F28HS2 Hardware-Software Interface

Lecture 4 Programming in C - 4

String

- array of `char`
- last char is `'\0'`
- no length information

"*characters*"

- string constant
- allocates space for *characters* ending with `'\0'`
- sets each byte to each character
- returns address of first character

String variables

`char` *name* `[`*int*`];`

- allocates space for *int* characters
- cannot assign string constant to *name*
  - must copy character by character

`char * name;`

- allocates space for pointer to characters
- can assign string constant to *name*
  - changes pointer
- otherwise, must allocate space for characters...

Example: string length

```
#include <stdio.h>

int slength(char s[])
{   int i;
    i =0;
    while(s[i]!='\0')
     i = i+1;
    return i;
}
```

Example: string length

```
#main(int argc,char ** argv)
{   char * q;
    q = "how long is this string?";
    printf("%s: %d
   characters\n",q,slength(q));
}

$ slength
how long is this string?: 24 characters
```

Example: string comparison

$s_0 == s_1$

- same length: n
- $0 <= i <= n-1$: $s_0[i] == s_1[i]$

- e.g. “banana” == “banana”

$s_0 < s_1$

- $0 <= i < j$: $s_1[i] == s_2[i]$
- $s_0[j] < s_1[j]$

- e.g. “banana” < “banish”
- e.g. “ban” < “band”

Example: string comparison

$s_0 > s_1$

- $0 <= i < j$: $s_1[i] == s_2[i]$
- $s_0[j] > s_1[j]$

- e.g. "banquet" > "banana"
- e.g. "bank" > "ban"
- `int scomp(char s0[],char s1[])`
  - `s0 == s1` ➔ `0`
  - `s0 <= s1` ➔ `-1`
  - `s0 >= s1` ➔ `1`

Example: string comparison

```
#include <stdio.h>

int scomp(char s0[],char s1[])
{   int i;
    i = 0;
    while(s0[i]==s1[i])
    {   if(s0[i]=='\0') return 0;
        i = i+1;
    }
    if(s0[i]=='\0' || (s0[i]<s1[i])) return -1;
    return 1;
}
```

Example: string comparison

```
main(int argc,char ** argv)
{  printf("banana banana %d\n",scomp("banana","banana"));
   printf("banana banish %d\n",scomp("banana","banish"));
   printf("ban band %d\n",scomp("ban","band"));
   printf("banquet banana %d\n",scomp("banquet","banana"));
   printf("bank ban %d\n",scomp("bank","ban"));
}
```

```
$ scomp
banana banana 0
banana banish -1
ban band -1
banquet banana 1
bank ban 1
```

Structures

- finite sequence of elements of potentially different types
- each element identified by a field name
- like a Java object with no methods

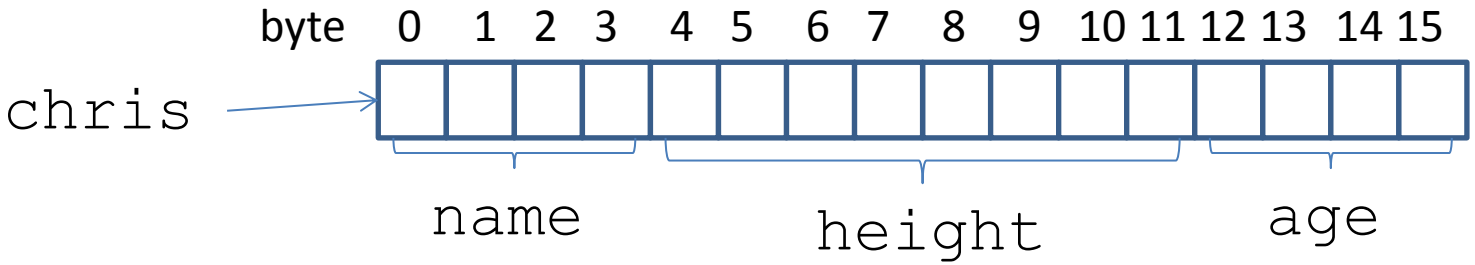
Structures

`struct {`*type*$_1$ *name*$_1$`; ... `*type*$_N$ *name*$_N$`; }` *name*`;`

- *name*$_i$ == field
- allocate: size of *type*$_1$ + ... + size for *type*$_N$ on stack
- fields held left to right
- NB *name* != &*name*
  - &*name* is address of 1st byte in sequence
  - *name* is value of byte sequence , depending on type context

Structure declaration: example

```
struct {char * name, float height,int age;}
  chris;
```



Structure access

- in expression

$exp.name_i$ ➔

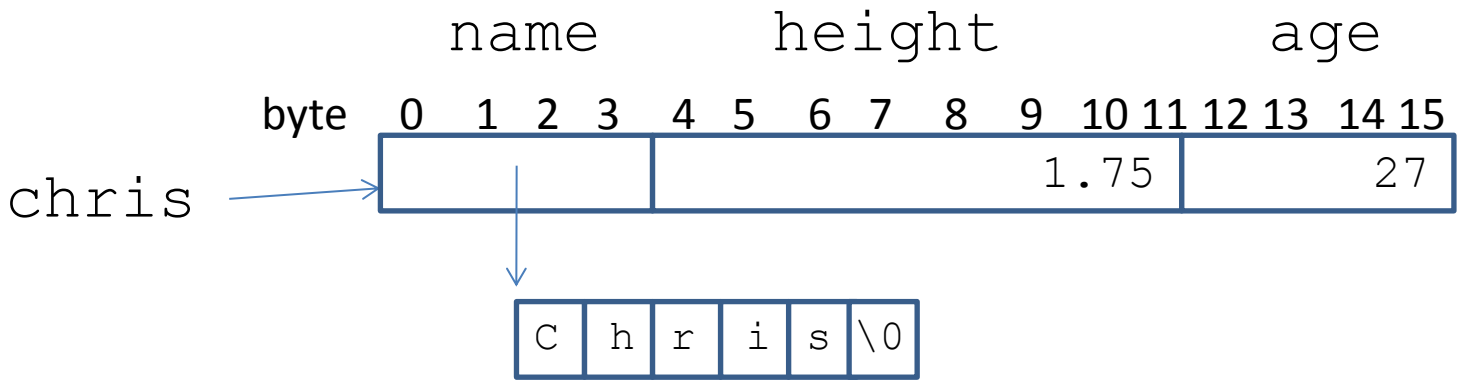
* (& exp + size for $type_1$ ...+ size for $type_{i-1}$)

- i.e. contents of offset of preceding fields from start of structure

Structure access: example

```
struct {char * name,int age,float height;}
  chris;
chris.name = "Chris";
chris.height = 1.75;
chris.age = 27;
```

- byte 0
- byte 0+4 == 4
- byte 0+4+8 == 12



Structure type definition

`struct` *name* `{`*type*$_1$ *name*$_1$`;` `...` *type*$_N$ *name*$_N$`;}`;

- `struct` *name* is type of structure
- defines type
- does not allocate space

`struct` *name*$_1$ *name*$_2$;

- allocates stack space
- associate *name*$_2$ with 1st byte of new sequence of type `struct` *name*$_1$

Example: character count

- count how often each distinct character appears in a file
- array of struct to hold character and count
- for each character in file
  - if character already has struct in array then increment count
  - if unknown character then add to array with count 1

Example: character count

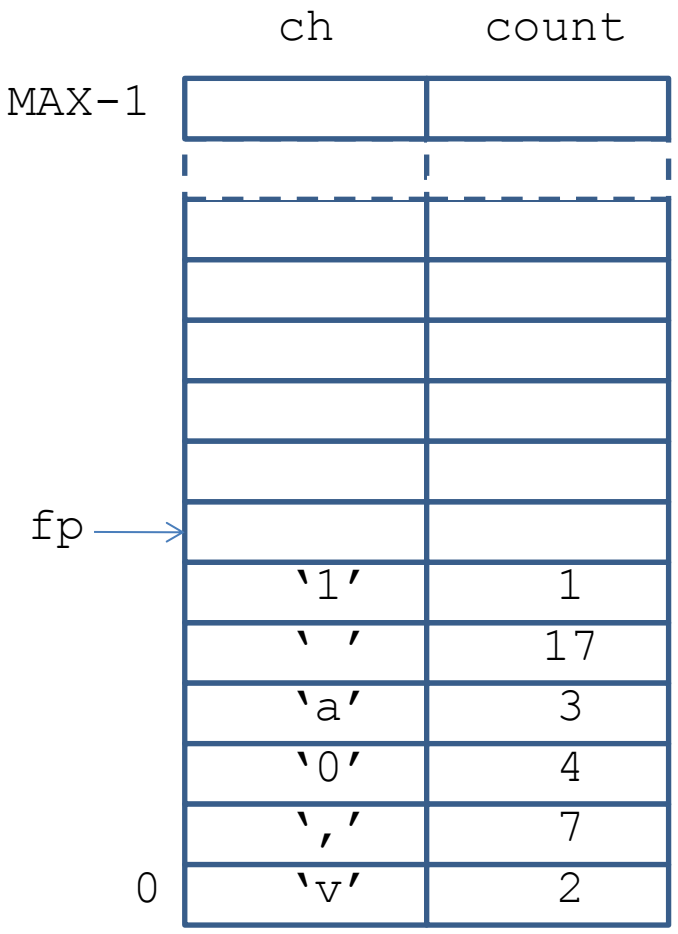
```
#include <stdio.h>
#include <stdlib.h>

#define MAX 255

struct freq {int ch;int count;};

struct freq f[MAX];
int fp;
```

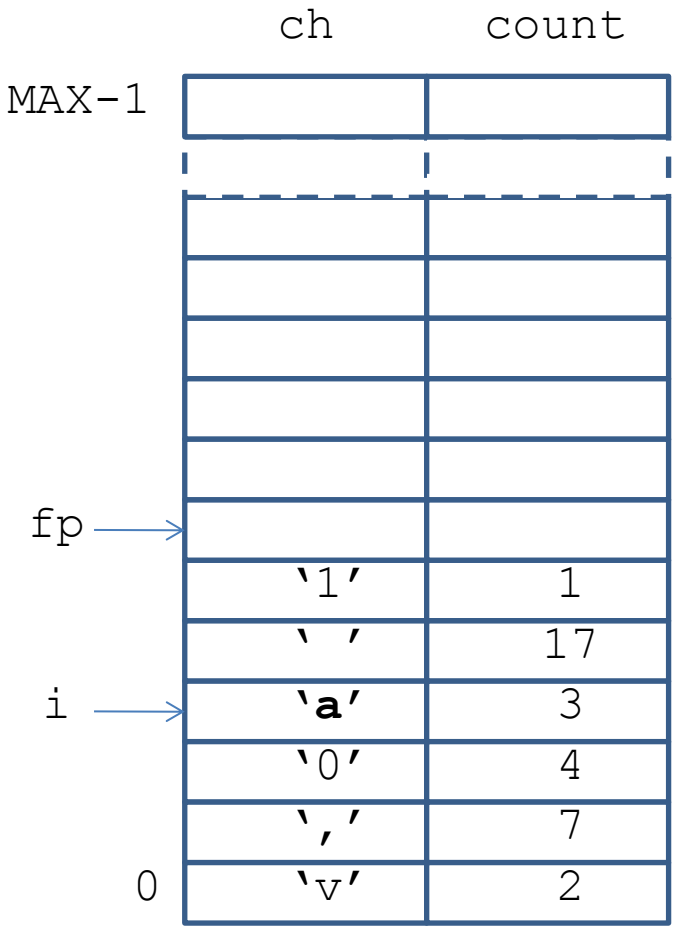
- `fp` is index for next free entry in `f`



Example: character count

```
incFreq(int ch)
{   int i;
    i=0;
    while(i<fp)
     if(f[i].ch==ch)
     {   f[i].count = f[i].count+1;
         return;
     }
     else
      i = i+1;
```

- search `f` for entry for `ch`
- if found, increment `count` and return

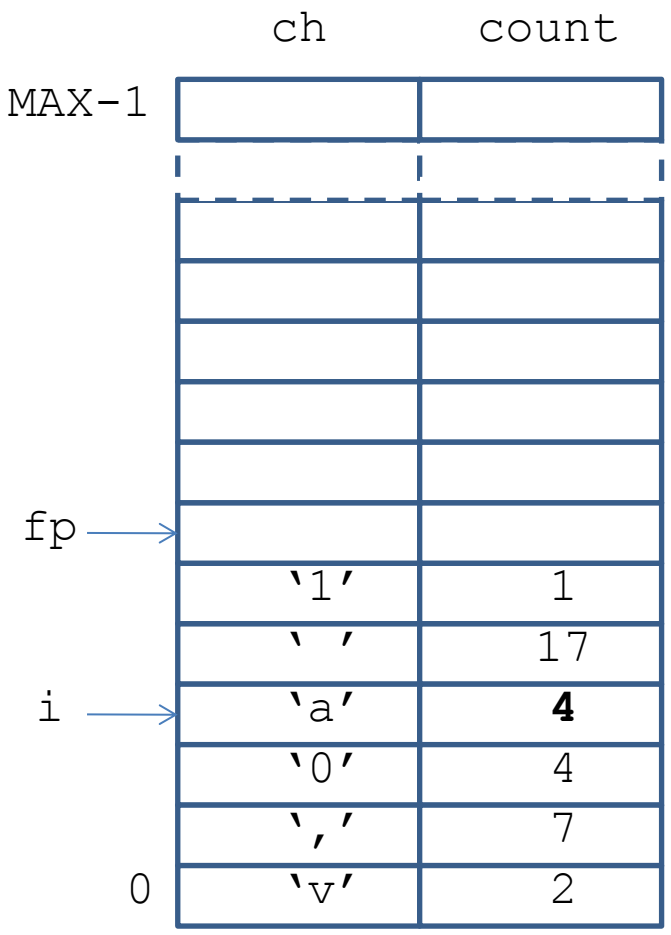


e.g. `ch ==‘a’`

Example: character count

```
incFreq(int ch)
{   int i;
    i=0;
    while(i<fp)
     if(f[i].ch==ch)
     {   f[i].count = f[i].count+1;
         return;
     }
     else
      i = i+1;
```

- search `f` for entry for `ch`
- if found, increment `count` and return

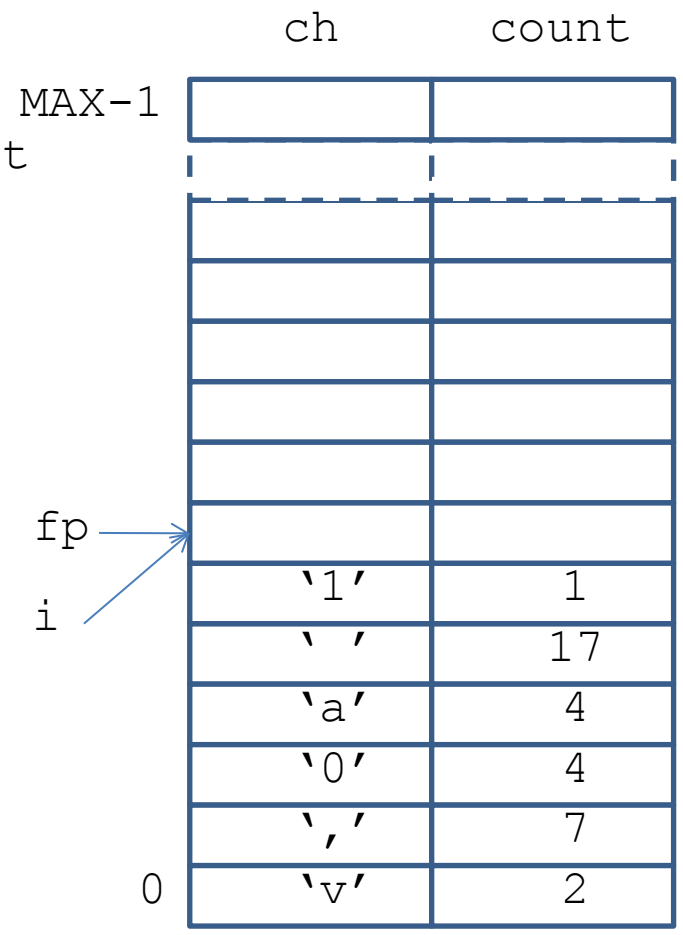


e.g. `ch ==‘a’`

Example: character count

```
  if(fp==MAX)
    {  printf("more than %d different
                    characters\n",MAX);
       exit(0);
    }
    f[fp].ch = ch;
    f[fp].count = 1;
    fp = fp+1;
}
```

- if `ch` not found
  - check for free entry in `f`
  - add `ch` /1 to next free entry
  - increment `fp`

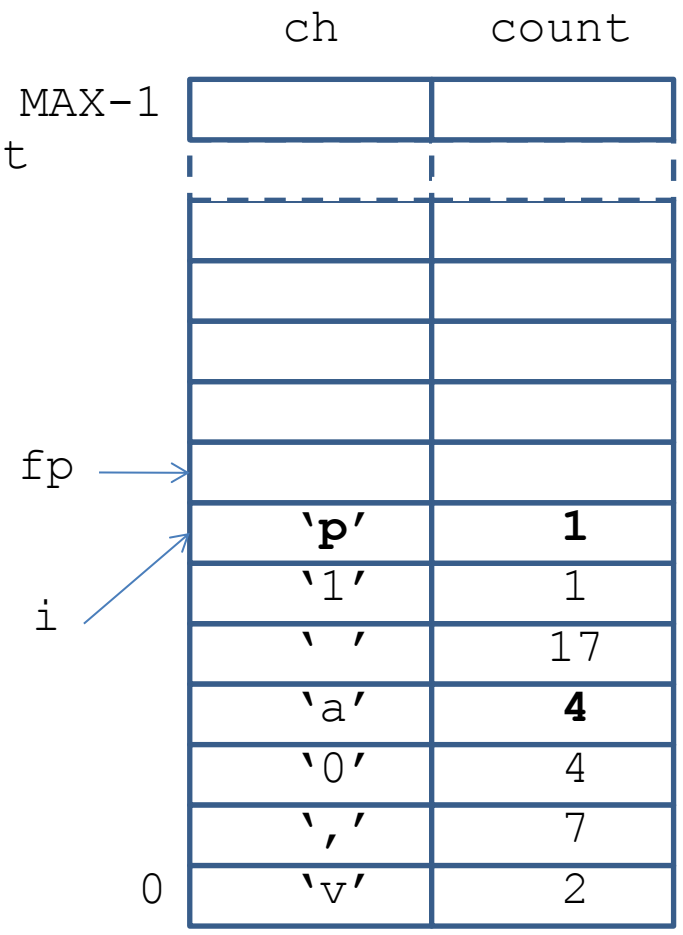


e.g. `ch == 'p'`

Example: character count

```
  if(fp==MAX)
    {  printf("more than %d different
                    characters\n",MAX);
       exit(0);
    }
    f[fp].ch = ch;
    f[fp].count = 1;
    fp = fp+1;
}
```

- if `ch` not found
  - check for free entry in `f`
  - add `ch` /1 to next free entry
  - increment `fp`



e.g. `ch == 'p'`

Example: character count

```
showFreq()
{   int i;
    i = 0;
    while(i<fp)
    {   printf("%c : %d\n",f[i].ch,f[i].count);
        i = i+1;
    }
}
```

Example: character count

```
main(int argc,char ** argv)
{  int ch;
   FILE * fin;
   if((fin=fopen(argv[1],"r"))==NULL)
   {  printf("can't open %s\n",argv[1]); exit(0); }
   fp = 0;
   ch = getc(fin);
   while(ch!=EOF)
   {  incFreq(ch);
      ch = getc(fin);
   }
   fclose(fin);
   showFreq();
}
```

Example: character count

```
$ freq freq.c
# : 3
i : 48
n : 36
c : 32
l : 8
u : 10
d : 8
e : 27
  : 185
< : 4
s : 10
t : 31
o : 12
. : 9
h : 22
> : 2
  : 54
b : 1
f : 35
M : 4
A : 4
X : 4
2 : 1
5 : 2
r : 23
q : 6
{ : 9
; : 29
} : 9
[ : 10
] : 10
p : 14
F : 6
( : 21
) : 21
= : 19
0 : 5
w : 5
+ : 4
1 : 7
" : 8
m : 2
a : 10
% : 4
\ : 3
, : 6
x : 2
: : 1
g : 6
* : 3
v : 3
I : 1
L : 3
E : 2
N : 1
U : 1
' : 1
! : 1
O : 1
```

Arrays and typed pointers

type name `[int];`

- allocates space for `int` elements of *type*

type `*` *name;*

- allocates space for pointer to *type*
- does not allocate space for elements

Array assignment

- cannot assign:
  - array to array
  - type pointer to array
- must copy element by element
- cannot assign array to type pointer
- must:
  - allocate space to type pointer
  - copy element by element
- can assign type pointer to type pointer

Dynamic Space Allocation

`char * malloc(`*`int`*`)`

- allocates *`int`* bytes on heap
- returns address of 1st byt
- like `new` in Java
- in `stdlib`

Coercions

(*type*) *expression*

1. evaluate *expression* to value
2. now treat value as if *type*

- *does not physically transform expression*
- as if overlaid template for *type* on value
- also called *cast*

Dynamic Space Allocation

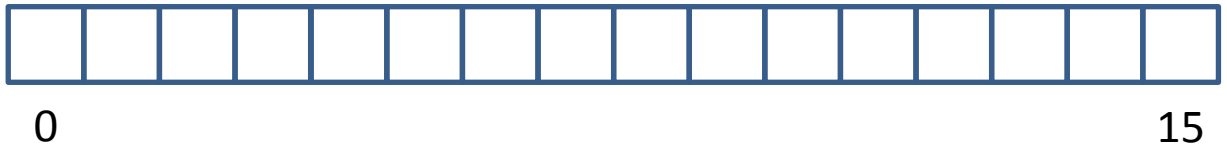
- use cast to coerce pointer to sequence of bytes to pointer to required type
- typical `malloc` use:

  (*type* `*)malloc(`*number*`*sizeof(`*type*`))`
- allocate space for *number* of size for *type*
  - but returns pointer to byte
    - so now treat address as if pointer to *type*

Dynamic Space Allocation

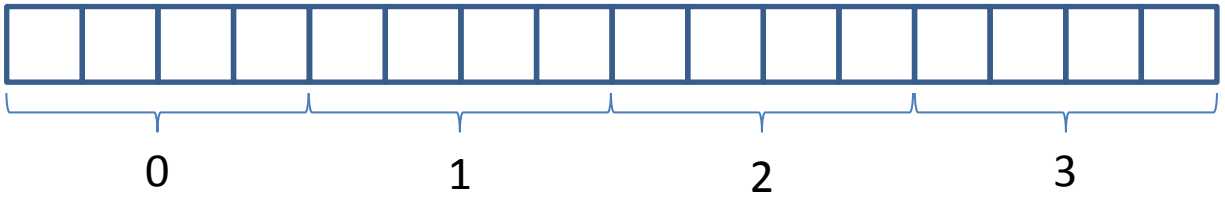
- e.g.

  `malloc(16)` ➔ pointer to 16 bytes



`(int *)malloc(4*sizeof(int))` ➔

`(int *)malloc(16)` ➔ pointer to 4 `int`s



Dynamic Space Allocation

`free(void *)`

- returns space to heap
- space must have been allocated by malloc

- NB does not recursively return space
  - must climb linked data structure freeing space

String Dynamic Space Allocation

`char * name;`

- allocates space for pointer to characters
- can assign string constant to *name*
  - changes pointer
- must allocate space for characters

  name `= malloc(`*size*`);`

Example: string copy

`char * scopy(char s[])`

- make a new copy of string `s`

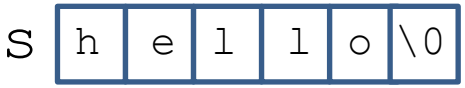
1. find length of `s`
2. allocate new byte sequence of this length
3. copy `s` element by element to new byte sequence
4. return address of new byte sequence

Example: string copy

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

char * scopy(char s[])
{   char * c; int l,i;
    l = slength(s);
    c = (char *)malloc(l);
    i = 0;
    while(i<l)
    {   c[i] = s[i];
        i = i+1;
    }
    return c;
}
```

e.g. `scopy("hello");`

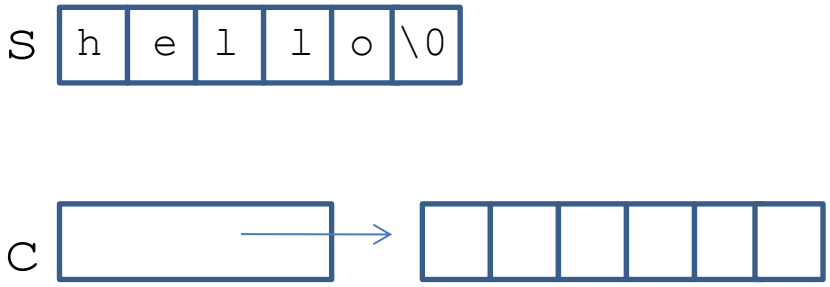


Example: string copy

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

char * scopy(char s[])
{   char * c; int l,i;
    l = slength(s);
    c = (char *)malloc(l);
    i = 0;
    while(i<l)
    {   c[i] = s[i];
        i = i+1;
    }
    return c;
}
```

e.g. `scopy("hello");`

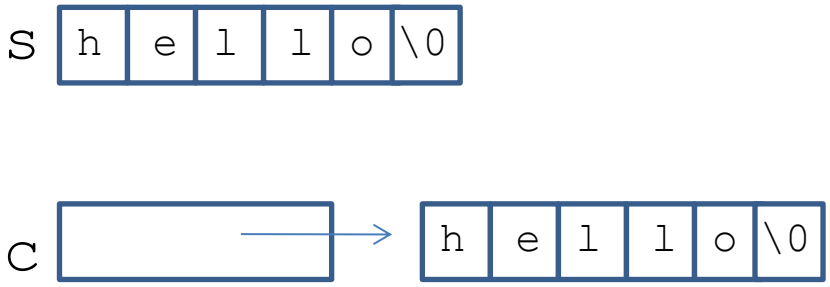


Example: string copy

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

char * scopy(char s[])
{   char * c; int l,i;
    l = slength(s);
    c = (char *)malloc(l);
    i = 0;
    while(i<l)
    {   c[i] = s[i];
        i = i+1;
    }
    return c;
}
```

e.g. `scopy("hello");`



Example: string copy

```
main(int argc,char ** argv)
{   char * s;
    s = scopy(“Able was I ere I saw Elba");
    printf("%s\n",s);
}

$ scopy
Able was I ere I saw Elba!
```

Structure assignment

- can assign one structure to another
  - fields copied automatically
  - two identical versions in different memory sequences
  - *copying*
- can assign pointer to structure to pointer to structure
  - both pointers now refer to same memory sequence
    - aliases
  - *sharing*

Example: sort character counts

```
quicksort(struct freq a[],int l,int r)
{   int p;
    if(l<r)
    {   p = partition(a,l,r);
        quicksort(a,l,p-1);
        quicksort(a,p,r);
    }
}
```

Example: sort character counts

```
int partition(struct freq a[],int l,int r)
{  int i,j,p;
   i = l; j = r;
   p = a[(i+j)/2].count;
   while(i<=j)
   {  while(a[i].count<p) i = i+1;
      while(a[j].count>p) j = j-1;
      if(i<=j)
      {  swap(a,i,j);
         i = i+1;j = j-1; }
   }
   return i;
}
```

Example: sort character counts

```
swap(struct freq a[],int i,int j)
{   struct freq t;
    t = a[i];
    a[i] = a[j];
    a[j] = t;
}
```

- copy whole `struct` from
  - `a[i]` to `t`
  - `a[j]` to `a[i]`
  - `t` to `a[j]`

Stack of `struct` v stack of pointers

- stack of entries
  - allocate more `struct` space than actually use
  - copy whole `struct`s in `swap`
- instead:
  - allocate stack space for pointers to `struct`
  - copy pointers in `swap`
- must:
  - `malloc` space for `struct`s
  - access fields from pointer

Structure space allocation

- for `struct` *name:*

`(struct` *name* `*)`

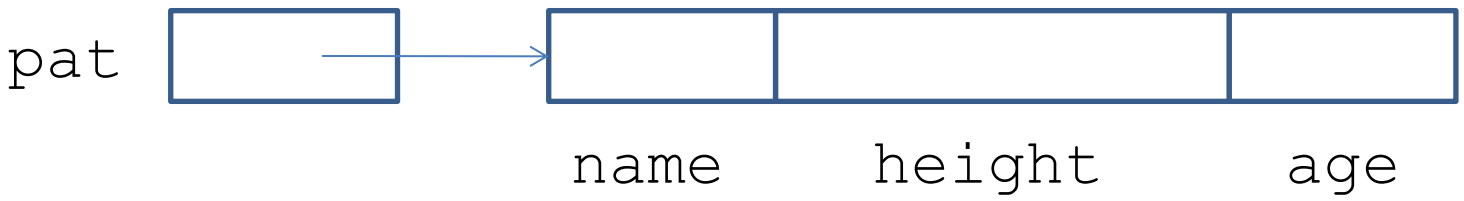
`malloc(sizeof(struct` *name*`)`

- allocate for *size of* type
- cast to *pointer to* type

Structure space allocation

- e.g.

```
struct person
 {char * name;double height;int age;};
struct person * pat;
pat = (struct person *)
          malloc(sizeof(struct person));
```



Structure pointer field access

$exp->name_i$ ➔

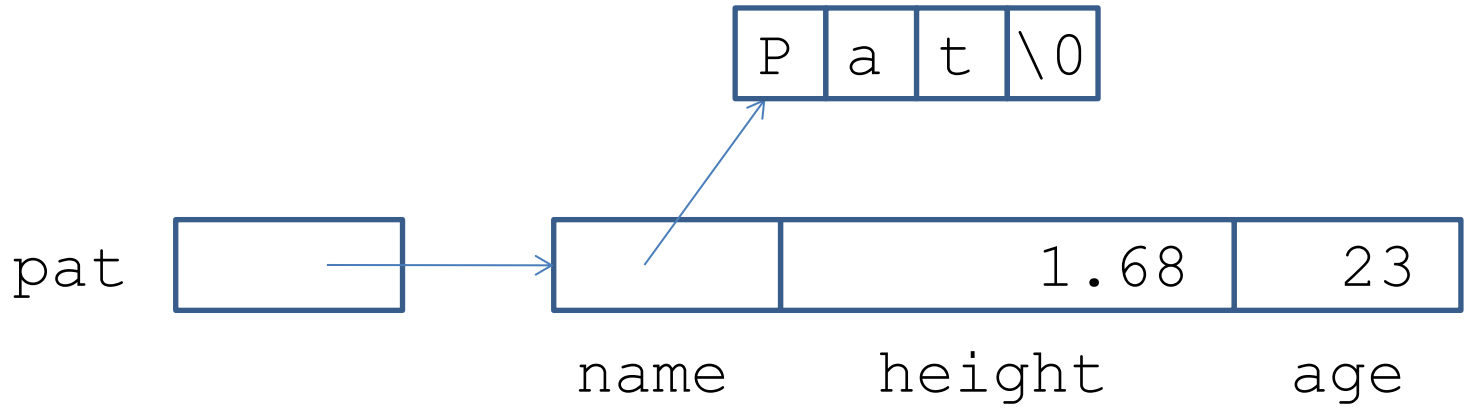
$(*exp).name_i$

- i.e. contents of offset of preceding elements from byte sequence that identifier points at

Structure pointer field access

- e.g.

```
pat->name = “Pat”;
pat->height = 1.68;
pat->age = 23;
```



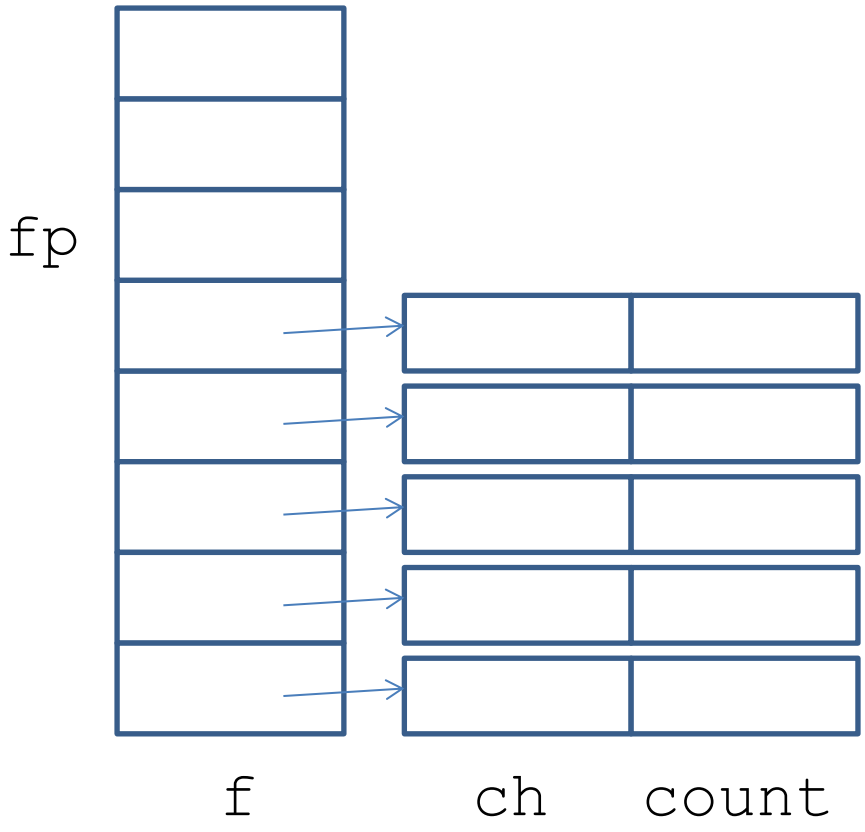
Example: character frequency 2

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

#define MAX 255

struct freq
 {int ch;int count;};

struct freq * f[MAX];
int fp;
```



Example: character frequency 2

```
incFreq(int ch)
{   int i;
    i=0;
    while(i<fp)
     if(f[i]->ch==ch)
     {   f[i]->count = f[i]->count+1;
         return;
     }
     else
      i = i+1;
```

Example: character frequency 2

```
    if(fp==MAX)
    {  printf("more than %d different
                    characters\n",MAX);
       exit(0);
    }
    f[fp] = (struct freq *)
                malloc(sizeof(struct freq));
    f[fp]->ch = ch;
    f[fp]->count = 1;
    fp = fp+1;
}
```

Example: character frequency 2

```
showFreq()
{   int i;
    i = 0;
    while(i<fp)
    {   printf("%c : %d\n, ",
                    f[i]->ch,f[i]->count);
        i = i+1;
    }
    printf("\n");
}
```

Example: character frequency 2

```
swap(struct freq * a[],int i,int j)
{   struct freq *t;
    t = a[i];
    a[i] = a[j];
    a[j] = t;
}
```

Example: character frequency 2

```
int partition(struct freq * a[],int l,int r)
{  int i,j,p;
   i = l; j = r;
   p = a[(i+j)/2]->count;
   while(i<=j)
   {  while(a[i]->count<p)
       i = i+1;
      while(a[j]->count>p)
       j = j-1;
```

Example: character frequency 2

```
        if(i<=j)
        {   swap(a,i,j);
            i = i+1;
            j = j-1;
        }
    }
    return i;
}
```

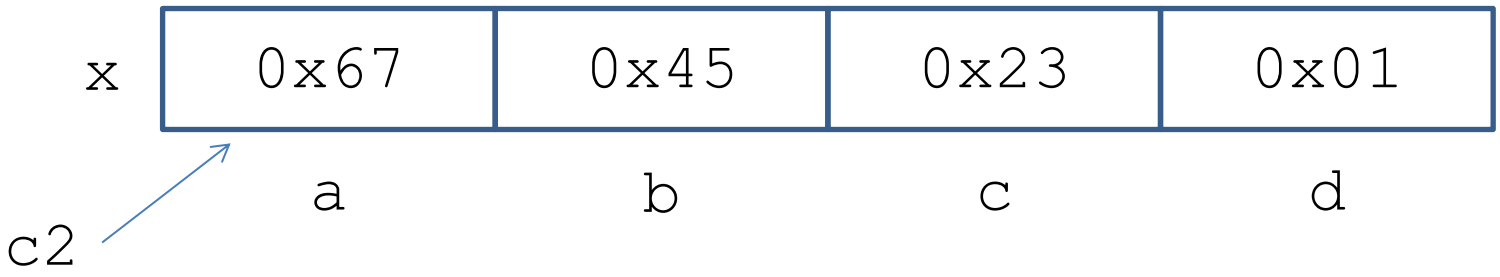
Casting int to struct

```
struct chars {char a;char b;char c;char d;};
int x;
x = 0x01234567;
c2 = (struct char *) &x;
printf("%x %x %x %x\n",
        c2->a,c2->b,c2->c,c2->d);
```

➔ 67 45 23 1

Casting int to to struct

- `&x` returns address of 1st byte of `int`
- `(struct chars *)` coerces address of 1st byte of `int` to address of 1st byte of struct of 4*`char`
  - `c2->a` is 1st byte of `x` == `0x67`
  - `c2->b` is 2nd byte of `x` == `0x45` etc



Pointers to functions

- functions are code sequences in memory
  - so functions have memory addresses
- function name is an alias for the address

```
#include <stdio.h>
sq(int x){  return x*x; }
main()
{  printf("sq: %d, &sq: %d\n",sq,&sq); } ➔

sq: 134513572, &sq: 134513572
```

Pointers to functions

type (* *name*) ()

- *name* is a pointer to function that returns a *type*

```
main()
{   int (*f)();
    f = sq;
    printf("%d\n",f(3));
} ➔

9
```

Higher Order Functions

- function that takes another function as argument
- e.g. map: apply arbitrary function to all elements of array

```
map(int (*f)(),int * a,int n)
{   int i;
    for(i=0;i<n;i++)
     a[i] = f(a[i]);
}
```

Higher Order Function

```
#include <stdio.h>

#define MAX 5

inc(int x){  return x+1;  }

twice(int x){  return 2*x;  }

show(int * a,int n)
{  int i;
   for(i=0;i<n;i++)
    printf("%d ",a[i]);
   printf("\n");
}
```

```
main()
{  int a[MAX];
   int i;
   for(i=0;i<MAX;i++)a[i] = i;
   show(a,5);
   map(inc,a,5);
   show(a,5);
   map(twice,a,5);
   show(a,5)
} ➔
```

```
0 1 2 3 4
1 2 3 4 5
2 4 6 8 10
```

Higher Order Function

- HOFs basis of algorithmic skeletons for parallelism
- e.g. for `map`, can run each `f` on a separate processor on one array element at same time
- e.g. Google map-reduce for filtering www searches
  - on farms of hundreds of thousands of core
  - map to search lots of different subsets of www
  - reduce to find common/most frequent hits