
(15)     putchar('A', stdout);
(16) }
(17)
(18) void q(void)
(19) {
(20)     putchar('C', stdout);
(21)     p();
(22)     putchar('L', stdout);
(23) }
```

Use line numbers as return addresses when you show the development of the stack. What does the program print?

# Procedure Call Stack (5)

- The data structure is called a stack because like a stack of plates, one always takes off only the topmost element (return address).

One cannot take a plate from the middle of the pile before one has taken all the plates above it.

- The topmost element is the one that was put last on the stack (among those still on the stack).
- Thus, stacks operate according to the **LIFO-principle: “last in, first out”**.

In contrast, queues follow the FIFO-principle: First in, first out.

# Procedure Call Stack (6)

- Stacks have two main operations:
  - ◇ `push(e)`: Puts an element `e` on the stack.
  - ◇ `e = pop()`: Takes the topmost element from the stack and assigns it to `e`.
- The following operations are slightly less common:
  - ◇ `clear()`: Initializes the stack to the empty set.
  - ◇ `b = empty()`: Checks whether the stack contains any elements (`b` is 1 if empty, 0 otherwise).
  - ◇ `e = top()`: `e` is set to the topmost stack element, but that element is not taken from the stack.

# Procedure Call Stack (7)

- Depending on the hardware (CPU),
  - ◇ the stack may grow towards smaller addresses,

Then the stack pointer points initially to the last array element. The push operation decrements it and the pop operation increments it.
  - ◇ and the stack pointer may point to the last used element instead of the next free element.
- E.g. this is the case in the Intel processors (and most other processors).

If one thinks only in terms of “push” and “pop”, this difference is not essential.

## Procedure Call Stack (8)

- Processors (e.g. Intel CPUs) have special “call” and “return” instructions that combine the steps that were mentioned above.
  - ◇ “CALL P” pushes the address of the next instruction on the stack and jumps to address “P”.
  - ◇ “RET” pops an address from the stack and jumps to that address.
    - RET has an optional argument that specifies the number of bytes to be removed (popped) from the stack in addition to the return address. That is only interesting for local variables (see below).

# Procedure Call Stack (9)

- The procedure call stack is a fixed size array.

The compiler usually has an option to specify its size (e.g. in MS VC++, it is set under project/settings/link/output). In earlier systems, the default size was 8KB, in MS VC++ it is 1MB.

- If it is too small, the following might happen:
  - ◇ The program is aborted with an error message.
  - ◇ Important data in memory is overwritten without warning.
  - ◇ More memory is automatically allocated.

# Procedure Call Stack (10)

- E.g. if one writes an infinite recursion, a stack overflow will happen sooner or later.
- E.g. if this version of factorial is called with a negative number, the recursion does not stop:

```
(1)  int factorial(int n)
(2)  {
(3)      if(n == 0)
(4)          return 1;
(5)      else
(6)          return n * factorial(n-1);
(7)  }
```

# Overview

1. Procedures/Functions: Call and Return

2. Procedures/Functions: Parameters

3. Global and Local Variables

4. Declaration Syntax

# Parameters (1)

- Functions can have 0, 1, or more input values.
- This is important, since otherwise a function would do exactly the same on each call.

This is a bit simplified: Besides the parameters, a function can also query global variables and get input from the keyboard, files, etc. Global variables are discussed below, but in general it is clearer to use explicit parameters.

- Program code becomes more generally usable if it is parameterized.

I.e. there are holes in it that can be filled with different values.

## Parameters (2)

- E.g. consider the following function/procedure:

```
int square(int n) { return n * n; }
```

- Suppose this function is called as: `m = square(5);`

- One usually distinguishes:

- ◇ The parameter, e.g. `n`.

The parameter is the name for the input value. Instead of parameter, one can also say formal argument or formal parameter.

- ◇ The argument, e.g. `5`.

The argument is the value for the parameter in a specific call. Instead of argument, one can also say actual argument/parameter.

## Parameters (3)

- In many languages, there are two different kinds of parameters:

- ◇ “call by value”: A data value (e.g. an integer) is passed to the procedure.

This is used for input parameters.

- ◇ “call by reference”: A variable is passed to the procedure. The procedure can query the current value of the variable and can assign a new value.

This is used for output parameters or input/output-parameters.

## Parameters (4)

- C has only “call by value” .
- However, the value can be a pointer. In this way, it is possible to pass the address of a variable to a function.
- E.g. this version of `square` returns the result in an output parameter, and not as function value:

```
(1) void square(int n, int *p)
(2) {
(3)     *p = n * n;
(4) }
```

## Parameters (5)

- The function call then looks as follows:

```
square(5, &m);
```

- This is basically the same as “call by reference”, but in the call it is explicitly shown that the address of the variable is passed, and not its value.

This might actually help the reader to understand the program more easily. Usual call by reference can modify variables in unexpected ways.

- Within the function body, one must dereference the pointer in all assignments.

## Parameters (6)

- In “call by value” parameters are treated like (local) variables that are initialized with the value specified in the call (so the value is copied).
- E.g. the procedure call

```
square(5, &m);
```

means that the following block is executed:

```
(1)  {  int n = 5;  
(2)      int *p = &m;  
(3)      *p = n * n;  
(4)  }
```

# Parameters (7)

Exercise: What does this program print?

```
(1)  #include <stdio.h>
(2)
(3)  int p(int m)
(4)  {  m++;
(5)      return m;
(6)  }
(7)
(8)  void swap(int *p, int *q)
(9)  {  int i = *p;
(10)      *p = *q;
(11)      *q = i;
(12)  }
```

# Parameters (8)

## Exercise (continued):

```
(13)
(14)  int main()
(15)  {
(16)      int x = 1, y = 2;
(17)      x = p(y);
(18)      swap(&x, &y);
(19)      x++;
(20)      printf("x = %d, y = %d\n", x, y);
(21)      return 0;
(22)  }
```

## Parameters (9)

- Parameters cannot really be arrays. If one declares parameters of array type, they are automatically converted to the corresponding pointer type.

In the same way, function types become “pointer to function”.

- E.g. the following two declarations are equivalent:

```
int f(char a[], int b[5]);  
int f(char *a, int *b);
```

- This especially means that arrays are not copied when they are passed as parameters to a function, only a pointer to the first array element is passed.

## Parameters (10)

- If the compiler has seen a function declaration before the call, the same type conversions occur as in an assignment of the arguments to the parameters.

This is the normal case. If the compiler has not seen a declaration of the function (or an “old style” declaration without parameters), special rules apply: Arguments of type `char` and `short` are automatically converted to `int`, and `float` is automatically converted to `double`. The reason for this is that e.g. no computations in C are done with `short`, so the type of `s+1` is `int` (when `s` has type `short`). Thus, the compiler cannot distinguish arguments of types `short` and `int`. If the function definition is old style, the values for the parameters are converted back to the declared parameter types. Problems may occur if the compiler has not seen the declaration of the function before the call (so it uses the “old style” rules), but it is later declared “new style”.

## Style Hints (1)

- Avoid functions having more than 6 parameters (use structures, global variables, or split the function).

Parameters and arguments are matched by position. If there are too many parameters, it might be difficult to remember their sequence or to quickly understand which argument belongs to which parameter.

- Be consistent (follow certain rules) with respect to the parameter sequence: E.g. first specify all input parameters, then all output parameters.

Also the sequence of parameters of different types should be specified. E.g. it is confusing that the file argument comes first in `fprintf`, but last in `fputs`.

## Style Hints (2)

- If a function returns only one value, one should use the function result and not an output parameter.

Function values are much easier to use: Output parameters are always something special because of the need to use the address and the dereference operators. It would be best to have always only input parameters, but some functions must return more than one value.

- I personally do not like input/output parameters (i.e. parameters that pass the address of an initialized variable to the function).

The more different possibility one uses, the more difficult programs are to read and understand. But there are always exceptions.

# Parameters of main (1)

- Most operating systems make it possible to specify arguments when a program is called.
- E.g. in the MS-DOS Prompt and the UNIX shell, a command line has the form

⟨Program⟩    ⟨Argument 1⟩    ⟨Argument 2⟩    ...

- E.g. for the command

```
copy a:c4_cdecl.tex slides
```

the program name is “copy”, the first argument is “a:c4\_cdecl.tex”, and the second is “slides”.

## Parameters of `main` (2)

- These command line arguments can be accessed in the program via parameters to `main`.
- In general, `main` has two parameters:
  - ◇ an integer that is the total number of “words” on the command line (i.e. the number of arguments plus one for the command name)
  - ◇ an array of strings that are the words on the command line (the first entry is the command name, other entries are arguments).

## Parameters of main (3)

- This program prints its command line arguments:

```
(1)  #include <stdio.h>
(2)  main(int argc, char *argv[])
(3)  {  int i;
(4)      if(argc == 1)
(5)          printf("No arguments!\n");
(6)      else
(7)          for(i = 1; i < argc; i++)
(8)              printf("Arg %d: '%s'\n",
(9)                  i, argv[i]);
(10)     return 0;
(11) }
```

## Parameters of main (4)

- An alternative for the for-loop is:

```
(7)         while(--argc > 0)
(8)             printf("%s\n", *++argv);
```

- The last element of the `argv` array is a NIL pointer, thus the following is also possible:

```
(7)         while(*++argv)
(8)             printf("%s\n", *argv);
```

However, some older implementations do not support this (so one better choose one of the above solutions).

## Parameters of `main` (5)

- In order to test such programs under MS VC++, start an MS-DOS prompt and go to the subdirectory “Debug” of the project folder — there the executable file is stored and can be directly called.

The problem is that if you click on `Build`→`Execute` as usual, the program is executed without arguments. One can specify command line options also under `Project`→`Settings`→`Debug`→`General`.

- If the command name is not available, `argv[0]` contains the empty string.

In MS VC++, `argv[0]` includes the directory path.

# Variable Parameterlists (1)

- In C, it is possible that a function has a variable number of parameters, which can also have varying types.
- This is e.g. used in the library function `printf`. It is declared in “`stdio.h`” as follows:

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

“`extern`” is the default storage class for functions (see below).

- The “`...`” at the end of the parameter list means that arbitrary arguments can still follow.

## Variable Parameterlists (2)

- C requires that functions with variable-length argument lists have at least one normal parameter.
- The normal parameter(s) must tell the function how many arguments of which types it actually has.
- The C compiler normally does not do any type checking for the additional arguments. So this mechanism is a bit dangerous.

The C compiler does not tell the function how many arguments of which types it has. It must rely on the programmer to pass arguments that are consistent with the specification in the normal parameters.

## Variable Parameterlists (3)

- However, since `printf` and `scanf` are so common, some compilers check their arguments.
- The declaration of `printf` is really

```
/* PRINTFLIKE1 */  
extern int printf(const char *, ...);
```

Some compilers understand the special comment.

The number “1” means that the format string is the first argument.

## Variable Parameterlists (4)

- The standard library header file “`stdarg.h`” declares a set of “macros” for accessing the parameters.

Macros are similar to functions. See next chapter.

- An example of using them is given on the next slide. It defines a function “`sum`” that can add any number of integers. The number of integers to add must be passed in the first argument.
- E.g. the result of `sum(3,10,20,30)` is 60.

# Variable Parameterlists (5)

## Example:

```
(1)  #include <stdio.h>
(2)  #include <stdarg.h>
(3)  int sum(int n, ...)
(4)  {   va_list args;
(5)      int result = 0;
(6)
(7)      va_start(args, n);
(8)      while(--n >= 0)
(9)          result += va_arg(args, int);
(10)     va_end(args);
(11)     return result;
(12) }
```

# Implementation (1)

- The standard implementation of parameters is to put (push) them on the stack (usually in inverse order and before the return address).
- E.g. consider the function

```
(1)  int max(n, m)
(2)  {
(3)      if(n < m)
(4)          return m;
(5)      else
(6)          return n;
(7)  }
```

## Implementation (2)

- When `max` starts executing, it knows that
  - ◇ Where the stackpointer (register `SP`) points to is the return address.

To be concrete, we assume an Intel CPU here (8085 to Pentium). The stackpointer points to the top element on the stack.

- ◇ At the address `SP+4` is the first argument `n`.

Note that on Intel processors, the stack grows from higher addresses towards lower addresses. So `n` is actually below the return address on the stack.

- ◇ At the address `SP+8` is the second argument `m`.

## Implementation (3)

- Conversely, if the compiler translates a procedure call, e.g. `"x = max(i, 0);"` it has to generate machine instructions that do the following:
  - ◇ Push the constant `0` on the stack.
  - ◇ Push the current value of the variable `i` on the stack.
  - ◇ Push the address of the next instruction on the stack and jump to the beginning of `max`.

This is done with a single instruction `"CALL max"`, where `"max"` is replaced by a specific memory address.

## Implementation (4)

- After the return from the procedure “`max`”, the caller still has to do the following:
  - ◇ Assign the return value of the function to the variable “`x`”.

The compiler could e.g. have the convention that called procedures leave the function return value in the register “`EAX`” (the “accumulator”). This is e.g. done in MS VC++.

- ◇ Remove the arguments `i` and `0` from the stack.

I.e. adjust the stackpointer with  $SP = SP + 8$ .

## Implementation (5)

- Compilers can have different “calling conventions”, i.e. rules for passing parameters and return values.

The calling convention also includes which registers the called procedure must save before it can use them (and restore at the end).

- So it is not clear that if one procedure (e.g. in a library) is translated into machine code with compiler  $X$ , it can be called from another procedure translated with compiler  $Y$ .

This is especially clear if the called procedure is written in some other language, not C. But even if both were written in C, there might be incompatibilities between the compilers.

## Implementation (6)

- In MS VC++, it is possible to choose a calling convention in the procedure declaration:
  - ◇ `extern int __cdecl max(int n, int m)`  
As explained above. This is the default.
  - ◇ `extern int __stdcall max(int n, int m)`  
As explained above, but the called procedure removes the arguments from the stack.
  - ◇ `extern int __fastcall max(int n, int m)`  
The first arguments are passed in registers.
- This is not a standard feature of C.

# Implementation (7)

- Variable parameter lists require `__cdecl`.

In this case, the called procedure does not know the number of parameters, therefore it cannot remove them from the stack. It might be that the programmer can determine the variable parameters from the fixed ones, but even if that would be reliable (it is not), the compiler cannot extract that knowledge from the program. Also, parameters cannot easily be passed via registers, since the `stdarg.h` macros manipulate stack addresses.

- The Windows API uses `__stdcall`.

The corresponding header files automatically include this keyword in all declarations. The generated code is slightly smaller with `__stdcall`, since the instructions for adjusting the stack are generated only once, and not in every call (it can also use the optional parameter to `RET`).

# Overview

1. Procedures/Functions: Call and Return
2. Procedures/Functions: Parameters
3. Global and Local Variables
4. Declaration Syntax

# Local Variables (1)

- Variables declared in a procedure can only be accessed in that procedure.
- Such variables are called “**local variables**” .  
They are local to the procedure. For “global variables” see below.
- E.g. the program on the next slide contains an error: Although **f** is called from **main**, statements in **f** cannot access variables declared in **main**.
- Variables must be declared before they are used. Their declaration is automatically revoked at the end of the procedure (block).

## Local Variables (2)

Example (“Undeclared Variable” Error):

```
(1)  #include <stdio.h>
(2)  void f(void);
(3)
(4)  int main()
(5)  {  int i = 1; /* Local Variable */
(6)      f();
(7)      return 0;
(8)  }
(9)
(10) void f(void)
(11) {  printf("%d\n", i); /* ERROR */
(12) }
```

## Local Variables (3)

- Of course, a function can pass a pointer to one of its variables to a called procedure, in which case the variable can be accessed via that pointer.

In the example, it would be sufficient to pass the value of the variable.

- But the name of the variable is still valid only within the procedure.
- The area in a program in which a declaration is valid is called the **scope** of the declaration.

Sometimes one also says scope of the name or of the variable. It is the area in which the variable is visible.

## Local Variables (4)

- C strictly enforces the rule that **variables must be declared before they can be used.**

Even an old-style C compiler prints an error message and not a warning if a variable is not declared. The reason is that the compiler cannot generate code if it does not know the type of the variable. In Fortran, variables do not have to be declared: There variables starting with I to N are integers, all other are reals. Not requiring variable declarations has led to numerous hard-to-find errors (a typing error in a variable name leads to a new variable, and not to an error message). Sometimes it is good to require a bit of redundancy from the programmer.

- In contrast, C does not strictly require that functions are declared before they are used.

## Local Variables (5)

- The “local variable” rule makes programs easier to understand, since one has to search for statements changing a local variable only within the procedure in which it was declared.

“Understanding a program” means (among other things) that one can explain the value of each variable at each line of the program. If a function call could change the values of all variables, this task would be significantly harder.

Since C normally has no protection against array boundary violations and assignments to non-initialized pointers etc., it actually can happen that variables change values in unexpected ways. These are errors that are hard to find.

## Local Variables (6)

- It is possible that variables with the same name are declared in different procedures.

These are simply distinct variables that happen to have the same name. Since in each procedure only local variables of that procedure can be accessed, it is always clear which of the variables is meant.

- In the same procedure (more precisely, in the same block), one cannot declare two variables with the same name.

Parameters are treated like local variables that are initialized in the procedure call: It is not possible to declare two parameters with the same name or a local variable with the same name as a parameter.

# Local Variables (7)

Exercise: What does this program print?

```
(1)  #include <stdio.h>
(2)
(3)  void f(void)  { int i; i = 2; }
(4)  void g(int i) { i = 3; }
(5)
(6)  int main()
(7)  { int i = 1;
(8)    f();
(9)    g(i);
(10)   printf("%d\n", i);
(11)   return 0;
(12) }
```

## Local Variables (8)

- If a procedure calls itself recursively, each incarnation has its own set of local variables:

```
(1)  int fib(int n)
(2)  {  int f1, f2;
(3)      if(n < 2)
(4)          return 1;
(5)      else {
(6)          f1 = fib(n - 1);
(7)          f2 = fib(n - 2);
(8)          return f1 + f2;
(9)      }
(10) }
```

## Local Variables (9)

- Therefore, local variables are allocated on the stack (like parameters).
- This required for recursion: The compiler cannot foresee how many copies of the same variable are needed.

So it must generate code that allocates space as needed. It cannot assigned fixed memory locations (addresses) to the variables.

- In addition, it also makes better use of memory: The memory space for local variables is only reserved as long as the procedure is active.

## Local Variables (10)

- When a procedure returns, the stackpointer is adjusted to its value before the procedure call. Thus, local variables of the procedure cease to exist.

Their memory space is reused for the next procedure call.

- Since the lifespan of local variables is the time the corresponding procedure call is active, it is an error to pass addresses of local variables to the caller.

In C, this is not an error, the compiler will give at most a warning (if at all). But it can crash the program, so it is also not legal. Algol68 has strict rules for this: It was forbidden to store the address of an object in a pointer variable that would live longer than that object.

# Local Variables (11)

Exercise: Is this program legal? What will happen?

```
(1)  #include <stdio.h>
(2)  void f(int n)
(3)  {   int i;
(4)      if(n == 1) i = 1;
(5)      if(n == 2) i++;
(6)      if(n == 3) printf("%d\n", i);
(7)  }
(8)
(9)  int main()
(10) {   f(1); f(2); f(2); f(3);
(11)     return 0;
(12) }
```

# Local Variables (12)

Exercise: And what about this program?

```
(1)  #include <stdio.h>
(2)
(3)  int *f(void)
(4)  {
(5)      int i;
(6)      i = 1;
(7)      return &i;
(8)  }
(9)
(10) void g(int *p)
(11) { *p = 0; }
```

The program is continued on the next slide.

# Local Variables (13)

## Exercise (continued):

```
(12)  int main()
(13)  {
(14)      int *p;
(15)      p = f();
(16)      g(p);
(17)      return 0;
(18)  }
```

The sequence on the stack is: Parameters, return address, and then local variables.

# Registers (1)

- CPUs usually contain memory for a few variables. This memory is extremely fast, but also very limited. The storage locations are called registers.
- Many machine instructions require that at least one operand is in a register. The result of the operation is also usually stored in a register.
- Depending on the processor architecture, registers can be more or less specialized.

Some CPUs have a small set of general registers, that can be explicitly selected in instructions. Some CPUs have specialized registers: Certain operations automatically use a specific register.

## Registers (2)

- Registers of the Intel 8086 processor (continued on the next slide):
  - ◇ Data registers (16 bit): **AX** (Accumulator), **BX** (Base), **CX** (Counter), **DX** (Data).

Each register can also be used as two 8-bit registers: E.g. **AX** can be split into **AH** (the higher valued bits) and **AL** (the lower valued bits). On earlier 8-bit CPUs, the register was simply called “A”.
  - ◇ Address registers (16 bit): **SP** (stack pointer), **BP** (base pointer), **SI** (source index), **DI** (destination index).

## Registers (3)

- Registers of the 8086 processor, continued:

- ◇ Instruction pointer (16 bit): **IP**.

- ◇ Flag register.

The flag register contains boolean variables: **CF** (carry), **AF** (auxiliary carry), **PF** (parity), **ZF** (zero), **SF** (sign), **TF** (trap: single step), **IF** (interrupt enable), **DF** (direction of string operations), **OF** (overflow). It also contains the 2-bit value **IOP** (I/O protection level).

- ◇ Segment registers (16-bit, modifies addresses to 20 bit): **CS** (code segment), **DS** (data segment), **SS** (stack segment), **ES** (extra segment).

## Registers (4)

- From the 80386 onward (including Pentium), registers were extended as follows:
  - ◇ Data registers now 32-bit: **EAX, EBX, ECX, EDX**.
  - ◇ Address registers now 32-bit: **ESP, EBP, ESI, EDI**.
  - ◇ Instruction pointer now 32-bit: **EIP**.
  - ◇ The segment registers remain 16-bit, but they now only select segment descriptors in memory. Two segment registers were added: **FS, GS**.
  - ◇ Several new flags were introduced.

## Registers (5)

- Until the 386, the floating point unit was a distinct processor (co-processor).
- Starting with the 486 (and including the Pentium), the floating-point unit was integrated into the main CPU.
- Therefore, 486 and Pentium have also 8 registers for floating-point numbers (80 bit each, plus 2 bit tag field).

Also other registers from the FPU were added (e.g. status register, data pointer).

# Implementation (1)

- The stack space allocated for a procedure call is called an “activation record” or “stack frame”.
- It contains (continued on next slide):
  - ◇ The actual parameters (arguments).
  - ◇ The return address.
  - ◇ Saved values of registers.

Procedures might have to make sure that registers (very fast and very small memory inside the CPU) have the same values when it returns as they had when it was called. Since the procedure itself wants to use the registers, it saves them into slower memory (the stack) at the beginning and restores them at the end.

## Implementation (2)

- Contents of an activation record (continued):
  - ◇ Local variables.
  - ◇ Temporary data.

When complex expressions are evaluated, intermediate results need to be stored somewhere. If they do not fit into the registers, they are saved on the stack.

- Since temporary data is pushed on the stack and popped again while expressions are evaluated, the stack pointer is not constant while the procedure body is executed.

## Implementation (3)

- In order to have a fixed position in the stack frame, many compilers use a second register (besides the stack pointer) that stays stable while the procedure body is evaluated.
- This register is called the “frame pointer” or “base pointer” .

MS VC++ normally uses the register `EBP` for this.

- Arguments and local variables are then addressed via the frame pointer (+/- an offset).

## Implementation (4)

- Traditionally, the frame pointer points at the saved value of the frame pointer register on the stack: In this way, one gets a dynamic link chain of all activation records on the stack.
- The frame pointer is not really needed: The compiler can calculate the offsets from the stack pointer.

Unless dynamic data is allocated on the stack: Some C implementations have a procedure `alloca` or `_alloca` to reserve memory on the stack (non-standard). In MS VC++ (Professional and Enterprise Editions), there is a compiler option `/Oy` to eliminate the frame pointer.

## Implementation (5)

- Consider the procedure:

```
(1)  int max(int n, int m)
(2)  {
(3)      int r = n;
(4)      if(m > r)
(5)          r = m;
(6)      return r;
(7)  }
```

- Suppose that the procedure is called as follows:

```
i = max(3,5);
```

## Implementation (6)

- The compiler generates the following code for the procedure call `i = max(3,5)`:

```
1000 PUSH 5
1002 PUSH 3
1004 CALL max (2000)
1011 ADD ESP, 8
1014 MOV DWORD PTR[EBP-4], EAX
```

The two `PUSH` instructions put the parameters on the stack (note that the right parameter is pushed first). The `ADD` instructions removes them again from the stack (by adding 8 to the stackpointer `ESP`). The function leaves the return value in the accumulator `EAX`. The `MOV`-instruction copies it to the variable `i` on the stack (it has the offset `-4` from the frame pointer `EBP` and is a double word: 32 bits).

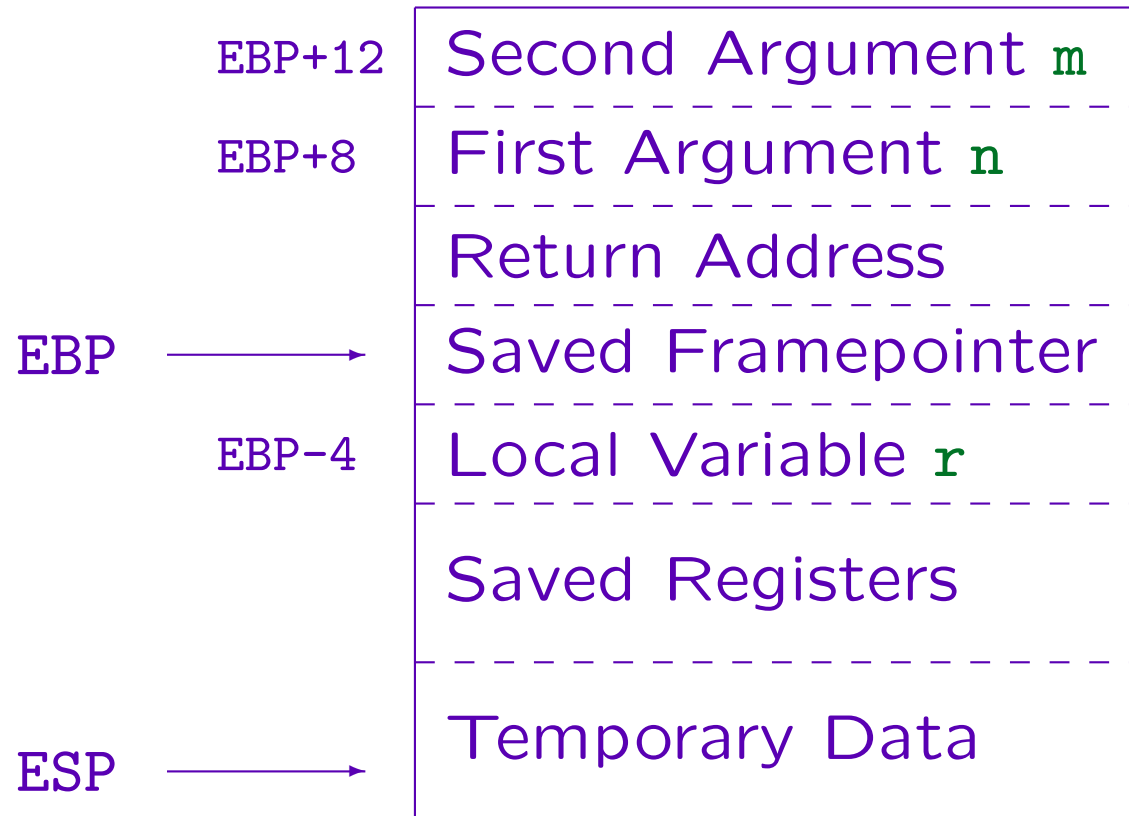
# Implementation (7)

- The **procedure prologue** (start of the procedure) looks as follows (2000 is the start address of `max`):

```
2000  PUSH  EBP
2001  MOV   EBP, ESP
2003  SUB   ESP, 4
2006  PUSH  EBX
2007  PUSH  ESI
2008  PUSH  EDI
```

This saves the frame pointer (from the calling procedure) on the stack and then sets the frame pointer register `EBP` to the current value of the stack pointer. In this way, the frame pointer points to the old frame pointer on the stack. The `SUB` reserves space for the local variable `r` (4 bytes). Then several registers are saved on the stack.

# Implementation (8)



## Implementation (9)

- The initialization `int r = n` is translated as follows (the declaration itself generates only indirectly code that reserves space for the variable on the stack):

```
2009  MOV  EAX, DWORD PTR [EBP+8]
2012  MOV  DWORD PTR [EBP-4], EAX
```

The first instruction load the value of the first argument `n` into the accumulator `EAX`, the second instruction stores this value into the local variable `r`. On many machines (CPUs), one of the arguments of an instruction must be a register, therefore memory-to-memory copies are done as shown here via a register (the accumulator).

# Implementation (10)

- The instruction `if(m > r)` is translated as follows:

```
2015  MOV  ECX, DWORD PTR [EBP+12]
2018  CMP  ECX, DWORD PTR [EBP-4]
2021  JLE  2029
```

The first instruction loads the value of the second argument `m` into the register `ECX`. The second instruction compares the value in `ECX` (i.e. `m`) with the value of the local variable `r` (it does the subtraction `m-r` without storing the result). This sets flags (boolean variables in the CPU). The instruction `JLE` (“jump if less than or equal”) jumps to the given address if the “sign flag” `SF` is set (meaning that the result was negative) or the zero flag `ZF` is set (actually, it also respects the overflow flag `OF` which means that the sign is inverted).

# Implementation (11)

- The instruction `r = m` is translated as follows:

```
2023 MOV EDX, DWORD PTR [EBP+12]
2026 MOV DWORD PTR [EBP-4], EDX
```

The second parameter `m` is stored at address `EBP+12` on the stack, the local variable `r` at address `EBP-4`.

- The instruction `return r` is translated as follows (it also includes the procedure epilogue below):

```
2029 MOV EAX, DWORD PTR [EBP-4]
```

Functions leave their result in the accumulator `EAX`.

## Implementation (12)

- The **procedure epilogue** finishes the procedure:

```
2032 POP EDI
2033 POP ESI
2034 POP EBX
2035 MOV ESP, EBP
2037 POP EBP
2038 RET
```

This restores all registers to their old value before the procedure call. Copying the value of **EBP** into **ESP** removes the local variable from the stack (actually, it is still there, but the space is reclaimed by adjusting the stack pointer). **RET** jumps to the address on top of the stack (that was pushed by **CALL**) and removes it.

# Register Variables (1)

- It is possible to add the keyword “`register`” to the declaration of a local variable or a parameter:

```
(1)  int square(register int n)
(2)  {
(3)      register int r;
(4)      r = n * n;
(5)      return r;
(6)  }
```

- This gives a hint to the compiler that this variable/parameter is very often used, so it would be good to keep it in a register.

## Register Variables (2)

- The compiler does not have to follow this hint.

If it prefers, it can simply ignore all “`register`” declarations (except for “`&`”, see below). Registers are a very scarce resource, and on some machines registers are additionally bound by their specialized function. It makes little sense to declare more than about three register variables per function. Note that the calling convention is not changed by declaring parameters as “`register`” (they are still passed on the stack).

- If one declares a variable as “`register`”, one cannot apply the address operator “`&`” to it.

Registers have no memory address, and if the compiler really puts the variable in a register, it will not be allocated on the stack. But even if the compiler decides not to put it in a register, “`&`” is illegal.

# Block Structure (1)

- Variables can be declared at the beginning of every block (i.e. after each “{” ), not only at the beginning of a procedure.
- The variables can be used after the declaration until the end of the block (the corresponding “}” ).
- The compiler may reuse the space of the variable once the control leaves the block in which it is declared.

Many compilers will not manipulate the stackpointer when a subordinate block (still within the same procedure) is left, but they will put variables declared in disjoint blocks at the same memory location.

## Block Structure (2)

- It is legal (but bad style) to declare a variable with the same name as a variable in an outer block.

Only at the beginning of the same block, it is forbidden to declare two variables with the same name.

- Then the variable declared in the outer block becomes inaccessible in the inner block (it is “shadowed” by the new declaration).

So the scope of the declaration in the outer block has a hole.

## Block Structure (3)

Exercise: What does this program print?

```
(1)  #include <stdio.h>
(2)  int main()
(3)  {
(4)      int i;
(5)      i = 1;
(6)      if(i < 10) {
(7)          int i;
(8)          i = 10;
(9)      }
(10)     printf("i = %d\n", i);
(11)     return 0;
(12) }
```

# Block Structure (4)

## Style Hints:

- My personal style is to declare variables only at the beginning of a procedure.

Then one can more easily find the declaration of a variable: One has to look for it only in one place.

- The main reason for declaring variables in the innermost possible block is that they cannot be accessed at unexpected positions in the program.

The programmer knows where the variable is needed, and the programming language should allow to specify that, so that the compiler can detect any violations.

## Block Structure (5)

- However, procedures should be small in C (normally not more than 20–70 lines).

So all statements that could access a local variable fit on a single screen or at least a single piece of paper. One does not gain too much safety by further reducing the scope of a variable.

- In C, it is not possible to define procedures within other procedures.

This would make blocks much larger and then it makes sense to use various degrees of local/global.

## Block Structure (6)

- When variables are declared in inner blocks, it might be possible to immediately initialize them, whereas the initial value might not yet be known at the beginning of the procedure.

I do accept that reason. C++ permits to freely mix declarations and statements in a block for precisely this reason.

- In the end it is a matter of personal taste: One has to choose between two things that are both bad.

My style makes the organisation of the program clearer by reducing the possibilities (the chaos), the other style (declaring variables only at the point where they are needed) is somehow safer.

# Global Variables (1)

- Global variables are variables that are declared outside of functions:

```
(1)  #include <stdio.h>
(2)  int global = 1;
(3)
(4)  void f(void) { global++; }
(5)
(6)  int main()
(7)  {   f();
(8)      printf("%d\n", global);
(9)      return 0;
(10) }
```

## Global Variables (2)

- Global variables can be accessed in all functions (after the declaration).
- Global variables exist for the entire duration of the program execution.

In contrast to local variables, they are not allocated on the stack. They have a fixed address that already the compiler/linker knows.

- Therefore, global variables keep their value between function calls.

When a function returns, all its local variables are destroyed. If it is called later again, new variables with the same names are created.

## Global Variables (3)

- Global variables are automatically initialized to 0 (but it might be better style to write an explicit initialization if the value is important).

In contrast, without initialization, local variables contain garbage. If one writes an initialization for a local variable, the generated machine code will assign a value at runtime. Initializations for global variables are usually done at compile-time: The executable program contains a data section that is loaded into main memory together with the instructions. The global variables simply have an address within this data section (at least the ones that do not have the initial value 0). Most operating systems initialize memory to 0 before they give it to a program/process (in order to protect the confidential data of the previous user). But C guarantees the initialization of global variables on any operating system.

## Global Variables (4)

- It is possible to declare a local variable with the same name as a global variable. These are two distinct variables that have the same name.
- The global variable is not accessible in the function that declared a variable with the same name, since there this name refers to the local variable.

The global variable becomes “shadowed” in the function. This might indicate an error (a variable was promoted from local to global, one forgot to remove the declaration of the local variable). In any case it is bad style. Some compilers give warnings.

## Style Hints (1)

- Using global variables reduces the number of parameters that must be passed to functions (especially the number of “reference” parameters).
- If a variable  $x$  must keep its value between calls to a function  $f$ , and only local variables can be used,
  - ◇ it must be declared in a function that remains active for the entire duration  $x$  is needed (e.g. `main`).
  - ◇ Then it is passed downwards as a reference parameter, via many intermediate procedures to  $f$ .
  - ◇ But  $x$  might be meaningless for these procedures.

## Style Hints (2)

- Changes to global variables are side effects that are not obvious when a procedure is called.
- For understanding a global variable, one must know all points in the program that read or write it.

And also in the right sequence, i.e. the sequence of execution.

- Using global variables too much makes a program difficult to understand and difficult to modify.

## Style Hints (3)

- Global variables must have meaningful names.

Names like `i`, `p` are only acceptable for local variables that are used in no more than 10 consecutive lines.

- There should be a comment/documentation for
  - ◇ every global variable that explains its meaning (maybe also references functions that access it),
  - ◇ every function that explains what the function does, when it may be called, what the parameters mean, and which global variables are accessed.

# Static Variables (1)

- Large C programs consist of several “C” source files that can be separately compiled.

This is explained in the next chapter in more detail.

- Global variables can be accessed in all source files: They must be declared in each file, but there is only one copy of the variable.

Only one of the declarations may contain an initialization.

- Such “very global” variables are seldom needed.

Normally, only a small set of functions accesses a global variable, and these functions are defined together with the global variable in the same source file.

## Static Variables (2)

- One can prefix the declaration of a global variable with “`static`” to make it slightly less global:

```
static int not_quite_global = 1;
```

- Then it can be accessed only by functions defined in the same source file.

If another source file should declare a static variable with the same name, there are two distinct variables.

- Otherwise they behave like global variables.

Static variables exist for the entire program execution (they are not allocated on the stack), keep their value between function calls, and are automatically initialized with 0.

## Static Variables (3)

- The different C source files of the program can be understood as different modules.
- Static variables are then variables that are local to the module.

In this way one knows at least that the only functions that can access the variable must be in the same source file. A single source file should normally not be longer than 500-1000 lines.

- Using really global variables (i.e. variables declared outside of functions without “`static`”) is questionable style and should be avoided.

# Local Static Variables (1)

- It is also possible to add the keyword “`static`” to declarations of local variables (i.e. variable declarations in functions).
- Then the variable will live for the entire duration of program execution.

It is not allocated on the stack, i.e. it is not destroyed when the procedure returns. It has a fixed address.

- The variable also keeps its value between function calls.

## Local Static Variables (2)

- Even if the function calls itself recursively, there will be only a single copy of this variable.
- Like a local variable, it can be accessed only in the function it is declared.
- Local variables that are not static (created on the stack) are called “automatic” in C.

One can explicitly declare local variables in functions as “`auto`”, but since this is the default, it is never done.

## Local Static Variables (3)

- In old-style C (not ANSI C), there were some restrictions for initializing automatic variables: Arrays and structures could only be initialized if they were global or static.

```
(1) static const char *weekdays[] = {  
(2)     "Monday", "Tuesday", "Wednesday",  
(3)     "Thursday", "Friday", "Saturday",  
(4)     "Sunday"  
(5) };
```

- It still might be slightly more efficient this way.

# Static Functions (1)

- Functions are by default also global: They can be called from every function in the same program, including functions in other modules (source files).
- If one prefixes a function definition with “`static`”, the function is visible only in the source file in which it is defined:

```
(1)  static void inc(int *p)
(2)  {
(3)      (*p)++;
(4)  }
```

## Static Functions (2)

- One should think about the external interface of every module: What are the “exported” functions, types, and (if really necessary) global variables?

Exporting global variables is never necessary.

- The interface should be as simple as possible, and should not contain unnecessary functions.

One might later want to exchange the implementation, but keep the interface. The less one has guaranteed, the more freedom one has.

- Auxillary functions that are needed only internally within the module should be declared “**static**”.

# Extern Declarations (1)

- For global variables, there is also a difference between definition (which reserves storage) and declaration (which only makes the name and type known).

Only here the two words are distinguished. Otherwise it is common to always say “declaration”, even when it is really a “definition”.

- If the same variable is used in several source files (modules), exactly one of these must contain a definition, the other must contain declarations.

## Extern Declarations (2)

- Declarations of global variables are marked by the keyword “extern”:

```
extern int global;
```

- Of course, such a declaration cannot contain an initialization.
- It is legal to have a declaration and a definition for the same variable in the same translation unit.
- C permits any number of declarations for the same name if the declarations agree in the type.

## Extern Declarations (3)

- If a global variable is declared without “**extern**” and without initialization, the C compiler treats this as a “tentative definition” .
- It is possible to have several such declarations for the same name (if they declare the same type), so it behaves like a declaration.

In contrast, there can only be one definition: It is not possible to declare the same name twice with an initialization, even if the initial values agree.

## Extern Declarations (4)

- But if at the end of the translation unit (module) the compiler still has not seen a definition with an initialization, it treats the tentative definition(s) as a single definition with initialization 0.
- That means that another module cannot contain a tentative definition: All other modules must use “**external**”.

Some linkers on some operating systems relax this rule: They treat all tentative definitions in all modules together and produce one definition for them (e.g. common under UNIX).

## Extern Declarations (5)

Example: This program is legal (but bad style):

```
(1)  #include <stdio.h>
(2)
(3)  extern int i;
(4)  int i = 1;
(5)  int i;
(6)  extern int i;
(7)  int i;
(8)
(9)  int main()
(10) { printf("%d\n", i);
(11)     return 0;
(12) }
```

## Extern Declarations (6)

- Functions can also be declared as “extern”:

```
extern void push(int n);
```

- However, here the keyword “extern” is simply ignored: The difference between declaration and definition is clear from the presence of a function body.
- One can declare (but not define) functions inside other functions. Some style guides suggest that there one should use the word “extern”.

## Extern Declarations (7)

- Names have “**external linkage**” if they are made known to the linker i.e. they are exported from the module.

The linker combines several modules and libraries to an executable program. Note that “external linkage” does not necessarily mean that “**extern**” is used: Definitions of global variables cannot use “**extern**” but they define objects with external linkage.

- In contrast, names that are declared with “**static**” have “**internal linkage**”.

## Extern Declarations (8)

- Variables have “static extent” if they live for the entire program execution.

This includes variables declared outside functions, even variables declared with “extern”.

- In contrast, variables that are declared inside functions (without `static`) have “local extent” (or automatic extent).

# Namespaces (1)

- We have seen that the same identifier (name) can be used in different functions, and the same identifier can be used as global variable and as local variable (even at different block nesting levels).
- But at each point in the program, an identifier had only one meaning.
- However, the same name can be used for different purposes, even in the same scope, because C has several different “namespaces”.

## Namespaces (2)

- There is a different namespace for each structure or union.
- Thus, one can declare several structures/unions that have components/fields with the same name.

The components can be of different type and can have different offsets with respect to the beginning of the structure in memory. In old style C all components of all structures were in a single namespace.

- The names of structure/union components also do not conflict with variable or function names.

The compiler knows from “.” or “->” that a structure/union component name follows.

## Namespaces (3)

- In the same way, structure, union, and enumeration names (“tags” used after `struct`, `union`, `enum`) generate no conflicts with variables etc.

However, there is only a single namespace for the three types of tags. This is actually a restriction introduced by ANSI C, in old-style C structure and union tags had different namespaces.

- Variables, enumeration constants, functions, and type names all share the same namespace.

E.g. there cannot be a global variable and a function with the same name.

- Labels have their distinct namespace.

# Overview

1. Procedures/Functions: Call and Return
2. Procedures/Functions: Parameters
3. Global and Local Variables
4. Declaration Syntax

# Variable Declarations (1)

- A variable declaration in C consists of two parts: The “declaration specifiers” on the left side and the “declarator” on the right side.
- Basically, the declaration specifiers define the type of the variable, and the declarator defines the name.
- However, the “declarator” (right side) may modify the type on the left side by making the variable a “pointer to” or “array of” the type on the left side.

Also a pointer to a function that returns the type on the left side is possible.

## Variable Declarations (2)

- One declaration can contain several declarators.

The declarators are separated by commas “,”.

- This is an abbreviation to declare multiple variables of a similar type, e.g.

```
long int n, *p, **q, a[10], *b[5], (*c)[3];
```

- “long int” are the declaration specifiers.

I.e. “long int” is the “basic type” of the declaration.

- `n` has type `long int`.
- `p` is a pointer to `long int`.

# Variable Declarations (3)

```
long int n, *p, **q, a[10], *b[5], (*c)[3];
```

- `q` is a pointer to a pointer to `long int`.

Thus `q` could contain the address of `p`.

- `a` is an array that has space for 10 “`long int`” values.
- `b` is an array of 5 pointers to “`long int`”.
- `c` is a pointer to an array of three “`long int`” values.

The parentheses show that one first has to dereference `c` and then one can do the array access. It is also possible to use parentheses for “`b`”: “`*(b[5])`”. However, since arrays of pointers are much more common than pointers to arrays, the first needs no parentheses.

# Variable Declarations (4)

- Variables can be initialized in the declaration:

```
long int n = 5 * 10, *p = 0, q = &p,  
        a[10] = { 0,1,2,3,4,5,6,7,8,9 },  
        *b[5] = { &n, &n, &n, &n, &n };
```

- Arrays (and structures) are initialized by listing all values in braces ({...}).
- It is legal to specify fewer values than the size of the array, but it is forbidden to specify more values.

If fewer values are specified, the remaining values will be 0 for global and static variables and undefined (garbage) for local (automatic) variables.

# Formal Syntax (1)

- A declaration consists of a sequence of declaration specifiers followed by a sequence of declarators (possibly with initializations):

declaration  $\rightarrow$  decl-spec-list ;

declaration  $\rightarrow$  decl-spec-list init-decl-list ;

- The declarators may only be missing if the declaration specifiers define a structure, union, or enumeration (which must be named with a “tag”).
- Declarations cannot be empty.

## Formal Syntax (2)

- An “init-decl-list” is a sequence of init-declarators separated by commas:

init-decl-list  $\rightarrow$  init-decl

init-decl-list  $\rightarrow$  init-decl , init-decl-list

- An “init-declarator” is a declarator possibly followed by an initializer:

init-decl  $\rightarrow$  declarator

init-decl  $\rightarrow$  declarator = initializer

## Formal Syntax (3)

- A “decl-spec-list” is a non-empty sequence of declaration specifiers:

decl-spec-list → decl-specifier

decl-spec-list → decl-specifier decl-spec-list

- A declaration specifier is a storage class specifier, a type specifier, or a type qualifier:

decl-specifier → storage-spec

decl-specifier → type-spec

decl-specifier → type-qualifier

## Formal Syntax (4)

- Storage class specifiers are the following:

storage-spec → auto

storage-spec → register

storage-spec → static

storage-spec → extern

storage-spec → typedef

- A declaration can contain at most one storage class specifier.

## Formal Syntax (5)

- If a declaration inside a function contains no storage class specifier, the following defaults apply:
  - ◇ Variables have storage class `auto`.
  - ◇ Functions have storage class `extern`.
- If a variable declaration outside a function contains no storage class specifier, it has external linkage and static extent.

This is not equivalent to `extern`: Only without storage class specification, one can reserve storage for a global variable (see above).

## Formal Syntax (6)

- If a function declaration contains no storage class specifier, it has external linkage.

If one writes “`extern`”, it has the same effect. But probably one should use “`extern`” only for declarations (without body), not for definitions.

- There are two type qualifiers:

type-qualifier → `const`

type-qualifier → `volatile`

“`volatile`” means that the value can change outside the control of this program, e.g. for a pointer to a device register (for memory mapped I/O). `const` and `volatile` together means that this program does not change the storage location, but it may be changed from the outside.

# Formal Syntax (7)

- The following possibilities exist for type specifiers:

type-spec → predefined-type-spec

type-spec → struct-union-spec

type-spec → enum-spec

type-spec → `identifier`

- The identifier must be a declared `typedef`-name.
- `predefined-type-spec`: One of `char`, `short`, `long`, `signed`, `unsigned`, `int`, `float`, `double`, `void`.

## Formal Syntax (8)

- A declaration may contain only one type specifier with the following exceptions:
  - ◇ At most one of `short` and `long` may appear together with at most one of `signed` and `unsigned` and/or together with `int`.
  - ◇ At most one of `signed` and `unsigned` may appear together with `char`.
- If a declaration contains no type specifier (or only `short`, `long`, `signed`, `unsigned`), `int` is assumed.

## Formal Syntax (9)

- C permits that declaration specifiers are written in any sequence, e.g.

```
long const static int unsigned x = 5;
```

- There must be at least one declaration specifier, but it does not have to be a type specifier:

```
static y; /* int is assumed by default */
```

- One should avoid such formulations: The goal is not to show how well you know the syntax of the language, but to write understandable programs.

# Formal Syntax (10)

- The type specifier can declare a structure or union:

`struct-union-spec` → `struct` `identifier`

`{` `mem-decl-list` `}`

`struct-union-spec` → `struct`

`{` `mem-decl-list` `}`

`struct-union-spec` → `struct` `identifier`

- The same three rules exist also with `union` instead of `struct`.

# Formal Syntax (11)

- A “mem-decl-list” is simply a non-empty list of member (component) declarations:

mem-decl-list  $\rightarrow$  mem-decl

mem-decl-list  $\rightarrow$  mem-decl mem-decl-list

- Like a variable declaration, a member declaration consist of specifiers and declarators and ends in a semicolon:

mem-decl  $\rightarrow$  m-spec-list m-decl-list ;

- Initializations are not allowed here.

## Formal Syntax (12)

- “m-spec-list” is a list of declaration specifiers:

m-spec-list → m-specifier

m-spec-list → m-specifier m-spec-list

- Specifiers are the same as in variable declarations, only a storage class cannot be specified for the single structure members:

m-specifier → type-specifier

m-specifier → type-qualifier

# Formal Syntax (13)

- “m-decl-list” is a list of declarators:

m-decl-list → m-declarator

m-decl-list → m-declarator m-decl-list

- Declarators are the same as in variable declarations:

m-declarator → declarator

m-declarator → declarator : const-expr

m-declarator → : const-expr

The second and third rule is used to define bit-fields: This is for components that need only a small number of bits (“packed structures”).

# Formal Syntax (14)

- Finally, a type specifier can declare an enumeration type. The general syntax is the same as for structures, but the list inside the “{...}” is different:

`enum-spec` → `enum` `identifier`  
`{ enum-list }`

`enum-spec` → `enum`  
`{ enum-list }`

`enum-spec` → `enum` `identifier`

# Formal Syntax (15)

- An “enum-list” is a non-empty list of enumeration constant declarations (separated by commas):

enum-list → enumerator

enum-list → enum-list , enumerator

- An enumeration constant declaration consists of an identifier that is optionally equated to an integer constant:

enumerator → identifier

enumerator → identifier = const-expr

# Formal Syntax (16)

- Declarators are the right side of declarations.

They contain the declared identifier, but can still modify the type.

- Declarators consist of an (optional) pointer prefix and a “direct declarator”:

declarator  $\rightarrow$  ptr-prefix direct-decl

- The pointer prefix is a sequence of 0 or more “\*” which may be followed by type qualifiers:

ptr-prefix  $\rightarrow$   $\epsilon$

ptr-prefix  $\rightarrow$  ptr-prefix \* qual-list

## Formal Syntax (17)

- “qual-list” is a possibly empty sequence of type qualifiers (`const` or `volatile`):

`qual-list`  $\rightarrow \epsilon$

`qual-list`  $\rightarrow$  `type-qualifier qual-list`

- E.g. `const char *p`: pointer to characters that could be stored in read-only memory.
- E.g. `char *const p`: constant pointer to characters (the characters can be modified, the pointer not).

# Formal Syntax (18)

- A direct declarator has one of the following forms:

`direct-decl`  $\rightarrow$  `identifier`  
`direct-decl`  $\rightarrow$  `( declarator )`  
`direct-decl`  $\rightarrow$  `direct-decl [ const-expr ]`  
`direct-decl`  $\rightarrow$  `direct-decl [ ]`  
`direct-decl`  $\rightarrow$  `direct-decl ( par-spec )`  
`direct-decl`  $\rightarrow$  `direct-decl ( id-list )`  
`direct-decl`  $\rightarrow$  `direct-decl ( )`

# Formal Syntax (19)

- The parameter specification consists of a parameter list that can be followed by “, ...” to indicate that this procedure has variable argument lists:

par-spec → par-list

par-spec → par-list , ...

- A parameter list is a non-empty list of parameter declarations (separated by commas):

par-list → par-decl

par-list → par-list , par-decl

# Formal Syntax (20)

- A parameter declaration is similar to a variable declaration without initialization:

par-decl → decl-spec-list declarator

par-decl → decl-spec-list abstract-decl

- In a function declaration (without body), the parameter names are not required: This is treated in the second rule.

Abstract declarators are explained below.

- Function definitions require parameter names.

# Formal Syntax (21)

- The only storage class permitted in parameter declarations is “`register`”.
- An ANSI C parameter list is never empty: Functions without parameters have the parameter list `void`.
- Identifier lists are used in old style function definitions, empty lists in old style function declarations:  
( ) is also used in old style definitions of parameterless functions.

`id-list` → `identifier`

`id-list` → `id-list` , `identifier`

## Formal Syntax (22)

- Abstract declarators are declarators that omit the identifier (the name of the variable or parameter).

But there is a unique position at which the identifier would be placed: One can get an abstract declarator from a declarator simply by deleting the variable name.

- They are used when types must be specified without declaring a variable/parameter:
  - ◇ In type casts, e.g. `(int *) p`.
  - ◇ In `sizeof`-expressions, e.g. `sizeof(int [5])`.
  - ◇ In function declarations if one does not want to specify parameter names.

## Formal Syntax (23)

- Like declarators, abstract declarators consist of an (optional) pointer prefix and a direct abstract declarator:

**abstract-decl** → **ptr-prefix direct-ad**

- Only non-empty abstract declarators may be put in parentheses (usual declarators are never empty).

This avoids the confusion with a function call. The official grammar (from the K&R book) handles this problem in a more elegant way than the equivalent grammar shown here. However, with the grammar shown here, it is more obvious that abstract declarators result from normal declarators by deleting the identifier.

# Formal Syntax (24)

- $\text{direct-ad} \rightarrow \epsilon$
- $\text{direct-ad} \rightarrow ( \text{nonempty-ad} )$
- $\text{direct-ad} \rightarrow \text{direct-ad} [ \text{const-expr} ]$
- $\text{direct-ad} \rightarrow \text{direct-ad} [ ]$
- $\text{direct-ad} \rightarrow \text{direct-ad} ( \text{par-spec} )$
- $\text{direct-ad} \rightarrow \text{direct-ad} ( )$

Note that the rules are the same as for direct declarators, but generate  $\epsilon$  (the empty word) instead of an identifier. They are also not used in old style function definitions, therefore the rule with id-list misses.

# Formal Syntax (25)

- nonempty-ad  $\rightarrow$  \* qual-list ptr-prefix direct-ad
- nonempty-ad  $\rightarrow$  nonempty-dad
- nonempty-dad  $\rightarrow$  ( nonempty-ad )
- nonempty-dad  $\rightarrow$  direct-ad [ const-expr ]
- nonempty-dad  $\rightarrow$  direct-ad [ ]
- nonempty-dad  $\rightarrow$  direct-ad ( par-spec )
- nonempty-dad  $\rightarrow$  direct-ad ( )

## Formal Syntax (26)

- A type name is used as argument to `sizeof` and in a type cast:

`type-name` → `m-spec-list abstract-decl`

`m-spec-list` is a list of type specifiers and type qualifiers as in structure member declarations (see above).

- The `sizeof` operator may not be applied to a function type or an incomplete type.

`sizeof` may also not be applied to bit fields.

## Formal Syntax (27)

- As explained above, the declarator (that contains the name of the declared variable plus type modifiers), can be followed by “=” and an initializer.
- There are two kinds of initializers, one for simple (unstructured) variables, and one for arrays, structures, and unions:

initializer → assignment

initializer → { init-list }

initializer → { init-list , }

## Formal Syntax (28)

- “assignment” is the most general kind of expression that excludes the comma-operator.
- For global and static variables, the initializer must be a constant expression.
- As a special case, an array of characters can be initialized by a string literal:

```
char s[] = "abc";
```

Note that this would not be a legal assignment, it is permitted only in an initialization. If the array size is not specified (as in the example), the terminating null byte is stored. If it is specified, the null byte is only stored if there is room for it: E.g. `char x[3] = "abc";` is legal.

## Formal Syntax (29)

- An “init-list” (inside the `{...}`) is a list of initializers (separated by commas):

`init-list` → `initializer`

`init-list` → `init-list` `,` `initializer`

- By the third rule for “initializer”, the list can actually end in a comma.

This might give a nicer format for array initializations, when the last element needs no special syntax (all elements are followed by “,”).

- Inside the braces `{...}`, only constant expressions are permitted.

## Formal Syntax (30)

- It is legal to include even the single value for a variable of arithmetic or pointer type in `{...}`.
- When initializing an array, `{...}` is required.
- When initializing a structure, `{...}` is required except the case when the initializer is an expression of structure type.
- When initializing a union, `{...}` can only contain an initialization for the first alternative (the alternative is again to use an expression of union type).

## Formal Syntax (31)

- The brace-enclosed list can never contain more initializers than are required for the structure or array, but it can contain less.

The remaining members are filled with 0 (for global/static variables).

- If one has an array of structures or another nested aggregation, one can use nested initializers:

```
struct s { int i, char c };  
struct s x[2] = {{1, 'a'}, {2, 'b'}};
```

- One can also use a flat list:

```
struct s x[2] = { 1, 'a', 2, 'b'};
```

## Formal Syntax (32)

- A function definition (that includes a body) has the following syntax:

fun-def → opt-d-specs declarator  
block

fun-def → opt-d-specs declarator  
decl-list block

- The first is the new style (ANSI C) form, the second the old style form.

## Formal Syntax (33)

- A function may return an arithmetic type, a pointer, a structure, a union, or `void`, but it may not return a function or an array.
- The declarator must explicitly specify that the declared identifier has function type (not via `typedef`). I.e. for new-style definitions it must contain

direct-decl ( par-spec )

where “direct-decl” is an identifier (possibly in parentheses).

# Formal Syntax (34)

- For old style function definitions, the declarator must contain one of:

direct-decl ( id-list )

direct-decl ( )

- For old-style declarations, the parameters are declared between the function head and the block:

decl-list  $\rightarrow \epsilon$

decl-list  $\rightarrow$  decl-list declaration

Only parameters may be declared, initialization is not permitted, the only storage class is `register`. Undeclared parameters have type `int`.

## Formal Syntax (35)

- “opt-d-specs” is a list of declaration specifiers. It is used to specify the return type of a function:

`opt-d-specs`  $\rightarrow$   $\epsilon$

`opt-d-specs`  $\rightarrow$  `opt-d-specs decl-specifier`

- The only storage classes permitted are `extern` and `static`.
- If the list of declaration specifiers is empty, `int` is assumed.

# Formal Syntax (36)

- Finally, a translation unit (the start symbol of the grammar) is a list of external declarations:

translation-unit  $\rightarrow$  external-decl

translation-unit  $\rightarrow$  translation-unit  
external-decl

- An external declaration is a function definition or a declaration:

external-decl  $\rightarrow$  fun-def

external-decl  $\rightarrow$  declaration

# Type Declarations (1)

- It is possible to define names for types.
- A type declaration looks exactly like a variable declaration, only the “storage class” `typedef` is used:

```
typedef int *int_ptr;
```

- Afterwards the declared name can be used like one of the built-in type names:

```
int_ptr p;
```

- The variable `p` now has the type `int *`.

## Type Declarations (2)

- “`typedef`” does not introduce new types, only abbreviations/synonyms for types that could also be specified directly.
- E.g. after

```
typedef int day_t, month_t, year_t;
day_t d; month_t m; year_t y; int i;
```

the variables `d`, `m`, `y`, and `i` have the same type: C will not complain about assignments between them.
- This differs from declaring structures and unions, where each declaration introduces a new type.

# Type Declarations (3)

- In order to decide whether two types are the same (“type equivalence”),
  - ◇ `typedef` names are replaced by their definition.

In contrast, for structures, unions, and enumerations, C only compares their tags (for a declaration without tag, C automatically assigns a unique tag).
  - ◇ parameter names of function types are removed,  
So they do not have to match.
  - ◇ for each list of declaration specifiers, only the set of type specifiers must be the same.

With the implicit `int` and `signed` (for `int`) added if necessary.  
The specific sequence of type specifiers is not important.

# Example: Linked List (1)

- Example (declaration for a linked list):

```
typedef struct list_s {  
    int          data;  
    struct list_s *next;  
} *list_t;
```

When the structure member/component “`next`” is declared, the type “`struct list_s`” is still incomplete. However, one can declare pointers to such types.

- If one declares

```
list_t p;
```

the variable `p` has the type `struct list_s *`.

## Example: Linked List (2)

- One can allocate memory for a list element with

```
p = (list_t) malloc(sizeof(struct list_s));
```
- The standard library function `malloc` (“memory allocation”) is declared in `stdlib.h` as

```
void *malloc(size_t number_of_bytes);
```
- The type `size_t` is declared in “`stddef.h`”, e.g. as

```
typedef unsigned int size_t;
```
- `malloc` returns 0 if there was not enough memory.
- Memory allocated with `malloc` is not initialized.

## Example: Linked List (3)

- If the list element is no longer used, one can give the memory back with `free(p);`

- `free` is declared in `stdlib.h` as

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

It is essential that “`ptr`” is a value previously returned from “`malloc`”.

- When a program ends, all memory allocated to this process is automatically returned to the OS.

On old/bad operating systems, this might not be the case, especially when the program terminates abnormally. Also `free` itself may not give the memory back to the OS, but keep it for calls to `malloc`.

# Function-Type Variables (1)

- One can declare variables, parameters, and structure/union members of type “pointer to function”.
- E.g. the variable `f` contains a pointer to a function with result type `int` and two `int` parameters:

```
int (*f)(int n, int m);
```

- The parentheses are needed: The following is simply a (forward) declaration for a function `f` with two `int`-parameters that returns a pointer to `int`:

```
int *f(int n, int m);
```

## Function-Type Variables (2)

- Suppose one has declared a function `max` of the appropriate type:

```
int max(int n, int m);
```

- Then the address of `max` can be assigned to `f`:

```
f = &max;
```

This is the start address of the machine instructions for the body.

- As for arrays, C automatically takes the address of a function name, so the following is equivalent:

```
f = max;
```

## Function-Type Variables (3)

- One can call the function currently stored in the variable `f` by first dereferencing the pointer:

```
i = (*f)(3, 10);
```

In this way, declaration and usage of `f` look very similar.

- The ANSI standard (“new style C”) also permits a call without explicit dereferencing:

```
i = f(3, 10);
```

This is equivalent to the call above. It was not legal in old-style C.

## Function-Type Variables (4)

- If one declares a function parameter of function type, it is implicitly changed to “pointer to function”.
- E.g. the following are equivalent:

```
void sort(char *input[],  
          int comp(char *s1, char *s2));
```

```
void sort(char **input,  
          int (*comp)(char *, char *));
```

This also shows that an array parameter is implicitly changed to a pointer. The names of parameters are not important when specifying function types and can be left out. In any case, they immediately go out of scope: There is no way to access `s1` and `s2` in the first example.

# Exercise

What do these declarations mean?

- `int (a);`
- `int *b(int);`
- `int c(int (*)(int), int);`
- `int (*d)(int *);`
- `int *(*e)(int);`
- `int (*f[10])();`
- `int (*g(int))(int);`