Functions. Callback and templates. Files.

Function pointers

In C/C++ languages, a function has an address, which is the memory address of its first instruction.

Therefore, this address can be used to initialize a pointer, and the function can be called using this pointer.

Function pointers

Declaration:

```
type (*pointer_name)(type1, type2,...,typeN) = fun;
```

type – the return type of the function;

type1, ..., typeN – the types of the function's arguments;

fun – the name or address of the function to which the pointer is declared.

When working with function pointers, it's convenient to introduce a type alias.

Function pointers

```
bool less(int arg1, int arg2){
    return arg1 <= arg2;
}

bool greater(int arg1, int arg2){
    return arg1 > arg2;
}


bool (*fun_ptr)(int, int) = greater;
bool res = fun_ptr(1, 2);                    false
```

Function pointers

When working with function pointers, it is convenient to enter an alias:

```
typedef bool (*CMP_FUN)(int, int);
```

Function pointers

```
void Sort(int* arr, int n, CMP_FUN cmpFun){
	...
	if(cmpFun(arr[i], arr[j])){
		...
	}
}

int main(){
	int arr[5] = {5, 2, 3, 4, 1};
	CMP_FUN cmp_fun = greater;
	Sort(arr, 5, cmp_fun);		Sort in ascending order

	cmp_fun = less;
	Sort(arr, 5, cmp_fun);		Sort in descending order
}
```

Function pointers

Array of function pointers:

```
bool greater(int A, int B){...}
bool less(int A, int B) {...}
bool equal(int A, int B) {...}

bool (*fun[])(int, int) = {
                    greater,
                    less,
                    equal
    };
```

Function pointers

```
bool (*fun[3])(int, int) = { 0 };

fun[0] = greater;
fun[1] = less;
fun[2] = equal;
```

Function pointers

```
typedef bool (*FUN_PTR)(int, int);
...

FUN_PTR fun[3] = { 0 };

fun[0] = greater;
fun[1] = less;
fun[2] = equal;
```

Function pointers

Calling functions from an array of function pointers:

```
for (int i = 0; i < 3; i++) {
    std::cout << fun[i](1, 2) << " ";
}
```

Callback

Callback (callback function) is the transfer of executable code as one of the parameters of other code.

In C/C++, function pointers are used to work with callback functions.

Callback

```
void Sort(int* arr, int n, CMP_FUN cmpFun)
```

cmpFun – callback function

Callback

Often callback functions are used to handle events:

```
DrObj->MajorFunction[IRP_MJ_READ] = DriverRead;
DrObj->MajorFunction[IRP_MJ_WRITE] = DriverWrite;
DrObj->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DriverDeviceControl;
```

Templates

A **template** is a feature of the C++ language designed for the development of algorithms whose operations are independent of the data types they work on.

The parameter of a template can be any type, including another template. The type must support the operations used by the template function.

Templates

Definition:

```
template<typename T>
type fun(type1 arg1, T arg2, ..., T argN);
```

Instead of the typename keyword, in most cases it is acceptable to use the class keyword.

Templates

There can be several template parameters:

```
template<typename T1, typename T2>
void fun(T1 arg1, T2 arg2)
{
    ...
}
```

Templates

```
template<typename T>
bool greater(T arg1, T arg2){
    return arg1 > arg2;
}

int main(){
    bool res1 = greater<int>(1, 2);
    bool res2 = greater<float>(0.1, 0.2);
}
```

The compiler will create two greater specializations: for int and for float.

Templates

Templates provide a short form for recording a section of source code.

But their use does not shorten the executable code, since for each set of parameters the compiler creates a separate instance of the function.

Templates

Integer constant expressions can be used as template parameters.

```
template<typename T, int N>
void Sort(T* arr, bool (*cmpFun)(T, T)){
      for(int i = 0; i < N; i++){
            ...
      }
}

int main(){
      float arr[5] = {0.1, 0.3, 0.2, 0.5, 0.8};
      Sort<float, 5>(arr, greater);
}
```

Templates

Template functions can have parameters that are only constant expressions:

```
template<int N> void createArray()
{
    int arr[N];
    for(int i = 0; i < N; i++)
        arr[i] = ...
}
```

Files.

Definitions

- A file is a named set of data located on an external storage device.
- A file pointer is the current position in the file (offset in bytes from the beginning of the file) from which the next read or write operation will be performed.
- A file path is a set of characters that indicates the location of a file or directory in the file system.

Path to file

When you specify only the file name, it is searched in the directory from which the program is launched.

To open a file from another directory, you can use absolute or relative paths.

Path to file

An absolute (full) path is a path that points to the same location in the file system, regardless of the current working directory or other circumstances. The full path always starts from the root directory.

For example,

“C:\\Windows\\System32\\notepad.exe”

Path to file

Relative path - a path relative to the current working directory (usually the directory from which the application is launched).

The relative path can be supplemented with symbols to go to the current or parent directory.

“..\\” – Go to parent directory

“.\\” – Go to current directory

Path to file

Let the current directory be “C:\\Windows\\System32”

“notepad.exe” – search for the notepad.exe file in the C:\Windows\System32 directory

“.\\setup\\cmmigr.dll” - search for the cmmigr.dll file in the directory C:\Windows\System32\setup

“..\\MEMORY.DMP” - search for the MEMORY.DMP file in the C:\Windows directory

“..\\SysWOW64\\notepad.exe” - search for the notepad.exe file in the C:\Windows\SysWOW64 directory

Operations with File

Basic operations with files:

- Creation/opening;
- Reading data from a file;
- Writing data to a file;
- Closing a file;
- Deleting a file.

Working with files in C-style

When working with C-style files, the `FILE` structure is used. This structure contains the information necessary to work with the file.

Working with files in C-style

Opening/creating a file:

```
FILE* fopen(char* filename, char* mode);
```

Opens a file named ***filename***. The file opening mode is specified by the ***mode*** parameter.

If successful, the function returns a pointer to the FILE structure. If an error occurs, a null pointer is returned.

Working with files in C-style

File opening modes:

“r”	The file is opened for reading. The file must exist.
“w”	An empty file is opened for writing (if the file exists, its contents are erased)
“a”	The file is opened to add information to the end. If the file does not exist, it is created.
“r+”	The file is opened for reading and writing. The file must exist.
“w+”	An empty file opens for reading and writing. If the file exists, its contents are erased.
“a+”	The file is opened for reading and writing. Writing is done at the end of the file. If the file does not exist, it is created.
“t”	The file opens in text mode.
“b”	The file opens in binary mode.

Working with files in C-style

By default, files open in text mode.

File opening modes can be combined with each other.

For example:

“r+b”, “rb+”, “wb+”

Working with files in C-style

To set a new file pointer value, use the function **fseek**:

```
int fseek(FILE* file, long int ofs, int orig);
```

The function sets the current position in the file file to the value **ofs** relative to **orig**.

If executed successfully, the function returns a zero value, otherwise it returns a non-zero value.

Working with files in C-style

The offset is specified relative to the reference value **orig**, which can have the following values:

SEEK_SET	Start of file
SEEK_CUR	Current position in the file
SEEK_END	End of file

Working with files in C-style

To get the current value of the file pointer, use the function **ftell**:

```
long int ftell(FILE* file);
```

The function returns the current file pointer value for a file ***file***.

Working with files in C-style

Basic functions for writing data to a file:

- fputc;
- fwrite;
- fputs.

Working with files in C-style

```
size_t fwrite(void* ptr, size_t size,
              size_t count,  FILE* file);
```

The function writes count elements of size **size** from the **ptr** buffer to the file **file**.

The function returns the number of elements successfully written.

The function increments the file pointer value by the amount of bytes written.

Working with files in C-style

```
int fputc(int ch, FILE* file);
```

The function writes the ch character to file and increments the file pointer.

In case of successful writing, the code of the symbol written to the file is returned. In case of error - special value EOF (end of file).

Working with files in C-style

```
int fputs(const char* str, FILE* file);
```

The function writes characters from the string str to the file file until it encounters a line end character ('\0'). The end of line character is not written to the file. The file pointer is incremented by the length of the line.

The function does not add a line break ('\n') to the file after writing str.

If successful, a non-negative value is returned. In case of error – EOF.

Working with files in C-style

```
FILE* file = fopen(“test.txt”, “w”);
char str[7] = “abcd”;
fputs(“qwerty”, file);
fputc(‘A’, file);
fwrite(str, 1, 7, file);
fclose(file);
```

File contents
qwertyAabcd\0\0\0

Working with files in C-style

Write to an arbitrary location in a file :

```
FILE* file = fopen(“test.txt”, “w”);
char str[7] = “abcd”;
fputs(“qwerty”, file);
fputc(‘A’, file);
fseek(file, 1, SEEK_SET);
fwrite(str, 1, 4, file);
fclose(file);
```

File contents
qabcdyA

Working with files in C-style

Writing binary data to a file :

```
FILE* file = fopen(“test.txt”, “w”);
char str[7] = “abcd”;
fwrite(str, 1, 4, file);
int val = 0x33445566;
fwrite(&val, 4, 1, file);
fclose(file);
```

File contents
abcdfUD3

Code	Character
0x33	‘3’
0x44	‘D’
0x55	‘U’
0x66	‘f’

Working with files in C-style

Basic functions for reading data from a file :

- fread;
- fgetc;
- fgets.

Working with files in C-style

```
size_t fread(void* ptr, size_t size,
                    size_t count, FILE* file);
```

The function reads **count** elements of size **size** from the file **file** into the **ptr** buffer. Memory for the buffer must be allocated.

The function returns the number of elements successfully read.

The function increments the file pointer value by the number of bytes read.

Working with files in C-style

```
int fgetc(FILE* file);
```

The function reads a character from file and increments the **file** pointer.

If successful, the function returns the code of the symbol read from the file. In case of an error or reaching the end of the file, a special value EOF (end of file).

Working with files in C-style

```
char* fgets(char* str, int num, FILE* file);
```

The function reads characters from file file and stores them in the C-style string str until it counts num-1 characters, encounters a newline character ('\n'), a line end character ('\0'), or will not reach the end of the file, whichever happens first.

If successful, the function returns a pointer to the read string (str), if unsuccessful, a null pointer.

Working with files in C-style

```
FILE* file = fopen(“test.txt”, “r”);
char str[10] = { 0 };
fgets(str, 10, file);
char ch = fgetc(file);

int val = 0;
fseek(file, -sizeof(int), SEEK_END);
fread(&val, 4, 1, file);
fclose(file);
```

str = a qwerty\n

ch = A

val = 0x33445566

File contents
a qwerty\n
AabcdfUD3

Working with files in C-style

```
FILE* file = fopen(“test.txt”, “r”);

char* res = NULL;
do {
    char buff[50] = { 0 };
    res = fgets(buff, 50, file);
    std::cout << buff << std::endl;
} while (res != NULL);
fclose(file);
```

File contents
qwerty\r\n
ABC\r\n
KLMN\r\n

```
qwerty
ABC
KLMN
```

Working with files in C-style

You can check whether the end of the file has been reached using the **feof** function. The function returns a non-zero value when the end of the file is reached. If the end of the file is not reached, then zero is returned.

```
while(!feof(file)){
    ...
}
```

Working with files in C-style

```
FILE* file = fopen(“test.txt”, “r”);

while(!feof(file)){
    char buff[50] = { 0 };
    fgets(buff, 50, file);
    std::cout << buff << std::endl;
}
fclose(file);
```

File contents
qwerty\r\n
ABC\r\n
KLMN\r\n

```
qwerty
ABC
KLMN
```

Working with files in C-style

File size calculation:

You can calculate the file size using the **fseek** and **ftell** functions:

```
fseek(file, 0, SEEK_END);
size_t file_size = ftell(file);
```

Working with files in C-style

Closing the file:

After finishing working with the file, you need to close it by calling the function **fclose**:

```
FILE* file = fopen(“test.txt”, “r”);
...
fclose(file);
```

Until the file is closed, write and delete operations on the file will be blocked.

Working with files in C++ style

Working with files in C++ is carried out using the file stream classes std::fstream and specialized classes std::ifstream, which implements a stream for reading data from a file and std::ofstream for writing data to a file.

To use these classes, you must include the fstream header file.

```
#include <fstream>
```

Working with files in C++ style

A file is opened when a file stream object is created by calling the constructor or after the object is created by calling the **open** method.

```
std::fstream file (“test.txt”, std::fstream::out |
std::fstream::trunc);
```

```
std::fstream file;
file.open(“test.txt”, std::fstream::out |
std::fstream::trunc);
```

Working with files in C++ style

File opening modes are specified by flags in the second parameter of the constructor or **open** method.

These modes can be combined using the operation | (bitwise OR).

Working with files in C++ style

File opening modes

in	The file is opened for reading. The file must exist.
out	The file is opened for writing. If the file exists, its contents are erased. If the file does not exist, it is created.
binary	The file opens in binary mode.
ate	Once a file is created/opened, the file pointer points to the end of the file.
app	The file is opened to add information to the end. If the file does not exist, it is created.
trunc	If the file exists, its contents are erased.

Working with files in C++ style

```
file.open(“test.txt”, std::fstream::in | std::fstream::out |
std::fstream::ate);
```

The file must exist. The file is opened for reading and writing, the contents of the file are not deleted. The file pointer points to the end of the file. The position of the file pointer can be changed.

```
file.open(“test.txt”, std::fstream::in | std::fstream::out |
std::fstream::app);
```

If the file does not exist, it is created. The file is opened for reading and writing, the contents of the file are not deleted. The file pointer for write operations always points to the end of the file. The file pointer position for a record cannot be changed.

```
file.open(“test.txt”, std::fstream::in | std::fstream::out);
```

The file must exist. The file is opened for reading and writing, the contents of the file are not deleted. The file pointer points to the beginning of the file. The position of the file pointer can be changed.

Working with files in C++ style

You can check whether a file stream object has been created for a file with the specified attributes using the **is_open** method.

The method returns `true` if a file stream object has been created and is associated with the specified file. If unsuccessful, the method returns `false`.

Working with files in C++ style

```
std::fstream file(“test.txt”,
std::fstream::in);

if(file.is_open())
    std::cout << “OK” << std::endl;
else
    std::cout << “Error” << std::endl;
file.close();
```

Working with files in C++ style

You can set the file pointer value using the **seekg** and **seekp** methods.

seekg(offset, orig)
seekp(offset, orig)

The **seekg** method is used to set the file pointer for read operations, and **seekp** is used for write operations.

The methods set the file pointer value to offset relative to orig.

Working with files in C++ style

The offset is specified relative to the reference value orig, which can have the following values:

beg	Start of file
cur	Current position in the file
end	End of file

Working with files in C++ style

To obtain the current value of the file pointer, the tellg and tellp methods are used, respectively.

Working with files in C++ style

Writing to a file is done using:

- Method write;
- Operator <<;
- put method.

Working with files in C++ style

```
write(const char* buff, size_t size)
```

Writes size bytes from the memory region pointed to by buff to a file. The method increments the file pointer by the number of bytes written.

Working with files in C++ style

```
std::fstream file(“test.txt”, std::fstream::out);
std::string str = “qwerty”;
char str_c[10] = “ABC”;

file.write(str.c_str(), str.length());
file.write(str_c, 3);
file.write(“KLMN”, 4);
file.close();
```

File contents
qwertyABCKLMN

Working with files in C++ style

Some types of data can be written to a file using the stream write operator <<.

In particular, this operator is overloaded for all standard (built-in) data types, C-style and C++-style strings. The file pointer is incremented by the number of bytes written.

In this case, as for I/O streams, you can use manipulators std::endl, std::hex, etc.

When writing numbers using the << operator, their string representation is written to the file. As a consequence, the line is written until the end of line character (‘\0’). The end of line character is not written to the file.

Working with files in C++ style

```
std::fstream file(“test.txt”, std::fstream::out);
std::string str = “qwerty”;
char str_c[10] = “ABC DEF”;
int val = 25;

file << val << “ ” << std::hex
<< 14 << std::endl;
file << str << std::endl;
file << str_c << std::endl;
file << “KLMN\0OP” << std::endl;
file.close();
```

File contents
25 e\r\n
qwerty\r\n
ABC DEF\r\n
KLMN\r\n

Working with files in C++ style

```
put(char ch)
```

The **put** method places the **ch** character into a file and increments the file pointer.

Working with files in C++ style

```
std::fstream file(“test.txt”, std::fstream::out);
std::string str = “qwerty”;
int val = 0x33445566;

file << std::hex << val << “ ” <<
std::dec << 14 << std::endl;
file << str << std::endl;
file.put(‘A’);
file.put(‘\n’);
file.write((char *)&val, sizeof(int));
file.close();
```

File contents
33445566 14\r\n qwerty\r\n A\r\n fUD3

Working with files in C++ style

Reading from a file is done using:

- Method read;
- Operator >>;
- getline method;
- get method.

Working with files in C++ style

```
read(char* buff, size_t size)
```

The method reads size bytes from the file into the memory region pointed to by buff. The memory region pointed to by buff must be allocated.

If there are less than size bytes left until the end of the file, then the number of bytes remaining until the end of the file will be read into the buff buffer.

The file pointer is incremented by the number of bytes read.

Working with files in C++ style

```
std::fstream file("test.txt", std::fstream::in);
char buff[7] = { 0 };

char* dyn_buff = new char[7];
memset(dyn_buff, 0, 10);

file.read(buff, 6);
file.read(dyn_buff, 10);
file.close();
delete[] dyn_buff;
```

File contents
qwertyABC

```
buff = qwerty
dyn_buff = ABC
```

Working with files in C++ style

Some data types can be read from a file using the stream read operator >>.

In particular, this operator is overloaded for all standard (built-in) data types, C-style and C++-style strings. The file pointer is incremented by the number of bytes read.

Reading occurs until the nearest space, line feed ('\n'), carriage return ('\r'), end of line ('\0'), or end of file.

Working with files in C++ style

```
std::fstream file(“test.txt”, std::fstream::in);
std::string str;
char str_c[10] = { 0 };
char* dyn_str = new char[10];
memset(dyn_str, 0, 10);
int val1 = 0, val2 = 0;

file >> val1;
file >> std::hex >> val2;
file >> str;
file >> str_c;
file >> dyn_str;
file.close();
```

File contents
25 e\r\n qwerty\r\n ABC KLMN\r\n

```
val1 = 25 val2 = 14
str = qwerty
str_c = ABC
dyn_str = KLMN
```

Working with files in C++ style

```
getline(char* str, size_t num)
getline(char* str, size_t num, char delim)
```

The method reads characters from the file and stores them in the memory region pointed to by str until it encounters a line break character ('\n'), a line end character ('\0'), or until num bytes have been read or reached end of file, whichever happens first.

When a delim delimiter is specified, reading occurs until the delimiter is encountered, a line end character is encountered, num bytes are read, or the end of the file is reached, whichever occurs first.

The file pointer is changed to the number of bytes read.

Working with files in C++ style

```
std::fstream file("test.txt", std::fstream::in);

while(!file.eof()){
    char buff[50] = { 0 };
    file.getline(buff, 50);
    std::cout << buff << std::endl;
}

file.close();
```

File contents
25 e\r\n qwerty\r\n ABC KLMN\r\n

```
25 e
qwerty
ABC KLMN
```

Working with files in C++ style

The get method allows you to get a symbol from a file. The method increments the file pointer.

```
std::fstream file(“test.txt”, std::fstream::in);

char ch = file.get();  ch =‘2’

file.close();
```

File contents
25 e\r\n qwerty\r\n ABC KLMN\r\n

Working with files in C++ style

You can check whether the end of the file has been reached using the eof method. The method returns true when the end of the file is reached, otherwise false.

```
while(!file.eof()){
    ...
}
```

Working with files in C++ style

After finishing working with the file, it must be closed by calling the **close** method.

```
file.close();
```

Until the file is closed, write and delete operations on the file will be blocked.

Working with files in C++ style

```
std::fstream file("test.txt", std::fstream::in);
std::string str;
int val1 = 0, val2 = 0, val3 = 0;
char ch = 0;

file >> std::hex >> val1;    val1 = 0x33445566
file >> std::dec >> val2;    val2 = 14
file >> str;                 str = qwerty
file.seekg(2, std::fstream::cur);
ch = file.get();             ch = 'A'
file.seekg(-(int)(sizeof(int)), std::fstream::end);
file.read((char *)&val3, sizeof(int));        val3 = 0x33445566
file.close();
```

File contents
33445566 14\r\n qwerty\r\n A\r\n fUD3

Working with files in C++ style

To remove a file, you can use the remove (for C) / std::remove (for C++) function from the stdio.h (for C) / cstdio (for C++) header file.

Working with files in C++ style

```
int remove(const char* filename)
```

The function deletes a file named filename.

If successful, the function returns a zero value, if an error occurs, a non-zero value.