Printing with `printf`

specifier	print as . . .
`%d`	decimal integer
`%6d`	decimal, at least 6 characters wide
`%f`	floating point
`%6f`	floating point, at least 6 characters wide
`%.2f`	floating point, 2 characters after decimal point
`%6.2f`	floating point, at least 6 wide and 2 after decimal point

- Further `printf`(3) recognizes `%o` for octal, `%x` for hexadecimal, `%c` for character, `%s` for string, `%p` for address (pointer), ...
- ISO C: 7.19.6 : Formatted input/output functions

The `for` loop

Fahrenheit-Celsius v2

```c
/* print fahrenheit-celsius table for fahrenheit = 0, 20, ..., 300 */

#include <stdio.h>

int main(void)
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr,
               (5.0/9.0)*(fahr-32));

    return 0;
}
```

Running:

```
$ ./a.out
  0  -17.8
 20   -6.7
 40    4.4
 60   15.6
 80   26.7
100   37.8
120   48.9
140   60.0
160   71.1
180   82.2
200   93.3
220  104.4
240  115.6
260  126.7
280  137.8
300  148.9
```

Symbolic constants

Fahrenheit-Celsius final

- Bad practice to bury “magic numbers” in a program
- Convey little information, hard to change in a systematic way
- A `#define` line defines a *symbolic name*

```
1  /* print fahrenheit-celsius table for fahrenheit = 0, 20, ..., 300 */
2
3  #include <stdio.h>
4
5  #define LOWER 0    /* lower limit of table */
6  #define UPPER 300  /* upper limit */
7  #define STEP  20   /* step size */
8
9  int main(void)
10 {
11     for (int fahr = LOWER; fahr <= UPPER; fahr += STEP)
12         printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
13
14     return 0;
15 }
```

1.4 Character input and output

- Standard library provides (among the others) `getchar`(3) and `putchar`(3).

```
#include <stdio.h>

int getchar(void);
int putchar(int c);
```

- `putchar`(3) prints a character to *stdout* each time it is called.
- `getchar`(3) reads the next input byte from *stdin* stream

Question Why does `getchar` return an `int` instead of `char`?

Answer

- When there is no more input, `getchar` returns a distinctive value called `EOF` (end of file; e symbolic name, defined in `<stdio.h>`), which cannot be confused with data.
- The return type must be big enough to hold `EOF` in addition to any possible `char`.

File Copying

Given `getchar` and `putchar` we can write a surprising amount of useful code without knowing anything more about input and output.

Algo Copying input to output one character at a time

read a character
while character is not end-of-file indicator **do**
  output the character just read
  read a character
end while

File Copying, v1

```c
#include <stdio.h>

/* copy input to output, v1 */
int main(void)
{
    int c = getchar();

    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
    return 0;
}
```

File Copying, v2

- An assignment, such as `c = getchar()` is an expression and has a value (value of the left hand side after the assignment)
- An assignment can appear as part of a larger expression

```
1  #include <stdio.h>
2
3  /* copy input to output, v2 */
4  int main(void)
5  {
6      int c;
7
8      while ((c = getchar()) != EOF)
9          putchar(c);
10
11     return 0;
12 }
```

Character Counting, v1

```c
#include <stdio.h>

/* count characters in input, v1 */
int main(void)
{
    long nc = 0;

    while (getchar() != EOF)
        nc = nc + 1;
    printf("%ld\n", nc);

    return 0;
}
```

Character Counting, v2

```c
#include <stdio.h>

/* count characters in input, v2 */
int main(void)
{
    long nc = 0;

    for (; getchar() != EOF; nc = nc + 1)
        ;
    printf("%ld\n", nc);

    return 0;
}
```

1.5 Functions

power(m,n)

- So far only `printf`(3), `getchar`(3), and `putchar`(3)
- Implement `power(m,n)` to raise an integer *m* to the power[13] of *n*.

A function definition has the form:

```
type name( type parameter [, ...] )      /* or: name(void) */
{
        declarations
        statements
}
```

[13] Only handles positive powers of small integers, in real life take `pow`(3)

```c
#include <stdio.h>

/* power: raise base to n-th power; n >= 0 */
int power(int base, int n)
{
    int i, p;

    p = 1;
    for (i = 0; i < n; ++i)
        p = p * base;
    return p;
}


/* test power function */
int main(void)
{
    int i;

    for (i = 0; i < 10; i++)
        printf("%d  %3d  %6d\n", i, power(2, i), power(-3, i));
    return 0;
}
```

Function Terminology

- A **function definition** gives signature and implementation:

```
4  int power(int base, int n)
5  {
6      int i, p;
7
8      p = 1;
9      for (i = 0; i < n; ++i)
10         p = p * base;
11     return p;
12 }
```

  - A **parameter** is a variable named in the argument list, *e.g.*, `base`, `n`.
  - An **argument** is a value used in a call of the function.

- A **function declaration** omits the implementation:

```
int power(int base, int n);  /* no body! */
```

  - A function must be declared *before* it can be used!
  - A definition also declares a function.
  - We will not need to write declarations for some time...

1.6 Call by value, call by reference

In C, all function arguments are passed **by value**

- The called function is given the values of its arguments in **temporary variables** (lifetime of function's execution) rather than the originals.
- The callee **cannot directly alter** a variable in the calling function.

Call **by reference** is possible

- by passing the **address** of a variable (*aka.* a pointer).
- The calle can access the variable *indirectly* by **dereferencing** the address.
- The pointer itself is passed by value.
- We will discuss pointers in more detail at a later point.

1.7 Arrays

One Dimensional Arrays

Syntax: `memberType arrayName[ numberOfMembers ];`

- Most simple:

```
int a[2];   /* at this point, the contents are undefined! */
a[0] = 23;  /* store 23 in 1st cell. */
a[1] = 42;
```

- Shortcut:

```
int a[2] = {23, 42};   /* initialize right away */
```

- Even shorter:

```
int a[] = {23, 42};    /* Compiler figures out size of array. */
```

- If not all items are given, the rest is initialised to 0.

```
int a[8] = {23, 42};   /* is the same as */
int a[] = {23, 42, 0, 0, 0, 0, 0, 0};
```

- Use `for` loop to initialize bigger arrays, or `memset`(3) (*cf.* later).

Multidimensional arrays

- Most simple:

```
int a[2][3];        /* at this point, the contents are undefined */
a[1][2] = 52;       /* assign to 3rd cell in 2nd array */
```

- Classic:

```
int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

- Shortcut:

```
int a[][3] = {{1, 2, 3}, {4, 5, 6}};
```

You may omit *only* the most significant (first, *i.e.*, outer) dimension!

- Stored in memory linearly, *i.e.*:

1	2	3	4	5	6

- Use `for` loop to initialize bigger arrays, or memset(3) (*cf.* later)
- If not all items are given, the rest is initialised to 0.

```
int a[3][4] = { {1,2}, {3} };   /* is the same as */
int a[][4] = { {1, 2, 0, 0}, {3, 0, 0, 0}, {0, 0, 0, 0} };
```

Counting digits, white spaces, and the rest

Count the number of occurrences of each digit, of white space characters (blank, tab, newline), and all other characters.

Intended usage:

```
$ ./a.out < count_digits.c
digits: 10 3 0 0 0 0 0 0 0 1, white space: 122, other: 360
```

Coding conventions

```
1  #include <stdio.h>  /* count digits, white space, others */
2  int main(void){
3  int c, nwhite, nother, ndigit[10];
4  nwhite = nother = 0;
5  for (int i = 0; i < 10; ++i)
6  ndigit[i] = 0;
7  while ((c = getchar()) != EOF)
8  if (c >= '0' && c <= '9') ++ndigit[c-'0']; else
9  if (c == ' ' || c == '\n' || c == '\t') ++nwhite;
10 else ++nother; printf("digits:");
11 for (int i = 0; i < 10; ++i) printf(" %d", ndigit[i]);
12     printf(", white space: %d, other: %d\n", nwhite, nother);
13 return 0;
14 }
```

- Coding conventions make your life harder only once.
- Ugly code sucks every time you read it.
- See examples[14] at the International Obfuscated C Code Contest.

[14] http://www.ioccc.org

```c
#include <stdio.h>

/* count digits, white space, others */
int main(void)
{
    int c, nwhite, nother, ndigit[10];

    nwhite = nother = 0;
    for (int i = 0; i < 10; ++i)
        ndigit[i] = 0;

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;

    printf("digits:");
    for (int i = 0; i < 10; ++i)
        printf(" %d", ndigit[i]);
    printf(", white space: %d, other: %d\n", nwhite, nother);

    return 0;
}
```

Fixed- and variable-length arrays

- C90 allowes only **constant**[15] **expressions** as array dimensions.

```
#define SIZE 1024
int a[42 * SIZE];
```

- In C99, **variable-length arrays** (VLAs) have been introduced.
  - They **cannot be initialised** in their declaration.
  - **Caution:** VLAs are rather tricky, and have a bunch of interesting consequences. You will not need them for your exercises.
  - You **cannot change** the size of a VLA once it is declared (*i.e.*, they are not *dynamic*).

```
1  #include <stdio.h>
2
3  int func(int c)
4  {
5      /* This is a conditional expression: */
6      return c < 10 ? 10 : c;
7  }
8
9  int main(void)
10 {
11     /* Bounds only known at runtime! */
12     int a[func(getchar())];
13
14     a[2] = 3;
15     printf("%d\n", a[2]);
16     return 0;
17 }
```

[15] *i.e.*, can be computed at compile-time by the compiler

Passing arrays to functions

An array is passed **by reference** to a function.

- This is **not** the entire truth! (We'll need pointers to understand the gory details)
- Careful: The callee may **modify** the array contents.

```c
#include <stdio.h>

void set(int a[], int i, int v)
{
    a[i] = v;
}

int main(void)
{
    int a[5] = { 0 };      /* What does this do? */

    set(a, 2, 23);
    set(a, 0, 42);

    for (int i = 0; i < 5; i++)
        printf("%d\n", a[i]);

    return 0;
}
```

Output

```
$ ./a.out
42
0
23
0
0
```

1.8 Character arrays

Definition A **string** is an array of characters terminated with a `'\0'` character (nul; numerical value is zero).

Yes, that is *nul*, with only one ℓ

- So is `"hello\n"` is stored as

`h`	`e`	`l`	`l`	`o`	`\n`	`\0`

- A string containing n characters requires $n + 1$ memory!
- A string does not know its own length.

Note You may have an **array of characters**, with none of them being nul.

- Perfectly valid, but **not a string!**
- String manipulating functions probably **will fail** on that data!

Initialization of character arrays

- Character by character:

```
char str[3];
str[0] = 'o';
str[1] = 'k';
str[2] = '\0';
```

- Shorter:

```
char str[] = {'o', 'k', '\0'};
```

- Initialising from a string literal:

```
char str[20] = "ok";          /* str[2] and onwards are automatically assigned '\0' */
```

- Without giving the dimension:

```
char str[] = "ok";                                  /* The dimension will be... What? */
```

Arrays of character arrays

- Initialised from string literals:

```
1 char arr[3][12]= { "University",
2                    "of",
3                    "Konstanz" };
```

- You are only allowed to omit the **outermost** dimension:

```
1 char arr[][12]= { "University",
2                   "of",
3                   "Konstanz" };
```

Question: How much memory does `arr` use?

Find the longest line

`longline.c` – reads a set of text lines and prints the longest[16] one.

Program outline:

while there is another line **do**
  if it's longer than the previous longest line **then**
    save it
    save its length
  end if
end while
print the longest line

[16] we assume an upper limit, say 1000 characters

Splitting the program into functions

The program divides naturally into pieces

- Function `getline` fetches the next line of input
  - It needs to signal end-of-file
  - We let it return the length of the line, or zero on `EOF`
  - Zero is appropriate because it is never a valid line length

    Since a line, by definition, ends in `'\n'`.
- Function `copy` copies a line to a safe place
- Function `main` to control `getline` and `copy`

main

```
1  #include <stdio.h>
2  #define MAXLINE 1000   /* maximum input line size */
```

...two helper functions are on the following slides...

```
28 /* print longest input line */
29 int main(void)
30 {
31     int len;                /* current line length */
32     int max;                /* maximum length seen so far */
33     char line[MAXLINE];     /* current input line */
34     char longest[MAXLINE];  /* longest line saved here */
35
36     max = 0;
37     while ((len = getline(line, MAXLINE)) > 0)
38         if (len > max) {
39             max = len;
40             copy(longest, line);
41         }
42     if (max > 0)     /* there was a line */
43         printf("%s", longest);
44     return 0;
45 }
```

Getting a line

```
4  /* getline: read a line into buf, return length */
5  int getline(char buf[], int lim)
6  {
7      int c, i;
8
9      for (i = 0;
10          i < lim-1 && (c = getchar()) != EOF && c != '\n';
11          i++)
12         buf[i] = (char)c;
13     if (c == '\n') {
14         buf[i] = (char)c;
15         i++;
16     }
17     buf[i] = '\0';
18     return i;
19 }
```

- `getline` adds `'\0'` (the *null character* nul; value is zero) at the end of the array to mark the end of the string
- returns the length of the string including newline

Copy a string

```
21 /* copy: copy 'from' into 'to'; assume 'to' is big enough */
22 void copy(char to[], char from[])
23 {
24     for (int i = 0; (to[i] = from[i]) != '\0'; i++)
25         ;
26 }
```

- `copy` does not return a value, `void` explicitly states this
- `copy` is used for its side-effect.

Testing

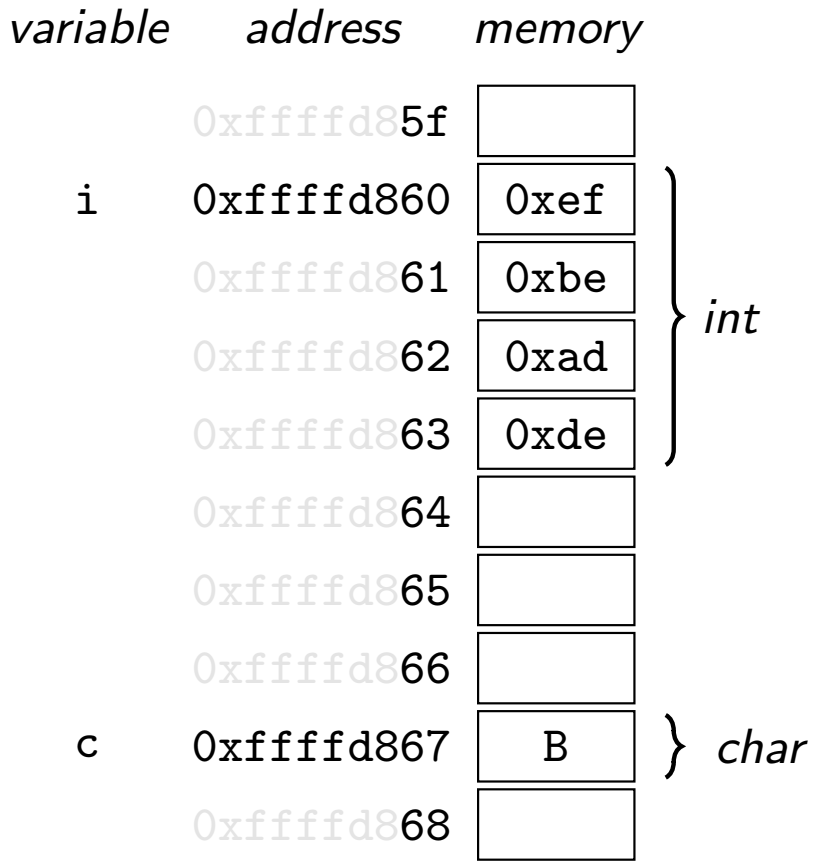
```
1 $ ./a.out <longline.c
2 /* copy: copy 'from' into 'to'; assume 'to' is big enough */
```

2
Pointers

2.1 Memory is just a sequence of bytes

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

    return 0;
}
```



- 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)
`sizeof(int)`	=	4	(this may vary)

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:
  - `short` and `int` are *at least* 16 bit
  - `long` is *at least* 32 bit
  - `short` is no longer than `int`, `int` is no longer than `long`
- Can be obtained with the `sizeof` operator.
- Numerical limits[17] are documented in `<limits.h>` and `<float.h>`. Additional limits are specified in `<stdint.h>`[18]

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

[17] ISO C99 : 7.10/5.2.4.2 : Numerical limits
[18] ISO C99 : 7.18 : Integer 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.

2.2 Introduction to pointers

- A **pointer** is just a variable that contains a memory address.
- Size of a pointer is 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 are exceptions.

- **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.*,

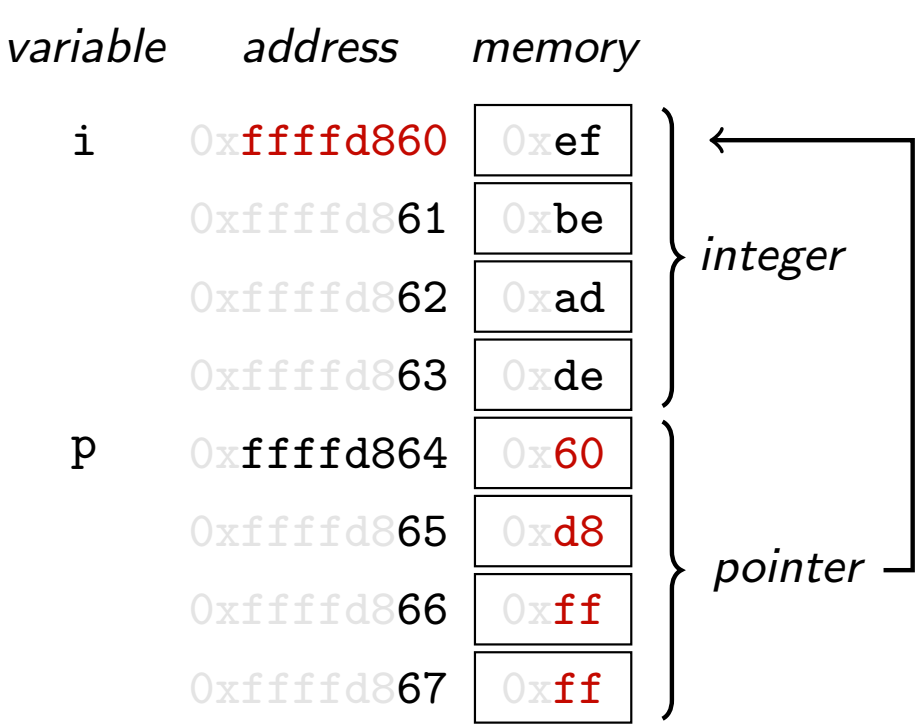
```
int *p, i, *q;     /* p, q are pointers, i is an int */
char c, *r;        /* r is a pointer, c is a char */
int* bad, style;   /* pun intended: discuss type of style */
```

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, ...

Exploring memory

We can actually observe this...

Note that this example was observerd on a 64bit 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**

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

- Cannot be dereferenced ⇒ 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.

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

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

```
7 #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?

2.3 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[19] 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...)

[19] the returned value is unused

Operator precedence rules

From highest (top) to lowest (bottom)

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	`?:`
2	right	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, `|=`
2	left	`,`

But `a?b,c:d` is parsed as `a?(b,c):d`

2.4 Call by reference

Passing pointers to functions

```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[20] The values are passed back through **seperate paths**.

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

[20] 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 }
```

2.5 Pointer arithmetics

- 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 */
```

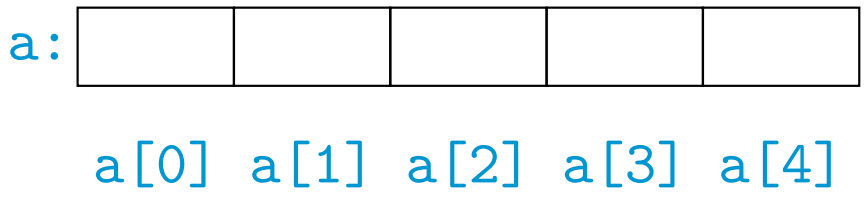
- ...a literal string initializer for a charachter array, the array is initialised with the string.

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

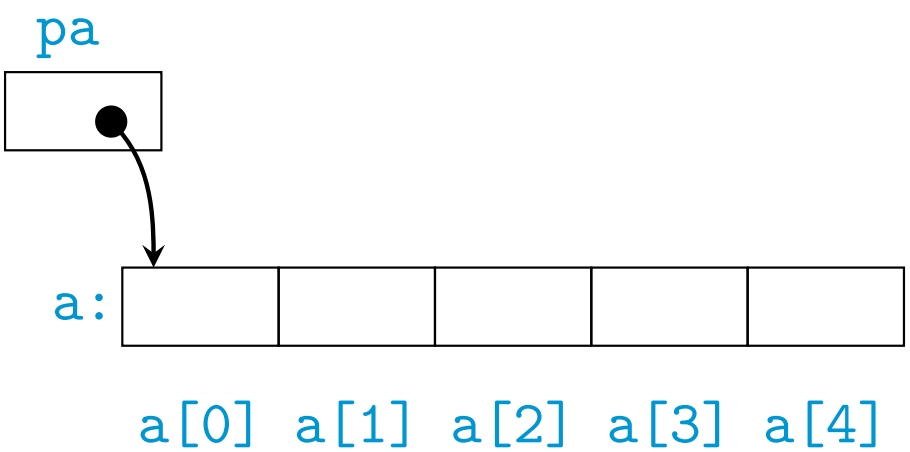
Pointer into array

- 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;
```



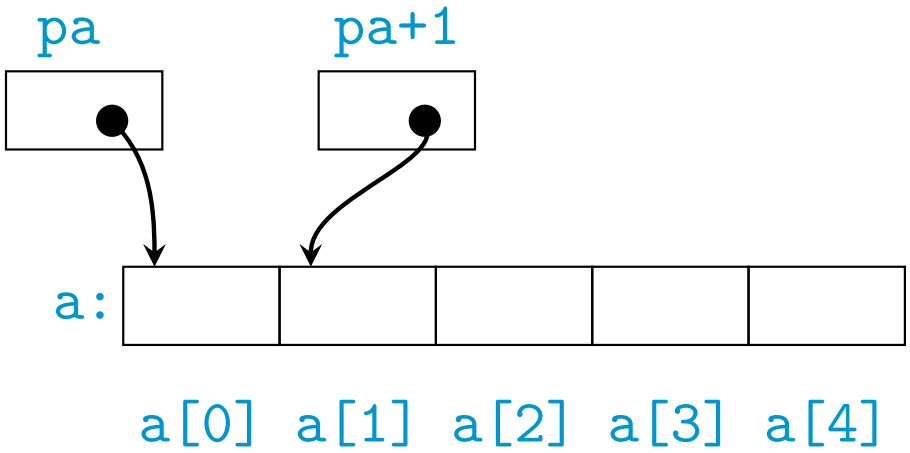

```
4 pa = &a[0]; /* same as pa = a */
5 x = *pa;
```



- 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!

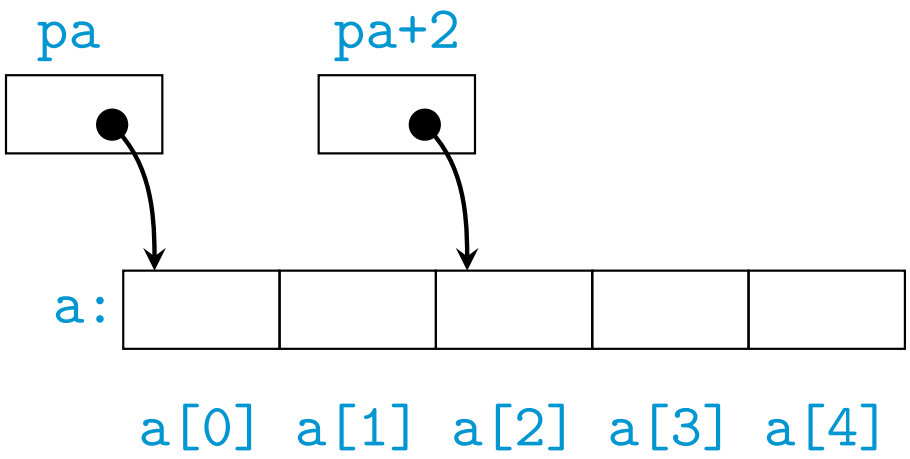


In general If `pa` points to any array element, then

- `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

```
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 *)(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.

Indexing Backwards

- With pointers into arrays we can use **pointer arithmetic** to access nearby cells of the array.
- If we are sure that an element exists, it is also possible to **index backwards** in an array `p[-1]`, `p[-2]`, ...
- This refers to **objects before** what `p` points to.
- Illegal to refer to objects that are not within the **array bounds**, but no one will check this for you. $\Rightarrow$ **Be careful**.

```
p[-3]   ≡   *(p + -3)   ≡   *(p - 3)
```

Warning It is **undefined** what happens, if you *calculate* an address outside the array! (Exception: Just right behind the last element is ok).

2.6 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**[21].
- 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!

[21] 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);`

- But recall backwards indexing (*cf.* page 78)!

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 }
```