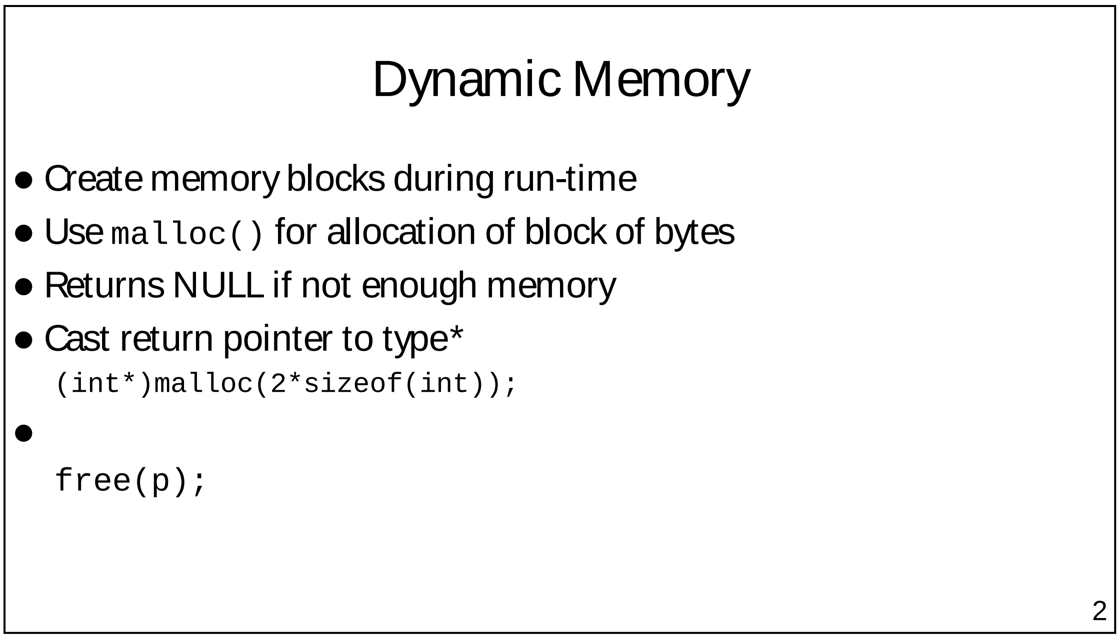
Dynamic memory



Dynamic Memory

- Create memory blocks during run-time
- Use `malloc()` for allocation of block of bytes
- Returns NULL if not enough memory
- Cast return pointer to type*

```
(int*)malloc(2*sizeof(int));
```

-

```
free(p);
```

2

Dynamic Memory

The arrays we created until now were always of a fixed size. These arrays are called static arrays.

```
void main()
{
    int arr[10]; /* Fixed array of 10 integer numbers */
}
```

If we need a bigger array we have to change its value and compile our code. One solution would be to create an array large enough for all our needs but than we would occupy memory which we perhaps will not use. A solution to this problem is by using so-called dynamic memory. By using a function *malloc()* we can request a block of memory during run-time. We specify how many bytes we want and the *malloc()* function allocates it for us and assigns it to our program. The size is the number of bytes we want to allocate. The *malloc()* function returns the address of the memory we requested. This address must be stored in a variable. Pointers can store address:

```
void main()
{
    int* p;

    p = (int*)malloc(20);
}
```

The function *malloc()* returns a void* thus we need to cast this pointer to the appropriate type.

The number of bytes we requested was 20 (10 times the size of an integer under DOS). This is very machine dependent. If an *int* value would occupy 4 bytes we would have an array of 5 integers instead of 10. To overcome this problem we calculate the number of bytes using the *sizeof()* operator.

```
void main()
{
    int* p;

    p = (int*)malloc(10*sizeof(int));
}
```

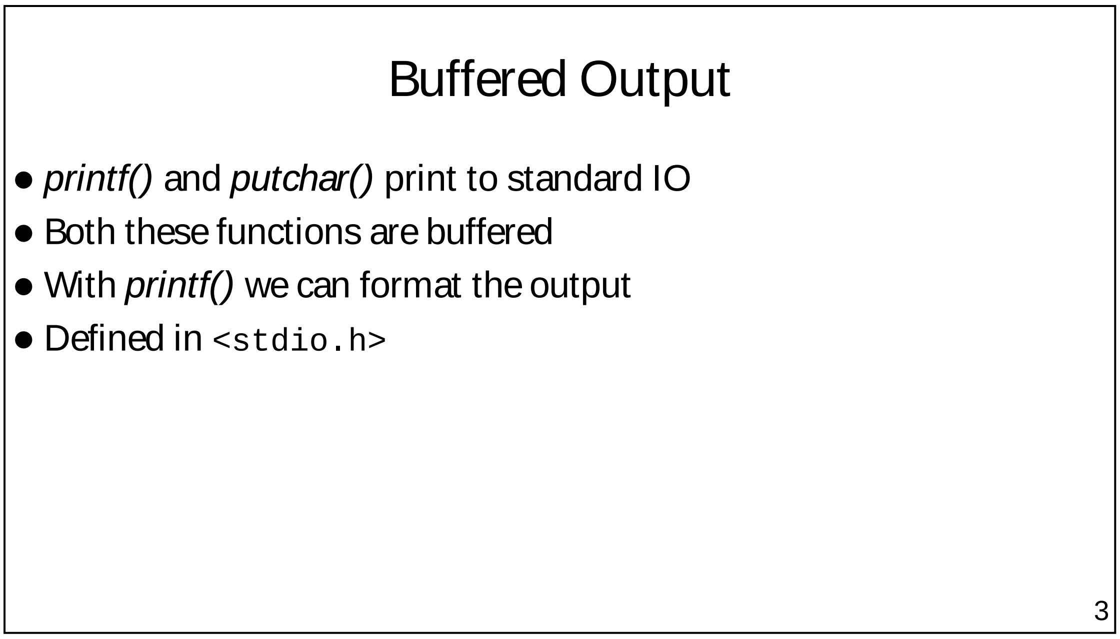
When we use dynamic memory we have to request it using *malloc()* but we also need to return it to the system when we do not use it anymore. If we do not return it to the system but keep requesting more memory we would run out of memory. The *free()* function returns the memory to the system.

```
void main()
{
    int* p;

    p = (int*)malloc(10*sizeof(int));

    /* Use the array */

    free(p);
}
```

The function *free()* does not check if the pointer points to something allocated using *malloc()*. If the pointer was not initialised using the *malloc()* function and we would try to *free()* it, unpredictable errors would occur.

If there is no more memory the *malloc()* function returns NULL. So always test if the *malloc()* function returned a valid address.

```
void main()
{
    int* p;

    p = (int*) malloc(10*sizeof(int));
    if (p != NULL)
    {
           /* Use the array */
           free(p);
    }
}
```



Buffered Output

- *printf()* and *putchar()* print to standard IO
- Both these functions are buffered
- With *printf()* we can format the output
- Defined in `<stdio.h>`

3

Output

In the previous modules we used the most simple form of printing to the screen, the *printf()* function. This function uses a format string to specify what needs to be printed which can be followed by arguments. The output of the *printf()* function is always the standard output. Usually this is the screen. It is possible to send the output of to a file using a DOS commando. On the command line you can specify where the output of a program should go to:

```
program.exe > filename
```

Until now we used the *printf()* function for simple output to the screen. It is however possible to use the *printf()* function for more complex output. In a previous module we showed a table with the specifiers for displaying the different types. These specifiers are placed after a '%'. Between the '%' and the specifier it is possible to place other format specifiers for creating an orderly output. The general form is:

```
%<flags><field width><.precision><type><format specifier>
```

Flags

Sign	Output
-	Place argument left in the available space (left align)
+	Always place the sign mark (+ or -) in front of the number
space	If first character is not a sign (+ or -) place a space
0 (null)	Fill available space using 0
#o	The octal number must be preceded with a 0 (null)
#x or #X	The hexadecimal number should be preceded by 0x or 0X
#e or #E	The number should always have a period '.'
#f	The number should always have a period '.'
#g or #G	The number should always have a period '.'

Field Width
The width of the output should be a number. The argument is printed with at least the specified width but can be larger than specified. If the argument contains fewer characters than specified the output is filled left with spaces (right aligned). If the flag '-' is specified these spaces are placed to the right (left aligned). If the flag '0' is specified the 0 will be used to fill instead of the space. It is possible to supply the field width as an argument. Instead of a number, a '*' is placed and the argument preceding the argument to be printed is used for the size. Of course this should be an int.

Period
A period is used to separate the field width from its precision.

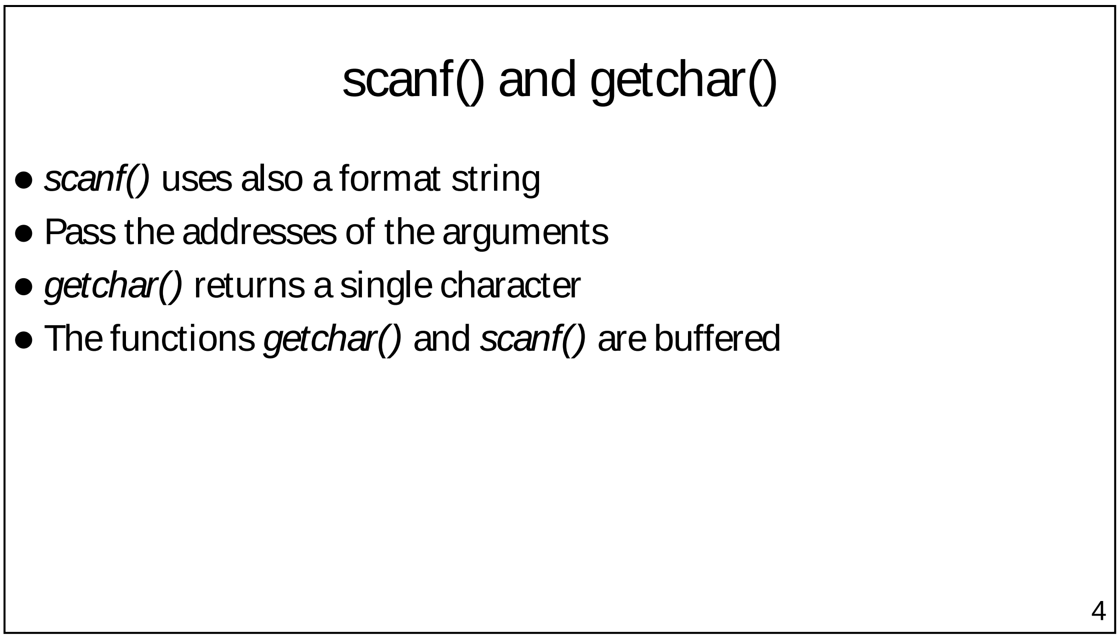
Precision
If the format specifier is 'e', 'E' or 'f' we can specify the number of digits. If the format specifier is 'g' or 'G' the precision specifies the number of significant digits. If the format specifier is 'd' or 'i', the precision is the minimum of numbers to print. If we use 's' as format specifier, the precision is the maximum number of characters displayed. We can again specify the precision using a '*'. The precision is then supplied as the argument preceding the argument we want to print.

Type Letter
These letters include 'h' and 'l' for integer types or 'L' for floating point types. The letter 'h' means a short number, the 'l' means a long and the 'L' is a long double.

```
#include <stdio.h>

void main()
{
    int number1 = 17;
    float number2 = 13.111111111f;
    float number3 = 13.000000000f;
    long number4 = 71234L;
    int width = 3;

    printf(":%6d:\n", number1);       // :    17:
    printf(":%-6d:\n", number1);      // :17    :
    printf(":%+-6d:\n", number1);     // :+17   :
    printf(":% -6d:\n", number1);     // : 17   :
    printf(":%06d:\n\n", number1);    // :000017:

    printf(":%o:\n", number1);        // :21:
    printf(":%#o:\n", number1);       // :021:
    printf(":%x:\n", number1);        // :11:
    printf(":%#x:\n", number1);       // :0x11:
    printf(":%g:\n", number3);        // :13:
    printf(":%#g:\n\n", number3);     // :13.0000:

    printf(":%4.3f:\n", number2);     // :13.111:
    printf(":%.6d:\n\n", number1);    // :000017:

    printf(":%ld:\n", number4);              // :71234:
    printf(":%.*f:\n", width, number2);      // :13.111:
}
```

The output:

```
:    17:
:17    :
:+17   :
: 17   :
:000017:

:21:
:021:
:11:
:0x11:
:13:
:13.0000:

:13.111:
:000017:

:71234:
:13.1111:
```



scanf() and getchar()

- *scanf()* uses also a format string
- Pass the addresses of the arguments
- *getchar()* returns a single character
- The functions *getchar()* and *scanf()* are buffered

4

Input

The functions *scanf()* and *getchar()* are used for input. These functions are both buffered. Buffered means that the characters the user types are first placed in a buffer which is then supplied to your program when the user presses enter. Both functions use standard input for getting there characters. In this case it is also possible to use an input file as input for the program. The *getchar()* function gets one character from standard input. The following program executes on the command line with a filename as input, would print its contents on the screen.

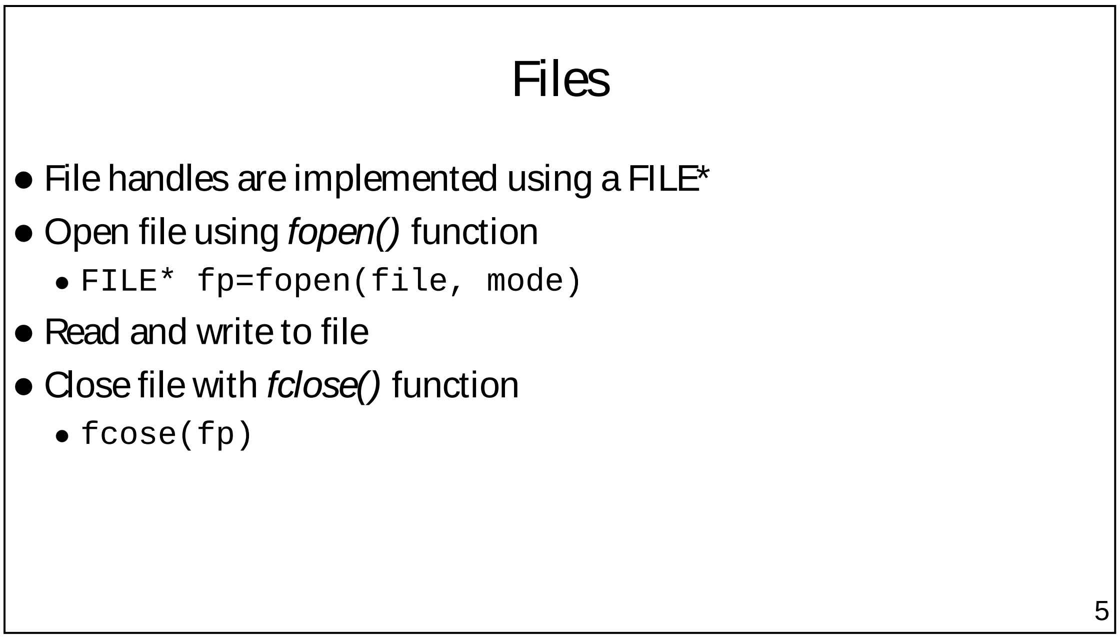
```
void main()
{
    int c;

    while ((c = getchar()) != EOF)
    {
          putchar(c);
    }
}

program.exe < inputfile
```

The *scanf()* function works almost the same as the *printf()* function. The first argument of this function is a format string specifying what types of values need to be entered by the user. These specifiers use the same format as that of *printf()*. The arguments following that format string are the variables where the input is to be placed in. The *scanf()* function uses pointer to pre-allocated variables for these arguments. Therefore the addresses of the variables need to be used. The format string of the *scanf()* function cannot be used to prompt the user with some text before getting the input. If a prompt needs to be displayed it needs to be printed using *printf()*.

```
void main()
{
    int number;
    char str[80];
    printf("Enter a number :");
    scanf("%d", &number);
    printf("Enter the string :");
    scanf("%s", str);
}
```

Notice that when we want to enter a string we do not specify the address of the array because the array is already the address. Remember from the array module that an array is implemented as being the address of the first element of the array.

The *getchar()* functions gets one character at a time. If the buffer contains more characters they stay in the buffer.



Files

- File handles are implemented using a FILE*
- Open file using *fopen()* function
  - `FILE* fp=fopen(file, mode)`
- Read and write to file
- Close file with *fclose()* function
  - `fcose(fp)`

5

Files

Files are always referenced using file handles. For storing the information about a file we need a variable of type *FILE*. The file handles are implemented in C by using *FILE**. File handles when we use the function *fopen()*. The *fopen()* function opens a file by specifying the name of the file and how it needs to be opened. This function returns the file handle (*FILE**). If opening the file failed the function returns *NULL*. The following example opens a file and exits if opening fails:

```
void main()
{
    FILE* fp;
    fp = fopen("File.txt", "r");
    if (fp == NULL)
    {
           printf("HELP\n");
           exit(1);
    }

    /* Program continues */

    fclose(fp);
}
```

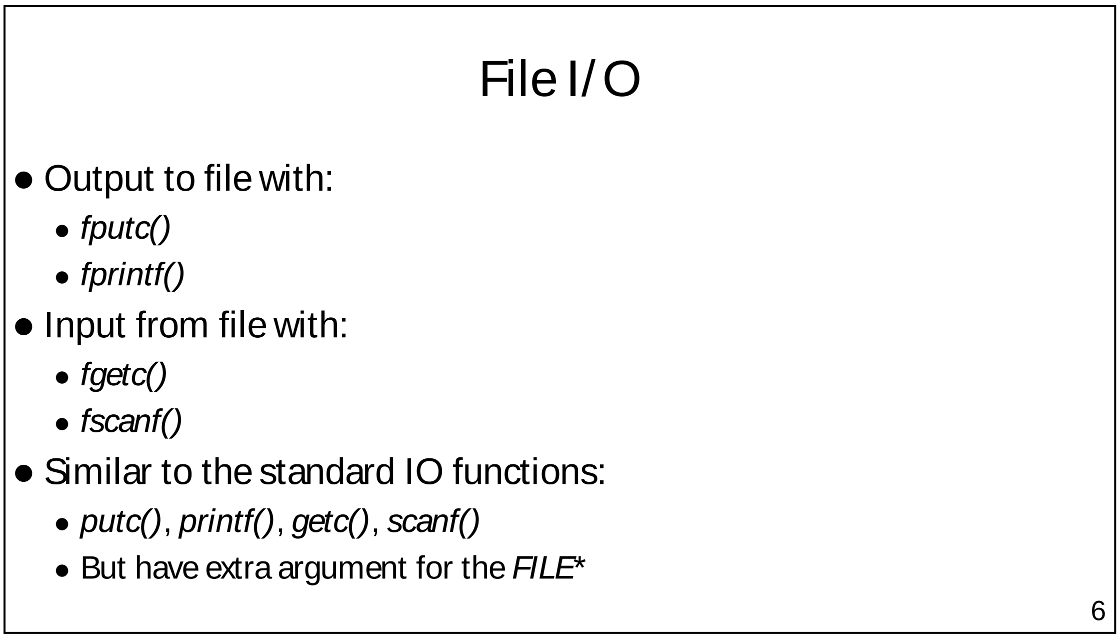
As you can see in the example the *fopen()* function has two arguments. The first argument is the file name and the second is a specification of how the file needs to be opened. The file is opened for read only in this case. The following table shows the available modes:

File open modes:

Mode	Open method
r	Open for read. The file must exist.
w	Open for write. If file does not exist, it is created first. Else the file is deleted first.
a	Append to file. If the file does not exist, it is created first.
r+	Open file for read and write. The file must exist.
w+	Open file for read and write. If file does not exist, it is created first. Else the file is deleted first.
a+	Open file for reading and appending. If the file does not exist, it is created first.
t	Open the file in text mode (default).
b	Open the file in binary mode.

These file modes create or open ASCII files. ASCII files are files which can be read using a text editor or at the command prompt using the *type* command. It is also possible to have so called binary files. In binary files all characters are stored using their value and not the ASCII representation of those characters. Binary files contain no readable characters. Binary files are specified using a 'b' in combination with the earlier mentioned file modes.

After opening and using a file we have to close it to make sure that all data is written to disk. We close a file with the *fclose()* function.



File I/O

- Output to file with:
  - *fputc()*
  - *fprintf()*
- Input from file with:
  - *fgetc()*
  - *fscanf()*
- Similar to the standard IO functions:
  - *putc()*, *printf()*, *getc()*, *scanf()*
  - But have extra argument for the *FILE**

6

Input and Output of a File

The functions we can use to get and put characters in a file we opened are almost the same as we used before for standard IO. The difference is that the file functions have an extra argument for the file pointer.

The function *fgetc()* gets one character at a time. Its argument is the file pointer.

```
int c;
c = fgetc(fp);
```

This function returns *EOF* if the end of a file is reached.

The *fprintf()* function is used the same as the *printf()* function but its first argument is the file pointer of the file to write to.

We can also print one character at a time using *fputc()*. Its first argument is the character to print and its second argument the file pointer of the output file. This function returns the character written or *EOF* if it failed.

```
if (fputc(c, fp) != c)
{
    printf("HELP");
}
```

Files and Strings

- Read a string with *fgets()*
- Write a string with *fputs()*

7

Files and Strings

In the previous section we showed that using *fscanf()* it is possible to get input from a file. If we specify “%s” we read a string from the input file. But when reading a string using *fscanf()* we do not read the entire line into the character buffer. The end of a string is a *space*, *tab* or *newline*. So if the input file has a sentence with a few spaces, only the first word is put in the character array. That is why it is sometimes better to use other functions for reading a string from a file, These functions are *fgets()* and *fputs()* which can be used to read and write strings in a file.

```
#define BUFLENGTH 80

void main()
{
    FILE* fp;
    char buffer[BUFLENGTH];

    fp = fopen("file.txt", "r");
    if (fp == NULL) exit(1);

    if (fgets(buffer, BUFLENGTH, fp) == NULL)
    {
          printf("HELP");
    }

    fclose(fp);
}
```

The first argument is the address of the buffer where the string is to be placed. The second argument specifies the maximum number of characters what can be read. *fgets()* reads one character less because of the ‘\0’ so the *BUFLENGTH* is inclusive the ‘\0’ character. The string is read until the ‘\n’ character is encountered or when the maximum number of characters is reached.

The *fputs()* function does the opposite. It writes a string to a file. The first argument is the string and the second the file handle. This function returns *EOF* if the operation fails.

Other Useful Functions

- String functions are defined in `<string.h>`
  `strcmp(), strcpy(), strcat(), etc.`
- Character functions defined in `<ctype.h>`
  `isalpha(), toupper(),`
  `tolower(), isdigit(), etc.`

8