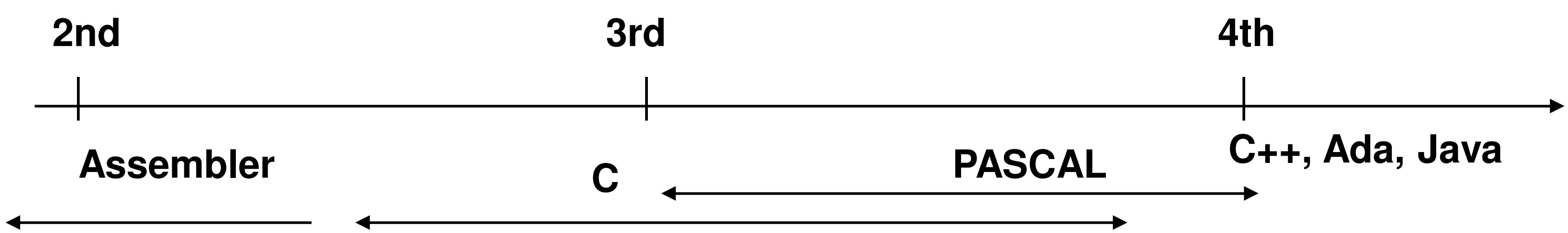
Preliminary

- C Language Characteristics
- Lexical convention
- Meaning of Identifiers
- Structure of a C Program

Preliminary

C Language Characteristics

- General Application purpose
- Designed by and for programmers
- Universal Language
- Quite efficient compared to other 3rd generation languages (Pascal)



C Language Characteristics

- C gives good support for the high-speed, low-level, input/output operations, which are essential to many automotive embedded systems
- Increased complexity of applications makes the use of a high-level language more appropriate than assembly language.
- C can generate smaller and less RAM-intensive code than many other high-level languages.

Lexical convention

- ***Tokens**:* there are six types of tokens:
  - *identifiers*
  - *keywords*
  - *constants*
  - *string literals*
  - *operators*
  - blanks, horizontal and vertical tabs, new lines and comments (collectively named, “*white spaces*”) are ignored, except as they separate tokens

Lexical convention

- ***Identifiers:***
  - An identifier is a sequence of letters and digits.
  - The first character must be a letter.
  - The underscore _ counts as a letter.
  - Upper and lower case letter are different.
  - Identifiers may have any length , and for internal identifiers (identifiers that don’t have *external linkage)*, at least the first 31 characters are significant.
  - Identifiers with *external linkage* are more restricted.

* *external linkage* will be discussed later.

Preliminary

Lexical convention

- ***Keywords***: the following identifiers are reserved for used as keywords, and may not be use otherwise:

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

- Some implementation also reserved the word **asm**

Preliminary

Lexical convention

- ***Constants:***

constant:

integer-constant

character-constant

floating-constant

enumeration-constant

Meaning of Identifiers

- ***Identifiers*** *or* ***names*** refer to a variety of things: functions, structures, unions and enumeration members of structures or unions, enumeration constants, typedef names and objects.

- *An* ***object***, sometimes called a ***variable***, is a location in storage and its interpretation depends on two main attributes:
  - *storage class*
  - *type*

Meaning of Identifiers

- The ***storage class*** determines the lifetime of the storage associated with the identified object
- The ***type*** determines the meaning of the value found in the identified object
- A ***name*** also has a:
  - ***scope***, which is the region of the program in which it is known
  - and a ***linkage*** which determines whether the same name in another scope refers to the same object of function.

Preliminary

Meaning of Identifiers

- ***Storage Class:***
  - ***automatic***:
    - objects are local to a block and are discarded on exit from the block
    - declaration within a block create automatic objects if no storage class specification is mentioned, or if the *auto* specifier is used.
  - ***static****:* objects may be local to a block or external to all blocks, but in either case retain their values across exit from and reentry to functions and blocks.

Preliminary

Meaning of Identifiers

- Storage class specifiers:
  - ***auto***
  - ***register***
  - ***static***
  - ***extern***

- The ***auto*** and ***register*** specifiers
  - give the declared objects *automatic storage class,* and may be used only within functions.
  - such declaration also serve as definitions and cause storage to be reserved.

Preliminary

Meaning of Identifiers

- The ***static*** specifier:
  - gives the declared object *static storage class*
  - may be used either inside or outside functions
  - inside a function, this specifier causes storage to be allocated, and serves as a definition;
  - outside a function this specifier gives *internal linkage*

- A declaration with ***extern,*** used inside a function, specifies that the storage of the declared objects is defined elsewhere.

Preliminary

Meaning of Identifiers

- Type specifiers:

void	**float**	**struct-or-union specifier**
char	**double**	**enum specifier**
short	**signed**	**typedef-name**
int	**unsigned**	
long		

- Type *qualifier*:
  - **const**
  - **volatile**

Preliminary

Meaning of Identifiers

- The *qualifier* may appear with any type *specifiers*.
- A ***const*** object may be initialized, but not thereafter assigned to.
- There are no implementation-independent semantics for ***volatile*** objects.
- The purpose of ***const*** is to announce objects that may be placed in read-only memory, and perhaps to increase opportunities for optimization.
- The purpose of ***volatile*** is to force an implementation to suppress optimization that could otherwise occur.

Preliminary

Structure of a C program

Main File

Data Declarations

Main
- Declarations
- Instructions
- Blocks

Sub program
- Declarations
- Instructions
- Blocks

Called Files

Called Sub-programs

Used Data

Preliminary

Structure of a C program

- Program

  A C program consists of a sequence of functions which mostly are placed in different modules

- Module

  A module is a group of functions and data . Large program can be divided into a collection of separately compiled modules. They can be placed in a library for use by different programs.

Preliminary

Structure of a C program

- Function
  - A function consists of a function header and a function block
  - The function header contains the function name and the function parameters.
  - The function block can contain a definition part and a statement part.
- Block structure
  - The curly brackets { and} are used for combining definitions and statements to one block.
  - Definitions and statements are terminated by a “;”.
  - A single “;” represents a statement
    In a function block further block can be nested.

Numbers, Variable, Data Types

- Numbers
- Variables
- Memory class modifier
- External variables
- Scope rules
- Static variables
- Register variables
- Initialization

- Data Types
- Data encapsulation with *struct*
- Data encapsulation with *union*
- Bitfields
- Enumerated data types
- Definition of private types
- Type Conversion

Numbers, Variable, Data Types

Two's complement

- Nonnegative integers from 0 to (2^{k-1} - 1) are representing in binary the same way as with sign-magnitude, 0 followed by a (k-1) – bit absolute value.
- Negative integers -(2^{k-1}) to -1 are represented by adding 2^k and expressing the result in binary.

- Algorithm for Negating a Number
  - a) Complement (flip) each bit.
  - b) Add 1
- Complementing a k-bit number is same as subtracting it from 2^{k-1}.

Example: k=6 ; 2^k=64

-2 → 64-2=62=111110

-13 → 64-13=51=110011

Numbers, Variable, Data Types

Numbers and constants

- The C-compiler differentiates between two types constants:
  - Text constants
  - Numerical constants

- Text constants:
  - char Ex: ‘A’, ‘E’, ‘9’, ‘#’
  - string Ex: “Continental”

- For string constants the compiler install an array of appropriate length and stores the character sequence and automatically adds ‘\0’ to the end of the string constant. Thus the string end can be recognized by the program.
- A char doesn’t end with ‘\0’

Numbers, Variable, Data Types

Numbers and constants

- Numbers always start with a digit.
- The base is characterized with
  - **0x** nnnn hexadecimal number (digits: 0-9, a-f or A-F)
  - **0** nnn octal number (digits: 0-7)

  Any other number is supposed to be decimal.

- Long integer constants are specified with an ending **L,** e.g.

  long a = 10L;

- Unsigned integer constants are specified with an ending **u,** e.g.

  unsigned int c = 3u;

- Floating point values should contain a decimal point, e.g.

  float b = 1.;

  and may be written with mantissa and exponent, e.g.

  float c = 0.5e-3;

Numbers, Variable, Data Types

Numbers and constants

Symbolic definition of Constants

- In general, it is recommended to define a symbol for all constants to be used in definitions. Thus the programs can be read and changed more easily.
- Symbolic constants are defined with the pre-processor directive **#define**

Ex:

```
#define MAX 100
#define PI 3.14
#define TEXT “Programming in C”
```

Numbers, Variable, Data Types

Variables

- Variables are characterized by their:
  - value
  - address
  - attribute(modifier)
  - type
  - lifetime

- Declaration of variables:

  <type> <name> ;

  or

  <modifier> <type> <name> ;

  or

  <type> <name1>, <name2>, ... ;

Numbers, Variable, Data Types

Variables

- “C“ knows two attributes that limits the type of accesses to variables:
  const and ***volatile***

volatile *data_type variable_name;*

- A variable defined as *volatile* must always be read from its original location and is not kept in a register at any time.
- Most of the optimizations of the compiler are not available for *volatile* data.
- So, the *volatile* keyword has to be applied only where necessary.

Numbers, Variable, Data Types

Variables

const *data_type variable_name;*

- The *const*-keyword allows only read access to the variable.
- Global *const* variables may only be initialized once in a program. The initialization is performed in the startup code.
- If a local variable is defined as *const* it is read only in the current block while it might be changed anywhere else in the code.
- Constants may be defined:
  - only once in a program
  - as a macro by the preprocessor
  - with the const-keyword as data in memory

Numbers, Variable, Data Types

Variables

- The advantage of a const definition is that this allows the compiler to perform type checking which might be desirable in some cases.
- The advantage of a macro definition is:
  - no memory is needed,
  - more efficient code can be generated,
  - it can be used in switch - case statements.
- Constant pointers treat their indirection as a constant data.

Memory class modifier

[memory_class] [data_type] name;

- There are four memory_class modifiers: *auto, register, extern*, and *static*.
- The modifiers have a different meaning for local and for global objects:

memory_class	local object	global object
`auto`	The object shall be located on the stack. As this is default the `auto` modifier may be omitted.	Meaningless.
`register`	The object shall be located in a register, if possible.	Meaningless.
`extern`	Impossible.	The object is declared and used in the current module but defined in a different one.
`static`	The object shall be located in memory but not on the stack.	The object shall not be public and only accessible in the current module.

Numbers, Variable, Data Types

External variables

- External variables and function have the property that all references to them by the same name, even from functions compiled separately, are references to the same thing. The standard calls this property *external linkage.*
- A variable is "external" if it is defined outside of any function.
- *Automatic* variables are internal to a function.
- "Internal" describes the arguments and variables defined inside functions.

Numbers, Variable, Data Types

Scope rules

- The “scope” of a name is the part of the program within which the name can be used.
- For automatic variables declared at the beginning of a function, the scope is the function in which the name are declared.
- Local variables of the same name in different functions are unrelated.
- The same is true of the parameters of the function, witch are in effect local variables.
- The scope of an external variable or function lasts from the point at which it is declared to the end of the file being compiled.

Scope rules

Remarks:

- It is important to distinguish between the **declaration** of an external variable and its **definition**. A **declaration** announces the properties of a variable (primarily its type); a **definition** also causes storage to be set aside.

- There must be only one **definition** of an external variable among all the files that make up the source program; other files may contain “extern” **declarations** to access it. (There may also be “extern” declarations in the file containing the definition).

Register variables

- A *register* declaration advises the compiler that the variable in question will be heavily used.
- The idea is that ***register*** variables are to be placed in machine registers, which may result in smaller an faster programs. But the compiler are free to ignore the advice.

- The ***register*** declaration can only be applied to automatic variables and to the formal parameters of a function.

Numbers, Variable, Data Types

Initialization

- In the absence of explicit initialization, **external** an **static** variables are guaranteed to be initialized to zero; **automatic** and **register** variables have undefined initial values.

- **Scalar** variables may be initialized when they are defined, by following the name with an equals sign and an expression.

- For **external** and **static** variables, the initializer must be a constant expression; the initialization is done once, conceptually before the program begins execution.

- For **automatic** and **register** variables, the initializer is not restricted to being a constant. It may be any expression involving previously defined values, even function calls.

Numbers, Variable, Data Types

Data Types

- Basic types : char, int, float, double
- The compiler uses the data types for
  - the definition of the range of values,
  - the size of memory,
  - the operations allowed, and
  - the scaling of pointers.

data type	size [in bytes]	range of values
void	undefined	none
signed char unsigned char	1 1	-128 .. +127 0 .. 255
signed short unsigned short	2 2	-32768 .. +32767 0 .. 65535
signed int unsigned int	min. 2 min. 2	compiler dependent
signed long unsigned long	4 4	-2 147 483 648 .. +2 147 483 647 0 .. 4 294 967 295
float	4	+/-1.176e-38 .. +/-3,40e+38
double	8	+/-2,225e-308 .. +/-1,798e+308
pointers	1 .. 4	up to 32 bit addresses

Numbers, Variable, Data Types

Data encapsulation with *struct*

- Often, it is desired to encapsulate several data of different types that belong to the same object. For this “C” offers two keywords to declare private data types: ***struct*** and ***union***.

  struct *[struct_name]*
  {
  data_type1 ivar_list1;
  data_type2 ivar_list2;
  ...
  } *[svar_list];*

- ***struct_name*** is the name the structure is given. If the structure declaration won’t be referenced in the program again, *struct_name* may be omitted. This is only true in connection with an immediate data definition with **svar_list** or in connection with the typedef keyword. The structure is then called anonymous.

Numbers, Variable, Data Types

Data encapsulation with *struct*

- ***data_type*** must be a standard type or a before declared user type. If the current structure is given a *struct_name*, this name is already known. So, for instance pointers to this type of structure might be an element of the structure itself. This is a basis for linked lists.

- ***ivar_list*** is a list of the structure elements of the same type. This list might consist of one element only.

- ***svar_list*** is the possibility to define variables, arrays, or pointers of type *struct struct_name* within the declaration. It is, however, advised to separate data declaration and data definition. In this manner, the declaration might be shared

Numbers, Variable, Data Types

Remarks:

- The access to a member of the structure is performed with the dot operator.
- Usually, data are aligned to even address boundaries. So, if there is a structure declaration

```
struct s1
{
        unsigned char sid;
        unsigned int size;
        unsigned char msg;
};
```

- The compiler may reserve four, six, or even more bytes of memory, depending on the alignment:

Numbers, Variable, Data Types

Remarks:

byte number	byte aligned	word aligned
0	`s1.sid`	`s1.sid`
1	`s1.size` (low byte)	*<hole>*
2	`s1.size` (high byte)	`s1.size` (low byte)
3	`s1.msg`	`s1.size` (high byte)
4		`s1.msg`
5		*<hole>*

- If a structure variable is assigned to another variable of the same type, member by member will be copied.
- Take caution if structure members are pointers because after the assignment there are two pointers that point to the identical location.

Numbers, Variable, Data Types

Data encapsulation with *union*

- A union means an overlay of elements of different data types to the same memory location. The *union* keyword has a similar declaration syntax:

```
union [union_name]
{
    data_type1 ivar_1;
    data_type2 ivar_2;
    ...
} [uvar_list];
```

- **union_name** is the name the union is given. If the union won't be referenced in the program again, **union_name** may be omitted. This is only true in connection with an immediate data definition with **uvar_list** or in connection with the typedef keyword. The union is then called anonymous.

Numbers, Variable, Data Types

Data encapsulation with *union*

- **data_type** must be a standard type or a user type which was declared before. If the current union is given a **union_name**, this name is already known.

- **ivar** is a member element of the union. All member elements start at the same base address of the union. The size of the union is determined by the size of its biggest member element.

- **uvar_list** is the possibility to define variables, arrays or pointers of type **union union_name** within the declaration. It is, however, advised to separate data declaration and data definition.

Numbers, Variable, Data Types

Data encapsulation with *union*

Example :

```
union
{
        unsigned char c[2];
        long              l;
} u1;
```

- The size of u1 is equivalent to the size of long (4 bytes). The lowest byte of u1 may now be accessed by u1.c[0] as well as by u1.l, as can be seen in the memory layout:

byte number	accessed by	accessed by
0	`u1.c[0]`	`u1.l` (lowest byte)
1	`u1.c[1]`	`u1.l`
2		`u1.l`
3		`u1.l` (highest byte)

Numbers, Variable, Data Types

Bitfields

- Bitfields offer the possibility to access single bits or groups of bits in the not bit addressable memory. The order of the bits can be defined with the help of the *struct* keyword:

```
struct [bitfield_name]
{
        data_type1 ivar_1: n_bit_1;
        data_type2 ivar_2: n_bit_2;
        ...
} [bitfield_list];
```

Numbers, Variable, Data Types

Bitfields

- **data_type** must be a standard type or a before declared user type. It is recommended to use unsigned types only.

- **ivar** is the name of a bit or bitgroup which is element of the bitfield.

- **n_bit** is the size of the bitfield element **ivar** in bits. Negative values are forbidden, values with more bits than that of the standard word width of the controller might lead to errors. A value of zero means that the current bitgroup fills up the remaining bits to the next word (=int) boundary.

- **bitfield_list** is the possibility to define variables of type **struct bitfield_name** within the declaration.

Numbers, Variable, Data Types

Remarks:

- Bitfields are used especially in connection with control and status registers of the periphery of microcontrollers.
- The address operator is not available for bitfields. For this reason, neither pointers to bitfields nor arrays of bitfields can be defined.
- ANSI-“C“ does not define anything that has to do with bits. Therefore, using bitfields usually leads to non-portable code because different compilers might use different conventions. For instance:
  – The ordering of the bits is compiler specific. This means that some compilers assign the LSBs to the first bits of the bitfield definition, while others use the MSBs. This is especially true for little-endian and big-endian controllers.

Numbers, Variable, Data Types

Example:

```
   struct TxIC
   {
       unsigned int glvl: 2;
       unsigned int ilvl: 4;
       unsigned int ie: 1;
       unsigned int ir: 1;
       unsigned int : 0;
   } t7ic;
t7ic.ilvl = 12;
```

The compiler will locate and assign the bits as given below:

bit-no.	15 - 8	7	6	5 - 2	1 - 0
bit name	?	ir	ie	ilvl	glvl
binary value				1100	

Numbers, Variable, Data Types

Enumerated data types

- A *enum* variable may become symbolic values that are defined within the *enum* declaration:

    enum *[enum_name]*

    {

    value_1 [= ival]

    [,value_2 [= ival]]

    ...

    } *[enum_list];*

- **enum_name** is the name of the enumerated type. It may be omitted if the declaration won't be referenced in the program again. This is only true in connection with an immediate data definition with the **enum_list** or in connection with the typedef keyword. The enumeration is then called anonymous.

Numbers, Variable, Data Types

Enumerated data types

- **value** is any free choosable name but not a number.
- **ival** is the value that **value** shall be represented with. If not specified **value_1** will be assigned 0, **value_2** = 1, etc.
- **enum_list** is the possibility to define variables of type **enum enum_name** within the declaration. Again, it is advised to separate data declaration and data definition.

Numbers, Variable, Data Types

Remarks:

- Internally, an enum variable is treated as a signed integer. The compiler does not check the variable against the defined values of the enumeration value list. That is why enum variables should only be used in assignment and comparison operations.

- Enumerated values can be used as if they were defined as a macro, what means that they are known at compile time and can thus be used in switch-case statements.

- Enumeration variables are best suited to represent states and transitions of automatons.

Numbers, Variable, Data Types

Definition of private types

- As shown above, the struct, union, and enum keywords allow both type declaration and data definition in one statement. In order to separate this, the typedef keyword can be employed:

    typedef *basic_type type_name;*

- **typedef** “C“-keyword for data structure declaration.
- **basic_type** may be any type such as char, int, float, struct, union, enum, etc.
- **type_name** is any name allowed.

Numbers, Variable, Data Types

Type Conversion - Implicit type conversion

If operands of different types are combined in expressions an implicit type conversion is performed

Rule No.1: all char and short operands are converted to int

Rule No.2: if one operand is unsigned, the other operand is converted to unsigned as well

Rule No.3: all float operand are converted to double

Rule No.4: if the operand of an expression are of different types, calculations always occur with the widest type. (Width of a data type simply means the number of bytes a value occupies.

Rule No.5: the result of an expression is always adjusted to the type of variable the result has.

Numbers, Variable, Data Types

Example:

```
char v1;
int v2;
double v3;

v2 = v1+v3                    /*expression is double, result is int*/

v1 = v2 - 2*v1  /* expression is int. The result variable is of type
                   char. The most significant part of result is lost */
```

Type Conversion - Explicit type conversion

- That's why there is the convention that every type conversion must be casted explicitly.
- For variables, this is expressed by writing the desired data type in parenthesis as an operator in front of them:

```
unsigned int i1;
char c1, c2;

c1 = (char)(i1 - (int)c2);
```

- In this example it is obvious to the reader that there is a value range reduction by casting the expression as char. If it was missing the compiler would produce a warning only.

Assignment, Expression, Operators

- Simple and multiple variable assignment
- Expression and operators
- Supplementary remarks on some Operators

Assignment, Expression, Operators

Simple variable assignment

variable = expression;

variable	defined or declared memory location
=	assignment operator
expression	constituted from one more operands and operator
;	end of statement

Multiple variable assignment

var_1 = var_2 = [=…] = expression;

Assigns the same value to multiple variables.

Remarks:

- This is the preferred method to assign the same value to multiple variables because efficient code is generated, unless the order of assignment is of importance. In this case, single variable assignments should be used.

Assignment, Expression, Operators

Expression

[unary operator] operand [binary operator] [operand][…];

An expression constitutes from operands and operators:

unary operator: concern a single operand only

+	positive sign
-	negative sign
++	increment
--	decrement
&	address of
*	indirection
sizeof(name)	size of name in byte
(type cast)	explicit type casting
!	logical negation
~	bit by bit inversion

Assignment, Expression, Operators

Expression

binary operator: performs a two operand operation

- arithmetic operators

+	sum
-	difference
*	multiplication
/	division
%	modulo operation

- comparison operators

<	less than
<=	less or equal
>	greater than
>=	greater or equal
==	equivalence
!=	not equal
&&	logical AND
||	logical OR

Assignment, Expression, Operators

Expression

- bit-by-bit operations

&	AND
|	OR
^	XOR (exclusive OR)
<<	shift left
>>	shift right

- compound assignment operators

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

ternary operator: performs a three operand operation

? :	conditional

operand: constant
variable
pointer
return value of a function

Assignment, Expression, Operators

Shift operations

result = operand << shiftwidth ;

result = operand >> shiftwidth;

- Zeros are shifted in from right when shifting left.
- A right shift distinguishes between unsigned and signed variables. For unsigned values zeros are shifted in, for signed values the sign (highest bit) is duplicated. This is, however, not true for all compilers because ANSI-“C“ does not define anything that deals with bits.
- With shift operations, multiplication and divisions can be avoided for operands of value 2x, what results in a faster code.

Examples:

1. z = x * 2; ==> z = x << 1;
2. z = x*(-1); ==> z = -x;
3. z = x / 2; ==> z = x >> 1;

- Shift left operations must be employed with care because there is no overflow checking mechanism available.

Assignment, Expression, Operators

Operator's Hierarchy

Category	Operator	Execution	Description

1.	`()`
	`[]`
	`->`
	`.`
		left → right	function call or term grouping
			array subscript
			indirect structure element selection
			direct structure element selection

2. unary operators	`!`
	`~`
	`+`
	`-`
	`++`
	`--`
	`&`
	`*`
	`sizeof`
	`(`*type*`)`
		right ← left	logical negation
			bit-by-bit inversion
			positive sign
			negative sign
			increment
			decrement
			address of
			indirection
			size in bytes
			explicit type casting

3. multiply / divide operators	`*`
	`/`
	`%`
		→	multiplication
			division
			modulo operation for integer values

4. additive operators	`+` `-`	→	addition subtraction
5. shift operators	`>>` `<<`	→	shift left shift right

6. relational operators	`<`
	`<=`
	`>`
	`>=`
		→	less than
			less or equal
			greater than
			greater or equal

Assignment, Expression, Operators

Operator's Hierarchy

7. equivalence operators	== !=	→	equal not equal
8.	&	→	bitwise AND
9.	^	→	bitwise XOR
10.	|	→	bitwise OR
11.	&&	→	logical AND
12.	||	→	logical OR
13.	? :	←	conditional
14. assignment operators	= *= /= %= += -= &= ^= |= <<= >>=	←	assignment assign product assign quotient assign integer remainder assign sum assign difference assign AND-masked value assign XOR-masked value assign OR-masked value assign left shifted value assign right shifted value
15. comma	,	→	separator

Assignment, Expression, Operators

Examples:

- Incrementation *before* the calculated value is used: x=++n;
- Incrementation *after* the calculated value is used: x=n++;

- ++n && ++i ==> if n evaluates to 0, ++I is not executed any more.

- x = ++ (x+y); x = 10++; not allowed

- x = x | MASK; all bits of x which are set in MASK are set

- n = ~ n; negation of bits

- res = status & (~1); clear the bit 0

Assignment, Expression, Operators

Supplementary remarks on some Operators

- The modulo operation works for integer-by-integer divisions only. It returns the integer remainder.

- To move the result of a comparison (**TRUE** or **FALSE**) to a variable, the long version with an if-statement is recommended (produces less code):

```
if (a != 0) b=1;      is better than      b = (a != 0);
else b=0;
```

- Instead of the conditional operator, an if-...construct should be used, e.g

```
x = (a ? 1 : 0);      should be replaced by      if (a) x=1;
                                                 else x=0;
```

Assignment, Expression, Operators

Supplementary remarks on some Operators

- C has no special type to represent logical or boolean values.
- It improvises by using any of the integral types char, int, short, long, unsigned, with a value of 0 representing false and any other value representing true.
- Comparison operations result in a logical statement, where **FALSE** = 0 and **TRUE** is not 0. It can be represented by a single bit value.

"Zero" values:

- 0 (16 bits integer) is false
- The null character (\0) is false
- The NULL pointer is false
- The float 0 value is false

Any other values is true:

- 1 is true
- -1 is true

The character 'a' is true

A pointer which has been initialized with any address is true

Program Flow Statements

- Sequences
- Single Branching: The “if“-Statement
- Double Branching: The “if-else“-Statement
- Multiple Branching: The “switch - case“-Statement
- Loop with Testing in the Beginning: The “while“-Statement
- Indexed Loop with Testing in the Beginning: The “for“-Statement
- Loop with Testing in the End: The “do - while“-Statement
- Other Program Flow Statements: continue, break, goto

Program Flow Statements

Sequences

- A “C“-sequence can be a single statement, a list of statements separated by commas

    statement_1 [, ... [, statement_n]];

or a block:

```
{
   [definitions; ]
   statement_1;
   ....
   statement_n;
}
```

- A block may contain local definitions (not executable statements) as well as executable statements. Definitions must precede other statements.

Program Flow Statements

Single Branching: The “if“-Statement

if *(expression)*
  sequence

- Only if the boolean **expression** in parenthesis evaluates to logical TRUE, the **sequence** will be executed.
- Note that, for a sequence consisting of a single statement, the trailing semicolon belongs to the sequence. Therefore, it is not given above nor in the sequel.

Remark:

- To compare an unsigned int variable with unlike 0 the following syntax should be preferred :

  if (var!=0) or if (var) instead of if (var>0)

Program Flow Statements

Double Branching: The “if-else“-Statement

```
if ( expression )
  sequence_1
else
  sequence_2
```

- Both **if** and **else** are “C“-keywords, where **else** can only be used together with a preceding **if**. **Sequence_2** will be performed, if the boolean **expression** in parenthesis evaluates as logical FALSE.

Remarks:

- In if-else constructs the expression should be formulated so that the most probable result is TRUE. Then, most often the branch to **sequence_2** may not be taken, which results in a shorter program execution time.

Program Flow Statements

Remarks:

- If there are several conditions that all lead to the execution of the same sequence an encapsulation of several “if - else“ statements can be avoided by using boolean algebra:

```
if ( expression_1 || ( expression_2 && expression_3) )
sequence_1
```

- Be careful with the ordering of the expressions in parenthesis if they contain assignments. In the example above the expressions 2 and 3 will not be evaluated if expression 1 is TRUE!

Program Flow Statements

Multiple Branching: The “switch - case“-Statement

switch (*expression*)
{
    case *const_expression_1: statement_1; break;*
    ...
    case *const_expression_n: statement_n; break;*
    default: *statement_n+1; break;*
}

- **expression** will be evaluated. The result is of type **int** or **unsigned int**.
- **const_expression** must be a constant value which is known at compile time.
- If the **expression** in parenthesis matches the **const_expression**, the subsequent statements are evaluated up to the next break statement.

Program Flow Statements

Multiple Branching: The “switch - case“-Statement

- **break** “C“-keyword that causes leaving the actual block. Every **case** should have a **break** statement.

- **default** is a predefined label for all cases that do not match any of the other constant expressions in the block. In ANSI-“C“ it may be missing, which however means a bad programming style.

Remark:

- It is advised to put the cases in the order according to their probability because the compiler sometimes generates an assembly code which rather reflects an if … elseif ... else structure, especially if only few cases are given.

Program Flow Statements

Loop with Testing in the Beginning: The “while“-Statement

while (*expression*)
sequence

- The condition given by **expression** is evaluated first. Only if the result is TRUE the sequence is executed. It will be repeated as long as the **expression** is TRUE.

Remark:

- Potentially endless loops must be equipped with a software watchdog:

```
while (*int_ptr <= TopValue && special_exit == 0);
```

Program Flow Statements

Indexed Loop with Testing in the Beginning: The “for“-Statement

for (*[init_list] ; [expression]; [continue_list]*)
    sequence

- **init_list** is a list of statements separated by commas which will be executed unconditionally in advance of the loop.
- **expression** results in a boolean value TRUE or FALSE. As long as it is TRUE, the **sequence** is executed, followed by the statements of the **continue_list**.
- **continue_list** is a list of statements separated by commas which are evaluated as long as the **expression** is TRUE.

Program Flow Statements

Remarks:

- The loop control variable must only be changed in the **continue_list** but nowhere else.

- Likewise, the loop exit condition must only be checked in the **expression**.

- Loop counters within local functions should be dynamic variables and defined as signed int. This results in an efficient code.

- To compare a decreasing local counter with unlike 0 the following syntax should be used: for (i=10; i>0; i--) instead of for (i=10; i!=0; i--)

Program Flow Statements

Loop with Testing in the End: The “do - while“-Statement

```
do
   sequence
while ( expression );
```

- The **sequence** is executed at least once. After that the **expression** is evaluated, and the **sequence** is repeated as long as it’s result is TRUE.

Program Flow Statements

Other Program Flow Statements

- There are three other “C“-keywords concerning the program flow: goto, continue and break. However, goto must and the other two instructions should be avoided as they lead to an ill structured program flow. They are given for completeness here.

goto *label*;

...

label: statement

- **goto** causes the program to continue at the label which is defined at any other place in the program by a subsequent colon.

Program Flow Statements

Other Program Flow Statements

```
{
   ...
   continue;
   ...
}
```

- The **continue** statement is used in loop constructs and means a shortcut to continue with testing the next loop condition. Therefore, the statements following the **continue** statement won't be executed.

```
{
   ...
   break;
   ...
}
```

- The **break** statement causes the program to continue at the next label outside the current block. It should be used in connection with **switch - case** constructs but not otherwise.

Arrays, Pointers, Functions

- Arrays
- Pointers
  - Pointers scaling
  - Pointers and Function Arguments
  - Pointers and Arrays
  - Pointers vs.. Multi-dimensional Arrays
- Functions
  - Function Header
  - Function Body
  - Calling a function

Arrays, Pointers, Functions

Arrays

- Encapsulate multiple elements of a single data type with one variable name.
- May be one or two dimensional
- No checking mechanism for the boundaries
- Index must be of an unsigned type
- Data are contiguously allocated

- Array Declaration:

  <type> <name> [*<size>*];
- **type** could be any type;
- **size** is mandatory: it's the maximal number of elements in the array

- The declaration is going to allocate enough bytes for the whole array

Arrays, Pointers, Functions

Arrays

- Array could be initialized when declaring :

  <type> <name> [<optional size>] = { <value list> }

  - the **type** should be compatible with the values
  - in this case, the **size** is optional (if the size is not given, the size of the array is set to the number of element in the declaration list)

- Examples :

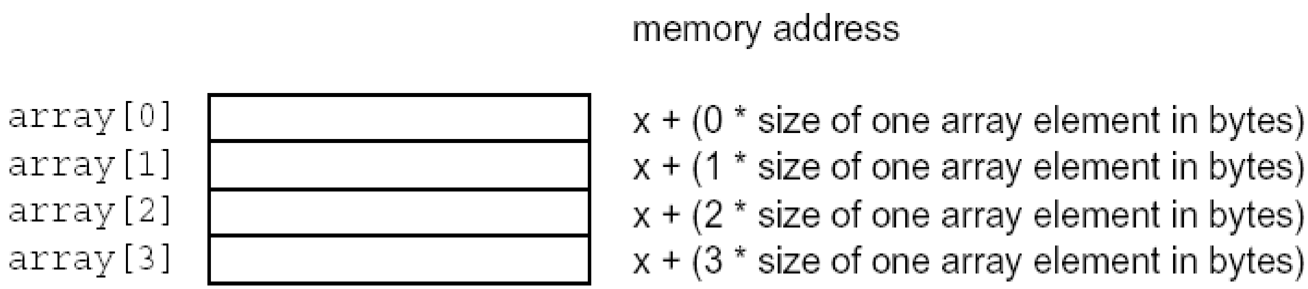
```
int t [4] ;
int j [] = {1, 2, 3 } ;        /* size 3 */
int k [5] = {1,2};             /* other elements are initialize to 0 */
float t[2][3] = {{2.3, 4.6, 7.2}, {4.9, 5.1, 9.3}};
int l [2] = {1, 2, 3 } ;       /* Incorrect */
char vocals[5] = {'a', 'e', 'i', 'o', 'u' };
char string[] = "This is a string";
```

Arrays, Pointers, Functions

Arrays

- The access to an element of the array is done with the index between brackets

  Example: r = a [i] + b [j];
- The first element of the array is 0 (zero-based index)



- Arrays address could be passed as parameter to function

Arrays, Pointers, Functions

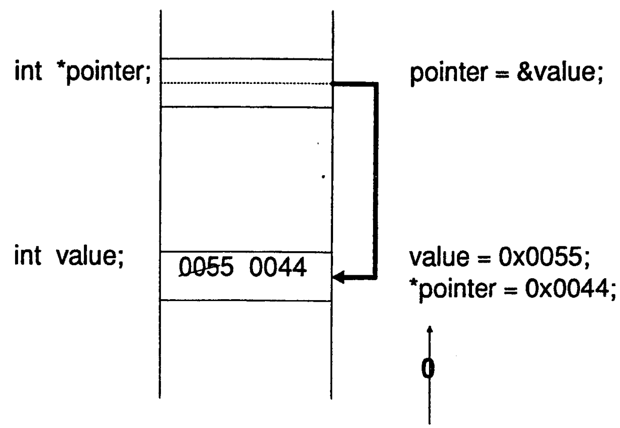
Pointers

- Are variables that contain the address of an object (variable or function).
- Enable indirect access to an object.
- Are defined with an asterix (*)

- **Definition:**

  *type * pointer_name;*

- **type** is the type of the object the pointer points to.
- ***** is the patch for a pointer.
- **pointer_name** is a free choosable name of the pointer.



Arrays, Pointers, Functions

Pointers

- A pointer is affected with another pointer, an address of an object with & operator, or a pointer expression:

```
char *p ; int *q; char a; int b;

        p = &a; /*correct*/
        p = &b; /* incorrect */

        q = &a; /* incorrect */
        q= &b ; /* correct */

        p = &q; /* address of the pointer */
```

Arrays, Pointers, Functions

Pointers

- Const keyword can be used in pointer declarations.

  For example:

```
int a; int *p;
int *const ptr = &a;        // Constant pointer
*ptr = 1;                   // Legal
ptr =p;                     // Error
```

- A pointer to a variable declared as const can only be assigned to a pointer that is also declared as const.

```
int a; int *p; const int *q;
const int *ptr = &a;        // Pointer to constant data
*ptr = 1;                   // Error
ptr =p;                     // Error
ptr =q;                     // Legal
```

Arrays, Pointers, Functions

Pointers

	Object a	Const object a

Pointer
Object
	-initialization
	p = &a;
	*p = a;	

Pointer Const object	- initialization p = &a;	- initialization p = &a;
Const pointer Object	- initialization *p = var;	
Const pointer Const object	- initialization	- initialization

Pointers and Function Arguments

- Pointer arguments enable a function to access and change objects in the function that called it.
- For instant, a sorting routine might exchange two out-of-order elements with a function called swap:

```
swap(&a, &b);
```

- Since the operator & produces the address of a variable, &a is a pointer to a.
- the parameters are declared to be pointers, and the operands are accessed indirectly through them.

```
void swap(int *px, int *py){      /* interchange *px and *py */
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

Pointers and Arrays

- Any operation which can be achieved by array subscripting can also be done with pointers.
- The pointer version will in general be faster
- The declaration

  int a[10]

  defines an array a of size 10, that is a block of 10 consecutive objects named a[0], a[1], ..., a[9].
- The notation a[i] refers to the i-th element of the array If pa is a pointer to an integer, declared as

  int *pa

  then the assignment

  pa = &a[0];

sets pa to point to element zero of a: that is, pa contains the address of a[0].

- Now the assignment x = *pa will copy the contents of a[0] into x.

Pointers and Arrays

- If pa points to a particular element of an array, then by definition:
  - pa+1 points to the next element, pa-i points i elements before pa, and
  - pa+i points i elements after.
- Thus, if pa points to a[0],

  *(pa+1)

  refers to the contents of a[1], pa+1 is the address of a[i], and *(pa+i) is the contents of a[i].
- By definition, the value of a variable or expression of type array is the address of element zero of the array. Thus after the assignment

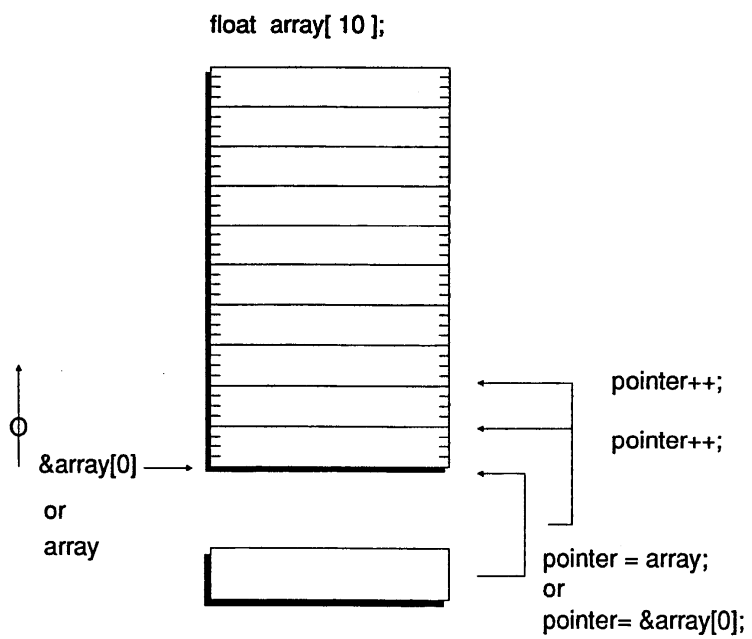
  pa = &a[0]

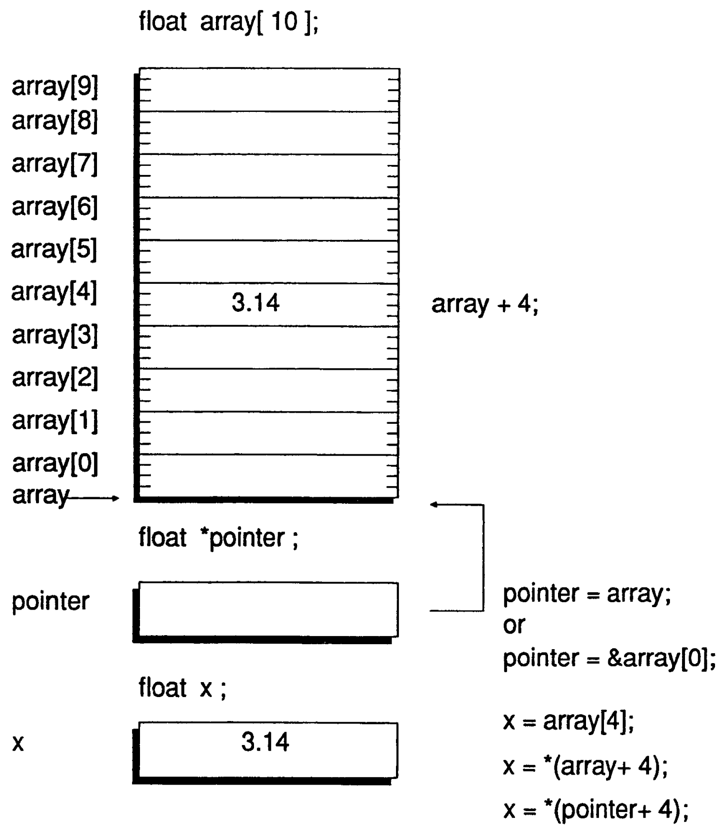
  pa and a have identical values
- The assignment pa=&a[0] can also be written as

  pa = a:

Arrays, Pointers, Functions

Pointers and Arrays





Arrays, Pointers, Functions

Pointers vs.. Multi-dimensional Arrays

- Given the declarations

```
int a[10] [20];
int *b[10];
```

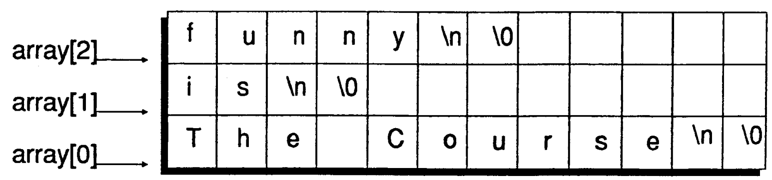
then a[3][4] and b[3][4] are both syntactically legal references to a single int

- But a is a true two-dimensional: 200 int-sized locations have been set aside, and the conventional rectangular subscript calculation 20xrow+col is used to find the element a[row,col]
- For b, however, the definition only allocates 10 pointers and does not initialize them; initialization must be done explicitly, either syntactically or with code.
- The important advantage of the pointer array is that the rows of the array may be of different lengths. That is, each element of b need not point to a twenty-element vector; some may point to two elements, some to fifty, and some to none at all.

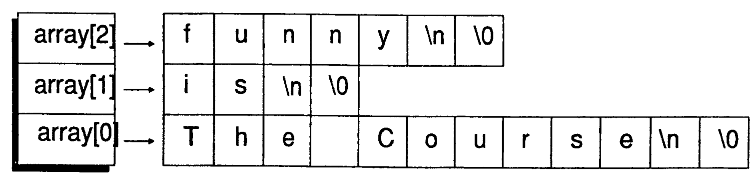
Arrays, Pointers, Functions

Pointers vs.. Multi-dimensional Arrays

```
char array [3][12] = {"The Course\n",
                      "is\n",
                      "funny\n"};
```




```
char *array[] = {"The Course\n",
                 "is\n",
                 "funny\n"};
```



Pointer to Strings

- C does not know string, only arrays of characters.
- To be able to use functions, pointers to strings are used when passing parameters.
- The whole processing of strings relies on the use of pointers.

- The incrementing of a pointer on characters shifts the pointer to the next valid element of the string.

```
char *text, character;
text = “C - training”;
character = *(text+4);
```

Arrays, Pointers, Functions

Pointers

- Only a small selection of operations are defined for pointers:

Operation	Sign	Examples:
		`int *iptr, *jptr;`
		`unsigned int offset, tmp;`

Assignment	=	`iptr = 0x4000;` `jptr = iptr;`
Increment Decrement	++ --	`iptr++;` `--jptr;`
Comparison	== != <= >= < >	`if (iptr >= 0x2000);` `tmp = (jptr != iptr);`
Addition	+	`jptr = iptr + offset;`
Subtraction	-	`jptr = iptr - 0xf800u;`
Pointer Distance	-	`tmp = iptr - jptr;`

Functions

- A function is defined by :
  - a header (name, type, formal parameters)
  - a body

- A function has:
  - either a return type
  - or a void type

- Non void functions have a result
  - add (x,y) return the addition of x and y

- Void functions do not return any result

Functions

- Function Header

Syntax:

<visibility><type><name>(<parameters list>)

type

- void
- scalar type(default int)

visibility

- *static* modifier used to hide the function

Arrays, Pointers, Functions

Functions

Function Header

- Parameters List
  - could be null, when no parameter is required (or void)
  - those parameters (the formal parameters) are considered as local variables, only available inside the body of the function
  - Two kinds of parameters are available :
    - input parameters (or value parameters) : when calling the function, the caller copy a value in each input parameter. The const keyword could be used to declare those parameters as constant (optional)
    - input-output parameters (or address parameters) : when calling the function, the caller give the address of a variable to the formal parameter. Those parameters are marked with the * operator

Functions

Function Body

- Delimited by { and }
- Allows local declarations
  - variables only
  - automatic data
  - the compiler is going to used register to store those data while registers are available
  - when no register is available, those variables are localized on the user stack
  - the static modifier forces the compiler to declare the data in a static area
  - the register modifier tells to the compiler that the data is going to be used very often (forces to reserve a register)

Pointer to Functions

- The address of a function can be assigned to a pointer variable.
- That means that a function can also be passed as a parameter.
- The call then happens with the aid of the dereferencing operator “*”.

```
type (*name)();

void sort(char *v[], int n, int
(*comp)())
{
            …
            int comp_res;
            comp_res =
(*comp)();
            ...
}
```

```
int main()
{
    extern int strcmp(), numcmp();
   …
    if(numerical)
            sort(lineptr, nlines, numcmp);
   else
            sort(lineptr, nlines, strcmp);
}
```

Preprocessor directives

Preprocessor directives

- Macro definition
- File inclusion
- Conditional compilation
- Memory model

Preprocessor directives

Preprocessor directives

- ANSI - C defines a set of preprocessor directives. These are always starting with # character in the first column line.
- The goal: flexible software for different application parameters and different software development tools.
- Preprocessing is scheduled before compilation process, and mainly consist in a text analyzing and processing tool. The produced file is used as input for compilation process.

Preprocessor directives

- **#define**
  introduces the definition of a preprocessor macro
- **#undef**
  clears a preprocessor macro definition
- **#include**
  includes the source text of another file
- **#if**
  evaluates an expression for conditional compilation.
- **#ifdef**
  checks for conditional compilation whether a macro is defined
- **#ifndef**
  checks for conditional compilation whether a macro is not defined

Preprocessor directives

- **#elif**

  introduces an alternative #if branch following a not compiled #if, #ifdef, #ifndef or #elif branch.
- **#else**

  introduces an alternative branch following a not compiled #if, #ifdef, #ifndef or #elif branch.
- **#endif**

  completes a conditional compilation branch
- **#line**

  indicates the line number and, optionally, a file name which is used in error logging files to identify the error position.
- **#error**

  reports an error which is determined by the user.

Preprocessor directives

- **#pragma**
  - Inserts a compiler control command.
  - Options for the compilation can be given just as in the command line.
  - A #pragma directive not known to the compiler is ignored and leads to a portable code.

Preprocessor directives

Macro definitions

#define *Macro_Name [[(parameters)] replace_text]*

- **#define**: preprocessor directive
- **Macro_Name**: is the name of the macro. Will be substituted by replace_text.
- **parameters**: used literally in the replace_text and replaced when the macro is expanded
- **replace_text**: the given text will replace the macro call literally

- It is forbidden to define macro with side effect (eg. increment) - disturbs the error checks consistency.
- **String operator: #**
  - is used in the replace_text to induce that the subsequent string is interpreted as the name of parameter. This name is replaced at the time the macro is expanded.
- **Token-pasting operator: ##**
  - precedes or follows a formal parameter. When it is expanded, the parameter is concatenated with the other text of the token.

Preprocessor directives

Files inclusion

#include *<file_name>*

#include *"file_name"*

- Additional information can be used in a source file (data types, function prototypes, ...) => header files.
- "…" - source file path is the first path were the include file is searched. If it can not be found there, the project specific include path will be inspected.
- <...> - omits the current source path and start the search of the include search path of the project.

- Some problem might arise if a header file defines some symbols and is included in multiple modules. The content of a header file must be delimited by a preprocessor switch.

Preprocessor directives

Predefined ANSI-“C“ Macros

- __FILE__ is replaced by the name without extension of the current file.
- __LINE__ is replaced by the current line number.
- __TIME__ is replaced by a string containing the time when the compilation was started.
- __DATE__ is replaced by a string containing the date when the compilation was started.
- __STDC__ is set to 1 for all compilers that are built up according to the ANSI standard.

Code-Tuning Techniques

Code-Tuning Techniques

Loops – Unswitching

- If the decision doesn't change while the loop is executing, you can unswitch the loop by making the decision outside the loop.

EXEMPLE OF UNSWITCHED LOOP

```
FOR I 0 I COUNT I++
  IF SUMTYPE NET
    NETSUM + AMOUNT I
   ELSE
     GROSSSUM + AMMOUNT I
```

EXEMPLE OF SWITCHED LOOP

```
IF SUMTYPE NET
   FOR I 0 I COUNT I++
     NETSUM + AMOUNT I
ELSE
   FOR I 0 I COUNT I++
     GROSSSUM + AMOUNT I
```

Code-Tuning Techniques

Loops - Jamming

- Is the result of combining two loops that operate on the same set of elements.

EXAMPLE OF SEPARATE LOOPS THAT COULD BE JAMMED

```
FOR I 0 I EMPL NUM I++
STATUS I EMPL STATUS I

FOR I 0 I EMPL NUM I++
AGE I EMPL AGE I
```

EXAMPLE OF A JAMMED LOOP

```
FOR I 0 I EMPL NUM I++
STATUS I EMPL STATUS I
AGE I EMPL AGE I
```

Code-Tuning Techniques

Loops - Unrolling

- The goal of loop unrolling is to reduce the amount of loop housekeeping.

EXAMPLE OF LOOP THAT CAN BEEN UNROLLED

```
I 1

WHILE I NUM

  A I I
  I++

```

EXAMPLE OF UNROLLED LOOP

```
I 1

WHILE I NUM

  A I I
  A I+1 I+1
  I+ 2


IF I NUM

  A I NUM

```

Code-Tuning Techniques

Loops - Minimizing the work inside loops

- One key to writing effective loops is to minimize the work done inside a loop.

```
EXAMPLE OF COMPLICATED POINTER EXPRESSION INSIDE A LOOP

for ( i = 0; i < num; i++ )
{
   netRate[ i ] = baseRate[ i ] * rates->discounts->factors->net;
}
```

```
EXAMPLE OF SIMPLIFYING A COMPLICATED POINTER EXPRESSION

   quantityDiscount = rates->discounts->factors->net;

   for ( i = 0; i < num; i++ )
   {
      netRate[ i ] = baseRate[ i ] * quantityDiscount;
   }
```

Code-Tuning Techniques

Loops– Sentinel Values

- When you have a loop with a compound test, you can often save time by simplifying the test.

EXAMPLE OF COMPOUND TEST INSIDE A LOOP

```
I 0
WHILE I ELEMENTCOUNT
  IF ITEM I TESTVALUE
    BREAK
  I++

/* CHECK IF ELEMENT FOUND */
IF I ELEMENTCOUNT
  ...
```

EXAMPLE OF USING A SENTINEL TO SPEED-UP A LOOP

```
ITEM ELEMENTCOUNT TESTVALUE
I 0
WHILE ITEM I TESTVALUE
  I++

/* CHECK IF ELEMENT FOUND */
IF I ELEMENTCOUNT
  ...
```

Code-Tuning Techniques

Loops– Putting the busiest loop on the inside

- When you have nested loops, think about which loop you want on the outside and which you want on the inside.

```
Example of having the busiest loop on the outside

sum = 0;

for (column = 0; column < 100; column++) {
   for (row = 0; row < 5; row++)
   {
      sum += table[row][column];
   }
}

/* 5*100 + 100 = 600 iteration tests */
```

```
Example of putting the busiest loop on the inside

sum = 0;

for (row = 0; row < 5; row++) {
   for (column = 0; column < 100; column++)
   {
      sum += table[row][column];
   }
}

/* 100*5 + 5 = 505 iteration tests */
```

Code-Tuning Techniques

Logic – Use table lookups for complicated expressions

- When you have nested loops, think about which loop you want on the outside and which you want on the inside.

```
EXAMPLE OF COMPLICATED CHAIN OF LOGIC

IF A C A B C
   CLASS 1
ELSE IF B A A C B
   CLASS 2
ELSE IF C A B
   CLASS 3
ELSE
   CLASS 0
```

```
EXAMPLE OF USING A TABLE LOOKUP TO REPLACE
COMPLICATED LOGIC

STATIC INT CLASSTABLE 2 2 2
   /* B C   BC   B C   BC */
      0   3   2   2   /* A */
      1   2   1   1   /* A */

CLASS CLASSTABLE A B C
```

Code-Tuning Techniques

Data Transformations – Use the fewest array dimensions possible

- Structure your data so that it's in a one-dimensional array rather than a two-dimensional or three-dimensional array

```
EXAMPLE OF STANDARD TWO DIMENSIONAL ARRAY
INITIALIZATION

FOR ROW 0 ROW NUMROWS ROW++
   FOR COL 0 COL NUMCOLS COL++
      MATRIX ROW COL 0
```

```
EXAMPLE OF ONE-DIMENSIONAL REPRESENTATION OF AN
ARRAY

FOR ENTRY 0 ENTRY NUMROWS*NUMCOLS ENTRY++
   MATRIX ENTRY 0
```

Code-Tuning Techniques

Data Transformations – Minimize array references

- Minimize array accesses.

Ex. of unnecessarily referencing an array inside loop

```
for ( discountLevel = 0; discountLevel < numLevels; discountLevel++ )
{
   for ( rateIdx = 0; rateIdx < numRates; rateIdx++ )
   {
      rate[ rateIdx ] = rate[ rateIdx ] * discount[ discountLevel ];
   }
}
```

Example of moving an array reference outside a loop

```
for ( discountLevel = 0; discountLevel < numLevels; discountLevel++ )
{
   thisDiscount = discount[ discountLevel ];
   for ( rateIdx = 0; rateIdx < numRates; rateIdx++ )
   {
      rate[ rateIdx ] = rate[ rateIdx ] * thisDiscount;
   }
}
```

Code-Tuning Techniques

Expressions – Use Caching

- Save a few values in such a way that you can retrieve the most commonly used values more easily than the less commonly used values.

```
EXAMPLE OF CACHING TO AVOID AN EXPENSIVE
COMPUTATION

UNSIGNED INT SQR UNSIGNED CHAR X
    IF X CACHED X

        RESULT CACHED SQR X

    ELSE

        CACHED X X
        RESULT CACHED SQR X X * X


RETURN RESULT
```

Code-Tuning Techniques

Expressions– Exploit algebraic identities

- You can use algebraic identities to replace costly operations with cheaper ones.

EXAMPLE OF ALGEBRAIC OPERATION

EXAMPLE OR REDUCING THE ALGEBRAIC OPERATION

Code-Tuning Techniques

Expressions– Initialize at compile time

- Precompute the result of functions called with constant arguments.

```
EXAMPLE OF LOG-BASE-TWO ROUTINE BASED ON SYSTEM
ROUTINES

UNSIGNED INT LOG2 UNSIGNED INT X

    RETURN UNSIGNED INT LOG X / LOG 2
```

```
EXAMPLE OF A LOG-BASE-TWO ROUTINE BASED ON A SYSTEM
ROUTINE AND A CONSTANT

UNSIGNED INT LOG2 UNSIGNED INT X

  DEFINE LOG2 0.69314718
  RETURN UNSIGNED INT LOG X / LOG2
```

Code-Tuning Techniques

Expressions – Be wary of system routines

- System routines are expensive and provide accuracy that's often wasted

```
EXAMPLE OF A LOG-BASE-TWO ROUTINE BASED ON
FLOATING POINTS

UNSIGNED INT LOG2 UNSIGNED INT X

   DEFINE LOG2 0.69314718
   RETURN UNSIGNED INT LOG X / LOG2
```

```
EXAMPLE OF A LOG-BASE-TWO ROUTINE BASED ON INTEGERS

UNSIGNED INT LOG2 UNSIGNED INT X

   IF X 2 RETURN 0
   IF X 4 RETURN 1
   IF X 8 RETURN 2
   IF X 16 RETURN 3
   IF X 32 RETURN 4
   ...................
   IF X 1024 RETURN 9
   IF X 2048 RETURN 10
   IF X 4096 RETURN 11
   IF X 8192 RETURN 12
   IF X 16384 RETURN 13
   IF X 32768 RETURN 14
   RETURN 15
```

Code-Tuning Techniques

Expressions – Precompute results

- Computing results before the program executes and wiring them into constants that are assigned at compile time
- Computing results before the program executes and hard-coding them into variables used at run time
- Computing results before the program executes and putting them into a file that's loaded at run time
- Computing results once, at program startup, and then referencing them each time they're needed
- Computing as much as possible before a loop begins, minimizing the work done inside the loop
- Computing results the first time they're needed and storing them so that you can retrieve them when they're needed again

Code-Tuning Techniques

Expressions – Eliminate common sub expressions

- If you find an expression that's repeated several times, assign it to a variable and refer to the variable rather than recomputing the expression in several places.

```
EXAMPLE OF COMMON SUBEXPRESSION

PAYMENT  LOANAMOUNT / POW RATE/12  3  + RATE/12
```

```
EXAMPLE OF ELIMINATING A COMMON SUB EXPRESSION

FACT  RATE/12
PAYMENT  LOANAMOUNT / POW FACT  3  + FACT
```

Bibliography

- C programming language - Kernighan, Ritchie
- C traps and pitfalls - Koening
- Code complete – McConnell
- MISRA guidelines for the use of the C language in vehicle based software
- Programming languages - C ISO/IEC 9899:1999(E)