Pointers

CS10001: Programming & Data Structures

Prof. Pallab Dasgupta
Professor, Dept. of Computer Sc. & Engg.,
Indian Institute of Technology, Kharagpur

Introduction

- A pointer is a variable that represents the location (rather than the value) of a data item.
- They have a number of useful applications.
  - Enables us to access a variable that is defined outside the function.
  - Can be used to pass information back and forth between a function and its reference point.
  - More efficient in handling data tables.
  - Reduces the length and complexity of a program.
  - Sometimes also increases the execution speed.

Basic Concept

- In memory, every stored data item occupies one or more contiguous memory cells.
  - 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 memory location(s) to hold the value of the variable.
  - Since every byte in memory has a unique address, this location will also have its own (unique) address.

Contd.

- 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 address location chosen is `1380`.

xyz	➔	variable
50	➔	value
1380	➔	address

Contd.

- 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 the address `1380`.

- 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.

Contd.

- Suppose we assign the address of `xyz` to a variable `p`.
  - `p` is said to point to the variable `xyz`.

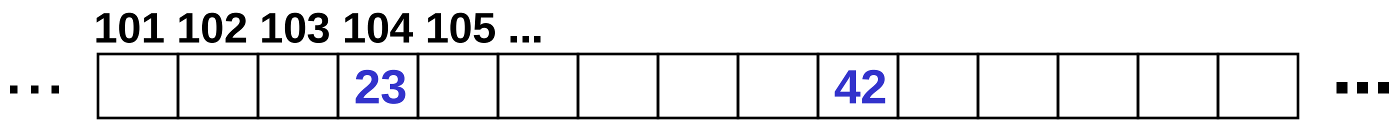
Variable	Value	Address
xyz	50	1380
p	1380	2545

```
p = &xyz;
```

Address vs. Value

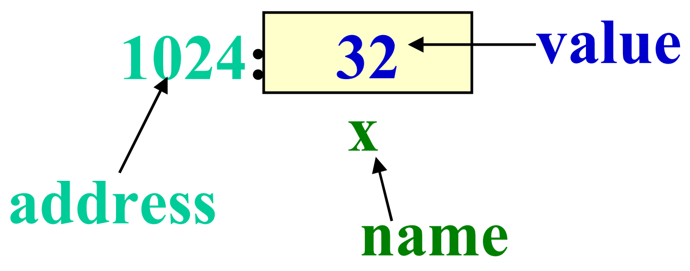
- Each memory cell has an address associated with it.
- Each cell also stores some value.

- Don't confuse the address referring to a memory location with the value stored in that location.



Values vs Locations

- Variables name memory locations, which hold values.



New Type : Pointer

Pointers

- A pointer is just a C variable whose value is the address of another variable!

- After declaring a pointer:

```
int *ptr;
```

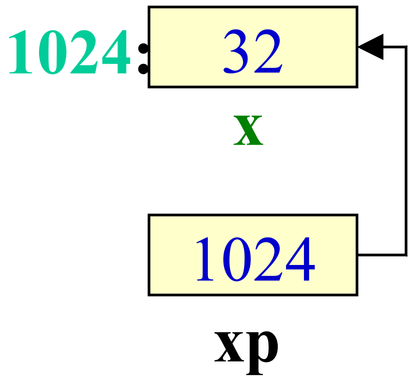
`ptr` doesn't actually point to anything yet. We can either:

  – make it point to something that already exists, or
  – allocate room in memory for something new that it will point to… (next time)

Pointer

```
int x;
int * xp ;

xp = &x ;
```



```
*xp = 0;        /* Assign 0 to x */
*xp = *xp + 1; /* Add 1 to x */
```

Pointers Abstractly

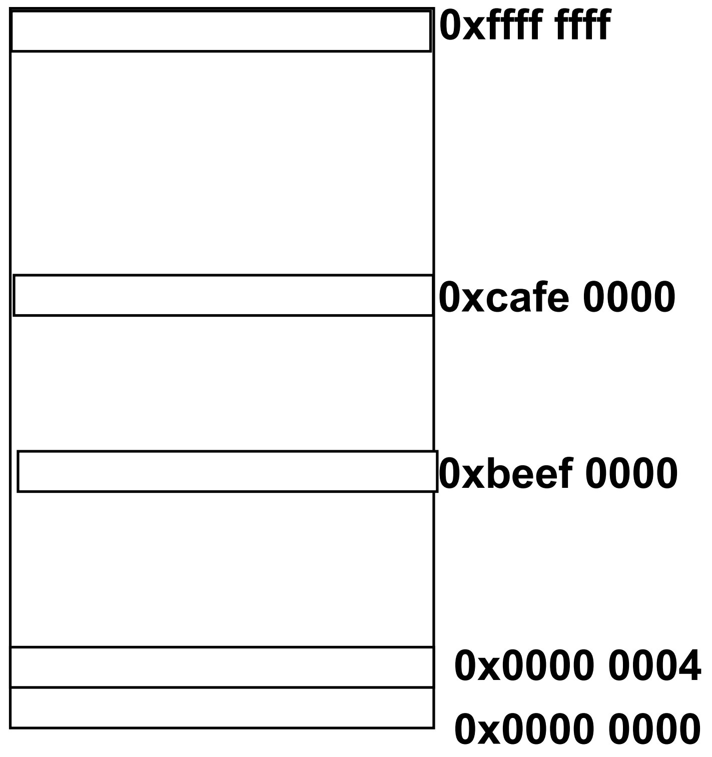
```
int x;
int * p;
p=&x;
...
(x == *p)    True
(p == &x)    True
```

Pointers

- Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!

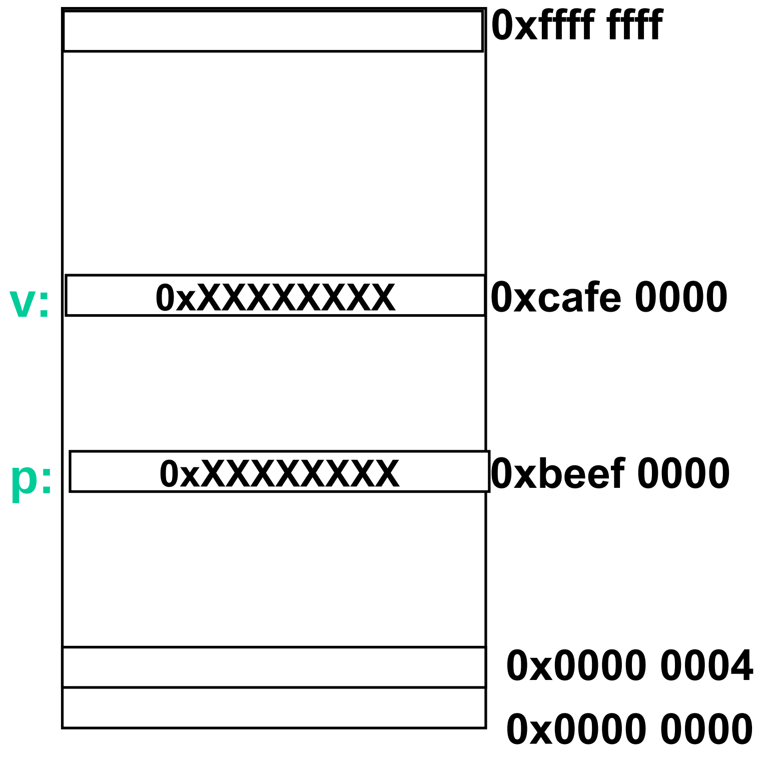
- Local variables in C are not initialized, they may contain anything.

Pointer Usage Example



Memory and Pointers:

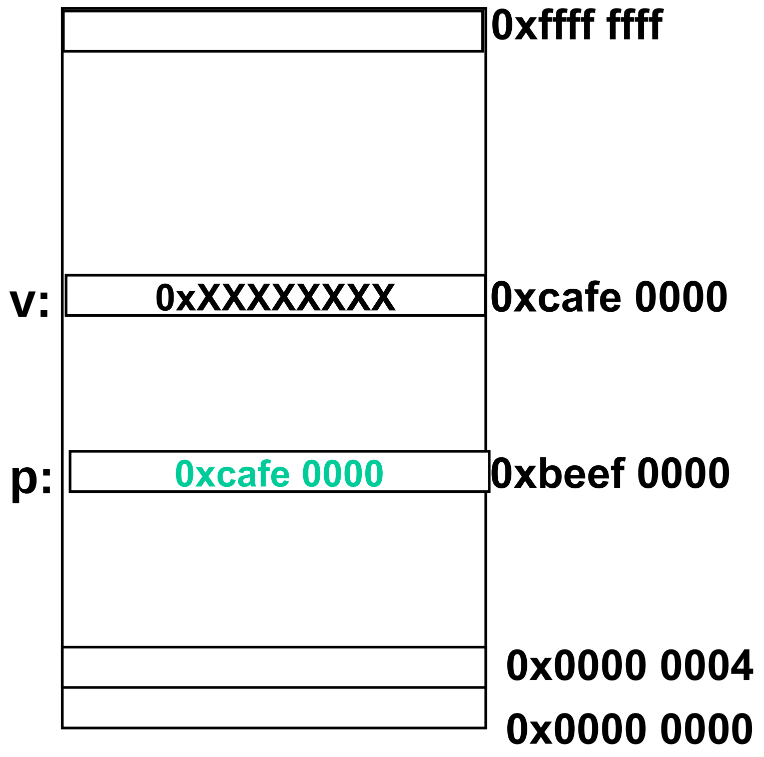
Pointer Usage Example



Memory and Pointers:

```
int *p, v;
```

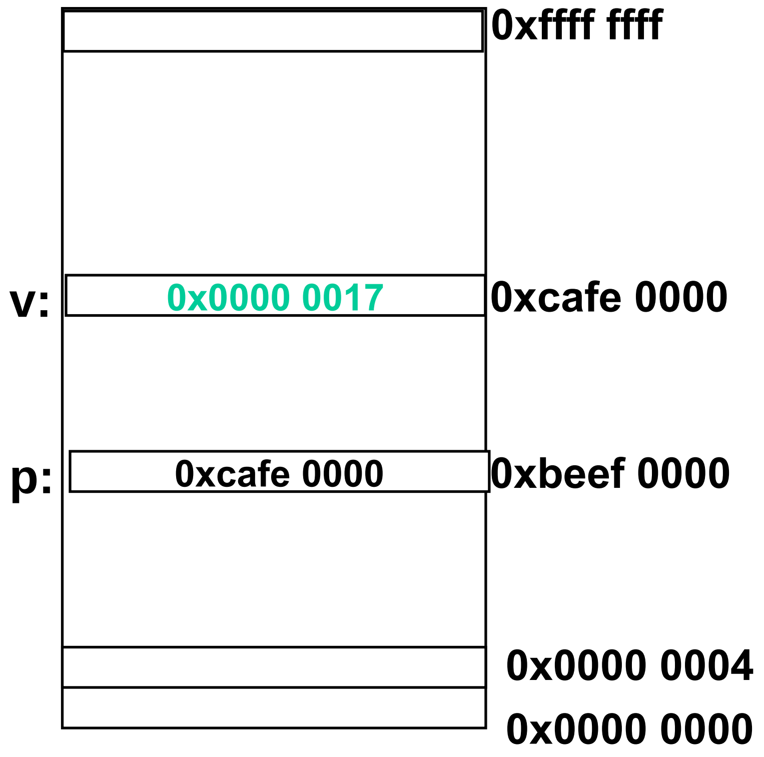
Pointer Usage Example



Memory and Pointers:

```
int *p, v;
p = &v;
```

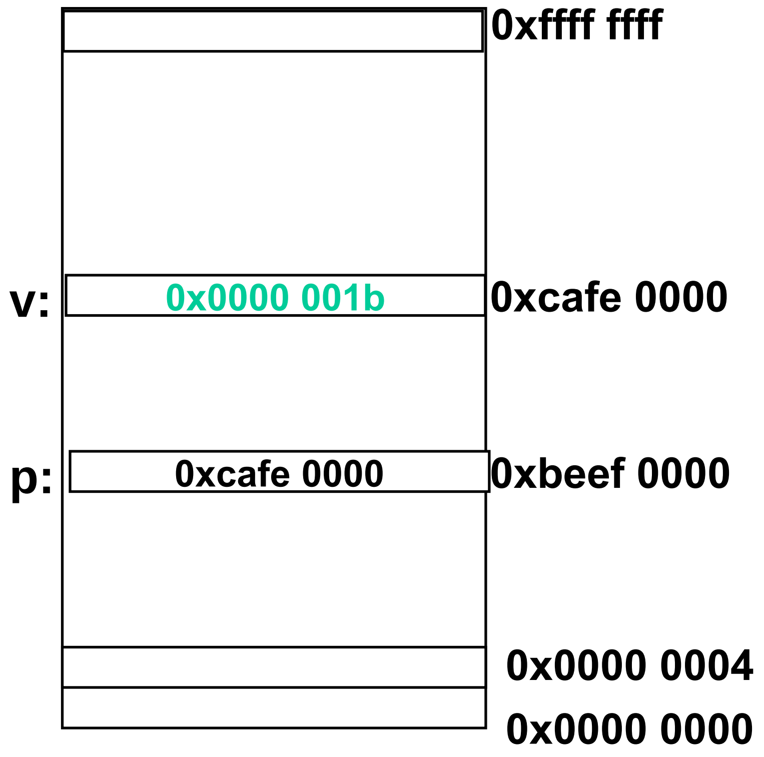
Pointer Usage Example



Memory and Pointers:

```
int *p, v;
p = &v;
v = 0x17;
```

Pointer Usage Example



Memory and Pointers:

```
int *p, v;
p = &v;
v = 0x17;
*p = *p + 4;
V = *p + 4
```

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.

- Example:

```
p = &xyz;
```

  - The *address* of xyz (1380) is assigned to p.

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

```
&distance
&x[0]
&x[i-2]
```

Contd.

- Following usages are illegal:

  &235

  - Pointing at constant.

  ```
  int   arr[20];
    :
  &arr;
  ```

  - Pointing at array name.

  &(a+b)

  - Pointing at expression.

Example

```
#include  <stdio.h>
main()
{
    int   a;
    float  b, c;
    double  d;
    char  ch;

    a = 10;   b = 2.5;  c = 12.36;  d = 12345.66;  ch = ‘A’;
    printf  (“%d is stored in location %u \n”,  a,  &a) ;
    printf  (“%f is stored in location %u \n”,  b,  &b) ;
    printf  (“%f is stored in location %u \n”,  c,  &c) ;
    printf  (“%ld is stored in location %u \n”, d,  &d) ;
    printf  (“%c is stored in location %u \n”,  ch, &ch) ;
}
```

Output:

```
10 is stored in location 3221224908
2.500000 is stored in location 3221224904
12.360000 is stored in location 3221224900
12345.660000 is stored in location 3221224892
A is stored in location 3221224891
```

Pointer Declarations

- Pointer variables must be declared before we use them.
- General form:

```
data_type *pointer_name;
```

- Three things are specified in the above declaration:
  - The asterisk (*) tells that the variable `pointer_name` is a pointer variable.
  - `pointer_name` needs a memory location.
  - pointer_name points to a variable of type `data_type`.

Contd.

- Example:

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

- Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement like:

```
int *p, xyz;
:
p = &xyz;
```

  – This is called *pointer initialization*.

Things to Remember

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

```
float   x;
int     *p;
:
p = &x;
```

➔ will result in erroneous output

- Assigning an absolute address to a pointer variable is prohibited.

```
int    *count;
:
count = 1268;
```

Accessing a Variable Through its Pointer

- Once a pointer has been assigned the *address* of a variable, the *value* of the variable can be accessed using the *indirection operator* (*).

```
int    a, b;
int    *p;
:
p = &a;
b = *p;
```

Equivalent to

```
b = a;
```

Example 1

```
#include  <stdio.h>
main()
{
    int   a, b;
    int   c = 5;
    int   *p;

    a  =  4  *  (c  +  5) ;

    p  =  &c;
    b  =  4  *  (*p  +  5) ;
    printf  ("a=%d  b=%d \n",  a, b);
}
```

Equivalent

```
a=40 b=40
```

Example 2

```
#include  <stdio.h>
main()
{
    int  x, y;
    int  *ptr;

    x = 10 ;
    ptr = &x ;
    y = *ptr ;
    printf ("%d is stored in location %u \n",  x,  &x) ;
    printf ("%d is stored in location %u \n",  *&x,  &x) ;
    printf ("%d is stored in location %u \n",  *ptr,  ptr) ;
    printf ("%d is stored in location %u \n",  y,  &*ptr) ;
    printf ("%u is stored in location %u \n",  ptr, &ptr) ;
    printf ("%d is stored in location %u \n",  y,  &y) ;

    *ptr = 25;
    printf ("\nNow x = %d \n", x);
}
```

```
Address of x:      3221224908
Address of y:      3221224904
Address of ptr:    3221224900
```

Output:

```
10 is stored in location 3221224908
10 is stored in location 3221224908
10 is stored in location 3221224908
10 is stored in location 3221224908
3221224908 is stored in location 3221224900
10 is stored in location 3221224904

Now x = 25
```

Pointer Expressions

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

- If p1 and p2 are two pointers, the following statements are valid:

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

`*p1` can appear on the left hand side

Contd.

- What are allowed in C?
  - Add an integer to a pointer.
  - Subtract an integer from a pointer.
  - Subtract one pointer from another (related).
    - If p1 and p2 are both pointers to the same array, then p2–p1 gives the number of elements between p1 and p2.

Contd.

- **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  *p1, *p2;
int  i, j;
:
p1 = p1 + 1;
p2 = p1 + j;
p2++;
p2 = p2 - (i + j);
```

  - **In reality, it is not the integer value which is added/subtracted, but rather the *scale factor* times *the value*.**

Contd.

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

– If p1 is an integer pointer, then

p1++

will increment the value of p1 by 4.

- **Note:**
  - The exact scale factor may vary from one machine to another.
  - Can be found out using the `sizeof` function.
  - Syntax:

```
sizeof (data_type)
```

Example: to find the scale factors

```
#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
```

Passing Pointers to a Function

- Pointers are often passed to a function as arguments.
  - Allows data items within the calling program to be accessed by the function, altered, and then returned to the calling program in altered form.
  - Called *call-by-reference* (or by *address* or by *location*).
- Normally, arguments are passed to a function *by value*.
  - The data items are copied to the function.
  - Changes are not reflected in the calling program.

Example: passing arguments by value

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

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

Output

```
a=5, b=20
```

Example: passing arguments by reference

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

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

Output

```
a=20, b=5
```

Pointers and Arrays

- **When an array is declared,**
  - **The compiler allocates 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};`

  – 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

Contd.

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

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

– 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 value of x[i]

Example: function to find average

```
#include <stdio.h>
main()
{
   int x[100], k, n;

   scanf ("%d", &n);

   for (k=0; k<n; k++)
       scanf ("%d", &x[k]);

   printf  ("\nAverage is %f",
                  avg (x, n));
}
```

```
float avg (array, size)
int array[], size;
{
   int  *p, i , sum = 0;

   p = array;

   for (i=0; i<size; i++)
       sum = sum + *(p+i);

   return ((float) sum / size);
}
```

Arrays and pointers

- An array name is an address, or a pointer value.
- Pointers as well as arrays can be subscripted.
- A pointer variable can take different addresses as values.
- An array name is an address, or pointer, that is fixed.

It is a CONSTANT pointer to the first element.

Arrays

- Consequences:
  - `ar` is a pointer
  - `ar[0]` is the same as `*ar`
  - `ar[2]` is the same as `*(ar+2)`
  - We can use pointer arithmetic to access arrays more conveniently.
- Declared arrays are only allocated while the scope is valid

```
char *foo() {
   char string[32]; ...;
   return string;
```
} is incorrect

Arrays

- Array size n; want to access from `0` to `n-1`, so you should use counter AND utilize a constant for declaration & incr
  - Wrong

    ```
    int i, ar[10];
    for(i = 0; i < 10; i++){ ... }
    ```

  - Right

    ```
    #define ARRAY_SIZE 10
    int i, a[ARRAY_SIZE];
    for(i = 0; i < ARRAY_SIZE; i++){ ... }
    ```

- Why? SINGLE SOURCE OF TRUTH
  - You’re utilizing indirection and avoiding maintaining two copies of the number 10

Arrays

- Pitfall: An array in C does <u>not</u> know its own length, & bounds not checked!
  - Consequence: We can accidentally access off the end of an array.
  - Consequence: We must pass the array <u>and its size</u> to a procedure which is going to traverse it.
- Segmentation faults and bus errors:
  - These are VERY difficult to find;
    be careful!
  - You'll learn how to debug these in lab…

Arrays In Functions

- An array parameter can be declared as an array <u>or</u> a pointer; an array argument can be passed as a pointer.
  - Can be incremented

```
int strlen(char s[])      int strlen(char *s)
{                         {


}                         }
```

Arrays and pointers

int a[20], i, *p;

- The expression a[i] is equivalent to *(a+i)
- p[i] is equivalent to *(p+i)
- When an array is declared the compiler allocates a sufficient amount of contiguous space in memory. The base address of the array is the address of a[0].
- Suppose the system assigns 300 as the base address of a. a[0], a[1], ...,a[19] are allocated 300, 304, ..., 376.

Arrays and pointers

```
#define N 20
int a[2N], i, *p, sum;
```

- p = a; is equivalent to p = *a[0];
- p is assigned 300.
- Pointer arithmetic provides an alternative to array indexing.
- p=a+1; is equivalent to p=&a[1]; (p is assigned 304)

```
for (p=a; p<&a[N]; ++p)
      sum += *p ;
```

```
p=a;
for (i=0; i<N; ++i)
      sum += p[i] ;
```

```
for (i=0; i<N; ++i)
      sum += *(a+i) ;
```

Arrays and pointers

```
int a[N];
```

- a is a constant pointer.

- ~~a=p; ++a; a+=2; illegal~~

Pointer arithmetic and element size

```
double * p, *q ;
```

- The expression **p+1** yields the correct machine address for the next variable of that type.
- Other valid pointer expressions:
  - p+i
  - ++p
  - p+=i
  - p-q /* No of array elements between p and q */

Pointer Arithmetic

- Since a pointer is just a mem address, we can add to it to traverse an array.
- p+1 returns a ptr to the next array element.
- `(*p)+1` vs `*p++` vs `*(p+1)` vs `*(p)++` ?
  - `x = *p++` ⇒ `x = *p ; p = p + 1;`
  - `x = (*p)++` ⇒ `x = *p ; *p = *p + 1;`

- What if we have an array of large structs (objects)?
  - C takes care of it: In reality, p+1 doesn't add 1 to the memory address, it adds the size of the array element.

Pointer Arithmetic

- **We can use pointer arithmetic to “walk” through memory:**

```
void copy(int *from, int *to, int n) {
    int i;
    for (i=0; i<n; i++) {
        *to++ = *from++;
    }
}
```

  - C automatically adjusts the pointer by the right amount each time (i.e., 1 byte for a `char`, 4 bytes for an `int`, etc.)

Pointer Arithmetic

- C knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes.

- So the following are equivalent:

```
int get(int array[], int n)
{
    return  (array[n]);
     /* OR */
    return *(array + n);
}
```

Pointer Arithmetic

- Array size n; want to access from `0` to `n-1`
  - test for exit by comparing to address one element past the array

```
int ar[10], *p, *q, sum = 0;
...
p = ar; q = &(ar[10]);
while (p != q)
        /* sum = sum + *p; p = p + 1; */
        sum += *p++;
```

  - Is this legal?
- C defines that one element past end of array must be a valid address, i.e., not cause an bus error or address error

Example with 2-D array

TO BE DISCUSSED LATER

Structures Revisited

- **Recall that a structure can be declared as:**

```
struct stud {
               int    roll;
               char   dept_code[25];
               float  cgpa;
            };
struct  stud  a, b, c;
```

- **And the individual structure elements can be accessed as:**

```
a.roll ,  b.roll ,  c.cgpa
```

Arrays of Structures

- **We can define an array of structure records as**

```
struct stud class[100];
```

- **The structure elements of the individual records can be accessed as:**

```
class[i].roll
class[20].dept_code
class[k++].cgpa
```

Example :: sort by roll number (bubble sort)

```
#include <stdio.h>
struct stud
{
    int  roll;
    char  dept_code[25];
    float  cgpa;
};

main()
{
  struc  stud  class[100], t;
  int  j, k, n;

  scanf  (“%d”, &n);
        /* no. of students */
```

```
for  (k=0; k<n; k++)
  scanf (“%d %s %f”, &class[k].roll,
               class[k].dept_code,
               &class[k].cgpa);
for (j=0; j<n-1; j++)
  for (k=j+1; k<n; k++)
  {
    if (class[j].roll > class[k].roll)
    {
       t = class[j];
       class[j] = class[k];
       class[k] = t;
     }
    }
    <<<< PRINT THE RECORDS >>>>
}
```

Example :: selection sort

```
int min_loc (struct stud x[],
                      int k, int size)

int j, pos;
{
    pos = k;
    for (j=k+1; j<size; j++)
        if (x[j] < x[pos])
             pos = j;
    return pos;
}
```

```
int selsort (struct stud x[],int n)

{
    int k, m;
    for (k=0; k<n-1; k++)
    {
        m = min_loc(x, k, n);
        temp = a[k];
        a[k] = a[m];
        a[m] = temp;
    }
}
```

```
main()
{
   struc  stud  class[100];
   int n;
   …
   selsort (class, n);
   …
```

Arrays within Structures

- C allows the use of arrays as structure members.
- Example:

```
    struct stud {
                   int   roll;
                   char  dept_code[25];

                   int   marks[6];
                   float cgpa;
                };
    struct  stud  class[100];
```

- To access individual marks of students:

```
    class[35].marks[4]
    class[i].marks[j]
```

Pointers and Structures

- You may recall that the name of an array stands for the address of its *zero-th element*.
  - Also true for the names of arrays of structure variables.
- Consider the declaration:

```
struct stud {
            int   roll;
            char  dept_code[25];
            float cgpa;
          } class[100],  *ptr ;
```

- The name `class` represents the address of the zero-th element of the structure array.
- `ptr` is a pointer to data objects of the type `struct stud`.

- The assignment

```
ptr = class;
```

  will assign the address of `class[0]` to `ptr`.

- When the pointer `ptr` is incremented by one (ptr++) :
  - The value of `ptr` is actually increased by `sizeof(stud)`.
  - It is made to point to the next record.

- Once `ptr` points to a structure variable, the members can be accessed as:

```
ptr -> roll;
ptr -> dept_code;
ptr -> cgpa;
```

  - The symbol "–>" is called the *arrow* operator.

A Warning

- **When using structure pointers, we should take care of operator precedence.**
  - **Member operator "." has higher precedence than "*".**

    ptr –> roll and (*ptr).roll mean the same thing.

    ***ptr.roll will lead to error.**

  - **The operator "–>" enjoys the highest priority among operators.**

    ++ptr –> roll will increment roll, not ptr.

    (++ptr) –> roll will do the intended thing.

Structures and Functions

- A structure can be passed as argument to a function.
- A function can also return a structure.
- The process shall be illustrated with the help of an example.
  - A function to add two complex numbers.

Example: complex number addition

```
#include <stdio.h>
struct complex {
                  float  re;
                  float  im;
                };

main()
{
   struct  complex  a, b, c;
   scanf  (“%f %f”, &a.re, &a.im);
   scanf  (“%f %f”, &b.re, &b.im);
   c  =  add (a, b) ;
   printf  (“\n %f %f”, c,re, c.im);
}
```

```
struct complex add (x, y)
struct complex x, y;
{
   struct complex  t;

   t.re = x.re + y.re ;
   t.im = x.im + y.im ;
   return (t)  ;
}
```

Example: Alternative way using pointers

```
#include <stdio.h>
struct complex  {
                   float  re;
                   float  im;
                };

main()
{
   struct  complex  a, b, c;
   scanf  ("%f %f", &a.re, &a.im);
   scanf  ("%f %f", &b.re, &b.im);
   add (&a, &b, &c) ;
   printf ("\n %f %f", c,re, c.im);
}
```

```
void add (x, y, t)
struct complex  *x, *y, *t;
{
   t->re = x->re + y->re;
   t->im = x->im + y->im;
}
```

Dynamic Memory Allocation

Basic Idea

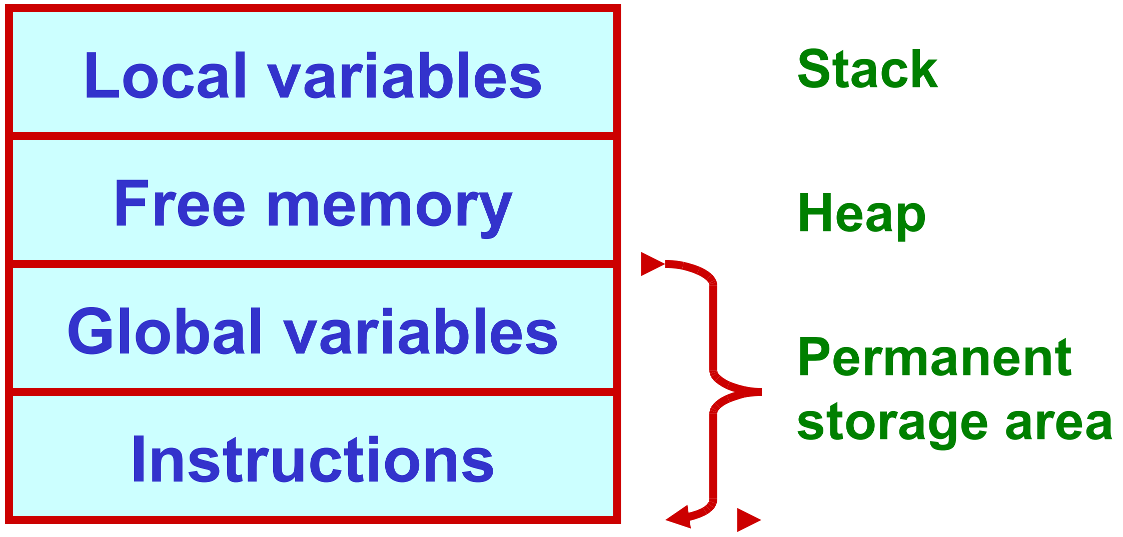
- Many a time we face situations where data is dynamic in nature.
  - Amount of data cannot be predicted beforehand.
  - Number of data items keeps changing during program execution.
- Such situations can be handled more easily and effectively using dynamic memory management techniques.

Contd.

- **C language requires the number of elements in an array to be specified at compile time.**
  - **Often leads to wastage or 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 Process in C



Contd.

- The program instructions and the global variables are stored in a region known as ***permanent storage area***.
- The local variables are stored in another area called ***stack***.
- The memory space between these two areas is available for dynamic allocation during execution of the program.
  - This free region is called the *heap*.
  - The size of the heap keeps changing.

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 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 return pointer can be type-casted to any pointer type.

- General format:

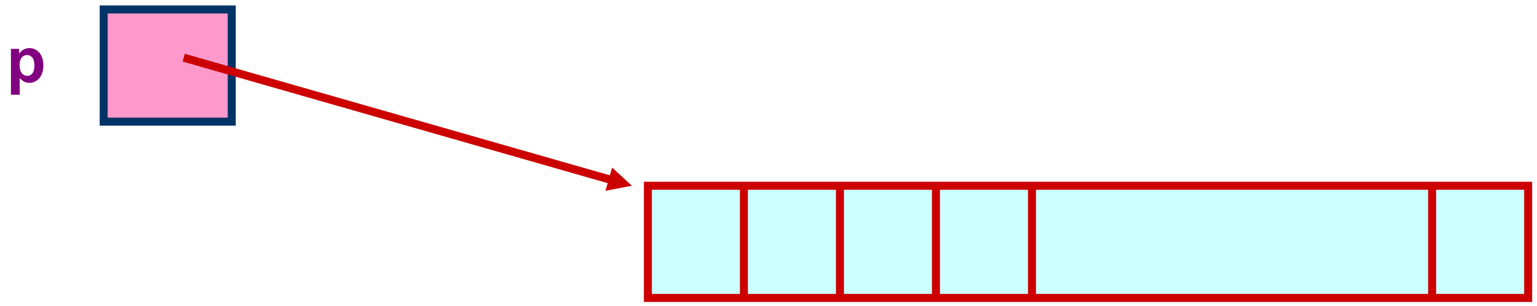
```
ptr =  (type *) malloc (byte_size);
```

Contd.

- **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`.



400 bytes of space

Contd.

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

– Allocates 20 bytes of space for the pointer `cptr` of type `char`.

```
sptr = (struct stud *) malloc
        (10 * sizeof (struct stud));
```

– Allocates space for a structure array of 10 elements. `sptr` points to a structure element of type “`struct stud`”.

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 ("\n Memory cannot be allocated”);
       exit();
  }
```

Example

```
#include <stdio.h>

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));
```

```
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);
}
```

Releasing the Used Space

- When we no longer need the data stored in a block of memory, we may release the block for future use.
- How?
  - By using the `free` function.
- General syntax:

```
free (ptr);
```

where `ptr` is a pointer to a memory block which has been previously created using `malloc`.

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);
```

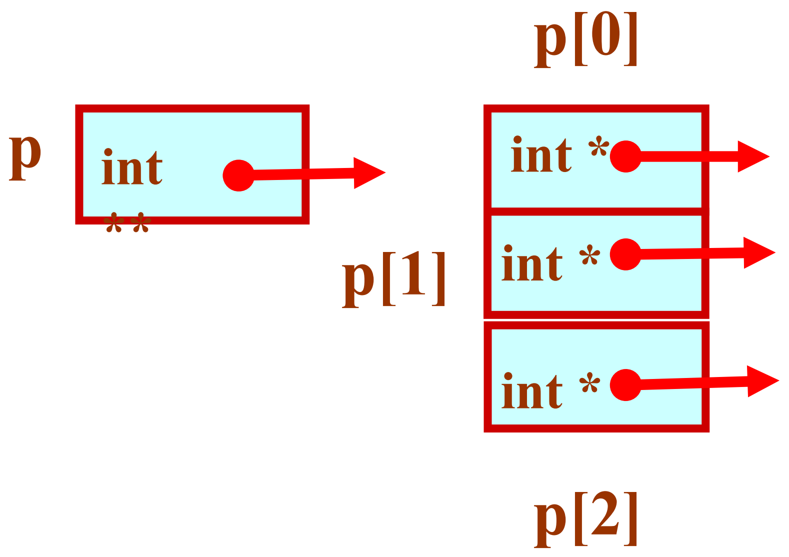
Contd.

- 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.

Pointer to Pointer

- **Example:**

```
int **p;
p = (int **) malloc(3 * sizeof(int *));
```



2-D Array Allocation

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

int **allocate (int h, int w)
  {
    int **p;
    int i, j;

    p = (int **) calloc(h, sizeof (int *) );
    for (i=0;i<h;i++)
      p[i] = (int *) calloc(w,sizeof (int));
    return(p);
  }
```

Allocate array of pointers

Allocate array of integers for each row

```
void read_data (int **p, int h, int w)
  {
     int i, j;
     for (i=0;i<h;i++)
       for (j=0;j<w;j++)
         scanf ("%d", &p[i][j]);
  }
```

Elements accessed like 2-D array elements.

2-D Array: Contd.

```
void print_data (int **p, int h, int w)
  {
     int i, j;
      for (i=0;i<h;i++)
      {
      for (j=0;j<w;j++)
        printf ("%5d ", p[i][j]);
       printf ("\n");
      }
}
```

```
main()
{
  int **p;
  int M, N;

  printf ("Give M and N \n");
  scanf ("%d%d", &M, &N);
  p = allocate (M, N);
  read_data (p, M, N);
  printf ("\nThe array read as \n");
  print_data (p, M, N);
}
```

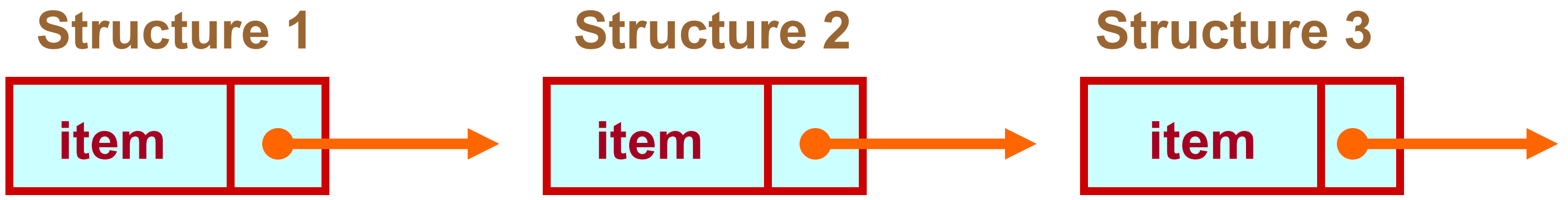
```
Give M and N
3 3
1 2 3
4 5 6
7 8 9
 The array read as
    1    2    3
    4    5    6
    7    8    9
```

Linked List :: Basic Concepts

- A list refers to a set of items organized sequentially.
  - An array is an example of a list.
    - The array index is used for accessing and manipulation of array elements.
  - Problems with array:
    - The array size has to be specified at the beginning.
    - Deleting an element or inserting an element may require shifting of elements.

Contd.

- A completely different way to represent a list:
  - Make each item in the list part of a structure.
  - The structure also contains a pointer or link to the structure containing the next item.
  - This type of list is called a linked list.



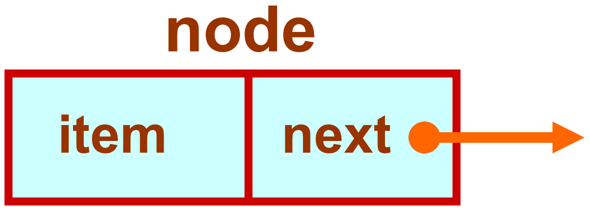
Contd.

- Each structure of the list is called a *node*, and consists of two fields:
  - One containing the item.
  - The other containing the address of the next item in the list.
- The data items comprising a linked list need not be contiguous in memory.
  - They are ordered by logical links that are stored as part of the data in the structure itself.
  - The link is a pointer to another structure of the same type.

Contd.

- Such a structure can be represented as:

```
struct node
    {
        int item;
        struct node  *next;
    }
```



- Such structures which contain a member field pointing to the same structure type are called ***self-referential structures***.

Contd.

- In general, a node may be represented as follows:

```
struct node_name
    {
        type  member1;
        type member2;
         ………
        struct node_name *next;
    }
```

Illustration

- Consider the structure:

```
struct stud
    {
        int  roll;
        char name[30];
        int  age;
        struct stud *next;
    }
```

- Also assume that the list consists of three nodes n1, n2 and n3.

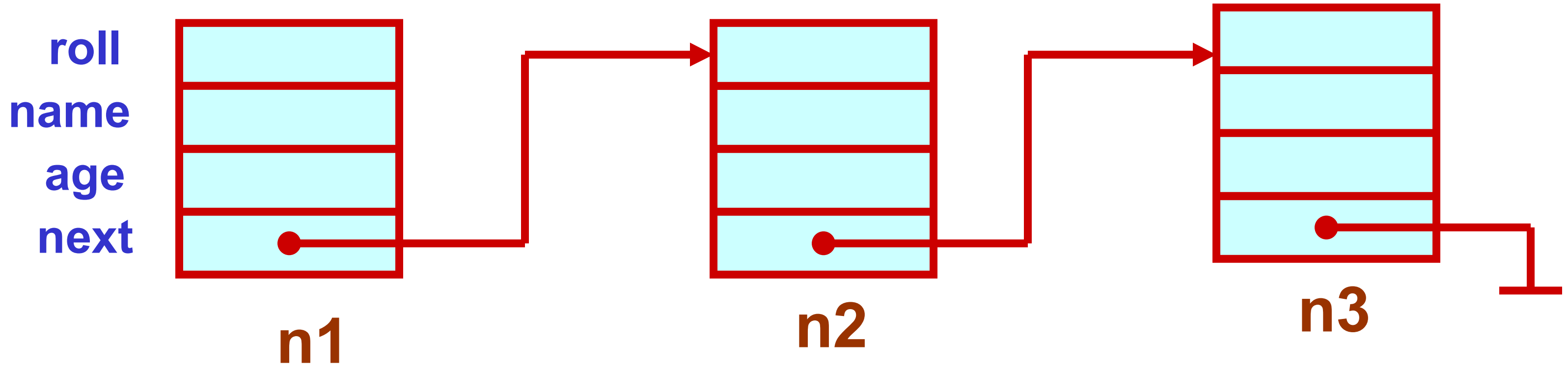
```
struct stud n1, n2, n3;
```

Contd.

- To create the links between nodes, we can write:

```
n1.next =  &n2 ;
n2.next =  &n3 ;
n3.next =  NULL ;  /* No more nodes follow */
```

- Now the list looks like:



Example

```
#include  <stdio.h>
struct  stud
  {
        int  roll;
        char  name[30];
        int  age;
        struct  stud  *next;
  }

main()
{
    struct  stud  n1, n2, n3;
    struct  stud  *p;

    scanf (“%d %s %d”, &n1.roll, n1.name, &n1.age);
    scanf (“%d %s %d”, &n2.roll, n2.name, &n2.age);
    scanf (“%d %s %d”, &n3.roll, n3.name, &n3.age);
    n1.next  =  &n2 ;
    n2.next  =  &n3 ;
    n3.next  =  NULL ;

   /* Now traverse the list and print the elements */

    p  =  n1 ;   /* point to 1st element */
    while  (p != NULL)
    {
        printf ("\n %d %s %d",
        p->roll, p->name, p->age);
        p  =  p->next;
    }
}
```

Alternative Way

- **Dynamically allocate space for the nodes.**
  - **Use malloc or calloc individually for every node allocated.**