Pointers and File Handling

From variables to their addresses

Pallab Dasgupta
Professor,
Dept. of Computer Sc & Engg

Basics of Pointers

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.

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.).

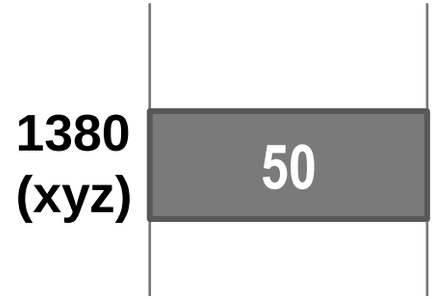
Whenever we declare a variable, the system allocates memory location(s) to hold the value of the variable.

- Since every byte in memory has a unique address, this location will also have its own (unique) address.

Example

Consider the statement

```
int  xyz = 50;
```



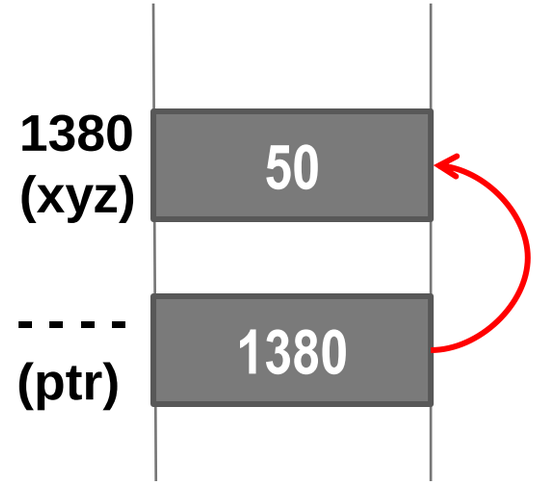
- This statement instructs the compiler to allocate a location for the integer variable xyz, and put the value 50 in that location.

- Suppose that the address location chosen is 1380.
- 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.

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.

Pointer Declaration

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

After declaring a pointer:

```
int *ptr;
```

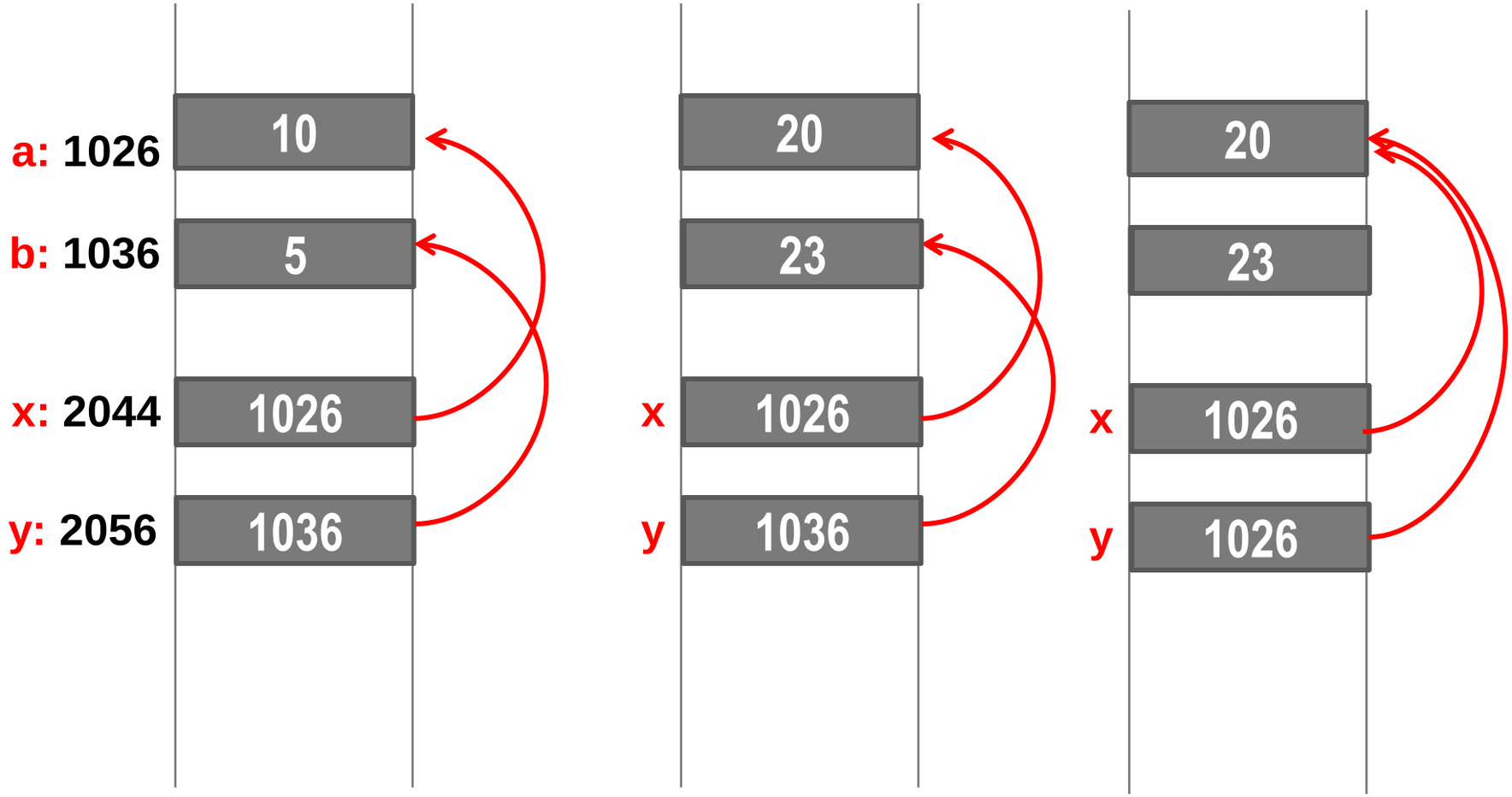
ptr doesn't actually point to anything yet.

We can either:

- make it point to some existing variable (which is in the stack), or
- dynamically allocate memory (in the heap) and make it point to it

Making it point

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



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]
```

Illegal usages

Following usages are illegal:

```
&235
```

- Pointing at constant.

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

- Pointing at array name.

```
&(a+b)
```

- Pointing at expression.

Pointer Declarations and Types

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.**

Pointers have types

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;          // This is an erroneous assignment
```

Assigning an absolute address to a pointer variable is prohibited.

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

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

More on pointer expressions

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.

More on pointer expressions

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 for an integer
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

```
struct  complex {
   float real;
   float imag;
};
struct complex x[10];

struct complex *p;

p = &x[0];                    // The pointer p now points to the first element of the array

p = p + 1;                    // Now p points to the second structure in the array
```

The increment of *p* is not by one byte, but by the size of the data type to which *p* points.

This is why we have many data types for pointers, 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 integer pointer, then

  p1++

  will increment the value of p1 by 4.

Scale factor may be machine dependent

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

```
#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.**

Passing arguments by value or 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=5, b=20

```
#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

Example (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]

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;
}
```

This is incorrect

```
char *foo( ) {
  char *string;
  string = malloc(32);  // Dynamic memory allocation
  return string;
}
```

This is okay

Arrays In Functions

An array parameter can be declared as an array or a pointer; an array argument can be passed as a pointer

```
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 ;
```

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

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

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
```

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.

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.

Use of pointers to structures

```
#include <stdio.h>
struct complex  {
                float  real;
                float  imag;
};

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

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

Dynamic Memory Allocation

Basic Idea

Sometimes 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.

Dynamic Memory Allocation

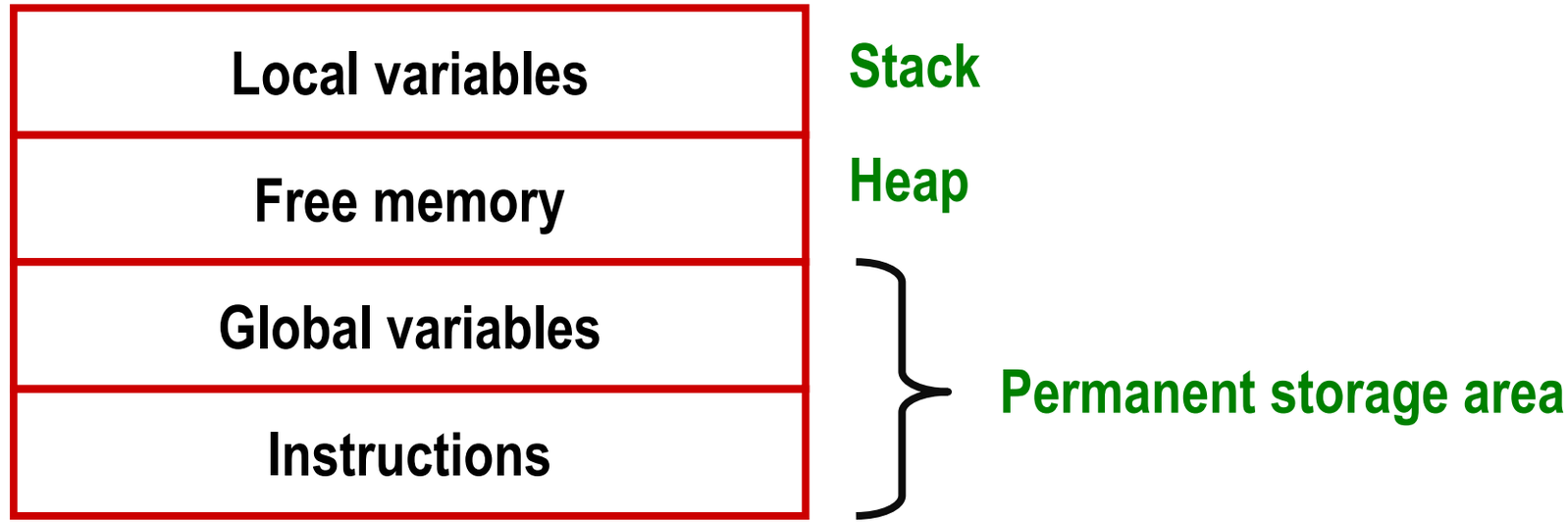
Normally the number of elements in an array is specified in the program

- **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

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:

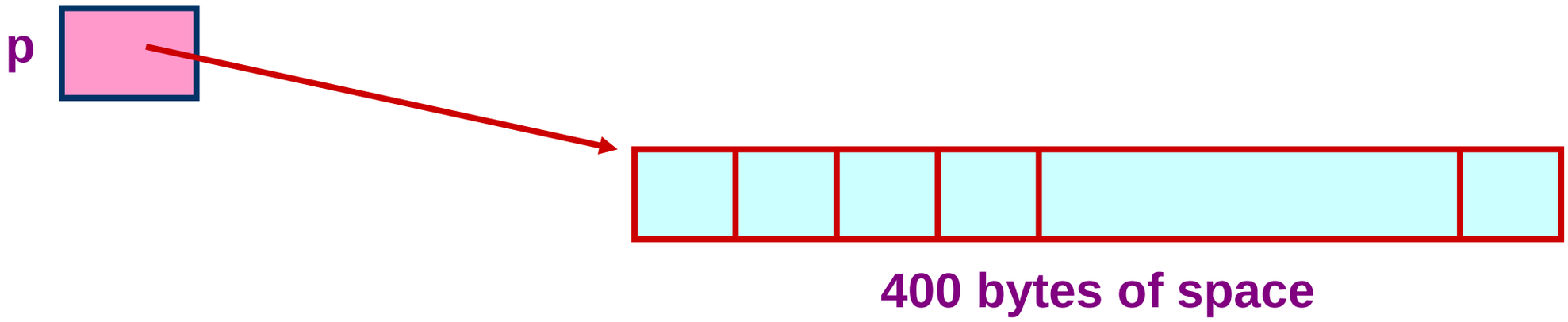
```
ptr =  (type *) malloc (byte_size);
```

Continued

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`.



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( ) ;
}
```

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.

File Handling

What is a file?

A named collection of data, stored in secondary storage (typically).

Typical operations on files:

- Open
- Read
- Write
- Close

How is a file stored?

- Stored as sequence of bytes, logically contiguous (may not be physically contiguous on disk).

File Types

- The last byte of a file contains the end-of-file character (`EOF`), with ASCII code 1A (hex).
- While reading a text file, the EOF character can be checked to know the end.

Two kinds of files:

- Text :: contains ASCII codes only
- Binary :: can contain non-ASCII characters
  - Image, audio, video, executable, etc.
  - To check the end of file here, the *file size* value (also stored on disk) needs to be checked.

File handling in C

In C we use `FILE *` to represent a pointer to a file.

`fopen` is used to open a file. It returns the special value `NULL` to indicate that it is unable to open the file.

```
FILE *fptr;
char filename[ ]= "file2.dat";

fptr = fopen (filename,"w");

if (fptr == NULL) {
  printf (“ERROR IN FILE CREATION”);
    /* DO SOMETHING */
}
```

Modes for opening files

The second argument of `fopen` is the *mode* in which we open the file. There are three modes.

"r" opens a file for reading.

"w" creates a file for writing, and writes over all previous contents (deletes the file so be careful!).

"a" opens a file for appending – writing on the end of the file.

Binary Files

We can add a “b” character to indicate that the file is a *binary* file.

- “rb”, “wb” or “ab”

```
fptr = fopen (“xyz.jpg”, “rb”);
```

The exit( ) function

Sometimes error checking means we want an "*emergency exit*" from a program.

In `main()` we can use `return` to stop.

In functions we can use `exit()` to do this.

Exit is part of the stdlib.h library.

```
exit(0);
```

exits the program

Usage of exit( )

```
FILE *fptr;
char filename[ ]= "file2.dat";
fptr = fopen (filename,"w");

if (fptr == NULL) {
  printf (“ERROR IN FILE CREATION”);
  exit(0);
}
```

Writing to a file using fprintf( )

`fprintf()` works just like `printf()` and `sprintf()`

except that its first argument is a file pointer.

```
int a=10, b=5;
FILE *fptr;
fptr = fopen ( "file.dat", "w“ );

fprintf (fptr, "Hello World!\n");
fprintf (fptr, “%d %d”, a, b);
```

Reading Data Using fscanf( )

```
int x, y;
FILE *fptr;
fptr = fopen (“input.dat”, “r”);

fscanf (fptr, “%d%d”, &x, &y);
```

The file pointer moves forward with each read operation

Reading lines from a file using fgets( )

We can read a string using `fgets()`.

```
FILE *fptr;
char line [1000];
……                                          /* Open file and check it is open */
while (fgets(line, 1000, fptr) != NULL)
{
   printf (“We have read the line: %s\n", line);
}
```

fgets( ) takes 3 arguments – a string, maximum number of characters to read, and a file pointer.
It returns NULL if there is an error (such as EOF).

Closing a file

We can close a file simply using `fclose()` and the file pointer.

```
FILE *fptr;
char filename[ ]= "myfile.dat";

fptr = fopen (filename,"w");

if (fptr == NULL) {
    printf ("Cannot open file to write!\n");
    exit(0);
}

fprintf (fptr,"Hello World of filing!\n");
fclose (fptr);
```

Three special streams

Three special file streams are defined in the `<stdio.h>` header

- stdin reads input from the keyboard
- stdout send output to the screen
- stderr prints errors to an error device (usually also the screen)

What might this do?

```
fprintf (stdout,"Hello World!\n");
```

An example program

```
#include <stdio.h>
main( )
{
     int i;

    fprintf(stdout,"Give value of i \n");
    fscanf(stdin,"%d",&i);
    fprintf(stdout,"Value of i=%d \n",i);
    fprintf(stderr,"No error: But an example to  show error message.\n");
 }
```

Output:

```
Give value of i
15
Value of i=15
No error: But an example to show error message.
```

Input File & Output File redirection

One may redirect the standard input and standard output to other files (other than `stdin` and `stdout`).

Usage: Suppose the executable file is `a.out`:

```
$ ./a.out <in.dat  >out.dat
```

`scanf()` will read data inputs from the file "`in.dat`", and `printf()` will output results on the file "`out.dat`".

A Variation

```
$ ./a.out <in.dat  >>out.dat
```

`scanf()` will read data inputs from the file "`in.dat`", and
`printf()` will append results at the end of the file "`out.dat`".

Reading and Writing a character

A character reading/writing is equivalent to reading/writing a byte.

```
int getchar( );
int putchar(int  c);
```

} stdin, stdout

```
int fgetc(FILE *fp);
int fputc(int c, FILE *fp);
```

} file

Example:

```
char c;
c = getchar();
putchar(c);
```

Command Line Arguments

What are they?

A program can be executed by directly typing a command at the operating system prompt.

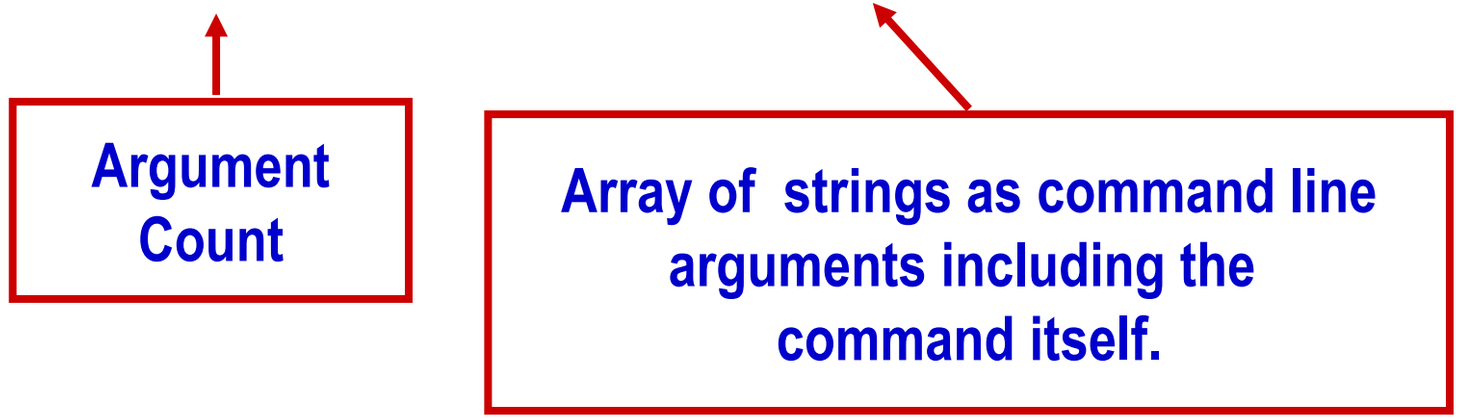
```
$ cc -o test test.c
$ ./a.out in.dat out.dat
$ prog_name param_1 param_2 param_3 ..
```

- The individual items specified are separated from one another by spaces.
  - First item is the program name.
- Variables *argc* and *argv* keep track of the items specified in the command line.

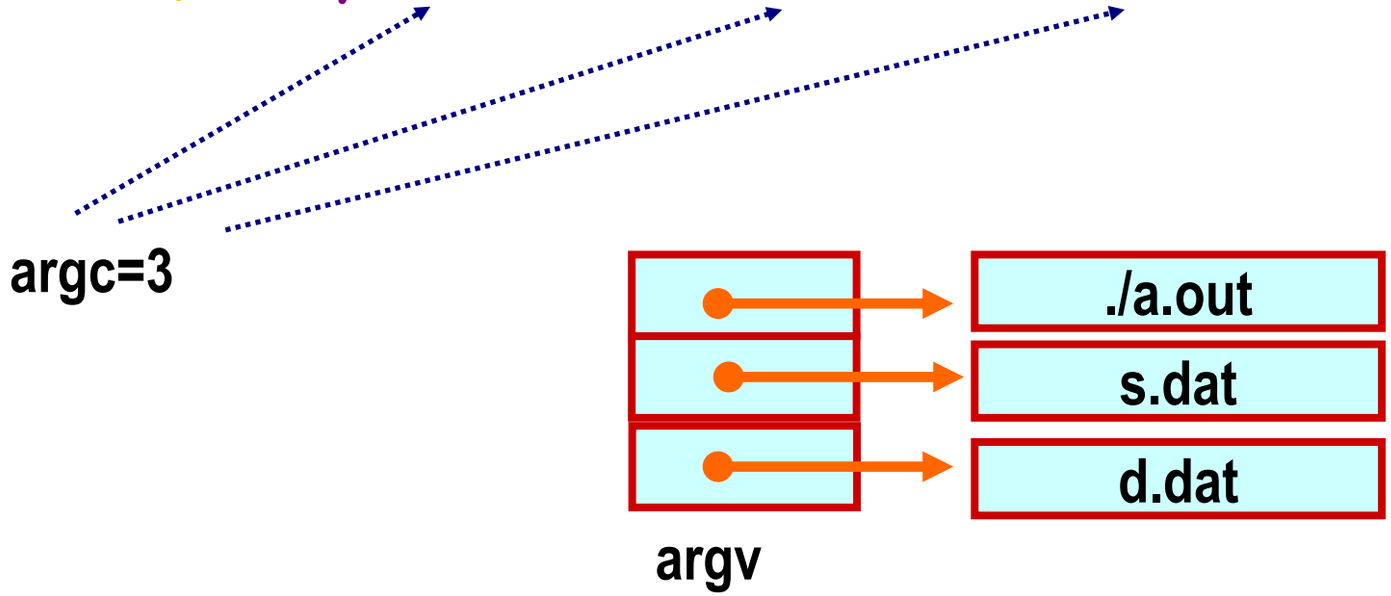
How to access them?

Command line arguments may be passed by specifying them under `main()`.

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



```
$  ./a.out  s.dat  d.dat
```



argv[0] = “./a.out” argv[1] = “s.dat” argv[2] = “d.dat”

Example: Program for Copying a File

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

int main( int argc, char *argv[ ] )
{
     FILE *ifp, *ofp;
     int i, c;
     char src_file[100], dst_file[100];

     if (argc!=3) {
                    printf ("Usage: ./a.out <src_file> <dst_file> \n");  exit(0);
     }
    else {
                    strcpy (src_file, argv[1]);   strcpy (dst_file, argv[2]);
}
```

Example: contd.

```
if ((ifp = fopen(src_file,"r")) == NULL)  {
    printf (“Input File does not exist.\n");   exit(0);
 }

 if ((ofp = fopen(dst_file,"w")) == NULL)  {
    printf (“Output File not created.\n");   exit(0);
 }

 while ((c = fgetc(ifp)) != EOF)  fputc (c,ofp);    // This is where the copying is done

 fclose(ifp);  fclose(ofp);
}
```