## X. Automatizarea testării cu ajutorul macrourilor

**X.1. Elementele pachetului AutoIt 3**

**AutoIt** este un limbaj de programare a macrourilor Windows cu o sintaxă moştenită din familia limbajelor BASIC. Un **macro** este o succesiune de acţiuni ale utilizatorului efectuate asupra **interfeţei grafice**, de regulă o combinaţie de evenimente declanşate cu ajutorul mouse-ului şi tastaturii. Un limbaj de programare a macrourilor, precum AutoIt, oferă structuri moştenite de la limbajele de programare cu ajutorul cărora macrourile pot fi create la un nivel de rafinament şi putere de prelucrare superioară celor oferite de **instrumentele de înregistrare şi redare a macrourilor** (MRP[[1]](#footnote-1)). Exemplele de MRP sunt numeroase atât la nivel de sistem de operare, cât şi la nivel de aplicaţie. Instrumente MRP sunt incluse în pachetul Microsoft Office (macrourile Word, Excel, etc.) - limbajul care stă la baza lor este VBA[[2]](#footnote-2). Mediul de programare Visual Fox oferă la rândul său un instrument MRP în cadrul aplicaţiei Automated Test Harness, instalată în pachetul Visual Fox auxiliar. Diverse sisteme de operare permit crearea de scripturi cu rol de macrouri iar pentru Windows s-a consacrat limbajul AutoIt, al cărui interpretor şi documentaţie sunt disponibile gratuit la adresa www.autoitscript.com.

Macrourile[[3]](#footnote-3) sunt un tip particular de programe cu o algoritmică de complexitate redusă şi care nu pun accentul pe procesarea de date, ci pe simularea comportamentului utilizatorului în scopul reproducerii sale. Algoritmul unui macro pune accent pe structura secvenţială din programarea clasică, mai precis pe secvenţialitatea operaţiilor declanşate în GUI[[4]](#footnote-4) în diferite scopuri: poate fi vorba de înregistrarea unei secvenţe reutilizabile de operaţii în cadrul unei aplicaţii (vezi macrourile MS Office) sau de secvenţe de automatizare a unor procese de la nivelul sistemului de operare (executarea repetitivă a unor aplicaţii, planificări de tip batch sau Task Scheduling, testarea automatizată a produselor software). Desigur, aspectul care justifică necesitatea macrourilor este **repetabilitatea** unui set de operaţii, aşadar *reproducerea comportamentului utilizatorului* denumită în mod generic **automatizare GUI**[[5]](#footnote-5). Deşi în mod tradiţional macrourile sunt secvenţe de operaţii, limbaje precum AutoIt permit programarea macrourilor la un nivel avansat, cu structuri de programare, variabile şi mecanisme de dialog cu utilizatorul, oferind chiar şi acces la regiştrii Windows sau la obiecte COM.

Operaţiile de bază pe care le simulează AutoIt sunt apăsarea tastelor, folosirea mouse-ului şi manipularea ferestrelor Windows. Iniţial, limbajul a fost conceput pentru automatizarea instalărilor şi configurărilor software de către administratorii de sistem care trebuie să instaleze în mod identic sute de PC-uri. Versiunea 3 a limbajului propune o sintaxă inspirată din limbajele BASIC (VBScript în special), suportă expresii complexe, funcţii ale utilizatorului şi structuri de programare tradiţionale. Rezumăm în continuare elementele de noutate ale versiunii 3:

* Macrourile AutoIt 3 pot fi executate prin interpretare sau compilate prin instrumentul **Aut2Exe**;
* S-a creat o versiune ActiveX + DLL numită **AutoItX** care permite integrarea AutoIt în alte limbaje de programare;
* Macrourile AutoIt 3 sunt stocate în fişiere de tip au3;
* Macrourile pot extrage date din obiecte GUI tradiţionale: casete de editare, check box, liste, butoane, etc. şi chiar obiecte GUI din alte ferestre decât cea activă;
* Macrourile pot folosi structuri IF, CASE, WHILE, FOR, funcţii ale utilizatorului şi expresii complexe, pste 25 de funcţii de manipulare a şirurilor de caractere şi date de tip numeric;
* Macrourile pot accesa ferestre Windows prin titlul ferestrei sau prin clasă.

Structura de directoare creată la instalarea pachetului AutoIt complet conţine elementele:

* AutoIt3.exe – interpretorul propriu-zis, nu are o interfaţă grafică proprie ci poate fi lansat din linia de comandă sau prin dublu-clic (care va solicita deschiderea unui macro au3)
* AutoIt3A.exe – versiunea AutoIt pentru Windows 98
* AU3Info.exe – **AutoIt Window Info Tool (AWIT)**, instrument capabil să extragă informaţii de la fereastra activă – titlul, conţinutul barei de stare, poziţia, dimensiunea, poziţia cursorului, culoarea pixelului de la poziţia cursorului, date privind obiectul GUI de la poziţia cursorului. Fereastra AU3Info este deschisă în regim Always on Top şi oferă informaţiile respective în timp real, simultan cu manevrarea mouse-ului în fereastra vizată. Informaţiile afişate de AWIT pot fi apoi folosite în cadrul macrourilor, fiind copiate în Clipboard prin dublu clic pe elementul dorit. Evident, copierea prin dublu clic nu va fi posibilă simultan cu mişcarea mouse-ului în fereastra vizată (datele legate de poziţia cursorului se modifică în timp real), de aceea instrumentul AWIT oferă o opţiune Freeze (Ctrl-Alt-F), pentru îngheţarea temporară a informaţiilor AWIT
* AU3InfoA.exe – versiunea AWIT pentru Windows 98
* AU3Check.exe – corectorul sintactic
* AutoIt.chm – documentaţia Help
* Psapi.dll – DLL necesar funcţiilor sub Windows NT 4
* Uninstall.exe – programul de dezinstalare
* AutoIt v3 Website.url – shortcut spre site-ul AutoIt
* Directorul Aut2Exe:
  + Directorul Icons cu pictogramele ce pot fi asociate extensiei au3
  + Aut2Exe – compilatorul pentru convertirea macrourilor în executabile
  + Aut2ExeA.exe – versiunea Windows 98 a compilatorului
  + AutoItSC.bin – stub executabil pentru macrourile compilate
  + UPX.exe – comprimă macrourile compilate
* Directorul Examples
  + Directorul GUI – exemple de macrouri GUI
  + Helpfile – exemple de macrouri folosite în exemplele din Help

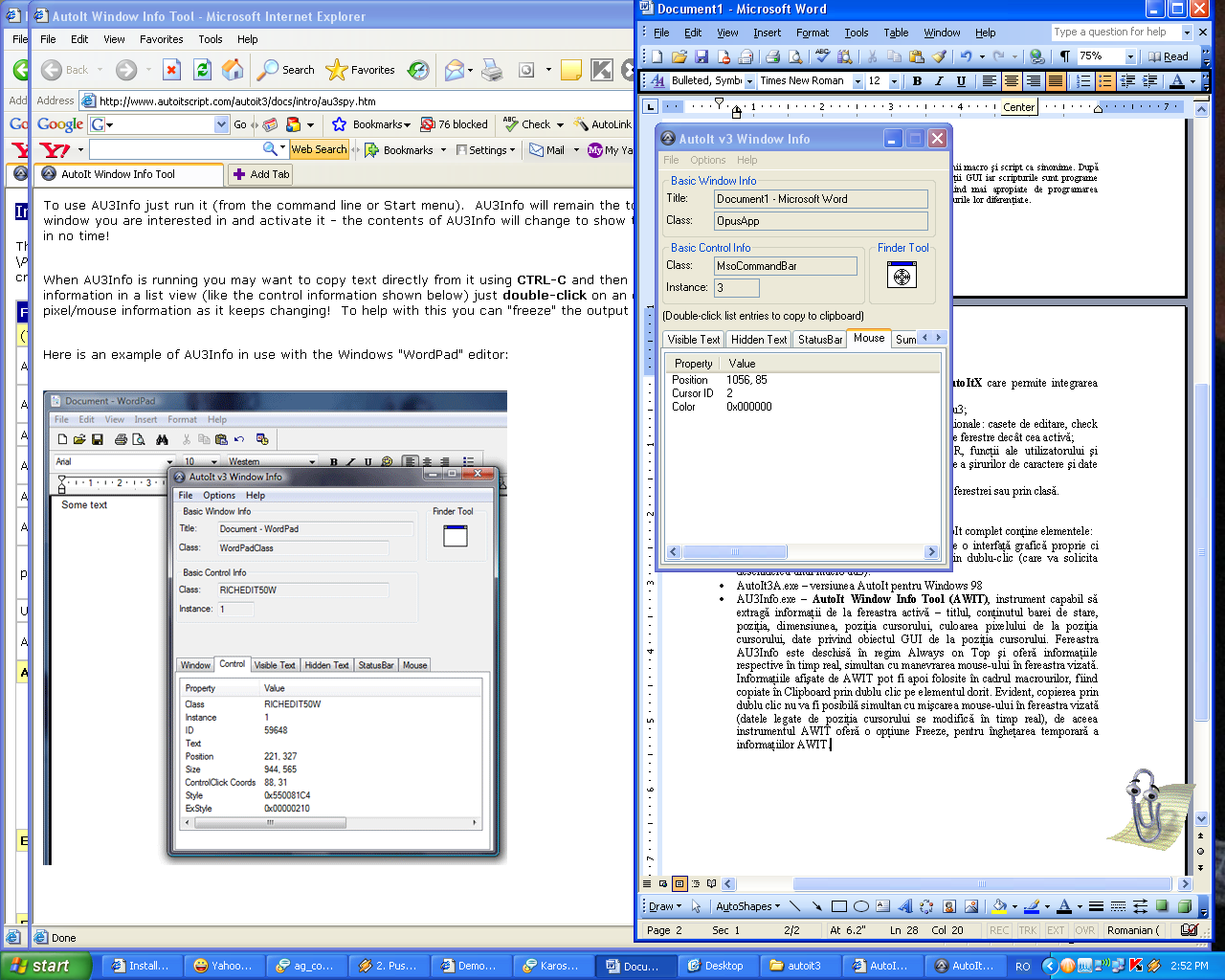


Fig. 1. Instrumentul AWIT, pagina de afişare a informaţiilor privind poziţia cursorului[[6]](#footnote-6)

* Directorul Extras
  + Directorul AutoUpdateIt – conţine un script de actualizare automată a versiunii de AutoIt
  + Directorul Editors – conţine definiţiile schemelor de colorare a sintaxei pentru editoare de text populare
  + Directorul Exe2Aut – conţine decompillatorul de macrouri, pentru obţinerea codului sursă din macrouri executabile
  + Directorul SQLite – conţine un executabil şi un help pentru lansarea prin linia de comandă
  + Directorul V2\_to\_v3\_Converter – conţine convertorul de macrouri de la versiunea anterioară la versiunea 3
* Directorul Icons – conţine pictograme pentru extensia au3
* Directorul Include – conţine biblioteca de funcţii predefinite ale utilizatorului
* Directorul AutoItX – conţine versiunea DLL a interpretorului ce oferă o interfaţă ActiveX/COM+DLL
* Directorul SciTE – conţine o versiune simplificată a editorului de macrouri **SciTE.** Versiunea completă a editorului poate fi descărcată gratuit de pe site-ul AutoIt şi instalată peste versiunea implicită din pachetul AutoIt. Deşi macrourile pot fi editate cu orice editor de texte, SciTE oferă facilităţi de colorare a sintaxei şi detectare a erorilor sintactice.

**X.2. Editorul SciTE**

Editorul de macrouri SciTE este la bază un editor de texte rezultat din proiectul open source SCIntilla, optimizat pentru editarea de cod sursă indiferent de limbaj (C++, PHP, Python etc.). Aceasta implică suport pentru stilizarea codului sursă, urmărirea liniilor de cod, indicatori de eroare şi sugestii de editare. Opţiunile de stilizare se referă la formatarea personalizată a componentelor codului sursă. Instrumentele oferite de versiunea completă a SciTE (numită SciTe4AutoIt, neinclusă în pachetul AutoIt dar disponibilă gratuit la aceeaşi adresă), sunt:

* Formatare implicită a codului AutoIt, inclusiv cu indentare automată;
* Sistem Help contextual (activat prin F1 la selectarea unei linii de cod);
* Tabel de funcţii pentru acces rapid la sintaxă şi help (Shift-F1);
* **AutoComplete**, instrument de sugerare în timpul editării codului sursă, pentru tastarea de instrucţiuni dar şi pentru numele funcţiilor sau variabilelor;
* **Intellisense**, instrument de sugerare a parametrilor funcţiilor;
* **AU3Spy**, instrument ce detectează nume şi valori ale obiectelor GUI;
* **CodeWizard, Koda** şi **GUIBuilder** instrumente de asistenţă în crearea dialogurilor cu utilizatorul şi a formularelor;
* **Tidy**, instrument precompilator cu rol de asigurare a lizibilităţii codului sursă: indentarea instrucţiunilor, convertirea cuvintelor cheie în majuscule, încărcarea funcţiilor utilizatorului (din Userfunctions.txt) în tabelul editorului, formatarea numelor de variabile, eliminarea spaţiilor redundante, generarea de comentarii care să indice numele funcţiei la fiecare final de funcţie AutoIt (Endfunc), generarea unei documentaţii cu logica programului şi rapoarte privind variabilele, detectarea unor erori de sintaxă, sortarea blocurilor Func-Endfunc (definiţiile funcţiilor). Fişierul Tidy.ini permite activarea sau dezactivarea individuală a acestor funcţionalităţi Tidy.
* **Obfuscator**, instrument precompilator cu rol de optimizare a unor elemente din codul sursă, creează un nou macro din codul sursă iniţial cu modificări precum: redenumirea variabilelor şi funcţiilor, conversia unor valori literale de tip string în variabile, conversia valorilor literale numerice în apeluri ale funcţiei Number, conversia referinţelor la alte macrouri în apeluri ale funcţiei Execute(), generarea de avertismente legate de o serie de funcţii a căror utilizare nu e recomandată, generarea unui jurnal cu modificările aduse.
* **AU3Check (SyntaxChecker)** este corectorul sintactic, precompilator.
* **AU3Recorder (Scriptwriter)** şi **AutoIt Macro Generator** sunt două instrumente MPR asociate mediului AutoIt, capabil să înregistreze activitatea mouse-ului şi tastaturii şi să genereze un macro AutoIt. Instrumentele sunt produse auxiliare create de autori diferiţi şi sunt frecvent folosite pentru a construi mai rapid acele părţi din macrouri complexe care nu necesită rafinare prin programare.
* **AutoIt3Wrapper** este o aplicaţie care integrează interpretorul şi compilatorul AutoIt într-o interfaţă grafică ce oferă opţiuni de precompilare şi postcompilare, cum ar fi activarea sau dezactivarea instrumentelor precompilatoare (Tidy, Obfuscator, AU3Check - acestea pot fi rulate şi ca executabile independente ce primesc un macro la intrare). Printre altele aplicaţia indică parametrii cu care pot fi executate instrumentele de precompilare, citeşte sau modifică directive implicite din fişiere .ini, execută diverse programe la precompilare sau postcompilare (Run Before/After), oferă posibilitatea de întrerupere şi reluare a compilării în funcţie de erorile detectate. Practic, aplicaţia AU3Wrapper permite utilizatorului să perceapă editorul SciTE ca un mediu de programare şi generare a proiectelor AutoIt ce oferă acces în mod interactiv la toate instrumentele AutoIt şi la numeroase instrumente auxiliare create de comunitatea care suportă acest limbaj.
* **Snippet holder,** un instrument de snippet management**[[7]](#footnote-7)**
* **Scripturi LUA** – LUA este un metalimbaj de programare multi-paradigmă (neconstrâns de tipul de problemă abordată) creat ca extensie pentru programarea procedurală şi obiectuală. LUA se aseamănă cu LISP şi XML prin aceea că e un limbaj minimalist şi generic care foloseşte o singură structură de date, metatabelul, pentru a implementa majoritatea structurilor tradiţionale (clase, obiecte, spaţii de nume, liste, arbori). LUA se aseamănă cu JavaScript prin aceea că e un limbaj interpretat care nu îşi propune să construiască programe de sine stătătoare, ci funcţii-script care să fie incluse şi executate (prin API[[8]](#footnote-8)) în programe gazdă (de regulă programe C, Java şi .NET, frecvent în domeniul jocurilor PC). Mediul AutoIt poate folosi o serie de funcţii LUA predefinite pentru eficientizarea procesului de editare prin SciTE[[9]](#footnote-9). Exemple frecvente de apelare a funcţiilor LUA sunt generarea sau eliminarea de comentarii la casetele de dialog, copierea liniilor de cod marcate anterior ca semne de carte, activarea unor mesaje de consolă pentru depanare etc. grupate în a doua jumătate a meniului Tools din SciTE[[10]](#footnote-10).
* **SciTE Config** permite configurarea schemei de colorare a sintaxei şi personalizarea mediului de lucru. Configurările specifice variatelor limbaje pentru care SciTE poate fi folosit ca editor de cod optimizat se realizează în lista fişierelor de configurare accesibile în meniul Options. Se poate remarca astfel versatilitatea SciTE, care poate deservi limbaje ca HTML, SQL, C, PASCAL, PYTHON şi multe altele. În materialul de faţă au relevanţă doar configurările privind AutoIt.
* **Resource Hacker,** un program ce oferă acces la diverse resurse[[11]](#footnote-11) din executabile Windows (pictograme, forme ale cursorului, sunete, casete de dialog, fonturi, meniuri). Poate fi utilizat ca instrument precompilator pentru adăugare, modificare sau ştergere de resurse în cadrul AutoIt3Wrapper dar nu e instalat implicit, fiind un produs third party disponibil la adresa http://angusj.com/resourcehacker/.

O parte din aceste instrumente sunt implicit active, majoritatea sunt disponibile în meniul Tools iar instrumentele precompilatoare pot fi gestionate şi cu ajutorul AutoIt3Wrapper, care se execută implicit prin opţiunea Tools - Compile. În plus, unele instrumente apar în două variante, Prod şi Beta. Prod corespunde versiunii AutoIt curente stabile (versiunea 3 la momentul scrierii acestui material) iar Beta corespunde versiunii AutoIt curente beta. Elementele ce pot rula diferenţiat după cele două versiuni sunt corectorul sintactic, helpul, compilatorul, lista funcţiilor, Tidy. Instrumentul **Switch Definitions[[12]](#footnote-12)** permite selectarea versiunii implicite la care să se raporteze aceste instrumente, precum şi actualizarea definiţiilor acestor versiuni conform site-ului oficial. În cele ce urmează materialul este dedicat versiunii Prod curente (3.2.8)

**Operaţii de bază în SciTE**

**Deschiderea unui macro AutoIt** se poate realiza la dublu clic pe un fişer de tip .au3. **Rularea** sa (cu corectare sintactică implicită) se poate realiza tot prin dublu clic[[13]](#footnote-13) sau cu ajutorul lui F5 sau Tools-Go (pentru versiunea Prod).

**Crearea** unui macro AutoIt se poate realiza în Windows Explorer din meniul contextual (lista New va fi actualizată cu tipul de fişiere AutoIt la instalarea pachetului) sau cu opţiunea File-New.

**Compilarea** în executabil a unui macro se realizează prin F7 sau Tools-Compile.

**Corectarea sintactică fără rulare** se realizează cu Ctrl-F5 sau Tools-Syntax Check Prod.

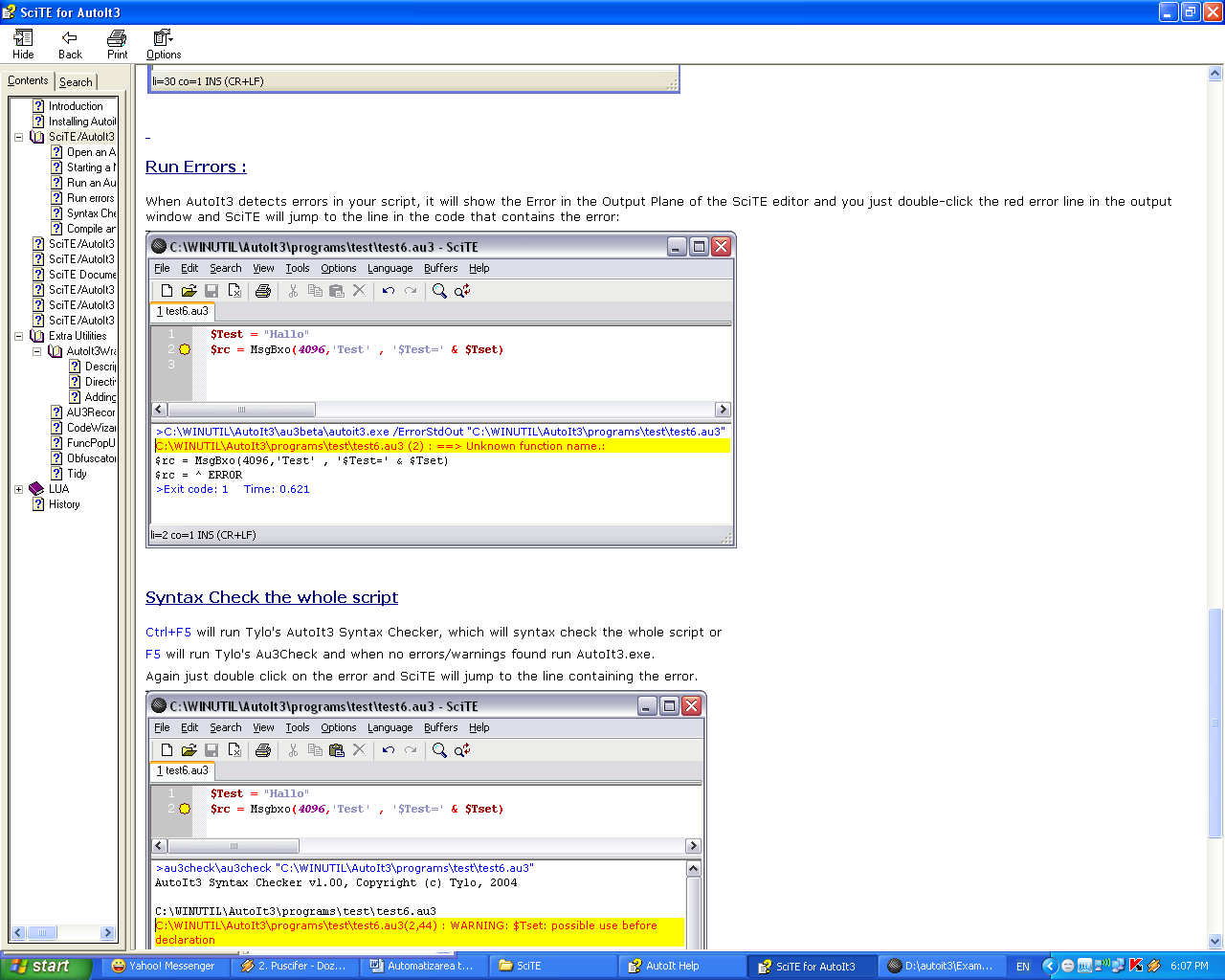


Fig. 2. Semnalarea erorilor de către corectorul sintactic SciTE []

**Mediul de lucru şi editarea în SciTE** sunt afectate de o serie de instrumente ale editorului:

* SciTE poate găzdui (edita, executa, interpreta şi compila) coduri sursă scrise într-o multitudine de limbaje de programare. Selecţia limbajului curent (şi a regulilor aferente) se realizează prin meniul Languages. Recunoaşterea limbajului se realizează pe baza extensiei fişierelor în lucru, definite în fişierele de configurare accesibile prin meniul Options. Executarea compilatoarelor şi interpretoarelor aferente se face automat (pe baza sistemului linie de comandă), dacă acestea au fost în prealabil instalate[[14]](#footnote-14);
* Instrumentele de sugestie oferă pentru selecţie în timpul editării lista cuvintelor cheie, a numelor de funcţii sau ordinea parametrilor unei funcţii;
* Sintaxa e evidenţiată prin colorarea elementelor codului sursă după o schemă de culori selectată prin Tools-SciTE Config;
* SciTE permite restrângerea sau extinderea structurilor imbricate (If-EndIf, Do-Until, While-Wend, #region-#endregion, blocuri de comentarii) pentru condensarea codului sursă şi derularea sa facilă;
* SciTE oferă un mecanism Find-Replace avansat, comparabil cu cel din Word;
* Instrumentul Ouput, în partea de jos a ferestrei, indică în timpul executării unui macro (sau program în limbajele găzduite de SciTE) care sunt ieşirile, care sunt fazele execuţiei sau compilării şi chiar mesajele de eroare ale compilatorului de la bază. Zona Output poate fi resetată cu Tools-Clear Output;
* Meniul Tools oferă o serie de opţiuni pentru întreruperea execuţiei (Stop Executing), precum şi pentru navigarea mesajelor de eroare (Next Message, Previous Message);
* SciTE îşi execută instrumentele (inclusiv compilatorul sau interpretorul pe care îl găzduieşte) prin comenzi adresate în linia de comandă. Compilatorul sau interpretorul care trebuie solicitat e recunoscut conform extensiilor fişierelor (declarate în fişierele de configurare specifice limbajelor găzduite din meniul Options). Fiind vorba de execuţie în linie de comandă macroul (programul) rulat nu va fi deschis într-o fereastră proprie. Pentru a modifica acest mod de execuţie implicit, variabila *command.go.subsystem* poate primi valorile:
  + 0, rulare în linia de comandă,
  + 1, programe ce îşi vor crea propria fereastră,
  + 2, rulare prin shell (Windows Explorer), deci similar cu executarea prin dublu clic în interfaţa sistemului de operare (recomandat pentru pagini HTML care nu îşi creează propria fereastră),
  + 3, rularea unei extensii,
  + 4, rularea prin sistemul HTMLHelp din Windows,
  + 5, rularea prin funcţia WinHelp.

Aceste valori pot fi atribuite în oricare din fişierele cu extensia .properties, care fixează condiţii implicite ale mediului de lucru (inclusiv variabile de mediu) la diferite nivele:

* + diferenţiat în fişierele de configurare a limbajelor din meniul Options (html.properties, cpp.properties, au3.properties etc.)
  + în Options-Open Global Options File (configurare iniţializată global),
  + în Options-Open User Options File (configurare specifică unui utilizator, o supradefineşte pe cea globală),
  + în Options-Open Directory Options File (configurare locală aplicabilă pentru fişierele din directorul în care e salvat SciTEDirectory.properties şi subdirectoarele sale, o supradefineşte pe cea a utilizatorului),
  + în Options-Open Local Options File (configurare locală aplicabilă pentru fişierele din directorul în care s-a salvat fişierul SciTE.properties, supradefineşte celelalte configurări)

Fişierul global de configurare, SciTEGlobal.properties, poate conţine o directivă

de anulare a configurărilor locale (ex: properties.directory.enable cu valori booleene). Se observă că toate aceste fişiere sunt de tip text şi conţin perechi nume=valoare pentru variabilele de mediu[[15]](#footnote-15). Configurările specifice platformei Windows sunt definite prin directiva condiţională IF PLAT\_WIN ....

* SciTE permite exportarea codului sursă în format PDF, Latex, XML, HTML, RTF;
* SciTE poate întreţine 100 de buffere simultan în memoria internă, fiecare cu propriul fişier. Lista lor e accesibilă în meniul Buffers, trecerea de la unul la altul realizându-se prin sistemul consacrat al Tab-urilor. Totalitatea bufferelor active (cu fişierele deschise) la un moment dat alcătuiesc o **sesiune**. Sesiunea poate fi salvată şi reîncărcată, ceea ce permite deschiderea simultană a mai multor fişiere (File-Load Session şi File-Save Session);
* Editarea în SciTE poate folosi abrevieri pentru mărirea vitezei de construire a codului sursă. Structuri complexe pot fi abreviate similar cu mecanismul AutoCorrect din Word, pe baza unei tabele cu perechi abreviere=expandare. Fişierul cu definirea acestor perechi e abbrev.properties şi poate fi deschis pentru editare cu Options-Open Abbreviations File. Sunt uzuale situaţiile în care o abreviere de 2-3 caractere e folosită pentru a genera o structură de programare, o definiţe de funcţie sau un obiect GUI. Abrevierile nu pot conţine semnul =. Expandările pot conţine caracterul de salt la rând nou "\n" şi caracterul de inserţie "|" care indică zone din codul sursă generat în care trebuie realizate completări (ex: abrevierea **fornext** va genera o structură FOR cu contorul i, pasul 1 şi un punct de inserţie acolo unde urmează să se completeze blocul repetitiv dde instrucţiuni din cadrul structurii FOR.). Recomandăm consultarea abrevierilor predefinite şi crearea de noi abrevieri pentru porţiuni de cod sursă frecvent reutilizate.

**X.3. AutoIt - Elemente de limbaj**

AutoIt foloseşte o sintaxă de tip BASIC, care implementează conceptele de bază ale limbajelor de programare structurată slab tipizate şi slab obiectualizate:

* Tip de date unic şi variant (o dată îşi poate schimba tipul prin conversii implicite provocate de operatori);
* Variabile şi constante;
* Macrovariabile (variabile de mediu, macrouri predefinite, caractere speciale);
* Operatori numerici, string, logici;
* Structuri de programare (condiţionale, repetitive) şi forţarea ieşirilor din structurile de programare;
* Comentarii;
* Directive;
* Funcţii:
  + Funcţii predefinite (incluse în pachet şi recunoscute de interpretorul AutoIt),
  + Funcţii ale utilizatorilor (create de programatorii AutoIt şi salvate în cadrul macrourilor),
  + Funcţii predefinite ale utilizatorilor (create de diverşi utilizatori, dar ataşate pachetului AutoIt sub forma bibliotecilor de funcţii din directorul Include; sunt recunoscute de interpretor cu condiţia să se includă în macro biblioteca aferentă);
  + Funcţia Opt, are rolul de a configura funcţionarea interpretorului.

Fiecare instrucţiune se tastează pe câte un rând. E posibil ca o linie de cod să fie prelungită pe mai multe rânduri, dacă linia întreruptă se finalizează cu perechea de caractere " \_" (spaţiu urmat de underscore). O linie de cod sursă poate fi întreruptă în orice punct, cu excepţia conţinutului şirurilor de caractere. Aşadar, nu e posibilă construcţia:

MsgBox (0, "titlu", "acesta este & \_

un şir de caractere întrerupt")

În schimb, se va folosi o concatenare pentru a face posibilă întreruperea:

MsgBox (0, "titlu", "acesta este" & \_

"un şir de caractere întrerupt")

Observaţie: pentru a testa valorile variabilelor, expresiilor, funcţiilor şi macrovariabilelor din exemplele ce urmează, se recomandă ca la finalul fiecărui exemplu să se genereze o casetă de dialog care să asigure vizibilitatea rezultatului, folosind după caz:

MsgBox(0,"",$numevariabila)

MsgBox(0,"",expresie)

MsgBox(0,"",@macrovariabila)

MsgBox(0,"",numefunctie())

**X.3.1. Tipizarea datelor AutoIt**

AutoIt face parte din rândul limbajelor slab tipizate. Teoretic, limbajul oferă un singur tip de date, numit **variant**. Practic, o dată variantă poate conţine o valoare numerică (zecimală sau hexazecimală), booleană, un şir de caractere (string) sau un cod hexazecimal. Comportamentul datelor de tip variant e contextual – diferă în funcţie de valoarea datei şi de modul în care folosită. Exemple:

* Expresia 2\*2 va fi considerată numerică, cu rezultatul 4, deoarece operaţia \* este disponibilă doar pentru valori numerice.
* Expresia 2\*"2" va fi considerată numerică, cu rezultatul 4, deoarece limbajele netipizate oferă un mecanism de **conversie implicită** la tipul sugerat de operator.
* Expresia "2"\*"2" va fi considerată numerică, cu rezultatul 4, datorită aceluiaşi mecanism de conversie implicită la tipul sugerat de operatorul \*
* Expresiile 2&2, "2"&"2" vor fi considerate expresii de tip text, datorită conversiei implicite provocate de operatorul de concatenare.
* Expresia True+100 va avea valoarea 101, datorită conversiei implicite de la boolean la numeric (True=1).

Denumirea tipului variant sugerează faptul că o aceeaşi dată îşi poate schimba tipul fără conversii explicite. Practic, putem afirma că limbajul lucrează cu patru tipuri de date – numeric, string, boolean şi hexazecimal, dar că tipul acestora este stabilit pe baza operatorilor din expresiilor în care sunt antrenate.

Conversia implicită nu provoacă niciodată erori – dacă şirul nu poate fi interpretat numeric, rezultatul conversiei va fi 0, ca în exemplul expresiei 2\*"a"=2\*0=0. Valorile booleene sunt True şi False (False are alternativele 0 – numeric- şi "" - string).

În cele ce urmează se va utiliza frecvent exprimearea consacrată *valoare literală[[16]](#footnote-16)*. Aceasta semnifică o valoare care e tastată explicit în codul sursă şi nu e reprezentată de o variabilă, nu e rezultatul unei expresii sau al unei funcţii. Exemplu: 2 + $a e o expresie care adună o valoare literală (2) cu valoarea unei variabile ($a).

**Datele numerice** pot fi folosite în următoarele variante de valori literale:

* în format zecimal (întregi pozitivi, negativi sau numere cu virgulă zecimală);
* în format exponenţial (2e3 reprezintă 2\*103 = 2000);
* în format hexazecimal.

**Datele de tip text (string)** pot fi folosite:

* în format delimitat cu ghilimele (ghilimelele conţinute în valoarea datei se dublează);
* în format delimitat cu apostroafe (dacă data conţine apostrof, acesta se dublează);
* în format sensibil la construcţiile speciale.

Cei doi delimitatori pot fi utilizaţi alternativ pentru a elimina necesitatea dublării (ghilimele la delimitare şi apostroafe în valoarea datei sau invers), ca în exemplul:

"Ion a spus 'Salut' si a plecat" (alternativă la "Ion a spus ""Salut"" si a plecat")

În mod normal, valorile de tip text sunt tratate literal (ca un şir de caractere tastate direct în codul sursă). Există posibilitatea de a modifica modul de tratare a lor, pentru a putea **detecta şi substitui construcţii speciale** care apar în cadrul şirurilor de caractere, cu ajutorul funcţiei Opt().

Spre exemplu, presupunem că variabila $a are valoarea 2.

* O valoare de tip text "a$a$" va fi tratată în mod normal ca un şir conţinând cele patru caractere dintre ghilimele;
* Funcţia Opt ("ExpandVarStrings", 1) va activa substituirea construcţiilor speciale, iar valoarea datei va deveni "a2", prin înlocuirea construcţiei speciale $a$ cu valoarea variabilei $a. Dacă s-a activat substituirea variabilelor în şiruri de caractere şi totuşi se doreşte utilizarea semnului $ ca un caracter oarecare (fără a provoca o substituire), prima apariţie a sa în construcţia specială trebuie dublată: "a$$a$".

Construcţiile speciale care pot fi substituite cu ajutorul funcţiei Opt sunt variabilele obişnuite (delimitate în text cu $), variabilele de mediu (delimitate în text cu %...%) şi macrovariabilele (delimitate în text cu @).

Exemple:

$a=2

Opt ("ExpandVarStrings", 1)

S-a atribuit valoarea numerică 2 variabilei a şi s-a activat substituirea de variabile în şiruri de caractere.

* Expresia "a$a" & "$a$" va avea valoarea "a$a2". Se poate concluziona că substituirea a avut loc înaintea concatenării. Aşadar, expresia a fost mai întâi convertită la "a$a" & "2" (primul caracter $ nu a provocat o substituire deoarece nu are un caracter pereche).
* Expresia "a$a$a$" va avea valoarea "a2a$".
* Expresia "a$$a$$a$$" va avea valoarea "a$a$a$". Se poate concluziona că substituirea are loc de la stânga spre dreapta:
  + substituirea primei construcţii "$$a$" e dezactivată prin dublarea semnului de la poziţia 2, deci se obţine în prima fază "a$a**$$a$**$"
  + mergând spre dreapta, apare o a doua construcţie $$a$ (porţiunea îngroşată, aflată acum la poziţia 4), deci se obţine rezultatul "a$a$a**$$**"
  + în final, ultima pereche de $ (acum la poziţia 6) e redusă la un singur $.
* Expresia "a$$a$a$" va avea valoarea "a$a2". Din nou, are loc aceeaşi substituire de la stânga la dreapta:
  + prima substituire (poziţia 2) e dezactivată, deci se obţine în prima fază "a$a**$a$**"
  + mergând spre dreapta, apare o a doua construcţie $a$ care va fi substituită (porţiunea îngroşată, acum la poziţia 4), deci se obţine rezultatul "a$a2".

Substituirile nu au loc, dacă valoarea variabilei nu a fost anterior definită (prin iniţializare, diverse atribuiri etc.)!

Dezactivarea senzitivităţii la construcţii speciale se realizează prin apelul

Opt ("ExpandVarStrings", 0)

Este evident faptul că senzitivitatea la construcţii speciale este o alternativă eficientă la concatenarea valorilor de variabile. Spre exemplu, următoarele două expresii au aceeaşi valoare:

* "a$a$" cu senzitivitatea la construcţii speciale activată;
* "a" & $a, indiferent dacă senzitivitatea a fost activată sau nu.

**Datele booleene** pot avea valorile **True** şi **False.**

* La conversia implicită string – boolean, şirul vid devine False, orice alt şir devine True (inclusiv şirul "False" sau şirul "0"!)
* La conversia implicită boolean – string, cele două valori booleene devin "False" şi "True"
* La conversia implicită numeric – boolean, 0 devine False, orice alt număr devine True
* La conversia implicită boolean – numeric, False devine 0, True devine 1

Exemple:

* Expresia "False" AND True va avea valoarea True
* Expresia "" AND True va avea valoarea False
* Expresia 1 OR "" va avea valoarea True
* Expresia 1+True va avea valoarea 2
* Expresia 1+"True" va avea valoarea 1 (conversie eşuată la al doilea operand)
* Expresia True & True va avea valoarea "TrueTrue"

**Datele hexazecimale** se împart în categoriile:

* *numere hexazecimale*, obţinute prin conversia unor date numerice întregi în baza de numeraţie 16;
* *coduri hexazecimale*, exemple: conversia unor caractere în coduri numerice, convertite apoi în baza de numeraţie 16 (coduri ASCII sau Unicode), codurile de culoare pe 3 octeţi.

Datele hexazecimale sunt marcate la stânga de combinaţia 0x, urmată de valoare. Numerele hexazecimale sunt limitate la 4 octeţi (4 perechi de cifre hexazecimale), codurile hexazecimale pot depăşi această limită. Toţi operanzii hexazecimali sunt consideraţi numere hexazecimale. Codurile hexazecimale apar doar în funcţiile de conversie. Exemplu:

0xFF+1

...primul operand va fi considerat un număr hexazecimal (e valoare literală);

Binary("abc")+1

...primul operand va fi considerat un cod hexazecimal.

Un *număr hexazecimal* se obţine prin conversia unui număr zecimal în baza 16.

Un *cod hexazecimal* se obţine prin diverse metode de codificare, pornind de la ideea că orice cod este reprezentat pe un număr întreg de octeţi. Un octet stochează două cifre hexazecimale. Aşadar, numărul zecimal 15, în hexazecimal F, se reprezintă ca 0x0F (1 octet, o pereche de cifre). Orice cod hexazecimal este reprezentat pe un şir de octeţi. Metodele de obţinere a codurilor hexazecimale implică două concepte de bază:

* codurile de caractere;
* ordinea octeţilor.

**Codurile de caractere** stau la baza codificării hexazecimale a şirurilor de caractere. Variantele cele mai întâlnite sunt:

* ASCII – setul de 256 caractere de bază, cu 1 octet pe caracter;
* Unicode – setul extins cu 1-4 octeţi pe un caracter, care are o serie de versiuni:
  + UTF-8, conceput pentru a fi compatibil cu ASCII: caracterele din setul de bază folosesc 1 octet şi codurile lor coincid cu cele ASCII, caracterele cu diacritice folosesc 2 octeţi, în plus mai există o serie de caractere cu utilizare rară ce ocupă 3 sau 4 octeţi;
  + UTF-16, incompatibil cu ASCII: caracterele occidentale uzuale, inclusiv cu diacritice, se stochează pe 2 octeţi, dar există şi coduri de 3 sau 4 octeţi.

Nu doar şirurile de caractere pot fi codificate hexazecimal. Un alt exemplu de cod hexazecimal sunt codurile de culoare. AutoIt lucrează cu codul RGB pe 3 octeţi, care rezervă câte un octet pentru intensitatea culorilor R (roşu), G (verde) şi B (albastru). Din cele 3 culori, fiecare cu intensităţi variabile de la 0 (00) la 255 (FF), se pot obţine toate celelalte culori. Spre exemplu:

* culoarea roşu e codificată 0xFF0000, deci R la intensitate maximă, G şi B la minim;
* culoarea albă e codificată 0xFFFFFF, cu toate culorile la intensitate maximă;
* culoarea neagră e codificată 0x000000, cu toate culorile la intensitate minimă;
* nuanţele de gri au coduri în care cele trei culori au intensitate egală: 0x111111 (mai aproape de negru), 0xA2A2A2 (mai aproape de alb).

Un alt exemplu de coduri hexazecimale sunt identificatorii ferestrelor şi obiectelor dintr-o interfaţă grafică (vezi capitolul dedicat programării GUI).

**Ordinea octeţilor** poate varia după două ordini consacrate de stocare a datelor complexe: **little-endian (LE)** şi **big-endian (BE)**.

* în sistemul de memorare little-endian (LE), gradul de semnificaţie a componentelor informaţiei creşte spre dreapta.
* în sistemul de memorare big-endian (BE), gradul de semnificaţie a componentelor informaţiei descreşte spre dreapta (se citeşte mai întâi cea mai semnificativă informaţie).

Exemple:

|  |  |  |
| --- | --- | --- |
| **Utilizări** | **Little - end** | **Big – end** |
| Date calendaristice: | 12-August-2007 | 2007-August-12 |
| Adrese poştale: | Nr.X, Strada X,  Orasul X, Judetul X | Judetul X, Orasul X,  Strada X, Nr.X |
| Nume persoane: | Prenume, Nume | Nume, Prenume |
| Valoarea 0x052346: | 0x462305 | 0x052346 |

Ordinile LE şi BE sunt folosite, după cum se vede din tabel, în numeroase contexte. Sistemul de numeraţie arab foloseşte o ordine BE: în numărul 653, 6 este cifra cea mai semnificativă (numărul de sute), iar gradul de semnificaţie descreşte spre dreapta. Sistemul de numeraţie roman este unul mixt: în numărul XV (15), ordinea este BE, în numărul IX (9), ordinea cifrelor este LE. După cum se vede în ultimul rând al tabelului, ordinea de reprezentare a octeţilor unei valori hexazecimale poate la rândul său să difere. De regulă diferenţa este dată de modelul de procesor (Intel şi AMD sunt procesoare implicit LE) dar metoda de codificare hexazecimală poate fi controlată şi de funcţii AutoIt specifice: există posibilitatea de a obţine coduri ASCII, UTF-8, UTF-16 LE, UTF-16 BE sau coduri LE obţinute din numere hexazecimale.

Pentru numerele hexazecimale se foloseşte ordinea intuitivă (pentru un cititor occidental, obişnuit să citească de la stânga la dreapta), deci BE. Aşadar, numărul 4096 se va scrie în hexazecimal 0x1000 (BE) şi nu 0x0010 (LE). În schimb, din numărul 4096 se poate obţine un cod hexazecimal prin diferite metode – ordonare LE, ordonare BE, tratarea cifrelor ca şi caractere ASCII sau Unicode, etc. Mai multe detalii se vor prezenta în subcapitolul rezervat funcţiilor de codificare hexazecimală.

Limita de 4 octeţi e impusă doar asupra numerelor hexazecimale şi la folosirea anumitor operaţii binare (ex: BitAND). Spre exemplu, o atribuire precum

$a=0x0123456789

...va provoca o eroare datorită depăşirii limitei (foloseşte 5 octeţi). În schimb, limita nu mai e valabilă în cazul codurilor hexazecimale obţinute printr-o conversie (funcţia Binary()):

$a=Binary("xyzxyzxyzxyz")

...valoarea variabilei va depăşi cu siguranţă cei 4 octeţi, fără să provoace o eroare (dar nu va putea fi folosită în operaţii binare limitate la 4 octeţi).

Concluzionăm cu **regulile de conversie implicită**:

* La string se poate converti orice alt tip
  + Prin adăugarea de ghilimele (numere zecimale, valori booleene, coduri hexazecimale)
  + Prin conversie la numeric zecimal urmată de adăugare de ghilimele (numere exponenţiale şi hexazecimale)
* La numeric:
  + Valorile care nu pot fi intrepretate numeric sunt convertite la 0. În această categorie intră *şirurile de caractere neconvertibile* (inclusiv "True") şi *codurile hexazecimale* (returnate de funcţii);
  + Se convertesc direct şiruri de caractere interpretabile numeric: în format zecimal, hexazecimal BE sau exponenţial (ex: "-12","0x0F", "1.0e+3", etc.)
    - numerele în format exponenţial se convertesc doar dacă exponentul are semn iar mantisa are măcar o zecimală – deci conversiile şirurilor "1e3" sau "1.e-3" vor eşua (rezultat 0), dar "1.0e-3" va avea succes (rezultat 0.001).
  + Valorile booleene se convertesc la 1 (True), respectiv 0 (False).
* La boolean se poate converti orice tip
  + "" şi 0 se convertesc la False
  + Orice alt număr sau şir de caractere se converteşte la True (inclusiv şirul "False" sau caracterul "0").
* La hexazecimal nu au loc conversii implicite – nu există nici un operator care să declanşeze conversii implicite de acest tip şi toate valorile literale hexazecimale sunt considerate numere. Codurile hexazecimale se obţin cu ajutorul funcţiilor specifice.

Operatorii din expresii vor stabili către care din primele trei tipuri se realizează conversia implicită. În acest sens, se va consulta tabelul descris la subcapitolul dedicat operatorilor.

Alături de conversia implicită provocată de operatori, există şi posibilitatea conversiei explicite, cu ajutorul funcţiilor de conversie, care vor fi discutate în subcapitolul dedicat funcţiilor AutoIt. Spre exemplu obţinerea de coduri hexazecimale e posibilă doar cu ajutorul acestor funcţii.

**X.3.2. Variabile şi constante**

Variabilele şi constantele sunt locaţii de memorie care pot stoca următoarele categorii de informaţii, în urma operaţiilor de **atribuire**:

* Valori elementare şi masive, literale sau rezultate ale unor expresii şi funcţii;
* Referinţe la obiecte COM sau proprietăţile acestora;
* Structuri de date C;
* Identificatori GUI.

Numele variabilelor şi constantelor este prefixat cu simbolul $. Declararea acestora se realizează cu instrucţiunile Dim, Local sau Global. Constantele sunt considerate variabile cu valoare fixată (vor suferi o singură atribuire). Constantele se declară adăugând cuvintele cheie Const sau Enum la oricare din cele 3 instrucţiuni de declarare. E posibil să se declare mai multe variabile cu aceeaşi instrucţiune şi e posibil să se iniţializeze oricare dintre acestea la declarare:

Dim $v1, $v2=3, $v3

Instrucţiunea Dim e implicită, de aceea poate să lipsească. Aceasta nu înseamnă că declararea e opţională, ci că declararea Dim se realizează implicit la iniţializarea unei variabile:

$v1=10

Presupunând că aceasta e prima atribuire a variabilei v1, aceasta este echivalentă cu o declarare precum:

Dim $v1=10

Datorită caracterului implicit, instrucţiunea Dim este în general evitată. Obligativitatea sa apare în cazurile:

* Obligativitatea instrucţiunilor de declarare (Dim, Local, Global) poate fi activată cu funcţia **Opt ("MustDeclareVars",1)** sau dezactivată prin apelul Opt ("MustDeclareVars",0);
* Instrucţiunile de declarare sunt obligatorii în cazul **variabilelor masive**;
* Instrucţiunile de declarare se folosesc atunci când se doreşte diferenţierea între tipurile de declaraţii (vezi diferenţele între Dim, Local şi Global).

Declararea de constante va conţine şi atribuirea valorii constante:

Dim Const $c=1

Dim Enum $c1=1, $c2, $c3

Declararea prin Enum are ca efect atribuirea de valori incrementate constantelor enumerate. În exemplul de mai sus, valorile primite de $c2 şi $c3 vor fi 2, respectiv 3. Opţional, instrucţiunea Enum permite precizarea pasului de incrementare[[17]](#footnote-17):

Enum Step 2 $c1=1, $c2, $c3

De data aceasta, valorile ultimelor două constante vor fi 3, respectiv 5. Pasul poate fi aditiv (implicit), dar şi multiplicativ:

Enum Step \*2 $c1=1, $c2, $c3

De data aceasta, valorile ultimelor două constante vor fi 2, respectiv 4. Dacă prima constantă nu e iniţializată, aceasta va primi valoarea implicită 0, folosită ca bază de incrementare. Incrementările vor fi evitate pentru acele constante cărora li se atribuie explicit valori:

Enum Step -2 $c1, $c2, $c3=10

Valorile celor trei constante vor fi, în ordine: 0 (implicită), -2 (decrementată cu pasul 2) şi 10 (explicită).

Diferenţele între Dim, Local şi Global sunt legate de domeniul de vizibilitate a variabilei atunci când programul conţine funcţii ale utilizatorului (vezi subcapitolul dedicat funcţiilor utilizatorului pentru exemplificări).

* O declaraţie Global face ca variabila să fie vizibilă global, în toate modulele (funcţiile) – creează riscul conflictelor de nume (variabile diferite denumite la fel în diferite module);
* Local declară variabile care sunt vizibile doar în modulul (funcţia) în care au fost declarate. Dacă alte funcţii (sau programul principal) au creat deja o variabilă globală cu acelaşi nume, va fi ignorată, deci se evită un conflict de nume;
* O declaraţie Dim face ca variabila să fie vizibilă local, doar dacă nu există deja una globală cu acelaşi nume. În cazul în care există, Dim NU creează o variabilă locală nouă (aşa cum face Local)!

Variabilele AutoIt pot fi elementare sau masive (arrays). **Masivele** pot avea până la 64 dimensiuni indexate. Masivele cu o dimensiune indexată sunt vectori, cele cu două dimensiuni indexate sunt matrici. Indexarea începe de la zero, deci ultimul element al unui vector declarat cu dimensiunea n va avea indicele n-1. Considerăm declaraţia:

Dim $A[2][3]=[["a","b","c"],[1,2,3]]

Ultimul element al acestei matrici, valoarea 3, va fi identificată prin $A[1][2]. Primul element al matricii, valoarea "a", va fi identificată prin $A[0][0].

Următorul exemplu declară şi iniţializează o serie de masive:

1. Dim $A[2][3]=[["a","b","c"],[1,2,3]]

2. $B=$A

3. $A="x"

4. Dim $C[2]=[1,$B]

5. Dim $v1=2

6. $v1=$C[1]

7. $v2=$v1[0][0]

Matricea $A are două dimensiuni, prima cu 2 elemente, a doua cu 3 elemente, deci 2 rânduri şi 3 coloane. Se observă din acest exemplu că masivele AutoIt sunt eterogene, exemplul conţine atât valori numerice cât şi şiruri de caractere. De fapt, matricea e omogenă, conţinând doar date de tipul variant!

Ultimele linii indică o serie de metode de atribuire speciale permise de AutoIt:

* a doua linie a exemplului copiază structura şi elementele masivului $A, în $B. Clauza Dim nu mai e necesară la declararea de masive iniţializate prin copiere!
* a treia linie atribuie o valoare elementară unei variabile masive. Efectul va fi eliberarea memoriei masivului şi convertirea sa la variabilă elementară (cu iniţializare)
* a patra linie sugerează că un element de masiv poate lua ca valoare un alt masiv. S-a definit vectorul $C, primul element fiind numărul 1, al doilea element fiind o copie a masivului $B
* a şasea linie extrage copia masivului $B din vectorul $C şi o atribuie unei variabile elementare (deci e posibilă şi conversia de la elementar la masiv, fapt confirmat de atribuirea de la linia 7 unde variabila cândva elementară $v1 e folosită ca masiv. Valoarea finală a lui $v2 va fi "a" (preluată de pe poziţia 0,0 – primul rând, prima coloană - din masivul $v1, care conţine al doilea element al masivului $C, care conţine masivul $B, care este o copie a matricii $A).

Copierea de masive (linia 2) şi conversia de la masiv la elementar (linia 3) sunt recomandate ca operaţii facile de manipulare a datelor din memoria internă. În schimb atribuirea de masive unei variabile elementare (linia 6) sau unui element de masiv (linia 5) pot afecta grav performanţa şi consumul de memorie internă.

Un masiv îşi poate modifica dimensiunile pe parcursul algoritmului. Aceasta necesită o redeclarare a sa cu instrucţiunea din exemplul:

ReDim $A[2][2]

Acest exemplu redeclară matricea $A, iniţializată în exemplul precedent. Redeclararea se face prin precizarea noilor limite ale dimensiunilor. Avantajul redeclarării este că ReDim păstrează valorile existente din matrice, vizibilitatea sa (Local, Global, Dim) şi îi modifică doar dimensiunile. E important de reţinut că ReDim nu va adăuga sau şterge dimensiuni, ci doar limita acestora. Astfel, următorul exemplu va defini un masiv nou, fără să păstreze nimic din matricea $A, deoarece s-a pierdut o dimensiune:

ReDim $A[2]

În mod similar cu majoritatea limbajelor pentru scripturi (JavaScript, VBScript), AutoIt are capabilităţi obiectuale limitate, fiind esenţială posibilitatea de a instanţia şi referi obiecte COM, colecţii de obiecte şi proprietăţile aferente (prin calificare). Astfel, o variabilă poate conţine o **referinţă** spre un obiect, proprietate sau colecţie de proprietăţi (care de regulă sunt la rândul lor obiecte). Acest lucru se realizează prin atribuire simplă. Membrul din dreapta atribuirii va conţine un obiect sau o funcţie care returnează un obiect, cum este cazul funcţiei de instanţiere ObjCreate:

$v=ObjCreate ("shell.application")

$ferestre=$v.windows

Prima variabilă va memora o referinţă spre o instanţă a clasei shell.application, care este de fapt aplicaţia shell folosită de sistemul de operare (Windows Explorer în mod implicit). A doua variabilă va memora o referinţă spre colecţia de proprietăţi windows, care reprezintă toate ferestrele shell deschise la momentul curent (ferestrele Windows Explorer). Deoarece nu toate calculatoarele conţin aceleaşi obiecte COM, se recomandă ca fiecare instanţiere să fie urmată de un test care verifică succesul acesteia (vezi funcţiile de testare a datelor). Mai multe detalii se vor prezenta în capitolul dedicat elementelor obiectuale din AutoIt.

O variabilă AutoIt mai poate stoca şi **structuri de date C**. Se ştie că C este un limbaj puternic tipizat, în care tipurile datelor sunt declarate explicit şi nu se modifică implicit. Aceasta are impact şi asupra variabilelor masive, care în C sunt considerate colecţii omogene de date, în timp ce în AutoIt, o matrice poate conţine date de orice tip. Pentru a crea o colecţie eterogenă de date, în C se poate defini o **structură de date,** printr-o construcţie precum:

struct {

int v1;

unsigned char v2;

unsigned int v3;

char v4[20];

}

AutoIt permite crearea structurilor de date C şi stocarea lor în variabile, cu ajutorul funcţiei DllStructCreate:

$v=DllStructCreate("int v1;ubyte v2;uint v3;char v4[20]")

Ulterior, componentelor structurii stocate în $v li se pot atribui valori cu ajutorul funcţiei DllStructSetData:

DllStructSetData($v,"v1",10)

...sau, componentele structurii pot fi atribuie unor variabile elementare, cu ajutorul funcţiei DllStructGetData:

$intreg=DllStructGetData($v,1)

O structură de date C e distrusă dacă variabilei i se atribuie valoarea 0:

$v=0

Mai multe detalii vor fi prezentate în cadrul capitolului privind structurile de date C.

În sfârşit, o variabilă mai poate conţine **identificatori GUI[[18]](#footnote-18)**, care se folosesc în manipularea interfeţelor grafice create de AutoIt sau de alte aplicaţii. Atunci când un macro AutoIt lucrează cu mai multe ferestre simultan, e important ca acestea să poată fi identificate şi referite fără ambiguitate. În acest scop, Windows alocă identificatori GUI unici la crearea unei ferestre sau a unui obiect GUI, sub forma unui cod hexazecimal:

$IDfereastra=GUICreate("Fereastra Mea",200,200)

Acest exemplu creează o fereastră la dimensiunile 200x200, cu titlul indicat de primul argument. Variabila $Idfereastra va stoca identificatorul GUI al ferestrei create. Atunci când fereastra sau obiectul sunt create de altă aplicaţie şi trebuie accesată de AutoIt, limbajul foloseşte o serie de funcţii pentru citirea identificatorului GUI, ca în exemplul:

$IDfereastra=WinGetHandle ("Untitled – Notepad")

Acest exemplu stochează în variabilă identificatorul ferestrei Notepad. Nu doar ferestrele au identificatori GUI, ci şi obiectele GUI care apar în cadrul lor (butoanele, casetele de editare, butoanele radio etc.). Identificatorii GUI sunt mecanismul prin care sistemul de operare identifică unic elementele interfeţei grafice Windows şi AutoIt are acces la acest mecanism. Mai multe detalii se vor discuta în cadrul capitolelor dedicate programării GUI, respectiv manipulării ferestrelor Windows prin macrouri.

**X.3.3. Macrovariabile**

**Macrovariabilele**, indicate prin nume prefixate cu @, pot fi considerate variabile read-only (nu li se pot atribui valori, nu se declară, nu se iniţializează). Acestea au valori preluate de la mediul de execuţie. În realitate, acestea sunt fie variabile de mediu, fie macrouri predefinite şi reutilizabile care returnează o valoare. Spre exemplu macrovariabila @ScriptName indică numele fişierului în care e stocat macroul curent. Lista completă a macrovariabilelor AutoIt este indicată în tabelul de mai jos. De cele mai multe ori e vorba de informaţii care descriu mediul de lucru – căi de directoare, informaţii privind sistemul de operare, informaţii privind data şi ora curentă, macrouri de manipulare a ferestrelor, informaţii privind obiectul GUI curent, dar şi caractere speciale precum Tab, CR, LF. Tabelul prezentat aici conţine toate macrovariabilele – semnificaţia lor este detaliată pe parcursul materialului, în cazurile concrete în care sunt folosite.

|  |  |
| --- | --- |
| **MACROVARIABILA** | **DESCRIERE** |
| **I. INFORMAŢII PRIVIND MACROUL AUTOIT CURENT** | |
| **@compiled** | Returnează 1 dacă macroul e compilat (executabil), altfel 0. |
| **@unicode** | Returnează 1 dacă executarea macroului e realizată de interpretorul AutoIt WinXP (Unicode), 0 dacă e realizată de interpretorul AutoIt Win98. Această verificare e necesară la generarea casetelor de dialog (casetele Win98 nu pot conţine butoanele Try again, Continue) |
| **@error** | Semnalează succesul sau eşecul celei mai recente funcţii apelate. |
| **@exitCode** | Oferă codul de terminare corespunzător celei mai recente instrucţiuni Exit şi poate fi testat de funcţia OnAutoItExit. |
| **@exitMethod** | Oferă motivul terminării macroului (terminare normală, Exit, Logoff, Shutdown) şi poate fi testat de funcţia OnAutoItExit. |
| **@extended** | Returnează valori suplimentare pentru anumite funcţii (StringReplace) |
| **@NumParams** | Folosită într-o funcţie a utilizatorului, returnează numărul de argumente din apelul curent al funcţiei. |
| **@ScriptName** | Returnează numele fişierului care conţine macroul curent. |
| **@ScriptDir** | Returnează numele directorului care conţine macroul curent. |
| **@ScriptFullPath** | Returnează calea completă a macroului curent - echivalent cu rezultatul concatenării *@ScriptDir & "\" & @ScriptName* |
| **@ScriptLineNumber** | Returnează numărul liniei de cod aflate în execuţie, utilă în depanare. |
| **@WorkingDir** | Returnează directorul de lucru curent |
| **@AutoItExe** | Returnează calea completă a macroului executabil curent |
| **@AutoItPID** | Returnează identificatorul procesului în care se execută macroul curent (PID). |
| **@AutoItVersion** | Returnează versiunea AutoIt |
| **@InetGetActive** | Returnează 1 dacă un download InetGet este activ, altfel 0. |
| **@InetGetBytesRead** | Returnează numărul de octeţi citiţi în timpul unui download InetGet, sau -1 la eroare de download. |
| **II. INFORMAŢII PRIVIND OBIECTUL GUI CURENT** | |
| **@COM\_EventObj** | Returnează evenimentul COM activat, poate fi folosit doar în funcţii privind evenimente COM |
| **@GUI\_CtrlId** | Folosit într-o funcţie handler, returnează identificatorul obiectului GUI sau evenimentului GUI care a apelat funcţia |
| **@GUI\_CtrlHandle** | Folosit într-o funcţie handler, returnează handlerul obiectului GUI care a apelat funcţia!!! |
| **@GUI\_DragID** | Returnează identificatorul obiectului GUI care a suferit o operaţie Drag. |
| **@GUI\_DragFile** | Returnează numele fişierului care a suferit o operaţie Drag and Drop. |
| **@GUI\_DropID** | Returnează identificatorul obiectului GUI care a suferit o operaţie Drop. |
| **@GUI\_WinHandle** | Folosit într-o funcţie handler, returnează identificatorul ferestrei care a apelat funcţia. |
| **@HotKeyPressed** | Returnează ultima tastă caldă apăsată, conform funcţiei [HotkeySet](mk:@MSITStore:d:\autoIT3\AutoIt3.chm::/html/functions/HotKeySet.htm) |
|  | |
| **III. INFORMAŢII DE CONFIGURARE A FERESTRELOR, utilizate împreună cu funcţiile** [**WinSetState**](mk:@MSITStore:d:\autoIT3\AutoIt3.chm::/html/functions/WinSetState.htm)**,** [**Run**](mk:@MSITStore:d:\autoIT3\AutoIt3.chm::/html/functions/Run.htm)**,** [**RunWait**](mk:@MSITStore:d:\autoIT3\AutoIt3.chm::/html/functions/RunWait.htm)**,** [**FileCreateShortcut**](mk:@MSITStore:d:\autoIT3\AutoIt3.chm::/html/functions/FileCreateShortcut.htm)**,** [**FileGetShortcut**](mk:@MSITStore:d:\autoIT3\AutoIt3.chm::/html/functions/FileGetShortcut.htm) | |
| **@SW\_DISABLE** | Fereastra devine inaccesibilă |
| **@SW\_ENABLE** | Fereastra devine accesibilă |
| **@SW\_HIDE** | Ascunde fereastra şi activează altă fereastră |
| **@SW\_LOCK** | Blochează poziţia ferestrei |
| **@SW\_MAXIMIZE** | Maximizează fereastra |
| **@SW\_MINIMIZE** | Minimizează fereastra şi activează următoarea fereastră conform ordinii de creare a lor. |
| **@SW\_RESTORE** | Activează şi afişează fereastra la dimensiunile şi poziţia sa originale (dacă a fost minimizată sau maximizată anterior) |
| **@SW\_SHOW** | Activează şi afişează fereastra la dimensiunea şi poziţia sa curentă |
| **@SW\_SHOWDEFAULT** | Afişează fereastra în modul implicit în care a fost definită de programul care a creat-o. |
| **@SW\_SHOWMAXIMIZED** | Activează fereastra şi o afişează maximizat |
| **@SW\_SHOWMINIMIZED** | Activează fereastra şi o afişează minimizat |
| **@SW\_SHOWMINNOACTIVE** | Afişează minimizat fereastra fără să o activeze |
| **@SW\_SHOWNA** | Afişează fereastra la dimensiunea şi poziţia curentă fără să o activeze. |
| **@SW\_SHOWNOACTIVATE** | Afişează fereastra la dimensiunea şi poziţia anterioară, fără să o activeze. |
| **@SW\_SHOWNORMAL** | Activează şi afişează fereastra la dimensiunea şi poziţia sa anterioară |
| **@SW\_UNLOCK** | Deblochează poziţia ferestrei |
| **IV. INFORMAŢII LEGATE DE BARA TRAY**  **(zona din dreapta a barei de sarcini – zona cu afişajul orei)** | |
| **@TRAY\_ID** | Identificatorul ultimei pictograme Tray pe care s-a dat clic. |
| **@TrayIconFlashing** | Returnează 1 dacă pictograma tray clipeşte, altfel 0. |
| **@TrayIconVisible** | Returnează 1 dacă pictograma tray este vizibilă, altfel 0. |
| **V. CARACTERE SPECIALE** | |
| **@CR** | Caracterul Carriage return cu codul ASCII 13 (revenire la început de rând). |
| **@LF** | Caracterul Line feed, cod ASCII 10 (salt la rând nou) |
| **@CRLF** | Echivalent cu concatenarea @CR & @LF (salt la rând nou cu revenire la început de rând) |
| **@TAB** | Caracterul Tab, cu codul ASCII 9 |
| **VI. CĂI ALE DIRECTOARELOR DE BAZĂ WINDOWS**  preluate din registrul *HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion* | |
| **@AppDataCommonDir** | Calea Application Data |
| **@DesktopCommonDir** | Calea Desktop |
| **@DocumentsCommonDir** | Calea Documents |
| **@FavoritesCommonDir** | Calea Favorites |
| **@ProgramsCommonDir** | Calea Programs (director din Start Menu) |
| **@StartMenuCommonDir** | Calea Start Menu |
| **@StartupCommonDir** | Calea Startup |
| **@AppDataDir** | Calea Application Data pentru utilizatorul curent |
| **@DesktopDir** | Calea  Desktop pentru utilizatorul curent |
| **@MyDocumentsDir** | Calea My Documents pentru utilizatorul curent |
| **@FavoritesDir** | Calea Favorites pentru utilizatorul curent |
| **@ProgramsDir** | Calea Programs (director din Start Menu) pentru utilizatorul curent |
| **@StartMenuDir** | Calea Start Menu pentru utilizatorul curent |
| **@StartupDir** | Calea Startup pentru utilizatorul curent |
| **@UserProfileDir** | Calea Profile pentru utilizatorul curent |
| **@HomeDrive** | Litera discului care conţine directorul de lucru al utilizatorului curent |
| **@HomePath** | Calea directorului de lucru al utilizatorului curent |
| **@HomeShare** | Serverul şi numele de share care conţine directorul de lucru al utilizatorului curent |
| **@LogonDNSDomain** | Domeniul DNS la care s-a conectat utilizatorul curent |
| **@LogonDomain** | Domeniul de login |
| **@LogonServer** | Serverul la care s-a făcut login |
| **@ProgramFilesDir** | Calea Program Files |
| **@CommonFilesDir** | Calea Common Files |
| **@WindowsDir** | Calea Windows |
| **@SystemDir** | Calea directorului sistem (System sau System32) |
| **@TempDir** | Calea directorului cu fişiere temporare |
| **@ComSpec** | Interpretorul de comenzi secundar (utilizat de funcţia Run pentru a executa comenzi MS-DOS) |
| **VII. INFORMAŢII PRIVIND SISTEMUL DE OPERARE** | |
| **@KBLayout** | Codul tastaturii instalate (identificator de localizare geografică) – 0409 pentru Engleza SUA, 0418 pentru Română[[19]](#footnote-19). |
| **@OSLang** | Codul de localizare a sistemului de operare. |
| **@OSType** | "WIN32\_NT" pentru NT/2000/XP/2003/Vista  "WIN32\_WINDOWS" pentru 95/98/Me |
| **@OSVersion** | "WIN\_VISTA", "WIN\_2003", "WIN\_XP", "WIN\_2000", "WIN\_NT4", "WIN\_ME", "WIN\_98", "WIN\_95" |
| **@OSBuild** | Numărul de build al sisemului de operare (ex:Windows 2003 Server are codul 3790) |
| **@OSServicePack** | Numărul de service pack al sistemului de operare |
| **@ProcessorArch** | Tipul de procesor: "X86", "IA64", "X64" |
| **@ComputerName** | Numele calculatorului în reţea |
| **@UserName** | Identificatorul utilizatorului |
| **@IPAddress1** | Adresa IP a primei plăci de reţea (127.0.0.1 dacă s-a definit adresa localhost) |
| **@IPAddress2** | Adresa IP a celei de-a doua plăci de reţea (alta decât 127.0.0.1 sau 0.0.0.0 dacă nu există). |
| **@IPAddress3** | Idem |
| **@IPAddress4** | Idem |
| **VIII.CONFIGURAREA VIDEO A SISTEMULUI** | |
| **@DesktopHeight** | Rezoluţia verticală |
| **@DesktopWidth** | Rezoluţia orizontală |
| **@DesktopDepth** | Adâncimea culorilor (biţi per pixel) |
| **@DesktopRefresh** | Rata de împrospătare în herzi |
| **IX. INFORMAŢII TEMPORALE CURENTE** | |
| **@SEC** | Secunde |
| **@MIN** | Minute |
| **@HOUR** | Ora |
| **@MDAY** | Ziua din lună |
| **@MON** | Luna |
| **@YEAR** | Anul în 4 cifre |
| **@WDAY** | Ziua din săptămână (numeric, cu 1 corespunzând lui Duminică) |
| **@YDAY** | Ziua din an (1-366) |

**X.3.4. Operatori**

|  |  |  |
| --- | --- | --- |
| **OPERATOR** | **DESCRIERE** | **Conversie implicită[[20]](#footnote-20)** |
| AND | Conjuncţie logică | La boolean |
| OR | Disjuncţie logică | La boolean |
| NOT | Negare logică | La boolean |
| += | Atribuire cu adunare | La numeric |
| -= | Atribuire cu scădere | La numeric |
| \*= | Atribuire cu înmulţire | La numeric |
| /= | Atribuire cu împărţire | La numeric |
| + | Adunare | La numeric |
| - | Scădere | La numeric |
| \* | Înmulţire | La numeric |
| / | Împărţire | La numeric |
| ^ | Ridicare la putere | La numeric |
| &= | Atribuire cu concatenare | La string |
| & | Concatenare | La string |
| = | Atribuire | Converteşte variabila din stânga la tipul expresiei din dreapta |
| = | Egalitate (test logic) | * La boolean, dacă măcar unul din operanzi e boolean * La numeric în comparaţiile numeric=string * Nu au loc conversii când ambii membri au acelaşi tip * Comparaţiile string=string nu sunt case-sensitive |
| == | Egalitate case-sensitive | * La string (deci nu sunt echivalente True şi 1, False şi 0 etc.) * Comparaţiile sunt case-sensitive |
| <> | Inegalitate | * La boolean, dacă măcar unul din operanzi e boolean * La numeric în comparaţiile numeric<>string * Nu au loc conversii când ambii membri au acelaşi tip * Comparaţiile string<>string nu sunt case-sensitive |
| > | Mai mare | La numeric, cu excepţia cazului în care ambii termeni sunt string (se aplică ordinea ASCII) |
| >= | Mai mare sau egal | La numeric, cu excepţia cazului în care ambii termeni sunt string (se aplică ordinea ASCII) |
| < | Mai mic | La numeric, cu excepţia cazului în care ambii termeni sunt string (se aplică ordinea ASCII) |
| <= | Mai mic sau egal | La numeric, cu excepţia cazului în care ambii termeni sunt string (se aplică ordinea ASCII) |

Operaţiile la nivel de bit (conjuncţie pe biţi, disjuncţie pe biţi, sau exclusiv etc.) nu se realizează prin operatori ci prin funcţii predefinite în acest scop.

E important de reţinut că operatorii de comparaţie *oferă rezultate logice în formate diferite*:

* >,<,>=,<= oferă rezultat numeric (0 sau 1)
* =, ==, <> oferă rezultat boolean (True sau False)

Comparaţiile se realizează:

* Conform ordinii numerice, între valori numerice, indiferent de format
* Conform ordinii ASCII sau Unicode, între valori string (sau coduri hexazecimale), cu excepţia faptului că majusculele sunt considerate egale cu minusculele[[21]](#footnote-21)
* Conform regulii True>False, între valori booleene

Exemple:

1. $a=("B">"A")
2. $b=("a">2)
3. $c=("2"=2)
4. $d=("a" == "A")
5. $e=(True>"0")
6. $f=("a" = 0)
7. $g=("a">"A") OR ("A">"a")
8. $h=("a">="A") AND ("a"<="A")
9. $i=("True"=0)
10. $j=(True="0")
11. $k=("False"=False)
12. $l=("False"==False)
13. $m=((3>2)==(True="2"))
14. $n=(0x0F=="15")
15. $o=("0x0F"==15)

Valorile stocate în variabilele din stânga acestor atribuiri vor fi, în ordine:

1. Valoarea numerică 1 corespunzătoare valorii logice "adevărat". Comparaţia între caracterele "B" şi "A" se realizează în baza codurilor ASCII (care respectă ordinea alfabetică);
2. Valoarea numerică 0, corespunzătoare valorii logice "fals". Comparaţia necesită o conversie la numeric. Conversia literei "a" la numeric euşează deci are ca rezultat cifra 0. În final, comparaţia 0>2 are rezultat fals;
3. Valoarea booleană True, având în vedere succesul conversiei de la "2" la numărul 2, şi faptul că egalitatea rezultată este adevărată;
4. Valoarea booleană False, având în vedere că egalitatea case-sensitive între "a" şi "A" este falsă;
5. Valoarea numerică 1, comparaţia realizată în urma conversiei la numeric fiind 1>0;
6. Valoarea booleană True, are loc conversia la numeric, care eşuează în cazul lui "a", deci practic se compară 0=0;
7. Valoarea booleană False, având în vedere că ambii membri ai disjuncţiei sunt afirmaţii False (nu are loc o comparare între codurile ASCII – "a" şi "A" sunt considerate egale);
8. Din acelaşi motiv, variabila primeşte valoarea True;
9. Egalitatea provoacă conversie implicită a şirului la numeric (dacă măcar un membru e numeric), deci se compară 0=0. Variabila primeşte valoarea True;
10. Fiind o compara'ie boolean=string, se face conversie la boolean, deci se compară True=True, cu rezultatul True (vezi regulile de conversie implicită la boolean din subcapitolul privind tipizarea datelor);
11. Fiind o comparaţie boolean=string, se face conversie implcită la boolean, deci se compară True=False (vezi regulile de conversie implicită la boolean din subcapitolul privind tipizarea datelor). Aşadar, rezultatul este False;
12. Operatorul == converteşte implicit la string, deci se compară "False"=="False", afirmaţie cu valoarea True;
13. Comparaţia care are loc este "1"=="True" (operatorul == converteşte la string). Variabila primeşte valoarea False;
14. În urma conversiei la string, se compară "15"=="15", cu valoarea True;
15. În urma conversiei la string, se compară "0x0F"=="15", cu valoarea False

Precedenţa operatorilor, aplicată pentru a stabili ordinea de calcul în expresiile fără paranteze (operatorii cu aceeaşi precedenţă sunt evaluaţi în ordinea în care apar în expresie, de la stânga la dreapta).

1. **NOT**
2. **^**
3. **\* /**
4. **+ -**
5. **&**
6. **< > <= >= = <> ==**
7. **AND OR**

Optimizarea expresiilor poate avea loc atunci când rezultatul unei expresii se cunoaşte înainte de evaluarea sa completă, mai precis în cazul disjuncţiei şi conjuncţiei:

$v= False AND (......)

Al doilea membru al conjuncţiei nu va mai fi procesat deoarece indiferent care ar fi acesta, valoarea atribuită va fi False. Acest comportament are impact atunci când membrii conjuncţiilor şi disjuncţiilor sunt care returnează valori booleene:

If F1() OR F2() Then ....

Dacă F1 returnează True, F2 nu va mai fi evaluată (deci conţinutul său nu va fi executat).

**X.3.5. Structuri condiţionale**

Structurile condiţionale sunt structuri clasice de ramificare a algoritmului pe baza unor situaţii decizionale. Condiţia care controlează ramificarea este evaluată ca o expresie cu rezultat boolean, True sau False, sau numerică, 1 sau 0 (vezi subcapitolul dedicat operatorilor) şi poate conţine operatori logici. AutoIt oferă trei structuri condiţionale: o structură IF şi două tipuri de structuri CASE: Select şi Switch.

$v= ("a" = "A")

If $v Then

$v&=2

Else

$v+=2

Endif

Prima linie a acestui exemplu atribuie variabilei $v, rezultatul egalităţii "a"="A". Rezultatul egalităţilor este o valoare booleană, în acest caz True, datorită faptului că operatorul = nu realizează comparaţie case-sensitive şi oferă rezultat în format boolean. Această valoare serveşte drept condiţie de ramificare a structurii IF. Cum valoarea este True, se execută doar ramura Then, care va modifica valoarea variabilei prin concatenarea lui 2 la valoarea True. Fiind vorba de concatenare, avem o conversie implicită la string, deci în final variabila $v va stoca valoarea "True2".

Ramura Else este opţională. Atunci când lipseşte, iar ramura Then e formată dintr-o singură instrucţiune, poate lipsi şi Endif:

If .... Then .....

Structurile cu ramificaţie multiplă (mai mult de 2 ramuri) se pot obţine pe căile:

* Folosirea ramurei ElseIf;
* Folosrea structurilor IF imbricate;
* Folosirea structurilor CASE, optimizate pentru ramificare multiplă.

Exemplu cu ramura ElseIf:

If .... Then

.........

ElseIf ..... Then

........

Else

.......

EndIf

Această construcţie va fi echivalentă cu două structuri IF imbricate:

If .... Then

.........

Else

If ..... Then

........

Else

.......

EndIf

EndIf

Ramificarea multiplă poate fi asigurată mai facil prin cele două variante ale structurii CASE. Diferenţa între Select Case şi Switch Case este:

* Select ramifică algoritmul în funcţie de condiţii logice (similare cu cele de la IF)
* Switch ramifică algoritmul în funcţie de valorile unei variabile de control.

$v1="2"

$v2=$v1+True

Select

Case $v1>2

$a=1

Case $v2=3

$a=2

Case "a"<True

$a=3

Case Else

$a=4

EndSelect

În urma primelor două atribuiri, $v1 are valoarea "2", v2 are valoarea 3 (prin conversia implicită la numeric a valorilor $v1 şi True). Valorile de adevăr ale condiţiilor Case vor fi, în ordine:

1. Se compară un string cu un număr, se aplică o conversie la numeric, se compară 2>2, fals
2. Se compară un număr cu un număr, adică 3=3, adevărat
3. Se compară un string cu o valoare booleană, ambele se convertesc la numeric, se compară deci 0 (conversie eşuată) < 1 (True), adevărat
4. Ramura Case Else e implicită şi se consideră întotdeauna validă de executat.

Aşadar, în structura Select, e posibil ca mai multe ramuri să fie valide pentru execuţie, deoarece fiecare ramură e controlată de propria condiţie, independentă de celelalte condiţii. În acest exemplu, ultimele trei ramuri sunt valide. Dintre acestea se execută doar prima, cele care urmează fiind ignorate (deci ramura Case Else se execută doar dacă nici una din ramurile anterioare nu e validă). În concluzie, valoarea finală a variabilei $a va fi 2 (prima dintre ramurile valide).

Switch @HOUR

Case 6 to 11

$a="dimineata"

Case 12 to 15

$a="amiaza"

Case 15 to 18

$a="dupaamiaza"

Case 19 to 22

$a="seara"

Case Else

$a="noapte"

EndSwitch

Structura consultă valoarea macrovariabilei ce indică ora curentă şi o compară cu intervalele indicate. Dacă nici unul din cele patru intervale nu încadrează valoarea, se execută ramura Case Else, deci atribuirea $a="noapte".

Chiar dacă aceeaşi variabilă de control (în acest exemplu macrovariabila) este verificată în toate ramurile, e posibil şi la structura Switch ca mai multe ramuri să fie simultan valide, dacă nu s-a asigurat corect disjuncţia între valorile variabilei de control. Spre exemplu, dacă structura conţine ramurile:

Case 6 to 12

$a="dimineata"

Case 11 to 15

$a="amiaza"

...cele două ramuri au intervalele de valori ale variabilei de control nondisjuncte (suprapuse pe intervalul 11-12). Ca şi în cazul Select Case, se va executa doar prima ramură validă. În plus faţă de cazul Select Case, la Switch există riscul de a produce erori prin neglijarea unei părţi din valorile posibile ale variabilei de control.

Nu e obligatoriu ca o ramură să fie definită de un interval de valori ale variabilei de control. O ramură Case poate fi definită şi în formele:

Case 6

..........

Case 7 to 10, 12 to 17

..........

În acest exemplu, prima ramură se execută dacă variabila de control ia exact valoarea indicată. A doua ramură se execută dacă variabila de control aparţine oricăruia dintre cele două intervale.

O structură CASE trebuie să conţină măcar o ramură Case. Celelalte, inclusiv ramura Case Else, sunt opţionale. Structurile CASE pot fi la rândul lor imbricate.

Structura CASE poate fi forţată pentru a executa mai multe ramuri CASE succesiv şi nu doar una dintre ele. Aceasta se realizează prin comanda ContinueCase, care întrerupe executarea ramurii curent şi forţează executarea ramurii următoare (chiar dacă aceasta e invalidă în raport cu condiţia CASE). Exemplu:

$v1="2"

$v2=$v1+True

Select

Case $v1>2

$a=1

Case $v2=3

$a=2

ContinueCase

Case 0=1

$a=3

Case Else

$a=4

EndSelect

S-a arătat deja că în lipsa instrucţiunii ContinueCase, variabila $a va avea în final valoarea 2 (de pe prima ramură validă). În prezenţa instrucţiunii ContinueCase, se continuă prin executarea ramurii 3 (deşi aceasta este invalidă conform condiţiei 0=1) deci în final variabila va avea valoarea 3.

**X.3.6. Structuri repetitive**

Structurile repetitive în AutoIt sunt în număr de cinci, împărţite în categoriile:

* Structuri cu număr cunoscut de iteraţii (repetiţii): FOR Next;
* Structuri cu iteraţii controlate de o condiţie: While Wend, Do Until
* Structuri de parcurgere a colecţiilor de proprietăţi obiectuale şi a masivelor: FOR In şi With Endwith.

Toate structurile repetitive pot fi imbricate, cu excepţia structurii With.

Exemplu de structură FOR Next:

$a=""

For $i=2 to 6 Step 2

$a&=$i

Next

Acest exemplu parcurge din 2 în 2 (clauza opţională step 2) valorile dintre 2 şi 6, cu ajutorul contorului de iteraţii $i. Contorul de iteraţie nu trebuie să fie o variabilă anterior declarată (nici măcar dacă s-a activat obligativitatea declarării – vezi subcapitolul *Variabile şi constante*), ci va fi creată ca variabilă cu vizibilitate locală. În acest exemplu, variabila $a va acumula, prin concatenare, toate valorile contorului de iteraţii, deci va avea în final valoarea "246".

Pasul indicat prin clauza Step este opţional şi atunci când lipseşte are implicit valoarea 1. Pasul poate avea orice valoare, de orice tip, aceasta va fi convertită implicit la numeric datorită operaţiei de adunare implicită aplicată între contorul de iteraţii şi pas. Exemplu:

$a=""

For $i=2 to 3 Step (2=2)\*"0.5"

$a&=$i

Next

În acest exemplu valoarea pasului, numerică, va fi (în urma conversiilor implicite) True\*"0.5"=1\*0.5=0.5. Valorile contorului de iteraţii vor fi, aşadar, 2, 2.5 şi 3. Prin concatenări succesive, variabila $a va acumula valoarea "22.53". Conversia la numeric se aplică şi valorilor iniţială şi finală a contorului:

$a=""

For $i=2 to (3=2) Step -1

$a&=$i

Next

În acest exemplu, valoarea finală este 0 (rezultatul conversiei din False), iar pasul este negativ, deci contorul descreşte. Valoarea acumulată în $a va fi "210".

Ciclul FOR se opreşte atunci când valoarea contorului o depăşeşte pe cea a valorii finale (nu trebuie să coincidă cu aceasta). Pentru a evita ciclurile infinite, structura FOR va fi ignorată de interpretor în situaţiile:

* Valoarea iniţială este mai mare decât cea finală dar *pasul e pozitiv sau zero;*
* Valoarea iniţială este mai mică decât cea finală dar *pasul e negativ;*

E posibil ca valoarea iniţială, finală şi pasul să fie variabile care îşi modifică valoarea în interiorul ciclului FOR. Astfel de modificări NU afectează numărul de iteraţii – *cele trei valori sunt fixate la prima iteraţie*. Exemplu:

$a=""

$vi=0

$vf=4

$pas=2

For $i=$vi to $vf Step $pas

$a&=$i

$pas+=1

$vf+=1

Next

În final, valorile variabilelor vor fi:

* $vi=0
* $vf=7
* $pas=5
* $a="024" (neafectată de faptul că pasul şi valoarea finală şi-au schimbat valorile în timpul iteraţiilor).

Atunci când are loc parcurgerea de masive, devine utilă funcţia care returnează limitele dimensiunilor masivului. În plus, trebuie să se ţină cont de indexarea pornind de la zero (o dimensiune cu n elemente va fi indexată de la 0 la n-1). Considerăm exemplul:

$a=""

Dim $V**[2][3]**=[["a","b","c"],[1,2,3]]

For **$i=0 to 1**

For **$j=0 to 2**

$a&=$V[$i][$j]

Next

Next

Se observă două cicluri FOR imbricate care parcurg toate elementele matricii şi le concatenează. În final, variabila $a va avea valoarea "abc123". Se observă intervalul de valori ale contoarelor-indici (0-1 pentru dimensiunea declarată cu limita 2, 0-2 pentru dimensiunea declarată cu limita 3). Atragem atenţia asupra frecvenţei mari a erorilor de indexare a masivelor, datorită faptului că există numeroase limbaje cu indexare pornind de la 1 şi pot apare confuzii la programatorii care nu cunosc modul de indexare folosit de fiecare limbaj în parte. În AutoIt, următoarea parcurgere a matricii va fi eronată, provocând o depăşire a limitelor dimensiunilor matricii:

$a=""

Dim $V[2][3]=[["a","b","c"],[1,2,3]]

For **$i=1 to 2**

For **$j=1 to 3**

$a&=$V[$i][$j]

Next

Next

Ciclurile FOR pot folosi funcţia Ubound, care returnează limita unei dimensiuni a masivului. Parcurgerea matricii anterioare se poate realiza cu structura imbricată:

$a=""

Dim $V**[2][3]**=[["a","b","c"],[1,2,3]]

For **$i=0 to Ubound($A) -1**

For **$j=0 to Ubound($A,2)-1**

$a&=$V[$i][$j]

Next

Next

Argumentul obligatoriu al funcţiei indică masivul vizat. Al doilea argument, numeric şi opţional (apare în ciclul $j), poate lua valorile:

* 1,2,3... indică numărul dimensiunii pentru care se va măsura limita (în ciclul $j e vorba de a doua dimensiune, deci se va returna numărul coloane a matricii)
* 0, caz în care funcţia va returna *numărul de dimensiuni* şi nu numărul de elemente dintr-o dimensiune.

Se observă că datorită indexării pornind de la zero, se scade 1 din valoarea funcţiei pentru a obţine indicele ultimului elemente de pe dimensiunea respectivă.

În cazul de eşec, funcţia Ubound returnează 0 şi comută macrovariabila @error în funcţie de situaţie:

* La 1 dacă argumentul obligatoriu nu e un masiv;
* La 2 dacă argumentul opţional nu e valid.

**Structura repetitivă condiţională** e prezentă în formele:

* While, cu pre condiţie de continuare
* Do Until, cu post condiţie de oprire

Structura While execută blocul în mod repetitiv atâta timp cât o condiţie logică este îndeplinită (expresia logică ia valoarea 1 sau True). While plasează condiţia la începutul blocului iterativ, aşadar e posibil ca blocul să nu se execute niciodată dacă precondiţia nu e îndeplinită încă de la prima tentativă de iteraţie. Următorul exemplu e echivalent cu prima structură FOR Next din acest subcapitol:

$i="2"

$a=$i

While **$i<6**

$i+=2

$a&=$i

WEnd

În final, $a="246". Se observă valoarea iniţială şi pasul sunt indicate prin atribuiri explicite, iar valoarea finală printr-o condiţie logică. Orice structură FOR Next poate fi uşor transpusă într-o structură While dar invers este mult mai dificil (necesită de obicei o combinaţie de structuri FOR şi IF imbricate). Condiţia While poate fi orice expresie logică, pusă asupra oricărei şi oricâtor variabile, în timp ce condiţia de oprire FOR este limitată la compararea contorului cu valoarea finală.

Structura Do Until execută blocul în mod repetitiv până când o condiţie logică devine adevărată (1 sau True). Aşadar, la While avem o *condiţie de continuare*, iar la Do avem o *condiţie de oprire*. Deoarece Do plasează condiţia la finalul blocului iterativ, acesta va fi executat cel puţin o dată.

$i="0"

$a=""

Do

$i+=2

$a&=$i

Until **$i=6**

Acest exemplu este echivalent ca rezultat ($a="246") cu exemplul anterior. Se poate observa condiţia de oprire Until ("execută până când i devine 6"), comparativ cu condiţia de continuare While ("execută atâta timp cât i nu a devenit încă 6"). La finalul ambelor iteraţii contorul $i are valoarea 6 (deci neagă condiţia de continuare While şi îndeplineşte condiţia de oprire Until). Se pot remarca şi diferenţele de iniţializare între cele două situaţii, date de faptul că structura Do se execută măcar o dată înainte de verificarea condiţiei de oprire.

**Instrucţiuni de forţare a structurilor de repetitive**

Structurile FOR Next, While şi Do Until pot conţine instrucţiunea ContinueLoop pentru a forţa saltul la iteraţia următoare, evitând executarea unei porţiuni din blocul iterativ. Indicăm doar un exemplu FOR:

$a=""

For $i=2 to 6 Step 2

If $i=4 Then ContinueLoop

$a&=$i

Next

Efectul acestui exemplu va fi că pentru valoarea 4 a contorului de iteraţii, va avea loc o iteraţie de excepţie: concatenarea la $a va fi ignorată, făcându-se un salt direct la iteraţia următoare. Astfel, valoarea acumulată în final a lui $a va fi "26" în loc de "246", cum era cazul exemplului iniţial. Se observă că în mod normal ContinueLoop apare în cadrul unui test (IF sau CASE) care identifică iteraţiile ce vor fi tratate prin excepţie (abateri de la caracterul repetitiv al structurii).

ContinueLoop poate primi opţional o valoare numerică, atunci când apare în cadrul unor structuri FOR Next, While sau Do Until imbricate. Numărul respectiv identifică structura repetitivă la care se referă ContinueLoop, prin nivelul de imbricare.

Dim $V[3][3]=[[1,2,0],[1,0,2],[0,1,2]]

$a=""

For $i=0 to 2

For $j=0 to 2

$a&=$V[$i][$j]

Next

Next

Acest exemplu parcurge matricea

1 2 0

1 0 2

0 1 2

... şi concatenează elementele sale în variabila $a care, în final, va avea valoarea "120102012". Se observă că parcurgerea unei matrici cu dimensiunile 3x3 se realizează prin indexare începând de la zero, deci indicii $i şi $j iau valori de la 0 la 2. Dacă dorim să se concateneze doar valorile nenule din matrice, vom folosi ContinueLoop:

Dim $V[3][3]=[[1,2,0],[1,0,2],[0,1,2]]

$a=""

For $i=0 to 2

For $j=0 to 2

If $V[$i][$j]=0 Then ContinueLoop

$a&=$V[$i][$j]

Next

Next

De data aceasta, valoarea finală a variabilei $a va fi "121212". Aşadar, la fiecare apariţie a lui zero s-a evitat iteraţia respectivă (deci concatenarea). În acest exemplu, ContinueLoop se referă la iteraţiile ciclului interior, cu contorul $j.

Dim $V[3][3]=[[1,2,0],[1,0,2],[0,1,2]]

$a=""

For $i=0 to 2

For $j=0 to 2

If $V[$i][$j]=0 Then ContinueLoop 2

$a&=$V[$i][$j]

Next

Next

În acest exemplu, ContinueLoop se referă la al doilea nivel de imbricare, deci la ciclul exterior (cu contorul $i) – numărarea nivelelor de imbricare are loc dinspre interior spre exterior. Aceasta înseamnă că la apariţia unui zero, întregul ciclu $j va fi oprit şi se va trece direct la următoarea iteraţie a ciclului $i. Deci, la fiecare apariţie a lui zero se va trece la un rând nou al matricii. Valoarea finală a variabilei $a va fi "121" (diagonala secundară şi elementele de sub ea vor fi ignorate).

O comandă similară este ExitLoop. Dacă ContinueLoop forţează încheierea iteraţiei curente, ExitLoop forţează terminarea ciclului curent. O valoare numerică poate indica nivelul de imbricare la care se referă comanda. Exemplul anterior are acelaşi rezultat cu cel care urmează:

Dim $V[3][3]=[[1,2,0],[1,0,2],[0,1,2]]

$a=""

For $i=0 to 2

For $j=0 to 2

If $V[$i][$j]=0 Then ExitLoop

$a&=$V[$i][$j]

Next

Next

Echivalenţa se datorează faptului că terminarea ciclului $j (ExitLoop) este echivalentă cu terminarea unei iteraţii a ciclului $i (ContinueLoop 2). Dacă în acest exemplu folosim ExitLoop 2, la apariţia primului zero se va forţa terminarea ciclului al doilea, $i (şi implicit a ciclului interior). Rezultatul stocat în variabila $a va fi "12" (parcurgerea matricii se încheie la apariţia primului zero).

Instrucţiunile de terminare forţată a unei iteraţii sau ciclu sunt utile în special în ciclurile infinite create intenţionat astfel, atunci când condiţia de terminare e prea complexă sau incomodă pentru a fi reprezentată printr-o expresie logică. Ciclul infinit se creează cu valoarea True în loc de condiţia While sau (False în loc de condiţia Do Until) şi în interiorul său o structură IF sau CASE va defini condiţia excepţională de oprire:

While 1

....

If ... Then ExitLoop

....

Wend

O altă instrucţiune de terminare forţată, dar care nu e legată strict de structurile repetitive, este Exit. Această opreşte definitiv macroul curent. Ca şi ExitLoop, e folosită cu precădere pentru a provoca terminări forţate ale unor cicluri infinite (vezi capitolul de programare GUI), dar ExitLoop e folosit atunci când în afara ciclului a mai rămas ceva de executat.

Instrucţiunea Exit poate fi însoţită de un cod de terminare ales de programator. Mai multe apariţii ale lui Exit, în diverse puncte ale algoritmului, vor genera astfel diferite coduri de terminare, ce pot fi preluate de o funcţie şi testate în vederea realizării unor operaţii finale, chiar înaintea terminării (afişare de mesaje de terminare în funcţie de motivul terminării, salvări finale ale unor date etc.). Mai multe detalii sunt prezentate în capitolul privind programarea GUI prin evenimente.

**Structurile de parcurgere a proprietăţilor obiectelor COM** sunt:

* FOR In, parcurge colecţii de proprietăţi (şi colecţii de elemente ale masivelor)
* With, permite parcurgerea facilă a proprietăţilor unui acelaşi obiect (care nu sunt neapărat organizate în colecţii)

Structura FOR In nu iterează o variabilă într-un interval de valori, ci într-o colecţie de elemente (elemente ale unui masiv sau ale unei colecţii de proprietăţi obiectuale).

Dim $V[3]=[1,2,3]

$a=""

For $i in $V

$a&=$i

Next

Ca şi în cazul FOR Next, variabila de control a iteraţiilor nu trebuie declarat şi e considerată variabilă cu vizibilitate locală. În acest exemplu, variabila $a acumulează prin concatenare valorile elementelor din vectorul $V, având în final valoarea "123".

Dacă masivul sau colecţia parcursă nu conţin nici un element, structura FOR este ignorată de interpretor. Dacă variabila-colecţie nu este nici masiv, nici colecţie de proprietăţi, se generează o eroare ce poate fi tratată cu un handler COM Error. Exemplul următor prezintă modul de parcurgere a unei colecţii de proprietăţi:

$obiect = ObjCreate("shell.application")  
$colectie=$obiect.windows  
  
$a=""

For $fereastra in $colectie  
    $a &= $fereastra.LocationName & @CRLF  
Next

Acest exemplu sugerează posibilităţile obiectuale AutoIt, mai precis posibilitatea de a accesa obiecte COM şi proprietăţile acestora cu ajutorul variabilelor. Prin funcţia ObjCreate se creează o referinţă spre un obiect COM (instanţă a clasei dintre paranteze). În acest caz e vorba de aplicaţia shell curentă a sistemului de operare (Windows Explorer). Colecţia windows a acestui obiect conţine toate ferestrele shell deschise. Această colecţie e parcursă prin variabila de control $fereastra. Pentru fiecare fereastră shell deschisă, se concatenează numele directorului curent din aceasta (din proprietatea LocationName) şi un salt la rând nou. În final, variabila $a va acumula prin concatenare numele folderelor curente din fiecare fereastră Windows Explorer deschisă.

Se recomandă ca orice instanţiere a unui obiect COM să fie urmată de verificarea succesului său, deoarece nu orice calculator oferă aceleaşi obiecte COM. Verificarea se va realiza adăugând o structură condiţională în exemplul anterior:

$obiect = ObjCreate("shell.application")  
$colectie=$obiect.windows

If Isobj($colectie) Then

$a=""

For $fereastra in $colectie  
    $a &= $fereastra.LocationName & @CRLF  
Next

Endif

Funcţia Isobj returnează 1 dacă argumentul este obiect sau colecţie de obiecte şi 0 dacă nu.

Structura With are rolul de a scurta referinţele lungi la proprietăţile obiectelor COM, atunci când se intenţionează operarea cu mai multe proprietăţi ale aceluiaşi obiect. Acel obiect se indică o singură dată cu la începutul structurii With şi va fi considerat părinte al tuturor proprietăţilor necalificate din structură:

$excel = ObjCreate("Excel.Application")  
$excel.visible =1  
$excel.workbooks.add  
  
With **$excel.activesheet**

.cells(2,2).value = 1

Sleep (2000)

.range("A1:B2").clear  
EndWith  
  
$excel.quit

Acest exemplu creează o instanţă COM a clasei Excel.Application, deci lansează o instanţă a aplicaţiei Excel. Aplicaţia devine vizibilă (proprietatea visible) şi se creează un document nou în cadrul acesteia (metoda add a proprietăţii workbooks).

Structura With declară o referinţă părinte pentru blocul de cod încadrat. Asta înseamnă că toate calificările din blocul de cod se vor subordona implicit obiectului indicat de referinţa With. Într-un limbaj informal, structura With s-ar traduce în felul următor:

Cu foaia Excel activă realizează operaţiile:

* Valorii celulei 2,2 i se atribuie valoarea 1;
* Aşteaptă 2 secunde (funcţia AutoIt Sleep);
* Domeniul de celule A1:B2 este şters.

Se poate concluziona că încadrarea structurii With la structuri repetitive e discutabilă. With nu facilitează repetitivitatea unor operaţii (asemeni celorlalte structuri), ci repetitivitatea calificărilor, când acestea se referă la un acelaşi obiect.

**X.3.7. Comentarii**

**Comentariile** au în general rolul de a mări claritatea şi lizibilitatea codului sursă dar sunt frecvent folosite ca metodă de dezactivare temporară a unor porţiuni de cod sursă fără a le şterge. În acest scop, AutoIt permite definirea a două tipuri de comentarii:

* Comentariile linie – orice apariţie a caracterului ";" (în afara unei valori string) are ca efect faptul că orice şir de caractere care urmează, până la capătul liniei, va fi considerat comentariu (deci ignorat de interpretor şi compilator). Astfel, e posibilă dezactivarea unei linii de cod sursă prin plasarea caracterului ";" la începutul său;
* Comentariile bloc – orice linii care sunt încadrate de directivele de delimitare #cs şi #ce sunt considerate comentarii. Directivele de delimitare pot dezactiva blocuri de linii de cod consecutive. Blocurile de comentarii se pot imbrica.

Se practică frecvent şi îmbinarea celor două metode. Apariţia unui caracter ";" în faţa directivei #cs, are ca efect convertirea blocului de comentarii în instrucţiuni AutoIt, ignorându-se apariţia directivei de închidere #ce.

Exemplu:

$a=1

#cs

$a+=1

#ce

$b=$a

În acest exemplu, atribuirea dintre directivele #cs şi #ce este o linie de cod dezactivată (convertită la comentariu), aşadar variabila $b va avea în final valoarea 1.

$a=1

; #cs

$a+=1

#ce

$b=$a

În acest exemplu, variabila $b va avea în final valoarea 2, deoarece a doua atribuire a fost activată prin dezactivarea directivei de deschidere a blocului de comentarii. Activarea şi dezactivarea temporară a anumitor linii sau blocuri de cod sursă e o tehnică frecvent folosită în testare şi depanare, pentru a izola anumite structuri de execuţie (ramuri Case, diverse structuri repetitive etc.).

**X.3.8. Funcţii ale utilizatorului**

**Funcţiile utilizatorului** sunt module de program (macro în acest caz) care pot fi programate şi izolate sub formă de funcţii în scopul reutilizării facile: acceptă intrări – *argumente, parametri* şi returnează ieşiri - *valoarea funcţiei*. Funcţiile utilizatorului se execută în acelaşi regim cu funcţiile predefinite AutoIt – prin *apel de funcţie*, care include argumentele ce vor servi ca parametrii de intrare, într-o ordine stabilită de definiţia funcţiei. Orice funcţie a utilizatorului returnează o valoare! *Dacă aceasta nu e precizată de funcţie va fi în mod implicit 0*. Structura de programare care constituie definiţia unei funcţii este sugerată de exemplul:

Func functiaMea ($p1, $p2)

; blocul de instrucţiuni al funcţiei

Return $v

Endfunc

Apoi, în cadrul programului principal, sunt posibile apeluri ale funcţiei[[22]](#footnote-22) cu argumente de diverse tipuri (preferabil tipurile aşteptate de funcţie pentru a evita conversiile implicite imprevizibile):

functiaMea(5,"a") ; executarea funcţiei fără să conteze valoarea returnată

$a=functiaMea(2,"pp") ; executarea funcţiei şi preluarea valorii returnate pentru

; utilizări viitoare

Elementele exemplului sunt:

* functiaMea este **numele funcţiei**, ales de programator;
* $p1, $p2 sunt intrările funcţiei: doi **parametri** care vor primi valori de la argumentele din apelul funcţiei; valorile lor vor fi utilizate de blocul funcţiei în diverse prelucrări;
* Return e **punctul de finalizare** a execuţiei funcţiei (comenzile din blocul funcţiei care urmează logic după Return vor fi ignorate);
* $v este ieşirea funcţiei: **valoarea returnată**; aceasta va fi un rezultat al operaţiilor din blocul funcţiei şi va putea fi utilizată la apelul funcţiei;
* (5,"a") şi (2,"pp") sunt două perechi de **argumente** ale funcţiei, corespunzătoare celor două **apeluri** ale funcţiei; argumentele vor transmite valori parametrilor de intrare:
  + $p1 va lua la primul apel valoarea 5, la al doilea valoarea 2
  + $p2 va lua la primul apel valoarea "a", la al doilea valoarea "pp"
* funcţia trebuie să fie pregătită să primească şi să prelucreze orice valori posibile ale argumentelor, eventual să trateze cazurile excepţionale prin testarea parametrilor la începutul blocului funcţiei.

Programul principal poate să lipsească, caz în care fişierul AutoIt nu mai poate fi considerat un macro, ci o **bibliotecă de funcţii** reutilizabile. Executarea sa nu va avea nici un efect (fără program principal nu există nici apeluri de funcţii), însă fişierul va putea fi inclus în alte macrouri care au nevoie de funcţiile din bibliotecă.

Numărul parametrilor nu este impus, poate fi chiar zero, caz în care funcţia se va apela fără argumente. Instrucţiunea Return poate lipsi, caz în care funcţia returnează zero. Oricare din parametri şi chiar valoarea returnată pot fi variabile masive.

Func adunare(**$p1,$p2,$p3**) ; definiţia parametrilor funcţiei (intrările)

**Return $p1+$p2+$p3** ; returnarea valorii funcţiei (ieşirea)

Endfunc

Dim $A[3]=[1,2,3]

$Suma=**adunare($A[0],$A[1],$A[2])** ; apelul funcţiei (transmiterea intrărilor)

Acest exemplu apelează o funcţie în scopul de a calcula suma elementelor vectorului $A. Se observă că:

* numărul valorilor-argument transmise la apelul funcţiei coincide cu numărul parametrilor de intrare definiţi;
* valorile parametrilor sunt implicate în calculul sumei (blocul funcţiei), iar rezultatul este returnat ca valoare a funcţiei;
* în urma apelului, valoarea funcţiei e atribuită variabilei $Suma.

Acelaşi rezultat se obţine dacă în loc să se transmită 3 parametri, se defineşte şi transmite un singur parametru de tip variabilă masivă:

Func adunare(**$V**)

$S=0

For $i=0 to Ubound(**$V**)-1

$S+=**$V**[$i]

Next

Return **$S**

Endfunc

Dim $A[3]=[1,2,3]

$Suma=adunare(**$A**)

Rezultatul obţinut este acelaşi cu al exemplului anterior, diferă doar metoda:

* s-a definit şi transmis un singur argument (parametru), de tip vector;
* blocul funcţiei conţine un ciclu de parcurgere a vectorului în cadrul căruia se calculează sum aprin acumulare.

Funcţiile utilizatorului pot folosi două tipuri de parametri (şi argumente), similar cu funcţiile predefinite:

* parametrii obligatorii sunt definiţi ca în exemplele anterioare, valorile lor trebuie transmise obligatoriu ca argumente în apelul funcţiei, altfel funcţia eşuează şi returnează 0;
* parametrii opţionali sunt definiţi la sfârşitul listei parametrilor prin atribuiri de valori implicite (folosite atunci când apelul funcţiei neglijează argumentele opţionale).

Func adunare($V, **$limita=4**)

$S=0

For $i=0 to $limita

$S+=$V[$i]

Next

Return $S

Endfunc

Dim $A[5]=[1,2,3,4,5]

**$n=2**

$SumaLimitata=adunare(**$A,$n**)

$SumaTotala=adunare(**$A**)

În acest exemplu, avem două apeluri, primul cu valoarea 2 pentru argumentul opţional, al doilea neglijează argumentul opţional (care va primi implicit valoarea 4). Argumentul opţional va fi transmis parametrului $limita care e folosit în funcţie pentru a indica valoarea finală, de oprire a ciclului FOR. În final:

* variabila $SumaLimitata va stoca valoarea 6 (suma elementelor cu indicii 0-2 din vector, conform argumentului opţional);
* variabila $SumaTotala va stoca valoarea 15 (suma tuturor elementelor vectorului, cu indicii 0-4, conform valorii implicite a argumentului neglijat de al doilea apel);

O altă versiune a acestui mecanism se obţine folosind cuvântul cheie **Default**, care este un locţiitor temporar pentru valoarea implicită a unui parametru.

Func adunare($V, **$limita=Default**)

**If $limita=Default Then $limita=Ubound($V)-1**

$S=0

For $i=0 to $limita

$S+=$V[$i]

Next

Return $S

Endfunc

Dim $A[5]=[1,2,3,4,5]

$n=2

$SumaLimitata=adunare($A,$n)

$SumaTotala=adunare($A)

Valoarea Default va fi atribuită implicit atunci când argumentul opţional lipseşte din apelul funcţiei. Blocul funcţiei tratează lipsa argumentului (deci prezenţa valorii Default) printr-o structură IF care atribuie o valoare de lucru implicită parametrului. Valoarea de lucru atribuită în acest exemplu este indicele ultimului element din vector, utilizat apoi în parcurgerea prin FOR. Avantajul acestei metode este că structura IF permite să se atribuie parametrului opţional şi expresii în care intervin parametrii obligatorii. Spre exemplu, în lipsa metodei Default, nu ar fi fost posibilă o definiţie ca cea de mai jos:

Func adunare($V, **$limita=Ubound($V)-1**)

...deoarece valoarea parametrului $V nu este disponibilă decât începând cu prima linie a blocului funcţiei!

Datorită parametrilor opţionali, e posibil ca numărul de argumente cu care s-a apelat funcţia să fie mai mic decât numărul parametrilor din definiţia funcţiei. În blocul funcţiei se poate folosi macrovariabila @NUMPARAMS pentru a obţine numărul argumentelor transmise de apelul curent al funcţiei:

Func adunare ($p1,$p2=0,$p3=0)

$S=$p1+$p2+$p3

Return "Suma celor " & @NUMPARAMS & " argumente este " & $S

EndFunc

MsgBox(0,"",adunare(2,3))

Rezultatul afişat în caseta de dialog va fi "Suma celor 2 argumente este 5"

Parametrii unei funcţii pot fi:

* de tip **read-write**, implicit, caz în care valoarea lor poate fi modificată în blocul funcţiei;
* de tip **read-only**, dacă în definiţia funcţiei parametrul e însoţit de cuvântul Const, caz în care valoarea sa poate fi DOAR citită în blocul funcţiei (parametrului nu i se vor putea atribui valori).

Func adunare($V, **$limita**)

$limita=2

$S=0

For $i=0 to $limita

$S+=$V[$i]

Next

Return $S

Endfunc

Dim $A[5]=[1,2,3,4,5]

$Suma=adunare(**$A,4**)

În acest exemplu parametrul $limita este de tip read-write: deşi i s-a transmis valoarea 4, în blocul funcţiei îi este modificată valoarea la 2, aşadar se calculează doar suma elementelor cu indicii 0-2. Dacă definiţia funcţiei ar arăta astfel:

Func adunare($V, **Const** **$limita**)

...atunci ar apare o eroare la atribuirea:

$limita=2

...datorită faptului că se încearcă modificarea unui parametru read-only. Dacă se doreşte transmiterea unei constante ca argument al unei funcţii, parametrul corespunzător trebuie obligatoriu să fie declarat de tip Const.

Argumentele variabile pot fi transmise spre o funcţie prin două metode:

* **prin valoare**, implicit, caz în care parametrul şi argumentul sunt considerate variabile diferite cu aceeaşi valoare (parametrului i se atribuie valoarea variabilei argument); parametrul va fi considerat o variabilă nouă, locală funcţiei, care conţine o copie a valorii argumentului;
* **prin referinţă**, dacă în definiţia funcţiei apare cuvântul cheie ByRef, caz în care parametrul şi argumentul sunt una şi aceeaşi variabilă; modificările aduse la parametru vor afecta şi variabila-argument.

Exemplu:

Func f1($p)

$pini=$p ; memorarea valorii iniţiale a parametrului

**$p\*=2 ; modificarea parametrului**

$rez=$p

MsgBox(0,"mesaj din functie" \_

,"Valoarea parametrului la intrare in functie:" & $pini & @CRLF & \_

"Valoarea parametrului la iesirea din functie:" & $p)

Return $rez ; valoarea returnată (dublul parametrului)

Endfunc

$arg=5

$argini=$arg ; memorarea valorii iniţiale a argumentului

**$dublu=f1($arg) ; apelul funcţiei**

Msgbox(0,"mesaj din afara functiei","Val returnata:" & $dublu & @CRLF \_

&"Val argumentului la apel:"& $argini & @CRLF \_

& "Val argumentului dupa apel:" & $arg)

S-a exemplificat transmiterea argumentelor prin valoare. Explicaţii:

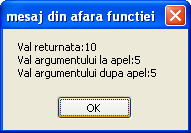
* apelul de funcţie transmite argumentul cu valoarea 5;
* blocul funcţiei calculează dublul argumentului şi îl returnează.

Datorită transferului prin valoare, $arg şi $p sunt considerate variabile diferite. Modificarea parametrului $p în blocul funcţiei, nu afectează variabila argument. Acest lucru e verificabil în cele două mesaje generate:

* din interiorul funcţiei se generează valoarea parametrului la intrare şi la finalizarea funcţiei (se observă modificarea datorită atribuirii $p\*=2);



* după apelul funcţiei se generează rezultatul returnat, valoarea iniţială a argumentului şi valoarea finală a argumentului (se observă că a rămas 5).



În cazul transferului prin valoare, parametrii funcţiei vor fi creaţi ca variabile Local, deci vizibile doar în interiorul funcţiei, indiferent dacă există variabile globale cu acelaşi nume. Aşadar, e posibil ca argumentul şi parametrul să aibă acelaşi nume – ele vor fi considerate variabile diferite, cu vizibilitate diferită! Aşadar, exemplul anterior oferă acelaşi rezultate dacă numele parametrului este identic cu al argumentului:

Func f1(**$arg**)

$pini=**$arg** ; memorarea valorii iniţiale a parametrului

**$arg\*=2** ; modificarea parametrului

$rez=**$arg**

MsgBox(0,"mesaj din functie" \_

,"Valoarea parametrului la intrare in functie:" & $pini & @CRLF & \_

"Valoarea parametrului la iesirea din functie:" & **$arg**)

Return $rez ; valoarea returnată (dublul parametrului)

Endfunc

**$arg**=5

$argini=**$arg** ; memorarea valorii iniţiale a argumentului

$dublu=f1**($arg)** ; apelul funcţiei

Msgbox(0,"mesaj din afara functiei","Val returnata:" & $dublu & @CRLF \_

&"Val argumentului la apel:"& $argini & @CRLF \_

& "Val argumentului dupa apel:" & **$arg**)

Din cele două casete de dialog obţinute, se remarcă faptul că valoarea argumentului după apel rămâne cea iniţială ($arg=5), deşi o variabilă cu acelaşi nume, a fost modificată în interiorul funcţiei ($arg\*=2). Transferul prin valoare face ca cele două variabile $arg să fie considerate variabile diferite. Variabila locală[[23]](#footnote-23) $arg (şi modificarea suferită de aceasta) nu sunt vizibile în afara funcţiei! Acesta este şi un exemplu privind faptul că variabilele locale permit evitarea conflictelor de nume prin diferenţa de vizibilitate, CHIAR şi când argumentul şi parametrul au acelaşi nume.

La transferul prin referinţă, în schimb, argumentul şi parametrul sunt una şi aceeaşi variabilă, CHIAR dacă au nume diferite! Orice modificare suferită în cadrul funcţiei de parametru, se răsfrânge asupra argumentului. Singura schimbare introdusă în exemplul anterior va fi definirea parametrului prin ByRef:

Func f1(**ByRef $p**)

$pini=$p ; memorarea valorii iniţiale a parametrului

**$p\*=2 ; modificarea parametrului**

$rez=$p

MsgBox(0,"mesaj din functie" \_

,"Valoarea parametrului la intrare in functie:" & $pini & @CRLF & \_

"Valoarea parametrului la iesirea din functie:" & $p)

Return $rez ; valoarea returnată (dublul parametrului)

Endfunc

$arg=5

$argini=$arg ; memorarea valorii iniţiale a argumentului

**$dublu=f1($arg) ; apelul funcţiei**

Msgbox(0,"mesaj din afara functiei","Val returnata:" & $dublu & @CRLF \_

&"Val argumentului la apel:"& $argini & @CRLF \_

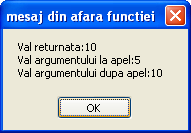
& "Val argumentului dupa apel:" & $arg)

Interpretorul consideră că $p şi $arg sunt aceeaşi variabilă. Modificarea suferită de $p are loc şi asupra lui $arg, ceea ce se poate verifica prin cele două mesaje:

* primul mesaj e identic cu cazul anterior, indică faptul că parametrul şi-a modificat valoarea:



* al doilea mesaj e schimbat – de data aceasta indică faptul că modificarea parametrului $p s-a aplicat şi argumentului $arg (mai exact, cele două nume de variabile reprezintă aceeaşi locaţie de memorie, cu aceeaşi informaţie).



Pentru claritatea codului sursă, se practică adesea ca:

* la transferul prin valoare, parametrul să primească alt nume decât argumentul (pentru a sugera că sunt variabile diferite);
* la transferul prin referinţă, parametrul să primească acelaşi nume cu argumentul (pentru a sugera că e vorba de aceeaşi variabilă).

Este important de reţinut că definiţiile funcţiilor utilizatorului NU pot fi imbricate între ele! Aşadar, nu e posibilă o construcţie precum:

Func f1(...)

Func f2(....)

; blocul funcţiei f2

Endfunc

; blocul funcţiei f1

Endfunc

În schimb, funcţiile utilizatorului pot conţine apeluri ale altor funcţii ale utilizatorului, ca în exemplul:

Func f1(...)

;blocul funcţiei f1

f2()

Endfunc

Func f2(...)

;blocul funcţiei f2

Endfunc

; aici începe programul principal

......

**$a=f1(...) ; apel al funcţiei f1, care la rândul ei va apela funcţia f2**

Lucrul cu funcţii ale utilizatorului acordă o importanţă crescută modului de declarare a variabilelor şi vizibilităţii acestora. O variabilă declarată de o funcţie ca fiind Local sau Dim, va fi vizibilă doar în acea funcţie (dacă e Local, va evita conflictele de nume, permiţând folosirea unui nume de variabilă care există deja în afara funcţiei). În exemplificarea vizibilităţii vom apela la funcţia IsDeclared, care testează existenţa (deci vizibilitatea) unei variabile în punctul dorit al programului. Pornim de la un exemplu similar celui precedent:

Func f1($p)

$pini=$p ; memorarea valorii initiale a parametrului

$p\*=2 ; modificarea parametrului

$rez=$p

MsgBox(0,"Vizibilitate in interiorul functiei a variabilei pini:",IsDeclared("pini"))

Return $rez ; valoarea returnata (dublul parametrului)

Endfunc

$arg=5

$argini=$arg ; memorarea valorii initiale a argumentului

$dublu=f1($arg) ; apelul functiei

MsgBox(0,"Vizibilitate in afara functiei a variabilei pini:",IsDeclared("pini"))

Variabilele locale ale funcţiei f1 sunt:

* $p (s-a arătat deja că parametrii transferaţi prin valoare sunt consideraţi variabile locale ale funcţiei);
* $pini (valoarea iniţială a parametrului), deoarece este iniţializată direct în funcţie şi nu are instrucţiune de declarare (ceea ce înseamnă că e declarată cu Dim, deci locală[[24]](#footnote-24));
* $rez din aceleaşi motive ca $pini – declarată prin Dim (implicit) direct în funcţie.

Caracterul local al celor trei variabile poate fi confirmat prin faptul că sunt vizibile în funcţie (deci pot apare în orice expresii din blocul funcţiei) dar sunt invizibile în afara sa (provoacă eroare de vizibilitate dacă apare în expresii din afara funcţiei). Pentru aceasta sugerăm modul de utilizare a funcţiei IsDeclared:

* Are ca argument de tip string numele unei variabile (fără $);
* Returnează -1 dacă variabila e vizibilă local;
* Returnează 1 dacă variabila e vizibilă (globală sau declarată în afara funcţiei);
* Returnează 0 dacă variabila e invizibilă.

Exemplul anterior testează vizibilitatea variabilei locale $pini, pentru $p şi $rez funcţionează identic. Vizibilitatea e testată în două puncte:

* În interiorul funcţiei (primul mesaj), testul va indica faptul că IsDeclared returnează -1, deci $pini e vizibilă local;
* În afara funcţiei (al doilea mesaj), testul va indica faptul că IsDeclared returnează 0, deci $pini e invizibilă în afara funcţiei; aceasta înseamnă că următorul exemplu va da eroare de vizibilitate:

Func f1($p)

**$pini=$p ; variabila creata local**

$p\*=2 ; modificarea parametrului

$rez=$p

Return $rez ; valoarea returnata (dublul parametrului)

Endfunc

$arg=5

$dublu=f1($arg) ; apelul functiei

MsgBox(0,"Valoarea actuala a variabilei este:", **$pini**)) ; eroare!

$a=**$pini**+3 ; eroare!

Eroarea de vizibilitate va fi detectată la ultimele două linii, cele două tentative de utilizare a variabilei în afara domeniului de vizibilitate. Problema dispare dacă $pini va fi declarată Global:

Func f1($p)

**Global $pini=$p ; variabila globala**

$p\*=2

$rez=$p

MsgBox(0,"Vizibilitate in interiorul functiei a variabilei pini:", \_

**IsDeclared("pini")**)

Return $rez

Endfunc

$arg=5

$argini=$arg ; memorarea valorii initiale a argumentului

$dublu=f1($arg) ; apelul functiei

MsgBox(0,"Vizibilitate in afara functiei a variabilei pini:",**IsDeclared("pini")**)

De data aceasta, cele două teste vor afişa:

* Valoarea 1 în mesajul din interiorul funcţiei, deci vizibilitate totală;
* Valoarea 1 în mesajul din afara funcţiei, deci vizibilitate totală.

Ce se întâmplă însă cu variabilele create în programul principal, de exemplu $argini? Vizibilitatea acesteia în interiorul funcţiei va fi testată prin exemplul:

Func f1($p)

$pini=$p

$p\*=2

$rez=$p

MsgBox(0,"Vizibilitate in interiorul functiei a variabilei $argini:", \_ **IsDeclared("argini")**)

MsgBox(0,"Vizibilitate in interiorul functiei a variabilei $b:", \_

**IsDeclared(" b")**)

Return $rez

Endfunc

$arg=5

**$argini=$arg ; variabila locala programului principal**

$dublu=f1($arg) ; apelul functiei

**$b=100 ; variabila local programului principal, creata dupa apel!**

MsgBox(0,"Vizibilitate in afara functiei a variabilei $argini:",**IsDeclared("argini")**)

MsgBox(0,"Vizibilitate in afara functiei a variabilei $b:",**IsDeclared("b")**)

Testele pentru $argini dau rezultatele:

* În interiorul funcţiei are vizibilitate globală (1);
* În afara funcţiei are vizibilitate globală (1).

Testele pentru $b dau rezultatele:

* 0: în interiorul funcţiei e invizibilă, deoarece variabila nu exista la momentul apelului funcţiei (a fost creată în instrucţiunea următoare apelului);
* 1: în afara funcţiei e vizibilă, de la momentul iniţializării până la final.

Concluzii:

* Nici o variabilă nu este vizibilă în porţiunea de algoritm care precede crearea sa (declararea sau iniţializarea);
* Orice variabilă creată în programul principal (chiar şi locală), e vizibilă în toate funcţiile apelate după crearea sa (variabilele din programul principal sunt tratate ca globale de toate funcţiile);
* Variabilele create de funcţii pot fi:
  + Globale, caz în care sunt vizibile în toate funcţiile apelate după crearea lor şi în porţiunea de program principal executată după crearea lor;
  + Locale, caz în care nu sunt vizibile decât în funcţia care le-a creat, după instrucţiunea la care a avut loc crearea. La această categorie se încadrează şi parametrii de intrare ai funcţiilor, precum şi:
    - Variabile locale declarate cu Dim (sau create prin iniţializare, fără Dim) – în cazul în care există deja o variabilă globală cu acelaşi nume, NU se creează o variabilă nouă, ci funcţia va utiliza variabila globală! *În această categorie se înscriu şi parametrii transferaţi prin referinţă*.
    - Variabile locale declarate cu Local – în cazul în care există deja o variabilă globală cu acelaşi nume, va fi ignorată, nu va fi afectată şi se va crea o variabilă nouă, cu acelaşi nume dar vizibilitate diferită! *În această categorie se înscriu şi parametrii transferaţi prin valoare*.

O categorie specială de funcţii ale utilizatorului sunt **funcţiile predefinite ale utilizatorului**. E vorba de funcţii construite după principiile prezentate în acest capitol, dar care sunt publicate în cadrul pachetului AutoIt, în regim open-source, de către diverşi utilizatori. Sunt *funcţii ale utilizatorului* deoarece nu sunt recunoscute implicit de interpretor (necesită copierea definiţiilor de funcţii din directorul Include, în care sunt stocate) şi sunt *funcţii predefinite*, deoarece sunt publicate în pachetul AutoIt, gata de a fi reutilizate, inclusiv cu comentarii şi documentaţie Help anexate. Funcţiile predefinite ale utilizatorului vor fi tratate într-un subcapitol special.

**X.3.9. Directive**

**Directivele** sunt instrucţiuni care nu au un impact direct asupra algoritmului AutoIt, ci configurează modul său de execuţie. Numărul directivelor AutoIt este redus, cel mai relevant aspect fiind posibilitatea de includere a fişierelor externe în codul sursă al unui macro.

**Directivele #cs şi #ce** delimitează un bloc de comentarii AutoIt, aşadar o porţiune a codului sursă care va fi ignorată de intepretor. Structurile #cs-#ce pot fi imbricate. Exemplu:

#cs

Comentariu sau cod sursă ignorat

#ce

**Directiva #NoTrayIcon** indică faptul că pictograma AutoIt nu va apare în zona Tray a barei de sarcini Windows (aşa cum se întâmplă în mod implicit la executarea unui macro). Dezactivarea directivei poate fi realizată prin funcţia Opt("TrayIconHide",0). Între apariţia directivei şi apelul funcţiei, pictograma AutoIt de pe bara Tray va fi invizibilă:

#NoTrayIcon

Sleep(5000)

Opt("TrayIconHide",0)

Sleep(5000)

Acest exemplu lansează un macro fără pictogramă Tray, apoi după 5 secunde apare pictograma, apoi după alte 5 secunde dispare datorită opririi execuţiei.

**Directiva #RequireAdmin** specifică faptul că rularea macroului va fi posibilă doar sub contul de administrator.

**Directiva #include** permite concatenarea unui fişier extern cu macroul curent, la poziţia la care apare directiva. Motivele includerii unui fişier extern pot fi multiple:

* Includerea unor porţiuni de cod sursă reutilizabil memorate în alt fişier;
* Includere unor fişiere de comentarii;
* Includerea unor definiţii de funcţii memorate în alte fişiere, spre exemplu importarea bibliotecilor funcţiilor predefinite ale utilizatorului (ataşate pachetului AutoIt în directorul Include). Fără această incluziune, funcţiile respective nu pot fi executate de interpretor, nefiind recunoscute implicit de acesta.

Formele directivei sunt:

#include "Cale "

#include <fisier.extensie>

Prima versiune permite să se indice calea completă a fişierului care se va include. A doua versiune este folosită pentru includerea bibliotecilor de funcţii. Funcţionează în condiţiile în care:

* bibliotecile sunt stocate în directorul Include (al directorului de instalare AutoIt);
* bibliotecile sunt stocate în acelaşi director cu macroul care le solicită;
* dacă sunt stocate altundeva, acel director trebuie să fie precizat în registrul Windows HKEY\_CURRENT\_USER\Software\AutoIt v3\AutoIt (atributul Include);

În cazul acestei forme, există riscul ca acelaşi fişier să fie inclus de mai multe ori (dacă apar copii ale sale atât în Include cât şi în celelalte două locaţii), ceea ce va provoca o eroare de *funcţie dublu definită*; evitarea acestui risc se realizează folosind directiva #include-once[[25]](#footnote-25).

**Directiva #include-once** nu apare în macroul care solicită includerea, ci pe prima linie a fişierului care trebuie inclus! Apariţia sa previne dubla includere a fişierului care conţine directiva. Exemplu:

Codul fişierului BibliotecaMea.au3 (salvat în directorul AutoIT3/Include):

**#include-once ; previne dubla includere a fişierului**

Func functiaMea()

MsgBox(0,"","Functie apelata din fisierul inclus!")

Endfunc

Codul fişierului MacroPrincipal.au3:

#include <BibliotecaMea.au3>

**#include <BibliotecaMea.au3> ; eroare potentiala de functie dublu definita!**

MsgBox(0,"","Urmeaza apelul functiei incluse!")

functiaMea()

Executarea acestui exemplu ar provoca eroare de funcţie dublu definită dacă BibliotecaMea.au3 nu ar conţine directiva #include-once. Prezenţa directivei face ca a doua directivă #include, în loc să provoace eroarea, să fie ignorată.

O metodă alternativă de includere este folosirea funcţiilor de acces la disc, în locul directivei #include (ex: funcţia FileInstall). Spre deosebire de #include, care realizează includerea la nivel de interpretor, FileInstall asigură includerea la momentul compilării.

**Programarea de interfeţe grafice în AutoIt**

**Concepte generale GUI**

**AutoIt** oferă posibilitatea de a crea **interfeţe grafice (GUI)**. În unele medii de programare, acestea se reduc la conceptul de **formulare**, datorită posibilităţii de a colecta date prin intermediul lor (formulare HTML, Word, generatoare de formulare Access, Fox etc.). AutoIt permite construirea de GUI prin programare sau prin generatorul de formulare Koda Form Designer (meniul Tools). Unele elemente GUI pot fi generate şi cu ajutorul Tools - Code Wizard, un wizard capabil să genereze codul sursă corespunzător funcţiilor ce asigură dialogul cu utilizatorul (MsgBox, InputBox, ToolTip etc.).

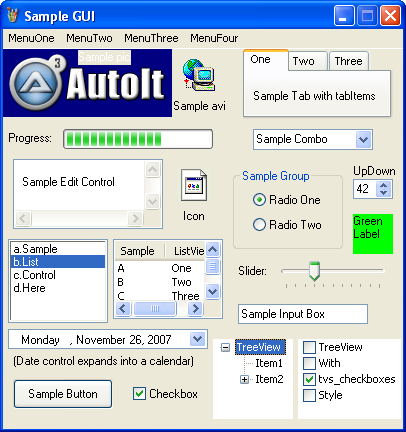
O GUI e alcătuită dintr-o succesiune de ferestre şi fiecare fereastră poate conţine un formular. Un formular e alcătuit din **obiecte GUI** (numite în jargon *controale*). O GUI poate fi programată pe bază de evenimente.

**Evenimentele** sunt stările prin care trec obiectelor GUI, provocate prin programare sau de către acţiunile utilizatorului asupra GUI (clic, tastare, dublu clic etc.). Fiecare obiect GUI poate fi programat pentru a reacţiona la gama dorită de evenimente.

Obiectele GUI ce pot fi create în AutoIt sunt cele definite de sistemul de operare Windows, conform ghidului de utilizabilitate a interfeţelor Windows, publicat de Microsoft. Acestea sunt[[26]](#footnote-26):

* Label – etichetă pentru afişare de text;
* Button – buton;
* Input – casetă de editare a textului, pentru introducere de date pe o singură linie;
* Edit – zonă multi-linie de editare a textului, pentru introducerea de date;
* Checkbox – casetă de validare cu două stări: bifat, nebifat;
* Radio – set de opţiuni semnalate prin butoane circulare din care poate fi selectată una singură;
* Combo – casetă cu listă derulantă;
* List – listă;
* Date – selector de date calendaristice;
* Pic – imagine;
* Icon – pictogramă;
* Group – chenar de încadrare a altor obiecte GUI;
* Progress – bară de progres;
* Tab – grup de etichete tabulatoare (pagini de opţiuni);
* TabItem – pagină de opţiuni identificată prin etichetă tabulatoare;
* UpDown – săgeţi ce pot fi ataşate unui obiect Input pentru derulare de opţiuni;
* Avi – clip video;
* Menu – meniu;
* MenuItem – opţiune de meniu;
* ContextMenu – meniul declanşate de clic dreapta;
* TreeView – structură arborescentă (similară cu panoul stâng al Windows Explorer);
* TreeViewItem – un element al unei structuri TreeView;
* Slider – glisor (similar cu selectorul volumului de sunet);
* ListView – structură de afişare tabelară;
* ListViewItem – un element al unei structuri ListView;
* Graphic – desen generat în AutoIt cu GUICtrlSetGraphic;
* Dummy – pseudo-obiect GUI;
* Obj – obiecte ActiveX.

Majoritatea obiectelor GUI sunt vizibile în captura de ecran următoare, rezultat al executării exemplului SampleControls.au3 (din directorul Examples\GUI).



În general alegerea obiectelor GUI se realizează în funcţie de evenimentele care se doresc programate: interacţiunile pe bază de clic se implementează prin butoane de diferite tipuri, interacţiunea bazată pe drag se implementează prin Slider, interacţiunea prin taste se implementează prin zone de editare ce preiau date de la tastatură. Aceasta este însă doar practica uzuală. Nu există nici un motiv tehnic care să împiedice programarea evenimentelor în mod neobişnuit: clic pe o etichetă de text sau drag pe suprafaţa unui buton. Totuşi, programarea GUI tinde să urmeze recomandările specialiştilor în utilizabilitate şi a ghidurilor create de Microsoft şi Apple pe baza cercetărilor în domeniul psihologiei utilizatorului. Aceste recomandări sugerează ca fiind esenţial caracterul intuitiv al GUI, prin care utilizatorul să poată manipula obiectele GUI cu un efort de învăţare minim. Din acest motiv în mod uzual butoanele se apasă, etichetele Label afişează text, zonele de editare permit să se introducă text, etc.

Se ridică problema identificatorilor atunci când se lucrează cu GUI (fie că e vorba de a crea o GUI cu AutoIt, fie că e vorba de a manipula prin macrouri GUI ale altor aplicaţii). În oricare din aceste situaţii se pune problema identificării unice a multiplelor ferestre sau a multiplelor obiecte GUI conţinute de acestea. În mod intuitiv, un utilizator uman le identifică după **textul obiectelor** – titlul ferestrei, textul unui buton, eticheta asociată unei liste de opţiuni, numele unui meniu etc. Problema cu identificarea prin text este că:

* pot exista mai multe obiecte sau ferestre cu acelaşi text (ex: dacă se lansează Notepad de 10 ori, se creează 10 ferestre cu titlul "Untitled – Notepad");
* textul unui obiect sau ferestre poate fi modificat în mod dinamic pe parcursul execuţiei!

Pentru a rezolva problema identificării GUI, sistemul de operare Windows foloseşte propriul mecanism de identificare, prin coduri hexazecimale unice numite *Winhandles* (*control handle* pentru obiecte, *window handle* pentru ferestre). Materialul de faţă va folosi pentru aceste coduri termenul **identificatori GUI,** care constituie un tip special de date ce pot fi atribuite variabilelor, pot fi convertite cu funcţia HWnd şi pot fi testate cu funcţia IsHWnd (vezi funcţiile orientate pe tipizarea datelor şi subcapitolul dedicat variabilelor). E important de reţinut că identificatorii GUI sunt generaţi la crearea GUI, deci nu îşi păstrează valoarea între două execuţii ale aplicaţiei ce a creat GUI (fie că e vorba de AutoIt sau de aplicaţia manipulată prin macrouri AutoIt).

Limbajul AutoIt lucrează cu identificatori GUI în două situaţii:

* creează identificatori GUI atunci când se creează o interfaţă;
* citeşte identificatori GUI ai interfeţelor existente (de exemplu, atunci când macroul manipulează alte aplicaţii).

Pentru obiectele GUI, AutoIt foloseşte şi o metodă alternativă de identificare: **identificatorul AutoIt de obiect (Control ID)**. Avantajele sale faţă de *identificatorul GUI de obiect* sunt:

* e un număr zecimal, deci mai uşor de folosit de către programator;
* valoarea sa e afişată de instrumentul AWIT, chiar şi pentru obiecte din alte aplicaţii;
* permite acces direct la obiect indiferent cărei ferestre aparţine obiectul (identificatorii GUI ai obiectelor necesită uneori cunoaşterea identificatorului GUI al ferestrei care le conţine).

Pentru lizibilitate, exemplele din acest material adaugă IDX la numele identificatorilor GUI şi ID la numele identificatorilor Control ID. De obicei ferestrele sunt accesate prin intermediul identificatorului GUI, iar obiectele prin intermediul Control ID (mai uşor de folosit iar numărul de obiecte este mai mare decât al ferestrelor).

Crearea GUI presupune etapele:

1. Includerea fişierului extern GUIConstants.au3 (din directorul Include), care este biblioteca de constante şi variabile predefinite necesare funcţiilor GUI (altele decât macrovariabilele);
2. Crearea ferestrelor (funcţia GUICreate). Funcţia de creare returnează **identificatorul GUI al ferestrei create** (cod hexazecimal) ;
3. Popularea fereastrelor cu obiecte GUI, create cu familia de **funcţii GUICtrlCreate** (numele acestor funcţii se finalizează cu numele obiectului GUI creat - spre exemplu GUICtrlCreateMenu e funcţia de creare a unui meniu). Fiecare funcţie din această familie returnează **identificatorul AutoIt de obiect** (Control ID, accesibil şi prin intermediul AWIT);
4. Afişarea ferestrelor (implicit ferestrele nu sunt vizibile) prin schimbarea stării ferestrei cu GUISetState.

După crearea GUI, o serie de funcţii permit manipularea sa:

* Ascundere, afişare şi manipulare de ferestre (GUISetState);
* Preluare de date din obiectul GUI (GUICtrlRead);
* Modificarea datelor afişate de obiectul GUI (GUICtrlSetData);
* Modificarea proprietăţilor unui obiect GUI (familia de funcţii GUICtrlSet – la numele funcţiei se adaugă tipul obiectului, similar cu familia funcţiilor de creare);
* Aşteptarea evenimentelor GUI (GUIGetMsg sau funcţii handler).

Exemplu:

#include <GUIConstants.au3> ; includere de constante predefinite

GUICreate("Fereastra mea", 200,200) ; creare fereastră

GUICtrlCreateLabel("Eticheta mea",0,0) ; populare cu obiecte

GUICtrlCreateButton("Butonul meu",50,50,50)

GUISetState(@SW\_SHOW) ; afişare fereastră

Sleep(5000)

Argumentele numerice la crearea ferestrei sunt dimensiunile;

Argumentele numerice la crearea obiectelor dau poziţia (relativ la fereastră);

Programarea GUI se poate realiza prin două metode:

* prin ascultare de mesaje (modul MessageLoop);
* prin gestionare de evenimente (modul OnEvent).

Modul implicit este MessageLoop. Activarea modului OnEvent e posibilă cu ajutorul funcţiei de configurare a interpretorului: **Opt("GUIOnEventMode",1)**. Ambele moduri de programare GUI au capabilităţi similare, diferă doar metoda.

**Programarea evenimentelor prin ascultare de mesaje**

**Ascultarea de mesaje** presupune folosirea unui ciclu infinit(!) de ascultare intermitentă a mesajelor generate de obiecte GUI. Obiectele GUI generează mesaje la fiecare schimbare a stării lor, deci la apariţia evenimentelor. Funcţia care returnează mesajul generat este **GUIGetMsg**. Ascultarea mesajelor trebuie să aibă o frecvenţă suficient de mare, astfel încât apelul acestei funcţii să aibă loc de câteva ori pe secundă. Deoarece FOR are un mecanism de prevenire a ciclurilor infinite (vezi subcapitolul dedicat structurilor repetitive), acesta va fi asigurat cu o structură precum:

While **1**

**$mesaj=GUIGetMsg()**

; urmeaza testarea mesajului si apelul functiei asociate valorilor sale

Wend

Se observă înlocuirea condiţiei de continuare While cu valoarea 1 (sau True) care garantează ciclul infinit. Ieşirea din ciclu se va realiza prin forţare (instrucţiunea Exit Loop ataşată unui buton, de exemplu). Frecvenţa de ascultare a mesajului este determinată de durata executării unei iteraţii While (de ordinul milisecundelor). Ciclul nu trebuie să conţină instrucţiuni de aşteptare (Sleep) care vor bloca temporar funcţionarea GUI. *Această metodă e potrivită atunci când singura sarcină a GUI este să aştepte acţiunile utilizatorului* *şi să reacţioneze la ele*.

Trei tipuri de evenimente sunt semnalate de funcţia GUIGetMsg:

* **Lipsa evenimentelor**, semnalată prin returnarea mesajului **0**;
* **Evenimente-obiect**, semnalate atunci când un obiect GUI îşi schimbă starea, prin returnarea valorii **Control ID** (identificatorul obiectului), creată de funcţia corespunzătoare din familia GUICtrlCreate; Valorile Control ID sunt numere pozitive;
* **Evenimente-sistem**, semnalate prin **mesaje predefinite** ca şi constante în biblioteca GUIConstants.au3:
  + $GUI\_EVENT\_CLOSE – închidere GUI;
  + $GUI\_EVENT\_MINIMIZE – minimizare GUI (prin butonul de minimizare, nu şi pe alte căi!);
  + $GUI\_EVENT\_RESTORE – restaurare GUI (prin butonul de pe taskbar);
  + $GUI\_EVENT\_MAXIMIZE – maximizare GUI (prin butonul de maximizare);
  + $GUI\_EVENT\_PRIMARYDOWN – apăsarea butonului mouse-ului;
  + $GUI\_EVENT\_PRIMARYUP – eliberarea butonului mouse-ului;
  + $GUI\_EVENT\_SECONDARYDOWN – apăsarea butonului drept al mouse-ului;
  + $GUI\_EVENT\_SECONDARYUP – eliberarea butonului drept al mouse-ului;
  + $GUI\_EVENT\_MOUSEMOVE – mişcarea cursorului;
  + $GUI\_EVENT\_RESIZED – dimensionare GUI;
  + $GUI\_EVENT\_DROPPED – finalizarea operaţiei Drag and Drop.

Exemplu:

#include <GUIConstants.au3>

GUICreate("Fereastra mea", 200,200)

GUICtrlCreateLabel("Eticheta mea",0,0)

**$IDButon**=GUICtrlCreateButton("Butonul meu",50,50)

GUISetState(@SW\_SHOW)

While 1

$mesaj=GUIGetMsg() ; ascultare mesaje

Switch $mesaj ; testarea valorilor posibile pentru mesaj

Case **$IDButon ; mesajul butonului**

MsgBox(0,"","S-a apasat butonul!")

Case **$GUI\_EVENT\_CLOSE ; mesajul evenimentului de închidere**

MsgBox(0,"","Se inchide fereastra!")

**ExitLoop ; oprirea ciclului de ascultare**

EndSwitch

Wend

Evident, ciclul infinit cu terminare forţată nu e obligatoriu, acesta fiind echivalent cu un ciclu cu condiţie fără terminare forţată, precum:

**$mesaj=""**

While **$mesaj<>$GUI\_EVENT\_CLOSE**

$mesaj=GUIGetMsg()

Switch $mesaj

Case $IDButon

MsgBox(0,"","S-a apasat butonul!")

Case $GUI\_EVENT\_CLOSE

MsgBox(0,"","Se inchide fereastra!")

EndSwitch

Wend

Se observă iniţializarea mesajului şi înlocuirea valorii 1 (ciclu infinit), cu condiţia de continuare a ciclului (atâta timp cât mesajul generat de GUI nu e mesajul de închidere). Faptul că nu s-a apelat la ciclu infinit, face să nu mai fie necesară nici terminarea forţată cu ExitLoop.

Evenimentele-obiect pot fi tratate ca în aceste exemple chiar şi atunci când GUI este formată din mai multe ferestre, deoarece valorile Control ID sunt unice indiferent câte ferestre s-au construit. În schimb, pot apare ambiguităţi la evenimentele-sistem, când nu e clar la care fereastră se referă un mesaj precum $GUI\_EVENT\_CLOSE.

Atunci când GUI conţine mai multe ferestre, ascultarea mesajelor se face apelând funcţia cu argumentul 1:

$mesaj=GUIGetMsg(1)

Valoarea returnată la acest apel nu va mai fi un mesaj, ci un vector elementele:

* Mesajul propriu-zis;
* **Identificatorul GUI de fereastră**, în cazul ferestrelor;
* Informaţii suplimentare, după caz, neglijate deocamdată de exemple (vezi detalierea funcţiei GUIGetMsg).

#include <GUIConstants.au3>

**$IDXf1**=GUICreate("Fereastra primara", 400,400) ; fereastra 1 devine implicita

GUICtrlCreateLabel("Eticheta mea",0,0)

$IDButon1=GUICtrlCreateButton("Butonul 1",50,50)

GUISetState(@SW\_SHOW)

**$IDXf2**=GUICreate("Fereastra a doua",200,200) ; fereastra 2 devine implicita

$IDButon2=GUICtrlCreateButton("Butonul 2",50,50)

GUISetState(@SW\_SHOW)

While 1

$m=GUIGetMsg(1)

Select

Case $m[0]=$IDButon1

MsgBox(0,"","S-a apasat butonul 1!")

Case $m[0]=$IDButon2

MsgBox(0,"","S-a apasat butonul 2!")

Case **$m[0]=$GUI\_EVENT\_CLOSE And $m[1]=$IDXf1**

MsgBox(0,"","Se inchide fereastra primara identificata prin" \_

&@CRLF&**$IDXf1**)

**GUIDelete($IDXf1)**

Case **$m[0]=$GUI\_EVENT\_CLOSE And $m[1]=$IDXf2**

MsgBox(0,"","Se inchide fereastra secundara identificata prin" \_

&@CRLF&**$IDXf2**)

ToolTip("Aplicatia GUI se inchide peste 5 secunde!")

Sleep(5000)

**ExitLoop**

EndSelect

Wend

Se observă că s-a folosit o structură Select CASE în loc de Switch CASE, deoarece de data asta nu toate ramurile CASE sunt controlate de aceeaşi variabilă (ultimele două ramuri sunt controlate de ambele componente ale mesajului).

Atunci când GUI e formată din multiple ferestre, se folosesc o serie de mecanisme pentru a indica la care fereastră se face referire:

* Fiecare fereastră, la creare, primeşte un identificator GUI de fereastră (returnat de funcţia de creare a ferestrelor GUICreate); spre deosebire de identificatorii Control ID, care sunt numere zecimale pozitive, identificatorii de fereastră sunt coduri hexazecimale;
* Operaţiile GUI pot folosi identificatorii pentru a indica fereastra asupra căreia are loc operaţia; dacă nu se precizează un identificator, atunci operaţia are loc asupra **ferestrei implicite**;
* **Fereastra implicită** este ultima fereastră creată; pentru a conferi acest statut altei ferestre, se foloseşte funcţia de comutare între ferestre GUISwitch (e vorba de fereastra implicită din punct de vedere al algoritmului şi nu de fereastra activă de pe ecran).

În ultimul exemplu, între liniile 1-4 fereastra implicită e cea cu identificatorul IDXf1. Aceasta înseamnă că operaţiile de creare a etichetei, a primului buton şi afişarea ferestrei (liniile 2-4) se referă la fereastra 1.

În restul macroului, fereastra implicită e cea de-a doua, deci imediat după crearea sa. Aşadar, al doilea buton şi a doua operaţie GUISetState se referă la aceasta.

Faptul că în cadrul structurii While a doua fereastră e implicită, obligă ca atunci când se face referire la prima fereastră, să se folosească identificatorul său, spre exemplu la închiderea ferestrei:

GUIDelete($IDXf1)

Dacă prima fereastră ar fi fost implicită, argumentul funcţiei nu mai era necesar. Spre exemplu, modificăm ramura CASE referitoare la închiderea primei ferestre astfel:

Case $m[0]=$GUI\_EVENT\_CLOSE And $m[1]=$IDXf1

MsgBox(0,"","Se inchide fereastra primara identificata prin" \_

&@CRLF&$IDXf1)

**GUISwitch($IDXf1)**

**GUIDelete()**

În urma acestei modificări, faptul că GUISwitch a realizat comutarea ferestrei implicite ne scuteşte de a o mai referi prin identificator în operaţiile ce urmează (GUIDelete nu mai are argument).

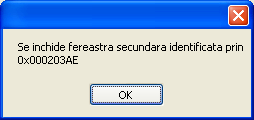
Sugerăm prin aceasta comportamentul general al funcţiilor GUI – *atunci când identificatorul GUI lipseşte dintre argumente, operaţia se referă la fereastra implicită* (definită de ultimul apel al funcţiilor GUICreate sau GUISwitch).

Un alt element care reiese de aici este faptul că s-au folosit metode diferite de închidere pentru cele două ferestre:

* Prima fereastră e închisă cu GUIDelete pentru a nu afecta a doua fereastră;
* A doua fereastră e închisă de ExitLoop pentru a opri definitiv aplicaţia (fapt asupra căruia se avertizează cu o etichetă ToolTip).

Aceasta sugerează că orice GUI trebuie să aibă un buton de închidere definitivă pe una din ferestre. Uneori este o opţiune meniu, un buton GUI special creat în acest scop sau, ca în exemplul de faţă, butonul de închidere al uneia din ferestre.

În sfârşit exemplul sugerează, prin casetele de dialog generate de închiderea ferestrelor, cum arată un identificator GUI de fereastră. Se remarcă faptul că sunt coduri hexazecimale pe 4 octeţi. Valoarea codului poate să difere de la o execuţie la alta a macroului, pentru o aceeaşi fereastră, dar se păstrează pe parcursul unei execuţii.



**Programarea evenimentelor prin gestiune de evenimente[[27]](#footnote-27)**

**Gestionarea de evenimente** presupune definirea în prealabil a unor **funcţii handler** pentru fiecare eveniment care se doreşte programat. La apariţia unui eveniment, macroul îşi întrerupe execuţia termporar până după executarea funcţiei handler. Prin această metodă, apelurile funcţiilor handler nu sunt explicite – ele sunt declanşate de evenimente. *Această metodă e potrivită atunci când GUI trebuie să îndeplinească şi alte sarcini decât să aştepte acţiunile utilizatorului*.

Trecerea în modul OnEvent se realizează apelând funcţia de configurare Opt("GUIOnEventMode",1). În acest mod, pot avea loc aceleaşi două tipuri de evenimente ca în cazul anterior:

* Evenimente-obiect;
* Evenimente-sistem.

Lipsa evenimentelor e considerată stare implicită a GUI şi nu există o valoare specială asociată. Pentru aceste tipuri de evenimente se pot defini două tipuri de funcţii handler, legate la evenimente cu ajutorul a două funcţii:

* GUICtrlSetOnEvent – leagă un handler la un obiect;
* GUISetOnEvent – leagă un handler la un eveniment-sistem (care afectează o fereastră întreagă).

Evenimentele pentru care nu se definesc funcţii handler sunt ignorate. În interiorul funcţiilor handler se pot utiliza câteva macrovariabile specifice:

* @GUI\_CTRLID reprezintă
  + valoarea Control ID a obiectului care a suferit un eveniment-obiect;
  + valoarea Event ID (**identificator de eveniment**) dacă e vorba de un eveniment-sistem;
* @GUI\_WINHANDLE – identificatorul GUI al ferestrei care a suferit evenimentul;
* @GUI\_CTRLHANDLE – identificatorul GUI al obiectului GUI care a suferit evenimentul.

Pentru exemplificare, reluăm primul exemplu, cu o GUI ce conţine un buton. Mai întâi se definesc funcţiile handler, apoi macroul de creare GUI şi legarea funcţiilor de evenimente:

#include <GUIConstants.au3>

**Func GestiuneButon()** ;handler pt buton

MsgBox(0,"","S-a apasat butonul!")

MsgBox(0,"", "Valori actuale ale macrovariabilelor:" & @CRLF & \_

"obiect apasat: " & @GUI\_CTRLID & \_

"fereastra curenta: " & @GUI\_WINHANDLE)

Endfunc

**Func GestiuneInchidere()** ;handler pt inchiderea ferestrei

MsgBox(0,"","Se inchide fereastra!")

MsgBox(0,"", "Valori actuale ale macrovariabilelor:" & @CRLF & \_

"eveniment in curs:" & @GUI\_CTRLID & \_

"fereastra curenta:" & @GUI\_WINHANDLE)

**Exit ; terminarea execuţiei**

Endfunc

**Opt("GUIOnEventMode",1) ; activare Event mode**

$IDXfer=GUICreate("Fereastra mea",200,200)

$IDbuton=GUICtrlCreateButton("Apasa-ma",20,20)

**GUISetOnEvent($GUI\_EVENT\_CLOSE,"GestiuneInchidere") ; legarea**

**GUICtrlSetOnevent($IDbuton,"GestiuneButon")**

GUISetState(@SW\_SHOW)

While 1 ; ciclu infinit

Sleep(1000)

Wend

În final, la macro se adaugă un ciclu infinit care să prevină terminarea execuţiei (aceasta fiind realizată forţat, prin Exit din cadrul funcţiei asociate închiderii ferestrei). În absenţa ciclului infinit, orice GUI se închide la terminarea macroului, în acest caz, chiar după crearea sa, fără a lăsa timp pentru utilizarea GUI. Ciclul infinit conţine instrucţiunea de aşteptare de 1 secundă. În modelul MessageLoop, o astfel de instrucţiune va bloca funcţionarea GUI. În modelul OnEvent, GUI este blocată doar până la declanşarea unui eveniment (care întrerupe macroul, deci întrerupe aşteptarea Sleep, până când funcţia handler ataşată e executată; apoi, ciclul de aşteptare infinită e reluat până la următorul eveniment).

Se observă că structura CASE de legare a mesajelor la codul sursă care le tratează (din modelul MessageLoop) e înlocuită aici cu o succesiune de legări GUISetOnEvent (între un eveniment-sistem şi funcţia sa) şi GUICtrlSetOnEvent (între un obiect GUI şi funcţia sa).

Din mesajele afişate de cele două funcţii se pot observa valorile macrovariabilelor @GUI\_CTRLID (identificatorul obiectului curent sau al evenimentului curent) şi @GUI\_WINHANDLE (identificatorul GUI al ferestrei curente).

În lucrul cu mai multe ferestre, apare din nou problema identificării acestora, tratată similar cu modelul MessageLoop – elementele cheie sunt cunoaşterea ferestrei implicite, comutarea sa cu GUISwitch (la nevoie) sau explicitarea ferestrei vizate în operaţii, prin identificatorul său, creat de GUICreate.

#include <GUIConstants.au3>

Func ApasareButon()

Switch **@GUI\_CtrlId**

Case $IDButon1

MsgBox(0, "", "S-a apasat buton 1")

Case $IDButon2

MsgBox(0,"", "S-a apasat buton 2")

EndSwitch

EndFunc

Func Inchidere()

Switch **@GUI\_WinHandle**

Case $IDXF1

MsgBox(0,"","Se inchide fereastra primara")

GUIDelete($IDXF1)

Case $IDXF2

MsgBox(0,"","Se inchide fereastra secundara SI aplicatia!")

Exit

EndSwitch

Endfunc

Opt("GUIOnEventMode", 1)

$IDXF1 = GUICreate("Fereastra primara", 300, 300)

$IDButon1 = GUICtrlCreateButton("Buton 1", 10, 10)

GUISetState(@SW\_SHOW)

**GUICtrlSetOnEvent($IDButon1, "ApasareButon") ; legari ale primei ferestre**

**GUISetOnEvent($GUI\_EVENT\_CLOSE, "Inchidere")**

$IDXF2 = GUICreate("Fereastra secundara", 200,200)

$IDButon2 = GUICtrlCreateButton("Buton 2", 10, 10)

GUISetState(@SW\_SHOW)

**GUICtrlSetOnEvent($IDButon2, "ApasareButon") ; legari ale ferestrei secundare**

**GUISetOnEvent($GUI\_EVENT\_CLOSE, "Inchidere")**

While 1

Sleep(1000)

WEnd

Din acest exemplu se remarcă un aspect esenţial – o aceeaşi funcţie handler poate gestiona mai multe obiecte în acelaşi timp:

* Functia Inchidere tratează închiderea ambelor ferestre (e legată la evenimentul de închidere a ambelor ferestre);
* Functia ApasaButon tratează ambele butoane (e legată la butoanele ambelor ferestre).

In ambele cazuri, diferenţa de tratare se face prin structuri CASE care verifică dacă evenimentul a fost declanşat în prima sau a doua fereastră. Ca şi în exemplul de la modelul MessageLoop, s-au definit două metode de închidere:

* GUIDelete pentru fereastra primară (a doua va rămâne pe ecran);
* Exit pentru fereastra secundară (închide definitiv aplicaţia, oprind ciclul infinit de aşteptare a evenimentelor).

Desigur e posibil să se creeze câte un handler pentru fiecare obiect şi pentru fiecare eveniment.

S-a amintit în capitolul dedicat structurilor repetitive şi terminărilor forţate ale acestora, că mai multe instrucţiuni Exit pot să apară în acelaşi macro, însoţite de diverse coduri de terminare care să sugereze motivul şi locul în care a avut loc oprirea execuţiei. Am amintit anterior că orice GUI trebuie să conţină un buton de oprire. Cu instrucţiuni Exit multiple, e posibil să se definească mai multe moduri de oprire. Modificăm exemplul anterior astfel încât ambele ferestre să provoace, la închidere, terminarea definitivă a macroului.

#include <GUIConstants.au3>

Func ApasareButon()

Switch @GUI\_CtrlId

Case $IDButon1

MsgBox(0, "", "S-a apasat buton 1")

Case $IDButon2

MsgBox(0,"", "S-a apasat buton 2")

EndSwitch

EndFunc

Func Inchidere()

Switch @GUI\_WinHandle

Case $IDXF1

**Exit 100 ; terminare cu codul 100**

Case $IDXF2

**Exit 200 ; terminare cu codul 200**

EndSwitch

Endfunc

Opt("GUIOnEventMode", 1)

$IDXF1 = GUICreate("Fereastra primara", 300, 300)

$IDButon1 = GUICtrlCreateButton("Buton 1", 10, 10)

GUISetState(@SW\_SHOW)

GUICtrlSetOnEvent($IDButon1, "ApasareButon")

GUISetOnEvent($GUI\_EVENT\_CLOSE, "Inchidere")

$IDXF2 = GUICreate("Fereastra secundara", 200,200)

$IDButon2 = GUICtrlCreateButton("Buton 2", 10, 10)

GUISetState(@SW\_SHOW)

GUICtrlSetOnEvent($IDButon2, "ApasareButon")

GUISetOnEvent($GUI\_EVENT\_CLOSE, "Inchidere")

While 1

Sleep(1000) ; Idle around

WEnd

**Func OnAutoItExit()**

**Switch @exitCode**

**Case 100**

**MsgBox(0,"","Terminare provocata din fereastra primara")**

**Case 200**

**MsgBox(0,"","Terminare provocata din fereastra secundara")**

**EndSwitch**

**EndFunc**

S-au evidenţiat în codul sursă cele două instrucţiuni Exit, cu codurile 100 şi 200. Codurile sunt alese de programator, pentru a diferenţia între cele două puncte de terminare. Doar unul dintre cele două puncte va termina programul la o execuţie. Punctul de terminare va fi identificat cu ajutorul macrovariabilei @EXITCODE, din funcţia OnAutoItExit(). Se observă în cadrul funcţiei că o ramură CASE testează care din cele două coduri de terminare a fost activat şi pentru fiecare dintre ele generează câte un mesaj. Condiţii:

* Blocul funcţiei OnAutoItExit() poate conţine orice cod sursă care să se execute imediat înainte de oprirea macroului, în acest caz generarea unui mesaj spre utilizator în funcţie de punctul de terminare;
* Numele funcţiei OnAutoItExit() este prestabilit, pentru a se evita un apel explicit al funcţiei; acest nume poate fi modificat printr-o configurare precum **Opt("OnExitFunc","functiaMea")**;
* Apelul funcţiei OnAutoItExit() are loc, practic, la executarea oricăreia din instrucţiunile Exit sau la terminarea naturală a macroului (fără forţare); faptul că e un apel implicit face ca această funcţie să poată fi considerată *un handler al evenimentului "Terminarea macroului"*.

Alături de @EXITCODE (cu valoarea preluată de la ultimul Exit care s-a executat), funcţia de gestiune a terminării poate referi şi macrovariabila @EXITMETHOD, cu valorile posibile:

* 0 – indică terminare naturală a macroului (fără Exit);
* 1 – terminare forţată cu Exit;
* 2 – terminare forţată cu ajutorul pictogramei AutoIt de pe bara Tray;
* 3 – terminare prin deconectarea utilizatorului de la sistemul de operare;
* 4 – terminarea prin oprirea sistemului de operare.

Aşadar, handlerul OnAutoItExit() nu gestionează doar efectele instrucţiunii Exit ci şi orice metodă de terminare a execuţiei. Aceasta face ca funcţia să fie deosebit de importantă în testarea software, deoarece poate salva automat date ale macroului la terminarea execuţiei, spre exemplu rezultatele aşteptate de la o aplicaţie sau fereastră testată.

Concluzionăm cu o comparaţie succintă între cele două metode:

* Ascultare de mesaje:
  + Fiecare eveniment generează un mesaj;
  + Într-un ciclu While infinit, se preia mesajul (valoarea funcţiei GUIGetMsg);
  + Se confruntă valoarea preluată cu toate mesajele urmărite (în funcţie de ce obiecte GUI există şi ce evenimente se aşteaptă) într-o structură CASE;
  + Fiecare ramură CASE corespunde unui mesaj aşteptat şi programează reacţia la acesta;
  + Mesajele pentru care nu s-au definit ramuri CASE reprezintă evenimente ignorate;
  + Mesajul 0 corespunde lipsei unui eveniment şi poate fi la rândul său programat într-una din ramurile CASE!
  + Execuţia macroului nu e întreruptă decât la oprirea definitivă;
  + Închiderea GUI (şi oprirea ciclului infinit) e definită de comanda ExitLoop asociată unuia din mesaje (de obicei mesajului $GUI\_EVENT\_CLOSE sau valorii Control ID a unui buton de închidere creat în acest scop).
* Gestionare de evenimente:
  + Fiecare eveniment provoacă un apel implicit la o funcţie handler;
  + Se definesc în prealabil toate funcţiile handler urmărite (în funcţie de obiectele GUI şi evenimentele aşteptate);
  + Evenimentele pentru care nu s-au definit funcţii handler sunt ignorate;
  + Execuţia macroului se finalizează cu un ciclu infinit de aşteptare; această aşteptare se întrerupe la fiecare eveniment, pentru executarea funcţiei handler asociate;
  + Închiderea GUI e definită de o funcţie handler asociată unui buton de închidere.

În ambele modele se pot defini multiple metode de închidere GUI (opţiune meniu, buton special, buton de închidere, butoane multiple de pe diferite ferestre etc.). Fiecărei metode de închidere i se va asocia o instrucţiune Exit cu un cod de terminare. Apoi, se defineşte o funcţie handler specială pentru evenimentul Exit, ce poate defini reacţii diferite la codurile de terminare.

**Funcţii GUI**

În cele ce urmează, avertizăm că majoritatea funcţiilor GUI din AutoIt se folosesc în conexiune cu o serie de concepte prezentate deja în acest material:

* Variabile ce memorează identificatorii GUI şi Control ID (subcapitolul privind concepte GUI);
* Funcţia de conversie (HWnd) şi de testare (ISHWnd) a identificatorilor GUI (subcapitolul privind funcţiile orientate pe tipizarea datelor);
* Funcţiile de gestiune a ferestrelor;
* Instrumentul AWIT (AutoIt Window Information Tool), componentă a pachetului AutoIt, pentru obţinerea de date privind atributele GUI.

La aceste concepte se adaugă unele legate de programarea GUI tradiţională pe platforme Microsoft, în special cele legate de clasificarea şi componentele ferestrelor.

Sistemul de operare Windows foloseşte o GUI în care unitatea fundamentală este fereastra. Componentele unei ferestre sunt:

* Bară de titlu – conţine implicit titlu şi un chenar de aceeaşi culoare în jurul ferestrei (caption);
* Chenare nedimensionabile:
  + Extern subţire (border);
  + Intern 3D (clientedge);
  + Intern (staticedge);
  + Chenar de tip casetă de dialog (dlgframe);
* Meniu de control (sysmenu) accesibil pe căile:
  + Pe bara de titlu, dacă e disponibilă;
  + Pe bara de sarcini, prin clic dreapta pe butonul de minimizare.
* Butoane de control (minimizare, maximizare, restaurare, Help, închidere);
* Chenar de dimensionare (sizebox);
* Componente interne: bare derulante (hscroll, vscroll).
* Suprafaţă internă (popup), numită şi suprafaţa client.

Diferite tipuri de ferestre se pot obţine combinând aceste componente. Ex: o fereastră de tip Popup nu are bară de titlu dar poate avea meniu de control accesibil pe butonul de minimizare şi poate fi dimensionabilă.

Componentele implicite ale unei ferestre la crearea sa (inclusiv în AutoIt) sunt:

* Suprafaţa;
* Butonul de minimizare (fără maximizare);
* Butonul de închidere;
* Bara de titlu cu titlu şi chenar în prelungire (nedimensionabil).

Aceste componente au ca rezultat o fereastră de tipul celor din Control Panel – nedimensionabile, nemaximizabile, folosite atunci când nu se pune la dispoziţia utilizatorului o suprafaţa internă extensibilă.

Stările posibile ale unei ferestre sunt[[28]](#footnote-28):

* Maximizată (acoperă tot ecranul);
* Minimizată – fereastra e înliocuită cu un buton:
  + pe bara de sarcini Windows dacă e o fereastră părinte;
  + deasupra barei de sarcini dacă e fereastră copil (ex: ferestrele documentelor dintr-o aplicaţie multidocument ca Access)
* Restaurată (apare la poziţia şi dimensiunile precedente ultimei maximizări sau minimizări);
* Afişată (vizibilă);
* Ascunsă (invizibilă);
* Inaccesibilă (afişată dar nu reacţionează la evenimente);
* Accesibilă (afişată şi reacţionează la evenimente);
* Inactivă (conţinutul său este vizibil dar ignoră evenimentele de la tastatură);
* Activă (conţinutul său este vizibil şi tastarea are loc implicit în aceasta);
* Închisă – invizibilă, inaccesibilă, inactivă.

O componentă a sistemului de operare numită managerul ferestrelor gestionează accesul la ferestre, relaţiile dintre ele şi modul lor de afişare. Pentru aceasta, la creare, fiecare fereastră primeşte un identificator GUI. Ferestrele Windows sunt create pe o ierarhie:

* **Fereastra rădăcină** e suprafaţa Desktop, aflată permanent în stare Full Screen şi lipsită de manevrabilitate (nu poate fi minimizată, mutată, dimensionată);
* Ferestrei Desktop i se subordonează **ferestrele de nivel înalt**, specifice aplicaţiilor Windows;
* Ferestrele de nivel înalt pot conţine **ferestre copil**, care sunt afişate în suprafaţa internă a părintelui, se minimizează odată cu părintele dar nu se minimizează pe bara de sarcini Windows;
* Ferestrele copil pot conţine mai departe ferestre copil, pe următoarele nivele ale ierarhiei; această relaţie este reflectată de aplicaţii multidocument ca MS Access;
* Alături de relaţiile părinte-copil şi relaţiile de fraţi (între ferestrele copil ale aceleiaşi ferestre părinte) mai există şi **relaţia deţinător-deţinut**: o fereastră poate deţine o altă fereastră dacă aceasta din urmă se comportă ca un copil, cu excepţia faptului că nu e afişată în interiorul ferestrei deţinător. De obicei casetele de dialog sunt ferestre deţinute ale aplicaţiilor. În AutoIt, ferestrele copil nu pot conţine bară de meniuri, dar ferestrele deţinute pot.
* Modul de suprapunere a ferestrelor e definit de regulile:
  + Suprafaţa internă a părintelui e acoperită de suprafaţa copiilor, dar elementele externe ale părintelui (bară de titlu, chenare) nu sunt niciodată acoperite de copii;
  + Ferestrele deţinute acoperă ferestrele care le deţin;
  + Ordinea de suprapunere a fraţilor e dată de indexul Z ("înălţimea" la care se află fereastra), care e stabilit prin ordinea de creare a fraţilor. Acţiunea de activare a unei ferestre are ca efect modificarea indexului Z prin aducerea sa în prim plan. Ferestrele cu atributul Always On Top se află mereu în prim plan, deci indexul Z al acestora nu va fi modificat de activarea altor ferestre.

Crearea fiecărei ferestre are ca efect generarea unui identificator GUI de fereastră şi încadrarea ferestrei în ierarhie, pe baza unui set de patru informaţii:

* Identificatorul GUI al primului copil;
* Identificatorul GUI al următorului frate;
* Identificatorul GUI al părintelui;
* Identificatorul GUI al ferestrei proprietar.

O serie de atribute de stil asigură **stilurile de ferestre**, împărţite în stiluri de bază (prefixate cu WS) şi stiluri extinse (prefixate cu WS\_EX). Stilul unei ferestre defineşte componentele unei ferestre, aspectul său, relaţia sa cu alte ferestre şi comportamentul său. Exemple de stiluri:

* WS\_OVERLAPPEDWINDOW defineşte o fereastră cu manevrabilitate completă, de obicei pentru ferestrele aplicaţiilor cu suprafaţă extensibilă;
* WS\_POPUPWINDOW defineşte o fereastră cu manevrabilitate scăzută, fără bară de titlu, de obicei pentru casete de dialog care trebuie să atragă atenţia şi să nu poată fi neglijate prin minimizare;
* WS\_CHILD defineşte o fereastră copil;
* stilul extins WS\_EX\_TOPMOST creează atributul Always On Top.

Ulterior, când fereastra se populează cu obiecte GUI, cea mai mare parte a stilurilor de ferestre pot fi folosite şi pentru obiecte. În plus, unele tipuri de obiecte au stiluri specifice[[29]](#footnote-29).

Orice interfaţă Windows este alcătuită dintr-o combinaţie de ferestre de diferite tipuri, cu diferite comportamente şi populate cu diverse obiecte GUI. Spre exemplu, aplicaţia Notepad nu este altceva decât o fereastră cu două tipuri de obiecte:

* O bară cu meniuri;
* Un obiect Edit care acoperă suprafaţa ferestrei!

Aplicaţia Access este o fereastră părinte care conţine mai multe tipuri de ferestre copil:

* Fereastra bazei de date, la rândul său cu obiecte GUI;
* Bara de meniu flotantă;
* Barele de butoane flotante.

Barele de butoane flotante, cum sunt cele din aplicaţiile MS Office, sunt la rândul lor ferestre! Caracteristicile lor sunt modificate: nu au bară de titlu, nu au chenar, sunt dimensionabile şi populate cu butoane sau liste derulante. Dacă o bară flotantă e trasă spre mijlocul suprafeţei Word, aceasta capătă forma unei casete de butoane şi apare chiar o bară de titlu şi un buton de închidere!

Operaţiile specifice ferestrelor sunt, conform dependenţelor de componentele ferestrei:

* Activarea (clic în fereastră sau prin navigarea listei de ferestre cu Alt-Tab[[30]](#footnote-30));
* Dezactivarea (activarea altei ferestre – una singură poate fi activă la un moment dat);
* Dimensionarea (dacă există chenar);
* Maximizarea, minimizarea, restaurarea, închiderea (dacă există bară de titlu fie cu butoane de control, fie cu meniu de control);
* Mutarea (dacă există bară de titlu);

**Crearea ferestrelor**

Funcţia GUICreate este prima funcţie apelată în programarea GUI şi are ca efect definirea unei ferestre (nu şi afişarea sa!). Argumentul său obligatoriu este titlul ferestrei. Valoarea returnată este identificatorul GUI al ferestrei în caz de succes. În caz de eşec se returnează 0 şi se comută @error la 1:

$IDXFer=GUICreate("Fereastra mea")

Rularea acestui exemplu nu va avea un efect vizibil. Implicit, ferestrele se creează invizibile, necesită o comandă de trecerea în starea vizibilă: GUISetState(@SW\_SHOW). Apoi, pentru ca fereastra să nu dispară imediat după afişarea sa (datorită terminării macroului, se recomandă ca la exemplele ce urmează să se adauge ciclul infinit de aşteptare cu programarea butonului de închidere (butoanele de închidere a ferestrelor nu funcţionează implicit!):

While 1

$msg=GUIGetMsg()

IF $msg=$GUI\_EVENT\_CLOSE Then Exit

Wend

În plus, e necesară includerea bibliotecii de constante pentru programarea GUI. Aşadar, crearea unei ferestre poate fi verificată prin macroul:

#include <GUIConstantsEx.au3>

$IDXFer=GUICreate("Fereastra mea")

GUISetState(@SW\_SHOW)

While 1

$msg=GUIGetMsg()

IF $msg=$GUI\_EVENT\_CLOSE Then Exit

Wend

Recomandăm folosirea acestei structuri în verificarea exemplelor, cu excepţia cazurilor în care se specifică acest lucru. *Pentru lizibilitate, includerea bibliotecii de constante şi ciclul de aşteptare nu vor mai apare în exemplele din acest capitol.* Acest lucru va fi sugerat de prezenţa unui şir de puncte acolo unde lipsesc porţiunile respective de cod.

Funcţia de creare a unei ferestre are următoarele argumente opţionale, în ordine:

* Lăţimea şi înălţimea (numerice, trebuie să apară ambele sau nici una);
* Poziţia: distanţa faţă de marginile stânga şi sus ale ecranului (implicit centrate, trebuie să apară ambele sau nici una);
* Stilul ferestrei (implicit nedimensionabilă, nemaximizabilă) – poate acumula opţiunile din tabelul de variante;
* Stilul suplimentar al ferestrei – poate acumula opţiuni din tabelul de variante;
* Identificatorul GUI al ferestrei deţinător sau părinte, dacă se doreşte crearea unei ferestre deţinute sau copil.

1. Macro recording and playback tools [↑](#footnote-ref-1)
2. Visual Basic for Applications [↑](#footnote-ref-2)
3. Unii autori, inclusiv documentaţia AutoIt, foloseşte termenii macro şi script ca sinonime. După alţi autori, macroul este un tip particular de script orientat pe operaţii GUI iar scripturile sunt programe interpretate ce automatizează diverse operaţii (nu doar GUI), fiind mai apropiate de programarea tradiţională. În materialul de faţă cei doi termeni se vor utiliza cu sensurile lor diferenţiate. [↑](#footnote-ref-3)
4. Graphic user interface – interfaţa grafică a utilizatorului [↑](#footnote-ref-4)
5. GUI automation [↑](#footnote-ref-5)
6. Deşi cursorul nu e vizibil în captură, poziţia sa este butonul Center din Word (indicată de eticheta Center) [↑](#footnote-ref-6)
7. Termenul snippet va fi utilizat în cele ce urmează ca o denumire scurtă pentru porţiuni reutiolizabile de cod sursă, o funcţionalitate inclusă de multe editoare de cod sursă moderne. [↑](#footnote-ref-7)
8. Interfeţe de programare [↑](#footnote-ref-8)
9. Similar cu modul în care instrumente WYSIWYG precum Dreamweaver permit generarea de funcţii JavaScript reutilizabile în pagini HTML (Behaviors, Actions) [↑](#footnote-ref-9)
10. Manualul oficial al limbajului LUA este accesibil la http://www.lua.org/manual/5.0/ [↑](#footnote-ref-10)
11. Aici termenul e folosit cu sensul de resursă Windows – componente înglobate la compilare în executabil dar la care Windows oferă acces pentru reutilizare şi personalizare facilă. [↑](#footnote-ref-11)
12. Poate fi lansat cu shortcutul aferent din meniul Windows sau cu executabilul AutoIt/SciTE/defs/Updatedefs.exe [↑](#footnote-ref-12)
13. La instalarea pachetului AutoIt, precum şi în SciTE Config se poate stabili dacă dublu clic pe fişierele au3 va avea ca efect deschiderea macroului în SciTE sau executarea sa. [↑](#footnote-ref-13)
14. Motiv pentru care se recomandă instalarea pachetului AutoIt înainte de versiunea completă SciTE [↑](#footnote-ref-14)
15. Lista variabilelor de configurare poate fi consultată la adresa http://scintilla.sourceforge.net/SciTEDoc.html, rubricile Defined variables in properties files şi File properties. Materialul de faţă va presupune că se lucrează pe configurările globale implicite, în caz contrar se va specifica acest lucru. [↑](#footnote-ref-15)
16. În original *literal value*, a nu se confunda cu valorile de tip caracter (string) [↑](#footnote-ref-16)
17. În exemplele ce urmează se va evita utilizarea instrucţiunii Dim [↑](#footnote-ref-17)
18. În original *control handles* sau *window handles* [↑](#footnote-ref-18)
19. Acestea sunt valorile hexazecimale. Lista codurilor de localizare geografică e disponibilă la

    http://msdn2.microsoft.com/en-us/library/ms930130.aspx [↑](#footnote-ref-19)
20. Mecanismul de conversie implicită nu e asigurat doar de operatori, ci şi de funcţiile care aşteaptă argumente de anumite tipuri. [↑](#footnote-ref-20)
21. Mai puţin în comparaţia cu operatorul == [↑](#footnote-ref-21)
22. Nu contează dacă definiţiile funcţiilor apar înainte de apelurile acestora sau după. Se recomandă să apară înainte, pentru claritatea codului sursă. [↑](#footnote-ref-22)
23. Reamintim că parametrii cu transfer prin valoare sunt consideraţi variabile locale ale funcţiei [↑](#footnote-ref-23)
24. Reamintim că diferenţa între Dim şi Local e că ultima evită conflictele de nume permiţând declararea unei variabile care are acelaşi nume cu una din afara funcţiei. [↑](#footnote-ref-24)
25. Evident, riscul de funcţie dublu definită nu apare doar la dubla includere a unui fişier, ci şi în cazuri cum ar fi existenţa a două funcţii diferite cu acelaşi nume, în fişiere incluse diferite. [↑](#footnote-ref-25)
26. Unele aplicaţii nu folosesc obiectele GUI Windows ci îşi definesc propriile GUI, chiar dacă acestea sunt vizual similare cu cele definite de Windows. Interfeţele GUI nonWindows nu pot fi accesate de AutoIt! [↑](#footnote-ref-26)
27. În original, *event handling* [↑](#footnote-ref-27)
28. Lista nu include stările specifice Windows Vista care foloseşte o interfaţă 3D ce creează câteva stări suplimentare (ex: retragerea ferestrei în umbră) [↑](#footnote-ref-28)
29. La bază, orice obiect GUI poate fi considerat o fereastră copil de dimensiuni mici, nemanevrabilă, cu aspect şi cu comportament special, preprogramat. Din acest motiv, mare parte a stilurilor de ferestre sunt aplicabile şi la obiecte. [↑](#footnote-ref-29)
30. Ferestrele copil nu apar în lista respectivă [↑](#footnote-ref-30)