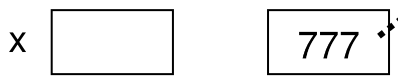
Lecture – Pointers, arrays, and files

Pointer types

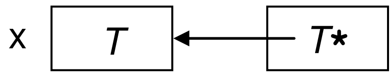
- Pointers are addresses of memory cells.

Memory address 777

address of variable x

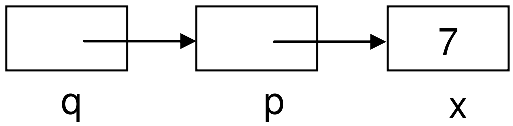


Abstract diagram



- Example

```
int main(void)
{
   int x=7;
   int* p=&x;
   int** q=&p;
   printf("%d%d%d",x,*p,**q);      // 777
   printf("%p",p);                 // 006AFDF4, say
}                 // Pointers are displayed in hexadecimal.
```



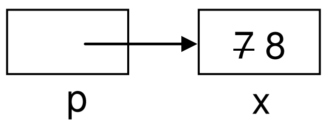
Comments

1 `int*` // pointer-type declarator
`*p` // indirection operator

2 `int*x,y;` // **`*x`** and **`y`** are of type **`int`**,
`int*x,*y;` // **`x`** and **`y`** are of type **`int*`**

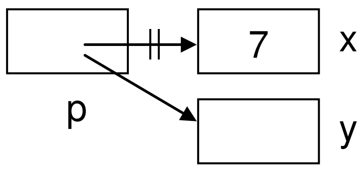
3 With pointers, there are two objects to manipulate:

the pointed-to object, e.g. **`*p=8`**



and

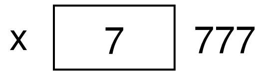
the point itself, e.g. **`p=&y`**



- Pointers as parameters

On parameter passing methods

Consider the call **`p(x)`**, should **`x`**'s value or address be passed?

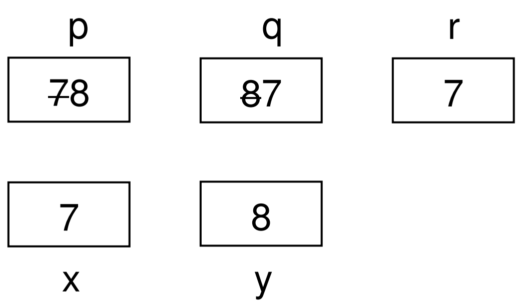


1 call by value – the value of actual parameter is passed
2 call by reference – the address of actual parameter is passed

C adopts call by value.
C++ adopts both call by value and call by reference.

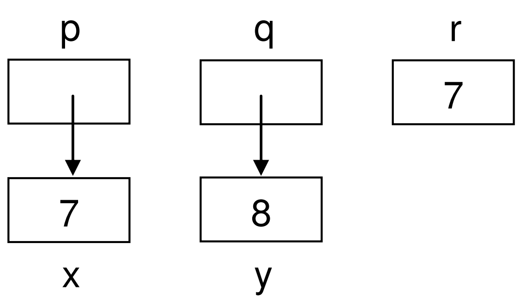
Characteristic of call by value
Modifying formal parameters does NOT alter the values of actual parameters.

```
void swap(int p,int q)
{
   int r=p; p=q; q=r;
}
int main(void)
{
   int x=7,y=8;
   swap(x,y);
   printf("%d%d",x,y);      // 78
}
```



Characteristic of call by reference
Modifying formal parameters alters the values of actual parameters

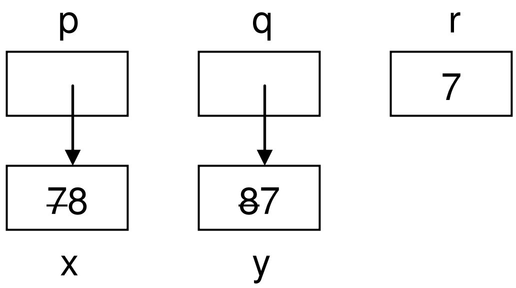
```
void swap(int& p,int& q)    // C++ only; reference type
{
   int r=p; p=q; q=r;
}
int main(void)
{
   int x=7,y=8;
   swap(x,y);
   printf("%d%d",x,y);      // 87
}
```



- Pointers as parameters (Cont'd)

In C/C++, call by reference may be simulated by call by value.

```
void swap(int* p,int* q)
{
   int r=*p; *p=*q; *q=r;
}
int main(void)
{
   int x=7,y=8;
   swap(&x,&y);
   printf("%d%d",x,y);      // 87
}
```

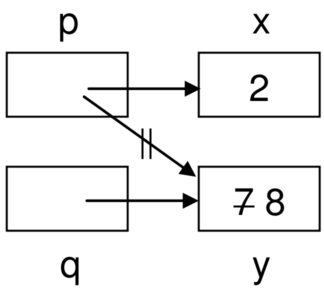


Comments

1 This is call by value, rather than call by reference. We don't modify the parameters **p** and **q** themselves. What we modify are the pointed-to objects ***p** and ***q**.

Put another way, if the formal parameter is modified, the actual parameter won't change:

```
void foo(int* p)
{
   int x=2;
   (*p)++;
   p=&x;
}
int main(void)
{
   int y=7,*q=&y;
   foo(q);
}
```

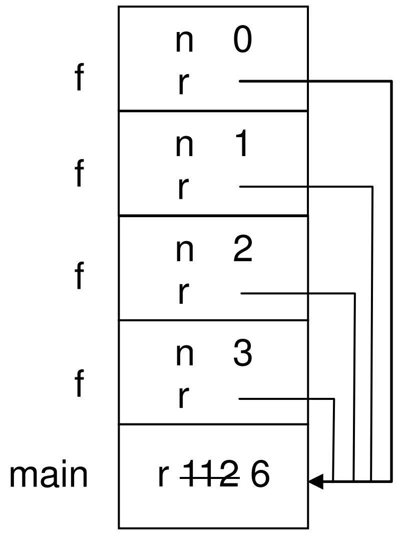


2 For call by value, the actual parameters needn't be variables, i.e. they needn't have addresses, e.g. **&x**,**&y**

- Pointers as parameters (Cont'd)

Another example – returning function values by parameters

```
void f(int n,int* r)
{
   if (n==0) *r=1;
   else {
      f(n-1,r); *r*=n;
   }
}
int main(void)
{
   int r;
   f(3,&r);
   printf("%d",r);
}
```



On parameter functionalities and parameter passing methods

There are three modes of functionalities:

1	in	caller → callee	e.g. f(in n,out r)
2	out	caller ← callee	
3	inout	caller ↔ callee	e.g. swap(inout p,inout q)

Parameter functionalities are high-level concepts.
Parameter passing methods are low-level mechanisms.

Functionality	Parameter passing method
in	call by value (for small object)
	call by const reference (for large object)
out	call by reference
inout	call by reference

For example, given

```
struct foo { double x[1000],y[1000]; } bar;
```

that contains the *x*- and *y*-coordinates of 1000 points, we want to compute the maximum distance from the origin.

- Pointers as parameters (Cont'd)

```
struct foo { double x[1000],y[1000]; } bar;
inline double sq(double x) { return x*x; }
double distance(const foo* p)   // in mode
{
   double d=0.0;
   for (int i=0;i<1000;i++) {
      double e=sqrt(sq(p->x[i])+sq(p->y[i]));
      if (d<e) d=e;
   }
   return d;        // p->x[i] ≡ (*p).x[i]
}
int main(void) { printf("%f",distance(&bar)); }
```

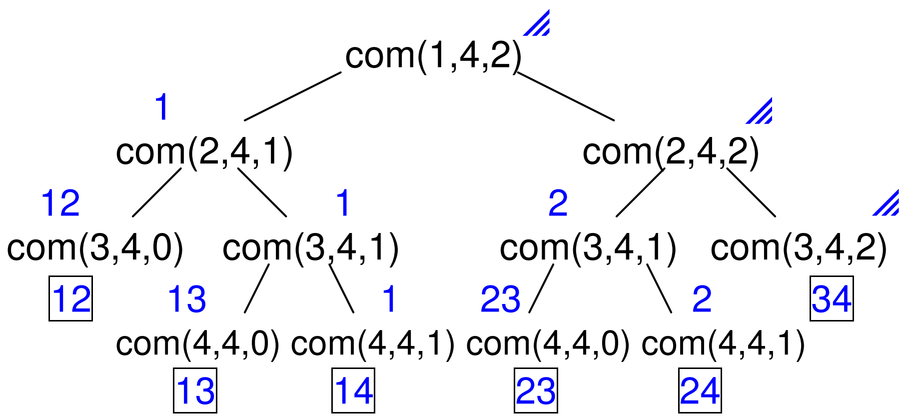
Occasionally, with guaranteed safety, we may simply use call-by-reference instead of call-by-const-reference for large in-mode objects.

Example – Combination generation (Revisited)

Instead of using a global stack or a local static stack, we may use a local auto stack.

Version C1 – call-by-value

Recall the search tree depicted in previous lecture



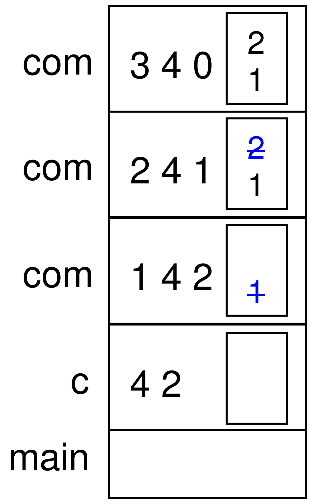
// generate elements in stack s + any k-permutation of $\{m, \ldots, n\}$

```
int com(int m,int n,int k,stack s);
```

in mode, call by value

- Pointers as parameters (Cont'd)

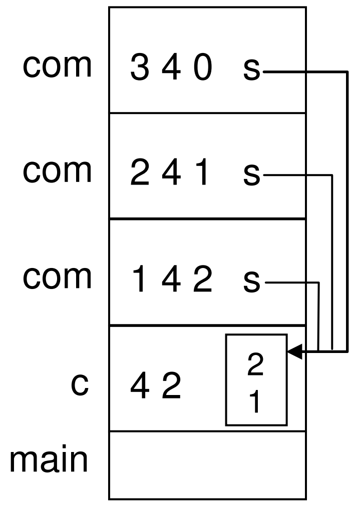
```
int com(int m,int n,int k,stack s)
{
   if (k==0||n-m+1==k) {
      // same as version B
   } else {
      s.stk[++s.top]=m;
      unsigned r=com(m+1,n,k-1,s);
      s.top--;
      return r+com(m+1,n,k,s);
   }
}
int c(int n,int k)
{
   stack s={-1};        // local auto stack
   return com(1,n,k,s);
}
```



Clearly, call-by-value is expensive. So, let's use call-by-reference.

Version C2 – call-by-reference

```
int com(int m,int n,int k,stack* s)
{
   if (k==0||n-m+1==k) {
      for (int i=0;i<=s->top;i++)
         printf("%d",s->stk[i]);
      if (k!=0)
         for (int i=m;i<=n;i++)
            printf("%d",i);
      printf("\n");
      return 1;
   } else {
      s->stk[++s->top]=m;          // *
      int r=com(m+1,n,k-1,s);
      s->top--;                    // *
      return r+com(m+1,n,k,s);
   }
}
```



- Pointers as parameters (Cont'd)

```
int c(int n,int k)
{
   stack s={-1};
   return com(1,n,k,&s);
}
```

Q: Why don't declare

```
int com(int m,int n,int k,const stack* s);
```

A: First of all, we can. However, if so declared, the two starred lines are erroneous, and explicit casts are necessary:

```
((stack*)s)->stk[++((stack*)s)->top]=m;
((stack*)s)->top--;
```

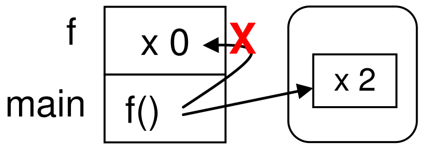
Moreover, our code is safe, since a push is followed by a pop. Finally, from another point of view, our code emphasizes that the stack is a working data structure cooperated by various invocations of the recursive function **com**.

- Pointers as function values

A function may return by value or return by reference.

Example – Simulation of return by reference in C/C++

```
int* f(void)                // int& f(void)
{
   static int x=0;   ①
   return &x;               // return x;
}
int main(void)
{
   for (int i=1;i<=10;i++)
      *f()+=i;       ②     // f()+=i;
   printf("%d",*f());       // f()
}
```



① Q: Can this be a local auto variable?
A: No! Were it a local auto variable, the pointer returned would be a *dangling pointer* pointing to a location that isn't in use.
Lesson–Never return a pointer pointing to a local auto variable

② The value of a local static variable is modified from outside!!

- Pointers to const objects

```
int x=7;
const int y=x;          // Ok; x and y are independent

const int x=7;
int y=x;                // Ok; x and y are independent

int x=7;                // int*->const int*
const int* y=&x;        // Ok; qualification conversion
x++;                    // x has full control over its nonconstness
printf("%d",*y);        // *y was modified silently.

const int x=7;          // const int*->int*
int* y=&x;              // Warning in C; Error in C++
(*y)++;                 // Were it allowed, x would lose control
                        // of its constness.

const int x=7;
int* y=(int*)&x;        // Ok; but be sure that it is really wanted
(*y)++;
printf("%d%d",x,*y);    // 88 in C; 78 in C++ (usually)
```

Why? It is because that x isn't a *constant expression* in C, but is a *constant expression* in C++. Thus, most C++ compilers replace x by 7.

- Const pointers

```
int x=7,y=8;
```

	`p=&y;`	`(*p)++;`
`int* p=&x;`	✓	✓
`int const* p=&x;`	✓	×
`int*const p=&x;`	×	✓
`int const*const p=&x;`	×	×

Example – Better simulations of by-reference by by-value

```
void swap(int*const p,int*const q);
double distance(const foo*const p);
```

- Cast between pointer types

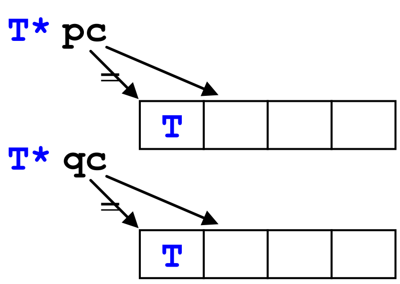
```
int x=7;
double y=x;          // Ok; x and y are independent

int x=7;
double* y=&x;        // Warning in C; Error in C++
printf("%f",*y);     // Were it allowed, the bit pattern of x would
                     // be reinterpreted.

double* y=(double*)&x;
                     // Ok; but be sure that it is really wanted
```

- Generic pointers

```
int x=7;
void* y=&x;          // Ok, pointer conversion (no reinterpretation
                     // takes place here)
int* c=y;            // Ok in C; Error in C++
                     // Were it allowed, the bit pattern of *y
                     // would be (re)interpreted (C++'s view).
*y                   // Error! Illegal indirection
```

Example

```
void swap(void* p,void* q,size_t sz)
{
   char* pc=(char*)p;
   char* qc=(char*)q;
   for (int i=0;i<sz;i++) {
      char c=*pc; *pc=*qc; *qc=c;
      pc++; qc++;
   }
}
int main(void)
{
   int a=2,b=3;
   swap(&a,&b,sizeof(int));
   double c=2.3,d=4.5;
   swap(&c,&d,sizeof(double));
}
```



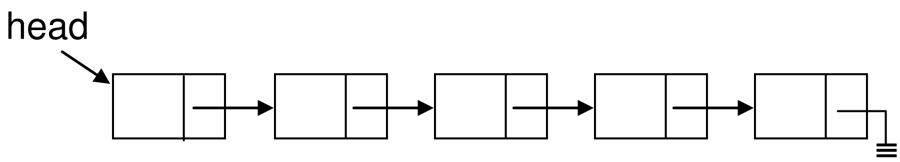
- Generic pointers (Cont'd)

Cf. One function for each type

```
void swap(int* p,int* q)
{
   int t=*p; *p=*q; *q=t;
}
void swap(double* p,double* q)
{
   double t=*p; *p=*q; *q=t;
}
```

- Null pointers

A null pointer is used to mark the end of a dynamic data structure, e.g. linked-list




```
int* p=NULL;
while (p!=NULL) …
*p                              // Error! Access violation
```

The macro **NULL** is defined in **<stdio.h>**, **<stddef.h>**, etc., as

```
#define NULL 0                  // C++
#define NULL ((void*)0)         // C
```

Comment

The internal representation of a null pointer depends on the compiler. It may be a zero address, a nonsexist address, etc.

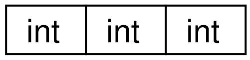
- Nonzero integers and pointers

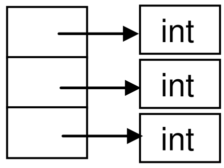
```
int* p=777;                     // Warning in C; Error in C++
int x=p;                        // Warning in C; Error in C++
```

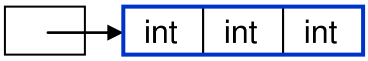
Array types

- Arrays, pointers to arrays, and arrays of pointers

`int[3]`	Arrays of 3 ints
`int*[3]`	Arrays of 3 pointers to ints
`int(*)[3]`	Pointers to arrays of 3 ints



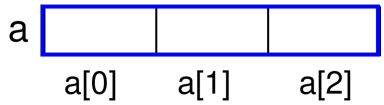




- Types of arrays

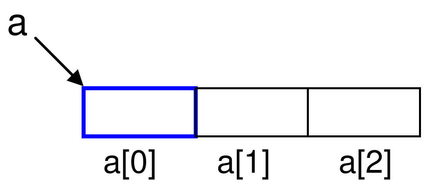
Given an array declared by

```
int a[3];
```



The array **`a`** is of the array type **`int[3]`**.

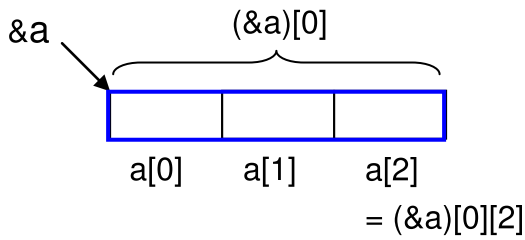
But in certain cases, it is implicitly converted to **`int*`** – the array name **`a`** is treated as a pointer pointing to the 0^{th} element of the array. This is called array-to-pointer conversion.



- Arrays as arrays

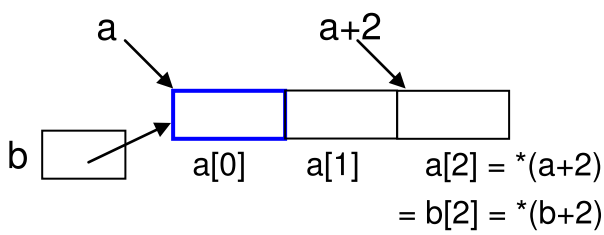
```
sizeof(a) ≡ sizeof(int[3])
&a
```

```
int(*)[3]
   &a
 int[3]
```



- Arrays as pointers

```
a[i] ≡ *(a+i)
int* b=a;
```



Comment

`int[3]`	// array-type declarator
`a[i]`	// indexing operator

- Pointer arithmetic 1

  Let p be a pointer of type T***** pointing to an array element and k be an integer, then

  1 $p \pm k$ or $\pm k+p$ is also a pointer of type T*****
  2 $p \pm k$ or $\pm k+p$ points to the kth element after/before p

  In other words,

  $p \pm k$ = **(**T***)((char*)**$p \pm k$***sizeof(**T**))**

  Note that if $p \pm k$ points outside the bounds of the array, except to the element past the high end of the array, the result is undefined.

- Array indexing

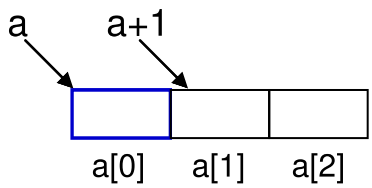
  $e1$**[**$e2$**]** $\equiv$ ***(**$e1$**+** $e2$**)**

  where one of the two expressions must be a pointer, and the other an integer.

  Observe that **&** $e1$**[**$e2$**]** $\equiv$ **&*(** $e1$**+**$e2$**)** $\equiv$ $e1$**+**$e2$

  Example

  a[2] **(a+1)[-1]**
  2[a] **(a+1)[1]** **1[&a[1]]**



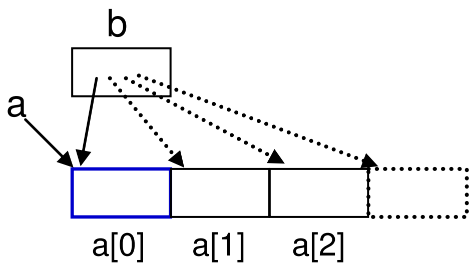
  Comment: Arrays may have negative indexes.

- Comments on arrays as pointers

  An array name viewed as a pointer does not occupy storage. In other words, the pointer is a constant.

  Example

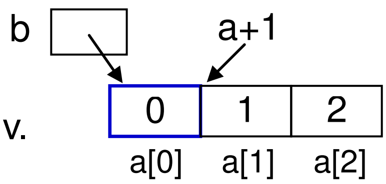
```
int a[3]={0,1,2};
int *b=a;
for (int i=0;i<3;i++)
    printf("%d",*b++);  ①
```



  ① or **b[i]**, or **a[i]**, but not ***a++**

- Comments on arrays as pointers (Cont'd)

```
int a[3]={0,1,2};
int *b=a;
```

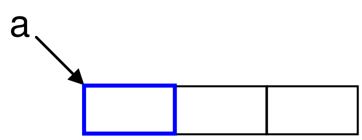


	value	side effect	equiv.
`*b++`	0	`b == a+1`	
`(*b)++`	0	`a[0]==1`	`(*a)++`
`*++b`	1	`b == a+1`	
`++*b`	1	`a[0]==1`	`++*a`

- Arrays as parameters

Version A – Arrays as pointers

```
int find(int* a,int sz,int key)     ①
{
   int i;
   for (i=0;i<sz;i++)
      if (a[i]==key) break;
   return i;
}
int main(void)
{
   int a[3]={0,1,2},key=5;
   if (find(a,3,key)!=3); else;
}
```

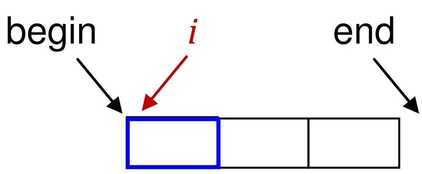


① Note that the array size isn't part of the parameter type **`int*`**. Alternatively, this function may be declared as

```
int find(int a[],int sz,int key)
int find(int a[777],int sz,int key)  // 777 ignored
```

Version B – Arrays as pointers as iterators

```
int* find(int* begin,int* end,int key)  // [begin,end)
{
   int* i;
   for (i=begin;i!=end;i++)
      if (*i==key) break;
   return i;
}
```

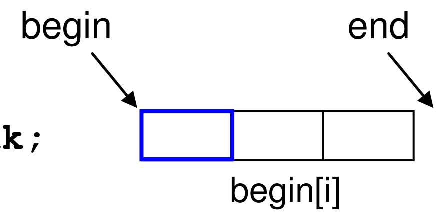


- Arrays as parameters (Continued)

```
int main(void)
{
   int a[3]={0,1,2},key=5;
   if (find(a,a+3,key)!=a+3); else;
}
```

Alternatively, we may write

```
int* find(int* begin,int* end,int key)
{
   int i;
   for (i=0;i<end-begin;i++)
      if (begin[i]==key) break;
   return begin+i;
}
```



Pointer arithmetic 2

Let p and q be two pointers of type T***** pointing to elements of the same array or to the element past the high end of an array, then

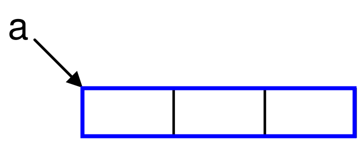
1 $|p-q|$ = the number of elements between p and q
2 $p-q$ is a signed integer of type **ptrdiff_t**, defined in **<stddef.h>**

In other words,

$p-q$ = **(ptrdiff_t)(((char*)**p**-(char*)q)/sizeof(**T**))**

Version C: Arrays as arrays

```
int find(int (*a)[3],int key)
{
   int i;
   for (i=0;i<3;i++)
      if ((*a)[i]==key) break;        // or, a[0][i]
   return i;
}
```



- Arrays as parameters (Continued)

```
int main(void)
{
   int a[3]={1,2,3},key=5;
   if (find(&a,key)!=3); else;
}
```

On by-value and by-reference array parameters

In C/C++, arrays are 2nd class objects.

They can't be passed by value.
However, the effect of passing an array by value may be obtained by passing a structure that contains an array by value.

For example,

```
struct ARRAY { int x[777]; };
void p(ARRAY a)
{
   for (int i=0;i<777;i++) printf("%d",a.x[i]);
}
```

They can only be passed by (simulation of) reference.

Method A: Arrays as pointers

```
int find(int* a,int sz,int key);
int* find(int* begin,int* end,int key);
```

Pro: Both apply to arrays of type `int[`*k*`]`, for any $k \geq 1$.

Con: The array size has to be passed.

Method B: Arrays as arrays (similar to simulations of other types)

```
int find(int (*a)[3],int key)
```

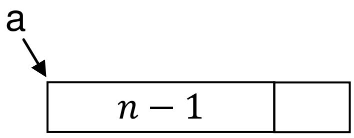
Pro: The array size needn't be passed.
Con: It applies only to arrays of type `int[3]`.

- Arrays as parameters (Continued)

Method A is particularly suitable for recursion.

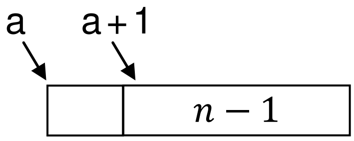
For example, to recursively sum up the elements of an array

Version A



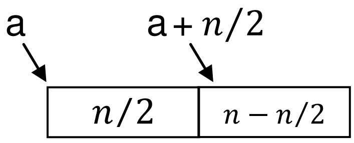

```
int sum(int* a,int n)
{
   return n==1? a[0]: a[n-1]+sum(a,n-1);
}
```

or




```
int sum(int* a,int n)
{
   return n==1? a[0]: a[0]+sum(a+1,n-1);
}
```

or




```
int sum(int* a,int n)
{
   return n==1? a[0]: sum(a,n/2)+sum(a+n/2,n-n/2);
}
int main(void)
{
   int a[7]={1,2,3,4,5,6,7};
   printf("%d\n",sum(a,7));
}
```

Version B

```
int sum(int* begin,int* end)
{
   ptrdiff_t n=end-begin;
   return n==1? *begin: sum(begin,begin+n/2)+
                                  sum(begin+n/2,end);
}
int main(void)
{
   int a[7]={1,2,3,4,5,6,7};
   printf("%d\n",sum(a,a+7));
}
```

Multidimensional arrays

- K-dimensional arrays

  T **a**[n_1][n_2]...[n_k];

  1 As an array T [n_1][n_2]...[n_k]
  2 As a pointer T (*)[n_2]...[n_k]

  The following discussions concentrate on $k = 2$.

- Array indexing

  a[i][j] = i[a][j] = j[a[i]] = j[i[a]]

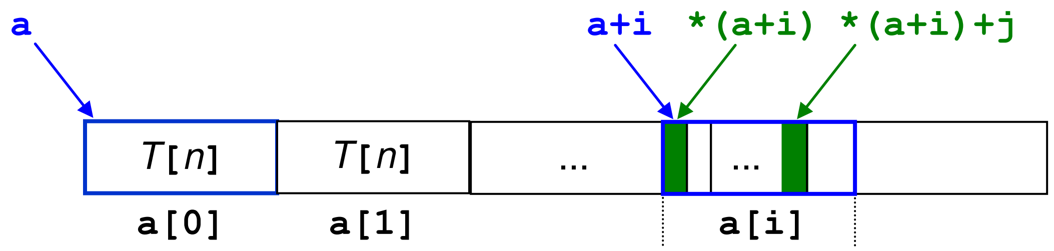
  a[i][j] = (*(a+i))[j] = *(a[i]+j) = *(*(a+i)+j)

  Note that ***(a+i)[j] = *a[i+j] = a[i+j][0]**

  Let **a** be an array of type T[m][n]. We have

  ***(*(a+i)+j)**

  = *(T***)((char*)a+i*sizeof(**T[n]**)+j*sizeof(**T**))**



Given

int a[2][3];

the types of subexpressions in ***(*(a+i)+j)** are shown below:

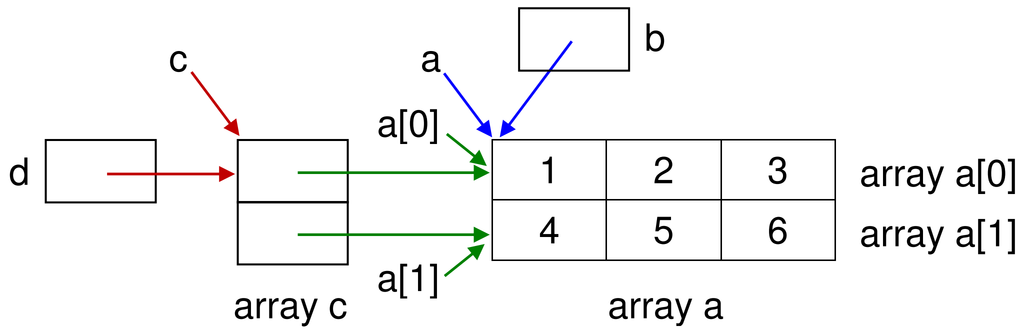
expression	type	array-to-point conversion
a	**int[2][3]**	
a+i	**int(*)[3]**	**a: int[2][3]→int(*)[3]**
***(a+i)**	**int[3]**	
***(a+i)+j**	**int***	***(a+i): int[3]→int***
***(*(a+i)+j)**	**int**	

- Arrays/pointers of/to arrays/pointers

`int[2][3]`	Arrays of arrays
`int*[2]`	Arrays of pointers
`int(*)[3]`	Pointers to arrays
`int**`	Pointers to pointers

Variable names appear aside (l.h.s of `[]` and r.h.s of `*`) the type declarator of the highest precedence.

```
int a[2][3]={{1,2,3},{4,5,6}};
int (*b)[3]=a;
int* c[2]={a[0],a[1]};
int** d=c;
```



With this diagram, the two-dimensional array may be accessed by any of the four names

Why? All can be though in terms of parameter-passing.

1 Passing **a** to **b** makes `a[i][j]==b[i][j]`
2 Passing `a[i]` to `c[i]` makes `a[i][j]==c[i][j]`
3 Passing **c** to **d** makes `c[i][j]==d[i][j]`

Another view of the relationships among the four names

a retains the sizes of both dimensions
b discards the size of the 1st dimension
c discards the size of the 2nd dimension
d discard the sizes of both dimensions

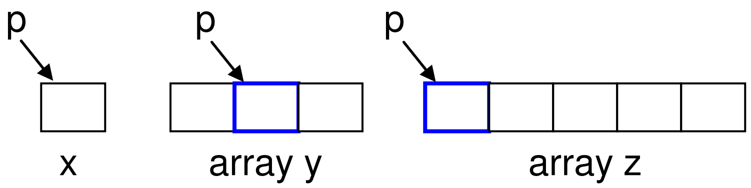
- Arrays/pointers of/to arrays/pointers (Cont'd)

  Remark – Pointers are more general than arrays.

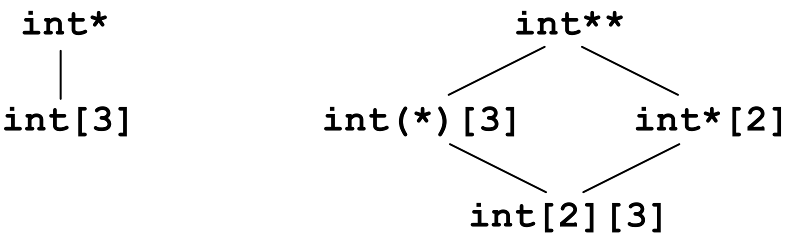
  For example,

  `int[3]` a fixed-size array
  `int*` the pointed-to **`int`** object may be a stand-alone **`int`** object or an **`int`** object of an array of any size.

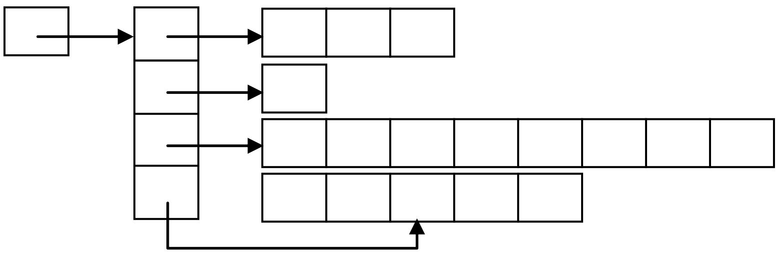
```
int x,y[3],z[5];
int* p;
p=&x;
p=y+1;
p=z;
```



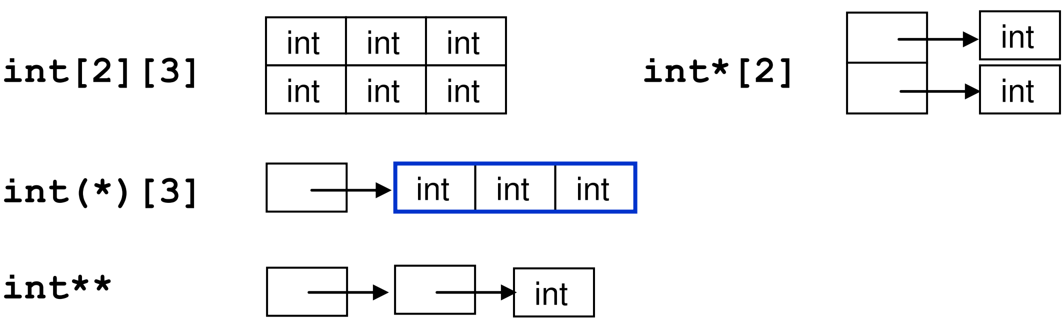
"More-general-than" relationship diagram



Here is a possible structure pointed to by a pointer of type **`int**`**



Simplest diagram for each type



- 2-dimensional arrays as parameters

Version A – Array of arrays

```
// Both dimensions fixed; Con – inflexible in both dimensions
void print(int (*a)[2][3])
{
   for (int i=0;i<2;i++) {
      for (int j=0;j<3;j++) printf("%d ",(*a)[i][j]);
      printf("\n");
   }
}
int main(void)
{
   int a[2][3]={1,2,3,4,5,6},b[1][3]={1,2,3};
   int c[2][2]={1,2,3,4};
   print(&a);          // ok
   print(&b);          // no
   print(&c);          // no
}
```

Version B – Pointer to arrays

```
// 1st dimension unfixed, 2nd dimension fixed
// Con – inflexible in the 2nd dimension
void print(int (*a)[3],int sz1)
{
   for (int i=0;i<sz1;i++) {
      for (int j=0;j<3;j++) printf("%d ",a[i][j]);
      printf("\n");
   }
}
int main(void)
{
   int a[2][3]={1,2,3,4,5,6},b[1][3]={1,2,3};
   int c[2][2]={1,2,3,4};
   print(a,2);         // ok
   print(b,1);         // ok
   print(c,2);         // no
}
```

- 2-dimensional arrays as parameters (Cont'd)

  Version C – Array of pointers

  // 1st dimension fixed, 2nd dimension unfixed // `int**[2]`
  // Con – inflexible in the 1st dimension // `int(**)[2]`

```
void print(int*(*a)[2],int sz2)    // int*(*)[2]
{
   for (int i=0;i<2;i++) {
      for (int j=0;j<sz2;j++)
         printf("%d ",(*a)[i][j]);
      printf("\n");
   }
}
int main(void)
{
   int a[2][3]={1,2,3,4,5,6},b[1][3]={1,2,3};
   int c[2][2]={1,2,3,4};
   int *aa[2]={a[0],a[1]},*bb[1]={b[0]};
   int *cc[2]={c[0],c[1]};
   print(&aa,3);     // ok
   print(&bb,3);     // no
   print(&cc,2);     // ok
}
```

  Version D – Pointer to pointers

  // Both dimensions unfixed; Con – Need extra space

```
void print(int** a,int sz1,int sz2)
{
   for (int i=0;i<sz1;i++) {
      for (int j=0;j<sz2;j++)
         printf("%d ",a[i][j]);
      printf("\n");
   }
}
```

- 2-dimensional arrays as parameters (Cont'd)

Version D (Cont'd)

```
int main(void)
{
   int a[2][3]={1,2,3,4,5,6},b[1][3]={1,2,3};
   int c[2][2]={1,2,3,4};
   int *aa[2]={a[0],a[1]},*bb[1]={b[0]};
   int *cc[2]={c[0],c[1]};
   print(aa,2,3);        // ok
   print(bb,1,3);        // ok
   print(cc,2,2);        // ok
}
```

Version E – Generic pointer

// Both dimensions unfixed; Con – low level code

```
void print(void* a,int sz1,int sz2)
{
   for (int i=0;i<sz1;i++) {
      for (int j=0;j<sz2;j++) {
         int* aij=(int*)((char*)a+i*sz2*sizeof(int)
                                  +j*sizeof(int));
      // int* aij=(int*)a+i*sz2+j;
         printf("%d ",*aij);
      }
      printf("\n");
   }
}

int main(void)
{
   int a[2][3]={1,2,3,4,5,6},b[1][3]={1,2,3};
   int c[2][2]={1,2,3,4};
   print(a,2,3);        // ok; or, print(&a,2,3);
   print(b,1,3);        // ok
   print(c,2,2);        // ok
}
```

- 2-dimensional arrays as parameters (Cont'd)

On complex types involving pointers and arrays

1. `[]`'s appear on the right and are left-associative
2. `*`'s appear on the left and are right-associative

```
int**[2][3]
```

In this case, there are $\frac{4!}{2!2!} = 6$ possible types, because `[]`'s must be left-associative and `*`'s must be right-associative.

```
int*[2][3]
```

In this case, there are $\frac{3!}{2!} = 3$ possible types, namely,

```
    3  1   2           2  1    3          1    2    3
int*[2][3]        int(*[2])[3]       int(*)[2][3]
```

N.B.
`int*[2]([3])` is illegal – it has to be written as `int*[3][2]`

On square matrices as parameters

Version A

```
void print(int (*a)[3][3]);
```

This guarantees that the argument is a 3×3 square matrix, e.g.

```
int a[2][3];
print(&a);                    // no
```

Version B, C, D, E

```
void print(int(*a)[3],int sz1);
void print(int*(*a)[2],int sz2)
void print(int** a,int sz1,int sz2);
void print(void* a,int sz1,int sz2);
```

These have no guarantee that the arguments will be 3×3 square matrices, e.g.

```
int a[2][3];
print(a,2,3);                 // ok, but square matrices expected
```

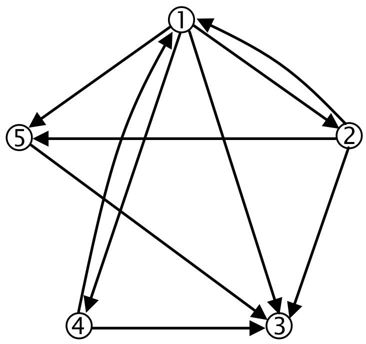
- Example – The celebrity problem

Given a be-known-to relation among persons, determine if there is a celebrity (who knows nobody, but is known to everybody).

(Whether a person knows himself is of no concern here. We shall assume that nobody knows himself.)

This problem may be restated as a graph problem:

Given a loopless digraph (directed graph) with n vertices, is there a vertex v such that in-degree(v) = $n-1$, and out-degree(v) = 0.



This digraph may be represented as an adjacency matrix.

	1	2	3	4	5
1	0	1	1	1	1
2	1	0	1	0	1
3	0	0	0	0	0
4	1	0	1	0	0
5	0	0	1	0	0

With this representation, the problem becomes:

Given an $n \times n$ square matrix, is there a row k of zeros such that the corresponding column k contains only ones (except for the crossroad of row k and column k)?

- Example (Cont'd)

Algorithm A – $O(n^2)$ in the worst case

```
const int n=5;
void celebrity(int (*a)[n][n])      // C++ only
{
   int i;
   for (i=0;i<n;i++) {
      int j;
      for (j=0;j<n;j++) if ((*a)[i][j]!=0) break;
      if (j==n) break;
   }
   if (i==n) { printf("No celebrity"); return; }
   int j;
   for (j=0;j<n;j++)
      if (i!=j&&(*a)[j][i]!=1) break;
   printf(j!=n?"No celebrity":"Celebrity %d",i+1);
}
int main(void)                 // extra argument has no harm
{
   int a[n][n]= {0,1,1,1,1,
                 1,0,1,0,1,
                 0,0,0,0,0,
                 1,0,1,0,0,
                 0,0,1,0,0};
   celebrity(&a);
}
```

Algorithm B – $O(n)$, prune and search

```
void celebrity(int (*a)[n][n])
{
   int c=0;
   for (int i=1;i<n;i++) if ((*a)[c][i]==1) c=i;
   int i;
   for (i=0;i<n;i++)
      if ((*a)[c][i]!=0||c!=i&&(*a)[i][c]!=1)
         break;
   printf(i!=n?"No celebrity":"Celebrity %d",c+1);
}
```

C-style strings

- A C-style string is an array of characters ending with a '\0', i.e. null character.
- Example

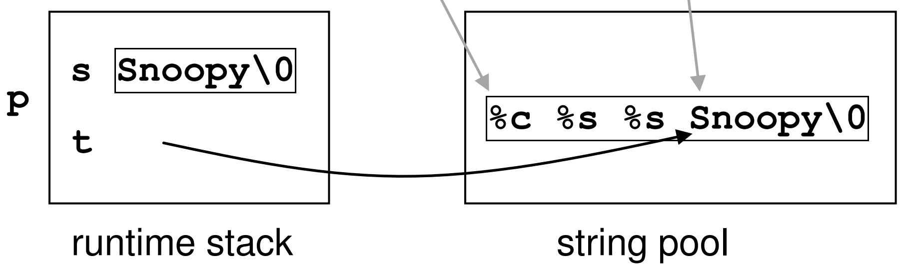
```
char r[7]="Snoopy";
char s[7]={'S','n','o','o','p','y','\0'};
char t[8]={'S','n','o','o','p','y','\0','!'};
char u[6]={'S','n','o','o','p','y'};
printf("%s",r);        // Snoopy
printf("%s",s);        // Snoopy
printf("%s",t);        // Snoopy
printf("%s",u);        // Snoopy… garbage until a '\0' is reached
```

- Example

```
void p(void)
{
   char s[]="Snoopy";
   char* t="Snoopy";
   t[0]='X';            // undefined; usually error
   printf("%c %s %s Snoopy","Snoopy"[0],s,t);
}
```

Comments

1 String literals that aren't bound to character arrays are usually collected together in a string pool, which is usually protected (at runtime) and shared.



Were `t[0]='X';` allowed, the output would be
`X Snoopy Xnoopy Xnoopy`

- Example (Cont'd)

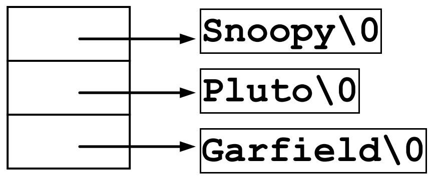
  2 `char* t="Snoopy";`
  This is OK in C, but deprecated in C++, due to the 2nd conversion
  `const char[7]` → `const char*` → `char*`
  Remedy
  `const char* t="Snoopy";`

- Example

```
char msg[][9]={"Snoopy","Pluto","Garfield"};
char* msg[]={"Snoopy","Pluto","Garfield"};
```

msg `Snoopy\0\0\0Pluto\0\0\0\0Garfield\0`

msg



- String output

```
char s[]="Snoopy";        // ok
char* s="Snoopy";         // ok
printf("%s\n",s) ≡ puts(s)
```

except that

```
int puts(const char*);
```

returns a nonnegative value (usually, 0) if successful, or EOF if an error occurs. (**`printf`** returns a negative value if an error occurs.)

- String input

```
char s[80];               // ok
char* s;                  // no
scanf("%s",s)             // not &s
```

1 skip the leading white spaces
2 read in the subsequent non-while-space characters and store them in **`s`**
3 stop at the 1st subsequent white space
4 append **`'\0'`** to **`s`**
5 undefined if shy of space (usually, corrupt)

- String input (Continued)

`gets(s)`

1 read all the characters in a line and store them in **`s`**, replacing **`'\n'`** by **`'\0'`**

2 undefined if shy of space (usually, corrupt)

3 **`char* gets(char* s);`**
return **`s`** or a null pointer on encountering an end-of-file

```
char s[20];
scanf("%s",s);
```

`snoopy pluto \n`	input buffer
`snoopy\0`	`s`

```
char s[20];
gets(s);
```

`snoopy pluto \n`	input buffer
`snoopy pluto \0`	`s`

Comment
`scanf` may be made safer by specifying the maximum number of characters to be stored.

```
char s[20];
scanf("%19s",s);    // store at most 19 characters
```

- Example – Some functions in **`<string.h>`**

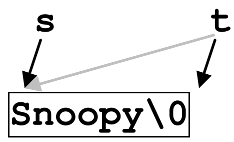
```
size_t strlen(const char* s)
{
    size_t cnt=0;    *s++!='\0'
    while (*s++) cnt++;
    return cnt;
}
```

`s` → `Snoopy\0`

or // **`strlen("Snoopy")`**

- Example (Cont'd)

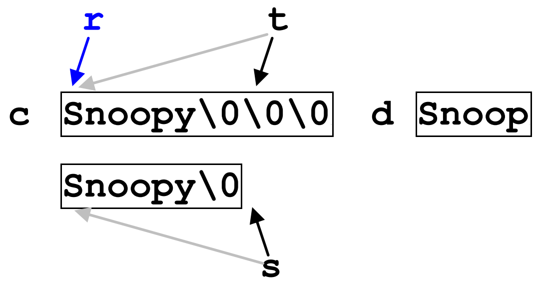
```
size_t strlen(const char* s)
{
   const char* t=s;
   while (*t++);
   return t-s-1;
}
```



// the destination **t** must have enough space
// remove the shaded code for **strcpy**

```
char* strcpy(char* t,const char* s);
char* strncpy(char* t,const char* s,size_t n)
{
   char* r=t;
   while (n>0&&(*t++=*s++)) { n--; }
   while (n>0) { *t++='\0'; n--; }
   return r;
}
```

```
char c[9],d[5];

strncpy(c,"Snoopy",9)

strncpy(d,"Snoopy",5)
```

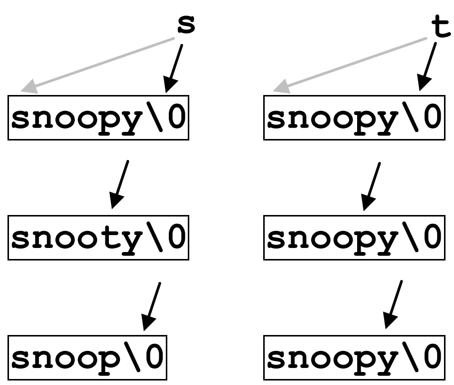


Note that **d** isn't a C-style string.

// strcmp(s,t)<0, if s<t; =0, if s=t; >0, if s>t

```
int strcmp(const char* s,const char* t)
{
   while (*s==*t&&*s) { s++; t++; }
   return *s-*t;
}
```

```
strcmp("snoopy","snoopy")

strcmp("snooty","snoopy")

strcmp("snoop","snoopy")
```



- Example – Command line arguments

// DOS / Unix echo command

```
#include <stdio.h>
int main(int argc,char *argv[])  // or, char** argv
{
   for (int i=1;i<argc;i++) {
      printf("%s ",argv[i]);      // *
   }
   printf("\n");
}
```

N.B. The starred line may be written as

```
for(int j=0;j<strlen(argv[i]);j++)
   printf("%c",argv[i][j]);
printf(" ");
```

Let **echo.exe** be the executable of this program, then

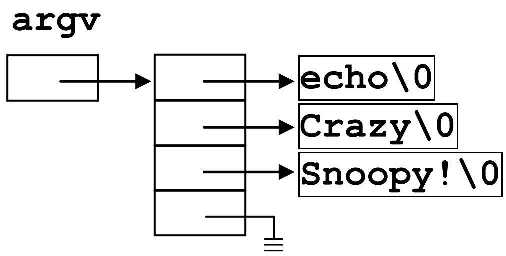
prompt> **echo Crazy Snoopy!**
Crazy Snoopy!

The command line

prompt> **echo Crazy Snoopy!**

sets **argc** to 3 and **argv** to

```
argv[0]="echo"
argv[1]="Crazy"
argv[2]="Snoopy!"
argv[3]=NULL
```



The following version also works.

```
int main(int,char *argv[])     // argc isn't used.
{
   while (*++argv)             // or, *++argv!=NULL
      printf("%s ",*argv);
   printf("\n");
}
```

- Example – Command line arguments

  // DOS copy command (Simplified version)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h> // exit, EXIT_FAILURE, EXIT_SUCCESS

inline void err(const char* msg)
{
   fprintf(stderr,msg); exit(EXIT_FAILURE);
}

int main(int argc,char *argv[])
{
   if (argc!=3) err("Illegal command.");
   if (strcmp(argv[1],argv[2])==0&&
                        strcmp(argv[1],"con")!=0)
      err("File can't be copied to itself.");
   FILE *infp,*outfp;
   if (strcmp(argv[1],"con")==0) infp=stdin;
   else {
      infp=fopen(argv[1],"rb");
      if (infp==NULL) err("File can't be opened.");
   }
   if (strcmp(argv[2],"con")==0) outfp=stdout;
   else {
      outfp=fopen(argv[2],"wb");
      if (outfp==NULL) {
         fclose(infp);
         err("File can't be opened.");
      }
   }
   int c;
   while ((c=fgetc(infp))!=EOF) fputc(c,outfp);
   fclose(infp);
   fclose(outfp);
   exit(EXIT_SUCCESS);
}
```

- Example (Cont'd)

  On **`exit`**

  `void exit(int` ***`status`*** **`);`**

  causes the program to terminate normally. The value of *status* is returned to the environment.

  `EXIT_SUCCESS`, **`EXIT_FAILURE`**

  are system-dependent integer values (usually 0 and 1).

  On **`FILE`**

  `FILE` is a structure defined in **`<stdio.h>`** that contains information to process a file.

  `FILE* fopen(const char*` ***`filename`*** **`,const char*`** ***`mode`*** **`);`**

  1. returns a file pointer if succeeded
  2. returns a null pointer if the file cannot be opened, e.g. the file doesn't exist, the user has no permission to open it, etc.
  3. mode strings
     text files: **`"r"  "w"  "a"     "r+"  "w+"  "a+"`**
     binary files: **`"rb"  "wb"  "ab"   "rb+"  "wb+"  "ab+"`**

  `int fclose(File*` ***`stream`*** **`);`**

  1. returns 0 if succeeded; or EOF, otherwise

  On text and binary files

  Text files

  1. divided into lines
     each line ends with an end-of-line marker (CRLF in Windows, LF in Unix)
     N.B. CR (Carriage Return, 13), LF (Line Feed, 10)
  2. may end with an end-of-file marker (optional Ctrl-Z in Windows Unix has no eof marker)

  Binary files

  1. no end-of-line and end-of-file markers
  2. all bytes are treated equally

- Example (Cont'd)

```
#include <stdio.h>
int main(void)
{
   FILE *outfp=fopen("X","wb");       ①
   fputc('a',outfp);
   fputc('\32',outfp);                // Ctrl-Z = 26
   fputc('b',outfp);
   fclose(outfp);
   FILE *infp=fopen("X","rb");        ②
   int c;
   while ((c=fgetc(infp))!=EOF)       // fgetc = getc
      fputc(c,stdout);                // fputc = putc
   fclose(infp);

}
```

① `"w"` and `"wb"` yield the same output file `X`. But, logically, file `X` should be declared as a binary file, since it is going to contain ctrl-Z in the middle of the file.

② In Unix, `"r"` and `"rb"` yield the same output `ab`. (Ctrl-Z has no associated printable character in Unix.)
In Windows, `"r"` yields the output `a`; whereas, `"rb"` yields the output `a→b`. (Ctrl-Z is printed as →.)

- Example (on-line judge; lexicographic order of words)

Input text file: `in.txt`
Contents

```
2                 1st line contains # of test cases
3                 # of words in the 1st test case
Yankees
    Dodgers       each line contains a word
  Angels
4                 # of words in the 2nd test case
Snoopy
Pluto
     Garfield
Mickey
```

- Example (Cont'd)

  Output text file: **out.txt**
  Required output format

Angels **Dodgers** **Yankees**	each line contains a word
*****	* terminates a test case
Garfield **Mickey** **Pluto** **Snoopy**	each line contains no leading spaces each line contains no trailing spaces
*****	last line; no more lines below

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

inline void err(const char* msg)
{
   fprintf(stderr,msg); exit(EXIT_FAILURE);
}

int main(void)
{
   FILE* infp=fopen("in.txt","r");
   if (infp==NULL) err("How can it be?");
   FILE* outfp=fopen("out.txt","w");
   if (outfp==NULL) {
      fclose(infp); err("How can it be?");
   }
   int m;
   fscanf(infp,"%d",&m);
   for (int i=1;i<=m;i++) {
      void test(FILE*,FILE*); test(infp,outfp);
      fprintf(outfp,"*\n");
   }
   fclose(infp); fclose(outfp);
}
```

- Example (Cont'd)

  // dynamically allocate an array of strings (see next lecture)
  // sort it and then deallocate it

```
void test(FILE* infp,FILE* outfp)
{
   int n;
   fscanf(infp,"%d",&n);
   char** a=(char**)calloc(n,sizeof(char*));
   for (int i=0;i<n;i++) {
      char s[80];
      fscanf(infp,"%s",s);
      a[i]=(char*)calloc(strlen(s)+1,sizeof(char));
      strcpy(a[i],s);
   }
   sort(char**,int); sort(a,n);
   for (int i=0;i<n;i++)
      fprintf(outfp,"%s\n",a[i]);
   for (int i=n-1;i>=0;i--) free(a[i]);
   free(a);
}
```

  // bubble-sort the array **a** of strings by swapping pointers

```
void sort(char** a,int n)
{
   for (int i=0;i<n-1;) {
      int k=n-1;
      for (int j=n-1;j>i;j--)
         if (strcmp(a[j-1],a[j])>0) {
            char* z=a[j]; a[j]=a[j-1]; a[j-1]=z;
            k=j;
         }
      i=k;
   }
}
```