Pointers
Part 1

Introduction

- Whenever we declare a variable, the system allocates memory to store the value of the variable.
  - Since every byte in memory has a unique address, this location will also have its own (unique) address.
- 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.).
- A pointer is a variable that represents the location (rather than the value) of a data item.

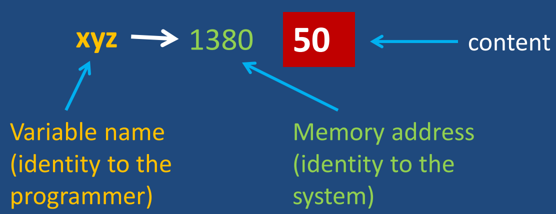
Example

- Consider the statement

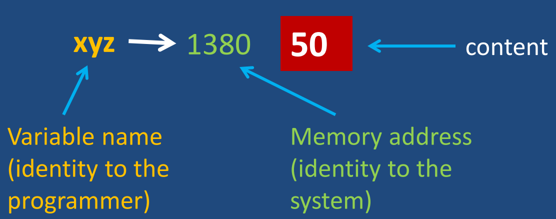
  int xyz = 50;

  – 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 address location chosen is 1380.



Example



Remember

```
scanf("%d",&xyz);
xyz=50;
printf("%d",xyz);
```

Access Protocol

1. During execution of the program, the system always associates the name xyz with the address 1380
2. Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory.
3. The value 50 can be accessed by using either the name xyz or the address 1380.

Pointers

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

Variable	Value	Address
xyz	50	1380
p	1380	2545

p = &xyz;

Example

```
#include <stdio.h>
int main()
{
      char a='A';
      int b=100;
      long int c=100;
      float d=100.0;
      double e=100.0;

      printf("a: size is %d, address is %x and content is %c\n",  sizeof(a), &a,a);
      printf("b: size is %d, address is %x and content is %d\n",  sizeof(b), &b,b);
      printf("c: size is %d, address is %x and content is %ld\n", sizeof(c), &c, c);
      printf("d: size is %d, address is %x and content is %f\n",  sizeof(d), &d, d);
      printf("e: size is %d, address is %x and content is %lf\n", sizeof(e), &e, e);

      return 0;
}
```

Returns no. of bytes required for data type representation

Example Output

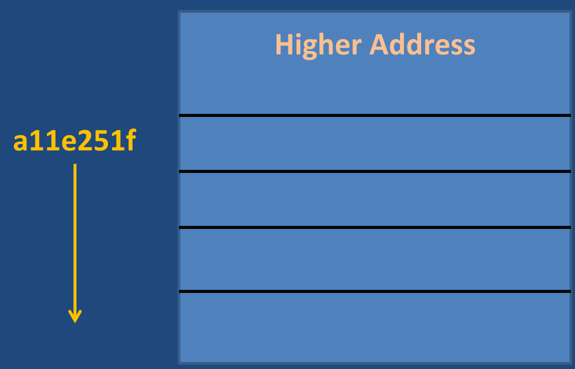
a: size is 1, address is a11e251f and content is A

b: size is 4, address is a11e2518 and content is 100

c: size is 8, address is a11e2510 and content is 100

d: size is 4, address is a11e250c and content is 100.000000

e: size is 8, address is a11e2500 and content is 100.000000



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:

```
int xyz;

p = &xyz;    // the address of xyz is assigned to p.
```

What is the data type of p?

Declaration of pointer

- int xyz;
- int *p;
- p=&xyz;

Summary

int *p;

Data Type Pointer Variable

- printf("%d",xyz); is equivalent to printf("%d",*p);
- So xyz and *p can be used for same purpose.
- Both can be declared simultaneously.
  - Example:
    - int xyz,*p;

Data Type

- Pointer must have a data type. That is the data type of the variable whose address will be stored.
  - int xyz, *p; // p is the pointer to data of type int.
  - float abc, *p1; // p1 is the pointer to data of type float.
  - long int pqr, *p2; // p2 is the pointer to data of type long int.

NOTE

int *ptr and int* ptr are same.

However the first one helps you to declare in one statement:
int *ptr, var1;

Remember

int x;
float *a;
a=&x; // NOT ALLOWED

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

Example

```
#include  <stdio.h>
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;
}
```

*&x⇔x

ptr=&x;
&x⇔&*ptr

Example Output

Output:

10 is stored in location 3599592540
10 is stored in location 3599592540
10 is stored in location 3599592540
10 is stored in location 3599592540
3599592540 is stored in location 3599592528
10 is stored in location 3599592536

Now x = 25

Address of x: 3599592540
Address of y: 3599592536
Address of ptr: 3599592528

Dereferencing Pointers

- Dereferencing is an operation performed to access and manipulate data contained in the memory location.
- A pointer variable is said to be dereferenced when the unary operator *, in this case called the indirection operator, is used like a prefix to the pointer variable or pointer expression.
- An operation performed on the dereferenced pointer directly affects the value of the variable it points to.

Example

```
#include<stdio.h>
int main()
{
        int *iptr, var1, var2;
        iptr=&var1;
        *iptr=25;
        *iptr += 10;
        printf("variable var1 contains %d\n",var1);
        var2=*iptr;
        printf("variable var2 contains %d\n",var2);
        iptr=&var2;
        *iptr += 20;
        printff("variable var2 now has %d\n",var2);
        return 0;
}
```

Example

```
variable var1 contains 35
variable var2 contains 35
variable var2 now has 55
```

Thus the two use of * are to be noted.

int *p for declaring a pointer variable

*p=10 is for indirection to the value in the address pointed by the variable p.

This power of pointers is often useful, where direct access via variables is not possible.

Typecasting

- Typecasting is mostly not required in a well written C program. However, you can do this as follows:
  - char c = '5‘
  - char *d = &c;
  - int *e = (int*)d;
  - Remember (sizeof(char) != sizeof(int))

Typecasting

- **void pointers**
  - Pointers defined to be of specific data type cannot hold the address of another type of variable.
  - It gives syntax error on compilation. Else use a void pointer (which is a general purpose pointer type), which can point to variables of any data type.
  - But while dereferencing, we need an explicit type cast.

Example

```
#include<stdio.h>
int main()
{
        float pi=3.14128;
        int num=100;
        void *p;
        p=&pi;
        printf("First p points to a float variable and access pi=%.5f\n",
                *((float *)p));
        p=&num;
        printf("Then p points to an integer variable and access num=%d\n",
                *((int *)p));
        return 0;
}
```

Output

First p points to a float variable and access pi=3.14128
Then p points to an integer variable and access num=100

Pointers to Pointers

- Pointer is a type of data in C
  - hence we can also have pointers to pointers
- Pointers to pointers offer flexibility in handling arrays, passing pointer variables to functions, etc.
- General format:
  - <data_type> **<ptr_to_ptr>;

    <ptr_to_ptr> is a pointer to a pointer pointing to a data object of the type <data_type>
- This feature is often made use of while passing two or more dimensional arrays to and from different functions.

Example

```
#include<stdio.h>
int main()
{
        int *iptr;
        int **ptriptr;
        int data;
        iptr=&data;
        ptriptr=&iptr;
        *iptr=100;
        printf("variable 'data' contains %d\n",data);
        **ptriptr=200;
        printf("variable 'data' contains %d\n",data);
        data=300;
        printf("variable 'data' contains %d\n",**ptriptr);
        return 0;
}
```

Output

```
variable 'data' contains 100
variable 'data' contains 200
variable 'data' contains 300
```

Examples of pointer arithmetic

```
int a=10, b=5, *p, *q;
p=&a;
q=&b;
printf("*p=%d,p=%x\n",*p,p);
p=p-b;
printf("*p=%d,p=%x\n",*p,p);
printf("a=%d, address(a)=%x\n",a,&a);
```

Output:

```
*p=10, p=24b3f6ac
*p=4195592, p=24b3f698
a=10, address(a)=24b3f6ac
```

Examples of pointer arithmetic

```
#include<stdio.h>
int main()
{
        int a=10, b=5, *p, *q;
        p=&a;   q=&b;
        printf("*p=%d,p=%x\n",*p,p);
        p=p-b;
        p=p+a;
        printf("*p=%d,p=%x\n",*p,p);
        p=p-a;
        printf("*p=%d,p=%x\n",*p,p);
        p=p+b;
        printf("*p=%d,p=%x\n",*p,p);
        printf("Size of int: %d\n",sizeof(int));
        return 0;
}
```

Output

```
*p=10,p=c9b2bdc
*p=0,p=c9b2bf0
*p=4195651,p=c9b2bc8
*p=10,p=c9b2bdc
Size of int: 4
```

If a pointer p is to a type, d_type, when incremented by i, the new address p points to is:
current_address+i*sizeof(d_type)

Similarly for decrementation

Subtraction of Pointers

```
#include<stdio.h>
main()
{
        int *p, *q;
        float *f, *g;
        q=p+1;
        g=f+1;
        printf("%d\n",(int *)q-(int *)p);
        printf("%d\n",(float *)g-(float *)f);
}
```

When two pointers are subtracted, the results are of type size_t

Both the printf statement outputs 1.

Even though the numerical values of the pointers differ by 4 in case of integers/float, this difference is divided by the size of the type being pointed to.

Invalid Pointer Arithmetic

- p=-q;
- p<<=1;
- p=p+q;
- p=p+q+a;
- p=p*q;
- p=p*a;
- p=p/q;
- p=p/b;
- p=a/p;
- &235

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.

Pointers and Arrays

- The elements of an array can be efficiently accessed by using a pointer.
- Array elements are always stored in contiguous memory space.
- Consider an array of integers and an int pointer:
  - #define MAXSIZE 10
  - int A[MAXSIZE], *p;
- The following are legal assignments for the pointer p:
  - p = A; /* Let p point to the 0-th location of the array A */
  - p = &A[0]; /* Let p point to the 0-th location of the array A */
  - p = &A[1]; /* Let p point to the 1-st location of the array A */
  - p = &A[i]; /* Let p point to the i-th location of the array A */
- Whenever p is assigned the value &A[i], the value *p refers to the array element A[i].

Pointers and Arrays

- Pointers can be incremented and decremented by integral values.
- After the assignment p = &A[i]; the increment p++ (or ++p) lets p one element down the array, whereas the decrement p-- (or --p) lets p move by one element up the array. (Here "up" means one index less, and "down" means one index more.)
- Similarly, incrementing or decrementing p by an integer value n lets p move forward or backward in the array by n locations. Consider the following sequence of pointer arithmetic:
  - p = A; /* Let p point to the 0-th location of the array A */
  - p++; /* Now p points to the 1-st location of A */
  - p = p + 6; /* Now p points to the 8-th location of A */
  - p += 2; /* Now p points to the 10-th location of A */
  - --p; /* Now p points to the 9-th location of A */
  - p -= 5; /* Now p points to the 4-rd location of A */
  - p -= 5; /* Now p points to the (-1)-nd location of A */

Remember: Increment/ Decrement is by data type not by bytes.

Pointers and Arrays

- Oops! What is a negative location in an array?
- Like always, C is pretty liberal in not securing its array boundaries.
- As you may jump ahead of the position with the largest legal index, you are also allowed to jump before the opening index (0).
- Though C allows you to do so, your run-time memory management system may be unhappy with your unhealthy intrusion and may cause your program to have a premature termination (with the error message "Segmentation fault").
- It is the programmer's duty to ensure that his/her pointers do not roam around in prohibited areas.

Example

- Consider the declaration:

```
int *p;
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

  – 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
```

Accessing Array elements

```
#include<stdio.h>
int main()
{
        int iarray[5]={1,2,3,4,5};
        int i, *ptr;
        ptr=iarray;
        for(i=0;i<5;i++) {
                printf("iarray[%d] (%x): %d\n",i,ptr,*ptr);
                ptr++;
        }
        return 0;
}
```

Output

```
iarray[0] (f4c709d0): 1
iarray[1] (f4c709d4): 2
iarray[2] (f4c709d8): 3
iarray[3] (f4c709dc): 4
iarray[4] (f4c709e0): 5
```

Accessing Array elements

```
#include<stdio.h>
int main()
{
        int iarray[5]={1,2,3,4,5};
        int i, *ptr;
        ptr=iarray;
        for(i=0;i<5;i++) {
                printf("iarray[%d] (%x): %d\n",i,ptr,*ptr);
                ptr++;
                printf("iarray[%d] (%x): %d\n",i, (iarray+i),*(iarray+i));
        }
        return 0;
}
```

NOTE

1. The name of the array is the starting address (base address) of the array.
2. It is the address of the first element in the array.
3. Thus it can be used as a normal pointer, to access the other elements in the array.

More examples

```
#include<stdio.h>
int main()
{
        int i;
        int a[5]={1,2,3,4,5}, *p = a;
        for(i=0;i<5;i++,p++) {
                printf("%d %d",a[i],*(a+i));
                printf(" %d %d %d\n",*(i+a),i[a],*p);
        }
        return 0;
}
```

Output

```
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5
```

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.

Swapping two numbers

```
void swap(int *a, int *b)
{
        int temp = *a;
        *a = *b;
        *b = temp;
}
```

```
void swap(int a, int b)
{
        int temp = a;
        a = b;
        b = temp;
}
```

```
void main( )
{
    int i, j;
    scanf("%d %d", &i, &j);
    printf("After swap: %d %d",i,j);
    swap(&i,&j);
    printf("After swap: %d %d",i,j);
}
```

scanf Revisited

```
int  x,  y ;
printf ("%d %d %d",  x, y, x+y) ;
```

- What about scanf ?

```
scanf ("%d %d %d", x, y, x+y);
```
NO

```
scanf ("%d %d", &x, &y);
```
YES

Example: Sort 3 integers

- Three-step algorithm:
  1. Read in three integers x, y and z
  2. Put smallest in x
     - Swap x, y if necessary; then swap x, z if necessary.
  3. Put second smallest in y
     - Swap y, z if necessary.

Hints

```
#include  <stdio.h>
int main()
{
    int  x, y, z ;
    ……….
    scanf(“%d %d %d”, &x, &y, &z) ;
    if  (x > y)   swap (&x, &y);
    if  (x > z)   swap (&x, &z);
    if  (y > z)   swap (&y, &z) ;
    ……….
}
```

Passing Arrays to a Function

- An array name can be used as an argument to a function.
  - Permits the entire array to be passed to the function.
  - Array name is passed as the parameter, which is effectively the address of the first element.
- Rules:
  - The array name must appear by itself as argument, without brackets or subscripts.
  - The corresponding formal argument is written in the same manner.
    - Declared by writing the array name with a pair of empty brackets.
    - Dimension or required number of elements to be passed as a separate parameter.

Example: function to find average

```
#include  <stdio.h>
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;
}
```

int *array

```
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);
}
```

p[i]

The Actual Mechanism

- When an array is passed to a function, the values of the array elements are not passed to the function.
  - The array name is interpreted as the address of the first array element.
  - The formal argument therefore becomes a pointer to the first array element.
  - When an array element is accessed inside the function, the address is calculated using the formula stated before.
  - Changes made inside the function are thus also reflected in the calling program.

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 , etc.
```

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: Sorting by Roll Numbers

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

void main()
{
    struct  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;
                }
          }
   for  (k=0; k<n; k++)
       printf ("%d %s %f", class[k].roll,
            class[k].dept_code, class[k].cgpa);
}
```

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;
            };
struct stud class[100],  *ptr;
```

Pointers and Structures

– 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.

Pointers and Structures

- 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.

Example

```
#include <stdio.h>

typedef struct {
            float real;
            float imag;
           } COMPLEX;

void swap_ref(COMPLEX *a, COMPLEX *b)
{
  COMPLEX tmp;
   tmp=*a;
   *a=*b;
   *b=tmp;
}

void print(COMPLEX *a)
{
  printf("(%f,%f)\n",a->real,a->imag);
}

void main()
{
 COMPLEX x={10.0,3.0}, y={-20.0,4.0};

  print(&x); print(&y);
  swap_ref(&x,&y);
  print(&x); print(&y);
}
```

Output

```
(10.000000,3.000000)
(-20.000000,4.000000)
(-20.000000,4.000000)
(10.000000,3.000000)
```

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;
                  };
struct  complex  add  (struct complex x, struct complex y)
{
    struct  complex  t;
    t.re = x.re + y.re ;
    t.im = x.im + y.im ;
    return (t) ;
}

void 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);
}
```

Complex number addition using pointers

```
#include  <stdio.h>
struct  complex {
                    float  re;
                    float  im;
                  };
void  add  (struct complex  *x, struct complex  *y, struct complex  *t)
{
    t->re = x->re + y->re ;
    t->im = x->im + y->im ;
}
void 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);
}
```

Dynamic Memory Allocation

Basic Idea

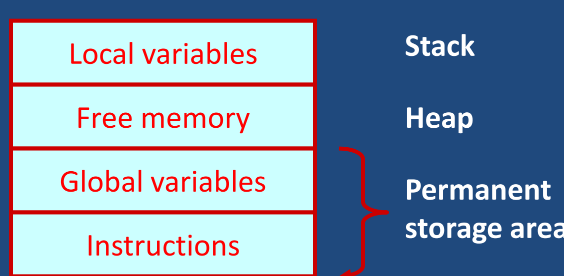
- Many a time we face situations where data is dynamic in nature.
  - Amount of data cannot be predicted beforehand.
  - Number of data item keeps changing during program execution.

- Such situations can be handled more easily and effectively using dynamic memory management techniques.

Basic Idea

- 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



Memory Allocation Process in C

- 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.

malloc()

- 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 assigned to any pointer type.

- General format:

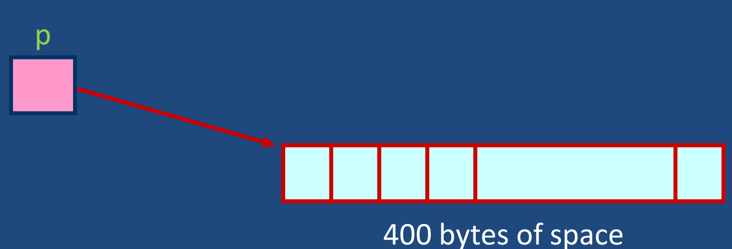
```
ptr  =  (type *)  malloc (byte_size) ;
```

malloc()

- 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.



malloc()

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

- Allocates 10 bytes of space for the pointer cptr of type char.

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

Determines the number of bytes required to store one structure data type viz., stud.

Point 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.

Example: malloc()

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
 int i,N;
 float *height;
 float sum=0,avg;

 printf("Input the number 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);
}
```

Output

```
Input the number of students.
5
Input heights for 5 students
23 24 25 26 27
Average height= 25.000000
```

calloc()

The **C** library function

– void ***calloc**(size_t nitems, size_t size)

allocates the requested memory and returns a pointer to it.

Allocates a block of memory for an array of *nitems* elements, each of them *size* bytes long, and initializes all its bits to zero.

calloc() or malloc()

- malloc() takes a single argument (memory required in bytes), while calloc() needs two arguments.
- malloc() does not initialize the memory allocated, while calloc() initializes the allocated memory to ZERO.
- calloc() allocates a memory area, the length will be the product of its parameters.

Example: calloc()

```c
#include <stdio.h>      /* printf, scanf, NULL */
#include <stdlib.h>     /* calloc, exit, free */

int main ()
{
  int i,n;
  int * pData;

  printf ("Amount of numbers to be entered: ");
  scanf ("%d",&i);

  pData = (int*) calloc (i,sizeof(int));
  if (pData==NULL) exit (1);
  for (n=0;n<i;n++)   {
              printf ("Enter number #%d: ",n+1);
              scanf ("%d",&pData[n]);
  }
  printf ("You have entered: ");
  for (n=0;n<i;n++)
              printf ("%d ",pData[n]);

  free (pData);
  return 0;
}
```

Output

```
Amount of numbers to be entered: 5
Enter number #1: 23
Enter number #2: 31
Enter number #3: 23
Enter number #4: 45
Enter number #5: 32
You have entered: 23 31 23 45 32
```

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 format:

```
free (ptr) ;
```

where ptr is a pointer to a memory block which has been already created using malloc() / calloc() / realloc().

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 by the statement

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

then reallocation of space may be done as

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

Altering the Size of a Block

- 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 . But, it does not free the original block.

Example: realloc()

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
	int *pa, *pb, n;
/* allocate an array of 10 int */
	pa = (int *)malloc(10 * sizeof *pa);
	if(pa) {
		printf("%zu bytes allocated. Storing ints: ", 10*sizeof(int));
		for(n = 0; n < 10; ++n)
			printf("%d ", pa[n] = n);
	}

	pb = (int *)realloc(pa, 1000000 * sizeof *pb); // reallocate array to a larger size
	if(pb) {
		printf("\n%zu bytes allocated, first 10 ints are: ", 1000000*sizeof(int));
		for(n = 0; n < 10; ++n)
			printf("%d ", pb[n]); // show the array
		free(pb);
	} else { // if realloc failed, the original pointer needs to be freed
		free(pa);
	}
	return 0;
}
```

Output

```
40 bytes allocated. Storing ints: 0 1 2 3 4 5 6 7 8 9
4000000 bytes allocated, first 10 ints are: 0 1 2 3 4 5 6 7 8 9
```