

# **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?

```

1 int main(void)
2 {
3     char c = 'B';
4     unsigned int i = 0xdeadbeef;
5
6     return 0;
7 }

```

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

`sizeof(char)` = 1 (by definition;  $\geq 8$  bits)

`sizeof(int)` = 4 (this may vary;  $\geq 16$  bits)

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

# 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<sup>1</sup> are documented in `<limits.h>` and `<float.h>`. Additional limits are specified in `<stdint.h>`<sup>2</sup>

<sup>1</sup>ISO C99 : 7.10/5.2.4.2 : Numerical limits

<sup>2</sup>ISO C99 : 7.18 : Integer Types (see also [https://en.wikipedia.org/wiki/C\\_data\\_types#Basic\\_types](https://en.wikipedia.org/wiki/C_data_types#Basic_types))

## On my machine

<code>char</code>	1
<code>short int</code>	2
<code>int</code>	4
<code>long int</code>	8
<code>long long int</code>	8
<code>float</code>	4
<code>double</code>	8
<code>long double</code>	16
<code>void *</code>	8

# 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;`  $\Rightarrow$  copy 1 byte.
- `double x, y;`  $\Rightarrow$  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: `&`

```

1 int main(void)
2 {
3     unsigned int i, *p;
4
5     i = 0xdeadbeef;
6     p = &i; /* p points to i */
7
8     (void)p;
9     return 0;
10 }

```

- **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	integer
	0xffffd861	0xbe	
	0xffffd862	0xad	
	0xffffd863	0xde	
p	0xffffd864	0x60	pointer
	0xffffd865	0xd8	
	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 *) 0x7fffffff6d4
(gdb) x/4b $2 // examine 4 bytes at the address
0x7fffffff6d4: 0xef 0xbe 0xad 0xde
(gdb) p &p // where is the pointer
$4 = (unsigned int **) 0x7fffffff6d8
(gdb) x/8b $4 // what does it look like? my pointers are 64 bit wide
0x7fffffff6d8: 0xd4 0xe6 0xff 0xff 0xff 0x7f 0x00 0x00
```



# The dereferencing operator: \*

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int i = 23,
6     *p = NULL; /* initialize pointer p, not the integer */
7
8     p = &i; /* copy address of i into p */
9     i = 42; /* change value of i */
10
11     printf("%d\n", *p); /* get what p points to */
12
13     return 0;
14 }
```

## Output:

```
1 $ ./a.out
2 42
```

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

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

- Cannot be dereferenced  $\Rightarrow$  typecast required.

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

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

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

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

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

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

# Watch out!

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int    *ip, i = 23;
6      double *dp, d = 3.14159;
7      void   *p;
8
9      p = &i;
10     ip = p;
11     printf("%p %d\n", p, *ip);
12
13     p = &d;
14     dp = p;
15     printf("%p %f\n", p, *dp);
16
17     ip = p;
18     printf("%p %d\n", p, *ip);
19
20     return 0;
21 }

```

## 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<sup>3</sup> 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...)

---

<sup>3</sup>the returned value is unused

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

# Call by reference

```
1 #include <stdio.h>
2
3 void swap(int *px, int *py)
4 {
5     int tmp;
6
7     tmp = *px;
8     *px = *py;
9     *py = tmp;
10 }
11
12 int main(void)
13 {
14     int x = 23, y = 42;
15
16     printf("x=%d, y=%d\n", x, y);
17     swap(&x, &y);
18     printf("x=%d, y=%d\n", x, y);
19
20     return 0;
21 }
```

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

## 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.

```
1 $ ./a.out <<<'0 1 -12 12345'  
2 0  
3 1  
4 -12  
5 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**<sup>4</sup> The values are passed back through **seperate paths**.

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

---

<sup>4</sup>This approach is used often in C



# Using `getint()`

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

```
31 int main(void)
32 {
33     int i;
34
35     while (getint(&i))
36         printf("%d\n", i);
37
38     return 0;
39 }
```

- 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()

```
6 int getint(int *p)      /* Return 0 on EOF, 1 otherwise. Store int at passed address */
7 {
8     int c, sign = 1;
9
10    while (isspace(c = getchar())) /* cf. isspace(3) */
11        /* skip white space */
12
13    if (c == '-') { /* store optional minus sign */
14        sign = -1;
15        c = getchar();
16    }
17
18    if (!isdigit(c)) /* pathological case */
19        return 0;
20
21    *p = c - '0'; /* parse the digits */
22    while (isdigit(c = getchar()))
23        *p = 10 * *p + c - '0';
24
25    *p *= sign; /* apply sign */
26
27    return 1;
28 }
```

# 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**.

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

a: 

--	--	--	--	--

a[0] a[1] a[2] a[3] a[4]

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

pa



a: 

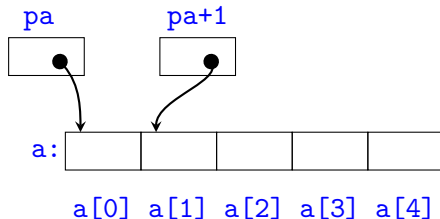
--	--	--	--	--

a[0] a[1] a[2] a[3] a[4]

- 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!

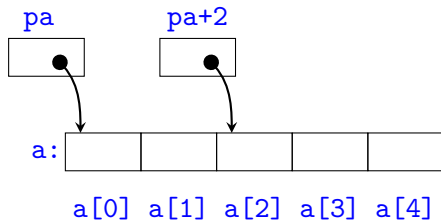


**In general** If `pa` points to any array element, then

- `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:

$$p[i] \equiv *(p + i)$$

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

```

1 int    a[5] = { 0 };           int    *pa = a;
2 char   b[5] = { '\0' };       char    *pb = b;
3 double c[5] = { 0.0 };        double *pc = c;
4
5 for (int i = 0; i < 5; i++)
6     printf("%d    %p    %p    %p\n",
7         i, (void *)(&a[i]), (void *)(&b[i]), (void *)(&c[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**<sup>5</sup>.
- 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!

---

<sup>5</sup>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]);  ≡  f(a+2);
```

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

```
1 int array[23];  
2  
3 void show(int a[]) {  
4     printf("%zu\n", sizeof a);  
5 }
```

## Example: Compute string length using a pointer

```
5 int strlen(char *s)
6 {
7     int n;
8
9     for (n = 0; *s != '\0'; s++)
10         n++;
11
12     return n;
13 }
```

- 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 strlen(char *s)` as above, which calls will work?

```
17 int main(void)
18 {
19     char array[] = "hello";
20
21     /* ok */
22     printf("%d\n", strlen(array));
23
24     /* warning we do not yet understand */
25     printf("%d\n", strlen("hello"));
26
27     return 0;
28 }
```

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

# 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.

```
1 #include <stddef.h>    /* includes type ptrdiff_t */
2
3 ptrdiff_t stringlen(char *s)
4 {
5     char *p = s;
6
7     while (*p) /* i.e., *p != '\0' */
8         p++;
9
10    return p - s;
11 }
```

- $p$  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.

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

A **constant pointer** to an object.

```
type * const var;
```

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



# Examples

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

- **c** and **d** are constant **ints**.

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

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

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

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

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

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

```

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 }

```

```

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!

```

1 int stringlen(char * foo);
2 stringlen("hello"); /* warning */

```

```

1 int stringlen(char const * foo);
2 stringlen("hello"); /* fine */

```

# Cast away `const` — pun intended

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

```
1 const2.c:28:2: warning: passing argument 1 of 'sloppy' discards 'const'  
2   qualifier from pointer target type [enabled by default]  
3   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:

```
1 int sloppy(int * x)  
2 {  
3     return *x + 2;  
4 }  
5 int modify(int * x)  
6 {  
7     (*x)++;  
8     return *x+2;  
9 }
```

```
1 int main(void)  
2 {  
3     int const j = 23;  
4  
5     sloppy((int*)&j); /* no warning */  
6     modify((int*)&j); /* you're on your own */  
7     printf("%d\n", j);  
8  
9     return 0;  
10 }
```

# Cast away `const` — broken promise

- A function may break its promise:

```
1 int evil(int const * x)
2 {
3     *(int *)x = 666;
4     return *x + 2;
5 }
```

```
6 int main(void)
7 {
8     int const j = 23;
9     evil(&j);
10    printf("%d\n", j);
11    return 0;
12 }
```

- 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.

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

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

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

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

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

# Character pointers & character arrays differ

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

```
1 const char *s1 = "hello";      /* from previous slide */
2
3 char s3[] = "world";          /* correct: writable array initialized from constant */
4 stringlen(s3);                /* correct */
5 stringlen2(s3);               /* correct */
6 s3[1] = 'X';                  /* correct: the array is writable */
7 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?

```
12 char t[100]; /* target array */
13 const char *s = "hello world";
14
15 int main(void)
16 {
17     /* we are looking for this */
18     strcpy(t, s);
19     printf("%s\n", t);
20
21     return 0;
22 }
```

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

```
3 void strcpy(char *t, char const *s)
4 {
5     int i;
6
7     for (i = 0; s[i] != '\0'; i++)
8         t[i] = s[i];
9     t[i] = '\0';
10 }
```

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

# String copy using pointers

```
3 void strcpy(char *t, char const *s)
4 {
5     while (*s != '\0')
6         *t++ = *s++;
7     *t = '\0';
8 }
```

- 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`  $\rightarrow$  `char`  $\rightarrow$  `short`  $\rightarrow$  `int`  $\rightarrow$  `long`  $\rightarrow$  `long long`  
 $\rightarrow$
- `float`  $\rightarrow$  `double`  $\rightarrow$  `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:

```

1 struct tag {
2     /* list of member declarations */
3     type name...;
4     type name...;
5 } 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:

```

1 struct point {
2     double x, y;
3 };
4 /* now "struct tag" serves as type name */
5 struct point p, q;

```

or equivalent

```

1 struct {
2     double x, y;
3 } 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.

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

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

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

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

- Members can be accessed using [name.member](#)

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

- There is a shortcut for handling pointers to structs: [ptr->name](#)

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

## ■ Structures can contain other structures

```
1 struct rect {  
2     struct point ul;  
3     struct point lr;  
4 } square;  
5  
6 square.ul.x = 0; square.ul.y = 1;  
7 square.lr.x = 2; square.lr.y = 0;
```

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

```
1 struct tnode {                /* the tree node: */  
2     int value;                /* node label */  
3     struct tnode *left;      /* left child */  
4     struct tnode *right;     /* right child */  
5 };
```

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

```
1 struct demo {  
2     int i;  
3     char c;  
4 };
```

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

## ■ Structures can be array elements

```
1 struct point {  
2   int x;  
3   int y;  
4 } points[] = {  
5   { 0, 1 },  
6   { 2, 3 },  
7   { 3, 5 }  
8 };
```

## ■ Structures are passed to functions **by value!**

```
1 struct point add(struct point p1, struct point p2)  
2 {  
3   p1.x += p2.x;  
4   p1.y += p2.y;  
5  
6   return p1;  
7 }
```

- 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:

```
1 union tag {  
2 /* list of member declarations */  
3 type name...;  
4 type name...;  
5 } 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.

```
1 union demo {
2   int i;
3   double d;
4   char c;
5 };
6
7 union demo u;
8
9 printf("size: %zu\n", sizeof(u));
10
11 u.i = 23; /* now u.d and u.c contain garbage! */
12 printf("u.i: %-16d    u.d: %-16e    u.c: '%c'\n", u.i, u.d, u.c);
13
14 u.d = 4.2; /* now u.i and u.c contain garbage! */
15 printf("u.i: %-16d    u.d: %-16e    u.c: '%c'\n", u.i, u.d, u.c);
16
17 u.c = 'X'; /* now u.i and u.d contain garbage! */
18 printf("u.i: %-16d    u.d: %-16e    u.c: '%c'\n", u.i, u.d, u.c);
```

```
1 $ ./a.out
2 size: 8
3 u.i: 23                u.d: 6.952931e-310      u.c: ''
4 u.i: -858993459        u.d: 4.200000e+00      u.c: '□'
5 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:

```
1 struct creature {  
2   char has_backbone;  
3   char has_fur;  
4   short num_of_legs_in_excess_of_4;  
5 };
```

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...

```
1 union secondary_characteristics {  
2   char has_fur;  
3   short num_of_legs_in_excess_of_4;  
4 };  
5 struct creature {  
6   char has_backbone; /* indicates valid union field! */  
7   union secondary_characteristics form;  
8 };  
9  
10 struct creature naked_mole_rat = {  
11   .has_backbone = 'y',  
12   .form.has_fur = 'n'      /* Note the .form prefix */  
13 };
```

## Use case 2: Data interpretation

```
1 union bits32_tag {  
2   int whole; /* one 32-bit value */  
3   char byte[4]; /* four 8-bit bytes */  
4 } 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 `cases` in `switch` statements over an enumeration.
- A **debugger** may also be able to print values of enumeration variables in symbolic form.

Definition syntax:

```
1 enum tag {  
2 name,  
3 name = val,  
4 ...  
5 } 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`.

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

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

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

```
1 enum answer { no, yes };  
2  
3 enum answer x;  
4 int i;  
5  
6 x = no;  
7 x = 42; /* x is just an int */  
8 i = yes;  
9 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.

```
1 enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,  
2 JUL, AUG, SEP, OCT, NOV, DEC };  
3 /* 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`.
- ⇒ We have just dereferenced a function!

## Example

```
1 size_t strlen(const char *s);    /* available with #include <string.h> */
2
3 size_t (*fp)(const char *);
4 fp = &strlen;
5 printf("result = %zu\n", (*fp)("hello world"));
```

- This can be abbreviated:

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

- Nicer with `typedef`

```
2 typedef size_t (*func)(const char *);
3 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] ?`