Lecture 6
Pointers-Part 4

Dr. Hacer Yalım Keleş

Ref: Programming in ANSI C, Kumar

Pointers to Pointers

A pointer provides the address of the data item pointed to by it. The data item pointed to by a pointer can be an address of another data item. Thus, a given pointer can be a pointer to a pointer to an object. Accessing this object from the given pointer then requires two levels of indirection. First, the given pointer is dereferenced to get the pointer to the given object, and then this later pointer is dereferenced to get to the object.

Consider, for example, the declarations

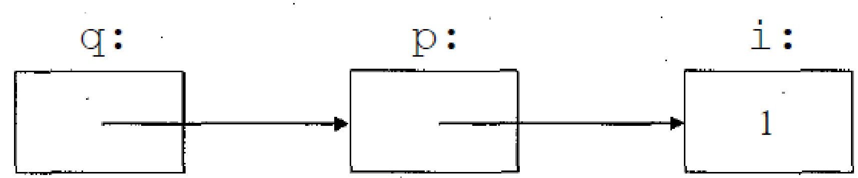
```
int i = 1;
int *p;
```

that declare two objects: `i,` an integer, and `p,` a pointer to an integer. We may apply the address operator & to both `i` and `p` as in

```
p  = & i;          / * make p  point to i * /
q  = &p;           /* make q  point to p  */
```

```
p  = & i;            / * make p  point to i * /
q  = &p;             /* make q  point to p  */
```

These statements imply that q is a pointer to the pointer p, which in turn points to the integer i. The relationship between i, p, and q is pictorially depicted below:



`int **q;`

`**q` means "apply the dereferencing operator to q twice"

In order to fetch the value of i, starting from q, we go through two levels of indirection. The value of *q is the content of p which is the address of i, and the value of * * q is * (& i) which is 1. Thus, each of the expressions

i + 1
*p + 1
**q + 1

has the value 2.

There is no limit on the number of levels of indirection, and a declaration such as

```
int ***p;
```

means

***p	is an integer,
**p	is a pointer to an integer,
*p	is a pointer to a pointer to an integer, and
p	is a pointer to a pointer to a pointer to an integer.

In the following definition of main

```
int main(int argc, char *argv[])
```

argv is really a pointer to a pointer, since C passes an array argument by passing the pointer to the first element of the array. Thus, the declaration

```
char *argv[]
```

can be replaced by an equivalent declaration

```
char **argv
```

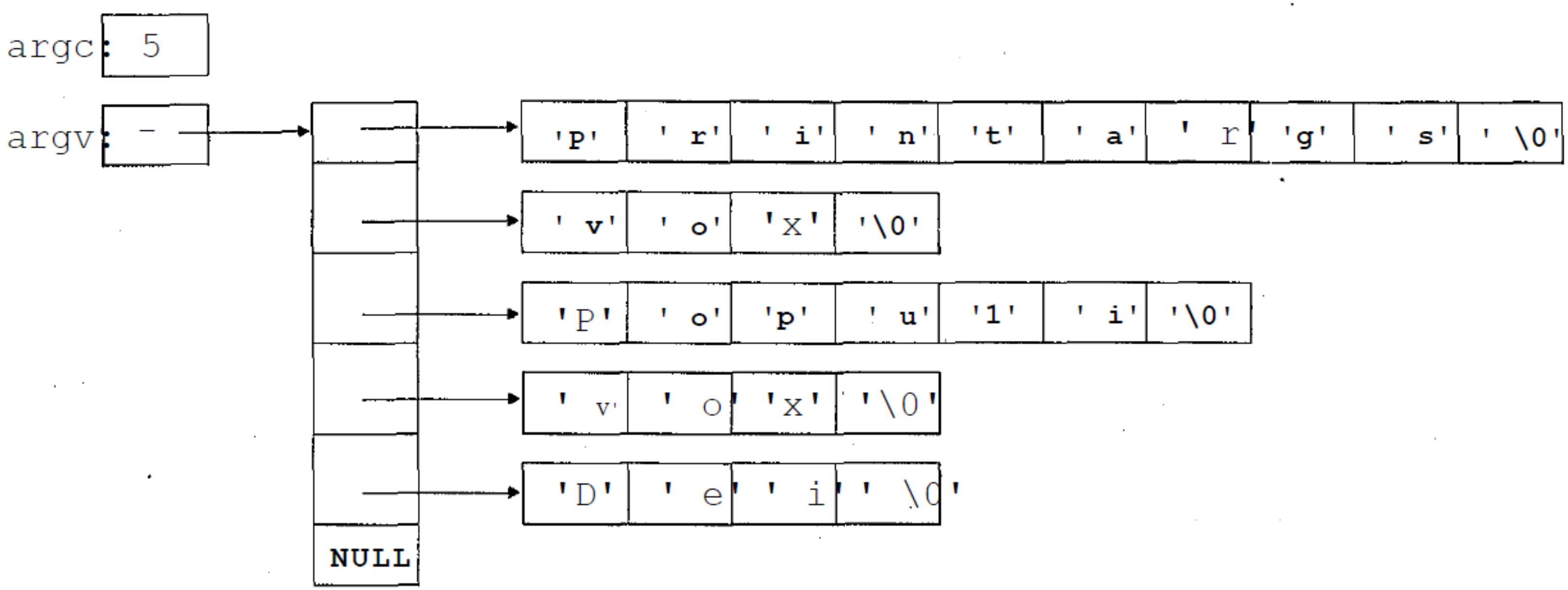
The following is the program printargs, rewritten treating argv as a pointer to a pointer:

```
#include <stdio.h>

int main(int argc, char **argv)
   {
   while (--argc) printf("%s ", *++argv);
      printf("\n");

      return 0;
   }
```

If the command line for a program `printargs` is

`printargs vox populi vox Dei`

then `argc` and `argv` will be as shown below:



```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h> /* defines INT_MAX */
#define MONTHS 12

char *mname[] =
  {
    "january", ''february", "march", "april",
    "may", "june", "july", "august", "September",
    "october", "november", "december"
  };

int  leap[]=
  {
    31, 29, 31, 30, 31, 30, 31,
    31, 30, 31, 30, 31, INT_MAX
  };
int regular[] =
  {
    31, 28, 31, 30, 31, 30, 31,
    31, 30, 31, 30, 31, INT_MAX
  };
int * days[] = {regular, leap };
```

Algorithm

Pseudocode to determine whether a year is a *leap year* or *not*

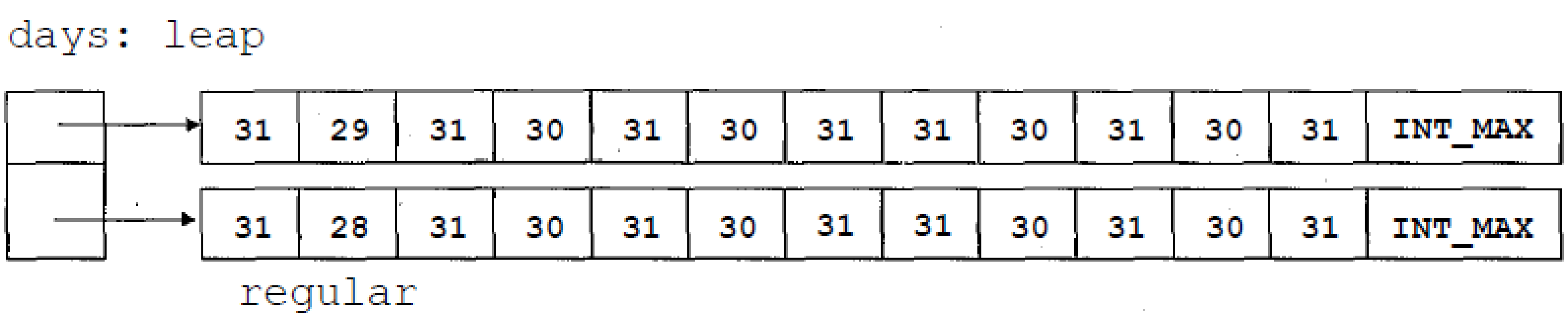
if year is divisible by 400 **then** is_leap_year
else if year is divisible by 100 **then** not_leap_year
else if year is divisible by 4 **then** is_leap_year
else not_leap_year

```
void datemonth(int day, int year,
                    int *pdate, int *pmonth)
   {
      int     leap , *pday;

      /* is it a normal or a leap year? */
         leap   =   ((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)

      /* get the appropriate element of days  array */
      pday = days[ leap ];

      /* find the month in which this day falls */
      while (day > *pday) day -= *pday++;

      *pdate = day;
      *pmonth = pday - days[ leap ];   /* how many times pday
   }                                      is incremented? */
```

```
char *name(int month)
  {
    return mname[month];
  }
```

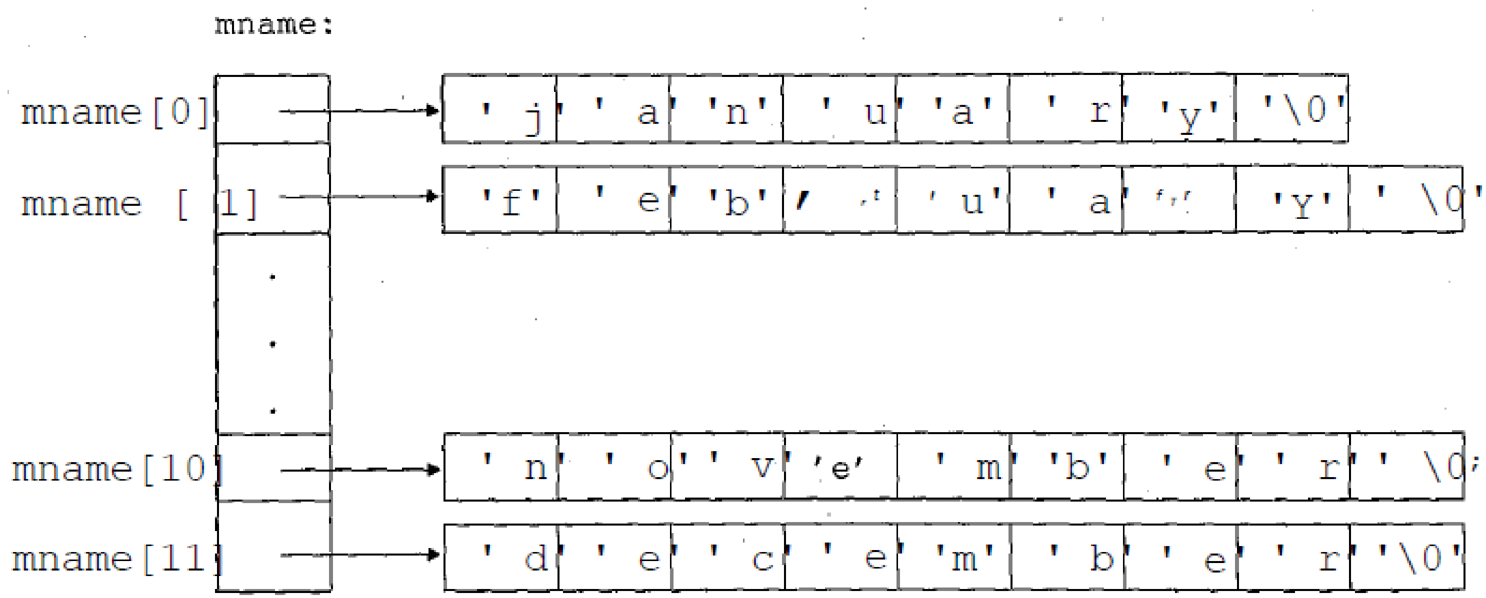
```
int. main(int argc, char *argv[])
  {
    int day, year, date, month;

    if (argc ! = .3)
      {
       printf("usage: %s <day> <year>\n", argv[0]);
       return 1;
      }

    day = atoi(argv[1]);
    year = atoi(argv[2]);

    datemonth(day, year, &date, &month);

    if (date > 0 && month < MONTHS)
      {
       printf("%d %s\n", date, name(month));
       return 0;
      }
    else
      {
       printf("an out-of-range day\n");
       return 1;
      }
  }
```

The function **datemonth** determines the date and month in which the given day of the year falls. It uses a pointer array **days** of two elements. As shown below, the first element points to an integer array **leap**, whose elements contain the number of days in the months of a leap year. The second element points to another integer array **regular**, whose elements contain the number of days in the months of a nonleap year.



The function `monthdate` first determines whether the given year is a regular or a leap year, and sets `pday` to point to the first element of the array `leap` if it is a leap year and `leap` turns out to be 1 ; otherwise, it sets `pday` to point to the first element of the array `regular`. The function then steps through the selected array by incrementing `pday` to determine the month in which the given `day` falls. The last dummy entry guarantees that we will not go out of bounds. This technique avoids the need for bounds checking inside the search loop, and is often used when searching an array for a value that may not be present.

The function **name** is straightforward. The pointer array **mname** stores the names of months as shown below:



The function `name` uses the given `month` to index into `mname` and returns a pointer to the corresponding name string.

POINTERS TO FUNCTIONS

In C, a function itself is not a variable, but it is possible to define a pointer to a function. A pointer to a function can best be thought of as the address of the code executed when the function is called.

```
int (*fp)(int i, int j);
```

declares `fp` to be a variable of type "pointer to a function that takes two integer arguments and returns an integer as its value." The identifiers `i` and `j` are written for descriptive purposes only. The preceding declaration can, therefore, also be written as

```
int (*fp) (int, int);
```

```
int (*fp) (int, int);
```

The parentheses around `*fp` are used to distinguish this declaration from that of a function, called `fp`, that returns a pointer to an integer. Keep in mind that:

`int i (void) ;`	declares `i` to be a function with no parameters that returns an `int`.
`int *pi (void) ;`	declares `pi` to be a function with no parameters that returns a pointer to an `int`.
`int (*ip) (void) ;`	declares `ip` to be a pointer variable that can be assigned to point to a function that returns an integer value and takes no arguments.

```
int (*fp)(int, int), gcd(int, int);
```

the function pointer `fp` can be made to point to the function `gcd` with the assignment

```
fp = gcd;
```

```
(*fp) (42, 56)
```

calls the function `gcd` pointed to by `fp`, passing 42 and 56 as arguments to it.

```
i = (*fp) (42, 56);
```

is equivalent to

```
i = gcd(42,56) ;
```

and assigns the value ofgcd(42,56), that is 14, to i.

DYNAMIC MEMORY MANAGEMENT

In many programs, the number of objects to be processed by the program and their sizes are not known *a priori*. One way to handle such situations is to make provision for the maximum number of objects expected and assume each to be of the maximum expected size. This is the approach taken in the example programs given in Section 6.3. This approach, however, may waste considerable memory and may even cause the size of the executable program to exceed the permissible size. Another disadvantage of this approach is that any guess eventually goes wrong and the program is presented with objects bigger and more in number than expected.

sizeof Operator

The `sizeof` operator is a unary operator that is used to obtain the size of a type or data object. It takes as its operand a parenthesized type name or an expression, and has the following forms:

`sizeof` *(typename)*
`sizeof` *expression*

The `sizeof` operator, when applied to a *typename,* yields the size in bytes of an object of the type named. For example, on an IBM PC, `sizeof (char)` is 1, `sizeof (int)` is 2,and `sizeof (long)` is4.

The `sizeof` operator, when applied to an *expression*, analyzes the expression at compile time to determine its type, and then yields the same result as if it had been applied to the type of the expression. For example, if

```
short s, *sp;
```

then

`sizeof (s)` is the same as `sizeof (short)`
`sizeof (sp)` is the same as `sizeof (short *)`
`sizeof (*sp)` isthesameas `sizeof (short)`

If the operand to `sizeof` is an n-element array of type T, the result of `sizeof` is n times the result of `sizeof` applied to the type T. Thus, if

```
int a[10];
```

then

```
sizeof (a);
```

is 20 on an IBM PC. Since a string constant is a null-terminated array of characters, `sizeof` applied to a string constant yields the number of characters in the string constant including the trailing '\0'. For example,

```
sizeof("computer")
```

is 9.

```
char c;
```

then `sizeof (c)` isthesameas `sizeof (char)`,
but `sizeof (c+0)` isthesameas `sizeof (int)`.

because the type of the expression `c+0`, after the usual type conversion, is `int`.

When **sizeof** is applied to an expression, the expression is compiled to determine its type; it is, however, not compiled into executable code, with the result that any side effects that are to be produced by the execution of the expression do not occur. Thus, on IBM PC, the expression

```
int i, j;
i = 1;
j = sizeof (--i);
```

assigns 2 to j, but **i** remains 1 after the assignment.

Dynamic Memory Management Functions

The four dynamic memory management functions are **`malloc`**, **`calloc`**, **`realloc`**, and **`free`**.

The functions **`malloc`** and **`calloc`** are used to obtain storage for an object,

The function **`realloc`** for changing the size of the storage allocated to an object,

The function **`free`** for releasing the storage.

```
void *malloc (size_t size);
```

The function **malloc** allocates storage for an object whose size is specified by *size*. It returns a pointer to the allocated storage and **NULL** if it is not possible to allocate the storage requested. The allocated storage is not initialized in any way.

For example, if

```
float *fp, fa[10];
```

then

```
fp = (float *) malloc(sizeof(fa));
```

allocates the storage to hold an array of 10 floating-point elements and assigns the pointer to this storage to **`fp.`** Note that the generic pointer returned by **`malloc`** has been coerced into **`float`** * before it is assigned to **`fp.`**

The function **`calloc`** allocates the storage for an array of *nobj* objects, each of size *size*. It returns a pointer to the allocated storage and **`NULL`** if it is not possible to allocate the storage requested. The allocated storage is initialized to zeros.

For example, if

```
double *dp, da[10];
```

then

```
dp = (double *) calloc(10, sizeof(double));
```

allocates the storage to hold an array of 10 `double` values and assigns the pointer to this storage to `dp`.

`void *realloc (void *p, size_t size);`

The function `realloc` changes the size of the object pointed to by *p* to *size*. It returns a pointer to the new storage and `NULL` if it is not possible to resize the object, in which case the object (*p*) remains unchanged. The new size may be larger or smaller than the original size. If the new size is larger, the original contents are preserved and the remaining space is uninitialized; if smaller, the contents are unchanged up to the new size.

For example, if

```
char *cp;
cp = (char *) malloc(sizeof("computer"));
strcpy(cp, "computer");
```

then `cp` points to an array of 9 characters containing the null-terminated string `computer`. The function call

```
cp.= (char *) realloc (cp, sizeof("compute”));
```

discards the trailing `'\0'` and makes `cp` point to an array of 8 characters containing the characters in `computer`, whereas the call

```
cp = (char *) realloc(cp, sizeof("computerization"));
```

makes `cp` point to an array of 16 characters, the first 9 of which contain the null-terminated string `computer` and the remaining 7 are uninitialized.

```
void free(void *p);
```

The function `free` deallocates the storage pointed to by p, where p is a pointer to the storage previously allocated by `malloc`, `calloc`, or `realloc`. If p is a null pointer, `free` does nothing. For example, the storage allocated through calling `malloc`, `calloc`, and `realloc` in the preceding examples can be deallocated by

```
free(fp);
free(dp);
free(cp);
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define STRING "quis custodiet ipsos custodes"

int main(void)
  {
    char *cp;

    cp = (char *) malloc(sizeof(STRING));
    if (!cp)
      {
        printf("no memory\n");
        return 1;
      }

    strcpy(cp, STRING);

    cp = (char *) realloc(cp, sizeof(STRING) + 1);
    if (!cp)
      {
        printf("no memory\n");
        return 1;
      }

    printf("%s\n", strcat(cp, "?"));

    free(cp);

    return 0;
  }
```