Declaration of variables in C++

```
void fun() // in C: void fun(void)
{
……………………………………..
 int n = strlen(name) + 1;
// the declaration may be anywhere before the first usage of the variable
// the initial value may be any expression that can be evaluated at the moment
// of declaration
for (int i = 0;  i < n;  i++)
………………………………..
}

double y = sqrt(double x = 5.6); // error
double x = 5.6, y = sqrt(x); // correct
```

Scope resolution operator

```
int iii;      // global
void fun()
{
  int iii;    // local in fun()
  iii = 5;   // to use the global iii write ::iii = 5
  while (1)
 {
    int iii;  // local in while block
    iii = 0; // to use the global iii write ::(::iii) = 0
              // to use the local iii write ::iii = 0
   ………….
 }
 iii = 10;   // to use the global iii write ::iii = 10
 ……..
}
```

Default arguments

```
void fun1(double, double, int = 0); // prototype
void fun1(double d1, double d2, int ii) // definition
{
…………………………………………..
}
fun1(5.0, 6.0, 1); // call to fun1(), d1 = 5.0, d2 = 6.0, ii = 1
fun1(5.0, 6.0);    // call to fun1(), d1 = 5.0, d2 = 6.0, ii = 0

void fun2(double = 0, double = 0, int = 0); // prototype
void fun2(double d1, double d2, int ii) // definition
{
…………………………………………..
}
fun2(); // call to fun2(), d1 = 0, d2 = 0, ii = 0
fun2(5.0);    // call to fun2(), d1 = 5.0, d2 = 0, ii = 0
fun2(5.0, 6.0);    // call to fun2(), d1 = 5.0, d2 = 6.0, ii = 0
fun2(5.0, 6.0, 1); // call to fun2(), d1 = 5.0, d2 = 6.0, ii = 1

void fun(double, double = 0, int); // error
```

Function overloading

```
void fun(int, int);  // 1
void fun(double, double); // 2
void fun(); // 3

fun(5, 6); // 1
fun(5.0, 6.0); // 2
fun(); // 3

fun(5, 6.0);   // error, the compiler cannot select between functions
```

fun(float);	fun(125.6F);
fun(double);	fun(125.6);
fun(long double);	fun(125.6L);
fun(int);	fun(125);
fun(unsigned int);	fun(125U);
fun(long int);	fun(125L);
fun(unsigned long int);	fun(125LU);

Inline functions and macros

```
double square(double); // prototype
inline double square(double d) // definition
{
  return d *d;
 }
 double x = square(y); // may be compiled also as x = y * y;

#define SQUARE(a)              (a) * (a)
double x = SQUARE(y); // the preprocessor replaces with x = (y) * (y)
```

The parenthesis in the body of macro are absolutely necessary:

```
double x = SQUARE(y + 1); // we get x = (y + 1) * (y + 1)
```

But if we define the macro as

```
#define SQUARE(a)              a * a
```

we get x = y +1 * y + 1 or actually we get x = 2 * y + 1

References

A reference is an alias for another variable. Below, "ri" is the alias of "i".

```
int i, *pi = &i, &ri = i;
```

All the following expressions do the same: write value “10” to the four-byte field having now two names: "i" and "ri":

```
i = 10;
*pi = 10;
ri = 10;
```

`void swap(int &, int &);`
`void swap(int &rx, int &ry)`
`{`
`  int z = rx; rx = ry;  ry = z;`
`}`
	`void swap(int *, int *);`
	`void swap(int *px, int *py)`
	`{`
	`  int z = *px; *px = *py; *py = z;`
	`}`

Call by reference:
`int a = 5, b = 6;`
`swap(a, b);`
Actually, swap works with "a" and "b", temporarily named also as "rx" and "ry"
	Call by value:
	`int a = 5, b = 6;`
	`swap(&a, &b);`
	swap accesses "a" and "b" undirectly, using pointers "px" and "py"

Modifier const (1)

```
const double d = 5.5; // any attempt to change the value of d causes an error
const char *pName = "Mister X";
const char* pAnimals[] = { "dog", "cat", "cow", "horse"};

// any attempt to change one of the strings presented above causes an  error.
// for example:
*(pName + 7) = 'Y'; // error message will follow
strcpy(pAnimals[0], "rat"); // error message will follow
```

Tricks that do not work:

```
double *p1 = &d; // error message will follow
*p1 = 6.5;

const double *p2 = &d;
*p2 = 6.5;  // error message will follow

char *pc = pAnimals[0]; // error message will follow
strcpy(pc, "rat");
```

Modifier const (2)

```
const char *pName = "Mister X";
```

pName is the pointer to constant object. Changing this object is not possible, but pName may set to point to another object, for example

```
pName = "Missis X"; // correct
*(pName + 7) = 'Y';  // error
```

```
char * const pName = "Mister X";
```

pName is the constant pointer to an object. Changing the pointer is not possible, but the object is not protected, for example

```
pName = "Missis X"; // error
*(pName + 7) = 'Y';  // correct
```

```
const char * const pName = "Mister X";
```

pName is the constant pointer to a constant object. Neither the pointer nor the object can be changed.

```
pName = "Missis X"; // error
*(pName + 7) = 'Y';  // error
```

Memory allocation

```
double **CreateMatrix(int n, int m)
{
  double **pMatrix = new double *[n];
                // pMatrix  = (double **)malloc(n * sizeof(double *));
   for (int i = 0;  i < n;  i++)
        *(pMatrix + i) = new double[m];
                // *(pMatrix + i) = (double *)malloc(m * sizeof(double));
   return pMatrix;
}

void DestroyMatrix(double **pMatrix)
{
   for (int i = 0;  i < n;  i++)
        delete *(pMatrix + i) ;    // free(*(pMatrix + i) );
   delete pMatrix; // free(pMatrix);
 }
```

Variable number of arguments

```
#include "stdarg.h"
double mean(int n, double x1,…)
{ // n – total number of double parameters
   // x1 – the first double parameter
  va_list indic;
  va_start(indic, x1);
   double sum = x1;
   for (int i = 1;  i < n;  i++)
        sum += va_arg(indic, double);
    return sum / n;
}
```

Usage:

```
double a, u, v, w, x, y, z;
a = mean(6, u, v, w, x, y, z);
```

Long jump

```
#include "setjmp.h"
jmp_buf env; // global variable, stores the current execution environment
int main()
{
    switch (setjmp(env)) // return point
   {
     case 0: StartOperations(); // on the first call the setjmp return value is 0
                break;
     case 1: printf("Success\n");
                break;
     case 2: printf("Failure\n");
                return 1;
     }
………………………………….
}
```

Somewhere in operations:

```
longjmp(env, 2);  // in case of failure jumps to the return point, setjmp returns 2
longjmp(env, 1);  // at the end of operations jumps to the return point, setjmp returns 1
```

Unicode in Windows C++ (1)

ASCII : one byte per character, max 256 different characters (0…255).

Unicode: international standard, each character has its own unique code point marked as U + xxxx (xxxx is a hexadecimal number).
The latest version 10.0, 136,765 characters including historic scripts (for example, the Egyptian hieroglyphs) and multiple symbol sets. Max 1,114,112 characters are possible.

Unicode is implemented by different encodings: UCS-2, UTF-16, UTF-32, UTF-8.
UCS - Universal Character Set, UTF - Unicode Transformation Format.

UCS-2: 2 bytes per character. The first 256 characters match the ASCII codes. Max 65,536 bytes.

UTF-16 (Windows, Java): 2 or 4 bytes per character. Mostly 2 bytes as in UCS-2. If the contents of two-byte field is from interval [0xD800 : 0xD8FF], the code occupies also the next two bytes. The four-byte codes (called as surrogates) are met very seldom.

UTF-32 (Linux, Unix): 4 bytes per character

Unicode in Windows C++ (2)

UTF-8 (web): 1, 2, 3 or 4 bytes per character. If the highest bit of the byte is 0, the code occupies 1 byte, otherwise we have to check the next byte. If the highest bit of the second byte is 0, the code occupies 2 bytes and so on.

Big endian: the most significant byte is the first. For example, 'Z' in UTF-16 big endian mode is coded as { 0x00, 0x7A }.
Little endian: the less significant byte is the first. For example, 'Z' in UTF-16 little endian mode is coded as {0x7A, 0x00 }.

In C++ new fundamental type wchar_t (wide character). The standard does not state how many bytes a wchar_t variable should occupy and which encoding system must be used.

ASCII	Unicode
char c = 'A';	wchar_t wc = L'A';
char *pc = "ABC"; String in ASCII is terminated with 0.	wchar_t *pwc = L"ABC"; String in Unicode is terminated by two 0-s

Unicode in Windows C++ (3)

ASCII	Unicode
strcpy	wcscpy
strcpy_s	wcscpy_s
strcmp	wcscmp
strcat	wcscat
strcat_s	wcscat_s
strchr	wcschr
strlen	wcslen
printf	wprintf

Generic programming: instead of char, wchar_t use TCHAR.
For example:

```
TCHAR c = _T('A'), *p1 = new TCHAR[100],
                    *p2 = _T("string");
```

_T is a macro defined in file tchar.h

Instead of strcpy, wcscpy, etc. use macros like _tcscpy, _tcscmp, _tcscat, _tcschr, _tcslen, _tprintf, etc.
For example:

```
_tcscpy(Buf, _T("Hello"));
_tprintf(_T("%d\n"), _tcslen(Buf));
```

Those macros are also defined on tchar.h

Unicode in Windows C++ (4)

Generic programming: the Visual Studio wizard generates stdafx.h. The *.cpp files should start with

```
#include "stdafx.h" // includes also file tchar.h
#include "Windows.h" // defines _UNICODE
```

Macros _T, _tcscpy, _tcscmp, _tcscat, _tcschr, _tcslen, _tprintf etc. are defined in the following way:

```
#ifdef _UNICODE
#define _tcslen wcslen
typedef wchar_t TCHAR
........................................
#else
#define _tcslen strlen
typedef char TCHAR
.........................................
#endif
```

Windows data types

```
typedef char                    CHAR;
typedef unsigned char           BYTE;
typedef int                     INT;
typedef int                     BOOL;
typedef unsigned int            UINT;
typedef short int               SHORT;
typedef unsigned short int      WORD;
typedef long int                LONG;
typedef unsigned long int       DWORD;
typedef unsigned long int *     LPDWORD
typedef const wchar_t *         LPCTSTR // #ifdef _UNICODE
typedef const char *            LPCTSTR // #ifndef _UNICODE
typedef void *                  PVOID
#define FALSE                   0
#define TRUE                    1
```

Error handling

In WinError.h:

```
#define ERROR_SUCCESS                0
#define ERROR_FILE_NOT_FOUND         2
#define ERROR_ACCESS_DENIED          5
DWORD ErrorCode = GetLastError();
SetLastError(ErrorCode);
```

http://msdn.microsoft.com/en-us/library/windows/apps/ms681381(v=vs.85).aspx

Error code structure:

Bits	Meaning
[31 : 30]	00 – success, 01 – success with additional information, 10 – success with warning, 11 - failure
29	Microsoft-defined : 0; user-defined : 1
28	Always 0
[27 : 16]	Facility code. [0 : 255] reserved by Microsoft
[15 : 0]	Error number

Windows files (1)

A file system specifies how to locate files on disk. It specifies also how to create, copy, delete, name, protect, etc. the files. In addition, the file system provides software for all those operations.
File systems: FAT (file allocation table), FAT32, NTFS (new technology file system) on hard disks, CDFS (CD file system), UDF (universal disk format) on DVD, specific file systems for servers.

```
HANDLE hFile = CreateFile(file_name_and_path, // TCHAR *
GENERIC_READ | GENERIC_WRITE, // or only one of them (desired access)
0, NULL,
CREATE_ALWAYS, // options: CREATE_NEW, OPEN_EXISTING,
// OPEN_ALWAYS, TRUNCATE_EXISTING (creation disposition), etc.
FILE_ATTRIBUTE_NORMAL, NULL);
if (hFile == INVALID_HANDLE_VALUE)
    _tprintf(_T("File not created, error %d"), GetLastError());

CloseHandle(hFile);
```

https://msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx

Windows files (2)

```
DWORD nBytesToWrite, nBytesWritten = 0, nBytesToRead, nReadBytes = 0;
BYTE *pBuffer;
HANDLE hFile;
int Result;

Result = WriteFile(hFile, pBuffer, nBytesToWrite, &nBytesWritten, NULL);
Result = ReadFile(hFile, pBuffer, nBytesToRead, &nReadBytes, NULL);
```

If errors, Result is FALSE, GetLastError() returns the error code

```
DWORD nBytesInFile = GetFileSize(hFile, NULL);
```

If errors, nBytesInFile is INVALID_FILE_SIZE, GetLastError() returns the error code

```
LONG DistanceToMove;
DWORD NewPosition = SetFilePointer(hFile, DistanceToMove, NULL,
                FILE_BEGIN); // options: FILE_CURRENT, FILE_END
```

If errors, NewPosition is INVALID_SET_FILE_POINTER, GetLastError() returns the error code. To get the file pointer current position, write:

```
DWORD CurrentPosition = SetFilePointer(hFile, 0, NULL, FILE_CURRENT);
```

File handling in Windows: example

```
#include "stdafx.h"
#include "Windows.h"

int _tmain(int argc, _TCHAR* argv[])
{
        HANDLE hFile = CreateFile(_T("FileExample.bin"),
                GENERIC_READ |  GENERIC_WRITE, 0, NULL,
                CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
     if (hFile == INVALID_HANDLE_VALUE)
    {
        _tprintf(_T("Unable to create file, error %d\n"), GetLastError());
        return 1;
    }
    int *pData1 = new int[10], i;
    DWORD nWritten;
    for (i = 0;  i < 10;  i++)
        *(pData1 + i) = i;
    if (!WriteFile(hFile, pData1, 10 * sizeof(int), &nWritten, NULL))
    {
        _tprintf(_T("Unable to write into file, error %d\n"), GetLastError());
        return 1;
    }
    if (nWritten != 10 * sizeof(int))
        _tprintf(_T("Only %d bytes were written\n"), nWritten);
    int *pData2 = new int[10];
    DWORD nRead;
    if (SetFilePointer(hFile, 0, NULL, FILE_BEGIN) == INVALID_SET_FILE_POINTER)
    {
        _tprintf(_T("Unable to set the file pointer, error %d\n"), GetLastError());
        return 1;
    }
    if (!ReadFile(hFile, pData2, nWritten, &nRead, NULL))
    {
        _tprintf(_T("Unable to read from file, error %d\n"), GetLastError());
        return 1;
    }
    for (i = 0;  i < 10;  i++)
        _tprintf(_T("%d\n"), *(pData2 + i));
    CloseHandle(hFile);
    return 0;
}
```

Console I/O

```
HANDLE hConsole = GetStdHandle(
            STD_INPUT_HANDLE); // stdin, the options are
                        // STD_OUTPUT_HANDLE, (stdout)
                        // STD_ERROR_HANDLE (stderr)
```

hConsole is actually a file handle, may be used in WriteFile(), ReadFile(), etc.

Keyboard input example:

```
SetConsoleMode(hConsole,
            ENABLE_LINE_INPUT |     // ENTER marks the end of input
            ENABLE_ECHO_INPUT |    // show typed characters
            ENABLE_PROCESS_INPUT | // may use BACKSPACE for correcting
            ENABLE_INSERT_MODE);  // may insert omitted characters
TCHAR Buf[81];
DWORD nRead;
ReadConsole(hConsole, Buf, 80, &nRead, NULL);
Buf[nRead - 2] = 0; // the last to characters in Buf are \r\n, but we need a regular
                    // zero-terminated string
```

Process

A process is an instance of a running application.
Keywords: multitasking, time slice or quantum, address space, graphical user interface (GUI), console user interface (CUI).

Wizard-created main functions for CUI-based applications:

```
int _tmain(int argc, TCHAR *argv[])
int _tmain(int argc, TCHAR *argv[], TCHAR *envp[])
```

Example: if the command line is

```
MyApp "c:\My data\Data1.bin" 25
```

then argc is 3, argv[1] is `_T("c:\\My data\\Data1.bin")` and argv[2] is `_T("25").`
argv[0] presents the application .exe file including the full pathname.

```
TCHAR *p = GetCommandLine();
```

Exiting:

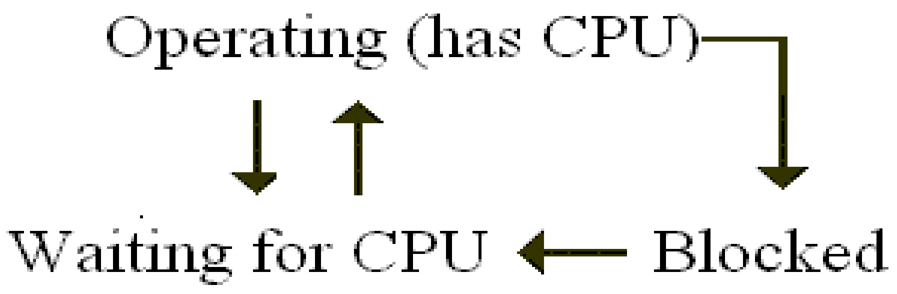
a) The main function returns with exit code
b) Call to function exit(*exit_code*) or to function _exit(*exit_code*);

Exit code: traditionally 0 means "no errors" and 1, 2 etc. are error codes.

Multithreaded applications (1)

A thread of execution is the smallest sequence of programmed instructions that an operating system can manage independently.
Multitasking: several processes run concurrently (or seem to run concurrently).
Between processes the computer resources (like memory, ports, etc.) are not shared,
For example each process has its own address space.
Multithreading: several threads run concurrently (or seem to run concurrently) within one process. Several threads can share the same resource, for example a section of memory.
On single processor systems the processor switches from one thread to another. When the time slice allocated for a thread has exipred, the processor starts to execute instructions from another thread. As the time slices are short, the user percieves that the threads are running on the same time. On multiprocessor systems each processor may run a particular thread, i.e. the threads are actually running on the same time.
A thread may be in three states:

When a thread needs a resource currently occupied by another thread or waits for data from external sources, it is in the blocked state. The other threads continue to work.



Multithreaded applications (2)

When you must create threads:
1. You have some subtasks that take a large amount of time (searching, printing, etc.)
2. You need to communicate with external data sources (web, devices connected to your computer, etc.) which may send data to you or accept data you want to send them at occasional moments. It may even happen that they do not want to communicate with you. A single-thread application waiting for data from external source freezes and becomes uncontrolable.
3. You have subtasks with different priority.
4. You have user interface (CUI or GUI) which must be responsive - i.e. react to the user actions (for example mouse clicks) immediately.

Instruction starting from _tmain() belong to the primary (or main) thread. The main thread may initialize and launch another threads, those in turn can also have its own child threads and so on.

In Windows there are two types of threads:
1. Worker threads (for CUI applications, also for longer background subtasks in GUI applications).
2. User interface threads (for GUI applications).

Multithreaded applications (3)

Each thread must have its thread entry point function which corresponds to the _tmain() of the primary thread.
Each thread has its own stack for local variables. The global variables are common to all the threads.

Prototype for the thread entry point function:
unsigned int __stdcall *entry_point_function_name*(void *param);
// __stdcall - calling convention
// return value - as exit code for main()
// function name - no restrictions

To launch the thread:
HANDLE hThread = _beginthreadex(*NULL*, *0*, pointer_to_entry_point_function, param, *0*, *NULL*);

When the thread has stopped running:
CloseHandle(hThread);
Do not use function CreateThread()

Multithreaded applications (4)

We have: `int SearchRecord(TCHAR *pFilename, TCHAR *pID);`

Return value: 0 – not found, 1 – found, 2 – problems. Have to wrap it into a regular thread entry point function.

Declare `struct ThreadPar { TCHAR *pFileName, *pID;  int Result; };`

Thread entry point function:

```
unsigned int __stdcall ThreadFun(void *param)
{ struct ThreadPar *p = (struct ThreadPar *)param;
   p->Result = SearchRecord(p->pFileName, p->pID);
   return 0; }
```

Launching thread:

```
Struct ThreadPar SearchParam;
SearchParam.pFileName = _T("c:\\Data\\Base.bin");
SearchParam.pID = _T("Jaan Tamm");
HANDLE hThread = _beginthreadex(NULL, 0, ThreadFun,
(void *)&SearchParam, 0, NULL);
```

Instead of creating struct ThreadPar we may introduce two global pointers: one for filename and the other for ID. In that case:

```
HANDLE hThread = _beginthreadex(NULL, 0, ThreadFun, NULL, 0, NULL);
```

Thread management

_endthreadex(*exit_code*);
// use as exit(exit_code) in processes
// be careful: does not release resources
TerminateThread(*thread_handle*, *exit_code*); // better not to use

SuspendThread(*thread_handle*);
ResumeThread(*thread_handle*);
Sleep(*number_of_milliseconds*);

SetThreadPriority(*thread_handle*, *priority_level*);
THREAD_PRIORITY_TIME_CRITICAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_IDLE
……………………………………………..
thread_handle = GetCurrentThread();

HANDLE hThread = beginthreadex(*NULL*, *0*, pointer_to_entry_point_function, param, CREATE_SUSPENDED, *NULL*); // use ResumeThread() to launch

Thread synchronization problem

- When one of the threads owns a resource (array, linked list, disk file, COM port, etc.), the other threads can access this resource for reading only. The have no right to change anything on that resource.
- If a thread needs a resource owned by another thread for writing (i.e. changing data), it must wait until the resource is released.
- When the thread owning a resource does not need it more, it has to release the resource and signal about it to the other threads.

Key problems: communication between threads, locking resources.

The code must be thread-safe.

Atomic operations

int x = 1;
In the first thread x++;
In the second thread printf("x = %d\n", x);
Incrementing steps:

a) Read x, send into the register
b) Increment
c) Move x back into memory, write

Incrementing may be interrupted after step (a) or after step (b) or after step (c).
Consequently the second thread may print x = 1 or x = 2.

An atomic operation is a machine instruction that is always executed without interruption. A sequence of two or more machine instructions isn't atomic since the operating system may suspend the execution of the current sequence of operations in favour of another task. A C++ statement is atomic if the compiler translates it into a single machine instruction. But each hardware architecture may translate the same C++ statement in its own different way. So it is wise to say that none of the C++ is statements is atomic.

Interlocked family of functions

long int volatile x = 0; // must be volatile

The volatile keyword warns the compiler that a field might be modified by multiple threads that are executing at the same time. Fields that are declared volatile are not subject to compiler optimizations. This ensures that the most up-to-date value is present in the field at all times.

```
long int y;
x += 10; // not atomic
y = InterlockedExchangeAdd(&x, 10); // atomic,  now x is 10, y is 0
```

Prototype: long int InterlockedExchangeAdd(volatile long int *, long int);

```
BOOL volatile x = TRUE;
BOOL y;
x = FALSE; // not atomic
y = InterlockedExchange((long int volatile *)&x, FALSE); // atomic
// now y is TRUE, x is FALSE
```

Prototype: long int InterlockedExchange (volatile long int *, long int);

About the other interlocked functions see **https://msdn.microsoft.com/en-us/library/aa911383.aspx**

Volatile variables and polling: example

```
#include "stdafx.h"
#include "windows.h"
#include "stdio.h"
#include "process.h"
#include "conio.h"
unsigned int __stdcall KeyboardThread(void *);
volatile BOOL bStop = FALSE;

int _tmain(int argc, _TCHAR* argv[])
{
 _beginthreadex(NULL, 0, KeyboardThread,
                    NULL, 0, NULL);
 while (!bStop)
     _tprintf(_T("Working, ESC to exit\n"));
 return 0;
}
```

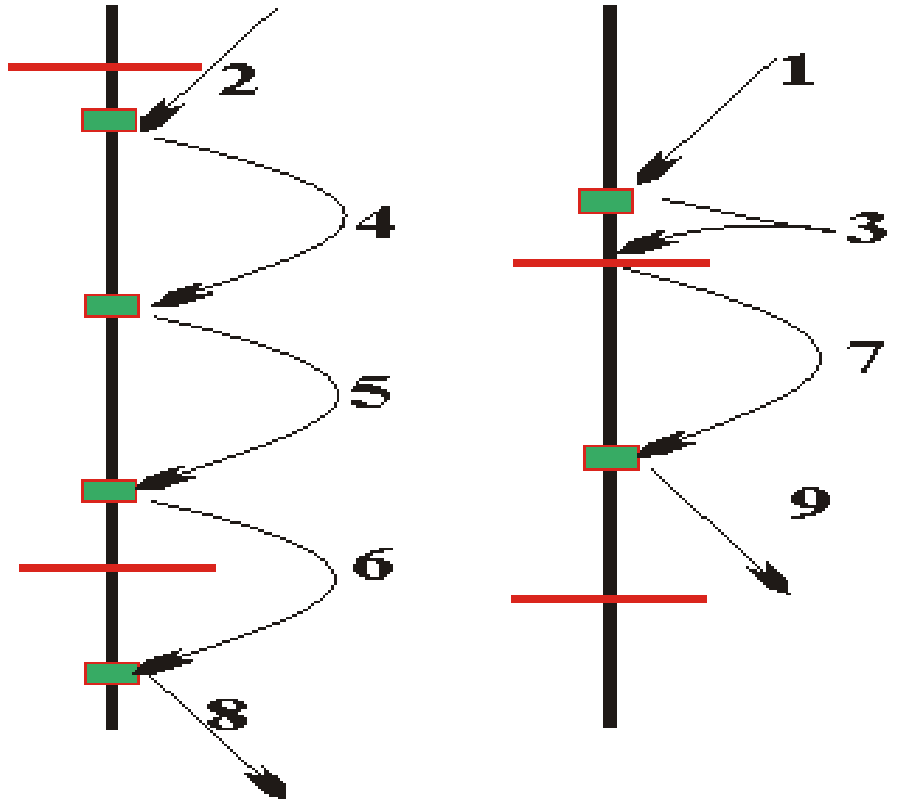
```
unsigned int __stdcall KeyboardThread(void *p)
{
 while (_getch() != 27); // 27 - ESC
 _tprintf(_T("Terminating...\n"));
 InterlockedExchange((long int volatile *)&bStop,
TRUE);
 return 0;
}
// Remark: here the interlocked exchange is not absolutely necessary, you may write simply
// bTerminate = TRUE;
```

Critical sections

```
CRITICAL_SECTION cs;

InitializeCriticalSection(&cs);
EnterCriticalSection(&cs);
..................... // code in critical section
LeaveCriticalSection(&cs);
DeleteCriticalSection(&cs);

if (!TryEnterCriticalSection(&cs))
{
  ............................... // do something else
}
else
{
  .................... // code in critical section
}
```



Critical sections: example

```
#include "stdafx.h"
#include "windows.h"
#include "stdio.h"
#include "process.h"
#include "conio.h"
#include "time.h"
unsigned int __stdcall KeyboardThread(void *);
unsigned int __stdcall ProducerThread(void *);
unsigned int __stdcall ConsumerThread(void *);
volatile BOOL bStop = FALSE;
int Buf[32];
CRITICAL_SECTION CritSect;
int _tmain(int argc, _TCHAR* argv[])
{
        InitializeCriticalSection(&CritSect);
        _beginthreadex(NULL, 0, KeyboardThread, NULL, 0, NULL); // to use ESC for terminating
        _beginthreadex(NULL, 0, ProducerThread, NULL, 0, NULL);
        _beginthreadex(NULL, 0, ConsumerThread, NULL, 0, NULL);
        while (!bStop)
                Sleep(0); // spinlock
        DeleteCriticalSection(&CritSect);
        return 0;
}

unsigned int __stdcall KeyboardThread(void *p)
{
        while (_getch() != 27);
```

```
            _tprintf(_T("Terminating...\n"));
            bStop = TRUE;
            return 0;
        }
```

```
unsigned int __stdcall ProducerThread(void *p)
{       // ProducerThread produces data
        // (array of random numbers)
        int i, delay;
        srand((unsigned)time(NULL));
       // initialize the generator of random numbers
        while (!bStop)
        {
                EnterCriticalSection(&CritSect); // 1
                delay = rand() / (RAND_MAX + 1) *
                                        (5000 - 1000) + 1000;
                Sleep(delay);
                for (i = 0;  i < 32;  i++)
                        Buf[i] = rand();
                LeaveCriticalSection(&CritSect); // 2
        }
        return 0;
}
```

```
unsigned int __stdcall ConsumerThread(void *p)
{       // ConsumerThread consumes the data
        // (computes and prints the checksum)
        long int sum = 0;
        int i;
        while (!bStop)
        {
                EnterCriticalSection(&CritSect); // 3
                for (i = 0, sum = 0;  i < 32;  i++)
                        sum += Buf[i];
                _tprintf(_T("%ld\n"), sum);
                LeaveCriticalSection(&CritSect); // 4
        }
        return 0;
}
```

```
// RAND_MAX is defined as 0x7FFF, here the random delay is between 1000ms and 5000ms
```

```
//
// This code is not 100% correct. The problem is as follows. Although the Producer starts before the
// Consumer, it may happen that row 3 is reached before row 1. In that case the first printed value is 0
// (the Consumer calculates the checksum using the initial values in Buf, i.e. zeroes).
//
// In  addition, this code may not work on the newest versions of Windows. It may happen that the call
// to LeaveCriticalSection() does not make the waiting thread schedulable. Sleeping at points 4 and 5
// mostly solve this problem, but we cannot be sure.
// The corrected code that mostly works:
//
unsigned int __stdcall ProducerThread(void *p)
{     // ProducerThread produces data (array of random numbers)
      int i, delay;
      srand((unsigned)time(NULL)); // initialize the generator of random numbers
      while (!bStop)
      {
            EnterCriticalSection(&CritSect); // 1
            delay = rand() / (RAND_MAX + 1) * (5000 - 1000) + 1000;
            Sleep(delay);
            for (i = 0;  i < 32;  i++)
                  Buf[i] = rand();
            LeaveCriticalSection(&CritSect); // 2
            Sleep(100); // 4
      }
      return 0;
}
```

```
unsigned int __stdcall ConsumerThread(void *p)
{       // ConsumerThread consumes the data (computes the checksum)
        long int sum = 0;
        int i;
        while (!bStop)
        {
                EnterCriticalSection(&CritSect); // 3
                for (i = 0, sum = 0;  i < 32;  i++)
                        sum += Buf[i];
                _tprintf(_T("%ld\n"), sum);
                LeaveCriticalSection(&CritSect); // 4
                Sleep(100); // 5
        }
        return 0;
}
```

Events

```
HANDLE hEvent = CreateEvent (NULL,
            TRUE, // manual reset, FALSE – auto reset
            FALSE, // initial state not signaled, TRUE - signaled
            NULL);
SetEvent(event_handle); // to signaled
ResetEvent(event_handle); // to nonsignaled
CloseHandle(event_handle);

int Result = WaitForSingleObject(event_handle, time_in_milliseconds);
Result = WaitForSingleObject(event_handle, INFINITE);
Results: WAIT_OBJECT_0, WAIT_TIMEOUT, WAIT_FAILED

int Result = WaitForMultipleObjects(number_of_handles,
pointer_to_array_of_handles,
FALSE,  // waits until one of the events has become signaled
        // TRUE - waits until all the events have become signaled
time_in_milliseconds);
Results: WAIT_OBJECT_0, WAIT_OBJECT_0 + 1, WAIT_OBJECT_0 + 2,…..
```

Events: example

```
#include "stdafx.h"
#include "Windows.h"
#include "process.h"  // necessary for threading
// Global variables
TCHAR CommandBuf[81];
HANDLE hCommandGot; // event "the user has typed a command"
HANDLE hStopCommandGot;
                    // event "the main thread has recognized that it was the stop command"
HANDLE hCommandProcessed;
                    // event "the main thread has finished the processing of command"
HANDLE hReadKeyboard;       // keyboard reading thread handle
HANDLE hStdIn;              // stdin standard input stream handle
// Prototypes
unsigned int __stdcall ReadKeyboard(void* pArguments);
//                                  THE MAIN THREAD
int _tmain(int argc, _TCHAR* argv[])
{
   // Initializations for multithreading
   if (!(hCommandGot = CreateEvent(NULL, TRUE, FALSE, NULL)) ||
      !(hStopCommandGot = CreateEvent(NULL, TRUE, FALSE, NULL)) ||
      !(hCommandProcessed = CreateEvent(NULL, TRUE, TRUE, NULL)))
   {
      _tprintf(_T("CreateEvent() failed, error %d\n"), GetLastError());
      return 1;
}
// Prepare keyboard, start the thread
hStdIn = GetStdHandle(STD_INPUT_HANDLE);
if (hStdIn == INVALID_HANDLE_VALUE)
{
    _tprintf(_T("GetStdHandle() failed, error %d\n"), GetLastError());
    return 1;
}
if (!SetConsoleMode(hStdIn, ENABLE_LINE_INPUT | ENABLE_ECHO_INPUT |
                                        ENABLE_PROCESSED_INPUT))
{
    _tprintf(_T("SetConsoleMode() failed, error %d\n"), GetLastError());
    return 1;
}
if (!(hReadKeyboard = (HANDLE)_beginthreadex(NULL, 0, &ReadKeyboard, NULL,
                                                            0, NULL)))
{
    _tprintf(_T("Unable to create keyboard thread\n"));
    return 1;
}
```

```
DWORD WaitResult;
// Main processing loop
    while (TRUE)
    {
        WaitResult = WaitForSingleObject(
                        hCommandGot, INFINITE);
        if (WaitResult == WAIT_OBJECT_0)
        { // The user has typed a command
            ResetEvent(hCommandGot);
                    // CommandGot back to unsignaled
            if (!_tcsicmp(CommandBuf, _T("exit")))
            { // It was the exit command
                SetEvent(hStopCommandGot);
                    // Tell that keyboard thread must quit
                break;
            }
            else
            {
                _tprintf(_T("Unrecognized command\n"),
                                        CommandBuf);
```

```
//              KEYBOARD READING THREAD
unsigned int __stdcall ReadKeyboard(void* pArguments)
{
    DWORD nReadChars;
    HANDLE KeyboardEvents[2];
    KeyboardEvents[1] = hCommandProcessed;
    KeyboardEvents[0] = hStopCommandGot;
    DWORD WaitResult;
    // Reading loop
    while (TRUE)
    {
        WaitResult = WaitForMultipleObjects(2,
                        KeyboardEvents, FALSE, INFINITE);
        if (WaitResult == WAIT_OBJECT_0)
            return 0; // Stop command got
        else if (WaitResult == WAIT_OBJECT_0 + 1)
        { // Command processed, it was not stop
            _tprintf(_T("Insert command\n"));
            if (!ReadConsole(hStdIn, CommandBuf, 80,
                                        &nReadChars, NULL))
            {
                _tprintf(_T("ReadConsole() failed, error %d\n"),
                                                GetLastError());
                return 1;
            }
            CommandBuf[nReadChars - 2] = 0;
            ResetEvent(hCommandProcessed);
            SetEvent(hCommandProcessed);
            // Tell that keyboard thread must continue
         }
      }
      else
      { // Waiting failed
         _tprintf(_T("WaitForSingleObject() failed,\
error %d\n"),  GetLastError());
         return 1;
      }
   }
```

```
                              // hCommandProcessed to non-signaled
         SetEvent(hCommandGot);
                                    // hCommandGot event to signaled
      }
      else
      {  // waiting failed
         _tprintf(_T("WaitForMultipleObjects() failed, \
error %d\n"),GetLastError());
         return 1;
      }
   }
   return 0;
}
```

```
//
// At the beginning the main thread is blocked because hCommandGot is non-signaled
// (WaitForSingleObject() cannot return)
//
// The keyboard thread waits until CommandProcessed or hStopCommandGot becomes
// signaled. Initially hCommandProcessed is signaled, so at the beginning
// WaitForMultipleObjects() returns immediately with WaitResult == WAIT_OBJECT_0 + 1 and
// ReadConsole() starts to wait for the user's command.
//
// When the user has typed a command, two operations are performed:
// 1. hCommandProcessed is set to non-signaled. Therefore WaitForMultipleObjects()
// blocks the keyboard thread.
// 2. hCommandGot is set to signaled. Due to that WaitForSingleObject() in the main thread
// returns, the waiting stops and the analyzing of inserted command may begin.
```

```
//
// When the main thread has ended the analyzing of command, it sets hCommandProcessed
// or hStopCommandGot to signaled. WaitForMultipleObjects returns thus allowing the
// keyboard thread to continue.

    //
    // Shut down
    //
    WaitForSingleObject(hReadKeyboard, INFINITE); // Wait until the end of keyboard thread
    CloseHandle(hReadKeyboard);
    CloseHandle(hStopCommandGot);
    CloseHandle(hCommandGot);
    CloseHandle(hCommandProcessed);
    return 0;
}
```

Semaphores

Functions to work with semaphores:

```
LONG counter_initial_value, counter_max_allowed_value, release_step;
DWORD timeout;
HANDLE hSemaphore = CreateSemaphore(NULL,
counter_initial_value, // if 0, the semaphore initial state is nonsignaled
counter_max_allowed_value, NULL);
ReleaseSemaphore(hSemaphore, release_step, NULL); // counter += step
CloseHandle(hSemaphore);
```

How to use semaphores:

When the semaphore is non-signaled (i.e. its counter is 0), WaitForSingleObject() blocks the thread:

```
WaitForSingleObject(hSemaphore, timeout);
```

When another thread calls ReleaseSemaphore(), the semaphore counter is increased, i.e. the semaphore becomes signaled and WaitForSingleObject() returns. In addition, if the counter was not 0, WaitForSingleObject() decrements it by 1.

If the counter is equal with the max allowed value, ReleaseSemaphore() does nothing.

Semaphores: Example

```
#include "stdafx.h"
#include "windows.h"
#include "stdio.h"
#include "process.h"
#include "conio.h"
#include "time.h"
// global variables
HANDLE hSemaphore;
volatile BOOL bStop = FALSE;
volatile long int nThread;
// prototypes
unsigned int __stdcall KeyboardThread(void *);
unsigned int __stdcall RequestThread(void *);
//                    THE MAIN THREAD
int _tmain(int argc, _TCHAR* argv[])
{
    srand((unsigned int)time(NULL)); // initialize the generator of random numbers
    hSemaphore = CreateSemaphore(NULL, 3, 3, NULL);
                // initialize the semaphore, max 3 concurrent threads are allowed
                // the counter max value is also 3
    _beginthreadex(NULL, 0, KeyboardThread, NULL, 0, NULL);
                                                          // to use ESC for terminating
    while (!bStop)
    {
        Sleep(250); // requests after each 0.25s
        _beginthreadex(NULL, 0, RequestThread, NULL, 0, NULL);
    }
```

```
   return 0;
}
//                   KEYBOARD THREAD
unsigned int __stdcall KeyboardThread(void *p)
{
   while (_getch() != 27);
   _tprintf(_T("Terminating...\n"));
   bStop = TRUE;
   return 0;
}
//                REQUEST PROCESSING THREAD
unsigned int __stdcall RequestThread(void *p)
{
   volatile long n = InterlockedIncrement(&nThread);
   if (WaitForSingleObject(hSemaphore, 0) == WAIT_OBJECT_0)
      _tprintf(_T("Thread %ld started immediately\n"), n);
                                                    // the semaphore was signaled
   else
   {
      _tprintf(_T("Thread %ld waiting for the permission to run\n"), n);
                                                    // the semaphore was not signaled
      WaitForSingleObject(hSemaphore, INFINITE);
      _tprintf(_T("Thread %ld waiting ended, processing started\n"), n);
                                                    // also, the counter was decremented
   }
   Sleep((int)((double)rand() / (RAND_MAX + 1) * (20000 - 1000) + 1000));
                                                    // imitates the processing of request
   ReleaseSemaphore(hSemaphore, 1, NULL); // Increments the counter.
```

```
        // 1. If the counter was 0, the semaphore was not signaled and there could be threads
        // waiting for the permission to run. After incrementing the counter is 1,
        // consequently one of the waiting threads can start.
        // 2. If the counter was not 0, the semaphore was signaled and there are no threads
        // waiting for the  permission to run. If the counter was equal with its max value
        // (here 3), the incrementing is omitted.
    _tprintf(_T("Thread %ld ended\n"), n);
    return 0;
}
```

Mutexes

Mutex - mutual exclusive access.
Mutex without owner thread is signaled, mutex owned by a thread is non-signaled.

```
HANDLE hMutex = CreateMutex(NULL,
FALSE,   // no one thread owns the mutex, i.e. the initial state is signaled
         // TRUE - the thread declaring the mutex also owns it,i.e. the initial state is
         // nonsignaled
NULL);
```

Corresponds to InitializeCriticalSection(*pointer_to_section*);

When WaitForSingleObject(hMutex, timeout); returns because the mutex has become signaled, the current thread is considered as the owner of mutex. Also, the mutex is set to nonsignaled (auto-reset). It corresponds to EnterCriticalSection(*pointer_to_section*);
Only one thread at a time can own a mutex.

ReleaseMutex(hMutex); sets the mutex to signaled and declares that the current thread is not the owner of mutex. Corresponds to LeaveCriticalSection(*pointer_to_section*);
If a thread owing mutex exits without releasing it, the waiting functions return with value WAITING_ABANDONED.

CloseHandle(hMutex); corresponds to DeleteCriticalSection(*pointer_to_section*);

Mutexes: example

```
#include "stdafx.h"
#include "windows.h"
#include "stdio.h"
#include "process.h"
#include "conio.h"
#include "time.h"
unsigned int __stdcall KeyboardThread(void *);
unsigned int __stdcall ProducerThread(void *);
unsigned int __stdcall ConsumerThread(void *);
volatile BOOL bStop = FALSE;
int Buf[32];
HANDLE hMutex;

int _tmain(int argc, _TCHAR* argv[])
{
   hMutex = CreateMutex(NULL, FALSE, NULL);
                  // The mutex is signaled,  the main thread is not the owner of mutex
   _beginthreadex(NULL, 0, KeyboardThread, NULL, 0, NULL);
   _beginthreadex(NULL, 0, ProducerThread, NULL, 0, NULL);
   _beginthreadex(NULL, 0, ConsumerThread, NULL, 0, NULL);
   while (!bStop)
      Sleep(0);
   CloseHandle(hMutex);
   return 0;
}
```

```
unsigned int __stdcall KeyboardThread(void *p)
{
   while (_getch() != 27); // ESC for terminating
   _tprintf(_T("Terminating...\n"));
   bStop = TRUE;
   return 0;
}


// The following code is not 100% correct:
// Although the Producer starts before the Consumer, it may
// happen that the Consumer grabs the mutex first. In that case
// the first printed value is 0 (the Consumer calculates
// the checksum using the initial values in Buf, i.e. zeroes).
```

```
unsigned int __stdcall ProducerThread(void *p)
{  // ProducerThread produces data
     // (array of random numbers)
   int i, delay;
   srand((unsigned)time(NULL));
     // initialize the generator of random numbers
   while (!bStop)
   {
      WaitForSingleObject(hMutex, INFINITE);
      // The mutex becomes non-signaled, the
      // producer thread is the owner of mutex.
      // The consumer thread is blocked
      delay = rand() / (RAND_MAX + 1) *
                                (5000 - 1000) + 1000;
      Sleep(delay);
      for (i = 0;  i < 32;  i++)
         Buf[i] = rand();
      ReleaseMutex(hMutex);
      // The mutex becomes signaled, the
      // producer thread is not the owner of
      // mutex. The consumer thread is released
   }
   return 0;
}
```

```
unsigned int __stdcall ConsumerThread(void *p)
{  // ConsumerThread consumes the data
   // (computes and prints the checksum)
   long int sum = 0;
   int i;
   while (!bStop)
   {
      WaitForSingleObject(hMutex, INFINITE);
      // The mutex becomes non-signaled, the
      // consumer thread is the owner of mutex.
      // The producer thread is blocked.
      for (i = 0, sum = 0;  i < 32;  i++)
         sum += Buf[i];
      _tprintf(_T("%ld\n"), sum);
      ReleaseMutex(hMutex);
      // The mutex becomes signaled, the
      // consumer thread is not the owner of
      // mutex. The producer thread is released.
   }
   return 0;
}
```

Asynchronous input/output

```
HANDLE hFile = CreateFile(file_name, // example: _T("\\.\\COM1") for serial port 1
GENERIC_READ | GENERIC_WRITE, 0, NULL, creation_disposition,
FILE_FLAG_OVERLAPPED, // specifies asynchronous input/output
NULL);
OVERLAPPED Overlapped;
memset(&Overlapped, 0, sizeof Overlapped); // do not forget
Overlapped.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL); // auto reset
DWORD nBytesToRead, nReadBytes = 0;
BYTE *pBuffer;
if  (!ReadFile(hFile, pBuffer, nBytesToRead, &nReadBytes, &Overlapped))
    switch (GetLastError())
    {
      case ERROR_IO_PENDING:
              WaitForSingleObject(Overlapped.hEvent, INFINITE);
              // when the reading has ended, Overlapped.hEvent becomes signaled
              GetOverlappedResult(hFile, &Overlapped, &nReadBytes, FALSE);
              break;
      default:
              _tprintf(_T("Errors"));
    }
```

Asynchronous input/output: example

```
#include "stdafx.h"
#include "Windows.h"
#include "process.h"
// Global variables
TCHAR CommandBuf[81];
HANDLE hCommandGot;       // event "the user has typed a command"
HANDLE hStopCommandGot;
        // event "the main thread has recognized that it was the stop command"
HANDLE hCommandProcessed;
       // event "the main thread has finished the processing of command"
HANDLE hReadKeyboard;     // keyboard reading thread handle
HANDLE hStdIn;            // stdin standard input stream handle
// Prototypes
unsigned int __stdcall ReadKeyboard(void* pArguments);
void __cdecl ReadComPort(void* pArguments);
// prototypes are different because we use different functions
// (_beginthreadex and beginthread) to start the thread
//          MAIN THREAD
int _tmain(int argc, _TCHAR* argv[])
{
// Initializations for multithreading
   if (!(hCommandGot = CreateEvent(NULL, TRUE, FALSE, NULL)) ||
       !(hStopCommandGot = CreateEvent(NULL, TRUE, FALSE, NULL)) ||
       !(hCommandProcessed = CreateEvent(NULL, TRUE, TRUE, NULL)))
```

```
{
   _tprintf(_T("CreateEvent() failed, error %d\n"), GetLastError());
   return 1;
}
// Prepare keyboard, start the thread
hStdIn = GetStdHandle(STD_INPUT_HANDLE);
if (hStdIn == INVALID_HANDLE_VALUE)
{
   _tprintf(_T("GetStdHandle() failed, error %d\n"), GetLastError());
   return 1;
}
if (!SetConsoleMode(hStdIn, ENABLE_LINE_INPUT |
                            ENABLE_ECHO_INPUT | ENABLE_PROCESSED_INPUT))
{
   _tprintf(_T("SetConsoleMode() failed, error %d\n"), GetLastError());
   return 1;
}
if (!(hReadKeyboard = (HANDLE)_beginthreadex(NULL, 0,
                                         &ReadKeyboard, NULL, 0, NULL)))
{
   _tprintf(_T("Unable to create keyboard thread\n"));
   return 1;
}
```

```
// Main processing loop
while (TRUE)
{
    if (WaitForSingleObject(hCommandGot, INFINITE) != WAIT_OBJECT_0)
    { // Wait until the command has arrived (i.e. until CommandGot is signaled)
        _tprintf(_T("WaitForSingleObject() failed, error %d\n"), GetLastError());
        return 0;
    }
    ResetEvent(hCommandGot); // CommandGot back to unsignaled
    if (!_tcsicmp(CommandBuf, _T("exit"))) // Case-insensitive comparation
    {
        SetEvent(hStopCommandGot); // To force the other threads to quit
        break;
    }
    else if (!_tcsicmp(CommandBuf, _T("read")))
    {
        SetEvent(hCommandProcessed);
                                    // To allow the keyboard reading thread to continue
        if (_beginthread(&ReadComPort, 0, NULL) == 1)
        {  // Here we use _beginthread instead of _beginthreadex. The reason is
            // that in case of _beginthread,  when the thread main function returns,
            // it automatically closes the handler. Due to it we can several times
            // create this thread without bothering about the handler.
            _tprintf(_T("Unable to create COM port thread\n"));
            return 0;
        }
    }
    else
```

```
        {
            _tprintf(_T("Do not know how to manage command \"%s\"\n"), CommandBuf);
            SetEvent(hCommandProcessed);
                                                  // To allow the keyboard reading thread to continue
        }
    }
    // Shut down
    WaitForSingleObject(hReadKeyboard, INFINITE); // Wait until the end of thread
    CloseHandle(hReadKeyboard);
    CloseHandle(hStopCommandGot);
    CloseHandle(hCommandGot);
    CloseHandle(hCommandProcessed);
    return 1;
}
```

```
//                         KEYBOARD READING THREAD
unsigned int __stdcall ReadKeyboard(void* pArguments)
{
   DWORD nReadChars;
   HANDLE KeyboardEvents[2];
   KeyboardEvents[1] = hCommandProcessed;
   KeyboardEvents[0] = hStopCommandGot;
   DWORD WaitResult;
   // Reading loop
   while (TRUE)
   {
      WaitResult = WaitForMultipleObjects(2, KeyboardEvents,
         FALSE, // wait until one of the events becomes signaled
         INFINITE);
              // Waiting until hCommandProcessed or hStopCommandGot becomes
              // signaled. Initially hCommandProcessed  is signaled, so at the beginning
              // WaitForMultipleObjects() returns immediately with WaitResult equal
              // with WAIT_OBJECT_0 + 1.
      if (WaitResult == WAIT_OBJECT_0)
         return 0;  // Stop command, i.e. hStopCommandGot is signaled
      else if (WaitResult == WAIT_OBJECT_0 + 1)
      { // If the signaled event is hCommandProcessed, the WaitResult is
        // WAIT_OBJECT_0 + 1
         _tprintf(_T("Insert command\n"));
         if (!ReadConsole(hStdIn, CommandBuf, 80, &nReadChars, NULL))
         { // The problem is that when we already are in this function, the only way
           // to leave it is to type something and then press ENTER. So we cannot
           // step into this function at any moment. WaitForMultipleObjects()
```

```
                // prevents it.
                 _tprintf(_T("ReadConsole() failed, error %d\n"), GetLastError());
                 return 1;
              }
              CommandBuf[nReadChars - 2] = 0;
                             // The command in buf ends with "\r\n", we have to get rid of them
              ResetEvent(hCommandProcessed);
              // Set hCommandProcessed to non-signaled. Therefore WaitForMultipleObjects()
              // blocks the keyboard thread. When the main thread has ended the analyzing of
              // command, it sets hCommandprocessed or hStopCommandGot to signaled and
              // the keyboard thread can continue.
              SetEvent(hCommandGot);
              // Set hCommandGot event to signaled. Due to that WaitForSingleObject() in the
              // main thread returns, the waiting stops and the analyzing of inserted command
              // may begin
           }
           else
           {  // waiting failed
              _tprintf(_T("WaitForMultipleObjects()failed, error %d\n"), GetLastError());
              return 1;
           }
        }
        return 0;
     }
```

```
//                    COM1 ASYNCHRONOUS READING THREAD
void __cdecl ReadComPort(void* pArguments)
{
   // Preparations
   OVERLAPPED Overlapped;
   BYTE DataBuf[1024];
   DWORD nReadBytes;
   HANDLE hCOM1 = CreateFile(_T("\\\\.\\COM1"),  // serial port COM1 as file
                                          GENERIC_READ,
                                          0, NULL,
                                          OPEN_EXISTING,  // no alternatives here
                                          FILE_FLAG_OVERLAPPED,
                                                            // necessary for asynchronous input/output
                                          NULL);
   if (hCOM1 == INVALID_HANDLE_VALUE)
   {
      _tprintf(_T("Unable to open COM1\n"));
      return;
   }
   memset(&Overlapped, 0, sizeof Overlapped);
   Overlapped.hEvent=CreateEvent(NULL, FALSE, FALSE, NULL);
                                                         // event "ReadFile() has returned"
   HANDLE ComPortEvents[2];
   ComPortEvents[1] = Overlapped.hEvent;
   ComPortEvents[0] = hStopCommandGot;
   // Reading
   if (!ReadFile(hCOM1,
                  DataBuf,  // buffer
                  1024,     // buffer length
                  &nReadBytes,  // number of read bytes
                  &Overlapped))
// As the last parameter is not 0 and the file is declared as FILE_FLAG_OVERLAPPED,
// ReadFile() returns immediately. If the reading ws successful, the return value is not
// zero. If the reading was not successful, the return value is zero. But this does not
// mean a catastrophe. If the GetLastError() gives error ERROR_IO_PENDING, the reading
// was not finished (probably because there is no data to read), so we have simply
// to wait.
{  // Was not able to read immediately, ReadFile() return value was 0
      DWORD lError = GetLastError();
      if (lError != ERROR_IO_PENDING)
      {  // Serious problems
         _tprintf(_T("Unable to read from COM port, error %d\n"), lError);
            goto out; // although goto is not recommended to use, here it is very practical
      }
      else
      {  // Simply could not to finish the reading in time, let us wait
         DWORD WaitResult = WaitForMultipleObjects(2, ComPortEvents,
            FALSE, // wait until one of the events becomes signaled
            INFINITE);
         switch (WaitResult) // analyse why the waiting ended
         {
            case WAIT_OBJECT_0:
            // Waiting stopped because hStopCommandGot has become signaled, i.e. the user
            // has decided to exit
               goto out;
            case WAIT_OBJECT_0 + 1:
            // Waiting stopped because Overlapped.hEvent has become now signaled, i.e. the
            // reading operation has ended. Now we may to see how many bytes we have got.
            // nReadBuytes may have any value between 0 and 1024
              if (GetOverlappedResult(hCOM1, &Overlapped, &nReadBytes, FALSE))
                _tprintf(_T("%d bytes read from COM1\n"), nReadBytes);
              else
                _tprintf(_T("GetOverlappedResult() failed, error %d\n"), GetLastError());
              goto out;
            default:    // Serious problems
              _tprintf(_T("WaitForMultipleObjects() failed, error %d\n"), GetLastError());
              goto out;
            }
        }
    }
    else
        _tprintf(_T("%d bytes read from COM1\n"), nReadBytes);
        // ReadFile() read data without any delay. nReadBuytes may have any value
        // between 0 and 1024
    // Shut down
    out:
    CancelIo(hCOM1); // If there still are pending I/O operations (they can be if the reading
                     // failed), cancels them
    CloseHandle(Overlapped.hEvent);
    CloseHandle(hCOM1);
    return;
  }
```

Sockets (1)

Visual Studio project features:
Project properties, linker input, additional dependencies: ws2_32.lib, mswsock.lib

```
#include "Winsock2.h" // instead of Windows.h
#include "mswsock.h"
```

Windows socket support:

```
WSADATA WSAData;
WSAStartup(MAKEWORD(2, 0), &WSAData); // initializes Winsock v2.0 support
WSACleanup(); // terminates the use of the Winsock v2.0
```

Socket handle:

```
SOCKET hSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (hSocket == INVALID_SOCKET)
    _tprintf("Failed to create socket, error %d\n", WSAGetLastError());
```

Sockets (2)

Preparations for client socket (binding the socket):

```
sockaddr_in SocketInfo;
memset(&SocketInfo, 0, sizeof SocketInfo); // necessary, do not forget
SocketInfo.sin_familiy = AF_INET;
SocketInfo.sin_port = htons(server_port_number); // for example htons(7654);
SocketInfo.sin_addr.s_addr = inet_addr(server_IP_address);
        // for example inet_addr("10.10.10.120");
        // if the server and client are running on the same computer, use address
        // inet_addr("127.0.0.1");
```

Connecting to server:

```
if (connect(hSocket, (SOCK_ADDR *)&SocketInfo, sizeof SocketInfo) ==
                        SOCKET_ERROR)
        _tprintf(_T("Connecting failed, error %d", WSAGetLastError());
```

Sockets (3)

Closing the connection:

```
shutdown(hSocket, SD_RECEIVE); // disable reading
// options: SD_SEND (disable writing), SD_BOTH (disable everything)
closesocket(hSocket); // first shut down, then close
```

Preparations for reading and writing:

```
WSABuf DataBuf;
DataBuf.len = buffer_length; // DWORD
DataBuf.buf = new char[DataBuf.len]; // char *
WSAOVERLAPPED Overlapped;
memset(&Overlapped, 0, sizeof Overlapped); // necessary, do not forget
Overlapped.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
 // not signaled, manual reset
 // option: Overlapped.hEvent = WSACreateEvent(); creates also a regular
 // non-signaled manual reset event
DWORD Flags= 0, nReceivedBytes = 0;
```

Sockets (4)

Reading:

```
int Result = WSARecv(hSocket, &DataBuf, 1, &nReceivedBytes, &Flags,
          &Overlapped, NULL);
if (Result == SOCKET_ERROR)
   switch (WSAGetLastError())
      {
       case WSA_IO_PENDING:
          WSAWaitForMultipleObjects(1, &Overlapped.hEvent, TRUE,
                   INFINITE, FALSE);
          // when the reading has ended, Overlapped.hEvent becomes signaled
          ResetEvent(Overlapped.hEvent); // manual reset
          // or WSAResetEvent(Overlapped.hEvent)
          WSAGetOverlappedResult(hSocket, &Overlapped, & nReceivedBytes,
                   FALSE, &Flags);
          break;
       default:
          _tprintf(_T("Errors"));
      }
else if (!nReceivedBytes)
    _tprintf(_T("Connection broken off"));
```

Sockets: example

```
#include "stdafx.h"
#include "Winsock2.h" // necessary for sockets, Windows.h is not needed.
#include "mswsock.h"
#include "process.h"  // necessary for threading
// Global variables
TCHAR CommandBuf[81];
HANDLE hCommandGot;        // event "the user has typed a command"
HANDLE hStopCommandGot;
                   // event "the main thread has recognized that it was the stop command"
HANDLE hCommandProcessed;
                   // event "the main thread has finished the processing of command"
HANDLE hReadKeyboard;     // keyboard reading thread handle
HANDLE hStdIn;            // stdin standard input stream handle
WSADATA WsaData;          // filled during Winsock initialization
DWORD Error;
SOCKET hClientSocket = INVALID_SOCKET;
sockaddr_in ClientSocketInfo;
HANDLE hReceiveNet;       // TCP/IP info reading thread handle
BOOL SocketError;
// Prototypes
unsigned int __stdcall ReadKeyboard(void* pArguments);
unsigned int __stdcall ReceiveNet(void* pArguments);
```

```
//                              THE MAIN THREAD
int _tmain(int argc, _TCHAR* argv[])
{
    // Initializations for multithreading
    if (!(hCommandGot = CreateEvent(NULL, TRUE, FALSE, NULL)) ||
        !(hStopCommandGot = CreateEvent(NULL, TRUE, FALSE, NULL)) ||
        !(hCommandProcessed = CreateEvent(NULL, TRUE, TRUE, NULL)))
    {
        _tprintf(_T("CreateEvent() failed, error %d\n"), GetLastError());
        return 1;
    }
    // Prepare keyboard, start the thread
    hStdIn = GetStdHandle(STD_INPUT_HANDLE);
    if (hStdIn == INVALID_HANDLE_VALUE)
    {
        _tprintf(_T("GetStdHandle() failed, error %d\n"), GetLastError());
        return 1;
    }
    if (!SetConsoleMode(hStdIn, ENABLE_LINE_INPUT | ENABLE_ECHO_INPUT |
                                                  ENABLE_PROCESSED_INPUT))
    {
        _tprintf(_T("SetConsoleMode() failed, error %d\n"), GetLastError());
        return 1;
    }
    if (!(hReadKeyboard = (HANDLE)_beginthreadex(NULL, 0, &ReadKeyboard,
                                                         NULL, 0, NULL)))
    {
        _tprintf(_T("Unable to create keyboard thread\n"));
    return 1;
}
// Initializations for socket
if (Error = WSAStartup(MAKEWORD(2, 0), &WsaData))
                                                        // Initialize Windows socket support
{
    _tprintf(_T("WSAStartup() failed, error %d\n"), Error);
    SocketError = TRUE;
}
else if ((hClientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) ==
                                                                        INVALID_SOCKET)
{
    _tprintf(_T("socket() failed, error %d\n"), WSAGetLastError());
    SocketError = TRUE;
}
// Connect client to server
if (!SocketError)
{
    ClientSocketInfo.sin_family = AF_INET;
    ClientSocketInfo.sin_addr.s_addr = inet_addr("127.0.0.1");
    ClientSocketInfo.sin_port = htons(1234);  // port number is selected just for example
    if (connect(hClientSocket, (SOCKADDR*) &ClientSocketInfo,
                                             sizeof(ClientSocketInfo)) == SOCKET_ERROR)
    {
        _tprintf(_T("Unable to connect to server, error %d\n"), WSAGetLastError());
        SocketError = TRUE;
    }
}
```

```
// Start net thread
if (!SocketError)
{
    if (!(hReceiveNet= (HANDLE)_beginthreadex(NULL, 0, &ReceiveNet,
                                                                          NULL, 0, NULL)))
    {
        _tprintf(_T("Unable to create socket receiving thread\n"));
        goto out;
    }
}
// Main processing loop
while (TRUE)
{
    if (WaitForSingleObject(hCommandGot, INFINITE) != WAIT_OBJECT_0)
    { // Wait until the command has arrived (i.e. until CommandGot is signaled)
        _tprintf(_T("WaitForSingleObject() failed, error %d\n"), GetLastError());
        goto out;
    }
    ResetEvent(hCommandGot); // CommandGot back to unsignaled
    if (!_tcsicmp(CommandBuf, _T("exit"))) // Case-insensitive comparation
    {
        SetEvent(hStopCommandGot); // To force the other threads to quit
        break;
    }
    else
    {
        _tprintf(_T("Command \"%s\" not recognized\n"), CommandBuf);
        SetEvent(hCommandProcessed); // To allow the keyboard reading thread to continue
        }
    }
    // Shut down
out:
    if (hReadKeyboard)
    {
        WaitForSingleObject(hReadKeyboard, INFINITE); // Wait until the end of keyboard thread
        CloseHandle(hReadKeyboard);
    }
    if (hReceiveNet)
    {
        WaitForSingleObject(hReceiveNet, INFINITE); // Wait until the end of receive thread
        CloseHandle(hReceiveNet);
    }
    if (hClientSocket != INVALID_SOCKET)
    {
        if (shutdown(hClientSocket, SD_RECEIVE) == SOCKET_ERROR)
        {
            if ((Error = WSAGetLastError()) != WSAENOTCONN)
                                        // WSAENOTCONN means that the connection was not established,
                                        // so the shut down was senseless
                _tprintf(_T("shutdown() failed, error %d\n"), WSAGetLastError());
        }
        closesocket(hClientSocket);
    }
    WSACleanup(); // clean Windows sockets support
    CloseHandle(hStopCommandGot);
    CloseHandle(hCommandGot);
    CloseHandle(hCommandProcessed);
    return 0;
}
//                    KEYBOARD READING THREAD
unsigned int __stdcall ReadKeyboard(void* pArguments)
{
    DWORD nReadChars;
    HANDLE KeyboardEvents[2];
    KeyboardEvents[1] = hCommandProcessed;
    KeyboardEvents[0] = hStopCommandGot;
    DWORD WaitResult;
    // Reading loop
    while (TRUE)
    {
        WaitResult = WaitForMultipleObjects(2, KeyboardEvents,
            FALSE, // wait until one of the events becomes signaled
            INFINITE);
            // Waiting until hCommandProcessed or hStopCommandGot becomes signaled. Initially
            // hCommandProcessed is signaled, so at the beginning WaitForMultipleObjects()
            // returns immediately with WaitResult equal with WAIT_OBJECT_0 + 1.
        if (WaitResult == WAIT_OBJECT_0)
            return 0;  // Stop command, i.e. hStopCommandGot is signaled
        else if (WaitResult == WAIT_OBJECT_0 + 1)
        { // If the signaled event is hCommandProcessed, the WaitResult is WAIT_OBJECT_0 + 1
            _tprintf(_T("Insert command\n"));
            if (!ReadConsole(hStdIn, CommandBuf, 80, &nReadChars, NULL))
            {  // The problem is that when we already are in this function, the only way to leave it
                // is to type something and then press ENTER. So we cannot step into this function at
                // any moment. WaitForMultipleObjects() prevents it.
                _tprintf(_T("ReadConsole() failed, error %d\n"), GetLastError());
                return 1;
            }
            CommandBuf[nReadChars - 2] = 0;
                                    // The command in buf ends with "\r\n", we have to get rid of them
            ResetEvent(hCommandProcessed);
            // Set hCommandProcessed to non-signaled. Therefore WaitForMultipleObjects() blocks
            // the keyboard thread. When the main thread has ended the analyzing of command, it
            // sets hCommandprocessed or hStopCommandGot to signaled and the keyboard thread
            // can continue.
            SetEvent(hCommandGot);
            // Set hCommandGot event to signaled. Due to that WaitForSingleObject() in the main
            // thread returns, the waiting stops and the analyzing of inserted command may begin
        }
        else
        {  // waiting failed
            _tprintf(_T("WaitForMultipleObjects()failed, error %d\n"), GetLastError());
            return 1;
        }
    }
    return 0;
}
```

```
//                        TCP/IP INFO RECEIVING THREAD
unsigned int __stdcall ReceiveNet(void* pArguments)
{
    // Preparations
    WSABUF DataBuf;  // Buffer for received data is a structure
    char ArrayInBuf[2048];
    DataBuf.buf = &ArrayInBuf[0];
    DataBuf.len = 2048;
    DWORD nReceivedBytes = 0, ReceiveFlags = 0;
    HANDLE NetEvents[2];
    NetEvents[0] = hStopCommandGot;
    WSAOVERLAPPED Overlapped;
    memset(&Overlapped, 0, sizeof Overlapped);
    Overlapped.hEvent = NetEvents[1] = WSACreateEvent(); // manual and nonsignaled
    DWORD Result, Error;
    // Receiving loop
    while (TRUE)
    {
        Result = WSARecv(hClientSocket,
                         &DataBuf,
                         1,  // no comments here
                         &nReceivedBytes,
                         &ReceiveFlags, // no comments here
                         &Overlapped,
                         NULL);  // no comments here
        if (Result == SOCKET_ERROR)
        {  // Returned with socket error, let us examine why
            if ((Error = WSAGetLastError()) != WSA_IO_PENDING)
```

```
{  // Unable to continue, for example because the server has closed the connection
   _tprintf(_T("WSARecv() failed, error %d\n"), Error);
   goto out;
}
DWORD WaitResult = WSAWaitForMultipleEvents(2, NetEvents, FALSE,
                                           WSA_INFINITE, FALSE); // wait for data
switch (WaitResult) // analyse why the waiting ended
{
case WAIT_OBJECT_0:
   // Waiting stopped because hStopCommandGot has become signaled, i.e.
   // the user has decided to exit
   goto out;
case WAIT_OBJECT_0 + 1:
   // Waiting stopped because Overlapped.hEvent is now signaled, i.e. the receiving
   // operation has ended. Now we have to see how many bytes we have got.
   WSAResetEvent(NetEvents[1]); // to be ready for the next data package
   if (WSAGetOverlappedResult(hClientSocket, &Overlapped, &nReceivedBytes,
                                                      FALSE, &ReceiveFlags))
   {
     _tprintf(_T("%d bytes received\n"), nReceivedBytes);
           // Here should follow the processing of received data
     break;
   }
   else
   {  // Fatal problems
     _tprintf(_T("WSAGetOverlappedResult() failed, error %d\n"), GetLastError());
     goto out;
   }
```

```
            default: // Fatal problems
                _tprintf(_T("WSAWaitForMultipleEvents() failed, error %d\n"),
                                                                                                    WSAGetLastError());
                goto out;
            }
        }
        else
        {  // Returned immediately without socket error
            if (!nReceivedBytes)
            {  // When the receiving function has read nothing and returned immediately,
                // the connection is off
                _tprintf(_T("Server has closed the connection\n"));
                goto out;
            }
            else
            {
             _tprintf(_T("%d bytes received\n"), nReceivedBytes);
                      // Here should follow the processing of received data
            }
        }
    }
out:
    WSACloseEvent(NetEvents[1]);
    return 0;
}
```

Kernel objects

Kernel objects for synchronization: events, semaphores, mutexes. Critical sections are not kernel objects.

Other kernel objects: file, socket, thread, processs, etc. Remark that one process can launch another process using the CreateProcess() function.

CreateEvent(), CreateSemaphore(), CreateFile() etc. create a structure administered by Windows. Object handle is the object ID for process that called the CreateXXX() function.

A kernel object may be common for several concurrently running processes. Those processes usually interacts (IPC - interprocess communication), using for example TCP/IP connection or specific mechanisms called pipes. If for example two processes share the same file, their operating must be synchronized.

Problems with kernel objects shared by several processes: a) security; b) naming (handles are valid in one process).

```
HANDLE hEvent = CreateEvent (pSecurityAttributes,
// Pointer to SECURITY_ATTRIBUTES, this struct presents the security descriptor
// NULL means that the default security is accepted
Manual_or_automatic_reset, Initial_state,
pObjectName); // String presenting the object name. If another process needs to use this
// event, it must first call the OpenEvent() function. Object name is one of parameters of
// OpenEvent().
```

Structured exception handling (1)

Some exceptions: EXCEPTION_ACCESS_VIOLATION, EXCEPTION_STACK_OVERFLOW, EXCEPTION_INT_DIVIDE_BY_ZERO, EXCEPTION_FLT_OVERFLOW (defined in WinBase.h)

Exceptions lead to application crash, but a crash is unacceptable. To foresee all the exceptions may be impossible and to insert checkings before each critical operation is cumbersome. The exception handlers are the solution.

```
__try // it’s a Microsoft complement, not equivalent with "try" from standard C++
{
 // Operations that may cause exceptions
}
__except (exception_filter)
{
// Exception handler block
}
```

Exception filters: EXCEPTION_EXECUTE_HANDLER (when the __except block has executed and it does not forces the function to return, the execution continues after the __except block), EXCEPTION_CONTINUE_EXECUTION (jumps back to the machine instruction that generated the exception), EXCEPTION_CONTINUE_ SEARCH.

```
DWORD ExceptionCode = GetExceptionCode();
```

Execption handler: example

```
#include "stdafx.h"
#include "Windows.h"

void *DuplicateArray(void *, int);
void *SafeDuplicateArray1(void *, int);
void *SafeDuplicateArray2(void *, int);
void *Dummy1(void *, int);
void *Dummy2(void *, int);
void *Dummy3(void *, int);

int _tmain(int argc, _TCHAR* argv[])
{
    BYTE *pArray = new BYTE[10];
    int i;
    for (i = 0;  i < 10;  i++)
        pArray[i] = i;
    //BYTE *pDuplicate = (BYTE *)DuplicateArray(pArray, 10);
    //BYTE *pDuplicate = (BYTE *)SafeDuplicateArray1(pArray, 10);
    BYTE *pDuplicate = (BYTE *)SafeDuplicateArray2(NULL, 10);
    if (pDuplicate)
    {
        for (i = 0;  i < 10;  i++)
            _tprintf(_T("%d\n"), *(pDuplicate + i));
    }
    else
        _tprintf(_T("Duplicate failed"));
    return 0;
}

void *DuplicateArray(void *pOriginal, int nLength)
{       // Problems due to incorrect input: pOriginal is NULL or nLength <= 0
        // Problems due to memory allocation errors: pCopy becomes NULL
        // In case of those problems the application crashes
    void *pCopy = (void *)new char[nLength];
    memcpy(pCopy, pOriginal, nLength);
    return pCopy;
}

void *SafeDuplicateArray1(void *pOriginal, int nLength)
{       // In case of problems SafeDuplicateArray1() returns NULL. The calling
        // function must take it into consideration
    void *pCopy = NULL;
    __try
    {
        if (nLength > 0)
        {
            pCopy = (void *)new char[nLength];
            memcpy(pCopy, pOriginal, nLength);
        }
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        if (pCopy)
            delete pCopy;
        pCopy = NULL;
    }
    return pCopy;
}

void *SafeDuplicateArray2(void *pOriginal, int nLength)
{
    if (nLength <= 0)
        return NULL;
    // when nLength < 0, the memory allocation system does not produce an exception,
    // it simply creashes. Therefore __try / __except mechanism does not solve that
    // problem and we need an additional check
        void *pCopy = NULL;
    __try
    {
        pCopy = Dummy1(pOriginal, nLength);
        // Dummy1() calls Dummy2(), Dummy2() calls Dummy3(), Dummy3()
        // calls DuplicateArray().
        return pCopy;
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {  // The exception was raised in DuplicateArray() but caught in SafeDuplicateArray2()
        if (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION)
            _tprintf(_T("Access violation exception\n"));
        if (pCopy)
            delete pCopy;
        return NULL;
    }
}
void *Dummy1(void *pOriginal, int nLength)
{
    return Dummy2(pOriginal, nLength);
}

void *Dummy2(void *pOriginal, int nLength)
{
    return Dummy3(pOriginal, nLength);
}

void *Dummy3(void *pOriginal, int nLength)
{
    return DuplicateArray(pOriginal, nLength);
}
```

Structured exception handling (2)

```
__try
{
 // Operations containing statements like return, break, continue, goto
}
__finally
{
// Termination handler block
}
```

In case of normal flow (no jumps out of the __try block due to return, break, etc.) the __finally block is executed right after the __try block.

In case of abnormal flow before jumping out due to return, break, etc. of the __try block the __finally block is executed.

Mostly, the __finally block is for clean up (closing the handles, releasing the memory, releasing other threads blocked by terminating thread, etc.).

Termination handle: example

```
// This code bases on the Asychronous input/output example. Differences are in the
// ReadComPort() thread
#include "stdafx.h"
#include "Windows.h"
#include "process.h"
// Global variables
TCHAR CommandBuf[81];
HANDLE hCommandGot;       // event "the user has typed a command"
HANDLE hStopCommandGot;
                          // event "the main thread has recognized that it was the stop command"
HANDLE hCommandProcessed;
                              // event "the main thread has finished the processing of command"
HANDLE hReadKeyboard;     // keyboard reading thread handle
HANDLE hStdIn;            // stdin standard input stream handle
// Prototypes
unsigned int __stdcall ReadKeyboard(void* pArguments);
void __cdecl ReadComPort(void* pArguments);
                      // prototypes are different because we use different functions
                      // (_beginthreadex and beginthread) to start the thread
//                        MAIN THREAD
int _tmain(int argc, _TCHAR* argv[])
{
  // Initializations for multithreading
  if (!(hCommandGot = CreateEvent(NULL, TRUE, FALSE, NULL)) ||
    !(hStopCommandGot = CreateEvent(NULL, TRUE, FALSE, NULL)) ||
    !(hCommandProcessed = CreateEvent(NULL, TRUE, TRUE, NULL)))
{
    _tprintf(_T("CreateEvent() failed, error %d\n"), GetLastError());
    return 1;
}
// Prepare keyboard, start the thread
hStdIn = GetStdHandle(STD_INPUT_HANDLE);
if (hStdIn == INVALID_HANDLE_VALUE)
{
    _tprintf(_T("GetStdHandle() failed, error %d\n"), GetLastError());
    return 1;
}
if (!SetConsoleMode(hStdIn, ENABLE_LINE_INPUT | ENABLE_ECHO_INPUT |
                                        ENABLE_PROCESSED_INPUT))
{
    _tprintf(_T("SetConsoleMode() failed, error %d\n"), GetLastError());
    return 1;
}
if (!(hReadKeyboard = (HANDLE)_beginthreadex(NULL, 0, &ReadKeyboard,
                                                                NULL, 0, NULL)))
{
    _tprintf(_T("Unable to create keyboard thread\n"));
    return 1;
}
// Main processing loop
while (TRUE)
{
```

```
if (WaitForSingleObject(hCommandGot, INFINITE) != WAIT_OBJECT_0)
{ // Wait until the command has arrived (i.e. until CommandGot is signaled)
    _tprintf(_T("WaitForSingleObject() failed, error %d\n"), GetLastError());
    return 0;
}
ResetEvent(hCommandGot); // CommandGot back to unsignaled
if (!_tcsicmp(CommandBuf, _T("exit"))) // Case-insensitive comparation
{
    SetEvent(hStopCommandGot); // To force the other threads to quit
    break;
}
else if (!_tcsicmp(CommandBuf, _T("read")))
{
    SetEvent(hCommandProcessed); // To allow the keyboard reading thread to continue
    if (_beginthread(&ReadComPort, 0, NULL) == 1)
    {       // Here we use _beginthread instead of _beginthreadex. The reason is that in case of
            //_beginthread, when the thread main function returns, it automatically closes the
            // handler. Due to it we can several times create this thread without bothering about
            // the handler.
        _tprintf(_T("Unable to create COM port thread\n"));
        return 0;
    }
}
else
{
    _tprintf(_T("Do not know how to manage command \"%s\"\n"), CommandBuf);
    SetEvent(hCommandProcessed); // To allow the keyboard reading thread to continue
}
```

```
    }
    // Shut down
    WaitForSingleObject(hReadKeyboard, INFINITE); // Wait until the end of thread
    CloseHandle(hReadKeyboard);
    CloseHandle(hStopCommandGot);
    CloseHandle(hCommandGot);
    CloseHandle(hCommandProcessed);
    return 1;
}
//                          KEYBOARD READING THREAD
unsigned int __stdcall ReadKeyboard(void* pArguments)
{
    DWORD nReadChars;
    HANDLE KeyboardEvents[2];
    KeyboardEvents[1] = hCommandProcessed;
    KeyboardEvents[0] = hStopCommandGot;
    DWORD WaitResult;
    // Reading loop
    while (TRUE)
    {
        WaitResult = WaitForMultipleObjects(2, KeyboardEvents,
            FALSE, // wait until one of the events becomes signaled
            INFINITE);
            // Waiting until hCommandProcessed or hStopCommandGot becomes signaled. Initially
            // hCommandProcessed  is signaled, so at the beginning WaitForMultipleObjects()
            // returns immediately with WaitResult equal with WAIT_OBJECT_0 + 1.
        if (WaitResult == WAIT_OBJECT_0)
            return 0;  // Stop command, i.e. hStopCommandGot is signaled
    else if (WaitResult == WAIT_OBJECT_0 + 1)
    { // If the signaled event is hCommandProcessed, the WaitResult is WAIT_OBJECT_0 + 1
       _tprintf(_T("Insert command\n"));
       if (!ReadConsole(hStdIn, CommandBuf, 80, &nReadChars, NULL))
       {  // The problem is that when we already are in this function, the only way to leave it
          // is to type something and then press ENTER. So we cannot step into this function at
          // any moment. WaitForMultipleObjects() prevents it.
          _tprintf(_T("ReadConsole() failed, error %d\n"), GetLastError());
          return 1;
       }
       CommandBuf[nReadChars - 2] = 0;
                                 // The command in buf ends with "\r\n", we have to get rid of them
       ResetEvent(hCommandProcessed);
       // Set hCommandProcessed to non-signaled. Therefore WaitForMultipleObjects() blocks
       // the keyboard thread. When the main thread has ended the analyzing of command, it
       // sets hCommandprocessed or hStopCommandGot to signaled and the keyboard thread
       // can continue.
       SetEvent(hCommandGot);
       // Set hCommandGot event to signaled. Due to that WaitForSingleObject() in the main
       // thread returns, the waiting stops and the analyzing of inserted command may begin
    }
    else
    {  // waiting failed
       _tprintf(_T("WaitForMultipleObjects()failed, error %d\n"), GetLastError());
       return 1;
    }
 }
 return 0;
```

```
}
//                    COM1 READING THREAD
void __cdecl ReadComPort(void* pArguments)
{
   // Preparations
   HANDLE hCOM1;          // serial port COM1 as file
   OVERLAPPED Overlapped;
   BYTE *pBuf = NULL;
   DWORD nReadBytes;
   memset(&Overlapped, 0, sizeof Overlapped);
   Overlapped.hEvent=CreateEvent(NULL, FALSE, FALSE, NULL);
                                                  // event "ReadFile() has returned"
   HANDLE ComPortEvents[2];
   ComPortEvents[1] = Overlapped.hEvent;
   ComPortEvents[0] = hStopCommandGot;
   __try
   {
      hCOM1 = CreateFile(_T("\\\\.\\COM1"), GENERIC_READ, 0, NULL,
                              OPEN_EXISTING,  FILE_FLAG_OVERLAPPED, NULL);
      if (hCOM1 == INVALID_HANDLE_VALUE)
      {
         _tprintf(_T("Unable to open COM1\n"));
         return;
      }
      pBuf = new BYTE[1024];
      if (!pBuf)
      {
         _tprintf(_T("Unable to allocate buffer\n"));
```

```
    return;
  }
  if (!ReadFile(hCOM1, pBuf, 1024, &nReadBytes, &Overlapped))
  {  // Was not able to read immediately
    unsigned long int lError = GetLastError();
    if (lError != ERROR_IO_PENDING)
    {  // Serious problems
      _tprintf(_T("Unable to read from COM port, error %d\n"), lError);
      return;
    }
    else
    {  // Simply could not to finish the reading in time, let us wait
       DWORD WaitResult = WaitForMultipleObjects(2, ComPortEvents,
                                                                  FALSE, INFINITE);
       switch (WaitResult)
       {
       case WAIT_OBJECT_0:
           // Waiting stop because hStopCommandGot has become signaled, i.e.
           // the user has decided to exit
           return;
       case WAIT_OBJECT_0 + 1:
           // Waiting stopped because Overlapped.hEvent is now signaled, i.e. reading
           // operation has ended. Now we have to see how many bytes we have got.
           // nReadBuytes may have any value between 0 and 1024
           if (GetOverlappedResult(hCOM1, &Overlapped, &nReadBytes, FALSE))
              _tprintf(_T("%d bytes read from COM1\n"), nReadBytes);
           else
              _tprintf(_T("GetOverlappedResult()failed, error %d\n"), GetLastError());
```

```
                return;
            default: // Serious problems
                _tprintf(_T("WaitForMultipleObjects()failed, error %d\n"), GetLastError());
                 return;
            }
        }
    }
    else
    {  // ReadFile returned immediately. nReadBuytes may have any value between 0 and 1024
        _tprintf(_T("%d bytes read from COM1\n"), nReadBytes);
        return;
    }
  }
  __finally
  {
    CloseHandle(Overlapped.hEvent);
    if (hCOM1 != INVALID_HANDLE_VALUE)
    {
        CancelIo(hCOM1); // Cancel all pending operations with this file
        CloseHandle(hCOM1);
    }
    if (pBuf)
        delete pBuf;
        // despite of a lot "return" commands above, this is the only exit point of function
        // ReadComPort() "return" in __try{} block means that at first the __finally{} block is
        // executed and only after that the function returns
  }
}
```

Dynamic link libraries

Application consists of more that one file: *.exe + several *.dll
Implicit linking: the DLLs are connected to the application when Windows is loading the application into memory.
Explicit linking: a DLL is connected only when the application needs it calling the LoadLibrary() function.

Why the DLLs are necessary:
1. Very large executive files can be divided into smaller modules.
2. In development: each programmer (or group of programmers or company) can link his part of code (a software module) as a DLL and thus work without disturbing the other participants.
3. In maintenance: the developer changes one of the DLLs and sends it to the customer.
4. Industrial software development: one DLLs may be used in many different applications.

Important for Visual Studio users: Project properties ➔C/C++ ➔ Code generation ➔ Runtime library: multithreaded /MT (runtime support libraries are linked to the application, the total amount of *.exe is large), or multithreaded DLL /MD (runtime support libraries are applied as DLLs, the amount of *.exe is smaller but when the customer's PC does not have all the necessary libraries, the application crashes).

DLL example

```
//  COMPULSORY FILE DLLExample.h PRESENTING THE PROTOTYPES OF
//  EXPORTED FUNCTIONS
#ifdef DLLEXAMPLE_EXPORTS
// DLLEXAMPLE_EXPORTS is one of the preprocessor constants (see the project properties)
// created by wizard. The wizard includes this constant only when the project is a DLL project.
#define LIBSPEC extern "C" _declspec(dllexport)
// LIBSPEC definition used for compiling DLL source code
#else
#define LIBSPEC extern "C" _declspec(dllimport)
// LIBSPEC definition used for compiling source code applying DLL implicit linking
#endif
LIBSPEC int Sum(int x1, int x2);
// Actually the prototype for DLL (when  DLLEXAMPLE_EXPORTS is defined) is:
// extern "C" _declspec(dllexport) int Sum(int x1, int x2);
// and for application using the DLL in implicit linking mode (when
DLLEXAMPLE_EXPORTS is not defined):
// extern "C" _declspec(dllimport) int Sum(int x1, int x2);
//                          DLL ENTRY POINT (File dllmain.cpp created by wizard)
#include "stdafx.h"
BOOL APIENTRY DllMain( HMODULE hModule,
                                           DWORD  ul_reason_for_call, LPVOID lpReserved)
// DllMain() is called automatically each time when the process or one of its threads attaches
// or detaches the DLL:
// In case of implicit linking when the process starts and terminates
// In case of explicit linking when the application calls LoadLibrary() and FreeLibrary()
// functions
{
        switch (ul_reason_for_call)
        {
                case DLL_PROCESS_ATTACH:
                case DLL_THREAD_ATTACH:
                        // Operations for initialization
                case DLL_THREAD_DETACH:
                case DLL_PROCESS_DETACH:
                        // Operations for clean-up
                        break;
        }
        return TRUE;
}
```

```
//      FUNCTIONS EXPORTED BY DLL  (File DLLExample.cpp created by developer)
#include "stdafx.h"
#include "DLLExample.h"
LIBSPEC int Sum(int x1, int x2)
{
        return x1 + x2;
}
```

```
//                    APPLICATION WITH IMPLICIT LINKING
#include "stdafx.h"
#include "Windows.h"
#include "DLLExample.h"
// DLLExample.h presents the protoypes of functions the DLL exports and must be delivered
// together with DLLExample.dll. Put it into the directory where all the other *.cpp and *.h files
// are located (in this example this directory is
...\Projects\DLLImplicitUsageExample\DLLImplicitUsageExample).
// To link you need also DLLExample.lib. Put it into the same folder. And in the project
// properties in the linker input additional dependencies cell write DLLExample.lib.
// The DLLExample.lib was created together with DLLExample.dll and must be delivered with
// DLLExample.dll. It contains so called stubs – short empty functions necessary to play a trick
// on linker. When the application is running, the stubs are replaced by actual functilns stored
// in the body of DLL.
int _tmain(int argc, _TCHAR* argv[])
{
	_tprintf(_T("%d"), Sum(5, 6)); // call the imported function as any other one, no differences
	return 0;
}
// The runtime standard search order for DLLs:
// 1. The directory containing the application *.exe file. In Visual Studio project frame the
// DLLImplicitUsageExample.exe is in folder ...\Projects\DLLImplicitUsageExample\Debug.
// We copied the DLLExample.dll from ...\Projects\DLLExample\Debug into the mentioned
// folder.
// 2. The current directory (for example, set by the MS-DOS "cd" command). Rather often
// the current directory is just the directory containing the *.exe.
// 3. The Windows system directory c:\windows\system32
// 4. The Windows directory c:\windows
// 5. Directories specified by the PATH environment variable.
//
// If the DLL was not found, the application cannot start
//                    APPLICATION WITH EXPLICIT LINKING
#include "stdafx.h"
#include "Windows.h"
int _tmain(int argc, _TCHAR* argv[])
{
        // Get the DLL handle.
        // The search order is as for implicit linking
        HMODULE hDLL = LoadLibrary(_T("DLLExample.dll"));
        if (hDLL == NULL)
        {
                _tprintf(_T("DLLExample.dll not found, error %d"), GetLastError());
                return 1;
        }
        // Get the pointer to the function you want to import
        FARPROC pSum = GetProcAddress(hDLL, "Sum");
                                        // only ANSI, _T("..") is not allowed
        if (pSum == NULL)
        {
                FreeLibrary(hDLL);
                _tprintf(_T("Function Sum() not found, error %d"), GetLastError());
                return 1;
        }
        // Call the imported function
         int result = ((int(*)(int, int))pSum)(5, 6);
        _tprintf(_T("%d"), result);
        // Detach the DLL
        FreeLibrary(hDLL);
        return 0;
}
// The runtime standard search order for DLLs:
// 1. The directory containing the application *.exe file. In Visual Studio project frame the
// DLLExplicitUsageExample.exe is in folder
// ...\Projects\DLLExplicitUsageExample\Debug. We copied the DLLExample.dll from
// ...\Projects\DLLExample\Debug into the mentioned folder.
// 2. The current directory (for example, set by the MS-DOS "cd" command). Rather
// often the current directory is just the directory containing the *.exe.
// 3. The Windows system directory c:\windows\system32
// 4. The Windows directory c:\windows
// 5. Directories specified by the PATH environment variable.
```