Structures

What is a Structure?

- Used for handling a group of logically related data items
  - Examples:
    - Student name, roll number, and marks
    - Real part and complex part of a complex number
- Helps in organizing complex data in a more meaningful way
- The individual structure elements are called members

Defining a Structure

```
struct tag {
        member 1;
        member 2;
        :
        member m;
};
```

- struct is the required C keyword
- tag is the name of the structure
- member 1, member 2, … are individual member declarations
- Do not forget the ; at the end!

Contd.

- The individual members can be ordinary variables, pointers, arrays, or other structures (any data type)
  - The member names within a particular structure must be distinct from one another
  - A member name can be the same as the name of a variable defined outside of the structure
- Once a structure has been defined, the individual structure-type variables can be declared as:

  struct tag var_1, var_2, …, var_n;

Example

- A structure definition

```
struct student {
        char name[30];
        int  roll_number;
        int  total_marks;
        char dob[10];
      };
```

- Defining structure variables:

```
struct student  a1, a2, a3;
```

A new data-type

A Compact Form

- It is possible to combine the declaration of the structure with that of the structure variables:

```
struct tag {
        member 1;
        member 2;
        :
        member m;
    }  var_1, var_2,…, var_n;
```

- Declares three variables of type struct tag
- In this form, tag is optional

Accessing a Structure

- The members of a structure are processed individually, as separate entities
  - Each member is a separate variable
- A structure member can be accessed by writing

  variable.member

  where variable refers to the name of a structure-type variable, and member refers to the name of a member within the structure
- Examples:

  a1.name, a2.name, a1.roll_number, a3.dob

Example: Complex number addition

```
struct  complex
        {
            float  real;
            float  img;
        };
int main()
{
     struct complex a, b, c;
     scanf (“%f %f”, &a.real, &a.img);
     scanf (“%f %f”, &b.real, &b.img);
     c.real = a.real + b.real;
     c.img = a.img + b.img;
     printf (“\n %f + %f j”, c.real, c.img);
     return 0;
}
```

Defines the structure

Declares 3 variable of type struct complex

Accessing the variables is the same as any other variable, just have to follow the syntax to specify which field of the Structure you want

Operations on Structure Variables

- Unlike arrays, a structure variable can be directly assigned to another structure variable of the same type

  a1 = a2;

  - All the individual members get assigned
- Two structure variables can not be compared for equality or inequality

  if (a1 == a2)…… ← this cannot be done

Arrays of Structures

- Once a structure has been defined, we can declare an array of structures

  struct student class[50];

  type name

  - The individual members can be accessed as:

    class[i].name

    class[5].roll_number

Example: Reading and Printing Array of Structures

```
int main()
{
        struct complex A[100];
        int n;
        scanf("%d", &n);
        for (i=0; i<n; i++)
                scanf("%f%f", &A[i].real, &A[i].img);
        for (i=0; i<n; i++)
                printf("%f + i%f\n", A[i].real, A[i].img);
}
```

Arrays within Structures

- A structure member can be an array

```
struct  student
{
    char  name[30];
    int  roll_number;
    int  marks[5];
    char  dob[10];
}  a1, a2, a3;
```

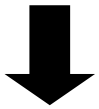
- The array element within the structure can be accessed as:

  a1.marks[2], a1.dob[3],…

Structure Initialization

- Structure variables may be initialized following similar rules of an array. The values are provided within the second braces separated by commas
- An example:

```
struct complex a={1.0,2.0}, b={-3.0,4.0};
```



```
a.real=1.0;   a.img=2.0;
b.real=-3.0;  b.img=4.0;
```

Parameter Passing in a Function

- Structure variables can be passed as parameters like any other variables. Only the values will be copied during function invocation

```
int chkEqual(struct complex a, struct complex b)
{
        if ((a.real==b.real) && (a.img==b.img))
            return 1;
        else return 0;
}
```

Parameter Passing in a Function

- Array of structures can be passed as parameters the same way as normal arrays
- Values are changed in the array as before

```
void (struct complex a[ ], struct complex b[ ], int n)
{
        int i;
        for (i=0; i<n, i++) {
                b[i].real += a[i].real;
                b[i].img += a[i].img;
        }
}
```

Returning structures

- It is also possible to return structure values from a function. The return data type of the function should be as same as the data type of the structure itself

```
struct complex add(struct complex a, struct complex b)
{
      struct complex tmp;

      tmp.real = a.real + b.real;
      tmp.img = a.img + b.img;
      return(tmp);
}
```

Direct arithmetic operations are not possible with structure variables

Defining data type: using typedef

- One may define a structure data-type with a single name

```
typedef struct newtype {
        member-variable1;
        member-variable2;
              .
        member-variableN;
} mytype;
```

- mytype is the name of the new data-type
  - Also called an alias for struct newtype
  - Writing the tag name newtype is optional, can be skipped
  - Naming follows rules of variable naming

typedef : An example

```
typedef struct {
        float real;
        float imag;
} _COMPLEX;
```

- Defined a new data type named _COMPLEX. Now can declare and use variables of this type

```
_COMPLEX a, b, c;
```

- Note: typedef is not restricted to just structures, can define new types from any existing type
- Example:
  - typedef int INTEGER
  - Defines a new type named INTEGER from the known type int
  - Can now define variables of type INTEGER which will have all properties of the int type

```
INTEGER a, b, c;
```

The earlier program using typedef

```
typedef struct{
                float real;
                float img;
        } _COMPLEX;

_COMPLEX add(_COMPLEX a, _COMPLEX b)
 {
        _COMPLEX tmp;

        tmp.real = a.real + b.real;
        tmp.img = a.img + b.img;
        return(tmp);
 }
```

Contd.

```
void print (_COMPLEX a)
{
     printf("(%f, %f) \n",a.real,a.img);
}

int main()
{
    _COMPLEX x={4.0,5.0}, y={10.0,15.0}, z;

     print(x);
     print(y);
      z = add(x,y);
      print(z);
     return 0;
}
```

Output

```
(4.000000, 5.000000)
(10.000000, 15.000000)
(14.000000, 20.000000)
```

Pointers: Basics

What is a pointer?

- First of all, it is a variable, just like other variables you studied
  - So it has type, storage etc.
- Difference: it can only store the address (rather than the value) of a data item
- Type of a pointer variable – pointer to the type of the data whose address it will store
  - Example: int pointer, float pointer,…
  - Can be pointer to any user-defined types also like structure types

- 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

- As seen before, 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.

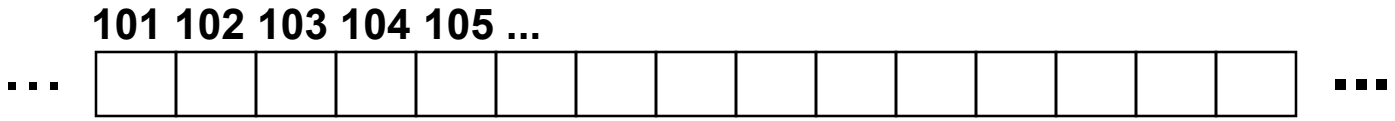
- Suppose we assign the address of xyz to a variable p
  - p is said to point to the variable xyz

Variable	Value	Address
xyz	50	1380
p	1380	2545

p = &xyz;

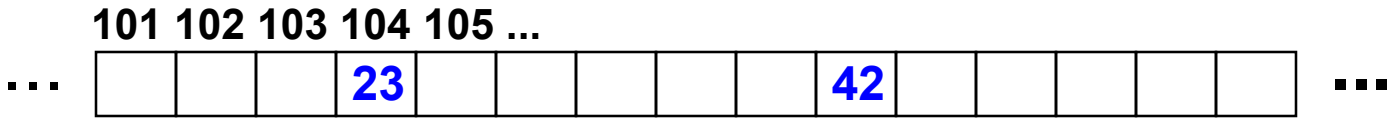
Address vs. Value

- Each memory cell has an address associated with it



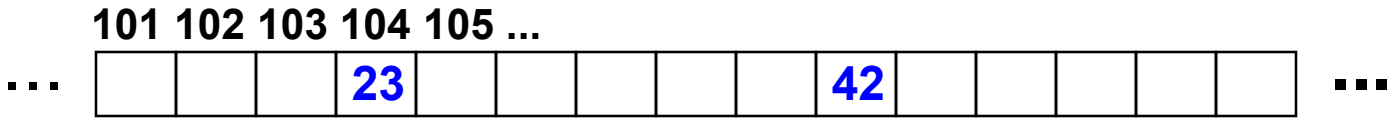
Address vs. Value

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



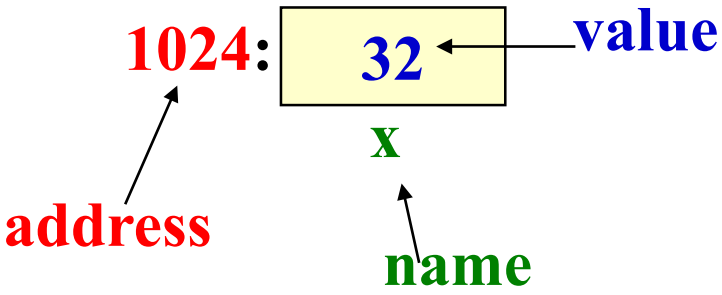
Address vs. Value

- 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



Pointers in C

- A pointer is just a C variable whose value can contain the address of another variable
- Needs to be declared before use just like any other variable
- 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

Example

```
int      *count;
float   *speed;
char *c;
```

- 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;
```

- Pointers can be defined for any type, including user defined types
- Example

```
struct name {
    char first[20];
    char last[20];
};
struct name *p;
```

  - p is a pointer which can store the address of a struct name type variable

Accessing the Address of a Variable

- The address of a variable is given by 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 (of any type, including user-defined types) or an array element

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

Illegal Use of &

- &235
  - Pointing at constant
- int arr[20];
  :
  &arr;
  - Pointing at array name
- &(a+b)
  - Pointing at expression

In all these cases, there is no storage,
so no address either

Example

```
#include  <stdio.h>
int 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  ("%lf is stored in location %u \n", d,  &d) ;
    printf  ("%c is stored in location %u \n",  ch, &ch) ;
    return 0;
}
```

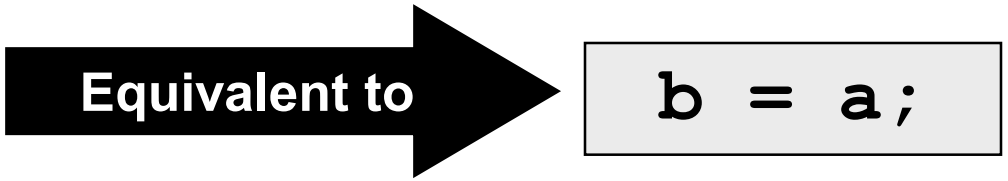
Output

```
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
```

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;
```



Example

```
#include  <stdio.h>
int 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);
    return 0;
}
```

Equivalent

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

Example

```
int 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);
    return 0;
}
```

Suppose that

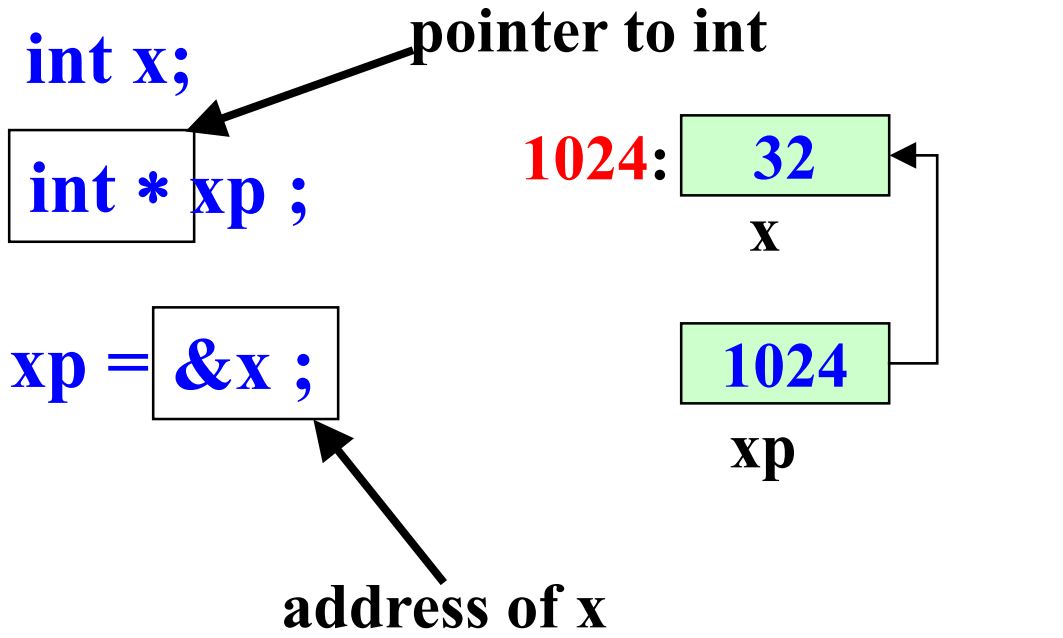
```
Address of x:      3221224908
Address of y:      3221224904
Address of ptr:    3221224900
```

Then output is

```
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
```

Example




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

Value of the pointer

- 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

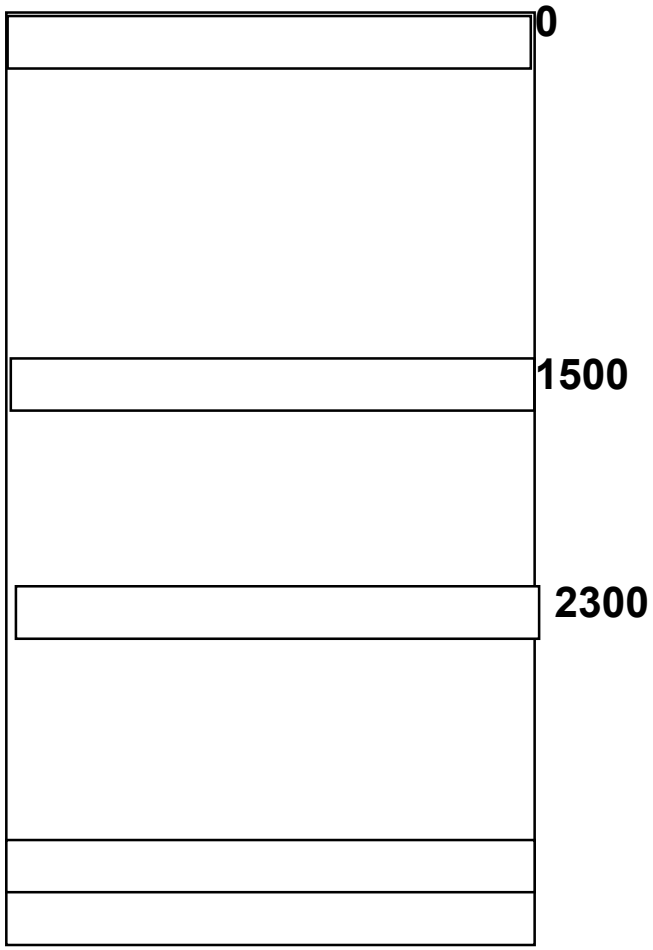
- After declaring a pointer:

```
int *ptr;
```

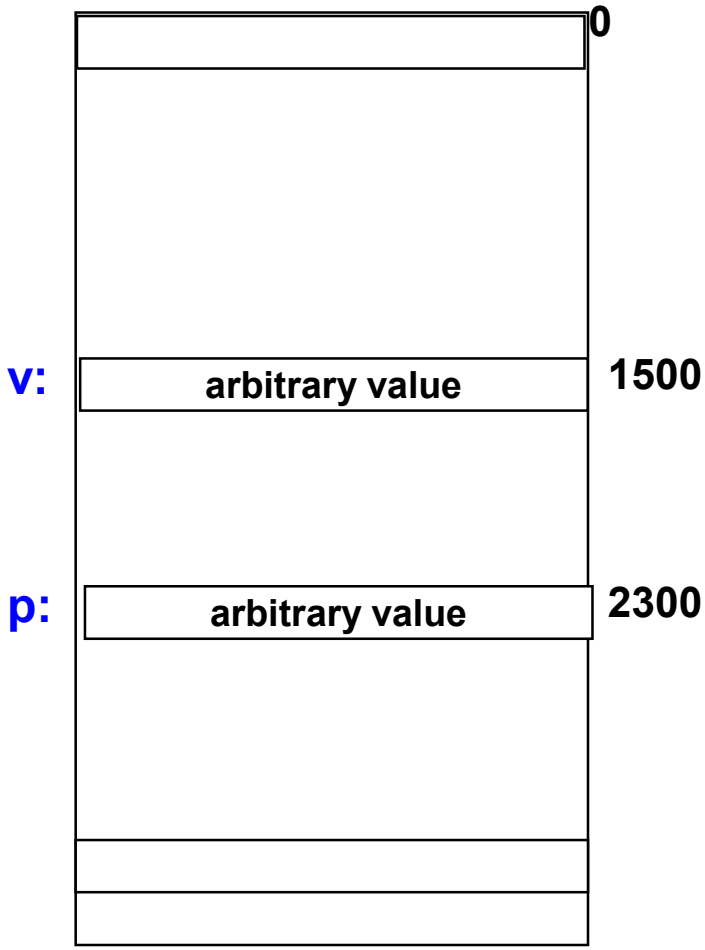
ptr doesn't actually point to anything yet. We can either:

  - make it point to something that already exists, or
  - allocate room in memory for something new that it will point to… (dynamic allocation, to be done later)

Example

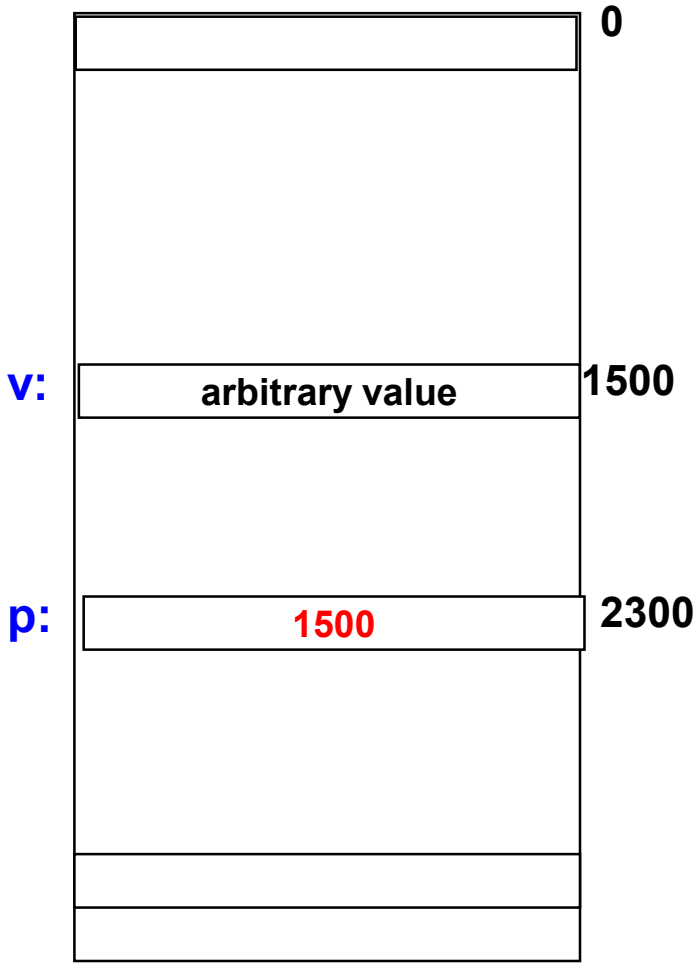


Memory and Pointers:



Memory and Pointers:

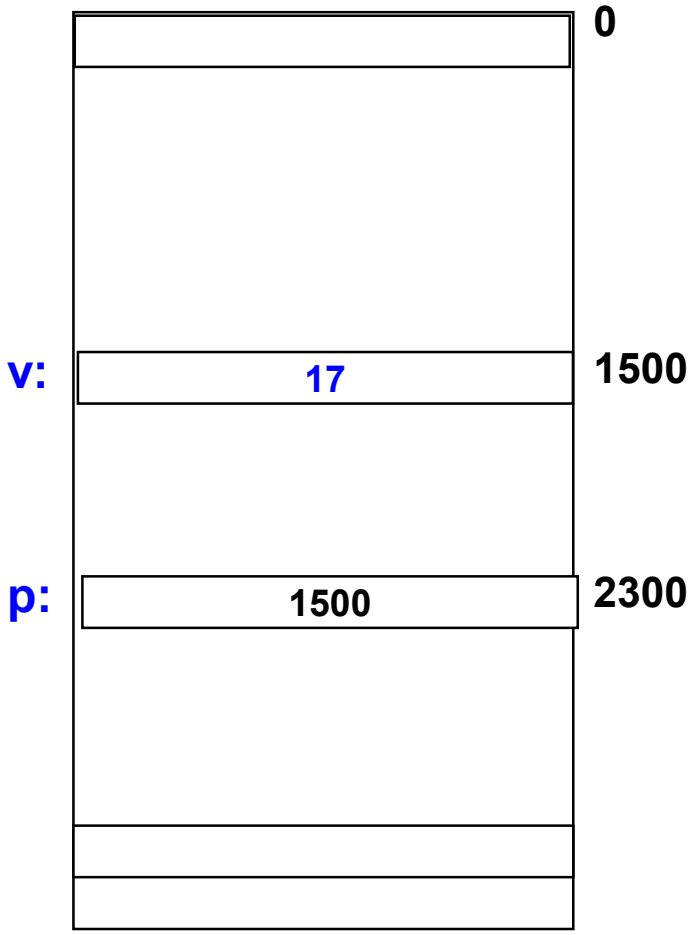
int *p, v;



Memory and Pointers:

int v, *p;

p = &v;

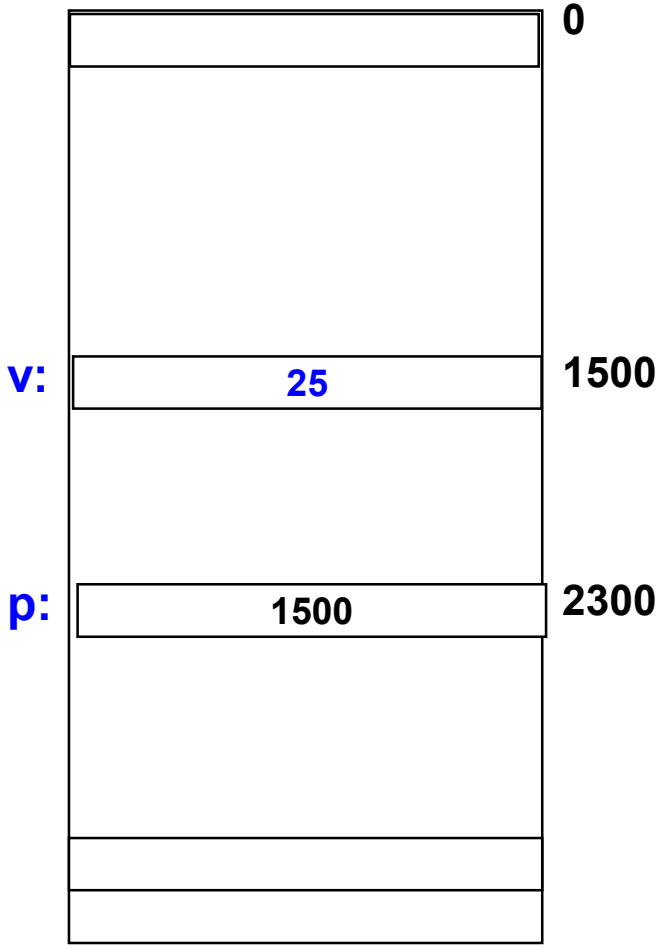


Memory and Pointers:

int v, *p;

p = &v;

v = 17;



Memory and Pointers:

```
int v, *p;
p = &v;
v = 17;
*p = *p + 4;
v = *p + 4
```

More Examples of Using Pointers 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

- Note that this unary * has higher precedence than all arithmetic/relational/logical operators

Important 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 wrong output

- Never assign an absolute address to a pointer variable

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

- Whenever you use *p to access the value of the location pointed to by a pointer variable p, always check that p has been assigned a valid value before by an assignment statement (p = …..)
  - Very common mistake while writing programs with pointers

```
int main()
{
    int *p;
    *p = 4;
    printf("*p = %d\n", *p);
}
```

Run it and see what happens. p is not assigned anything. So whatever the content of p is, when *p is done, it tries to write to that location. So if p contained 1325 (say), it will try to write at memory location with address 1325. This may cause an error (OS does not allow writes to some addresses) or will overwrite whatever that location contained, which may corrupt other variable values. Second case is very hard to debug, as to the compiler 1325 is a free location and can be given to other variables later, which will then overwrite again.

Pointer Expressions

- Like other variables, pointer variables can appear in expressions
- 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?
  - Adding 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

- The scale factor indicates the number of bytes used to store a value of that type
  - So the address of the next element of that type can only be at the (current pointer value + size of data)
- The exact scale factor may vary from one machine to another
- Can be found out using the sizeof function
  - Gives the size of that data type
- Syntax:

  sizeof (data_type)

Example

```
int main()
{
  printf ("No. of bytes in int is %u \n",       sizeof(int));
  printf ("No. of bytes in float is %u \n",     sizeof(float));
  printf ("No. of bytes in double is %u \n", sizeof(double));
  printf ("No. of bytes in char is %u \n",      sizeof(char));

  printf ("No. of bytes in int * is %u \n",       sizeof(int *));
  printf ("No. of bytes in float * is %u \n",     sizeof(float *));
  printf ("No. of bytes in double * is %u \n",  sizeof(double *));
  printf ("No. of bytes in char * is %u \n",      sizeof(char *));
  return 0;
}
```

Output on a PC

```
No. of bytes in int is 4
No. of bytes in float is 4
No. of bytes in double is 8
No. of bytes in char is 1
No. of bytes in int * is 4
No. of bytes in float * is 4
No. of bytes in double * is 4
No. of bytes in char * is 4
```

- Note that pointer takes 4 bytes to store, independent of the type it points to
- However, this can vary between machines
  - Output of the same program on a server

```
No. of bytes in int is 4
No. of bytes in float is 4
No. of bytes in double is 8
No. of bytes in char is 1
No. of bytes in int * is 8
No. of bytes in float * is 8
No. of bytes in double * is 8
No. of bytes in char * is 8
```

- Always use sizeof() to get the correct size`
- Should also print pointers using **%p** (instead of %u as we have used so far for easy comparison)

Example

```
int main()
{
  int A[5],  i;

  printf("The addresses of the array elements are:\n");
  for (i=0; i<5; i++)
     printf("&A[%d]: Using \%p = %p,   Using \%u = %u", i, &A[i], &A[i]);
  return 0;
}
```

Output on a server machine

```
&A[0]: Using %p = 0x7fffb2ad5930,   Using %u = 2997705008
&A[1]: Using %p = 0x7fffb2ad5934,   Using %u = 2997705012
&A[2]: Using %p = 0x7fffb2ad5938,   Using %u = 2997705016
&A[3]: Using %p = 0x7fffb2ad593c,   Using %u = 2997705020
&A[4]: Using %p = 0x7fffb2ad5940,   Using %u = 2997705024
```

0x7fffb2ad5930 = 140736191093040 in decimal (**NOT 2997705008**) so print with %u prints a wrong value (4 bytes of unsigned int cannot hold 8 bytes for the pointer value)

Pointers and Arrays

Pointers and Arrays

- When an array is declared,
  - The compiler allocates 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 each integer requires 4 bytes
- Compiler allocates a contiguous storage of size 5x4 = 20 bytes
- Suppose the starting address of that storage is 2500

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

Contd.

- The array name x is the starting address of the array
  - Both x and &x[0] have the value 2500
  - x is a constant pointer, so cannot be changed
    - X = 3400, x++, x += 2 are all illegal
- If int *p is declared, then

  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
```

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

- C knows the type of each element in array x, so knows how many bytes to move the pointer to get to the next element

Example: function to find average

```
int 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));
  return 0;
}
```

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

  p = array;

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

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

The pointer p can be subscripted also just like an array!

```
int 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));
  return 0;
}
```

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

  p = array;

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

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

Important to remember

- Pitfall: An array in C does not know its own length, & bounds not checked!
  - Consequence: While traversing the elements of an array (either using [ ] or pointer arithmetic), we can accidentally access off the end of an array (access more elements than what is there in the array)
  - Consequence: We must pass the array and its size to a function which is going to traverse it, or there should be some way of knowing the end based on the values (Ex., a –ve value ending a string of +ve values)
- Accessing arrays out of bound can cause strange problems
  - Very hard to debug
  - Always be careful when traversing arrays in programs

Pointers to Structures

Pointers to Structures

- Pointer variables can be defined to store the address of structure variables
- Example:

```
struct student {
        int   roll;
        char  dept_code[25];
        float cgpa;
      };
struct student *p;
```

- Just like other pointers, p does not point to anything by itself after declaration
  - Need to assign the address of a structure to p
  - Can use & operator on a struct student type variable
  - Example:

```
struct student x, *p;
scanf("%d%s%f", &x.roll, x.dept_code, &x.cgpa);
p = &x;
```

- Once p points to a structure variable, the members can be accessed in one of two ways:
  - (*p).roll, (*p).dept_code, (*p).cgpa
    - Note the ( ) around *p
  - p –> roll, p –> dept_code, p –> cgpa
    - The symbol –> is called the arrow operator
- Example:
  - printf("Roll = %d, Dept.= %s, CGPA = %f\n", (*p).roll, (*p).dept_code, (*p).cgpa);
  - printf("Roll = %d, Dept.= %s, CGPA = %f\n", p->roll, p->dept_code, p->cgpa);

Pointers and Array of Structures

- Recall that the name of an array is the address of its 0-th element
  - Also true for the names of arrays of structure variables
- Consider the declaration:

```
struct student class[100],  *ptr ;
```

Pointers and Array of Structures

- Recall that the name of an array is the address of its 0-th element
  - Also true for the names of arrays of structure variables
- Consider the declaration:

```
struct student class[100],  *ptr ;
```

- The name class represents the address of the 0-th element of the structure array
  - ptr is a pointer to data objects of the type struct student
- The assignment

  ptr = class;

  will assign the address of class[0] to ptr
- Now ptr->roll is the same as class[0].roll. Same for other members
- When the pointer ptr is incremented by one (ptr++) :
  - The value of ptr is actually increased by sizeof(struct student)
  - It is made to point to the next record
  - Note that sizeof operator can be applied on any data type

```
struct student {
    char name[20];
    int roll;
};
int main()
{
    struct student class[50], *p;
    int i, n;
    scanf("%d", &n);
    for (i=0; i<n; i++)
        scanf("%s%d", class[i].name, &class[i].roll);
    p = class;
    for (i=0; i<n; i++) {
        printf("%s  %d\n", class[i].name, class[i].roll);
        printf("%s  %d\n", (*(p+i)).name, (*(p+i)).roll);
        printf("%s  %d\n", (p+i)->name, (p+i)->roll);
        printf("%s  %d\n", p[i].name, p[i].roll);
    }
}
```

Output

```
3
Ajit 1001
Abhishek 1005
Riya 1007
Ajit  1001
Ajit  1001
Ajit  1001
Ajit  1001
Abhishek  1005
Abhishek  1005
Abhishek  1005
Abhishek  1005
Riya  1007
Riya  1007
Riya  1007
Riya  1007
```

A Warning

- When using structure pointers, be careful 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 ptr->roll, not ptr
    - (++ptr) –> roll will access (ptr + 1)->roll (for example, if you want to print the roll no. of all elements of the class array)
- When not sure, use (and) to force what you you want