High level constructs width IDA Pro. © DataRescue 2005

Data and operands available in the disassembly aren't always interpreted in the most suitable way.: IDA's interactivity allows you to change their type and representation. It even makes high level languages like constructs possible.

The C program.

To introduce these possibilities, let's analyze a small C program using particular data and constructions.

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

// our structures
// ==============

// information about our customers
struct customer_t { // a typical structure
  long id;
  char name[32];
  char sex;       // 'm'ale - 'f'emale
};


// we sell books
struct book_t {
  char title[128]; // an ASCII string
};

// and we sell computer softwares
struct software_info_t { // a structure containing various bitfields
  unsigned int plateform : 2; // 2 bits reserved for the plateform -
                              //     plateforms can be combined (0x03)
#define PC           0x1      //   0x01
#define MAC          0x2      //   0x02
  unsigned int os : 3;        // 3 bits reserved for the OS -
                              //     OS can be combined (0x1C)
#define WINDOWS      0x1      //   0x04
#define DOS          0x2      //   0x08
#define OS_X         0x4      //   0x10
  unsigned int category : 2;  // 2 bits reserved for the category -
                              //     categories can't be combined (0x60)
#define DISASSEMBLY  0x1      //   0x20
#define RECOVERY     0x2      //   0x40
#define CRYPTOGRAPHY 0x3      //   0x60
};

struct software_t {
  software_info_t info;
  char name[32];
};
```

```
// generic products we're selling
enum product_category_t { // an enumerated type
  BOOK,
  SOFTWARE,
  HARDWARE  // we actually don't sell hardware
};

union product_u { // an union to contain product information
                  // depending on its category
  book_t     book;
  software_t software;
  // struct hardware_t hardware; // we actually don't sell hardware
};

struct product_t { // a structure containing another structure
  long id;
  product_category_t category;
  product_u          p;
};


// our data
// ========

// our customers
customer_t customers[] = { // an initialized array to memorize our customers
  { 1, "Peter", 'm' },
  { 2, "John",  'm' },
  { 3, "Mary",  'f' },
  { 0 }
};

// our products
book_t ida_book = { "IDA QuickStart Guide" };

softwares_t softwares = // an initialized variable length structure
{
  3,
  {
    { { PC,     WINDOWS|DOS,  DISASSEMBLY  }, "IDA Pro" },
    { { PC|MAC, WINDOWS|OS_X, RECOVERY     }, "PhotoRescue" },
    { { PC,     WINDOWS,      CRYPTOGRAPHY }, "aCrypt" }
  }
};

#define PRODUCTS_COUNT 4
```

```
// our functions
// =============

// check software information
int check_software(software_info_t software_info)
{
  bool valid = true;
  if (software_info.plateform & PC)
  {
    if (! (software_info.plateform & MAC) && (software_info.os & OS_X))
      valid = false; // OS-X isn't yet available on PC ;)
  }
  else if (software_info.plateform & MAC)
  {
    if (! (software_info.plateform & PC) && ((software_info.os & WINDOWS) ||
(software_info.os & DOS)))
      valid = false; // Windows & DOS aren't available on Mac...
  }
  else
    valid = false;
  return valid;
}


// check product category
int check_product(product_category_t product_category)
{
  bool valid = true;
  if (product_category == HARDWARE)
  {
    valid = false;
    printf("We don't sell hardware for the moment...\n");
  }
  return valid;
}


// print customer information
void print_customer(customer_t *customer)
{
  printf("CUSTOMER %04X: %s (%c)\n", customer->id,
                                     customer->name,
                                     customer->sex);
}


// print book information
void print_book(book_t *book)
{
  printf("BOOK: %s\n", book->title);
}
```

```
// print software information
void print_software(software_t *software)
{
  printf("SOFTWARE: %s:", software->name);
  // plateform
  // we use 'if', as plateforms can be combined
  if (software->info.plateform & PC)
    printf(" PC");
  if (software->info.plateform & MAC)
    printf(" MAC");
  printf(";");
  // OS
  // we use 'if', as os can be combined
  if (software->info.os & WINDOWS)
    printf(" WINDOWS");
  if (software->info.os & DOS)
    printf(" DOS");
  if (software->info.os & OS_X)
    printf(" OS-X");
  printf(";");
  // category
  // we use 'switch', as categories can't be combined
  switch(software->info.category)
  {
    case DISASSEMBLY:
      printf(" DISASSEMBLY");
      break;
    case RECOVERY:
      printf(" RECOVERY");
      break;
    case CRYPTOGRAPHY:
      printf(" CRYPTOGRAPHY");
      break;
  }
  printf("\n");
}


// print product information
bool print_product(product_t *product)
{
  if (! check_product(product->category))
    return false;

  printf("PRODUCT %04X: ", product->id);

  switch(product->category) {
    case BOOK:
      print_book(&product->p.book);
      break;
    case SOFTWARE:
      print_software(&product->p.software);
      break;
  }
  return true;
}
```

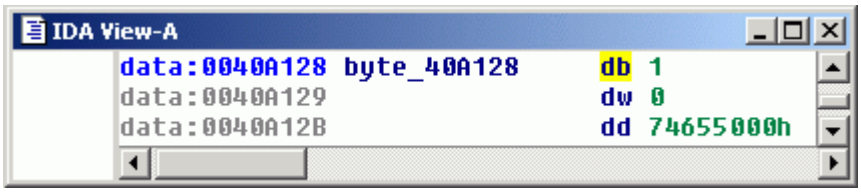
```
// our main program
// ================

void main()
{
  // print customers listing
  printf("CUSTOMERS:\n");
  customer_t *customer = customers;
  while (customer->id != 0)
  {
    print_customer(customer);
    customer++;
  }

  // allocate a small array to store our products in memory
  product_t *products = (product_t*) malloc(PRODUCTS_COUNT * sizeof(product_t));
  // insert our products
  products[0].id         = 1;
  products[0].category   = BOOK;
  products[0].p.book     = ida_book;
  products[1].id         = 2;
  products[1].category   = SOFTWARE;
  products[1].p.software = softwares.softs[0]; // we insert softwares from our
                                               // variable length structure
  products[2].id         = 3;
  products[2].category   = SOFTWARE;
  products[2].p.software = softwares.softs[1];
  products[3].id         = 4;
  products[3].category   = SOFTWARE;
  products[3].p.software = softwares.softs[2];

  // verify and print each product
  printf("\nPRODUCTS:\n");
  for (int i = 0; i < PRODUCTS_COUNT; i++)
  {
    // check validity of the product category
    if (! check_product(products[i].category))
    {
      printf("Invalid product !!!\n");
      break;
    }
    // check validity of softwares
    if (products[i].category == SOFTWARE)
    {
      if (! check_software(products[i].p.software.info))
      {
        printf("Invalid software !!!\n");
        break;
      }
    }
    // and print the product
    print_product(&products[i]);
  }

  free(products);
}
```

Running this program gives us the following result:

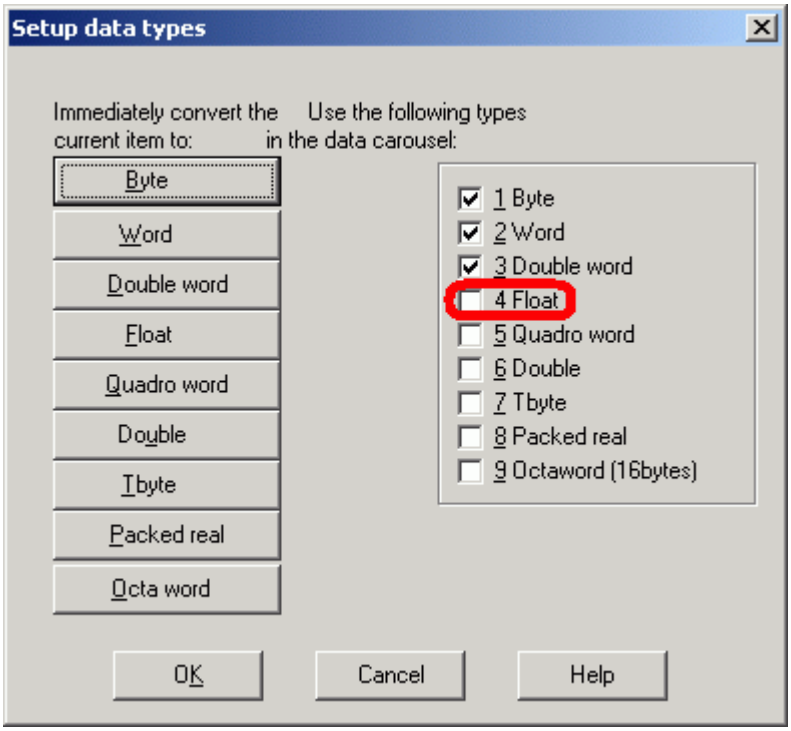
```
CUSTOMERS:
CUSTOMER 0001: Peter (m)
CUSTOMER 0002: John (m)
CUSTOMER 0003: Mary (f)
PRODUCTS:
PRODUCT 0001: BOOK: IDA QuickStart Guide
PRODUCT 0002: SOFTWARE: IDA Pro: PC; WINDOWS DOS; DISASSEMBLY
PRODUCT 0003: SOFTWARE: PhotoRescue: PC MAC; WINDOWS OS-X; RECOVERY
PRODUCT 0004: SOFTWARE: aCrypt: PC; WINDOWS; CRYPTOGRAPHY
```

Let's load the compiled binary file in a database to analyze it.

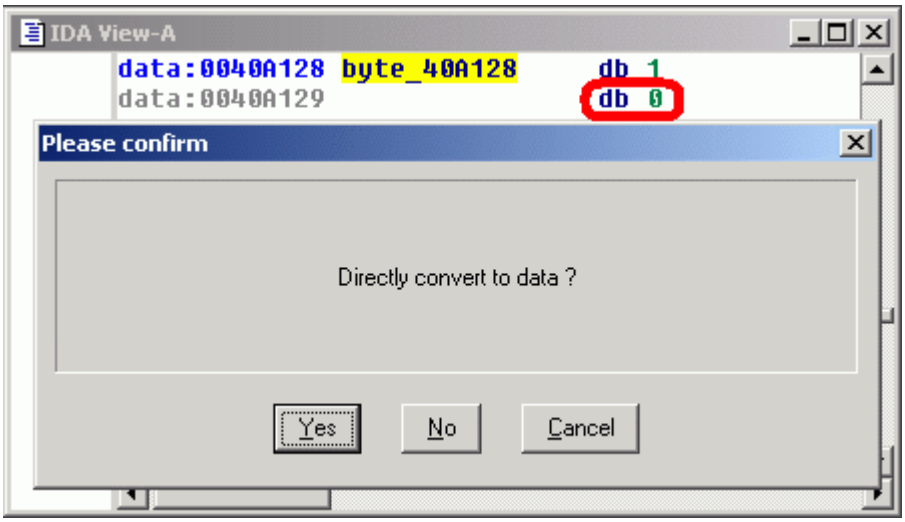
It is easy to associate a fundamental type to data: press 'D' on an undefined byte to cycle through the *db*, *dw* and *dd* data types.



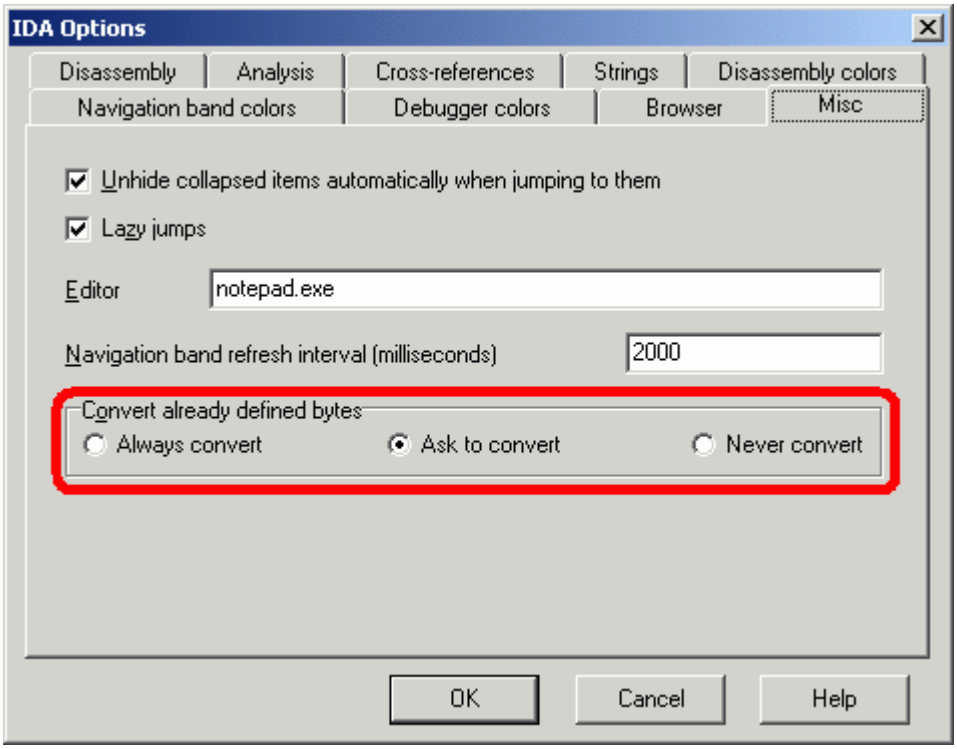
You can define how IDA cycles through data types through the *Setup data types* command in the Options menu. Just tick the data types you want IDA to cycle through. Let's add *Float* to the data carousel: pressing D on a data previously defined as *dd* will convert it to a float.



Notice that the size of the data changes according to its type. Here, we pressed 'D' on a defined byte (to convert it to a word), but since the next byte (db 0) is already defined IDA prompts us for a confirmation.



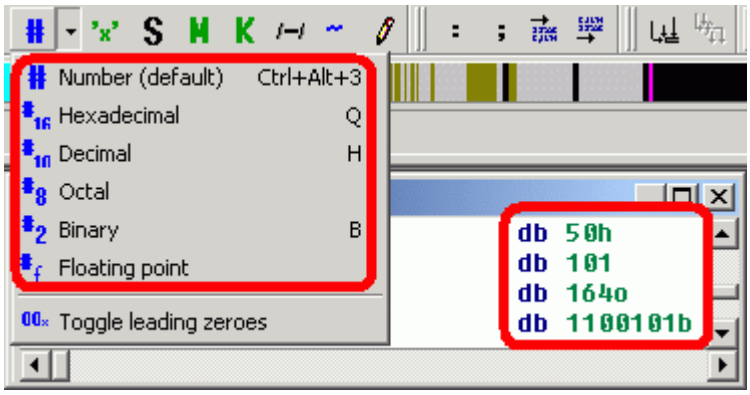
This default behavior can be modified through the *Convert already defined bytes* option in the *Options* dialog.



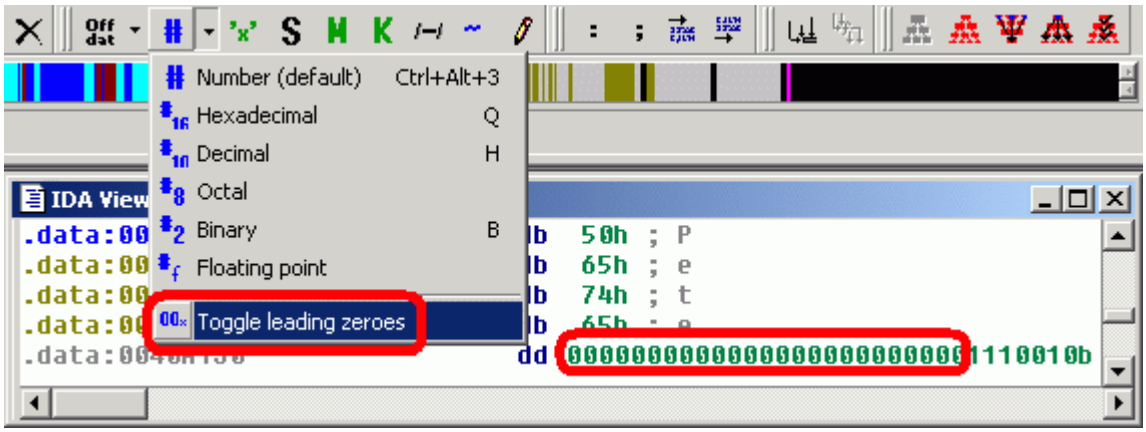
To undefine already defined data, press the 'U' key.

Once the type of the data has been specified, one might want to display it using a particular format. IDA proposes different commands to change the format. Let's have a look at the more interesting ones. Please note that all these commands can also be applied to instruction operands.

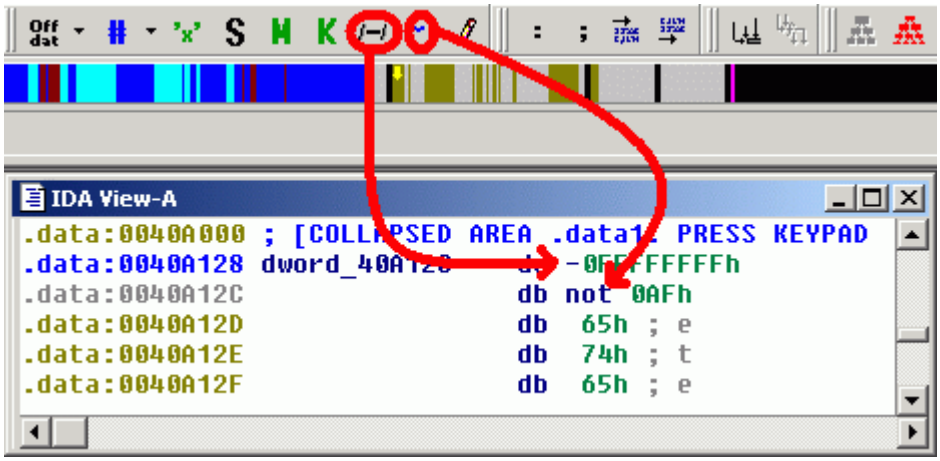
Through the *Number* commands found on the *Operands* toolbar, we switch from one numeric format to another.



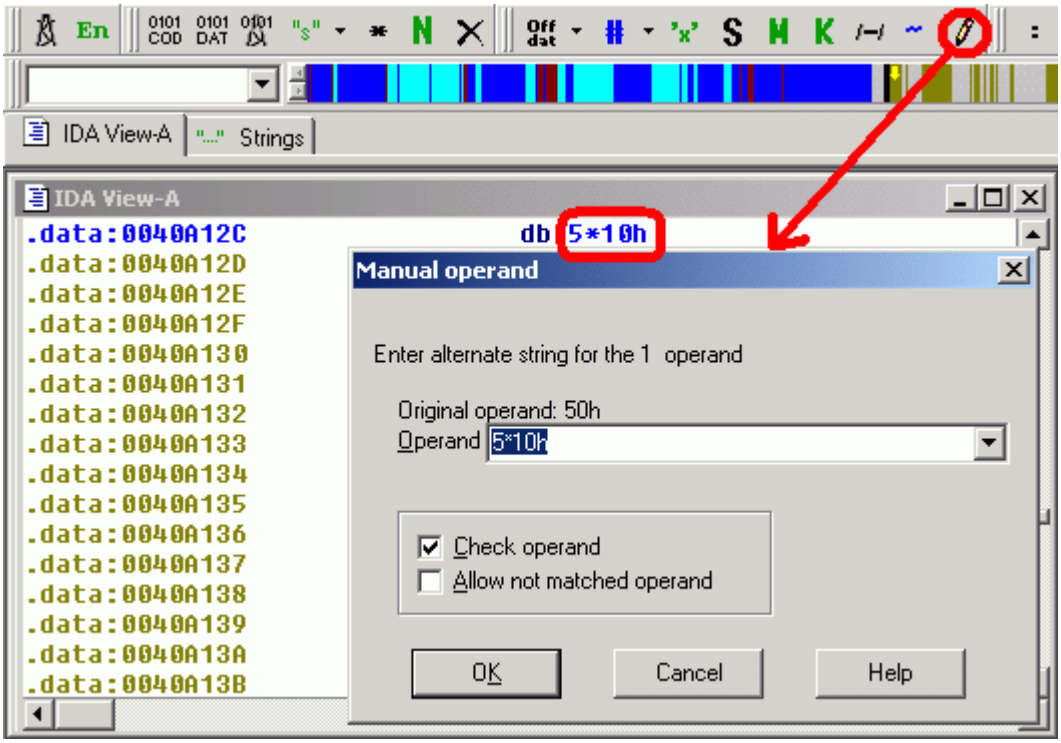
We can print leading zeros for numeric values.



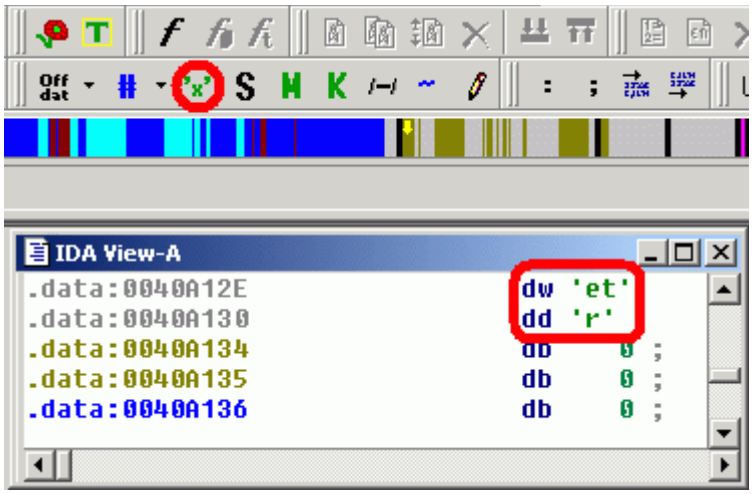
It is also possible to change the sign of an operand and to perform a bitwise negation.



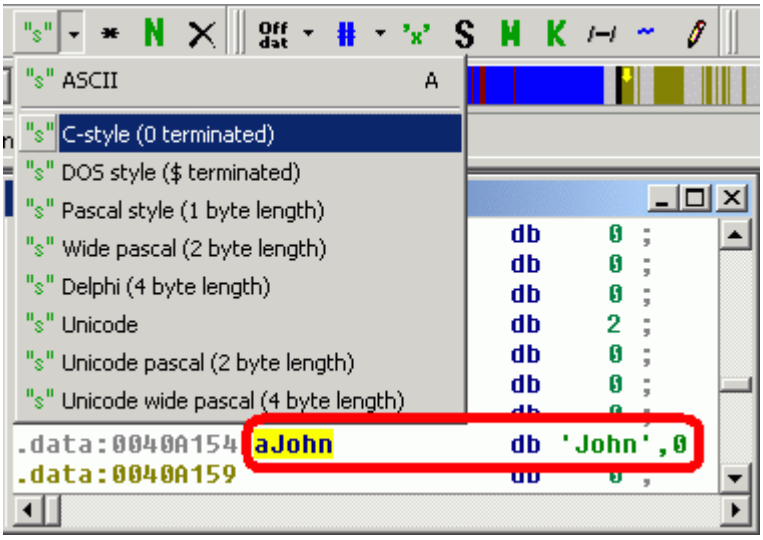
Finally, if the format you want isn't there, it can be manually defined.



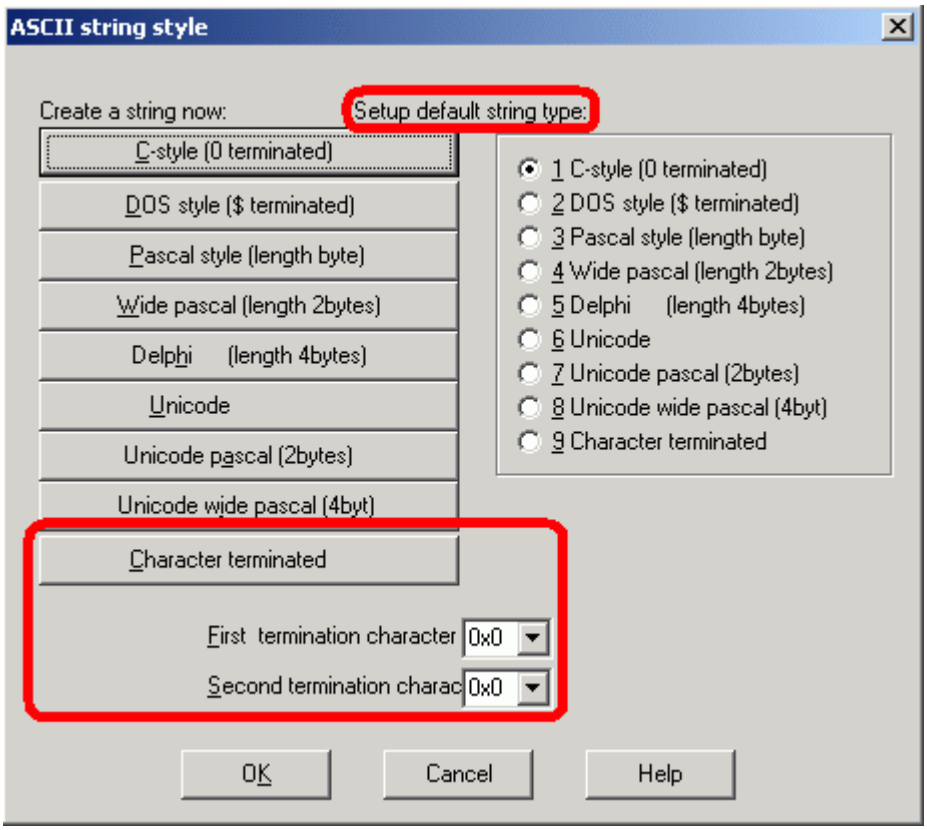
Most programs contain strings. To specify that defined data must be displayed as chars, we use the string command from the *Operands* toolbar.



There are, of course, lots of different string types. IDA supports most of them, through the *Strings* commands. Once you create a string, IDA automatically gives a name to its address. Let's apply this to some strings found in our C program.

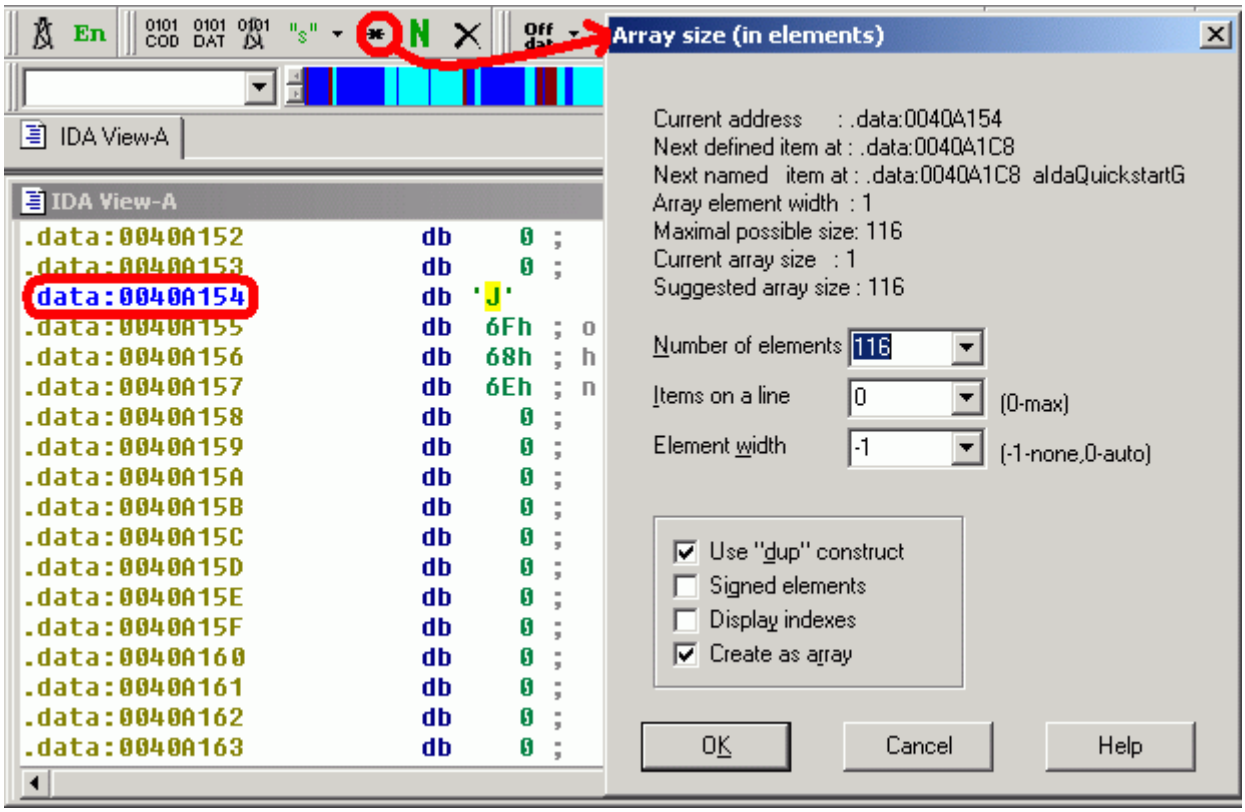


What if you aren't disassembling a C program? The *ASCII string style* item from the *Options* menu. allows you to change the default string type associated with the 'A' key, or to define special string formats with unusual termination characters.



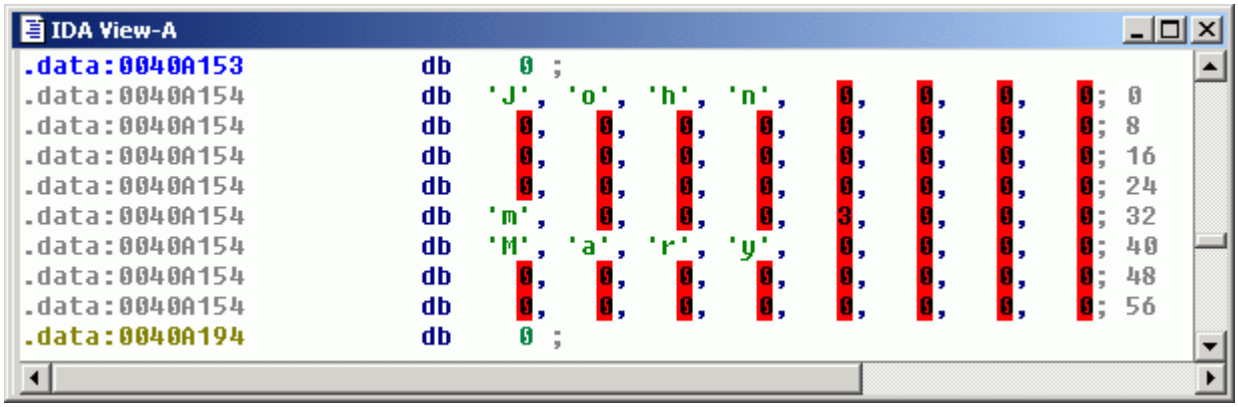
In C, ASCII strings are represented as arrays of chars. How does IDA deal with arrays?

We begin by defining the first element of the array with the usual commands. In this case, we set the first element type as byte and set its format as char. Then we press the '*' key or use the *Array* command from the *Edition* toolbar to create the actual array. A dialog box opens, allowing us to specify various settings.



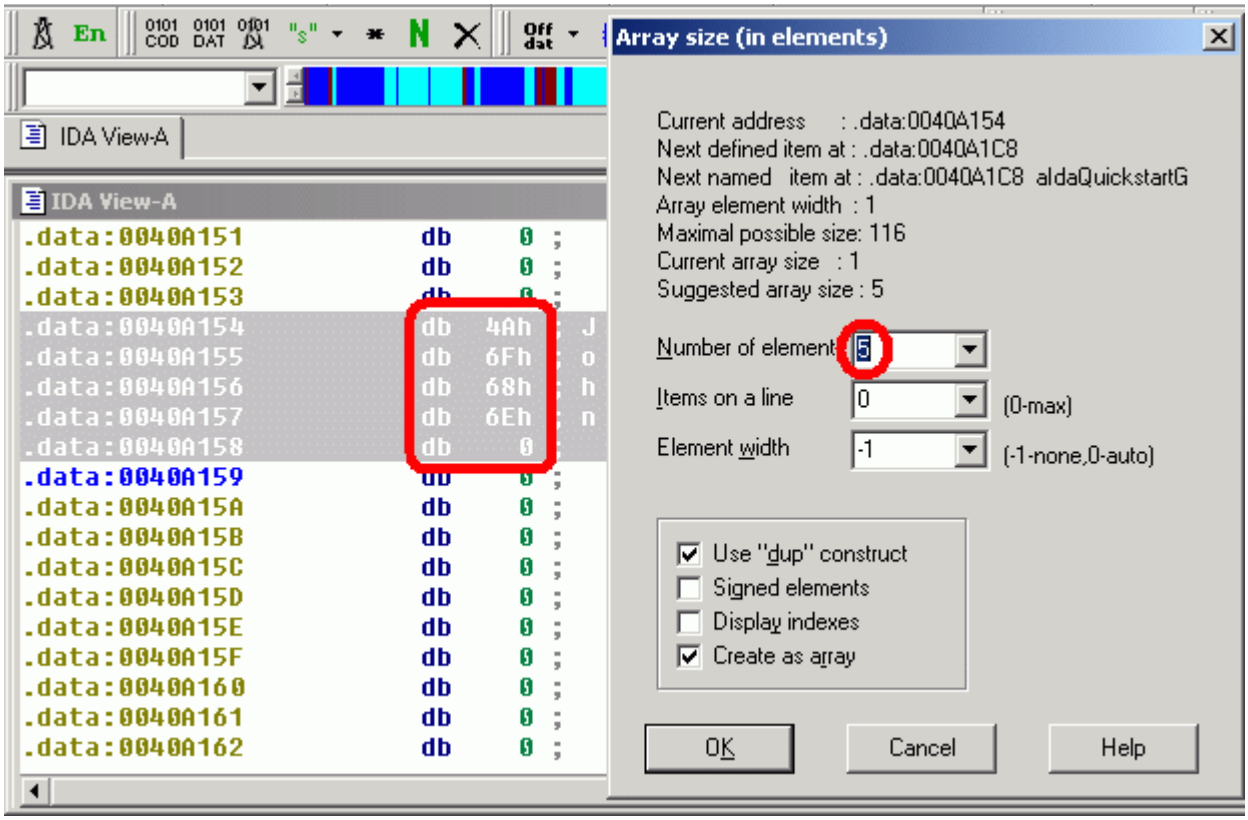
IDA suggests an array size, based on the maximum size it can use without undefining existing data. You can specify the number of elements to put on a line, and *Element width* allows you to align items. The *Use dup construct* option allows to group similar consecutive bytes and the *Display index option* displays array indexes as comments.

For example, if we create an array of 64 elements, with 8 elements on a line, a width of 4 for each element, without dup constructs, and with index comments, we obtain the following array:



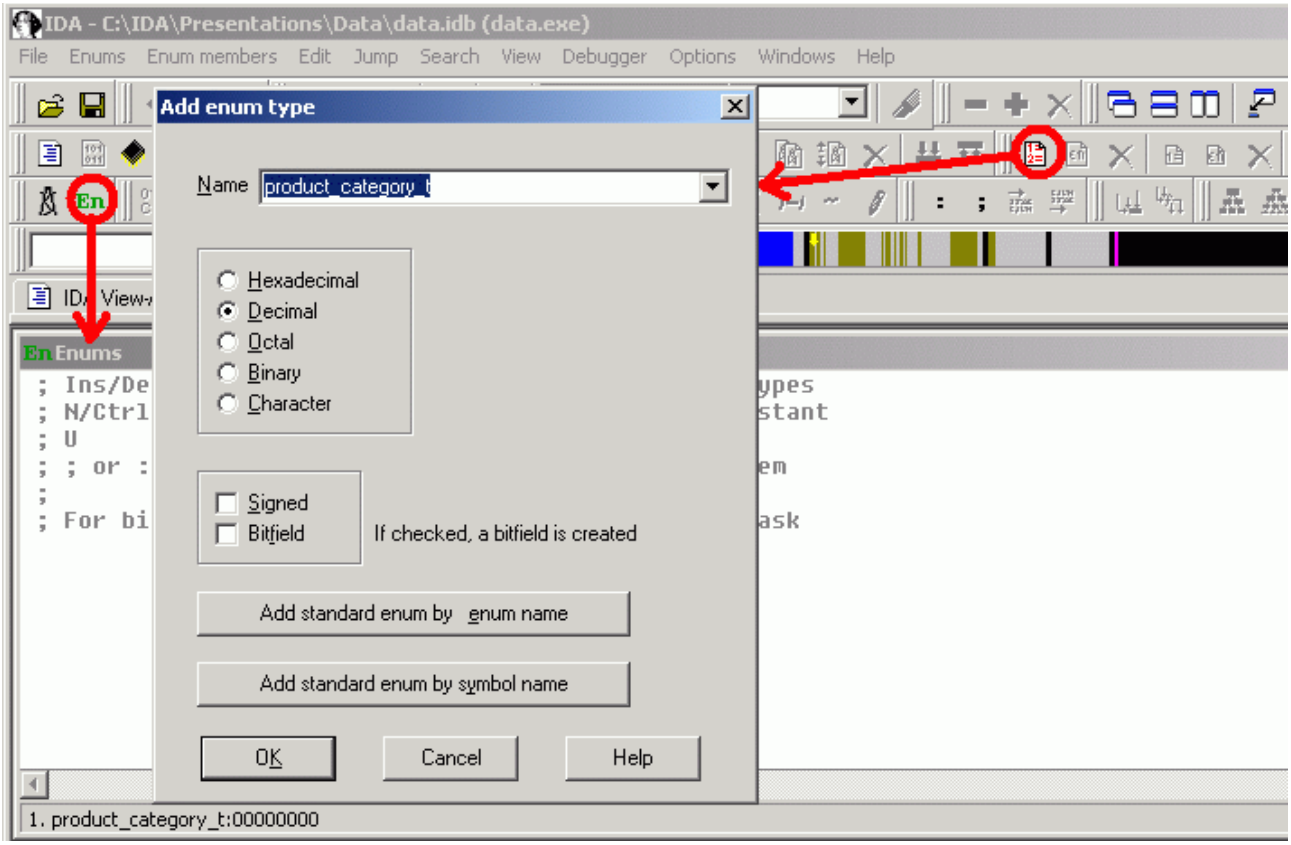
When IDA Pro can't represent bytes in the selected type - chars in this case - it highlights them in red.

It is also possible to select a range: IDA will then propose to create an suitable array.

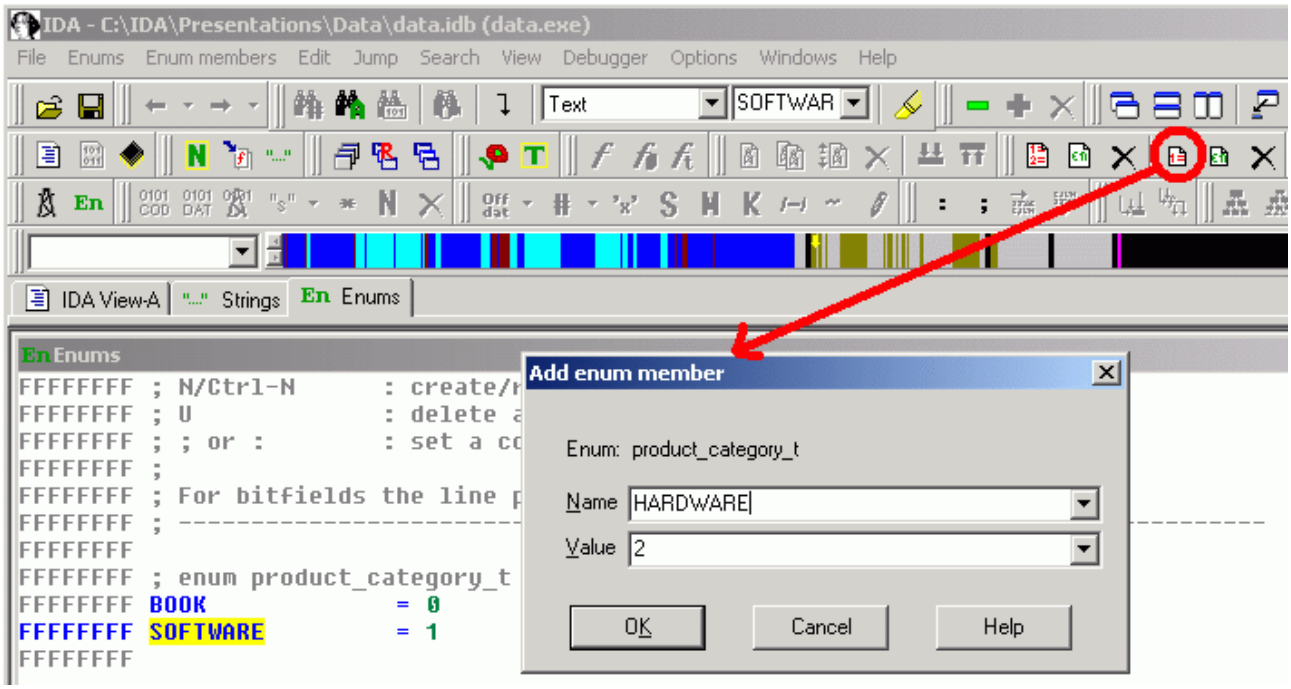


Remember the *product_category_t* type defined in the C program ? Let's try to define it in IDA by using *Enumerations*.

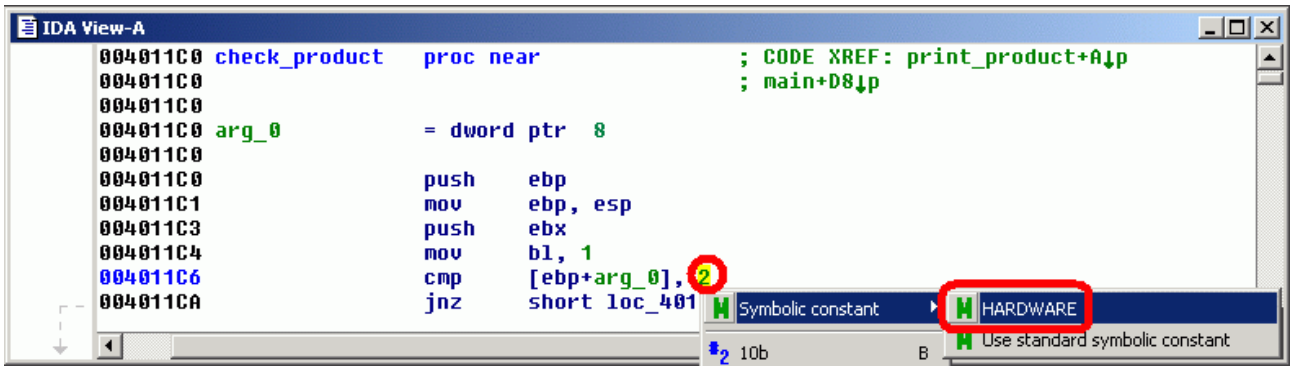
First, we open the *Enumerations* window and create a new enumeration type.



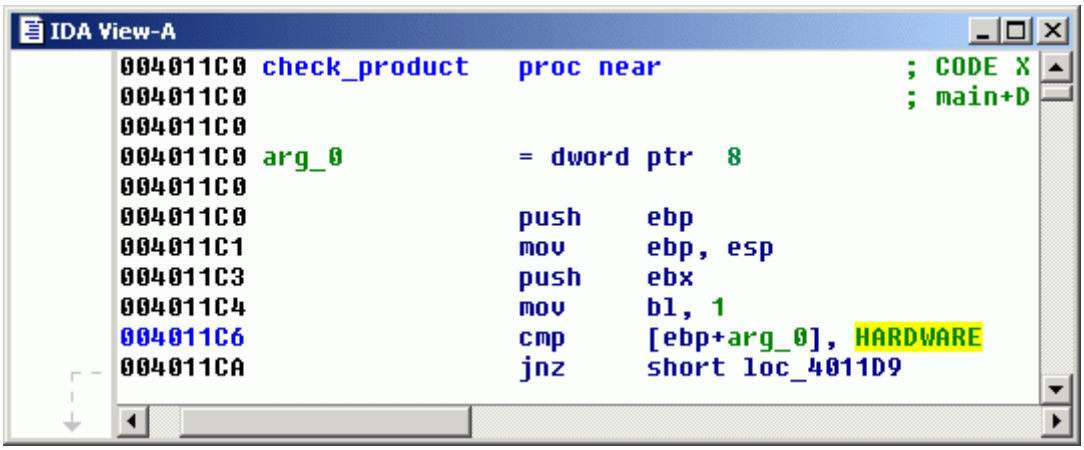
We add our enumeration values.



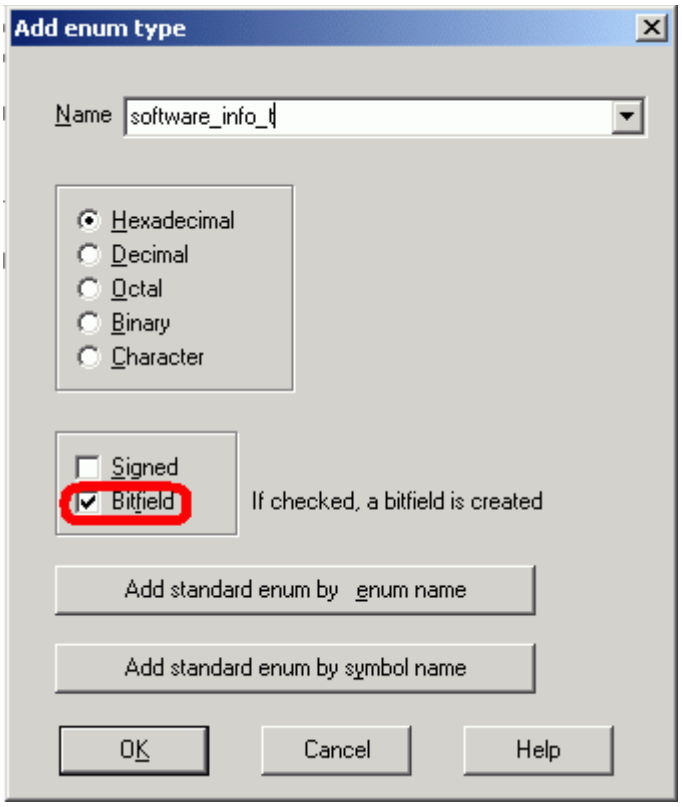
In the *check_product()* function, we can format operands using this enumeration. Right click on the numeric value to get the popup menu, and select *Symbolic constant*: IDA will list all enumeration values matching the current numeric value.



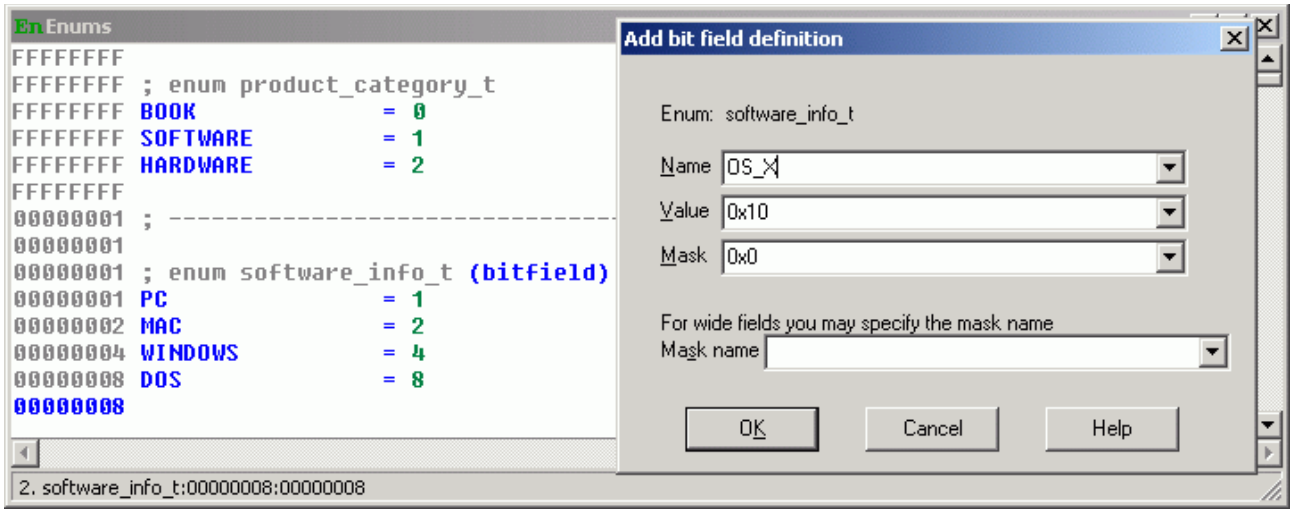
Once applied, we get the following result.



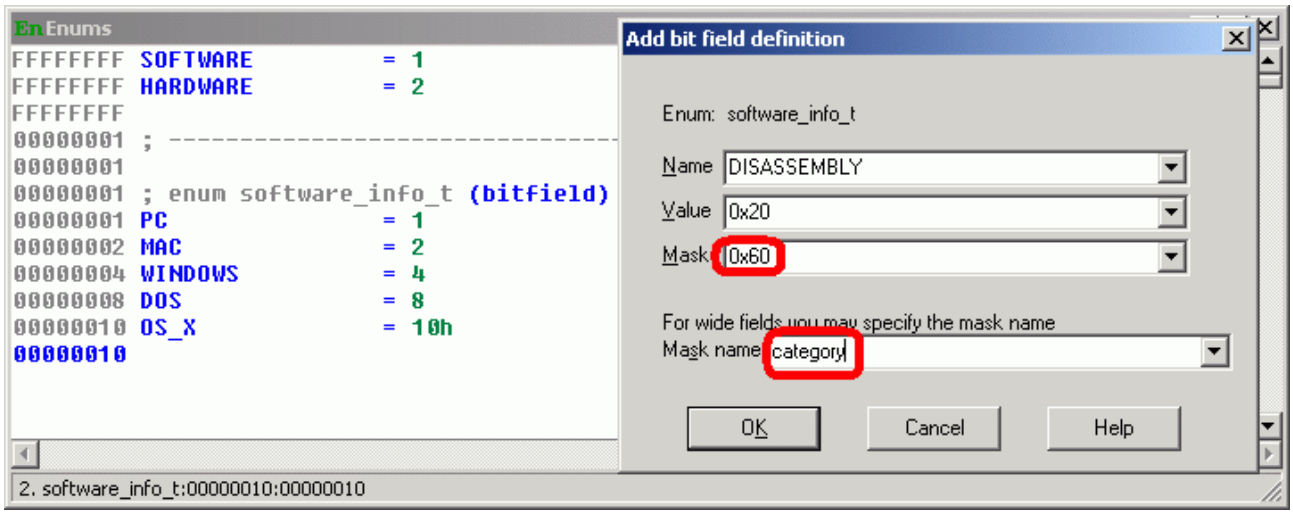
Now, let's try to define the bitfields defined in the *software_info_t* structure from our C program. From IDA's point of view, bitfields are only special enumeration types. We select the *Bitfield* option in the enumeration type creation dialog box.



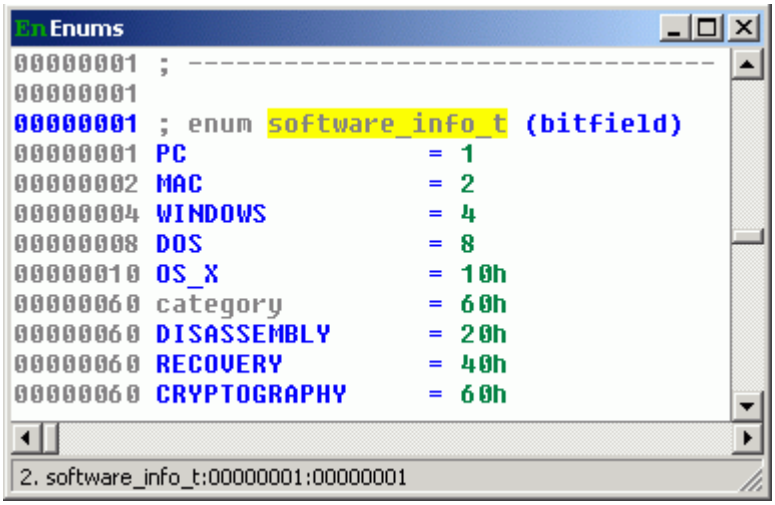
Remark we have two different types of bitfields in our program. The *plateform* and *os* bitfields contain a mask of combined values (by using the *or* boolean operator): a product can exist on **several** plateforms or OS. On the other hand, the *category* bitfield contains a number representing one category: at the same time, a product can only belong to one category! For IDA, a bitfield can only contain **one** value in a specified mask. So, to represent the *plateform* and *category* bitfields, we have to create tiny bitfields of one bit for each value, to allow combining those.



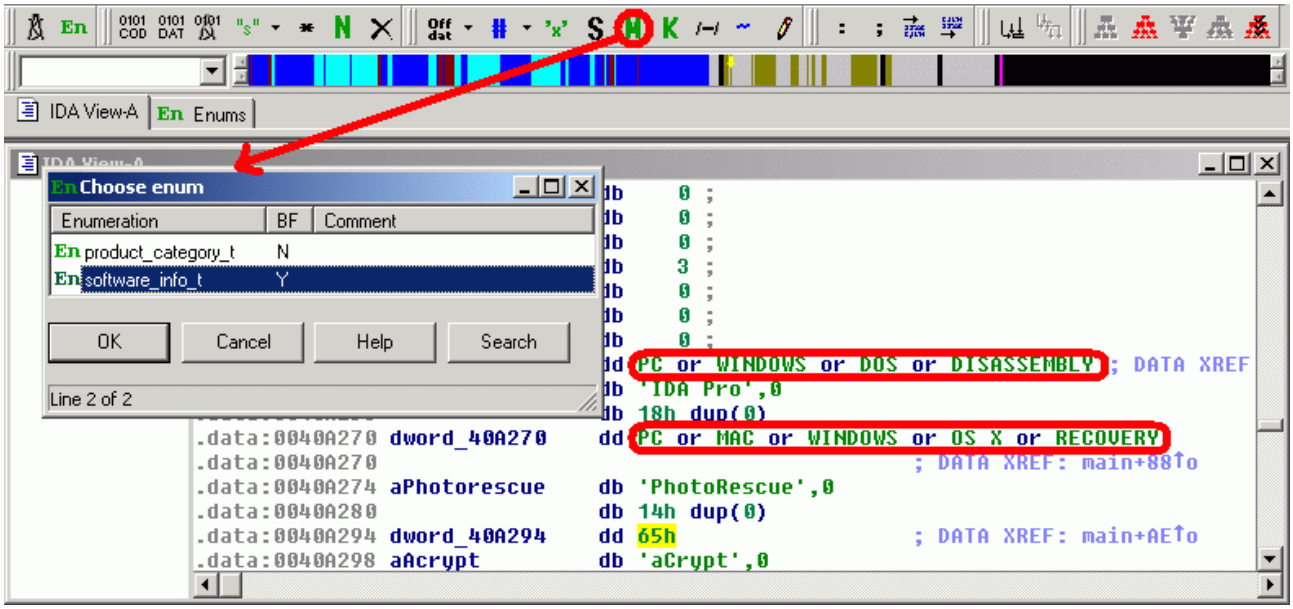
Now, we can create the *category* bitfield, with a mask value of 0x3 (2 bits). We specify a member name, a member value and the bitfield mask. We can also specify a mask name: this one will not be used by IDA, it is only intended as a memory helper.



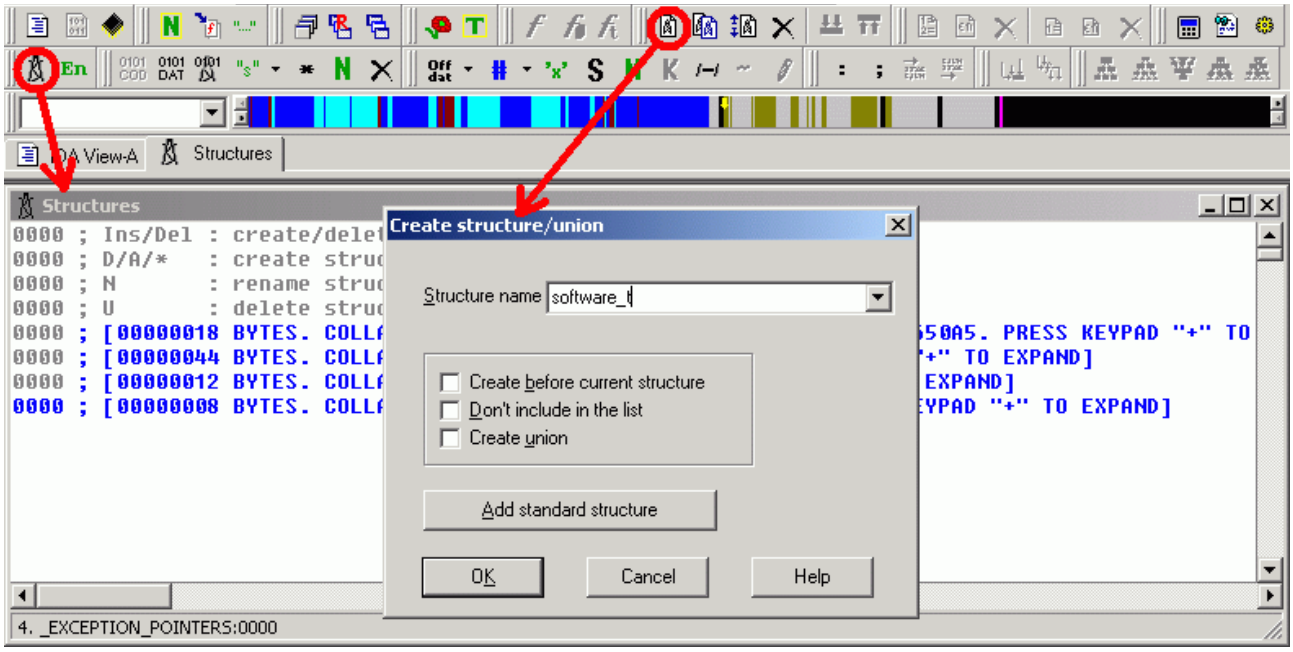
Once all bitfields are inserted, we get the following definition.



Use the the *Enum member* command from the *Operands* toolbar to apply those bitfield definitions to our software data.

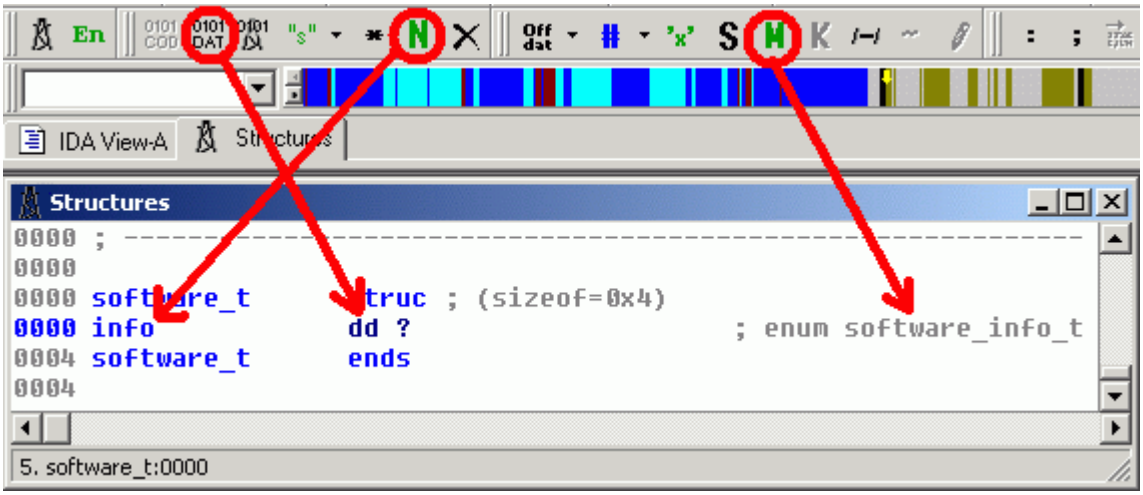


Our program contains lots of structures. Let's declare these in IDA, to see how it can improve the disassembly's readability. First, we must open the *Structures* window and create a new structure type.

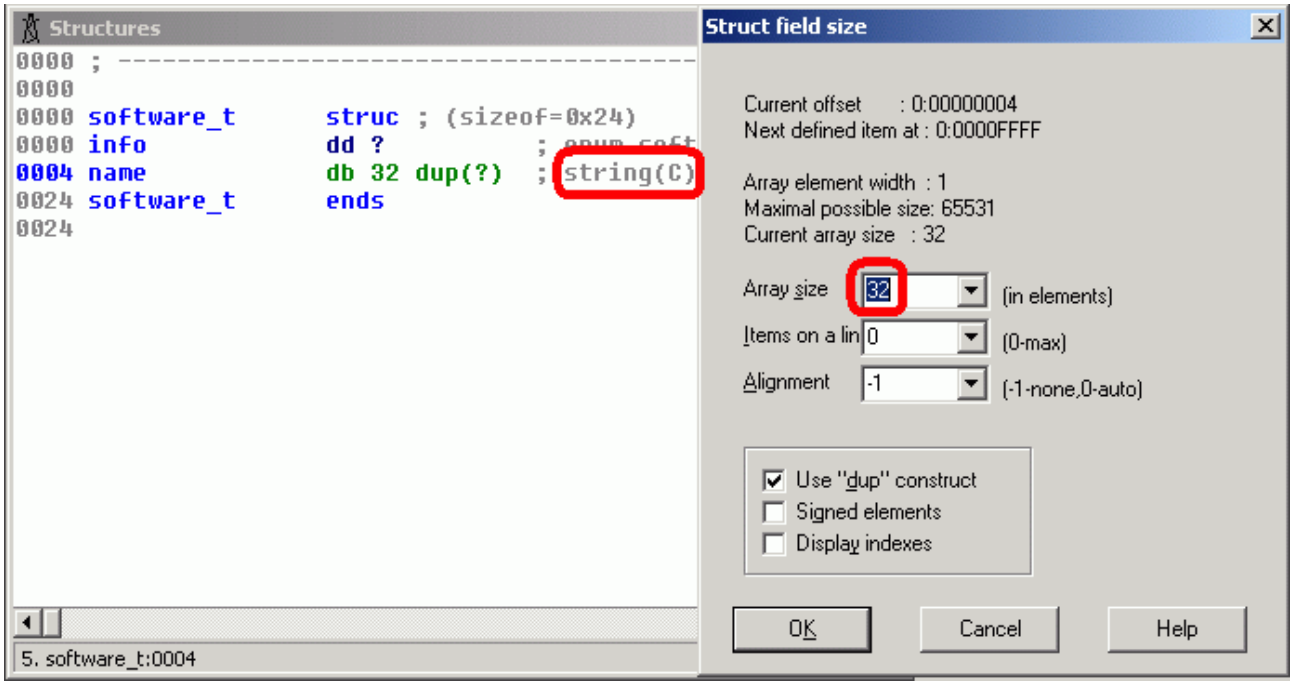


We define our structure members as if we were defining data in a disassembly view. Let's define the first member of the *software_t* structure. Press 'D' until we obtain a *dd*, indicating the value is stored in a *dword*.

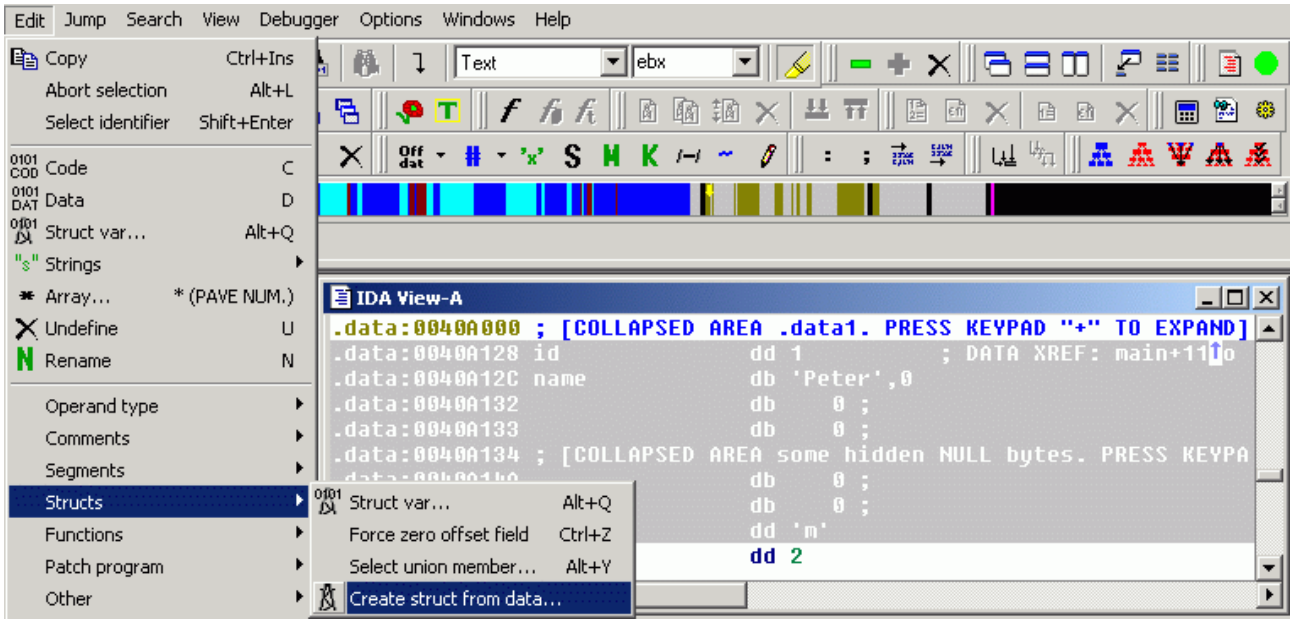
We specify its format as our previously defined *software_info_t* enumeration, and we give it an appropriate name, *info*, by using the *Rename* command.



We define the second member of our structure by using the usual *ASCII* command. In this case, IDA opens a special dialog box asking us the size of the string.



We can also create a structure type from already defined data. For example, suppose we defined data in a range with particular types and names precisely representing our *customer_t* structure. We can create this structure in IDA from these definitions, by selecting the adequate range and using the *Create struct from data* command.



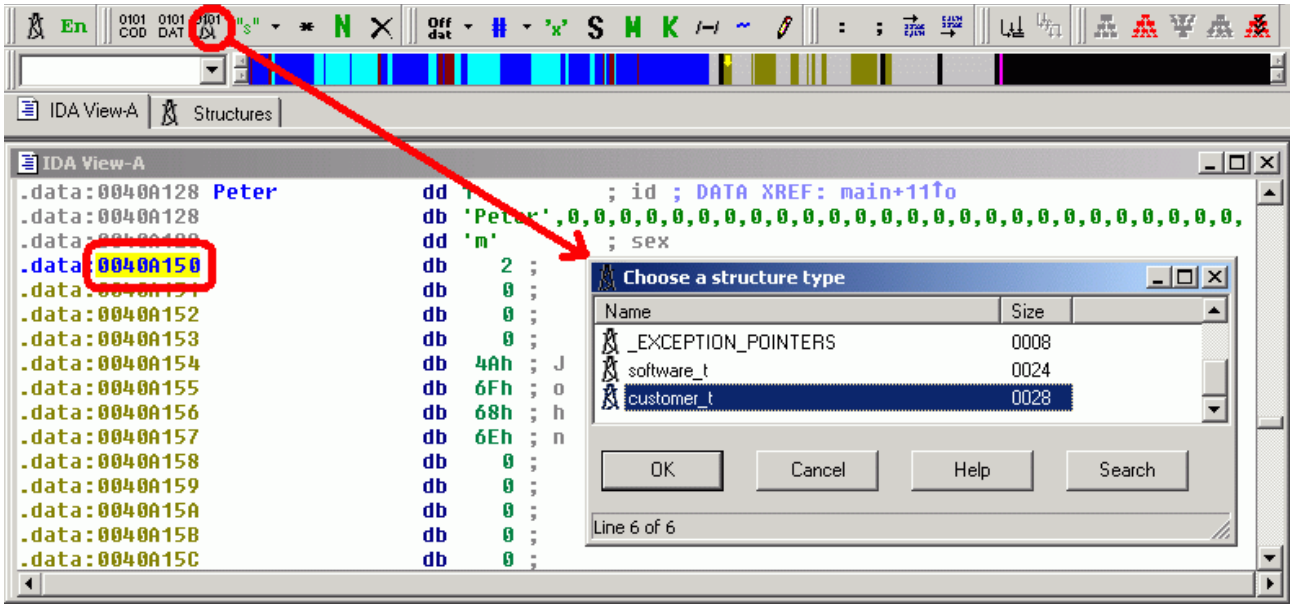
Once we run this command, IDA creates a corresponding structure and opens the Structures window. To obtain a perfect structure type, we just correct the length of the name member to 32 bytes (as defined in our source code) by pressing the 'A' key, and give the structure a more accurate name.

What can we do with these structure types ? IDA offers us two possibilities:

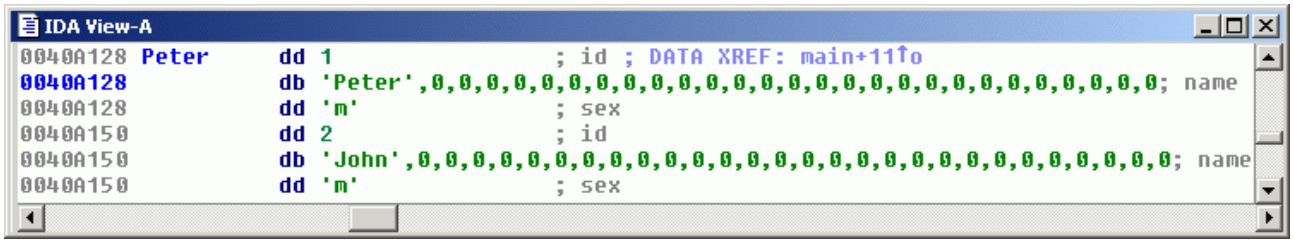
- Apply structure types to initialized data in the program.
- Convert operands as offsets inside structures.

We will present both possibilities in the continuation of this tutorial.

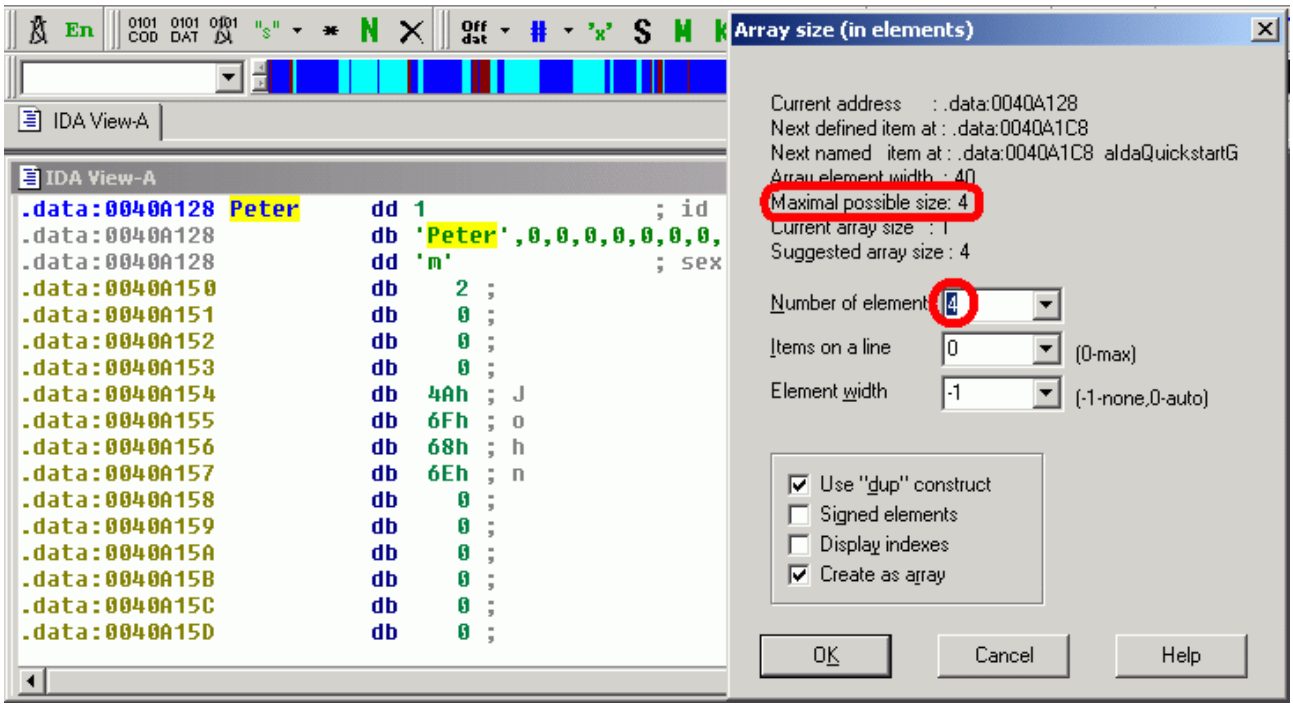
Let's define data containing information about one of our customer, John, as a *customer_t* structure. We put the cursor on the first byte of the data representing the structure, and use the *Struct var* command.



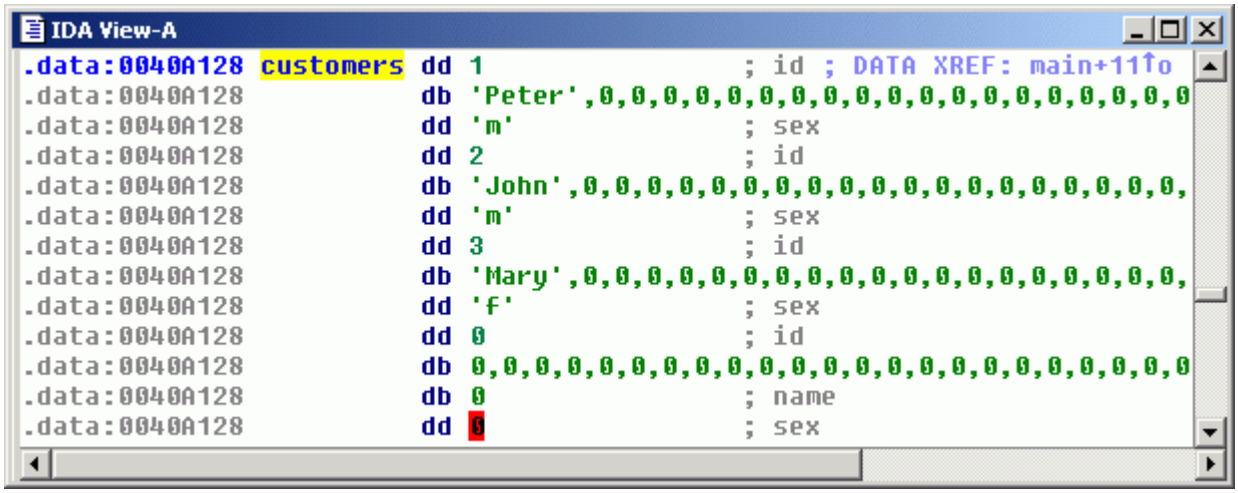
We obtain a new structure variable. Notice how IDA displays structure member names as comments.



By looking at our source code, we know *customers* is an array containing 4 elements. We also previously defined *Peter* and *John* as *customer_t* structures. Let's undefine the *John* structure, and create our *customers* array, by pressing the '*' key on the *Peter* structure. IDA opens the array settings dialog box and detects that the maximum array we can create contains a maximum of 4 elements.

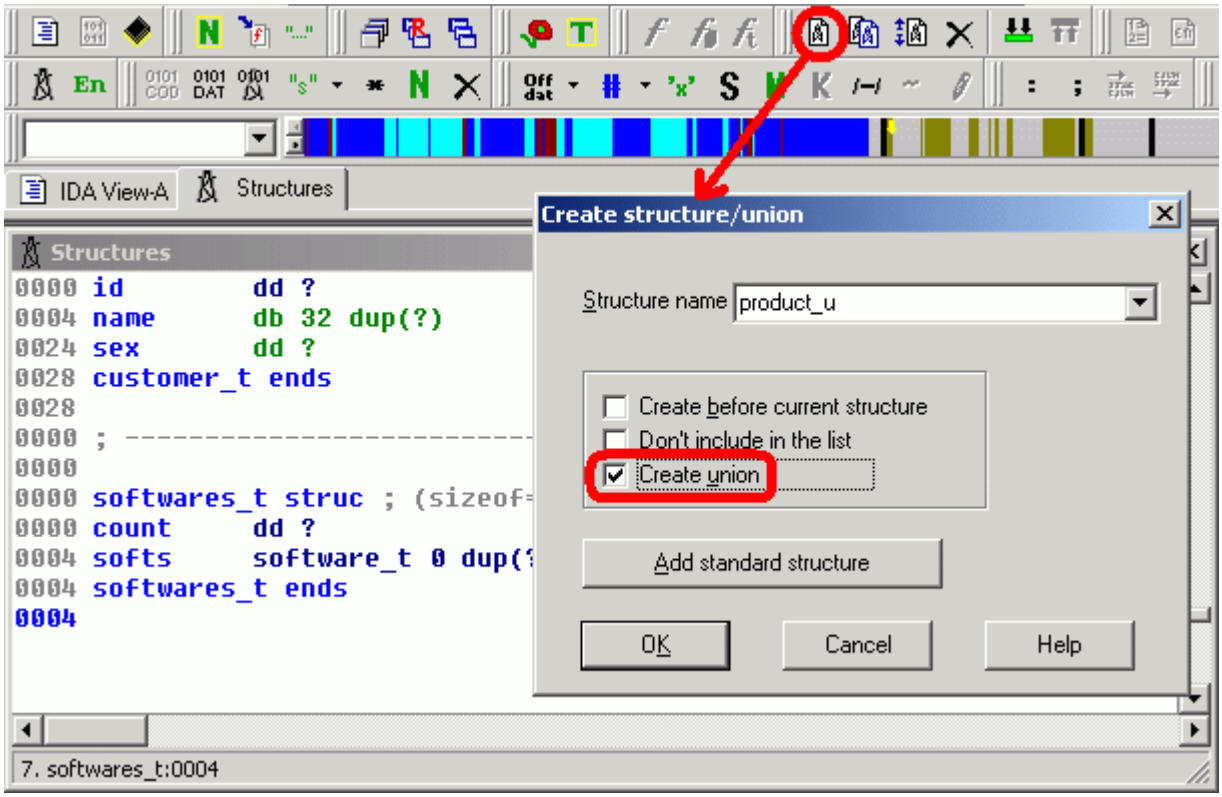


Let's create the array, and rename it properly.

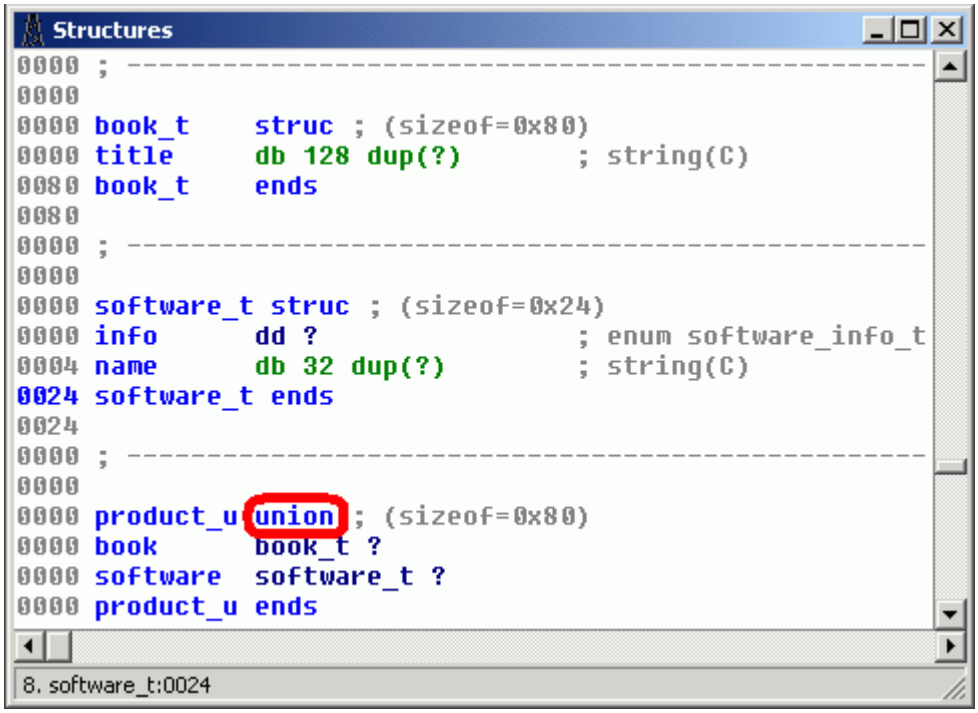


IDA allows you to define unions as easily as you define classical structures.

Let's try to define the *product_u* union. We suppose the *book_t* and *software_t* structures are already defined. For IDA, unions are special structures: so we open the *Structures* window and click on the *Add struct type* command. In this dialog box, we select the *Create union* option.

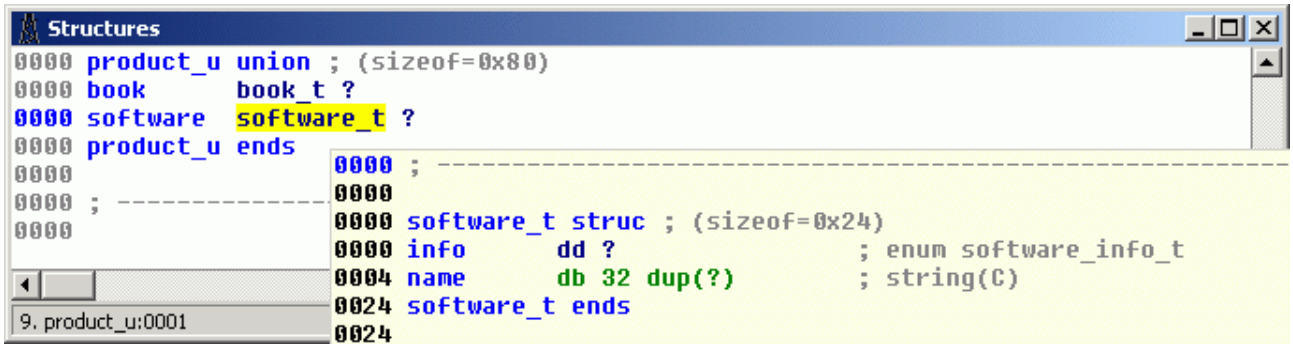


We can create union members by using all regular data definition commands. In our case, we define a *book* member of *book_t* type, and a *software* member of *software_t* type.

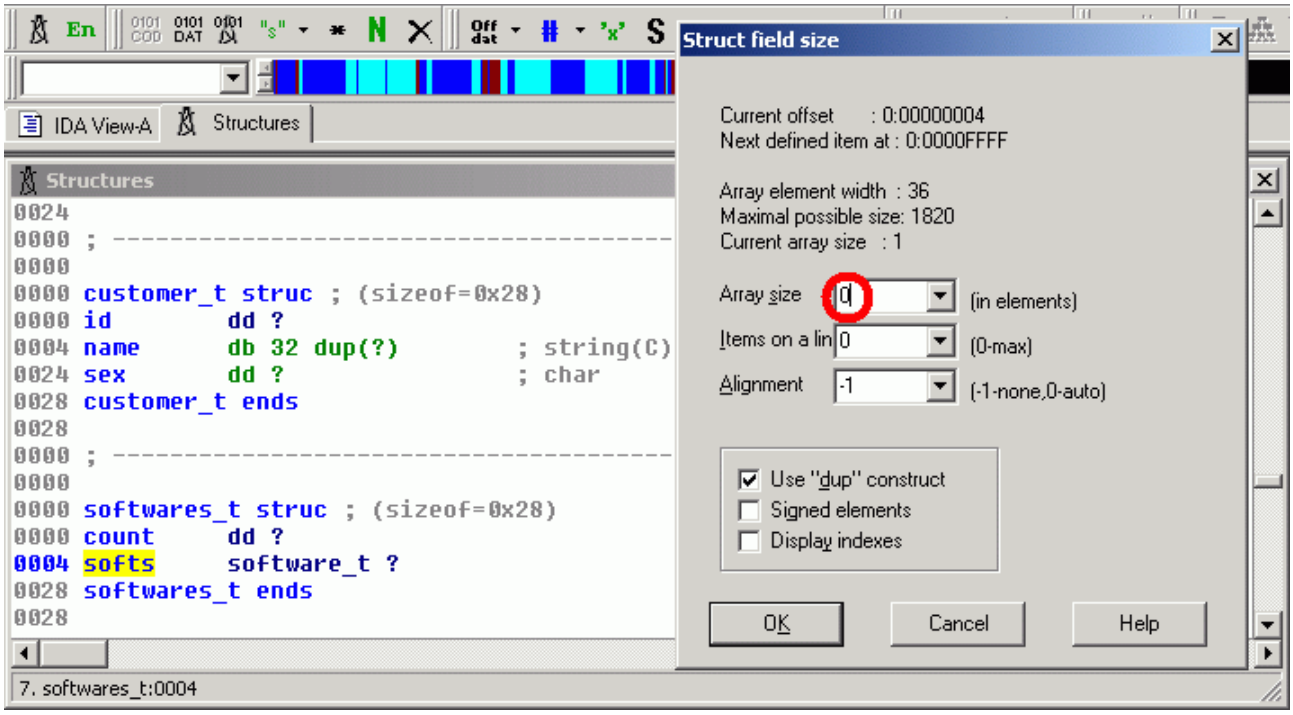


It is also possible to nest structures within structures. In fact, we just did it to create our union members in the previous example. Remember: IDA considers unions are nothing more than special structures.

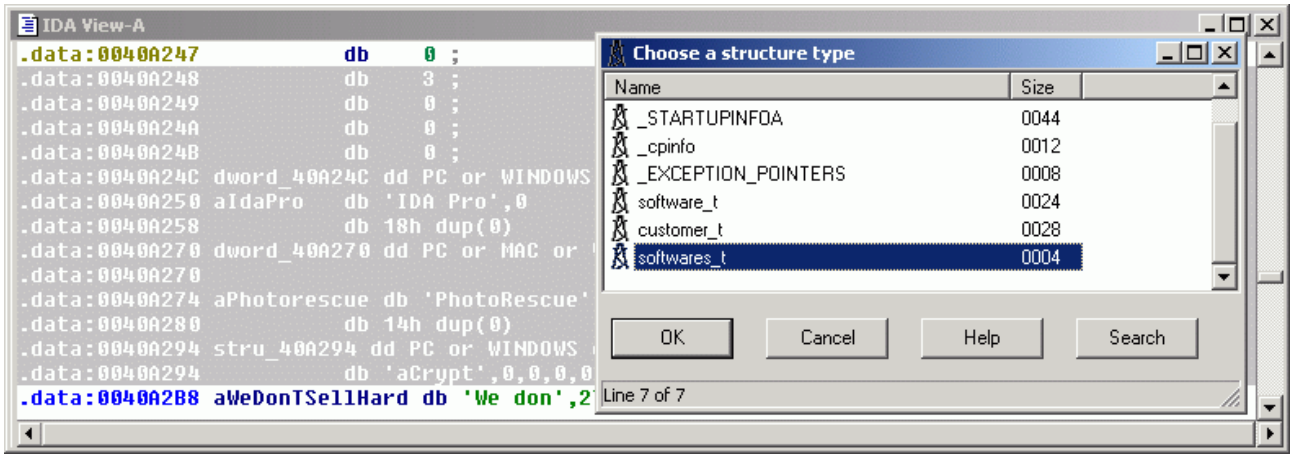
Simply put your mouse over a member's structure name to see how its associated structure type is declared.



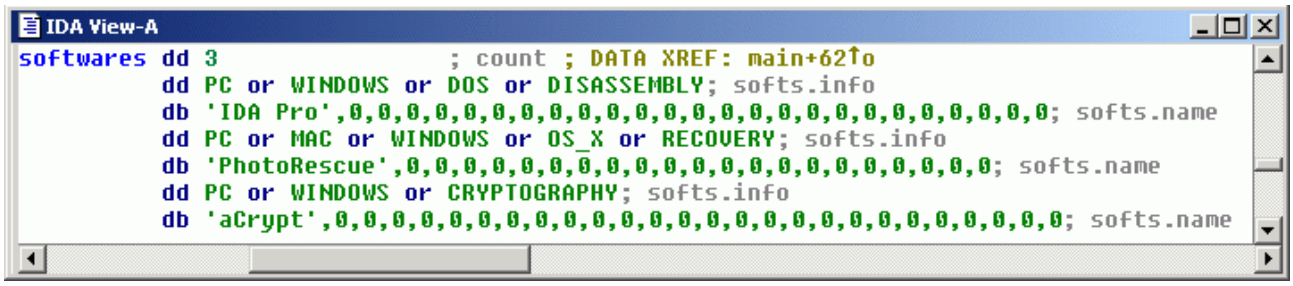
Now, let's have a look at the *softwares_t* structure. The length of the *softs* field of this structure is not specified. To create such structures in a disassembly, we must create a special kind of structure called a variable sized structure. Such structures are created just as a normal structure: the only difference is that the last member of the structure should be declared as an array with 0 elements.



Since the structure size can't be calculated by IDA, we specify the desired structure size by selecting an area.

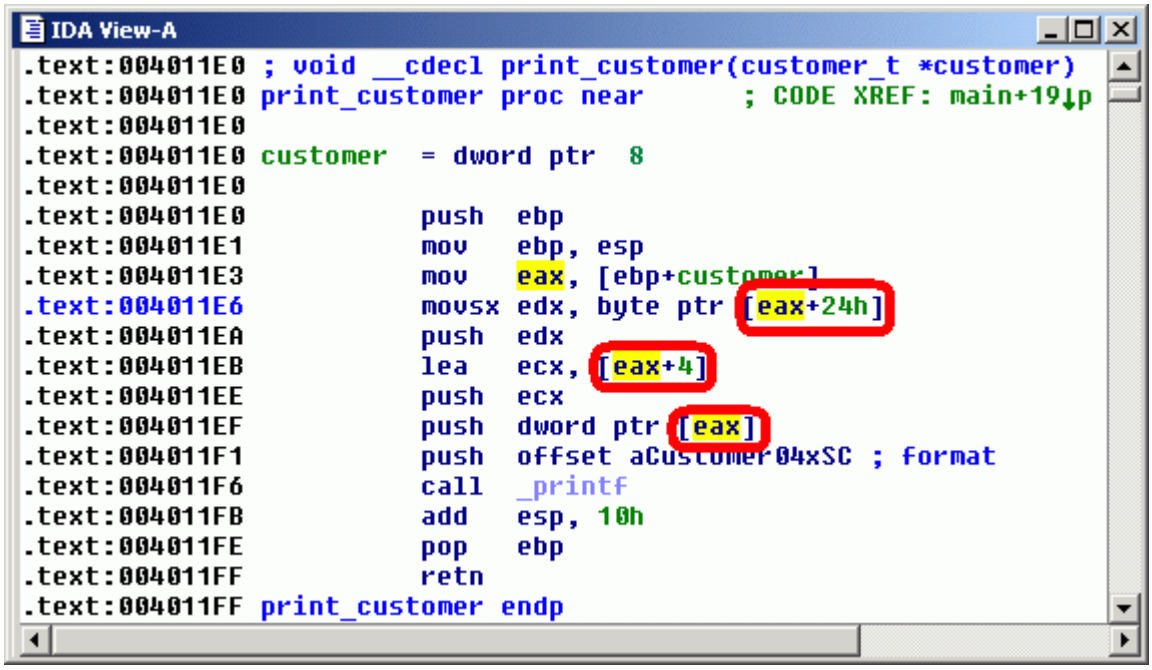


Notice how IDA applies all type information and add member names as comments.

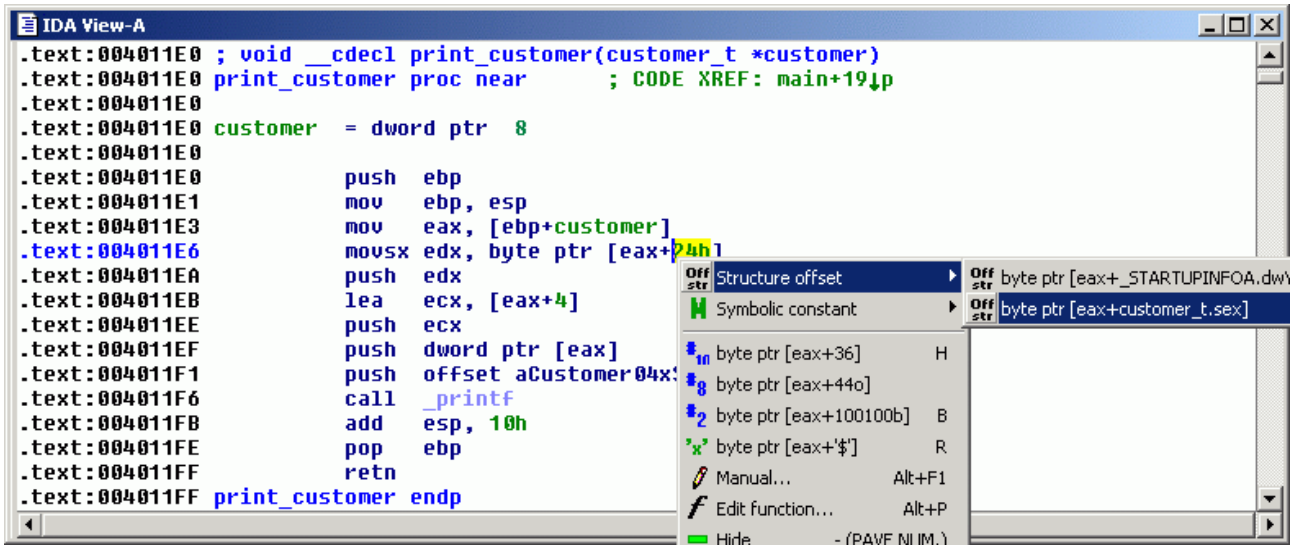


Now that we know how to declare the unions and structures we need, let's have a look at how we can transform numeric operands to offsets inside those structures.

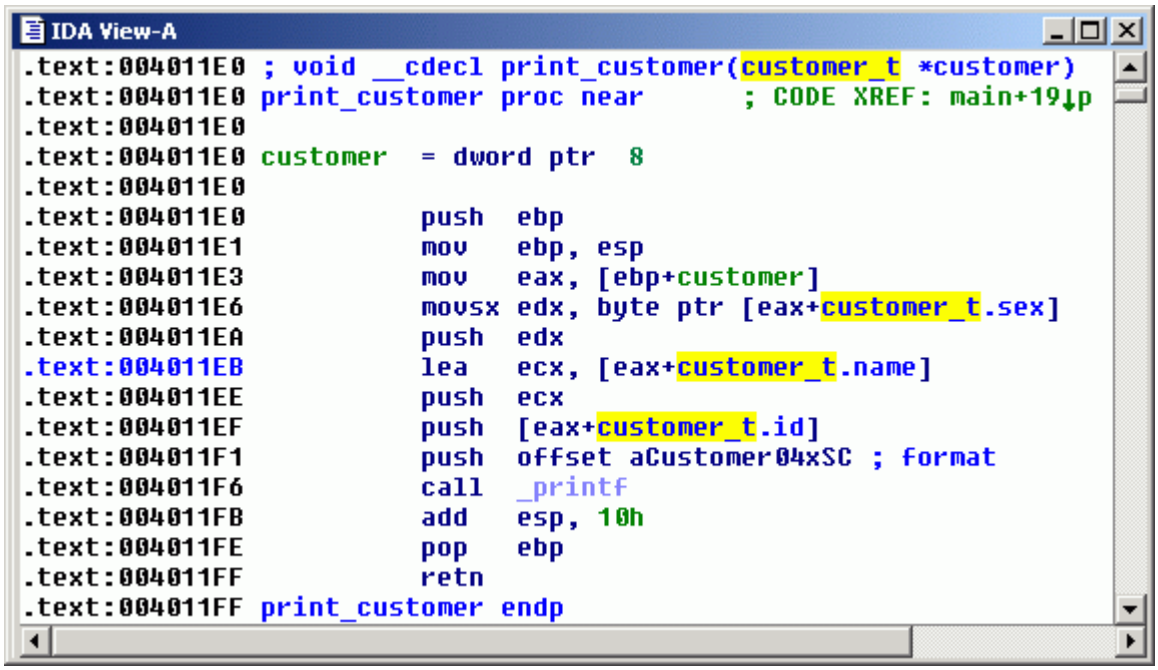
In the *print_customer()* function, we know that the only argument is a pointer to a *customer_t* structure. The EAX register is initialized with the value of this pointer: it points to a *customer_t* structure. Therefore, we deduce that all operands of the form [EAX+...] represent in fact offsets in the *customer_t* structure.



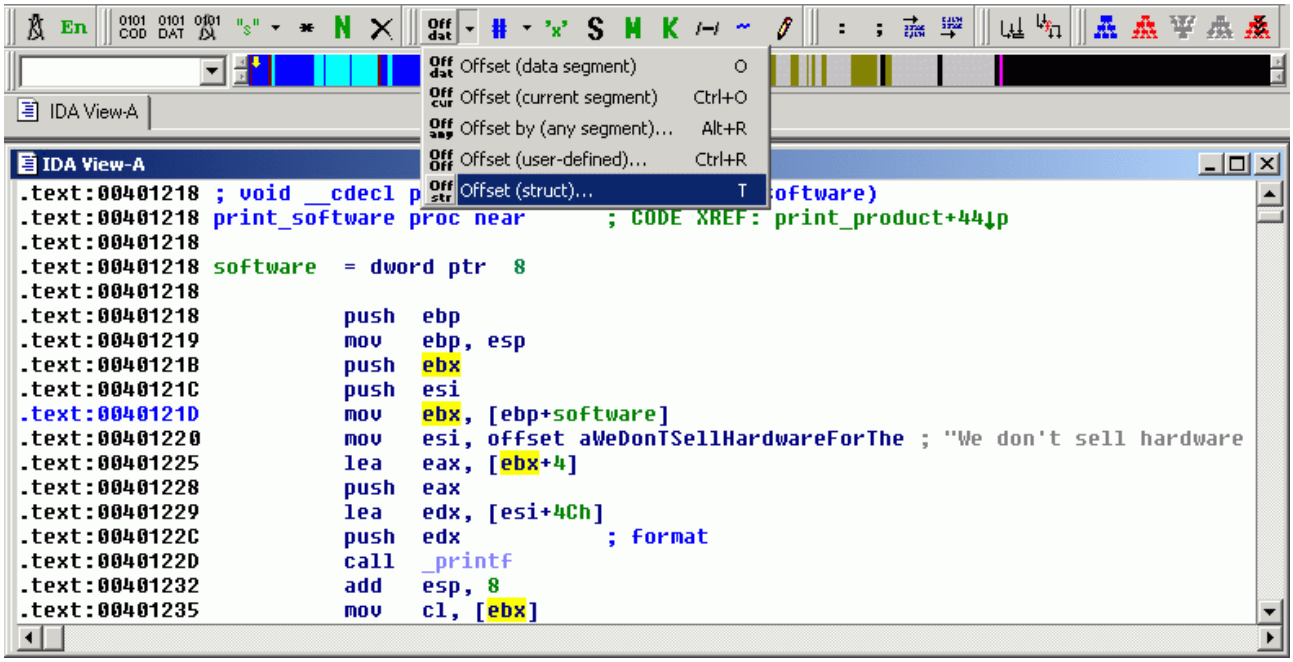
To format each operand as a structure offset, we right click on it: IDA suggests all possible structure offsets.



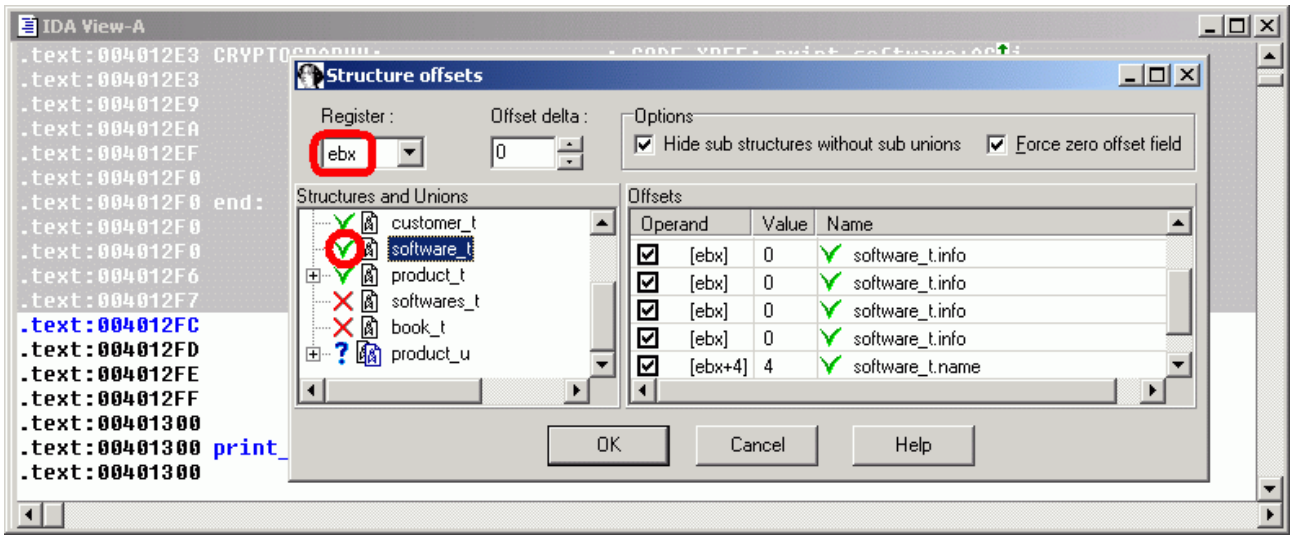
Once we have applied this command to each offset, the disassembly becomes more understandable.



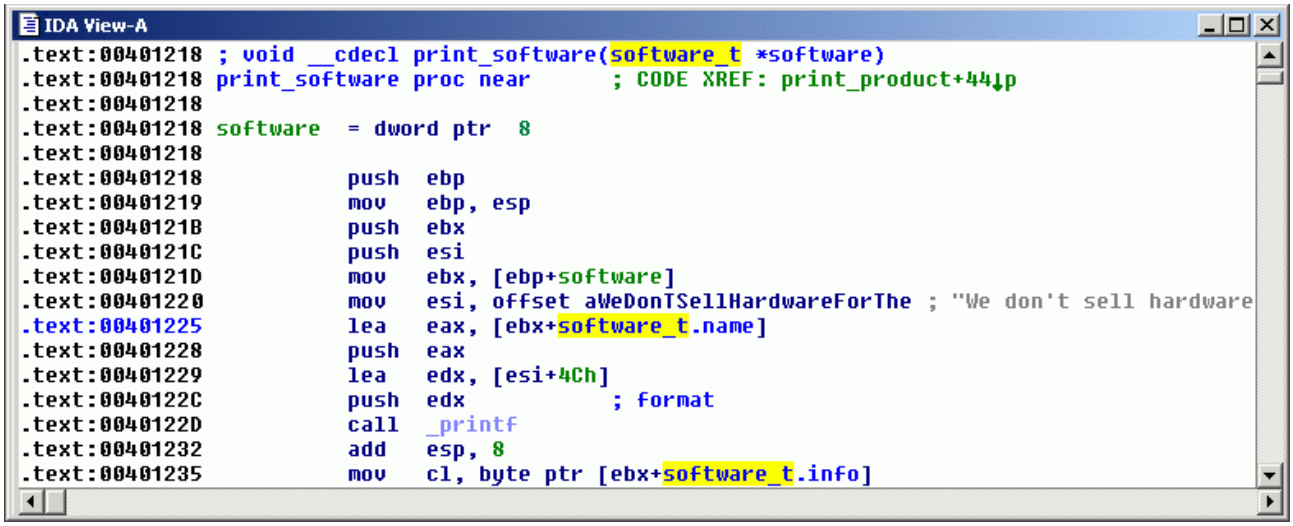
The *print_software()* function is another example of this: the EBX register is initialized with the value of a pointer to a *software_t* structure. Remark this EBX register is used throughout the function to access this structure. Fear not, in a single operation IDA can change all the offsets in a selection: click on the *Offset (struct)* command in the *Operands* toolbar.



The *Structure offsets* dialog box opens. Let's select the EBX register in the list of available registers. The tree view on the left of the dialog box shows all structures defined in IDA. The list on the right shows all operands related to EBX. If we select a structure in the tree view, IDA formats the selected operands as offsets into this structure. Different symbols help to determine if all selected operands match existing offsets for the selected structure. In our case, the *software_t* structure seems to match all our operands.

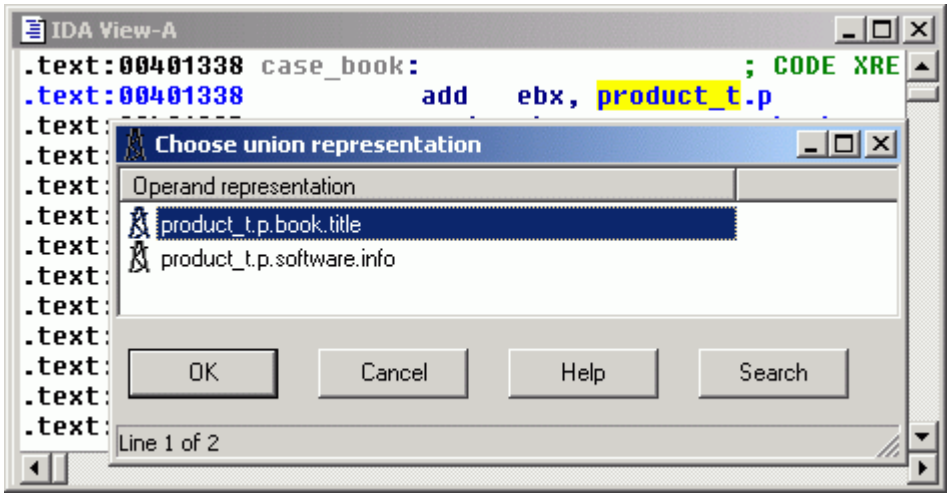


Once applied, we obtain the following result:

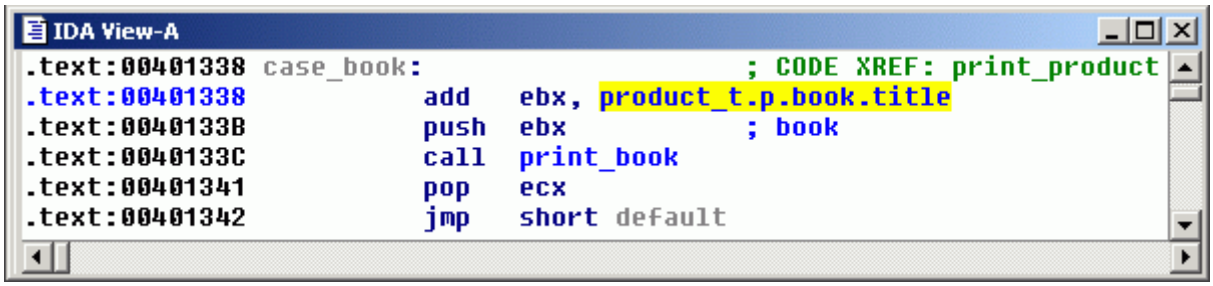


The *print_product()* uses the EBX register to point to a *product_t* structure. At the end of this function, depending on the product category, we call the adequate function to print product information.

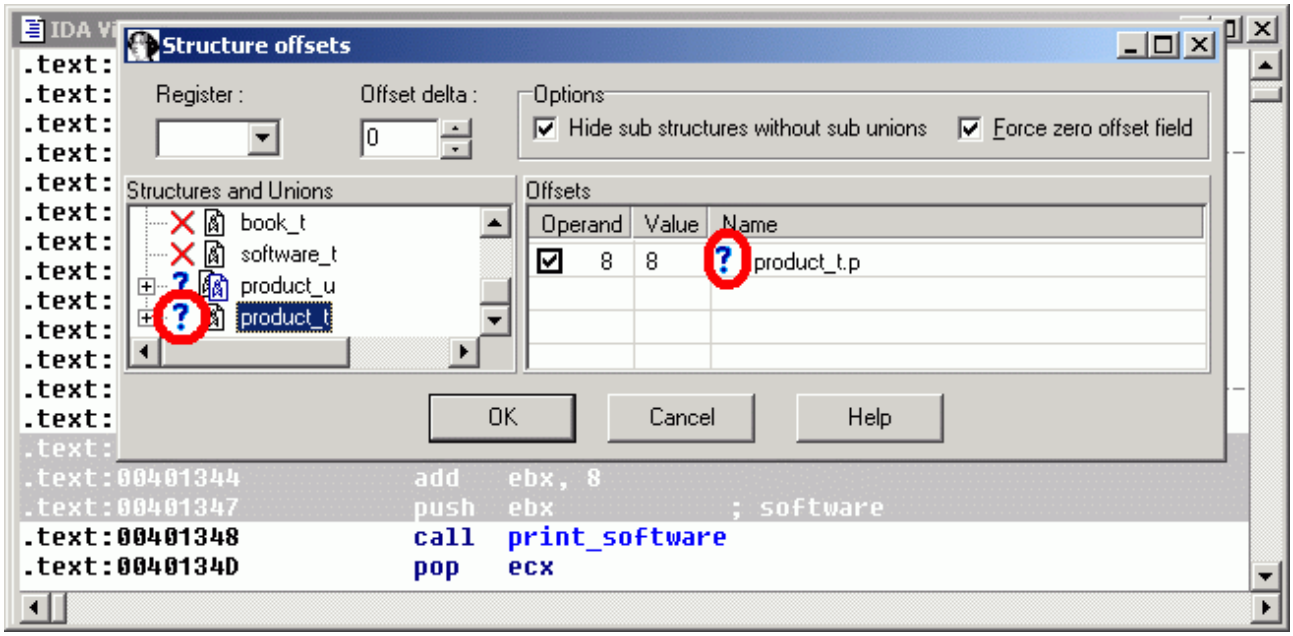
But the structure offset will have a different representation if it represents an offset in the first member of the *product_u* union or an offset in the second member of this union! To choose the adequate member, use the *Select union member* from the *Edit struct* menu. In the dialog, we select the desired union member.



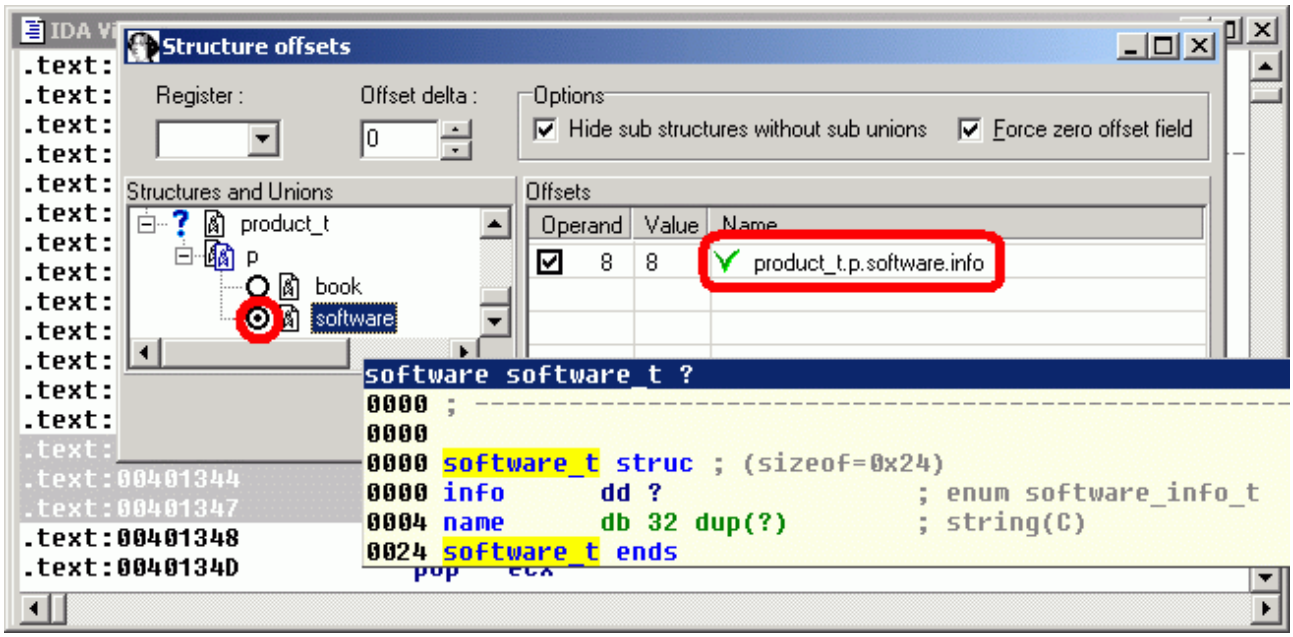
And here is the result.



The *Structure offsets* dialog shows how choosing an union member affects the offset representation. If we select an area and open this dialog, we remark that union types are preceded by a *?* symbol.

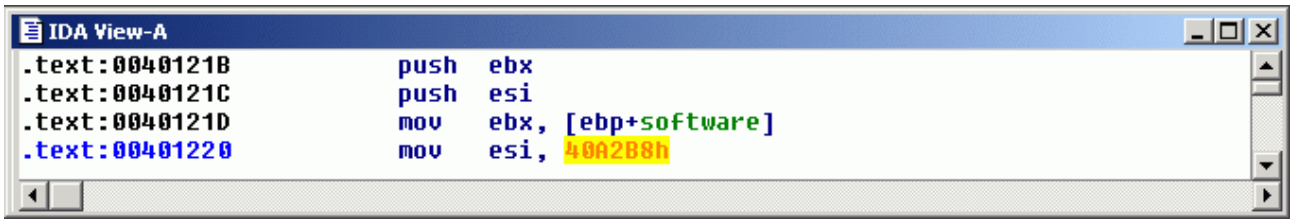


If we expand the adequate branch of the tree, we can choose the union member that represents operand offsets. Once a union member is selected (*software* in our case), IDA shows by a green symbol that the offset matches a record in this union member.

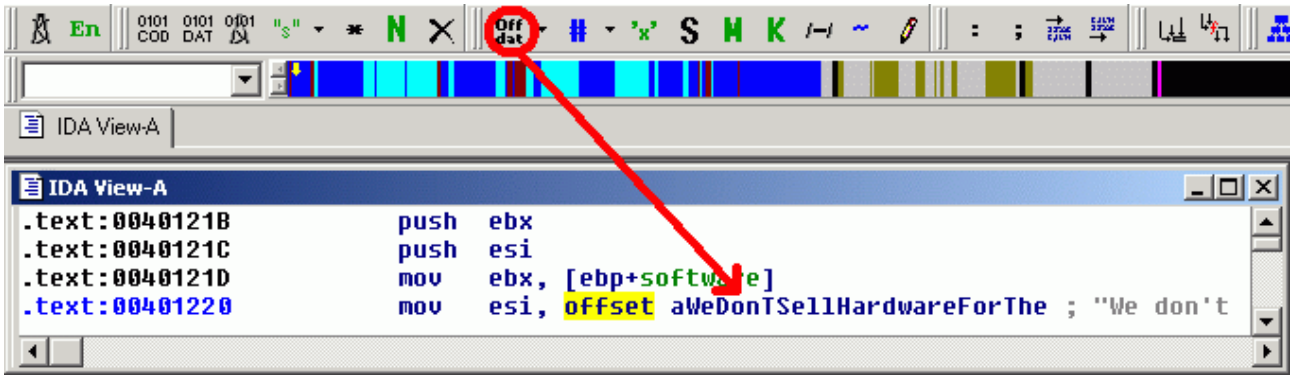


Finally, make full use of hints in the tree view to see structure type declarations, as in the previous screenshot.

IDA can also represents operands as offsets in the disassembled program. In the following example, the orange color indicates a possible valid reference.



Use the *Offset* button from the *Operands* toolbar to convert this operand to an offset.



To end this tutorial, we propose our final interactively disassembled code.

```
; ---------------------------------------------------------------
customer_t struc ; (sizeof=0x28)
id dd ?
name db 32 dup(?) ; string(C)
sex dd ? ; char
customer_t ends

; ---------------------------------------------------------------
softwares_t struc ; (sizeof=0x4, variable size)
count dd ?
softs software_t 0 dup(?)
softwares_t ends

; ---------------------------------------------------------------
book_t struc ; (sizeof=0x80)
title db 128 dup(?) ; string(C)
book_t ends

; ---------------------------------------------------------------
software_t struc ; (sizeof=0x24)
info dd ? ; enum software_info_t
name db 32 dup(?) ; string(C)
software_t ends

; ---------------------------------------------------------------
product_u union ; (sizeof=0x80)
book book_t ?
software software_t ?
product_u ends

; ---------------------------------------------------------------
product_t struc ; (sizeof=0x88)
id dd ?
category dd ? ; enum product_category_t
p product_u ?
product_t ends

; ---------------------------------------------------------------
; enum product_category_t
BOOK = 0
SOFTWARE = 1
HARDWARE = 2

; ---------------------------------------------------------------
; enum software_info_t (bitfield)
PC = 1
MAC = 2
WINDOWS = 4
DOS = 8
OS_X = 10h
category = 60h
DISASSEMBLY = 20h
RECOVERY = 40h
CRYPTOGRAPHY = 60h
```

```
;
; +-------------------------------------------------------------------------+
; |     This file is generated by The Interactive Disassembler (IDA)        |
; |     Copyright (c) 2005 by DataRescue sa/nv, <ida@datarescue.com>        |
; |                  Licensed to: Eric <eric@datarescue.be>                 |
; +-------------------------------------------------------------------------+
;
; File Name   : C:\IDA\Presentations\Data\data.exe
; Format      : Portable executable for IBM PC (PE)
; Section 1. (virtual address 00001000)
; Virtual size                  : 00009000 (  36864.)
; Section size in file          : 00008E00 (  36352.)
; Offset to raw data for section: 00000600
; Flags 60000020: Text Executable Readable
; Alignment     : 16 bytes ?


unicode         macro page,string,zero
  irpc c,<string>
  db '&c', page
  endm
  ifnb <zero>
  dw zero
  endif
endm

  .686p
  .mmx
  .model flat

; ---------------------------------------------------------------------------

; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use32
  assume cs:_text
  ;org 401000h
; [COLLAPSED AREA .text1. PRESS KEYPAD "+" TO EXPAND]

; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦


; Attributes: bp-based frame

; int __cdecl check_software(software_info_t software_info)
check_software proc near ; CODE XREF: main+108p

software_info= byte ptr  8

  push  ebp
  mov   ebp, esp
  mov   al, 1
  mov   dl, [ebp+software_info]
  and   edx, PC or MAC
  test  dl, PC
  jz    short not_PC
  mov   cl, [ebp+software_info]
  and   ecx, PC or MAC
  test  cl, MAC
  jnz   short end
```

```
  mov   dl, [ebp+software_info]
  shr   edx, 2
  and   edx, (WINDOWS or DOS or OS_X) >> 2
  test  dl, OS_X >> 2
  jz    short end
  xor   eax, eax
  jmp   short end
; ---------------------------------------------------------------------------

not_PC:             ; CODE XREF: check_software+Ej
  mov   cl, [ebp+software_info]
  and   ecx, PC or MAC
  test  cl, MAC
  jz    short not_MAC
  mov   dl, [ebp+software_info]
  and   edx, PC or MAC
  test  dl, PC
  jnz   short end
  mov   cl, [ebp+software_info]
  shr   ecx, 2
  and   ecx, (WINDOWS or DOS or OS_X) >> 2
  test  cl, WINDOWS >> 2
  jnz   short not_windows
  mov   dl, [ebp+software_info]
  shr   edx, 2
  and   edx, (WINDOWS or DOS or OS_X) >> 2
  test  dl, DOS >> 2
  jz    short end

not_windows:        ; CODE XREF: check_software+4Fj
  xor   eax, eax
  jmp   short end
; ---------------------------------------------------------------------------

not_MAC:            ; CODE XREF: check_software+36j
  xor   eax, eax

end:                ; CODE XREF: check_software+19j check_software+27j ...
  xor   edx, edx
  mov   dl, al
  mov   eax, edx
  pop   ebp
  retn
check_software endp

; ---------------------------------------------------------------------------
  align 4
```

```
; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦

; Attributes: bp-based frame

; int __cdecl check_product(product_category_t product_category)
check_product proc near ; CODE XREF: print_product+Ap main+D8p

product_category= dword ptr  8

  push  ebp
  mov   ebp, esp
  push  ebx
  mov   bl, 1
  cmp   [ebp+product_category], HARDWARE
  jnz   short not_hardware
  xor   ebx, ebx
  push  offset aWeDonTSellHardwareForThe ; format
  call  _printf
  pop   ecx

not_hardware:       ; CODE XREF: check_product+Aj
  xor   eax, eax
  mov   al, bl
  pop   ebx
  pop   ebp
  retn
check_product endp


; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦

; Attributes: bp-based frame

; void __cdecl print_customer(customer_t *customer)
print_customer proc near ; CODE XREF: main+19p

customer= dword ptr  8

  push  ebp
  mov   ebp, esp
  mov   eax, [ebp+customer]
  movsx edx, byte ptr [eax+customer_t.sex]
  push  edx
  lea   ecx, [eax+customer_t.name]
  push  ecx
  push  [eax+customer_t.id]
  push  offset aCustomer04xSC ; format
  call  _printf
  add   esp, 10h
  pop   ebp
  retn
print_customer endp
```

```
; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦

; Attributes: bp-based frame

; void __cdecl print_book(book_t *book)
print_book proc near ; CODE XREF: print_product+38p

book= dword ptr  8

  push  ebp
  mov   ebp, esp
  push  [ebp+book]
  push  offset aBookS ; format
  call  _printf
  add   esp, 8
  pop   ebp
  retn
print_book endp

; ---------------------------------------------------------------------------
  align 4

; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦

; Attributes: bp-based frame

; void __cdecl print_software(software_t *software)
print_software proc near ; CODE XREF: print_product+44p

software= dword ptr  8

  push  ebp
  mov   ebp, esp
  push  ebx
  push  esi
  mov   ebx, [ebp+software]
  mov   esi, offset aWeDonTSellHardwareForThe ; "We don't sell hardware for the
moment..."...
  lea   eax, [ebx+software_t.name]
  push  eax
  lea   edx, (aSoftwareS - aWeDonTSellHardwareForThe)[esi] ; "We don't sell
hardware for the moment..."...
  push  edx          ; format
  call  _printf  add   esp, 8

  mov   cl, byte ptr [ebx+software_t.info]
  and   ecx, PC or MAC
  test  cl, PC
  jz    short not_pc
  lea   eax, (aPc - aWeDonTSellHardwareForThe)[esi] ; "We don't sell hardware
for the moment..."...
  push  eax          ; format
  call  _printf
  pop   ecx
```

```
not_pc:
  mov   dl, byte ptr [ebx+software_t.info]
  and   edx, PC or MAC
  test  dl, MAC
  jz    short not_mac
  lea   ecx, (aMac - aWeDonTSellHardwareForThe)[esi] ; "We don't sell hardware
for the moment.."...
  push  ecx         ; format
  call  _printf
  pop   ecx

not_mac:            ; "We don't sell hardware for the moment.."...
  lea   eax, (asc_40A31B - aWeDonTSellHardwareForThe)[esi]
  push  eax         ; format
  call  _printf
  pop   ecx
  mov   dl, byte ptr [ebx+software_t.info]
  shr   edx, 2
  and   edx, (WINDOWS or DOS or OS_X) >> 2
  test  dl, WINDOWS >> 2
  jz    short not_windows
  lea   ecx, (aWindows - aWeDonTSellHardwareForThe)[esi] ; "We don't sell
hardware for the moment.."...
  push  ecx         ; format
  call  _printf
  pop   ecx

not_windows:
  mov   al, byte ptr [ebx+software_t.info]
  shr   eax, 2
  and   eax, (WINDOWS or DOS or OS_X) >> 2
  test  al, DOS >> 2
  jz    short not_dos
  lea   edx, (aDos - aWeDonTSellHardwareForThe)[esi] ; "We don't sell hardware
for the moment.."...
  push  edx         ; format
  call  _printf
  pop   ecx

not_dos:
  mov   cl, byte ptr [ebx+software_t.info]
  shr   ecx, 2
  and   ecx, (WINDOWS or DOS or OS_X) >> 2
  test  cl, OS_X >> 2
  jz    short not_os_x
  lea   eax, (aOsX - aWeDonTSellHardwareForThe)[esi] ; "We don't sell hardware
for the moment.."...
  push  eax         ; format
  call  _printf
  pop   ecx
```

```
not_os_x:               ; "We don't sell hardware for the moment.."...
  lea   edx, (asc_40A331 - aWeDonTSellHardwareForThe)[esi]
  push  edx             ; format
  call  _printf
  pop   ecx
  mov   cl, byte ptr [ebx+software_t.info]
  shr   ecx, 5
  and   ecx, category >> 5
  dec   ecx
  jz    short DISASSEMBLY
  dec   ecx
  jz    short RECOVERY
  dec   ecx
  jz    short CRYPTOGRAPHY
  jmp   short end
; ---------------------------------------------------------------------------

DISASSEMBLY:            ; "We don't sell hardware for the moment.."...
  lea   eax, (aDisassembly - aWeDonTSellHardwareForThe)[esi]
  push  eax             ; format
  call  _printf
  pop   ecx
  jmp   short end
; ---------------------------------------------------------------------------

RECOVERY:               ; "We don't sell hardware for the moment.."...
  lea   edx, (aRecovery - aWeDonTSellHardwareForThe)[esi]
  push  edx             ; format
  call  _printf
  pop   ecx
  jmp   short end
; ---------------------------------------------------------------------------

CRYPTOGRAPHY:           ; "We don't sell hardware for the moment.."...
  lea   ecx, (aCryptography - aWeDonTSellHardwareForThe)[esi]
  push  ecx             ; format
  call  _printf
  pop   ecx

end:                    ; "We don't sell hardware for the moment.."...
  lea   eax, (asc_40A358 - aWeDonTSellHardwareForThe)[esi]
  push  eax             ; format
  call  _printf
  pop   ecx
  pop   esi
  pop   ebx
  pop   ebp
  retn
print_software endp

; ---------------------------------------------------------------------------
  align 4
```

```
; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦

; Attributes: bp-based frame

; int __cdecl print_product(product_t *product)
print_product proc near ; CODE XREF: main+128p

product= dword ptr  8

  push  ebp
  mov   ebp, esp
  push  ebx
  mov   ebx, [ebp+product]
  push  [ebx+product_t.category] ; product_category
  call  check_product
  pop   ecx
  test  eax, eax
  jnz   short check_product_ok
  xor   eax, eax
  pop   ebx
  pop   ebp
  retn
; ---------------------------------------------------------------------------

check_product_ok:   ; CODE XREF: print_product+12j
  push  [ebx+product_t.id]
  push  offset aProduct04x ; format
  call  _printf
  add   esp, 8
  mov   edx, [ebx+product_t.category]
  sub   edx, 1
  jb    short case_book
  jz    short case_software
  jmp   short default
; ---------------------------------------------------------------------------

case_book:          ; CODE XREF: print_product+2Ej
  add   ebx, product_t.p.book.title
  push  ebx         ; book
  call  print_book
  pop   ecx
  jmp   short default
; ---------------------------------------------------------------------------

case_software:      ; CODE XREF: print_product+30j
  add   ebx, product_t.p.software.info
  push  ebx         ; software
  call  print_software
  pop   ecx

default:            ; CODE XREF: print_product+32j print_product+3Ej
  mov   al, 1
  pop   ebx
  pop   ebp
  retn
print_product endp

; ---------------------------------------------------------------------------
  align 4
```

```
; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦

; Attributes: bp-based frame

; void __cdecl main()
main proc near          ; DATA XREF: .data:0040A0D0o
  push  ebp
  mov   ebp, esp
  push  ebx
  push  esi
  push  edi
  push  offset aCustomers ; format
  call  _printf
  pop   ecx
  mov   ebx, offset customers
  jmp   short loc_401376
; ---------------------------------------------------------------------------

loop_print_customer: ; CODE XREF: main+25j
  push  ebx             ; customer
  call  print_customer
  pop   ecx
  add   ebx, 40

loc_401376:             ; CODE XREF: main+16j
  cmp   [ebx+customer_t.id], 0
  jnz   short loop_print_customer
  push  544             ; size
  call  _malloc
  pop   ecx
  mov   ebx, eax
  mov   [ebx+product_t.id], 1
  xor   eax, eax        ; BOOK
  mov   [ebx+product_t.category], eax
  mov   esi, offset aIdaQuickstartG ; "IDA QuickStart Guide"
  lea   edi, [ebx+product_t.p.book.title]
  mov   ecx, 32
  rep movsd
  mov   dword ptr [ebx+product_t[1].id], 2
  mov   dword ptr [ebx+product_t[1].category], SOFTWARE
  mov   esi, offset softwares.softs
  lea   edi, [ebx+product_t[1].p.software]
  mov   ecx, 9
  rep movsd
  mov   dword ptr [ebx+product_t[2].id], 3
  mov   dword ptr [ebx+product_t[2].category], SOFTWARE
  mov   esi, (offset softwares.softs.info+24h)
  lea   edi, [ebx+product_t[2].p.software]
  mov   ecx, 9
  rep movsd
  mov   dword ptr [ebx+product_t[3].id], 4
  mov   dword ptr [ebx+product_t[3].category], SOFTWARE
  mov   esi, (offset softwares.softs.info+48h)
  lea   edi, [ebx+product_t[3].p.software]
  mov   ecx, 9
  rep movsd
  push  offset aProducts ; format
  call  _printf
  pop   ecx
  xor   esi, esi
```

```
loop_verify_print_product: ; CODE XREF: main+132j
  mov   eax, esi
  shl   eax, 4
  add   eax, esi
  push  [ebx+eax*8+product_t.category] ; product_category
  call  check_product
  pop   ecx
  test  eax, eax
  jnz   short product_is_valid
  push  offset aInvalidProduct ; format
  call  _printf
  pop   ecx
  jmp   short exit
; ---------------------------------------------------------------------------

product_is_valid:   ; CODE XREF: main+E0j
  mov   edx, esi
  shl   edx, 4
  add   edx, esi
  cmp   [ebx+edx*8+product_t.category], SOFTWARE
  jnz   short print_product
  mov   ecx, esi
  shl   ecx, 4
  add   ecx, esi
  push  [ebx+ecx*8+product_t.p.software.info] ; software_info
  call  check_software
  pop   ecx
  test  eax, eax
  jnz   short print_product
  push  offset aInvalidSoftwar ; format
  call  _printf
  pop   ecx
  jmp   short exit
; ---------------------------------------------------------------------------

print_product:      ; CODE XREF: main+FBj main+110j
  imul  eax, esi, 88h
  add   eax, ebx
  push  eax         ; product
  call  print_product
  pop   ecx
  inc   esi
  cmp   esi, 4
  jl    short loop_verify_print_product

exit:               ; CODE XREF: main+EDj main+11Dj
  push  ebx         ; block
  call  _free
  pop   ecx
  pop   edi
  pop   esi
  pop   ebx
  pop   ebp
  retn
main endp
```

```
; [COLLAPSED AREA .text2. PRESS KEYPAD "+" TO EXPAND]
; Section 2. (virtual address 0000A000)
; Virtual size                  : 00003000 (  12288.)
; Section size in file          : 00002800 (  10240.)
; Offset to raw data for section: 00009400
; Flags C0000040: Data Readable Writable
; Alignment     : 16 bytes ?
; ---------------------------------------------------------------------------

; Segment type: Pure data
; Segment permissions: Read/Write
_data segment para public 'DATA' use32
  assume cs:_data
  ;org 40A000h
; [COLLAPSED AREA .data1. PRESS KEYPAD "+" TO EXPAND]
customers customer_t <1, 'Peter', 'm'> ; DATA XREF: main+11o
  customer_t <2, 'John', 'm'>
  customer_t <3, 'Mary', 'f'>
  customer_t  <0>
aIdaQuickstartG db 'IDA QuickStart Guide',0 ; DATA XREF: main+3Fo
  db 6Bh dup(0)
softwares dd 3              ; count ; DATA XREF: main+62o
  dd PC or WINDOWS or DOS or DISASSEMBLY; softs.info
  db 'IDA Pro',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; softs.name
  dd PC or MAC or WINDOWS or OS_X or RECOVERY; softs.info
  db 'PhotoRescue',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; softs.name
  dd PC or WINDOWS or CRYPTOGRAPHY; softs.info
  db 'aCrypt',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; softs.name
aWeDonTSellHardwareForThe db 'We don',27h,'t sell hardware for the
moment...',0Ah,0
                        ; DATA XREF: check_product+Eo print_software+8o ...
aCustomer04xSC db 'CUSTOMER %04X: %s (%c)',0Ah,0 ; DATA XREF: print_customer+11o
aBookS db 'BOOK: %s',0Ah,0 ; DATA XREF: print_book+6o
aSoftwareS db 'SOFTWARE: %s:',0 ; DATA XREF: print_software+11r
aPc db ' PC',0      ; DATA XREF: print_software+27r
aMac db ' MAC',0    ; DATA XREF: print_software+3Br
asc_40A31B db ';',0 ; DATA XREF: print_software:not_macr
aWindows db ' WINDOWS',0 ; DATA XREF: print_software+5Cr
aDos db ' DOS',0    ; DATA XREF: print_software+72r
aOsX db ' OS-X',0   ; DATA XREF: print_software+89r
asc_40A331 db ';',0 ; DATA XREF: print_software:not_os_xr
aDisassembly db ' DISASSEMBLY',0 ; DATA XREF: print_software:DISASSEMBLYr
aRecovery db ' RECOVERY',0 ; DATA XREF: print_software:RECOVERYr
aCryptography db ' CRYPTOGRAPHY',0 ; DATA XREF: print_software:CRYPTOGRAPHYr
asc_40A358 db 0Ah,0 ; DATA XREF: print_software:endr
aProduct04x db 'PRODUCT %04X: ',0 ; DATA XREF: print_product+1Bo
aCustomers db 'CUSTOMERS:',0Ah,0 ; DATA XREF: main+6o
aProducts db 0Ah    ; DATA XREF: main+C0o
  db 'PRODUCTS:',0Ah,0
aInvalidProduct db 'Invalid product !!!',0Ah,0 ; DATA XREF: main+E2o
aInvalidSoftwar db 'Invalid software !!!',0Ah,0 ; DATA XREF: main+112o
; [COLLAPSED AREA .data2. PRESS KEYPAD "+" TO EXPAND]
; ---------------------------------------------------------------------------

; [00001000 BYTES: COLLAPSED SEGMENT _tls. PRESS KEYPAD "+" TO EXPAND]
; ---------------------------------------------------------------------------

; [00001000 BYTES: COLLAPSED SEGMENT _rdata. PRESS KEYPAD "+" TO EXPAND]
; ---------------------------------------------------------------------------

; [000000C4 BYTES: COLLAPSED SEGMENT _idata. PRESS KEYPAD "+" TO EXPAND]
```

Revision 1.1

DataRescue SA/NV
40 Bld Piercot
4000 Liège, Belgium
T: +32-4-3446510 F: +32-4-3446514