Systems 3

Pointers

Marcel Waldvogel

Department of Computer and Information Science
University of Konstanz

Winter 2019/2020

These slides are based on previous lectures held by Alexander Holupirek, Roman Byshko, and especially Stefan Klinger.

Chapter Goals

- How are variables stored/accessed in memory?
- What data types do exist? What sizes do they come in? What does the `sizeof` operator return when?
- What is the difference between accessing a variable by name and through its address/through a pointer?
- What are the benefits/dangers of pointers?
- How does pointer arithmetic work?
- What are the similarities/differences between array accesses and pointer arithmetic?
- How to use pointers as parameters?
- What are the benefits/dangers of `const`? When are 'constants' writable?
- How do `struct`, `union`, and `enum` work? What about pointers to them?

```c
int main(void)
{
    char c = 'B';
    unsigned int i = 0xdeadbeef;

    return 0;
}
```

- The memory cells are enumerated ⇒ **Memory address**
- Variables occupy **space** in memory, the amount depending on their **type**.
- The `sizeof` operator (*cf.* later) gives the size of a type:

variable	*address*	*memory*	
	0xffffd85f		
i	0xffffd860	0xef	*int*
	0xffffd861	0xbe	
	0xffffd862	0xad	
	0xffffd863	0xde	
	0xffffd864		
	0xffffd865		
	0xffffd866		
c	0xffffd867	B	*char*
	0xffffd868		

`sizeof(char)` = 1 (by definition; ≥8 bits)

`sizeof(int)` = 4 (this may vary; ≥16 bits)

Data types and sizes

Sizes are machine-dependent

- Each compiler is free to choose **appropriate sizes** for its own hardware. ISO C defines compile-time limits:
  - `char` is *at least* 8 bits (CHAR_BIT)
  - `short` and `int` are *at least* 16 bits
  - `long` is *at least* 32 bits
  - `short` is no longer than `int`, `int` is no longer than `long`
- Can be obtained with the `sizeof` operator.
- Numerical limits[1] are documented in `<limits.h>` and `<float.h>`. Additional limits are specified in `<stdint.h>`[2]

On my machine

char	1
short int	2
int	4
long int	8
long long int	8
float	4
double	8
long double	16
void *	8

[1] ISO C99 : 7.10/5.2.4.2 : Numerical limits

[2] ISO C99 : 7.18 : Integer Types (see also https://en.wikipedia.org/wiki/C_data_types#Basic_types)

Assignments

An **assignment** stores data at a location in memory.

```
x = y;
```

- Symbol `x` refers to the **place in memory** where the variable content is stored.
  - This is called an **l-value**, as in *locator*, or *left-hand-side*.
- Symbol `y` refers to the **data** stored at `y`'s place in memory.
  - This is called an **r-value**, as in *right-hand-side*.
- An r-value that is not an l-value: `x+y`, the result of which lies in a CPU register.

The type determines **how much data** is copied in the assignment:

- `char x, y;` ⇒ copy 1 byte.
- `double x, y;` ⇒ copy 8 bytes.

Introduction to pointers

- A **pointer** is just a variable that contains a memory address.
- Size of a pointer is usually 4/8 bytes on a 32/64 bit machine, **independent** of the type of data it points to.
- At compile time, the compiler knows what type of data a pointer points to (we will use this information later).
  There is one exception: `void *`.
- **Declaring a pointer**

```
int *p;       /* variable p points to an int */
char *q;      /* variable q points to a char */
```

- Note, that the `*` belongs to the *variable*, not to the *type*, *i.e.*,

```
int *p, i, *q;      /* p, q are pointers, i is an int */
char c, *r;         /* r is a pointer, c is a char */
int* bad, style;    /* pun intended: discuss type of style */
```

Question What is this?

```
double *dp, atof(char *);
```

The address operator: &

```c
int main(void)
{
    unsigned int i, *p;

    i = 0xdeadbeef;
    p = &i;  /* p points to i */

    (void)p;
    return 0;
}
```

- Unary operator & gives the **starting address** of an object.
- & only applies to objects in memory, *e.g.*, variables, array elements..., **not** &(x+3), &42, ...

variable	*address*	*memory*	
i	0xffffd860	0xef	
	0xffffd861	0xbe	*integer*
	0xffffd862	0xad	
	0xffffd863	0xde	
p	0xffffd864	0x60	
	0xffffd865	0xd8	*pointer*
	0xffffd866	0xff	
	0xffffd867	0xff	

Exploring memory

We can actually observe this...

Note that this example was observed on a 64 bit machine. Thus, the pointers use 8 bytes!

```
$ pk-cc pointer_int.c
$ gdb a.out
GNU gdb (GDB) 7.6.1
[...]
(gdb) start
[...enter s (i.e., step) several times...]
9               return 0;
(gdb) p/x i // print var i in hex
$1 = 0xdeadbeef
(gdb) p p // print var p
$2 = (unsigned int *) 0x7fffffffe6d4
(gdb) x/4b $2 // examine 4 bytes at the address
0x7fffffffe6d4: 0xef 0xbe 0xad 0xde
(gdb) p &p // where is the pointer
$4 = (unsigned int **) 0x7fffffffe6d8
(gdb) x/8b $4 // what does it look like? my pointers are 64 bit wide
0x7fffffffe6d8: 0xd4 0xe6 0xff 0xff   0xff 0x7f 0x00 0x00
```

The dereferencing operator: *

```c
#include <stdio.h>

int main(void)
{
   int i = 23,
   *p = NULL;  /* initialize pointer p, not the integer */

   p = &i;  /* copy address of i into p */
   i = 42;  /* change value of i */

   printf("%d\n", *p);  /* get what p points to */

   return 0;
}
```

Output:

```
$ ./a.out
42
```

- `*p` returns the data `p` points to.
- The special value `NULL` can be assigned to any pointer.
  It **must not** be dereferenced ⇒ points nowhere.

`void` pointer

- A `void` pointer carries an address, but the compiler **does not know** (*i.e.*, does not maintain) the type of data pointed to.
- **Declaration**

```
void *p;      /* type of referenced data unknown */
```

- Cannot be dereferenced ⇒ typecast required.

```
int y, x = 23;
p = &x;          /* p gets address of x, but type information is not passed on */
y = *(int *)p;
```

- Can be assigned to/from any pointer variable. (without typecast)

```
int *ip;
ip = p;       /* ip points to x, assuming an int there */
```

- In fact, somewhere in the `#include`d (*cf.* later) code, there is:

```
#define NULL (void *)0
```

- `printf`(3) directive `%p` prints the value of a (`void`) pointer.

Watch out!

```c
#include <stdio.h>

int main(void)
{
    int    *ip, i = 23;
    double *dp, d = 3.14159;
    void   *p;

    p = &i;
    ip = p;
    printf("%p %d\n", p, *ip);

    p = &d;
    dp = p;
    printf("%p %f\n", p, *dp);

    ip = p;
    printf("%p %d\n", p, *ip);

    return 0;
}
```

Output:

```
$ ./a.out
0xffee9480 23
0xffee9478 3.141590
0xffee9478 -266631570
```

- What went wrong here?

Operator precedence

Unary operators

- Unary operator * and & bind more tightly than binary arithmetic ops.

```
y = *ip + 1;
```

  - takes whatever ip points at
  - adds 1, and assigns the result to y

- The following statements[3] have the same effect:

```
*ip += 1;
++*ip;
(*ip)++;
```

  - All statements increment what ip points at
  - The **parentheses are necessary** in this last example.
    Otherwise, the expression would **increment the pointer** ip instead of what it points to. (We will use this later...)

[3] the returned value is unused

arity	assoc.	operators
1	postfix	`++`, `--`, `()`, `[]`
2	left	`.` `->`
1	prefix	`++`, `--`, `+`, `-`, `!`, `~`, `(type)`, `*`, `&`, `sizeof`
2	left	`*`, `/`, `%`
2	left	`+`, `-`
2	left	`<<`, `>>`
2	left	`<`, `>`, `<=`, `>=`
2	left	`==`, `!=`
2	left	`&`
2	left	`^`
2	left	`|`
2	left	`&&`
2	left	`||`
3	right ?	`?:` But `a?b,c:d` is parsed as `a?(b,c):d`
2	right	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, `|=`
2	left	`,`

Call by reference

```c
#include <stdio.h>

void swap(int *px, int *py)
{
   int tmp;

   tmp = *px;
   *px = *py;
   *py = tmp;
}

int main(void)
{
   int x = 23, y = 42;

   printf("x=%d, y=%d\n", x, y);
   swap(&x, &y);
   printf("x=%d, y=%d\n", x, y);

   return 0;
}
```

- Pointer arguments enable a function to access and change objects in the calling function.
- Can you use `swap` to swap `char`s? `double`s?

Function `getint()`: Get integer from input

- The program is fed with a space-separated sequence of integers.
- Write a function `getint()`, which reads the next integer from *stdin* every time it is called, as long as there is more input.

```
$ ./a.out <<<'0 1 -12 12345'
0
1
-12
12345
```

The function `getint()` has to

- return the integer values it found, and
- signal end of file (`EOF`, when there is no more (valid) input.

Question: What is the problem?

Seperate paths back to caller

Problem Statement:

- No matter what value is used for `EOF`, it could also be the value of an input integer.

Solution[4] The values are passed back through **seperate paths**.

- Let `getint return` an **indicator of success**.
- Use **pointer argument** to hand back the converted integer.

[4]This approach is used often in C

Using `getint()`

Repeatedly get and print an integer, until end of file, or invalid input:

```
int main(void)
{
    int i;

    while (getint(&i))
        printf("%d\n", i);

    return 0;
}
```

- Each call returns the next integer found in input.
- It is essential to pass the **address of** `i` to `getint`, this is where the converted integer is "returned" to the caller.

getint()

```
int getint(int *p)        /* Return 0 on EOF, 1 otherwise.  Store int at passed address */
{
   int c, sign = 1;

   while (isspace(c = getchar()))     /* cf. isspace(3) */
        /* skip white space */

   if (c == '-') {      /* store optional minus sign */
      sign = -1;
      c = getchar();
   }

   if (!isdigit(c))     /* pathological case */
      return 0;

   *p = c - '0';                      /* parse the digits */
   while (isdigit(c = getchar()))
      *p = 10 * *p + c - '0';

   *p *= sign;    /* apply sign */

   return 1;
}
```

Pointer arithmetics

Arrays and Pointers

- An array variable never is an l-value (*i.e.*, one cannot assign to it).
- The value of an array variable is the **address of the first element**.

```
int a[2];
int *pi = a;  /* the same as &a[0] */
```

Exceptions If the array is...

- ...operand of `sizeof`, the size of the array is returned,

```
printf("%zu\n", sizeof(a));    /* size in chars, not array cells */
```

- ...operand of `&`, the address of the first element is returned, typed as "pointer to array"

```
int (*pi2)[] = &a;       /* we will come back to this later */
```

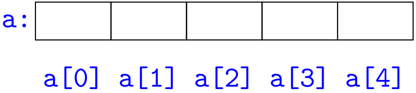
- ...a literal string initializer for a character array, the array is initialised with the string.

```
char a[] = "Hello world";
```

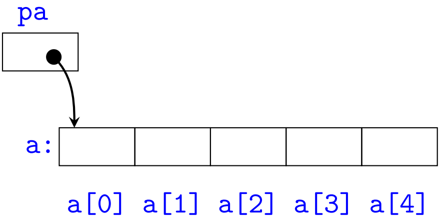
Pointer into array

- Any operation achieved by **array subscripting** can also be done with **pointers**.

```
int a[5];     /* Define an array a of size 5 */
int *pa;      /* Pointer to an integer */
int x;
```



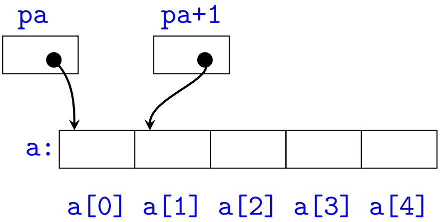

```
pa = &a[0];  /* same as  pa = a */
x = *pa;
```



- Assignment `pa = &a[0]` sets `pa` to point to element zero of `a`.
- Assignment `x = *pa;` copies the content of `a[0]` into `x`

Adding 1 to a pointer

- If `pa` points to a particular element of an array,
- then, *by definition*, `pa+1` points to the **next element**
  - Here, the size of the type is utilised!

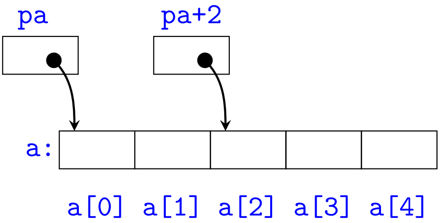


In general If `pa` points to any array element, then

- `pa+1` points to the next element (if it exists), and
- `pa-1` points to the previous element (if it exists).

Adding `i` to a pointer

- If `pa` points to `a[0]`, then `*(pa+2)` refers to the contents of `a[2]`.



In general If `pa` points to `a[k]`, then

- `pa+i` evaluates to the address of `a[k+i]`,
- `*(pa+i)` evaluates to the contents of `a[k+i]`.

Warning You must not go outside the array bounds! Nobody will check this for you. Your program may fail in the most inconvenient way.

Scaling and pointer arithmetics

A pointer and an integer may be added (or subtracted):

- The construction `p + n` means the **address of the `n`-th object** beyond the one `p` currently points to.
- `n` is **scaled** according to the **size of the object** `p` points to (which is determined by the **type** given in the declaration of `p`).
- Holds **regardless of the type or size** of the variables in the array.

Transformation of array access into pointer form:

$$\texttt{p[i]} \quad \equiv \quad \texttt{*(p + i)}$$

This is done by the compiler, quite tenaciously:

```
char a[] = "hello world";
printf("%c\n", 4[a]);        /* what is this? */
```

Example: Scaling according to type

```
int    a[5] = { 0 };        int    *pa = a;
char   b[5] = { '\0' };     char   *pb = b;
double c[5] = { 0.0 };      double *pc = c;

for (int i = 0; i < 5; i++)
printf("%d   %p   %p   %p\n",
i, (void *)(pa+i), (void *)(pb+i), (void *)(pc+i));
```

Will produce the following table:

i	pa+i (int)	pb+i (char)	pc+i (double)
0	0x7fff89e611a0	0x7fff89e61190	0x7fff89e61160
1	0x7fff89e611a4	0x7fff89e61191	0x7fff89e61168
2	0x7fff89e611a8	0x7fff89e61192	0x7fff89e61170
3	0x7fff89e611ac	0x7fff89e61193	0x7fff89e61178
4	0x7fff89e611b0	0x7fff89e61194	0x7fff89e61180

Note the increment in the addresses corresponding to `sizeof` the type.

Passing an array to a function

When an **array name** is passed to a **function**

- what is passed is the **location** of the initial element,
- what is passed is **a pointer**.

Note As function *parameter* `char s[]` and `char *s` are **equivalent!**

```
f(int arr[]) { ... }
/* these two are equivalent */
f(int *arr) { ... }
```

- Since a pointer is passed in reality, using the **array notation** can be considered **bad style**[5].
- No matter what notation you use, the function body may **at its convenience** believe that it has been handed either an array or a pointer, and manipulate it accordingly. One can even do both!

[5] See Linus Torvald's rant at https://lkml.org/lkml/2015/9/3/428.

Passing parts of an array to a function

- It is possible to pass a “part of an array” to a function, by passing a pointer to the beginning of the subarray.
- So, as far as `f` is concerned, the fact that the parameter refers to part of a larger array is of no consequence.

Example Pass address of subarray that starts at `a[2]` to the function `f`.

`f(&a[2]);` $\equiv$ `f(a+2);`

Question What is the output of `show(array+3);` ?

```
int array[23];

void show(int a[]) {
   printf("%zu\n", sizeof a);
}
```

Example: Compute string length using a pointer

```
int stringlen(char *s)
{
   int n;

   for (n = 0; *s != '\0'; s++)
      n++;

   return n;
}
```

- Since `s` is a pointer, incrementing it is perfectly legal.
- `s++` has no effect on the character string in the caller function, it merely increments `strlen`'s private copy of the pointer.

Legal calls to `strlen`?

Given `int stringlen(char *s)` as above, which calls will work?

```
int main(void)
{
    char array[] = "hello";

    /* ok */
    printf("%d\n", stringlen(array));

    /* warning we do not yet understand */
    printf("%d\n", stringlen("hello"));

    return 0;
}
```

- We need to elaborate on this! (*cf.* page 32)
- For now, only pass variables declared as `char[]` to `stringlen`.

Comparison of pointers

Pointers may be **compared** under certain circumstances:

- If `p` and `q` point to members of **the same** array, then comparison like `==`, `!=`, `<`, `>=`, *etc.* work properly.
  E.g., `p < q` is true, if `p` points to an earlier member of the array than `q` does.
- The behaviour is **undefined** for arithmetic or comparisons with pointers that do not point to members of the same array.
- There is one exception: The address of the first element past the end of an array can be used in pointer arithmetic.

Pointer subtraction

- If p and q point to elements of the same array and p < q, then q-p+1 is the number of elements from p to q inclusive.

```c
#include <stddef.h>    /* includes type ptrdiff_t */

ptrdiff_t stringlen(char *s)
{
   char *p = s;

   while (*p)  /* i.e., *p != '\0' */
      p++;

   return p - s;
}
```

- p is initialized to s, *i.e.*, point to the first character of the string.
- while loop: examine each char until '\0' is seen.
- Use pointer subtraction to determine string length.

Valid Pointer Operations

Legal pointer operations summarized

- Assignment of pointers of the same type, or `void *`.
- Assigning or comparing to `NULL`.
- Adding or subtracting a pointer and an integer.
- Subtracting or comparing pointers to members of the **same array** (or same memory area as returned by e.g. `malloc`.

Illegal pointer operations

- Multiply, divide, shift, or mask pointers.
- Add `float` or `double` to pointers.
- Assign a pointer of one type to a pointer of another type without cast (exception is `void *`).
- Subtracting or comparing pointers to members of **different arrays**, or not pointing to arrays at all.

The `const` type qualifier

The **keyword** `const` can be used to make a variable **readonly**.

```
const type var;
```

```
type const var;
```

- Both forms above are **equivalent**.
- General rules:
  - If `const` is **next to a type specifier** (*e.g.*, `int`, `double`, …), it applies to that type specifier.
  - Otherwise, it applies to the pointer **asterisk to its left**.

Note The position of `const` relative to an `*` is relevant:

A pointer to a **constant object**.

```
type const * var;
const type * var;
```

You may assign to the pointer, but not to its target.

A **constant pointer** to an object.

```
type * const var;
```

You may assign to the target, but not to the pointer.

- Hint: Read pointer declarations from **right to left**.

Examples

```c
int i;
int const c = 32, d;
int * const p1 = &i, * p2 = &i, * const p3;
int const * p4;
```

- `c` and `d` are constant `int`s.

```c
i = c;   /* ok: copy value from c, and store in i */
c = i;   /* error: assignment of read-only variable c */
d = 23;  /* error: assignment of read-only variable d */
```

- `p1`, `p3` are constant pointers to `int`, **but** `p2` is a pointer to `int`.

```c
p1 = &i;   /* error: assignment of read-only variable p1 */
*p1 = 12;  /* ok: write to the integer, not the pointer! */
p2 = &i;   /* ok: p2 is not const */
```

- `p4` is a pointer to a constant `int`.

```c
p4 = &i;   /* ok: p2 is not const */
*p4 = 34;  /* error: assignment of read-only location *p4 */
i = 99;    /* ok: i is not constant */
```

A function can **promise not to modify** a value passed by reference:

```c
1  #include <stdio.h>
2
3  int nice(int const * x)
4  {
5     /* *x = 3; */  /* causes error */
6     return *x + 2;
7  }
8
9  int sloppy(int * x)
10 {
11    return *x + 2;
12 }
```

```c
13 int main(void)
14 {
15    int i = 12;
16    int const j = 23;
17
18    nice(&i);
19    nice(&j);
20    sloppy(&j);  /* causes warning */
21
22    printf("%d\n", i);
23    return 0;
24 }
```

- Passing a reference to a constant object to a function that does not promise not to modify it, causes a **warning**! (line 20)
- A **string literal** in C is constant, and must not be written to!

```c
int stringlen(char * foo);
stringlen("hello");  /* warning */
```

```c
int stringlen(char const * foo);
stringlen("hello");  /* fine */
```

Cast away `const` — pun intended

- Review the warning issued by line 20 on the previous slide:

```
const2.c:28:2: warning: passing argument 1 of ‘sloppy’ discards ‘const’
qualifier from pointer target type [enabled by default]
sloppy(&j);
```

- If you
  - absolutely must use that function (it may come from a library),
  - and you absolutely know that it will not change the value
  - and you absolutely cannot create a copy and pass that instead,

  then you may cast the type into a non-`const` one:

```c
int sloppy(int * x)
{
   return *x + 2;
}
int modify(int * x)
{
   (*x)++;
   return *x+2;
}
```

```c
int main(void)
{
   int const j = 23;

   sloppy((int*)&j);  /* no warning */
   modify((int*)&j);  /* you're on your own */
   printf("%d\n", j);

   return 0;
}
```

Cast away const — broken promise

- A function may break its promise:

```c
int evil(int const * x)
{
   *(int *)x = 666;
   return *x + 2;
}
```

```c
int main(void)
{
   int const j = 23;
   evil(&j);
   printf("%d\n", j);
   return 0;
}
```

- Writing such functions is a very bad idea:
  - You **break the promise** given in the function's signature!

String literals are constant

In C, a **string literal** is a constant, that you **must not write** to.

- Why? May be shared. Stored in a read-only location (*cf.* later).

Examples

- You must not write to literals.

```
char *s1 = "hello";        /* warning: initialization discards const */
s1[3] = 'X';               /* this will segfault (i.e., access violation) */
```

- You cannot pass literals to functions accepting a non-const.

```
const char *s2 = "hello";  /* correct */
int stringlen(char *s);    /* assume we have that function */
stringlen(s2);             /* warning: discards 'const' qualifier */
stringlen("hello");        /* warning, because the literal is const */
```

- Use `const` to indicate where your functions behave nice.

```
int stringlen2(const char *s);  /* assume we have that function */
stringlen2(s2);                 /* correct */
stringlen2("world");            /* correct */
```

Character pointers & character arrays differ

A `char` **array** initialised from a constant is **writable**!

```
const char *s1 = "hello";      /* from previous slide */

char s3[] = "world";    /* correct: writable array initialized from constant */
stringlen(s3);          /* correct */
stringlen2(s3);         /* correct */
s3[1] = 'X';            /* correct: the array is writable */
s3 = s1;                /* wrong: array name used as l-value */
```

- The array is initialized from a literal!
- The array is writable, the literal is not.

How can we copy strings?

```c
char t[100];  /* target array */
const char *s = "hello world";

int main(void)
{
    /* we are looking for this */
   stringcpy(t, s);
   printf("%s\n", t);

   return 0;
}
```

Function to copy string `s` into array `t`:

```c
void stringcpy(char *t, char const *s)
{
   int i;

   for (i = 0; s[i] != '\0'; i++)
   t[i] = s[i];
   t[i] = '\0';
}
```

Question Why is the `const` *necessary* in the specification of parameter `s`?

String copy using pointers

```
void stringcpy(char *t, char const *s)
{
   while (*s != '\0')
   *t++ = *s++;
   *t = '\0';
}
```

- The value of `*s++` is the character that `s` pointed to before `s` is incremented. (*cf.* page 12)
- The postfix `++` doesn't change `s` until after this character has been fetched.

More on Types

Structures, unions, enumerations
Defining types
Unscrambling C declarations

Type Conversions

Type Conversions

- Type ranking
  - `_Bool` → `char` → `short` → `int` → `long` → `long long` →
  - `float` → `double` → `long double`
- Typing constants
  - `L` (`long`), `LL` (`long long`)
  - `U` (`unsigned`), `UL` (`unsigned long`), `ULL` (`unsigned long long`)
  - `F` (`float`), `L` (`long double`)
- Automatic promotion to (`unsigned`) `int`
  - The signedness of the higher-ranked type takes precedence
  - For equal ranks, `unsigned` takes precedence

Structures

Declaring structures

- A structure allows a group of variables to be accessed via one name.
- To this end, a structure introduces a **new type**.

The definition has four parts:

```
struct tag {
/* list of member declarations */
type name...;
type name...;
} variable...;
```

- the keyword `struct`
- an optional *structure tag*
- brace-enclosed list of declarations for the **members**
- list of variables of the new structure type (optional)

Declaration examples:

```
struct point {
double x, y;
};
/* now “struct tag” serves as type name */
struct point p, q;
```

or equivalent

```
struct {
double x, y;
} p, q;  /* directly name variables
/* But you cannot reuse this struct!
```

Using structures

- A list of constant member values in the right order initialises a structure. Or use individual members by name, in any order.

```
struct point p1 = { 320, 200 };
```

```
struct point p2 = { .x = 320 };
```

- Structures can be assigned as a unit, or be returned from a function.

```
struct point p = q;        /* copy all members */
struct point mkpoint(); /* declares function returning a point structure */
```

- Members can be accessed using name.member

```
struct point center;
printf("%f, %f\n", center.x, center.y);
```

- There is a shortcut for handling pointers to structs: ptr->name

```
struct point origin, *pp;
pp = &origin;                  /* so you can get the address of a structure */
printf("origin is (%f,%f)\n", (*pp).x, (*pp).y);
printf("origin is (%f,%f)\n", pp->x, pp->y);   /* this is equivalent */
```

- Structures can contain other structures

```
struct rect {
struct point ul;
struct point lr;
} square;

square.ul.x = 0; square.ul.y = 1;
square.lr.x = 2; square.lr.y = 0;
```

- Structures can be self-referential **via pointers**.

```
struct tnode {              /* the tree node: */
int value;              /* node label */
struct tnode *left;     /* left child */
struct tnode *right;    /* right child */
};
```

- The **size** of a struct may be *larger* than the sum of its members!

```
struct demo {
int i;
char c;
};
```

```
/* prints 8 on my machine */
printf("%zu\n", sizeof(struct demo));
```

- Structures can be array elements

```
struct point {
int x;
int y;
} points[] = {
{ 0, 1 },
{ 2, 3 },
{ 3, 5 }
};
```

- Structures are passed to functions **by value**!

```
struct point add(struct point p1, struct point p2)
{
p1.x += p2.x;
p1.y += p2.y;

return p1;
}
```

  - The **whole struct** is copied!
  - This also works for the `return` value!

Unions

- A *union* is a **variable** that may hold (at different times) objects of **different types** and sizes.
- Unions provide a way to manipulate different kinds of data in a **single area of storage**.

The syntax is similar to structures:

```
union tag {
 /* list of member declarations */
type name...;
type name...;
} variable...;
```

- the keyword `union`
- an optional *union tag*
- brace-enclosed list of declarations for the **members**
- list of variables of the new union type (optional)

- Union variables will be large enough to hold the **largest** of the member types. (the specific size is implementation-dependent)
- It is the programmer's responsibility to keep track of which member currently holds a value. **Only one** can be used at any time.

```c
union demo {
int i;
double d;
char c;
};

union demo u;

printf("size: %zu\n", sizeof(u));

u.i = 23;    /* now u.d and u.c contain garbage! */
printf("u.i: %-16d   u.d: %-16e   u.c: '%c'\n", u.i, u.d, u.c);

u.d = 4.2;   /* now u.i and u.c contain garbage! */
printf("u.i: %-16d   u.d: %-16e   u.c: '%c'\n", u.i, u.d, u.c);

u.c = 'X';   /* now u.i and u.d contain garbage! */
printf("u.i: %-16d   u.d: %-16e   u.c: '%c'\n", u.i, u.d, u.c);
```

```
$ ./a.out
size: 8
u.i: 23                 u.d: 6.952931e-310      u.c: ''
u.i: -858993459         u.d: 4.200000e+00       u.c: '□'
u.i: -858993576         u.d: 4.200000e+00       u.c: 'X'
```

Use case 1: Saving space

- Usually occur as a part of a larger struct that also has implicit or explicit information about the data.
- Used to save space.

Example Zoological information on certain species. First attempt:

```
struct creature {
char has_backbone;
char has_fur;
short num_of_legs_in_excess_of_4;
};
```

However...

- All creatures are either vertebrate or invertebrate.
- Only vertebrates have fur and only invertebrates have more than four legs.
- Nothing has more than four legs and fur.

That is why...

```
union secondary_characteristics {
char has_fur;
short num_of_legs_in_excess_of_4;
};
struct creature {
char has_backbone; /* indicates valid union field! */
union secondary_characteristics form;
};

struct creature naked_mole_rat = {
.has_backbone = 'y',
.form.has_fur = 'n'      /* Note the .form prefix */
};
```

Use case 2: Data interpretation

```
union bits32_tag {
int whole;  /* one 32-bit value */
char byte[4];  /* four 8-bit bytes */
} value;
```

- Take the whole with `value.whole`
- Take 3rd byte with `value.byte[2]`

Notes

- You need to check your compiler's documentation to make proper use of this!
- Generally, structs are about one hundred times more common than unions.

Enumerations

- Enumerations provide a convenient way to associate **constant integer** values with **names**.
- An alternative to `#define` with the advantage that the values can be generated automatically.
- A compiler can warn about missing `case`s in `switch` statements over an enumeration.
- A **debugger** may also be able to print values of enumeration variables in symbolic form.

Definition syntax:

```
enum tag {
name,
name = val,
...
} variable...;
```

- the keyword `enum`
- an optional *enumeration tag*
- brace-enclosed list of *members*, sep. by **comma**, with optional assignment
- optional list of variables of the new type

- Declaring an enumeration is similar to `enum` and `struct`.

```
enum answer { no, yes }; /* definition */
enum answer x; /* declaration */
```

```
/* shorthand */
enum { no, yes } x;
```

- An enumeration is a list of **constant integer values**.

```
enum answer { no, yes };

enum answer x;
int i;

x = no;
x = 42;      /* x is just an int */
i = yes;
no = 23;     /* invalid — not an lvalue! */
```

- If not assigned explicitly, the names are assigned consecutive integer constants, starting from 0.
- Enumeration continues from an explicit assignment.

```
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
JUL, AUG, SEP, OCT, NOV, DEC };
/* FEB is 2, MAR is 3 ... */
```

Functions and function pointers

- A **function** itself *is not* a variable.
- But it is possible to define **pointers** to functions.
- Can be **assigned**, placed in **arrays**, **passed** to/**returned** from functions.

Syntax step-by-step examples of declarations:

- `int fun(char c, double x);` nothing new!
  - The expression `fun('Q', 3.14)` is of type `int`.
- `int *fun(char c, double x);` nothing new!
  - The expression `*fun('Q', 3.14)` is of type `int`.
  - Dereferencing `fun('Q', 3.14)` is of type `int`.
- `int (*fun)(char c, double x);`
  - The expression `(*fun)('Q', 3.14);` is of type `int`.
  - Dereferencing `fun`, and applying the result to `'Q',3.14`, is of type `int`.

$\Rightarrow$ We have just dereferenced a function!

Example

```
size_t strlen(const char *s);    /* available with #include <string.h> */

size_t (*fp)(const char *);
fp = &strlen;
printf("result = %zu\n", (*fp)("hello world"));
```

- This can be abbreviated:

```
printf("result = %zu\n", fp("hello world"));
```

- Nicer with `typedef`

```
typedef size_t (*func)(const char *);
func fp = &strlen;
```

Note Function pointers are heavily used in the real world! *E.g.*,

- pass a comparing function to a queue datastructure;
- installing signal handlers (*cf.* OS lecture, and later in this course); and
- abstractions (syscall interface, subclasses, VFS, ...).

Question Can you read `int (*(*f)(void))[2]` ?