UES103

Programming for Problem Solving

Saif Nalband, PhD

1A27-31/1A42-46

Lab Password: cslab768

Pointers

Basic Concept

In memory, every data item occupies one or more contiguous memory cells.

- **A cell in memory is typically a byte. The number of memory cells required to store a data item depends on its type (char, int, double, etc.).**

Whenever we declare a variable, the system allocates the required amount of memory cells to hold the value of the variable.

- **Since every byte in memory has a unique address, this location also has its own (unique) address. For a multi-byte data, this is usually specified by the address of the first byte.**

C allows you to play with addresses.

Accessing the Address of a Variable

The address of a variable can be determined using the '&' operator.

- **The operator '&' immediately preceding a variable returns the *address* of the variable**
- **& is the "address-of" operator**

Example: `&xyz`

The '&' operator can be used only with a *simple variable* or an *array element*.

```
&distance
&x[0]
```

Following usages are illegal:

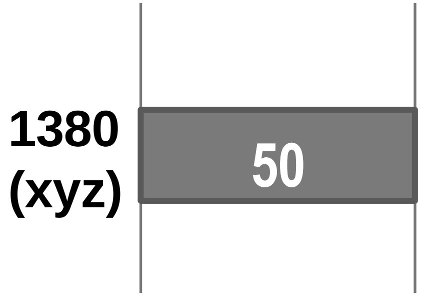
`&235` – **address of a constant is not defined**

`&(a+b)` – **address of an expression is not defined**

Example

Consider the statement

```
int  xyz = 50;
```



- This statement instructs the compiler to allocate a location for the integer variable xyz, and put the value 50 in that location.

- Suppose that the (starting) address location chosen is 1380.
- During execution of the program, the system always associates the name xyz with the address 1380.

- The value 50 can be accessed by using either the name (xyz) or by looking at whatever is written in the address (&xyz which equals 1380 in this example).

Pointer Declaration

A pointer is just a C variable whose value is the address of another variable! Pointer variables must be declared before we use them.

General form:

data_type *pointer_name;

Example:

```
int *ptr;
```

Three things are specified in the above declaration:

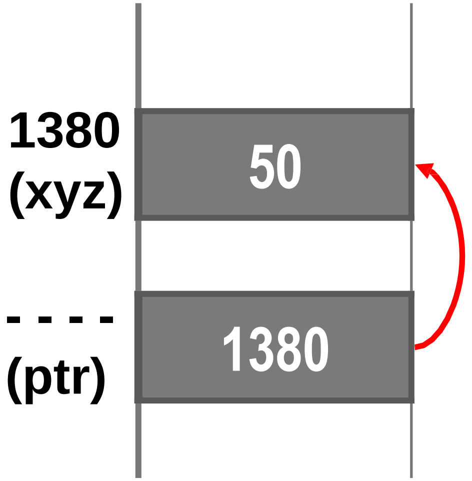
- The asterisk (*) tells that the variable ptr is a pointer variable.
- ptr will be used to point to a variable of type int.

Just after declaring a pointer, ptr does not actually point to anything yet (remember: a pointer is also a variable; hence can contain garbage until it is assigned to some specific value). You can iniialize or set a pointer to the NULL pointer which points nowhere: int *ptr = NULL;

Pointers are variables and are stored in the memory. They too have their own addresses (like &ptr).

Example (Contd.)

```
int xyz = 50;
int *ptr;     // Here ptr is a pointer to an integer
ptr = &xyz;
```



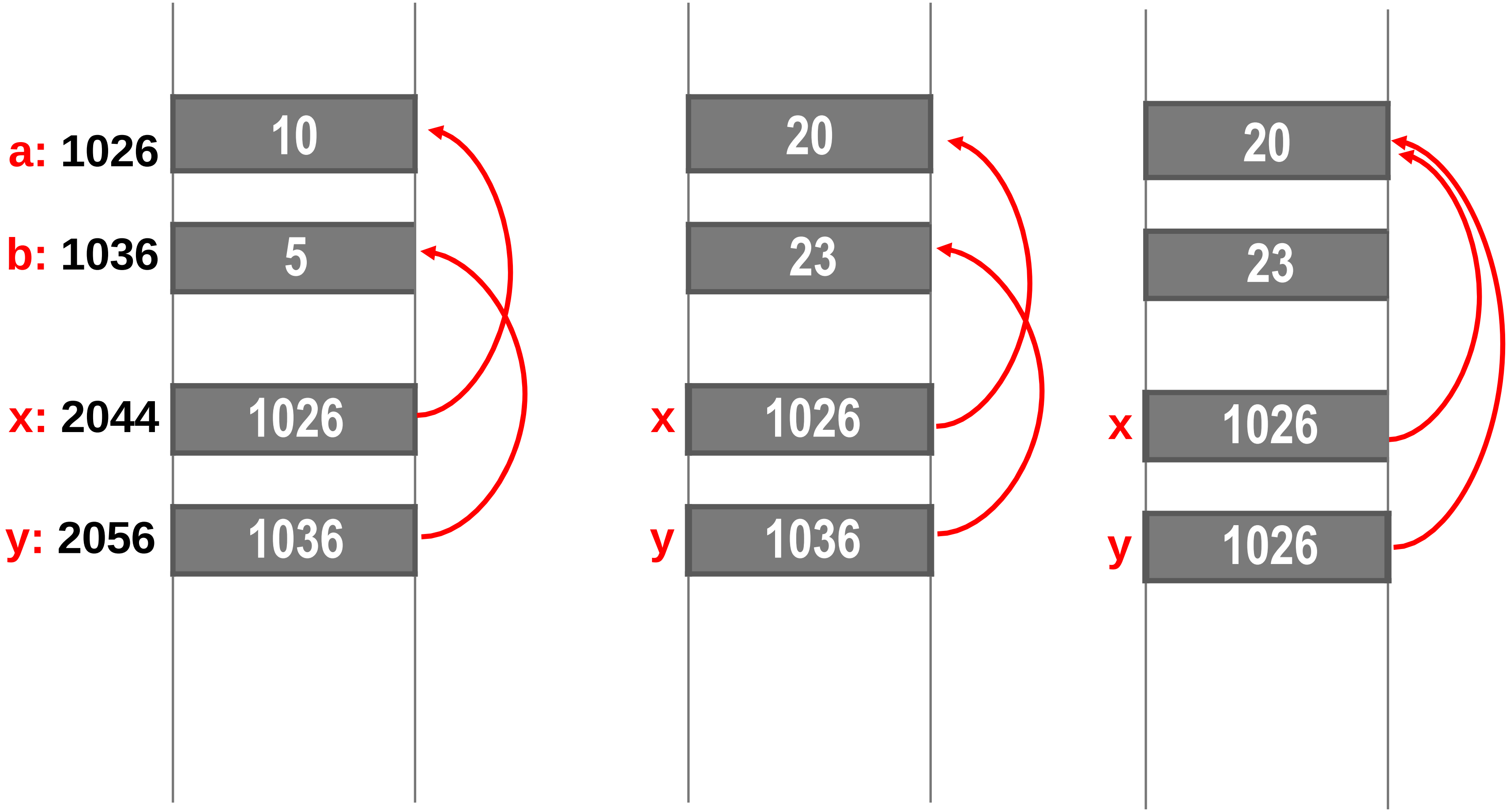
Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory.

- **Such variables that hold memory addresses are called *pointers*.**
- **Since a pointer is a variable, its value is also stored in some memory location.**

Once ptr has been assigned a valid memory address, the * operator can be used to access the value at that address. * is the "value-at" operator; can be used only with a pointer variable

Example: Making a pointer point to a variable

```
int a = 10, b = 5;
int *x, *y;
x = &a;  y = &b;
*x = 20;
*y = *x + 3;
y = x;
```



Given a pointer variable, we can either:

- make it point to (i.e., store the address of) some existing variable, or
- dynamically allocate memory and make it point to it (to be discussed later)

Things to Remember

Pointers have types, e.g. :

```
int *count;
float *speed;
```

Pointer variables should always point to a data item of the *same type*.

```
double x;
int *p;
p = &x;    // You should not generally do this, compiler will complain
```

However, type casting can be used in some circumstances – we will see examples later.

```
p = (int *)&x;
```

Pointers and arrays

Pointers and arrays

When an array is declared:

- **The array has a *base address* and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.**
- **The *base address* is the location of the first element (*index 0*) of the array.**
- **The compiler also defines the array name as a *constant pointer* to the first element.**

Example

Consider the declaration:

```
int x[5] = {1, 2, 3, 4, 5};
int *p;
```

- Suppose that the base address of x is 2500, and each integer requires 4 bytes.

Element	Value	Address
x[0]	1	2500
x[1]	2	2504
x[2]	3	2508
x[3]	4	2512
x[4]	5	2516

Both x and &x[0] have the value 2500.

p = x; and p = &x[0]; are equivalent.

Example (contd)

```
int x[5] = {1, 2, 3, 4, 5};
int *p;
```

- Suppose we assign p = &x[0];
- Now we can access successive values of x by using p++ or p-- to move from one element to another.

Relationship between p and x:

p = &x[0] = 2500

p+1 = &x[1] = 2504

p+2 = &x[2] = 2508

p+3 = &x[3] = 2512

p+4 = &x[4] = 2516

(p+i) gives the address of x[i]

(p+i) is the same as &x[i]

*(p+i) gives the value of x[i]

For any array A, we have: A+i = &A[i] is the address of A[i], and *(A+i) = A[i].

Printing pointers with %p

```
#include <stdio.h>
int main ()
{
  int A[4] = {2, 3, 5, 7}, i, *p;
  for (i=0; i<4; ++i)
     printf("&A[%d] = %p, A[%d] = %d\n", i, A+i, i, *(A+i));
  p = A;
  printf("p = %p, &p = %p\n", p, &p);
  return 0;
}
```

Output

```
&A[0] = 0x7ffd66659050, A[0] = 2
&A[1] = 0x7ffd66659054, A[1] = 3
&A[2] = 0x7ffd66659058, A[2] = 5
&A[3] = 0x7ffd6665905c, A[3] = 7
p = 0x7ffd66659050, &p = 0x7ffd66659048
```

But then, what is &A?

It is not an int pointer.

Pointer to an array vs pointer to a pointer

```c
#include <stdio.h>
int main ()
{
  int A[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 8, 7, 6, 5, 4, 3, 2}, *p;
  printf("A      = %p\n", A);
  printf("A + 1  = %p\n", A + 1);
  printf("&A     = %p\n", &A);
  printf("&A + 1 = %p\n\n", &A + 1);
  p = A;
  printf("p      = %p\n", p);
  printf("p + 1  = %p\n", p + 1);
  printf("&p     = %p\n", &p);
  printf("&p + 1 = %p\n", &p + 1);
  return 0;
}
```

Output

```
A      = 0x7ffd428a9520
A + 1  = 0x7ffd428a9524
&A     = 0x7ffd428a9520
&A + 1 = 0x7ffd428a9560

p      = 0x7ffd428a9520
p + 1  = 0x7ffd428a9524
&p     = 0x7ffd428a9518
&p + 1 = 0x7ffd428a9520
```

Pointer expressions
Pointer arithmetic

Pointers in Expressions

Like other variables, pointer variables can be used in expressions.

If p is an int pointer, then *p is an int variable (like any other int variable). If `p1` and `p2` are two pointers, the following statements are valid:

```
sum = (*p1) + (*p2);
prod = (*p1) * (*p2);
*p1 = *p1 + 2;
x = *p1 / *p2 + 5;
```

You can do arithmetic on pointers themselves

What are allowed in C?

- **Add an integer to a pointer.**
- **Subtract an integer from a pointer.**
- **Subtract one pointer from another.**
  - **If p1 and p2 are both pointers to the same array, then p2–p1 gives the number of elements between p1 and p2.**

Pointer arithmetic

What are not allowed?

- **Add two pointers.**

```
p1 = p1 + p2;
```

- **Multiply / divide a pointer in an expression.**

```
p1 = p2 / 5;
p1 = p1 - p2 * 10;
```

Scale Factor

We have seen that an integer value can be added to or subtracted from a pointer variable.

```
int x[5] = {10, 20, 30, 40, 50};
int *p;

p = &x[1];
printf ("%d", *p);  // This will print 20

p++;                                    // This increases p by the number of bytes
forint
printf ("%d", *p);  // This Will print 30

p = p + 2;                    // This increases p by twice the sizeof(int)
printf ("%d", *p);  // This will print 50
```

More on Scale Factor

```
char carr[5] = {'A', 'B', 'p', '?', 'S'};
int darr[5] = {10, 20, 30, 40, 50}
char *p;  int *q;

p = carr;    // The pointer p now points to the first element of carr
q = darr;           // The pointer q now points to the first element of darr
p = p + 1;   // Now p points to the second element in the array "carr"
q = q + 1;   // Now q points to the second element in the array "darr"
```

When a pointer variable is increased by 1, the increment is not necessarily by one byte, but by the *size of the data type* to which the pointer points.

This is why pointers have types (like int pointers, char pointers). They are not just a single "address" data type.

Pointer types and scale factor

Data Type	Scale Factor
char	1
int	4
float	4
double	8

- **If p1 is an int pointer, then**

```
p1++
```

will increment the value of p1 by 4.

- **If p2 is a double pointer, then**

```
p2--
```

will decrement the value of p2 by 8.

Scale factor may be machine dependent

- The exact scale factor may vary from one machine to another.
- Can be found out using the `sizeof` operator.
- You can supply a variable name or a variable type to it to get its size.

```
#include  <stdio.h>
main( )
{
  printf ("No. of bytes occupied by int is %d \n",   sizeof(int));
  printf ("No. of bytes occupied by float is %d \n",sizeof(float));
  printf ("No. of bytes occupied by double is %d \n", sizeof(double));
  printf ("No. of bytes occupied by char is %d \n",    sizeof(char));
}
```

Output

Number of bytes occupied by int is 4
Number of bytes occupied by float is 4
Number of bytes occupied by double is 8
Number of bytes occupied by char is 1

Example of scale factors

```
#include <stdio.h>
int main ()
{
   char C[10], *cp;
   int I[20], *ip;
   float F[30], *fp;
   double D[40], *dp;
   cp = C; printf("cp = %p, cp + 1 = %p\n", cp, cp+1);
   ip = I; printf("ip = %p, ip + 1 = %p\n", ip, ip+1);
   fp = F; printf("fp = %p, fp + 1 = %p\n", fp, fp+1);
   dp = D; printf("dp = %p, dp + 1 = %p\n", dp, dp+1);
   return 0;
}
```

Output

```
cp = 0x7ffd297f1d8e, cp + 1 = 0x7ffd297f1d8f
ip = 0x7ffd297f1b70, ip + 1 = 0x7ffd297f1b74
fp = 0x7ffd297f1bc0, fp + 1 = 0x7ffd297f1bc4
dp = 0x7ffd297f1c40, dp + 1 = 0x7ffd297f1c48
```

Pointers and functions

Passing pointers to a function

In C, arguments are passed to a function *by value*.

- The data items are copied to the function.
- Changes made in the called function are not reflected in the calling function.

Pointers are often passed to a function as arguments.

- Allows data items within the calling function to be accessed by the called function (through their address) and modified.

Passing pointers as arguments to functions

```
#include  <stdio.h>
int main()
{
     int  a, b;
     a = 5;  b = 20;
     swap (a, b);
     printf ("a = %d, b = %d\n", a, b);
}

void  swap (int x, int y)
{
     int  t;
     t = x;  x = y;  y = t;
}
```

```
#include  <stdio.h>
int main()
{
     int  a, b;
     a = 5;  b = 20;
     swap (&a, &b);
     printf ("a = %d, b = %d\n", a, b);
}

void swap (int *x, int *y)
{
   int  t;
   t = *x;   *x = *y;   *y = t;
}
```

Output

a = 5, b = 20

Output

a = 20, b = 5

A useful application of pointers

In C, a function can only return a single value.

Suppose you want to write a function that computes two values. How to send both the computed values back to the calling function (e.g., main function)?

One way:

- **Declare variables within the main (calling) function.**
- **Pass addresses of these variables as arguments to the function.**
- **The called function can directly store the computed values in the variables declared within main.**

Example of "returning" multiple values using pointers

```
#include <stdio.h>
int f ( int a, int b, int *p, int *q )
{
   *p = a + b;
   *q = a - b;
   return a * b;
}
int main ()
{
   int u = 55, v = 34, x, y, z;
   z = f (u, v, &x, &y);
   printf("x = %d, y = %d, z = %d\n", x, y, z);
}
```

Output

x = 89, y = 21, z = 1870

Pointers or arrays in function prototypes?

There is no difference among the following functions prototypes.

```
... func_name ( int A[], ... );
... func_name ( int A[100], ... );
... func_name ( int *A, ... );
```

In all the cases, A is an int pointer. It does not matter whether the actual parameter is the name of an int array or of an int pointer. Inside the function, A is a copy of the address passed.

For readability, use the following convention.

- **If the parameter passed is a pointer to an individual item (like x = &a in the swap example), use the pointer notation in the function prototype.**
- **If the parameter passed is an array, you can use any of the two notations in the function prototype. The array notation may be preferred for readability.**

A function can return a pointer

A program to locate the first upper-case letter (if any) in a string

```
#include <stdio.h>
char *firstupper ( char S[] )  // You can use char *S as the formal parameter
{
   while (*S) if ((*S >= ‘A’) && (*S <= ‘Z’)) return S; else
   ++S; return NULL;
}
int main ()
{
   char *p, S[100];
   scanf(“%s”, S);
   p = firstupper(S);
   if (p) printf(“%c found\n”, *p); else printf(“No upper-case letter
   found\n”); return 0;
}
```

Note: A function should not return a pointer to a local variable. After the function returns, the local variable no longer exists.

Multi-Dimension Arrays

Two Dimensional Arrays

We have seen that an array variable can store a list of values.

Many applications require us to store a table of values.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Student 1	75	82	90	65	76
Student 2	68	75	80	70	72
Student 3	88	74	85	76	80
Student 4	50	65	68	40	70

Two Dimensional Arrays

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Student 1	75	82	90	65	76
Student 2	68	75	80	70	72
Student 3	88	74	85	76	80
Student 4	50	65	68	40	70

The table contains a total of 20 values, five in each line.

- **The table can be regarded as a matrix consisting of four rows and five columns.**

C allows us to define such tables of items by using two-dimensional arrays.

Declaring 2-D Arrays

General form:

```
type array_name[row_size][column_size];
```

Examples:

```
int  marks[4][5];
float  sales[12][25];
double  matrix[100][100];
```

First index indicates row, second index indicates column.

Both row index and column index start from 0 (similar to what we had for 1-d arrays)

Declaring 2-D Arrays

```
int  m[4][5];
```

	Column 0	Column 1	Column 2	Column 3	Column 4
Row 0	m[0][0]	m[0][1]	m[0][2]	m[0][3]	m[0][4]
Row 1	m[1][0]	m[1][1]	m[1][2]	m[1][3]	m[1][4]
Row 2	m[2][0]	m[2][1]	m[2][2]	m[2][3]	m[2][4]
Row 3	m[3][0]	m[3][1]	m[3][2]	m[3][3]	m[3][4]

Accessing Elements of a 2-D Array

Similar to that for 1-D array, but use two indices.

- **First index indicates row, second index indicates column.**
- **Both the indices should be expressions which evaluate to integer values.**

Examples:

```
x[m][n] = 0;
c[i][k] += a[i][j] * b[j][k];
val = sqrt(arr[j*3][k+1] );
```

How is a 2-D array stored in memory?

Starting from a given memory location (starting address of the array), the elements are stored row-wise in consecutive memory locations.

- **x: starting address of the array in memory**
- **c: number of columns**
- **k: number of bytes allocated per array element, e.g., sizeof(int)**

- `a[i][j]` **is allocated memory location at address** `x + (i * c + j) * k`

a[0][0] a[0][1] a[0][2] a[0][3]	a[1][0] a[1][1] a[1][2] a[1][3]	a[2][0] a[2][1] a[2][2] a[2][3]
Row 0	**Row 1**	**Row 2**

Array Addresses

```
int main()
 {
  int a[3][5];
  int i, j;

  for (i=0; i<3;i++)
  {
    for (j=0; j<5; j++)
        printf ("%u\n", &a[i][j]);
    printf ("\n");
  }
  return 0;
 }
```

Output

```
3221224480
3221224484
3221224488
3221224492
3221224496

3221224500
3221224504
3221224508
3221224512
3221224516

3221224520
3221224524
3221224528
3221224532
3221224536
```

How to read the elements of a 2-D array?

By reading them one element at a time

```
for (i=0; i<nrow; i++)

    for (j=0;j<ncol;j++)
        scanf  ("%f", &a[i][j]);
```

- **The ampersand (&) is necessary.**
- **The elements can be entered all in one line or in different lines.**

We can also initialize a 2-D array at the time of declaration:

```
int a[MAX_ROWS][MAX_COLS] = { {1,2,3}, {4,5,6}, {7,8,9}};
```

How to print the elements of a 2-D array?

By printing them one element at a time.

```
for (i=0; i<nrow; i++)

   for (j=0; j<ncol; j++)
      printf  ("%f  ", a[i][j]);
```

This will print all elements in one line.

```
for (i=0; i<nrow; i++) {
   for (j=0; j<ncol; j++)
      printf ("%f  ", a[i][j]);
        printf("\n");
}
```

This will print the elements with one row in each line (matrix form).

Example: Matrix addition

```
int main()
{
    int  a[100][100], b[100][100],
               c[100][100], p, q, m, n;

    printf ("Enter dimensions: ");
    scanf ("%d %d", &m, &n);

    for  (p=0; p<m; p++)
         for  (q=0; q<n; q++)
              scanf ("%d", &a[p][q]);

    for  (p=0; p<m; p++)
         for  (q=0; q<n; q++)
              scanf ("%d", &b[p][q]);
```

```
   for (p=0; p<m; p++)
      for  (q=0; q<n; q++)
          c[p][q] = a[p][q] + b[p][q];

     for (p=0; p<m; p++)
     {
          for (q=0; q<n; q++)
              printf ("%d ", c[p][q]);
              printf  ("\n");
      }
     return 0;
}
```

A 2-D array is an array or 1-D arrays, and so a row pointer

```
#include <stdio.h>
int main ()
{
   int i, j, A[4][5] = { { 7, 14,  3, 16,  6}, {11,  5,  9, 13, 18},
                         { 2, 15, 20,  1, 19}, {10,  4, 12, 17,  8} };
   for (i=0; i<4; ++i) {
      for (j=0; j<5; ++j) printf("%p ", &A[i][j]);
          printf("\n");
   }
   printf("sizeof(A)  = %3lu,   A = %p,   A + 1 = %p\n", sizeof(A),   A,  A + 1);
   printf("sizeof(*A) = %3lu,  *A = %p,  *A + 1 = %p\n", sizeof(*A), *A, *A + 1);
   printf("sizeof(&A) = %3lu,  &A = %p,  &A + 1 = %p\n", sizeof(&A), &A, &A + 1);
   return 0;
}
```

Output

```
0x7ffc314fe100 0x7ffc314fe104 0x7ffc314fe108 0x7ffc314fe10c 0x7ffc314fe110
0x7ffc314fe114 0x7ffc314fe118 0x7ffc314fe11c 0x7ffc314fe120 0x7ffc314fe124
0x7ffc314fe128 0x7ffc314fe12c 0x7ffc314fe130 0x7ffc314fe134 0x7ffc314fe138
0x7ffc314fe13c 0x7ffc314fe140 0x7ffc314fe144 0x7ffc314fe148 0x7ffc314fe14c
sizeof(A)  =  80,   A = 0x7ffc314fe100,   A + 1 = 0x7ffc314fe114
sizeof(*A) =  20,  *A = 0x7ffc314fe100,  *A + 1 = 0x7ffc314fe104
sizeof(&A) =   8,  &A = 0x7ffc314fe100,  &A + 1 = 0x7ffc314fe150
```

Passing 2-d arrays to functions

Passing 2-D arrays to functions

Similar to that for 1-D arrays.

- **The array contents are not copied into the function.**
- **Rather, the address of the first element is passed.**

For calculating the address of an element in a 2-D array, the function needs:

- **The starting address of the array in memory (say, x)**
- **Number of bytes per element (say, k)**
- **Number of columns in the array, i.e., the size of each row (say, c)**

`a[i][j]` is located at memory address $x + (i * c + j) * k$

The above three pieces of information must be known to the function.

Example

```
int main()
{
    int  a[15][25],  b[15][25];
    …
    …
    add (a, b, 15, 25);
    …
    …
}
```

```
void  add (int x[][25], int y[][25],
                        int rows, int cols)
{

}
```

We can also write

int x[15][25], y[15][25];

The first dimension is ignored. But the second dimension *must* be given.

Example: Matrix addition with functions

```
void ReadMatrix (int A[][100], int x, int y)
{
    int i, j;
    for (i=0; i<x; i++)
        for (j=0; j<y; j++)
            scanf (“%d”, &A[i][j]);
}
```

```
void AddMatrix( int A[][100], int B[][100], int C[][100], int x, int y)
{
    int i, j;
    for (i=0; i<x; i++)
        for (j=0; j<y; j++)
            C[i][j] = A[i][j] + B[i][j];
}
```

Example: Matrix addition

```
void PrintMatrix (int A[][100], int x, int y)
{
    int i, j;
    printf ("\n");
    for (i=0; i<x; i++)
    {
       for (j=0; j<y; j++)
           printf (" %5d", A[i][j]);
       printf("\n");
    }
}
```

```
int main()
{
    int  a[100][100], b[100][100],
         c[100][100], p, q, m, n;

    scanf ("%d%d", &m, &n);

    ReadMatrix(a, m, n);
    ReadMatrix(b, m, n);

    AddMatrix(a, b, c, m, n);

    PrintMatrix(c, m, n);
    return 0;
}
```

Example:

```
#include  <stdio.h>
int main() {
  int  a[15][25], b[15][25], c[15][25];
  int m, n;
  scanf ("%d %d", &m, &n);
  for (p=0; p<m; p++)
     for  (q=0; q<n; q++)
        scanf ("%d", &a[p][q]);
  for (p=0; p<m; p++)
     for (q=0; q<n; q++)
        scanf ("%d", &b[p][q]);
  add (a, b, m, n, c);
  for (p=0; p<m; p++)  {
     for  (q=0; q<n; q++)
        printf("%f   ", c[p][q]);
     printf("\n");
    }
}
```

```
void add (int x[][25], int y[][25], int m,
          int n, int z[][25])
{
     int p, q;
     for (p=0; p<m; p++)
       for (q=0; q<n; q++)
          z[p][q] = x[p][q] + y[p][q];
}
```

Note that the number of columns has to be fixed in the function definition.

- **There is no difference between**
  void add(int x[ ][25], …) and
  void add(int x[15][25], …)
- **Specifying the first dimension is not necessary, but not a mistake.**

Example: Transpose of a matrix

- #include <stdio.h>

- void transpose (int x[][3], int n)
- {
  - int p, q, t;

    - for (p=0; p<n; p++) for (q=0; q<n; q++)
      - {
        - t = x[p][q]; x[p][q] = x[q][p]; x[q][p] = t;
      - }
- }

```
main()
{
  int a[3][3], p, q;

  for (p=0; p<3; p++)
    for (q=0; q<3; q++)
      scanf ("%d", &a[p][q]);

  transpose (a, 3);

  for (p=0; p<3; p++)
  {
    for (q=0; q<3; q++)
      printf ("%d  ", a[p][q]);
    printf ("\n");
  }
}
```

Example: Transpose of a matrix

- #include <stdio.h>

- void transpose (int x[][3], int n)
- {
  - int   p, q, t;
    - for (p=0; p<n; p++)
    - for (q=0; q<n; q++){
      - t = x[p][q];
      - x[p][q] = x[q][p];
      - x[q][p] = t;
    - }
- }

This function is wrong. Why?

```
main()
{
  int a[3][3], p, q;

  for (p=0; p<3; p++)
    for (q=0; q<3;q++)
      scanf ("%d", &a[p][q]);

  transpose (a, 3);

  for (p=0; p<3;p++)
  {
    for (q=0; q<3; q++)
      printf ("%d  ", a[p][q]);
    printf ("\n");
  }
}
```

The Correct Version

```
void transpose (int x[][3], int n)
{
   int  p, q, t;

   for (p = 0; p < n; p++)
     for  (q = p; q < n; q++)
       {
         t = x[p][q];
         x[p][q] = x[q][p];
         x[q][p] = t;
       }
}
```

10	20	30
40	50	60
70	80	90

10	40	70
20	50	80
30	60	90

You may start the inner loop with `q = p + 1`

Another application of Pointers: Dynamic memory allocation

Problem with arrays

Sometimes:

- Amount of data cannot be predicted beforehand (may be driven by user input).
- Number of data items keeps changing during program execution.

Example: Search for an element in an array of N elements

One solution: assume a maximum possible value of N and allocate an array of N elements.

- Wastes memory space, as N may be much smaller in some executions.
- Example: maximum value of N may be 10,000, but a particular run may need to search only among 100 elements.
    - Using array of size 10,000 always wastes memory in most cases.
- On the other extreme, the program cannot handle N larger than 10,000.

Better solution

Dynamic memory allocation

- Know how much memory is needed after the program is run
    - Example: ask the user to enter from keyboard
- Dynamically allocate only the amount of memory needed

C provides functions to dynamically allocate memory

- malloc, calloc, realloc

Dynamic Memory Allocation

Normally the number of elements in an array is pre-specified in the program.

- **Often leads to wastage of memory space or program failure.**

Dynamic Memory Allocation

- **Memory space required can be specified at the time of execution.**
- **C supports allocating and freeing memory dynamically using library routines.**

Memory Allocation Functions

`malloc`

- Allocates requested number of bytes and returns a pointer to the first byte of the allocated space.

`calloc`

- Allocates space for an array of elements, initializes them to zero and then returns a pointer to the first byte of the memory.

`free`

- Frees previously allocated space.

`realloc`

- Modifies the size of previously allocated space.

Allocating a Block of Memory

A block of memory can be allocated using the function malloc.

- **Reserves a block of memory of specified size and returns a pointer of type void *.**
- **The returned pointer can be type-casted to any pointer type.**

General format:

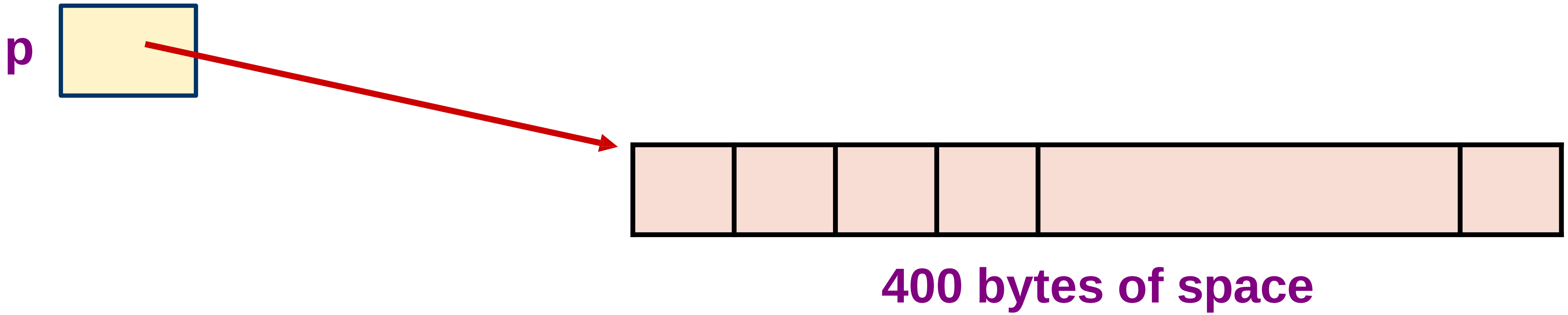
```
ptr =  (type *) malloc (byte_size);
```

Allocating a Block of Memory

Examples

```
p = (int *) malloc(100 * sizeof(int));
```

- A memory space equivalent to (*100 times the size of an int*) bytes is reserved.
- The address of the first byte of the allocated memory is assigned to the pointer `p` of type `int*`.



Allocating a Block of Memory

```
cptr = (char *) malloc (20);
```

- **Allocates 20 bytes of space for the pointer cptr of type char *.**

Points to Note

`malloc` always allocates a block of contiguous bytes.

- The allocation can fail if sufficient contiguous memory space is not available.
- If it fails, `malloc` returns NULL.

```
if  ((p = (int *) malloc(100 * sizeof(int))) == NULL)
     { printf ("Memory cannot be allocated\n");
     exit(1);
}
```

You can use `exit(status)` instead of `return status`. For using `exit()`, you need to `#include <stdlib.h>`.

Example of dynamic memory allocation

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
   int *A, n, i;
   printf(“How many integers will you enter? ”); scanf(“%d”, &n);
   if (n <= 0) { printf(“Wow! How come?\n”); exit(1); }
   A = (int *)malloc(n * sizeof(int));
   if (A == NULL) { printf(“Oops! I cannot store so many integers.\n”); exit(2); }
   for (i=0; i<n; ++i) {
      printf(“Enter integer no. %d: “, i); scanf(“%d”, A+i);
   }
   /* Now, do what you want to do with the integers read and stored in A[] */
   ...
   exit(0);
}
```

Can we allocate only arrays?

`malloc` can be used to allocate memory for single variables also:

```
p = (int *) malloc (sizeof(int));
```

- Allocates space for a single int, which can be accessed as *p or p[0]
- Single variable allocations are just special case of array allocations
    - Array with only one element

Single variable allocations are useful for building linked structures as we will see later.

Using the malloc’d Array

Once the memory is allocated, it can be used with pointers, or with array notation.

Example:

```
int *p, n, i;
scanf(“%d”, &n);
p = (int *) malloc (n * sizeof(int));
for (i=0; i<n; ++i)
   scanf(“%d”, &p[i]);
```

The n integers allocated can be accessed as `*p, *(p+1), *(p+2), ..., *(p+n-1)`

or just as `p[0], p[1], p[2], ..., p[n-1]`

Releasing the allocated space: free

An allocated block can be returned to the system for future use, by the free function.

General syntax:

```
free (ptr);
```

where `ptr` is a pointer to a memory block which has been previously created using `malloc` (or `calloc` or `realloc`).

No size is to be mentioned for the allocated block. The system remembers it. The function frees the entire block allocated by an earlier malloc() type of call.

ptr must be the starting address of an allocated block. A pointer to the interior of a block cannot be passed to free().

Dynamically allocated memory stays until explicitly freed or the program terminates.

You cannot free an array `A[]` defined like this: `int A[50];`

Example of free

```c
int main()
{
  int i,N;
  float *height;
  float sum=0,avg;

  printf("Input no. of students\n");
  scanf("%d", &N);

  height = (float *)
       malloc(N * sizeof(float));
```

```c
  printf("Input heights for %d students\n",N);
  for (i=0; i<N; i++)
    scanf ("%f", &height[i]);

  for(i=0;i<N;i++)
    sum += height[i];

  avg = sum / (float) N;

  printf("Average height = %f\n",avg);
  free (height);
  return 0;
}
```

Altering the Size of a Block

Sometimes we need to alter the size of some previously allocated memory block.

- **More memory needed.**
- **Memory allocated is larger than necessary.**

How?

- **By using the `realloc` function.**

If the original allocation is done as:

```
ptr = malloc (size);
```

then reallocation of space may be done as:

```
ptr = realloc (ptr, newsize);
```

Altering the Size of a Block (contd.)

- **The new memory block may or may not begin at the same place as the old one.**
    - **If it does not find space, it will create it in an entirely different region and move the contents of the old block into the new block.**
- **The function guarantees that the old data remains intact.**
- **If it is unable to allocate, it returns `NULL` and frees the original block.**

Example of realloc

```
int main ()
{
   int *A = (int *)malloc(10 * sizeof(int)), allocsize = 10, n = 0, x;

      printf(“Keep on entering +ve integers. Enter 0 or a -ve integer to 
stop.\n”);
      while (1) {
            printf(“Next integer: ”); scanf(“%d”, &x);
            if (x <= 0) break;
            ++n;
      if (n > allocsize) {
         allocsize += 10;
         A = (int *) realloc(A, allocsize * sizeof(int));
      }
            A[n-1] = x;
      }
      A = (int *) realloc(A, n * sizeof(int)); allocsize = n;
      // Process the integers read from the user
      ...
      free(A);
      return 0;
}
```

Dynamically allocating 2-d arrays A brief discussion

You may recall …

We have discussed the issue of dynamically allocating space for 1-D arrays

- Using `malloc()` library function.

```
int *ptr;

ptr = (int*) malloc( 100 * sizeof(int) );
```

How to dynamically allocate a 2-d array?

Many variations possible:

1. **Fixed number of rows, but variable number of columns**
2. **Variable number of rows, but fixed number of columns**
3. **Both number of rows and columns variable**

We will discuss only the first variation:

Fixed number of rows, but variable number of columns

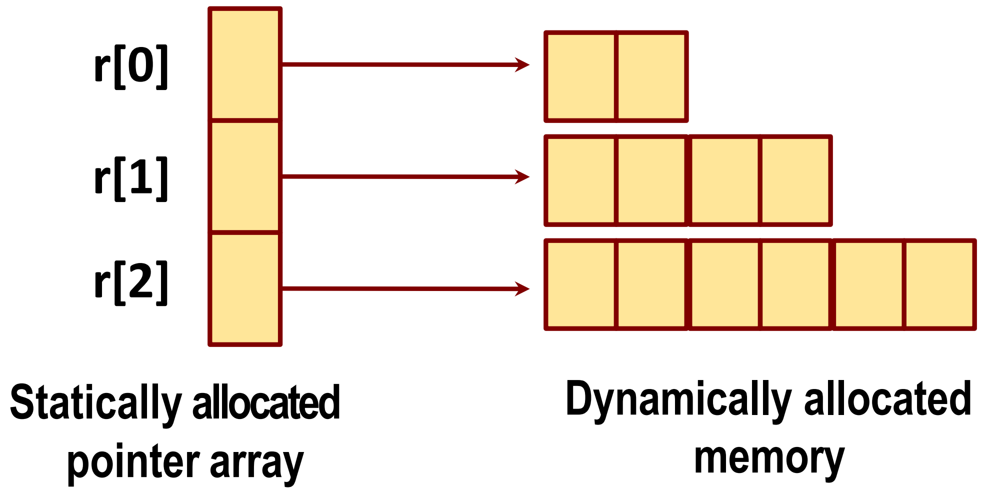
Fixed number of rows, but variable number of columns

Let us assume the number of rows is fixed to 3.

We can use an array of pointers of size 3, where the i[th] element of this array (a pointer) will point to the i[th] row of the 2-d array.

```
int  *r[3], i, c;
printf ("Enter nos. of columns of the 2-d array:");
scanf("%d", &c);                          // each row will have c elements
for (i=0;i<3;i++)
  r[i] = (int *) malloc(c * sizeof(int));     // allocate i-th row
```

Possible to have rows with different number of elements



```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int *r[3], i, j, col;
  for (i=0; i<3; ++i) {
    col = 2 * (i+1);
    r[i] = (int *) malloc (col*sizeof(int));
    for (j=0; j<col; ++j)
      r[i][j] = i + j;
  }
 for (i=0; i<3; ++i) {
    col = 2 * (i+1);
    for (j=0; j<col; ++j)
       printf("%d ", r[i][j]);
    printf("\n");
  }
  return 0;
}
```

Output

```
0 1
1 2 3 4
2 3 4 5 6 7
```

We have studied only 2-d arrays.
C allows arrays of higher dimensions as well.

Practice problems

1. Write a function that takes an n x n square matrix A as parameter ($n < 100$) and returns 1 if A is an upper-triangular matrix, 0 otherwise.
2. Repeat 1 to check for lower-triangular matrix, diagonal matrix, identity matrix.
3. Consider a n x n matrix containing only 0 or 1. Write a function that takes such a matrix and returns 1 if the number of 1's in each row are the same and the number of 1's in each column are the same; it returns 0 otherwise.
4. Write a function that reads in an m x n matrix A and an n x p matrix B, and returns the product of A and B in another matrix C. Pass appropriate parameters.
5. Write a function to find the transpose of a non-square matrix A in a matrix B.
6. Repeat the last exercise when the transpose of A is computed in A itself. Use no additional 2- d arrays.

For each of the above, also write a main function that reads the matrices, calls the function, and prints the results (a message, the result matrix etc.)

ADVANCED TOPICS

Pointers equivalent to two-dimensional arrays

Generalization from one-dimensional arrays

Consider the statically allocated 1-d array:

```
int A[20];
```

A pointer that can browse through A is declared as:

```
int *p;
```

Such a pointer can be allocated dynamic memory and freed as:

```
p = (int *)malloc(20 * sizeof(int));
free(p);
```

- **What are the analogous pointers for 2-d arrays that you have seen earlier?**
- **How can these pointers be allocated and deallocated their own memory?**

What are our 2-d arrays?

We have seen two types of 2-d arrays:

```
int A[10][20];
int *B[10];
```

Both these arrays are statically allocated.

- **A is an array of arrays, and has no dynamic component.**
- **B is an array of pointers. Individual pointers in B[] can be dynamically allocated.**

As statically allocated arrays, both A and B suffer from the two standard disadvantages:

- **Waste of space**
- **Inadequacy to handle larger than the allocated space**

Dynamic versions of A and B overcome these shortcomings.

Dynamic version of A

```
int A[10][20];
```

A pointer matching A should be a pointer to an array of 20 int variables.

But

```
int *p[20];
```

declares an array of 20 int pointers, not a pointer to an array.

Three ways of defining the correct pointer equivalent to A:

Method 1:
```
int (*p)[20];
```

Method 2:
```
typedef int row[20];
row *p;
```

Method 3:
```
typeof(int [20]) *p;        // Not available in the original C specification
```

In all the cases, p is a *single* pointer.

Dynamic version of B

```
int B[10];
```

B is an array of 10 int pointers.
The equivalent pointer is a pointer to an int pointer.

```
int **q;
```

A 2-d array declared by q is fully dynamic.

- The number of rows can be decided during the run of the program.
- The size of each row can also be decided *individually* during the run.

Note: It is *illegal* to set `q = A;` or `p = B;` Expect segmentation fault if you do so (ignoring the warnings issued by the compiler).

Dynamic memory for p

p is a single pointer, and can be allocated and deallocated memory in a single shot.

- **Method 1:**
  ```
  p = (int (*)[20])malloc(10 * 20 * sizeof(int));
  ```
- **Method 2:**
  ```
  p = (row *)malloc(10 * sizeof(row));
  ```
- **Method 3:**
  ```
  p = (typeof(int [20]) *)malloc(10 * 20 * sizeof(typeof(int [20])));
  ```

Freeing requires only one call.

```
free(p);
```

Four types of 2-d arrays

Declaration	Number of rows	Number of columns
`int A[10][20];`	Static	Static
`int (*p)[20];`	Dynamic	Static
`int *B[10];`	Static	Dynamic
`int **q;`	Dynamic	Dynamic