Pointer

Pointers as Arguments to Function

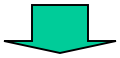
Real World Application:

Reversing A String in Place

Pointers and Multidimensional Arrays

Pointers as Arguments to Function

```
void f( int p );
main( ) {
      int x = 999;
      /* f can't change x */
      f(x); /* call by value, pass the
                  copy of x */
}
void f( int p ) {
      /* p changed but x unchanged */
      p += 2;
}
```



999	999	1001
x in main	p in f	p in f after p += 2; executes in f

```
void f( int* p);
main( ) {
      int x = 999;
      /* simulating call by reference */
      f( &x );      /* call by reference, pass
            the address of x*/
}
void f( int* p ) {
      /* p is a reference to argument x */
      *p += 2; /* changes the value of x */
}
```

999	999	1001
x in main	p in f	x in main after *p += 2; executes in f

```
/* Version 1: Call by Value */
#include <stdlib.h>
main( ) {
        int num1 = 28;
        int num2;
        int add1(  num1 );
        num2 = add1( num1 );
        return EXIT_SUCCESS;
}
int add1( int num) {
        return ++num;
}
```

after definition

main

num1 | 28

num2 |

```
/* Version 2: Passing a Pointer */
#include <stdlib.h>
int add1( int *p );
main( ) {
        int num1 = 28;
        int num2, *ptr = &num1;
        num2 = add1( ptr );
        return EXIT_SUCCESS;
}
int add1( int *p) {
        return ++*p;
}
```

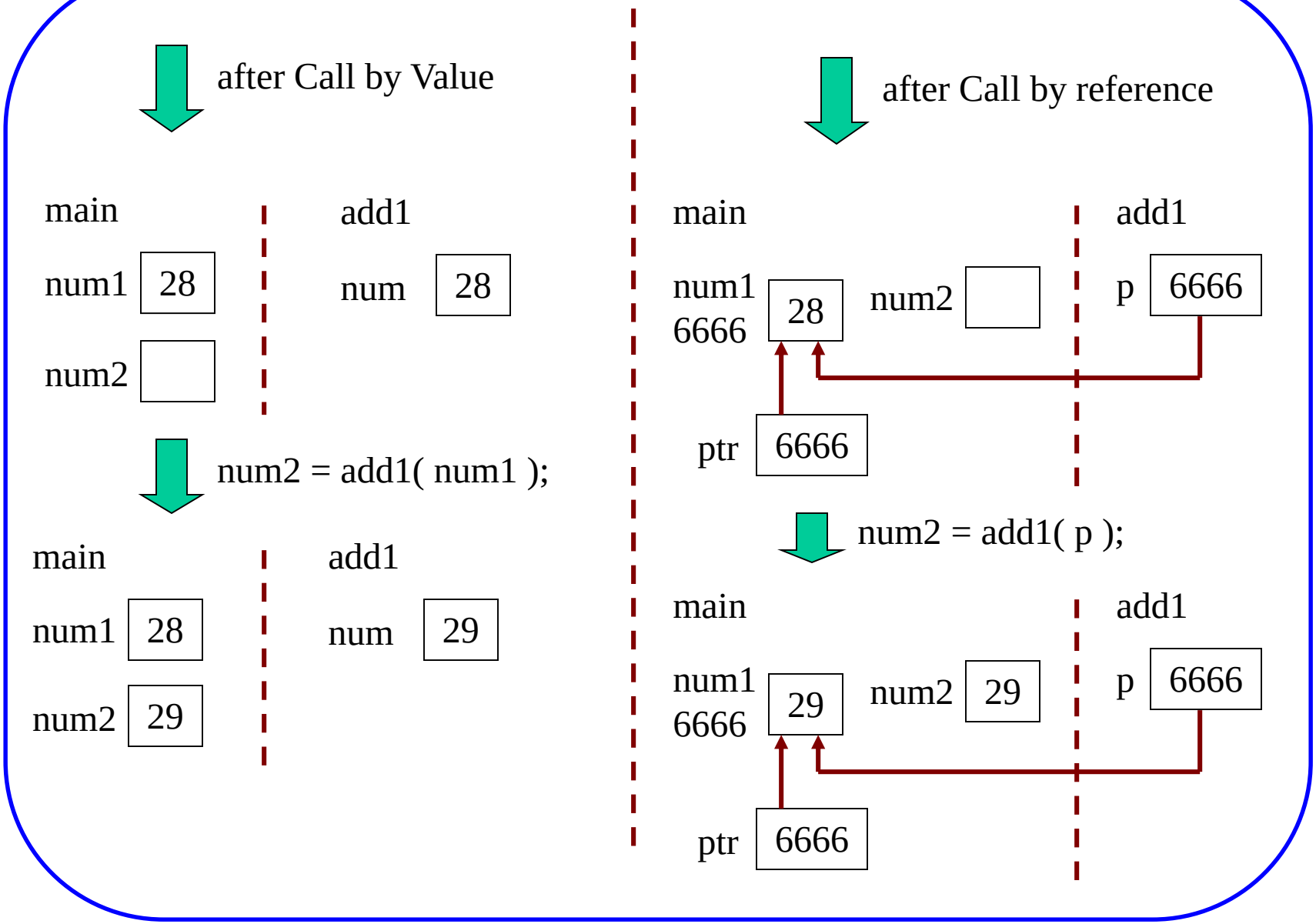
after definition

main

num1 6666 | 28

ptr | 6666

num2 |


after Call by Value
main
num1
28
num2
add1
num
28
num2 = add1(num1);
main
num1
28
num2
29
add1
num
29
after Call by reference
main
num1
6666
28
num2
add1
p
6666
ptr
6666
num2 = add1(p);
main
num1
6666
29
num2
29
add1
p
6666
ptr
6666

```
void f( int a[ ] );
main( ) {
        int s[ 10 ];
        …
        f( s );
        ...
}
```

```
void f( int* a );
main( ) {
        int s[ 10 ];
        …
        f( s );
        ...
}
```

in the body of the function, we may use either array syntax

```
a[ I ];
```

or

```
*(a + I);
```

to access a cell. Because a is a pointer variable.

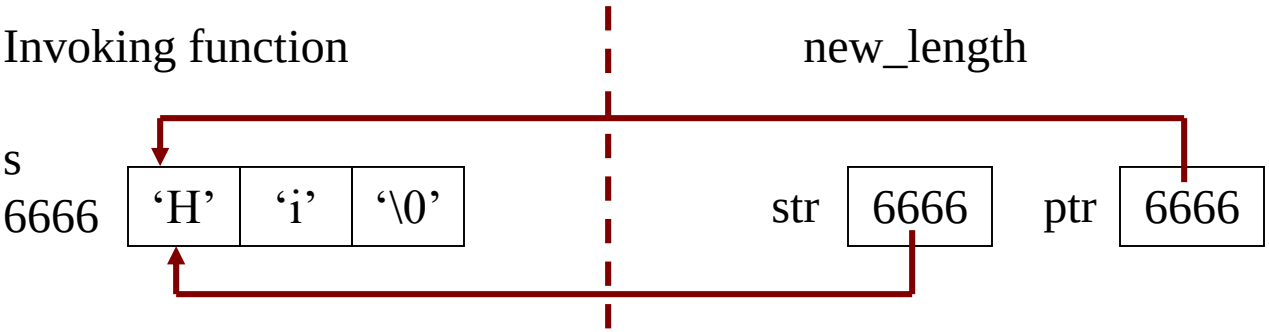
To illustrate the use of pointer syntax, we rewrite the function that computes the length of a string.

```
#include <stdio.h>
#include <stdlib.h>
int new_length( char* str ) ;
main( ) {
        char s[ ] = { 'H', 'i', '\0' };
        int slen = new_length( s );
        printf( "The length of s is %d.\n", slen );
        return EXIT_SUCCESS;
}
int new_length( char* str ) {
        char* ptr = str;
        while( *str ) /*  *str is a char; but here it is treated as ASCII code */
                ++str;
        return str - ptr;
}
```

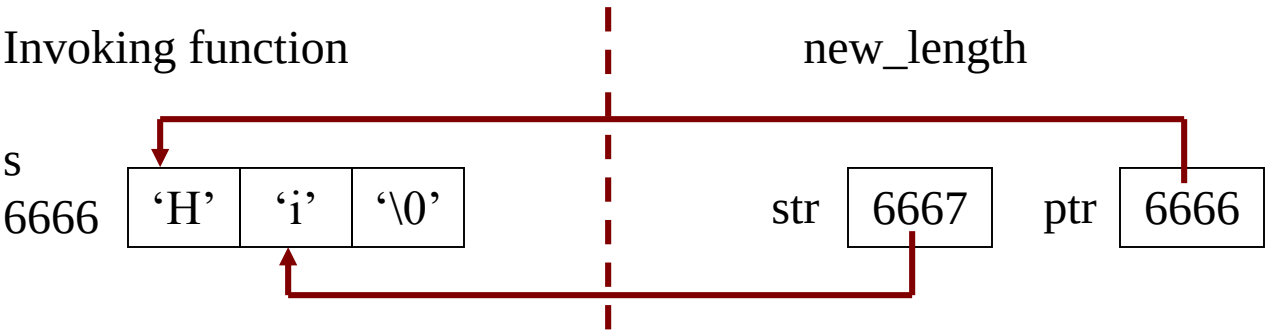
the output is

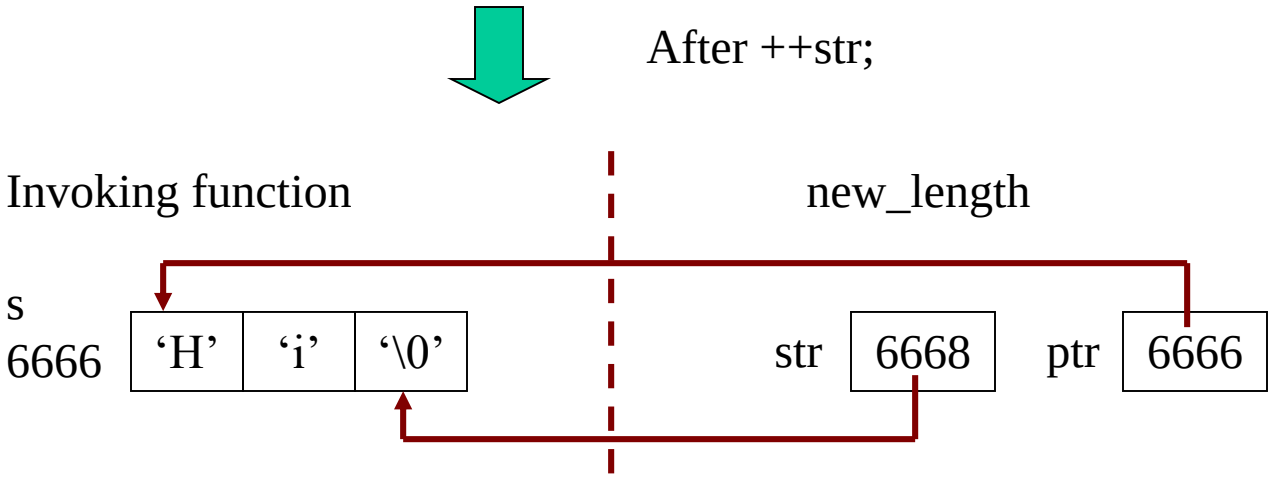
The length of s is 2.

Passing an array s to a function new_length



After ++str;





After second ++str, the value of *str is 0 (= '\0'), the while loop terminates. The value of expression

str - ptr

is 2.

The advantages of call by value and call by reference:

- Call by value protects the arguments. Because the function receives a copy of the argument's value and not the argument's address, the function cannot alter the value of the argument.

- If you want an invoked function to alter the value of a variable in the invoking function, and passing a pointer provides the means.

➢Real World Application: Reversing A String in Place

♦ ***Problem***

Write a program that reads a string and prints it in reverse.

♦ ***Problem***

Input is color; output is black.

Enter a string: STAR

Reversed string: RATS

♦ ***Solution***

The function **main** issue the prompt, reads the string, invokes a function **rev** that reverses the string, and writes the reversed string. The function **rev** reverses the string in place and uses pointer syntax.

♦ ***C Implementation***

```
/* This program reads a string of up to 100 characters with no embedded
   blanks and writes the reversed string. */
```

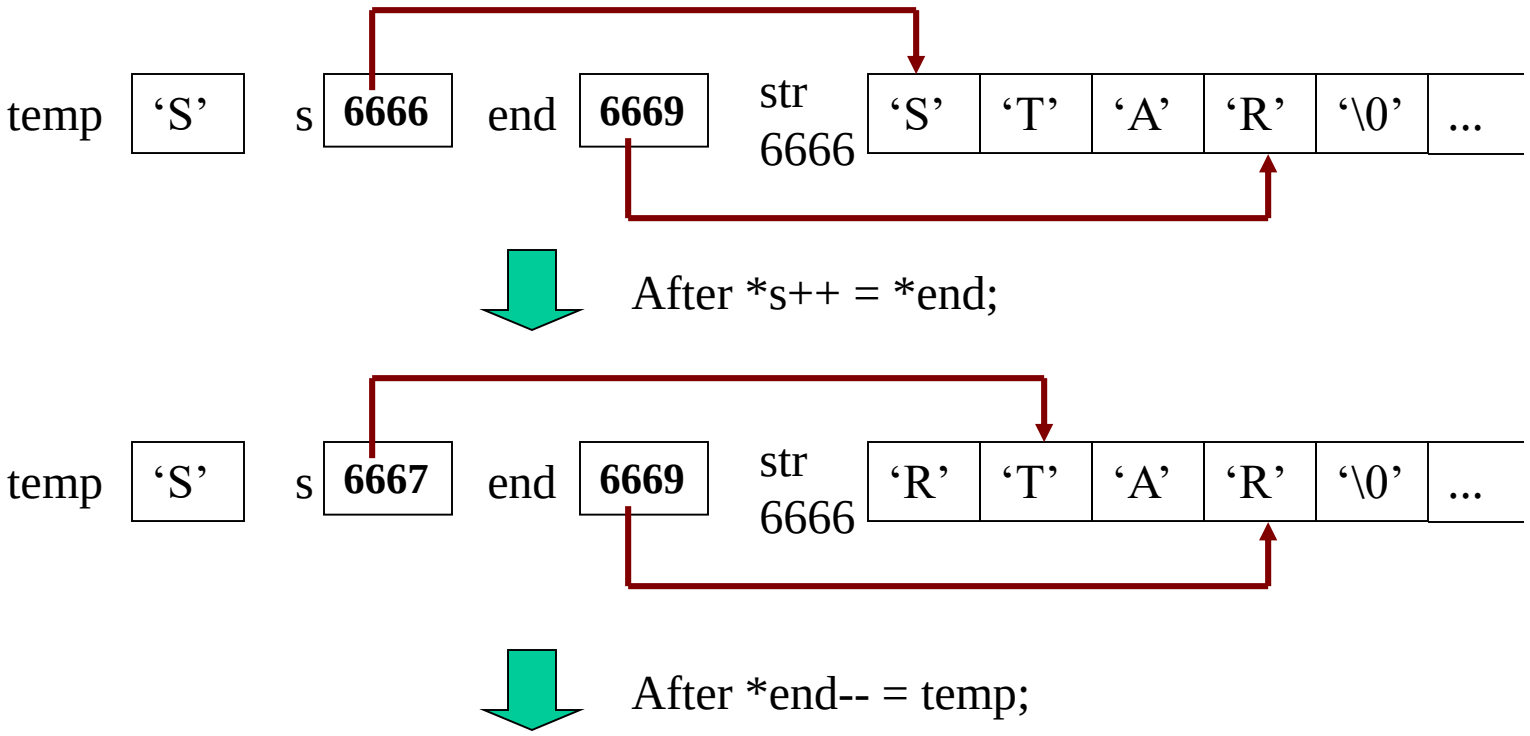
```
#include <stdio.h>
#include <stdlib.h>
void rev( char* s );
main ( ) {
        char str[ 101 ];        /* storage for up to 100 chars and a null terminator */
        printf( "\n\nEnter a string:\t" );
        scanf( "%s", str );
        rev( str );
        printf( "\n\nReversed string:\t%s\n", str );
        return EXIT_SUCCESS;
}
void rev( char* s ) {
        char temp, *end;
        end = s + strlen( s ) - 1    /* end points to last nonnull character in s */
        while( s < end ) {
                temp = *s;                  /*put content of *s to temp */
                *s++ = *end;                /*put content of *end to *s then move up */
                *end-- = temp;              /* put temp to *end and move end down*/
        }
}
```

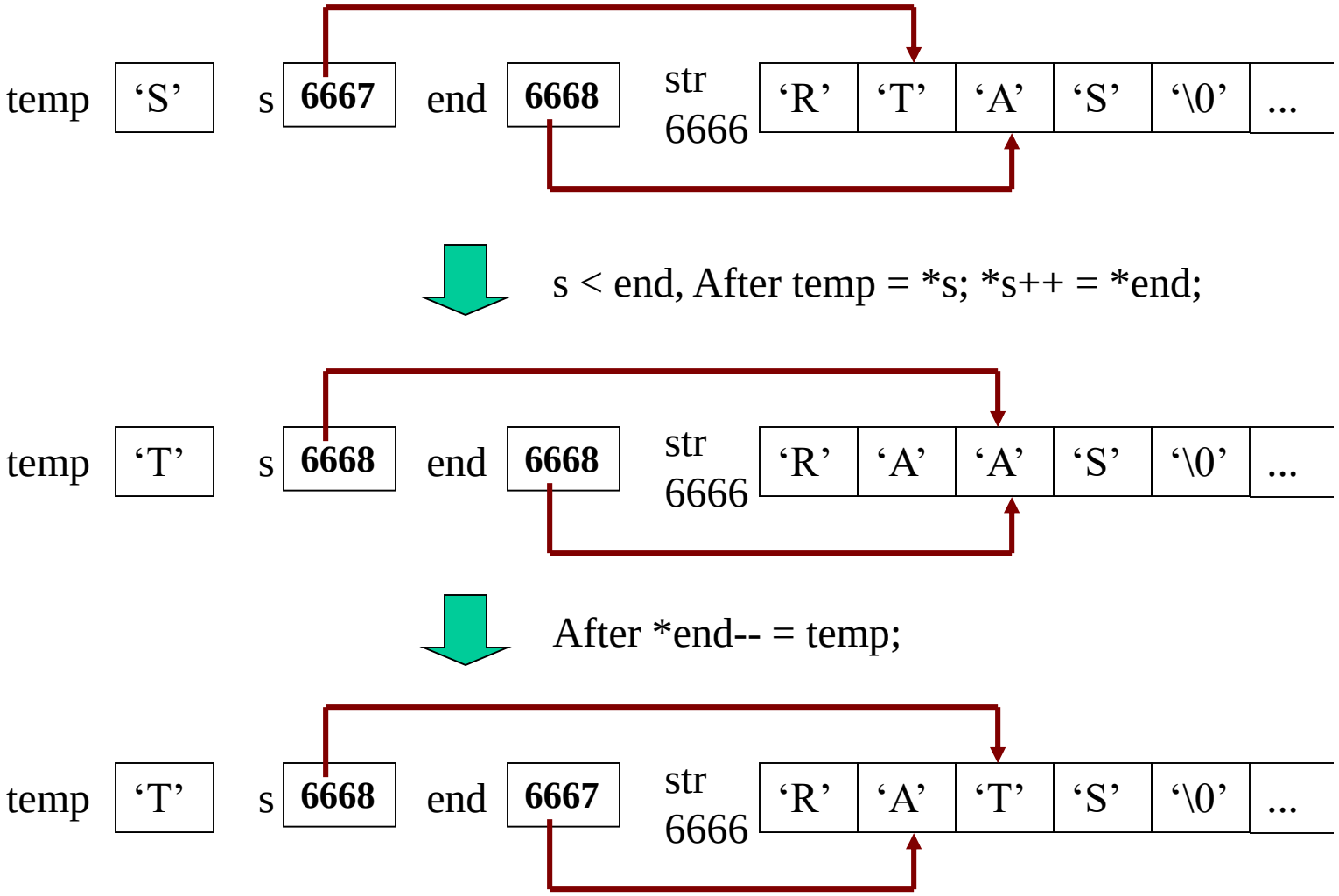
Disscussion

We show how the program executes if the string

STAR

is passed to **rev**.

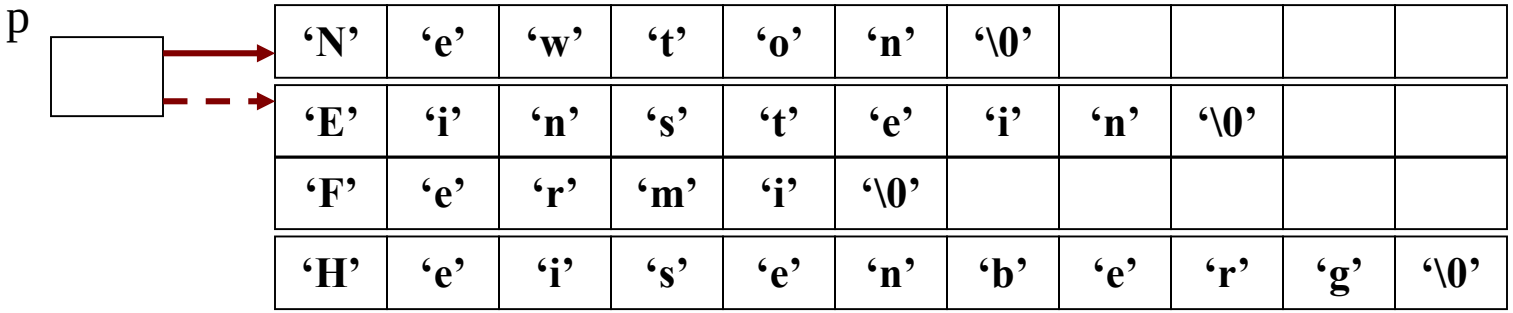



temp
'S'
s
6667
end
6668
str
6666
'R'
'T'
'A'
'S'
'\0'
...
s < end, After temp = *s; *s++ = *end;
temp
'T'
s
6668
end
6668
str
6666
'R'
'A'
'A'
'S'
'\0'
...
After *end-- = temp;
temp
'T'
s
6668
end
6667
str
6666
'R'
'A'
'T'
'S'
'\0'
...

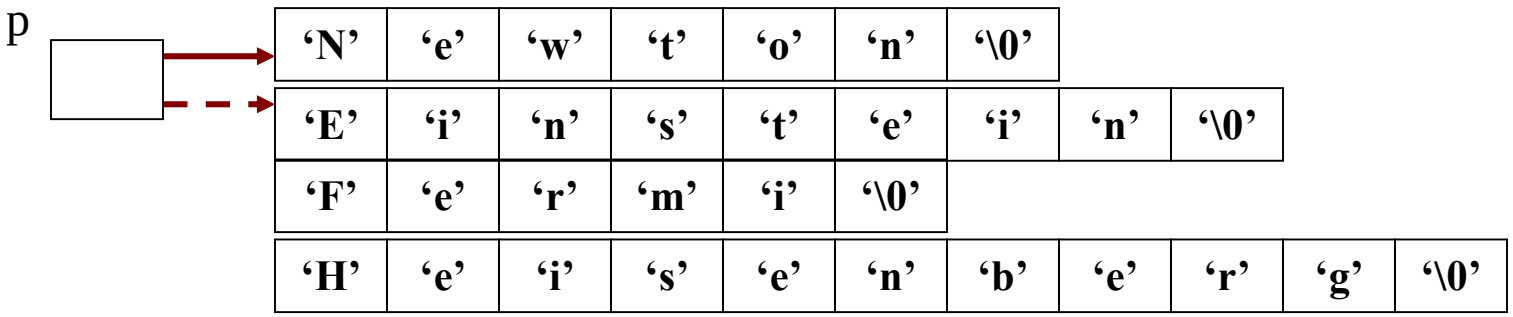
➢Pointers and Multidimensional Arrays

```
#include <stdio.h>
#include <stdlib.h>
void print_names( char p[ ] [ 11 ], int n );
main( ) {
        char phys[ 4 ] [ 11 ] = { “Newton”, “Einstein”, “Fermi”, “Heisenberg” );
        print_names( phys, 4 );
        return EXIT_SUCCESS;
}
void print_names( char p[ ] [ 11 ], int n ) {
        int I;
        for( I = 0; I < n; I++ )
                printf( “%s\n”, *p++ );
}
```

- A two-dimensional array is actually a one-dimensional array each of whose members is a one-dimensional array.




```
#include <stdio.h>
#include <stdlib.h>
void print_names( char *p[ ], int n );
main( ) {
      char* phys[ 4 ] = { "Newton", "Einstein", "Fermi", "Heisenberg" );
      print_names( phys, 4 );
      return EXIT_SUCCESS;
}
void print_names( char *p[ ], int n ) {
      int I;
      for( I = 0; I < n; I++ )
            printf( "%s\n", *p++ );
}
```



void print_names(char* p[ ], int n); ⟺ void print_names(char** p, int n);

➢Pointers to Functions

A function's name, like an array's name, is a **pointer constant**. The value of such a pointer constant can be regarded as the address of the code that represents the function.

To define the variable **ptr** to be of type "pointer to a function that has one parameter of type char and returns an **int** ", we write

int (***ptr**) (**char**);

Note: The asterisk and name must be enclosed in parentheses. If we write,

int *ptr (**char**);

we declaring **ptr** to be a function (as opposed to a pointer to a function) that has one parameter of type **char** and returns a pointer to an **int;** the parentheses have a higher precedence than the star does.

To invoke the function to which **ptr** points with the argument (letter) by dereferencing ptr and supplying the argument:

(***ptr**) (letter);

or

ptr(letter);

The two forms have identical meaning.

A parameter of type pointer to function is described in the usual way.

sum a function

returns no value

has parameter ptr

where ptr is a pointer to another function that has

one parameter char and returns int

void sum(**int** (*ptr) (**char**));

Sorting

- Arranging array elements in ascending or descending order
- Three simple sorting codes
  - selection
  - bubble
  - insertion

Sort using Selection

Assume ascending sorting of a[n]

- assume the min is the first element, min = 0
- compare 2nd element with a[min]
  - if a[1] > a[0],do nothing and continue
  - otherwise
    - set min =1
    - swap values between cell of new min and old min
    - continue
  - repeat above for j = 2, 3,...
  - end the array is reached; the min element is now in a[0]
- repeat above starting with element 2, a[1]
- until n-1 element contains the 2nd largest value in the array

For float a [n], use ordinary comparison operator
> or <

For char a [n], use

strcmp(string1,string2) to compare

strcpy (string1,string2) in swap

Sort using Bubble

Works well when array is almost sorted

- starts with first pass I = 1
- starts with j=0
- compare a[j] and a[j+1]
  - if in right order; do nothing then continue
  - otherwise interchange a[j] and a[j+1]; continue
- j=j+1; repeat above
- nth element is reached; largest element is at bottom
- I=2; repeat above; stops at (n-1)th element
- use flag to detect no interchange in a pass; break out

Insertion sorting

Assume that x[0] thru x[I] are in ascending order

1 examine x[I+1]

2 hold value of x[I+1] in temp

3 let j=i

4 is x[I+1] < x[j] and j >= 0?

True; replace x[j +1] with x[j]

j = j - 1; go to 4

False: x[j+1] = temp

these steps put x[0] thru x[I+1] in order

Run the above steps from I=0 to n-1

x[ ]={3,4,2,5,6,1}

I = 0; j=0

temp = 4

is 4 < 3?

No: x[1] = 4; x[ ]={3,4,2,5,6,1}

I =1; j = 1

temp = x[2] = 2

is 2 < 4

true: x[2] = 4; j=0; x[]={3,4,4,5,6,1}

is 2 < 3

true: x[1] = 3; x[ ]={3,3,4,5,6,1}

j = -1; x[0] = 2; x[]={2,3,4,5,6,1}

I =2; j =2

etc.

Binary searching

in an array for a matching element

Assume that x[0] thru x[n] are in ascending order, the value to be matched is xp

1. Start the search range from 0 to n
2. Find the middle element of x, call it x[loc]
3. If xp > x[loc], then set the search range to (loc,n)

   if xp < x[loc], then set the search range to (0,loc)

   if xp = x[loc], then it is found; break
4. go to 2