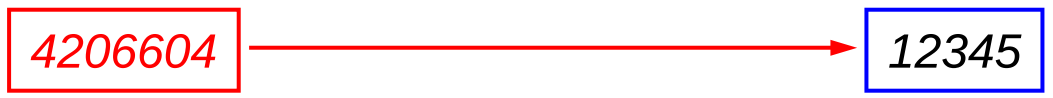
Pointer – Definition

Def: A Pointer is a variable or expression that refers to some memory location.

Pointer variable ptr

Variable a

Type: int* (pointer to int)



type int (4 Byte)

uses memory locations 4206604-4206607 (0x40300C-0x40300F)

Pointer variable ptr contains the address of the first (smallest) memory location that holds contents of variable a.

The pointer variable itself request memory space too.
For 32-bit wide address spaces (x86) the pointer size is 4 byes. On a 64-bit system (x86_64, IA64, PPC64) 8 bytes are needed.

Note: casts from a pointer type to `int` are dangerous. The following code works on x86 but not on x86_64 (`int` is only 4 byte on both platforms):

&aVar retrieves address of aVar

```
int aVar;
int address = (int)&aVar;
```

Variables, Addresses and Pointers

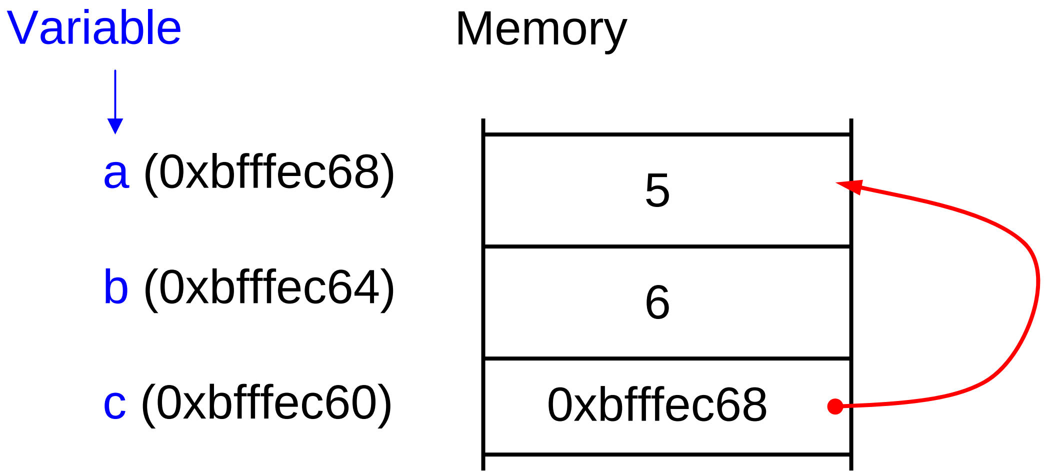
A simple C program with three local variables:

```
main()
{
   int a = 5;
   int b = 6;
   int* c;

   // c points to a
   c = &a;
}
```

defines c as a pointer to a variable of type **`int`**

address operator retrieves memory location of variable a



Defining a Pointer in ANSI-C

There are two identical styles for declaring a pointer:

```
int*   x;
float* y;
char*  z;

etc.
```

or alternatively
(Asterisk to the variable)

```
int   *x;
float *y;
char  *z;

etc.
```

Note:

- Both types are really the same!
- When a pointer variable is declared space is only allocated for the storage of the pointers content (a memory address) but not for the variable the pointer points to.
- A pointer is bound to the data type that was specified in the definition.
- Use **`void*`** as a general pointer type that can point to any variable/address.
- Every pointer can be cast in a **`void*`** pointer and vice versa.

Reference and Dereference

Reference **&**   Retrieve the memory address of a variable

```
int a = 6;
int* c = &a;  // &a is the memory location of variable a
```

Dereference *****   Accessing the variable (content) the pointer points to
(Indirection)

```
int a = 6;
int* c = &a;

*c = 7;       /* Changes content of variable a by using
                 its address stored in pointer c */
```

equivalent to

```
a = 7;
```

Using Pointers for Out-Parameters in Functions

Example: Lets write a function `swap(a, b)` that exchanges the content of the two variables **a** and **b**.

However the following code has a flaw:

```
void swap(int x, int y)
{
   int tmp;
   tmp = x;
   x = y;
   y = tmp;
}
```

When calling swap(a,b) the formal parameters x, y of the function are replaced by the values of a and b

The swap-code in the function does indeed exchange the content of x and y.

But how is the caller affected by this change?

In order to really exchange the content of the variables their addresses must be passed to the function not their values. Call by-reference instead of call by-value is required.

$\Rightarrow$ Use `int*` instead of `int`.

Using Pointers for Out-Parameter in Functions (2)

Change the programs as follows:

```c
void swap(int* x, int* y)
{
   int tmp;
   tmp = *x;
   *x = *y;
   *y = tmp;
}
```

When calling swap use address operator **&** to get the addresses of the two variables.

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

Function Pointers

- A function pointer is a variable that points to a function.
- Definition of a function pointer:

-
  ```
  returnType  (*functPtr)(paramType1, paramType2)
  ```

- **Example:** **`fptr`** is a function pointer that can point to any function that takes an **`int`** and **`float`** argument and returns a **`float`**.

```
float (*fptr)(int, float)
```

- Assigning a function pointer

```
float func(int i, float f) {
   return i+f;
}
...
fptr = func;        // assignment ptr to function
```

- Calling a function the function pointer points to

```
fptr(42, 3.14f);
```

Function Pointer – Example

- Calculator

```
int add(int x, int y) { return x+y; }
int sub(int x, int y) { return x-y; }
int mul(int x, int y) { return x*y; }
int div(int x, int y) { return x/y; }

int evaluate(unsigned int op, int x, int y) {
   int (*eval[])(int, int) = { add, sub, mul, div };
   if (op>3) {
      printf("error invalid function");
      return 0;
   }
   return eval[op](x, y);
}

main()
{
   printf("%d\n", evaluate(3, 42, 3));
}
```

each operation is enumerated 0=add, 1=sub, 2=mul, 3=div

array function pointers to function with signature int f(int, int).

one function for each enumerated operation.

Const and Pointers

- In C a pointer can be declared "constant" in two ways
- Constant pointer
  - Syntax: **`TYPE * const ptrName = &aTYPEVar;`**
  - Pointer variable: constant (cannot be overwritten)
  - Data the pointer points to: not constant
  - Example

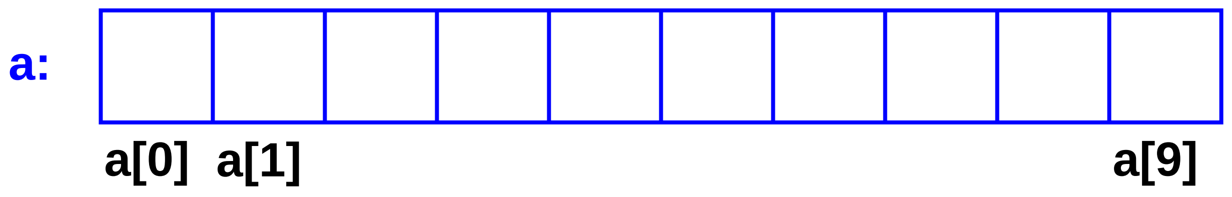
```
int a = 42, b = 42;
int* const ptr = &a;
*ptr = 1; // ok
ptr = &b; // wrong
```
→ gcc error: assignment of read-only variable 'ptr'
- Pointer to constant data
  - Syntax: **`const TYPE * ptrName = &aTYPEVar;`**
  - Pointer variable: not constant
  - Data the pointer points to: constant (cannot be overwritten)
  - Example

```
int a = 42, b = 42;
const int* ptr = &a;
*ptr = 1; // wrong
ptr = &b; // ok
```
→ gcc error: assignment of read-only location

Declaring Arrays in C

Following declaration defines an array with 10 elements of type **`MyType`**. The fields can be accessed by an index, a[0], a[1], …, a[9].

```
MyType a[10];
```



`a[i]` is the ith element des Arrays, **`i`** is called index.

Initialisation of Arrays

An array can be initialised when it is declared.

Example:

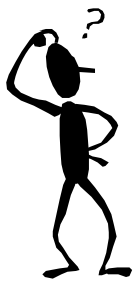
```
int primes[] = {2, 3, 5, 7, 11, 13};
```

Creates an array with 6 elements and initialises it with the specified values.
Equivalent code:

```
int primes[6];
primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
primes[3] = 7;
primes[4] = 11;
primes[5] = 13;
```

Variant with curly braces is probably better.

Accessing Arrays



If you run the following program you get a strange output

```
#include <stdio.h>

int a[12], b;

main()
{
   int i;
   b = 5;
   printf(“b=%d\n“, b);

   for (i=0; i<=12; i++) {
      a[i] = i;
   }
   printf(“b=%d\n“, b);
}
```

Output:

```
b=5
b=12
```

What‘s wrong here?

fix: use < instead of <= !

The last element written is a[12]. But array was initialised as with length 12, thus last element is a[11]. We are writing across the array borders and therefore into variable b!

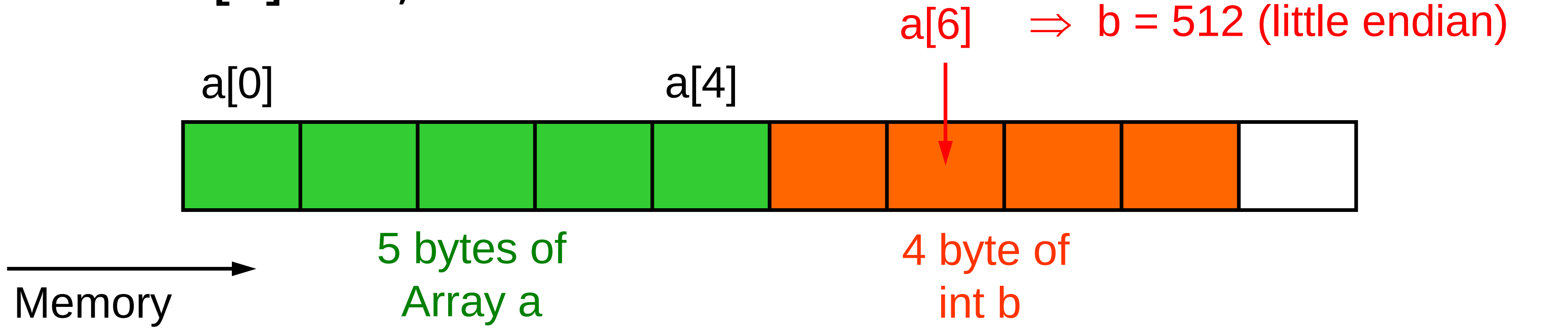
Accessing Arrays (2)

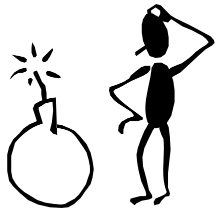
In C there is no boundary check when accessing arrays, i.e., the runtime system does not check if index **`i`** of **`a[i]`** points to a valid array element.
It just adds the offset (index) to the array‘s base address.

Example:

```
char a[5];
int b = 0;

a[6] = 2;
```


Overwrites the second byte of **int b**!
⇒ b = 512 (little endian)
a[6]
a[0]
a[4]
5 bytes of Array a
4 byte of int b
Memory



Warning: These bugs are really hard to find!

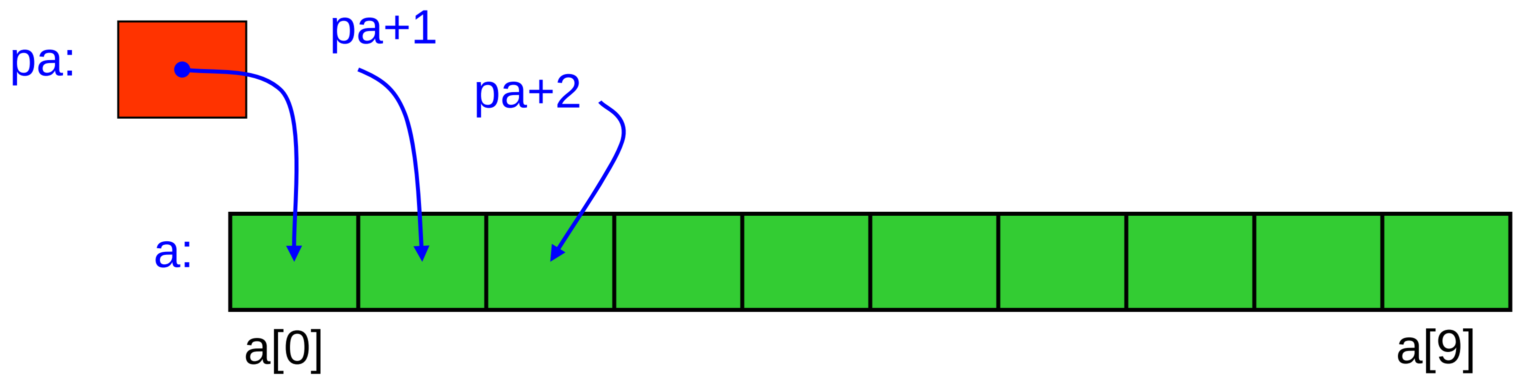
Pointer and Arrays

Arrays and Pointer are related. Instead of accessing an array element by its index also a pointer can be used:

```
int a[10];
int* pa;
pa = &a[0];
```

Accessing the first array element as

```
x = *pa;  // equivalent to x=a[0]
```



For accessing the ith element, i.e., **`a[i]`**:

```
y = *(pa+i);
```

Note: That pa+1 is not one added to the value of pa. Instead the address is pa + size of an element, i.e., **`sizeof(*pa)`**.

Pointer and Arrays (2)

An array can be considered as a pointer to the first element of the array. Therefore the following expressions are equivalent:

`pa = &a[0];`	$\Leftrightarrow$	`pa = a;`
`&a[i]`	$\Leftrightarrow$	`a+i`
`pa[i]`	$\Leftrightarrow$	`*(pa+i)`
`char s[]`	$\Leftrightarrow$	`char* s`

An array can be passed (by-reference) to a function.

```
int a[10];

f(a);    // or equivalent
f(&a[0]);
```

Function f:

```
void f(int* array)
{
    ...
}

// or equivalent
void f(int array[])
{
    ...
}
```

Arithmetic with Pointers

Pointers can be added and subtracted:

```
int a[10];
int* pa = a;

x = *pa;     // x = a[0]

pa++;
x = *pa;     // x = a[1]

pa = pa+5;   // x = a[6]
x = *pa;

pa = pa-4;   // x = a[2]
x = *pa;

pa += 3;     // x = a[5]
x = *pa;
```

Dynamic Allocation of Arrays

- Sometimes the size of an array is unknown at compile time
  ⇒ dynamic allocation at runtime
- Dynamic allocation with runtime support
  Function **`malloc`** from Standard-C-Library fetches memory from operating system
- Dynamically allocated memory on heap
- In order to use **`malloc`** the prototype must be present, it can be included from **`stdlib.h`** header file.

```
#include <stdlib.h>
```

$\longleftarrow$ At the beginning of the C file

Prototype of **`malloc`**:

```
void* malloc(size_t size);
```

number of bytes to allocate

return value: pointer to first byte of array-block, or 0 (NULL) if allocation failed.

malloc returns a **`void*`** pointer (general untyped pointer) to the first element of of the array. For later assignment to the array pointer a cast is necessary from **`void*`** to the effective pointer type of the array.

Dynamic Allocation of Arrays (2)

Example:

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

main()
{
   int* a;

   // allocates 4 MB main memory (1M ints)
   a = (int*)malloc(4*1024*1024);
```

Malloc returns a **`void*`** Pointer. Since we work with **`int*`** we need to cast the universal type to the more specific type **`int*`**.

```
   // deallocates (frees) array
   free(a);
}
```

Deallocation of previously allocated Memory

- A previously allocated block of memory eventually must be returned to the operating system.
  - implicitly when the program exits and the process terminates
  - explicitly by calling a function **`free`**
- Function **`free`**, is also part of the Standard-C-Library
- Only memory allocated earlier with **`malloc`** can be freed
- Always the entire memory block has to be freed (otherwise use **`realloc`**)

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

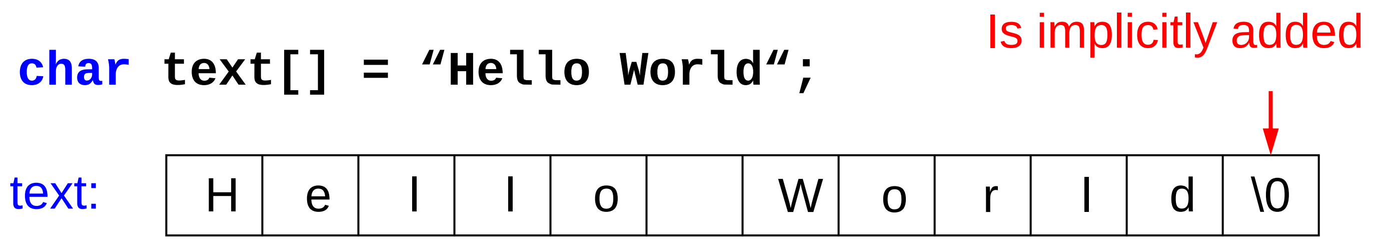
Pointer to first byte of memory block
to be deallocated.
Note: This address has to be the address return
by malloc when the block was allocated.

C-Strings

A String is an array consisting out of char elements. String literals are written in double-quotes: **`“Hello World“`**

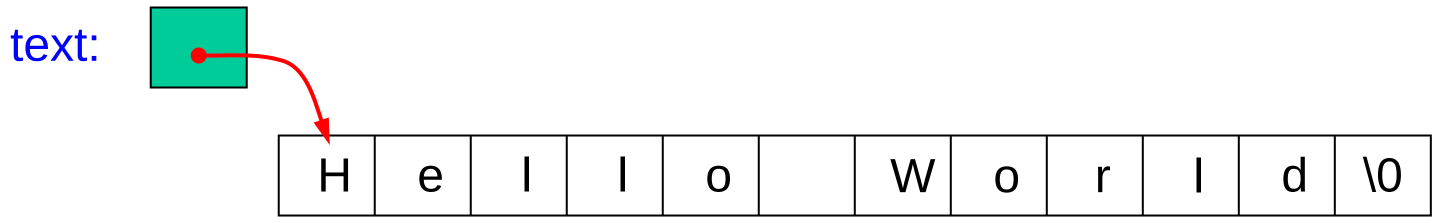
The end of a C-String must be marked with a Null-Character **‘\0‘** (with a String-Terminator).

```
char text[] = “Hello World“;
```



Also possible is the following declaration:

```
char* text = “Hello World“
```



The C-Language does not provide any other features for working with C-Strings, e.g., copying strings or retrieve the length of strings.
⇒ Instead use the function from the Standard-C-Library.

C-Strings (2)

Standard-C-Library provides a series for function that work on C-Strings.
In order to use these function you need the prototypes from the **`strings.h`** header file.

```
#include <strings.h>
```
$\longleftarrow$ At the beginning of the C file

Some useful functions:

```
// copies string source to destination
char* strcpy(char* destination, char* source);

// get number of characters in the string
size_t strlen(const char* str);

// compare string str1 with string str2
int strcmp(const char* str1, const char* str2);
```

Multi-dimensional Arrays

- Declaration

  `TYPE array2d[num_dim1][num_dim2];` 2-dimensional

  `TYPE array3d[num_dim1][num_dim2][num_dim3];` 3-dimensional
- **`num_dim1`**, …, **`num_dim3`** is number of indices in each dimension
- Array elements are accessed as

  `array2d[i][j]` and **`array3d[i][j][k]`**

  with 0 ≤ **`i`** < **`num_dim1`**, 0 ≤ **`j`** < **`num_dim2`** and 0 ≤ **`k`** < **`num_dim3`**
- Array elements are **stored in row-wise order**
- Initialisation of multi-dimensional arrays

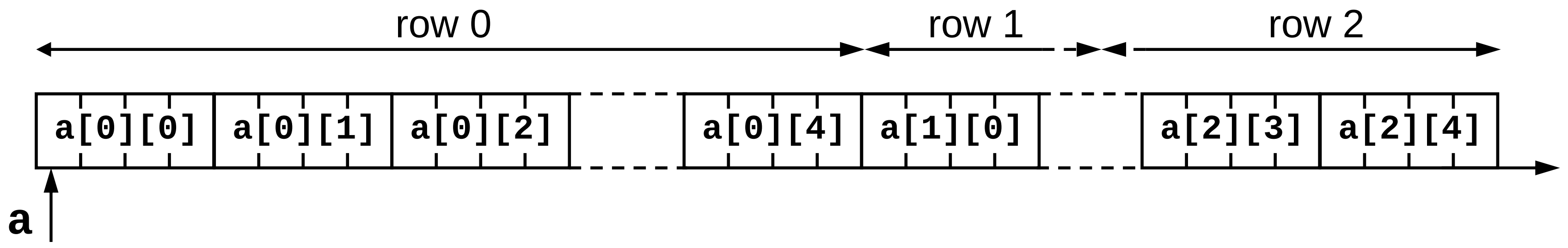
```
double matrix[4][4] = { {1.0, 0.0, 0.0, 0.0},
                        {0.0, 1.0, 0.0, 0.0},
                        {0.0, 0.0, 1.0, 0.0},
                        {0.0, 0.0, 0.0, 1.0} };
```

can be omitted,
i.e. []

specifies storage size
(row-stride) required by the
compiler to compute address

2-dimensional Arrays

- Example: **`int a[3][5];`** (int: 4 byte)
- Row-wise storage:
  - stride (distance) between elements a[i][j] and a[i][j+1]: 4 bytes
  - stride between elements a[i][j] and a[i+1][j]: 4*5 bytes



- Index computation:
  - Array declaration: **`TYPE array[n][m]`**
  - Element access: **`array[i][j]`**
  - Pointer to element **`array[i][j]`**
    `TYPE *element = (TYPE*)array+i*m+j`
  - Byte address to element **`array[i][j]`**
    $\texttt{char *byteaddr = ((char*)array) + (i*m + j)*}\underbrace{\texttt{sizeof(TYPE)}}_{}$

    number of bytes required for a variable of type **TYPE**

Pointers and Multi-dimensional Arrays

- In order to compute the address of an element the compiler needs to now the number of columns (2-dim)
- Then how to pass array to a function (**`TYPE*`** does not work)?
- Example: **`int array[3][15]`** is to be passed to function **`f`** which is then invoked as **`f(array)`**
- Possible declarations of function f
  - **`f(int x[3][15]) { … }`**
  - **`f(int x[][15]) { … }`**
  - **`f(int (*x)[15]) { … }`**

  identical
  since the number of
  rows is irrelevant
- Note that for the last declaration the parentheses around **`*x`** **cannot** be omitted then
  - **`f(int *x[15]) { … }`**
  - specifies an array of 15 pointer to integer instead of a pointer to an array with 15 integer!

Multi-dimensional Arrays – Limitations

- ANSI C[89] standard does not allow multi-dimensional arrays whose dimensions are determined at runtime[1], i.e., row-stride must be known at compile time.
  Example:

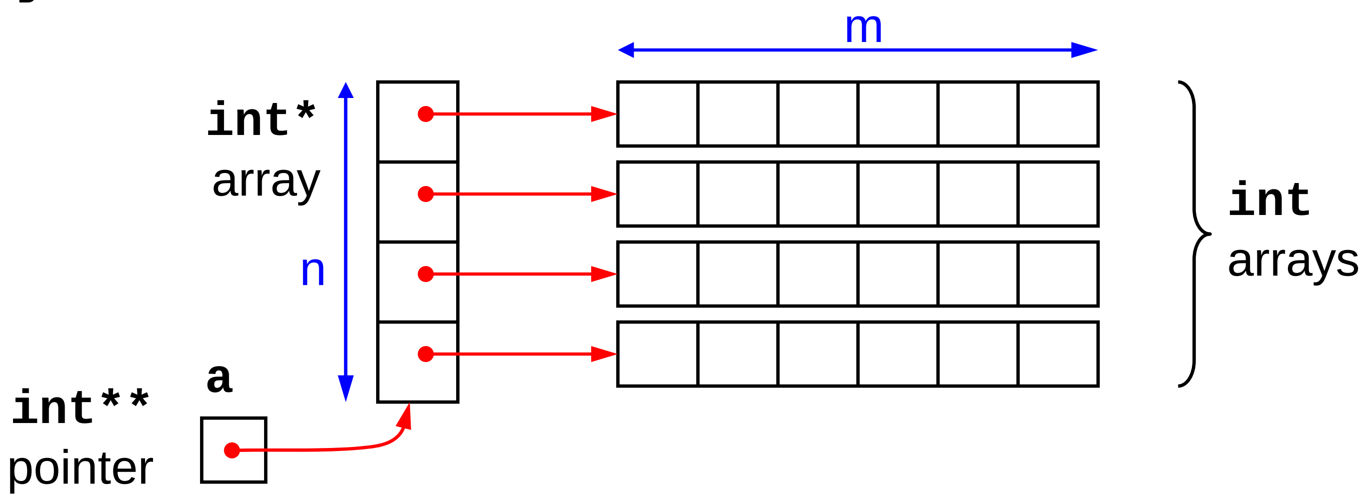
  -
    ```
    void foo() {
       int n,m,i,j;
       printf("n m: ");
       scanf("%d %d", &n, &m);
       int a[n][m];
       for (i=0; i<n; i++)
          for (j=0; j<m; j++)
             a[i][j] = …
    }
    ```
    size unknown at compile time
  - Permitted in C99 and does compile on GCC (even if -std=c89 is specified)
  - Works for local arrays that can be allocated on stack. But what about global arrays (stored in data segment (.bss or .data) but not on the stack)?

[1] Ritchie D.M.: Variable-Size Arrays. Journal of C Language Translation. Vol. 2 No. 2, 1989

Alternative 1: Arrays of Pointers

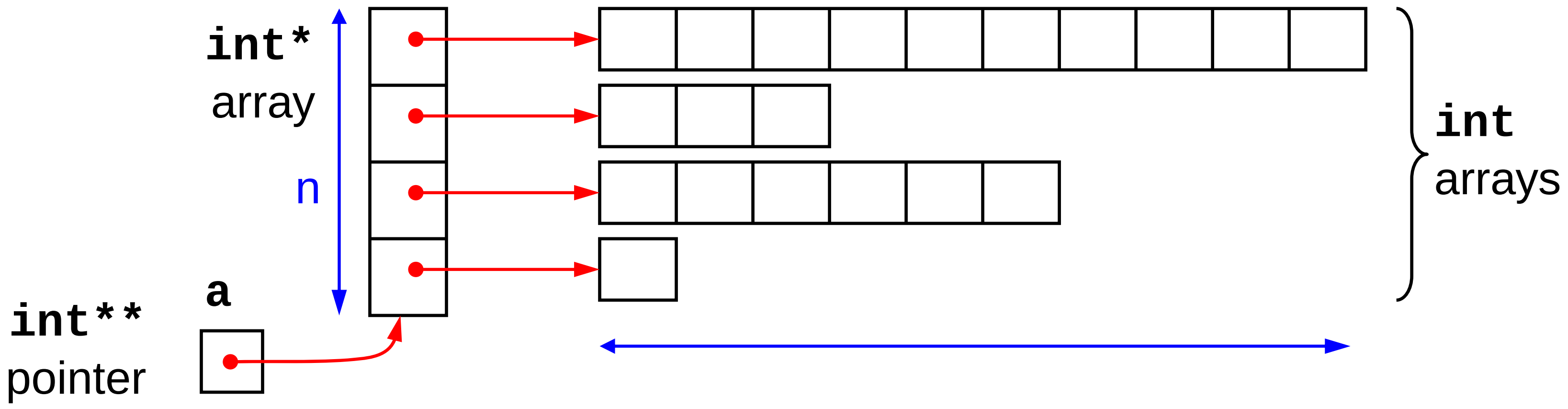
- Define multi-dimensional array as arrays of pointers
- Example 2-dimensional array

```
int i, n, m, **array;
printf(“n m: “);
scanf(“%d %d”, &n, &m);
array = (int**)malloc(n*sizeof(int*));
for (i=0; i<n; i++) {
   array[i] = (int*)malloc(m*sizeof(int));
   for (j=0; j<m; j++) array[i][j] = f(i,j);
}
```



Alternative 1: Arrays of Pointers (2)

- Advantage
  - Permits dynamic array allocation for global multi-dimensional arrays
  - Allows definition “non-rectangular” arrays



- Disadvantage
  - Complicated allocation and deallocation
  - Memory leak if deallocation (with **`free`**) is forgotten
  - Element access requires two memory operations
  - Row-arrays not necessarily stored on consecutive memory location $\rightarrow$ non-uniform strides $\rightarrow$ negative effect on caching

Alternative 2: “Manual” Multidimensional Arrays

- Idea for array **`TYPE x[n][m][r]`**
  - map n-dimensional array to a 1-dimensional array and manually compute index
  - 1. Create 1-dimensional array of size n*m*r elements
    `TYPE *x = (TYPE*)malloc(n*m*r*sizeof(TYPE));`
  - 2. Compute index **`x[i][j][k]`** manually
    for row-wise storage: **`x[i*m*r+j*r+k]`**
    for column-wise storage: **`x[i+j*n+k*m*n]`**
- Suggestion: define a macro
  - **`#define array(i,j,k) x[(i)*m*r+(j)*r+(k)]`**
  - Usage: **`y = array(1,2,3)`**
- Advantage
  - Direct control about storage locations of elements, possibility to exploit characteristics of caching $\rightarrow$ better performance possible

Complicated Pointer Declarations

- **`char **argv`**
  argv: pointer to pointer to char
- **`int (*x)[13]`**
  x: pointer to array[13] of int
- **`int *x[13]`**
  x: array[13] of pointer to int
- **`void *comp()`**
  comp: function return pointer to void
- **`void (*comp)()`**
  comp: pointer to function returning void
- **`char (*(*x())())[5]`**
  x: function returning pointer to array[] of pointer to function returning char
- **`char (*(*x[3])())[5]`**
  x: array[3] of pointer to function returning pointer to array[5] of char
  – Isn’t C beautiful?

source: K&R p. 122

Enumeration Types

An enumeration is a list of constant integer values. A variable of this enumeration type can be a value of the enumeration list.

tag is the name of the enumeration

```
enum  tag  { list of names };
```

Example:

Tag ‚direction‘ is the name of the enumeration (north, south, east, west)

```
enum direction { north, south, east, west };
```

```
enum direction d;
```

Variable d is an enumeration of ‚direction‘

```
direction = east;
```

Any value of the enumeration list can be used in the assignment. The symbolic name is substituted by the integer constant.

```
if (direction == south)
   printf(“wrong way!“);
```

Enumeration Types (2)

Enumeration are represented as integer values. In symbolic names are enumerated automatically in the definition statement.

```
enum direction { north, south, east, west };

printf(“%d\n“, north);    ⇒ 0
printf(“%d\n“, south);    ⇒ 1
printf(“%d\n“, east);     ⇒ 2
printf(“%d\n“, west);     ⇒ 3
```

Optionally the corresponding integer value can be specified for each symbolic name in the enumeration list.

Example: Definition for missing Boolean type in C as enumeration:

```
enum bool { true=1, false=0 };

enum bool test = true;
printf(“%d\n“, test);
```

Enumeration Types (3)

Anonymous enums do not have a type name. Thus no other variables can exist except the ones in the declaration statement of the enumeration list.

```
enum { SEEKING, FOLLOWING, STOP } state, anotherState;
```

In general a tag name and a number of variables of that enum type can be specified in the declaration statement:

```
enum state_type { SEEKING, FOLLOWING, STOP }
     state, anotherState;
```

This is identical to the following three lines:

```
enum state_type { SEEKING, FOLLOWING, STOP };
enum state_type state;
enum state_type anotherState;
```

Enumeration Types (4)

- Note since an enum is in fact an integer variable of type enum it basically can have any integer assigned to it.
- Generally the compilers do not complain if a value is assigned that is not in the enumeration list (even though they might in order to assure type consistency)

```
enum bool { true=1, false=0 };
enum bool value;
```

`value = 5;` Valid even though 5 is no in the enumeration list

- The GCC does not check that the value is part of the enumeration list.

- An enumeration type enhances legibility of the code, e.g. the debugger can replace integer values by the symbolic names.

Type Names and Aliases (typedef)

typedef creates a new name for an existing type.

existing type

new alias, i.e., name of new type

```
typedef  type  new-name;
```

Example:

```
typedef unsigned char BYTE;
typedef int seconds;
```

BYTE is now an unsigned 8 Bit type

```
BYTE biteMe;
seconds waitTime;
```

Variable biteMe is of type BYTE and and thus also of type “unsigned char”.

Therefore the variable declaration is equivalent to:

```
unsigned char myBite;
int waitTime;
```

Typedef and Enumeration Types

Variables of enumeration type have to be declared with keyword **enum** followed by the tag name.
Using **typedef** an “real” type alias can be defined. The new type can be used as any other type (e.g. **int**, **float**). The keyword enum no longer has to be used in the variable declaration:

```
enum bool { true=1, false=0 };
typedef enum bool boolean;

boolean aBoolean;          // uses typedef alias
enum bool anotherBoolean; // uses original enum definition
```

Since the enumeration type tag ‘bool’ is no longer used, one can declare the enumeration as an anonymous enum:

```
typedef enum { true=1, false=0 } boolean;
boolean aBoolean;
```

Structs – Example

For a geometry package a point object is used that is represented by two coordinates. The components are combined to a single entity, the struct. (related to OO, in fact in C++ **`struct`** is the same keyword of **`class`** except that all members are public).

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

As for the enumeration types, for a struct a tag name has to be specified. The struct with tag name “point” then consists of two integer members x and y.

Variable myPoint of type point is declared as (similar to enums):

```
struct point mypoint;
```

The members of a struct can be accessed by their name preceded by a dot ‘.’:

```
mypoint.x = 4;
mypoint.y = 3;
```

Structs

As for enumeration types, in the declaration statement also variables can be defined.

```
struct {
    int x;
    int y;
} A, B;
```

Variables A and B are structs with components x and y. Important: The struct itself is nameless, i.e. no further variables of this struct can be defined.

```
struct point {
    int x;
    int y;
} A, B;
```

Variables A und B are both types of struct point with two integer components x and y.

Struct variables can be initialised in the declaration statement. The initial values are specified in braces. The order of the values is the same as the member list in the definition of the struct.

```
struct point A = { 3, 4 };
```

$\xleftrightarrow{\text{identical}}$

```
struct point A;
A.x = 3;
A.y = 4;
```

Structs – Summary

The general syntax of struct definition is:

```
struct tag-name {
   type1 component1;
   type2 component2;
   ...
   typen componentn
 } variable1, variable2;
```

`tag-name`	name of the struct
`type1`	type of component1
`componente1`	name of member
`variable1`	variable1 is of this struct type

Later further variables of this struct can be defined using keyword **`struct`** and the tag name:

```
struct tag-name  variable-name;
```

The member of the struct can be access the variable name followed by a dot '.' and the name of the member.

```
variable.component
```

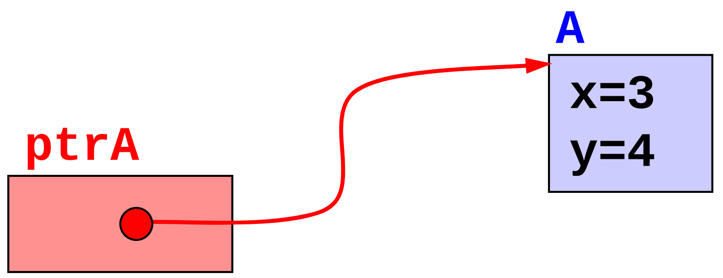
Struct variables can be initialised in the declaration statement. The initial values are specified in braces in the same order as the member in the struct definition.

```
struct tag-name variable-name = { initval1, ... };
```

Structs und Pointers

Pointer to struct variables are also possible:

```
struct point A = { 3, 4 };
struct point* ptrA = &A;
```



However there is an important difference. Members in struct variables are accessed using a dot '**.**' between variable name and the name of the struct member.

```
A.x = 5;
A.y = 6;
```

But using a pointer to a structs the arrow '**->**' has to be used for the accessing the member.

```
ptrA->x = 2;
ptrA->y = 5;
```

Since pointer **ptrA** points to **A** the members of variable are manipulated.

```
printf("(%d, %d)\n", A.x, A.y);
```
$\Rightarrow$ (2, 5)

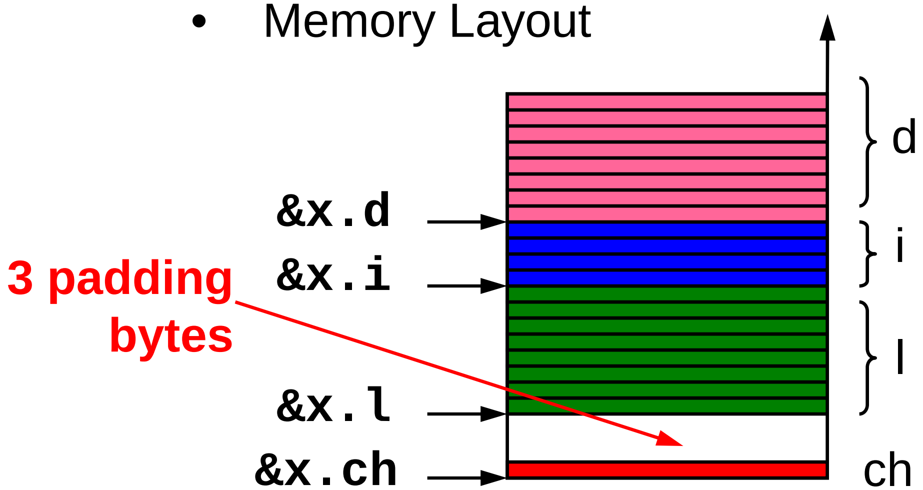
Memory Layout of Structs (IA32 Linux)

Example:

```
struct foo {
   char ch;
   long long int l;
   int i;
   double d;
};
main() {
   struct foo x;
   printf("ch:%p\n",&x.ch);
   printf("l: %p\n",&x.l);
   printf("i: %p\n",&x.i);
   printf("d: %p\n",&x.d);
   printf("%d\n",sizeof(x));
}
```

compiled with GCC
(for Linux on IA32)

- Output
```
ch:0xbfced6e0
l: 0xbfced6e4
i: 0xbfced6ec
d: 0xbfced6f0
24
```
- Memory Layout



- GCC does not use dense packing, it inserts padding bytes between the struct members
- By using of padding bytes the members can be word-aligned which speeds up memory access.

Memory Layout of Structs (IA32 Windows)

Example:

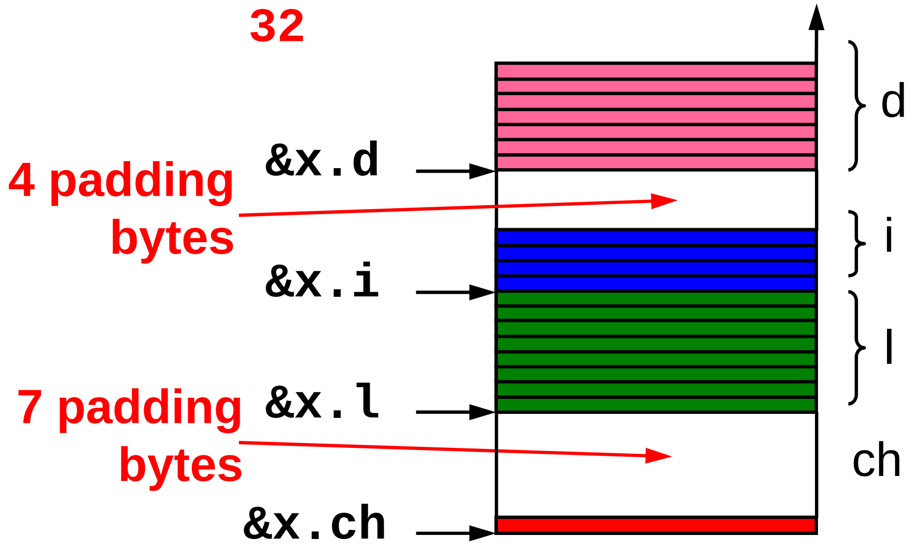
```
struct foo {
   char ch;
   long long int l;
   int i;
   double d;
};
main() {
   struct foo x;
   printf("ch:%p\n",&x.ch);
   printf("l: %p\n",&x.l);
   printf("i: %p\n",&x.i);
   printf("d: %p\n",&x.d);
   printf("%d\n",sizeof(x));
}
```

compiled with GCC

(for Windows on IA32)

- Output

```
ch:0x22ccc0
l: 0x22ccc8
i: 0x22ccd0
d: 0x22ccd8
32
```



- GCC does not use dense packing, it inserts padding bytes between the struct members
- By using of padding bytes the members can be word-aligned which speeds up memory access.

Memory Layout of Structs (Packed, IA32 Linux)

Enforcing packed structs on GCC:

```
struct foo {
  char ch;
  long long int l;
  int i;
  double d;
} __attribute__((packed));
main() {
  struct foo x;
  printf("ch:%p\n",&x.ch);
  printf("l: %p\n",&x.l);
  printf("i: %p\n",&x.i);
  printf("d: %p\n",&x.d);
  printf("%d\n",sizeof(x));
}
```

`__attribute__((packed))` instructs GCC not to insert padding bytes.

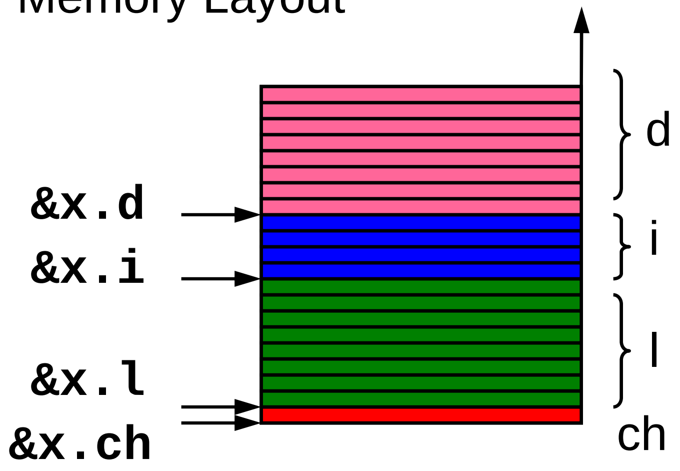
Note: this only works for GCC not necessarily for other compilers.

(might not be supported by all platforms)

- Output

```
ch:0xbfcd7d20
l: 0xbfcd7d21
i: 0xbfcd7d29
d: 0xbfcd7d2d
21
```

- Memory Layout



Typedefs and Structs

As for enums, typedef can be used to create an alias to the struct type. Thus if the typedef type is used, the keyword **struct** can be omitted in the variable declaration.

```
typedef struct {  // anonymous struct
   int x;
   int y;
} point;          // typedef alias with name point

point A;  // definition using struct and typedef name
point B = {3, 2};
```

Without using **typedef** the declaration is:

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

struct point A;  // point is tag name of the struct
struct point B;
```

Structs in Linked Lists

- There is an important difference between the tag name and the typedef name of a struct when the struct itself is used as a member of itself, e.g., in linked lists
- Example:
```
typedef struct {
   node* previous;
   node* next;
   int value;
} node;
```
- Cannot be compiled: “error: parse error before 'node'”. The identifier “node” is not known when it is used in the definition of the member **`previous`**.

- Instead the struct tag name has to be used:
```
typedef struct node_t {
   struct node_t *previous;
   struct node_t *next;
   int value;
} node;
```

Unions

unions provide a way to manipulate different kinds of data in a single area of storage. Unions consists of several members of different types that are referenced to by a name. But in contrast to structs all members of a union are mapped to the same memory location.

Example:

tag name of this union type

```
union int_float {
    unsigned int int_value; // 32 bit signed integer
    float float_value;      // 32 bit floating point number
} x;
```

x is a variable of this union type

As for structs, union members can be accessed with dot '**.**' followed by the union name (additionally, for union pointers members have to be accessed with arrow '**->**').

```
x.float_value = 3.141592654f;  // set as float
printf("%f", x.float_value);
```

⇒ 3.14593

```
// read content as unsigned int (decimal and hex)
printf("%u\n", x.int_value);
printf("0x%X\n", x.int_value);
```

⇒ 1078530011

⇒ 0x40490FDB

Unions (II)

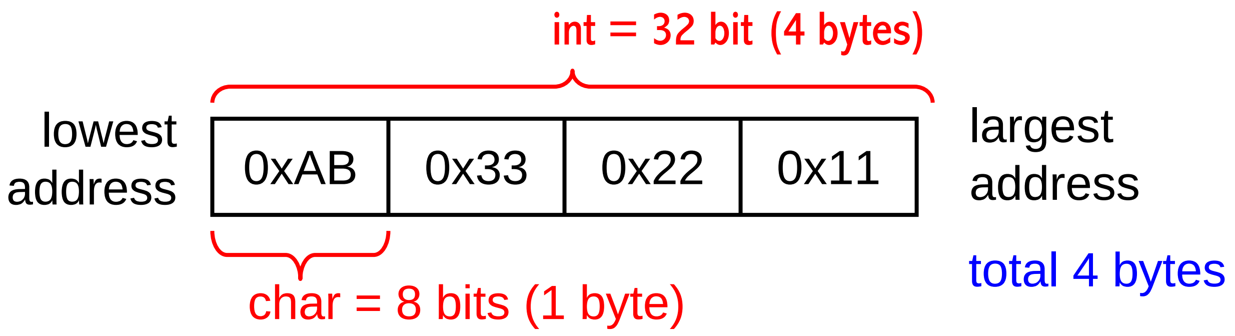
If the member types of a union differ in size the size of the union is equal the to the size of the largest member type.

```
union int_char {   ←  the size of this union is 4 bytes (= 32 bits)
   unsigned char char_value;  // 8 bit
   unsigned int int_value;    // 32 bit
} x;

x.int_value = 0x11223344;
x.char_value = 0xAB;

printf(“int:    0x%X\n“, x.int_value);  ⇒ 0x112233AB
printf(“char:   0x%X\n“, x.char_value); ⇒ 0xAB
printf(“sizeof: %d\n“, sizeof(x));      ⇒ 4     (bytes)
```

Representation of x in memory: Little Endian representation (Intel & Co.)



Unions – Summary

The general syntax of a union definition consists of a tag name, the member (with their types) and a list of variables of this union types.

```
union tag-name {
   type1 member1;
   type2 member2;
   ...
   typen membern
 } variable1, variable2;
```

`tag-name`	Type name of this union
`type1`	Type of member1
`member1`	member name
`variable1`	variable of this union type

Definition of variables of union with specified union tag name:

```
union tag-name  variable-name;
```

Union members are accessed with a dot '.':

```
variable-name.member
```

and for pointers to unions (as with structs) with '**->**':

```
union tag-name* ptr;
ptr->member;
```

Typedef und Union

As for enums and structs, a type alias can be defined. Instead of using the keyword union and the union name the typedef alias can be used to declare a variable.

```
typedef union {
   unsigned char char_value;
   unsigned int int_value;
} int_char;

int_char x;  // definition using typedef alias
```

Or using the union tag name:

```
union int_char {
   unsigned char char_value;
   unsigned int int_Value;
};

union int_char x;  // int_char is the union tag name
```

Application of Unions

- Representation of IP addresses in BSD sockets
```
struct sockaddr_in {
  short   sin_family;       // e.g. AF_INET for internet
  u_short sin_port;         // port number
  struct  in_addr sin_addr; // ip address
  char    sin_zero;
};
struct in_addr {
  union {
    struct { u_char s_b1, s_b2, s_b3, s_b4 };
    struct { u_short s_w1, s_w2 };
    u_long S_addr;
  } S_un;
};
```
- Set IP “129.132.98.12”:
```
struct sockaddr_in ddr;
addr.sin_addr.S_un.s_b1 = 129;
addr.sin_addr.S_un.s_b2 = 132;
addr.sin_addr.S_un.s_b3 = 98;
addr.sin_addr.S_un.s_b4 = 12;
```
- Macros **`htonl(…)`** used to adapt endianess from host to network order, and **`ntohl(…)`** from network to host order.

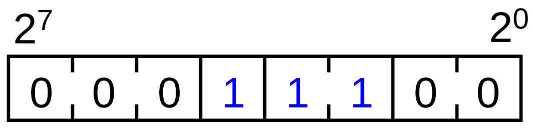
Bit Fields

- When storage space is a premium, it may be necessary to pack several objects into a single machine word, e.g. a collection of bit flags.
- Example

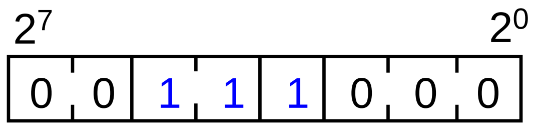
```
union error_flags {
  unsigned char error_byte;
  struct {
    unsigned int cntr:3; // 3 bits
    unsigned int ovfl:1; // 1 bit
    unsigned int regs:2; // 2 bits
    usignedd int flgs:2; // 2 bits
  } error_bits;
} error;
error.error_byte = 0;
error.error_bits.ovfl = 1;
error.error_bits.regs = 3;
printf(“0x%X”, error.error_byte);
```

- Aligning of bit field depends also on endian-ess of the machine.

on PowerPC: $\Rightarrow$ 0x1C



on x86: $\Rightarrow$ 0x38



Summary Types

- Enum types are integer types whose values are specified as symbolic names in the enumeration list.
- Structs are a collection of variables of different types and size. Structs allow encapsulation of attributes to objects.
- Unions is a collection of variables of different type and size. But in contrast to structs all members of the union are mapped to the same storage area. The size of a union is equal to the largest size of its member types.
- Typedefs can be used to create alias for enum, struct and union types. When the typedef name is used the keywords enum, union and struct can be omitted. In fact with its typedef name a user defined type can be treated as any other data type (int, float, etc.)