Igor's tip of the week

season one

from 07/08/2020 to 13/08/2021

hex-rays

Today, Hex-Rays is excited to launch a special blog series where Igor, one of the experts behind IDA, will provide useful tips and functionalities of IDA that are not always known or less obvious to its users.

The first episode of this blog series covers the most useful keyboard shortcuts that will certainly speed up your IDA experience.

So, we hope you enjoy this first post and tune in every Friday to read Igor's tip of the week!

Posted the 7th August 2020 by Igor Skochinsky

Usage: basic and advanced usage of IDA features

- **#01:** Lesser-known keyboard shortcuts in IDA
- **#03:** Selection in IDA
- **#04:** More selection!
- **#05:** Highlight
- **#09:** Reanalysis
- **#13:** String literals and custom encodings
- **#14:** Comments in IDA
- **#15:** Comments in structures and enums
- **#28:** Functions list
- **#30:** Quick views
- **#31:** Hiding and Collapsing
- **#34:** Dummy names
- **#35:** Demangled names
- **#36:** Working with list views in IDA
- **#37:** Patching
- **#46:** Disassembly operand representation
- **#47:** Hints in IDA

Navigation: moving around the database

- **#16:** Cross-references
- **#17:** Cross-references 2
- **#20:** Going places
- **#23:** Graph view
- **#38:** Hex view
- **#48:** Searching in IDA
- **#49:** Navigation band
- **#50:** Execution flow arrows

Types: working with types

- **#10:** Working with arrays
- **#11:** Quickly creating structures
- **#12:** Creating structures with known size
- **#51:** Custom calling conventions
- **#52:** Special type attributes

Hidden: hidden gems, not widely known but useful functionality

- **#06:** IDA Release notes
- **#19:** Function calls
- **#21:** Calculator and expression evaluation feature in IDA
- **#24:** Renaming registers
- **#39:** Export Data
- **#41:** Binary file loader
- **#44:** Hex dump loader

Decompiler: related to the Hex-Rays decompiler

- **#18:** Decompiler and global cross-references
- **#27:** Fixing the stack pointer
- **#40:** Decompiler basics
- **#42:** Renaming and retyping in the decompiler
- **#43:** Annotating the decompiler output
- **#45:** Decompiler types

Automation: automating repetitive tasks

- **#07:** IDA command-line options cheatsheet
- **#08:** Batch mode under the hood
- **#32:** Running scripts

Customization: customizing IDA UI to better suit your workflow

- **#02:** IDA UI actions and where to find them
- **#22:** IDA desktop layouts
- **#25:** Disassembly options
- **#26:** Disassembly options 2
- **#29:** Color up your IDA
- **#33:** IDA's user directory (IDAUSR)

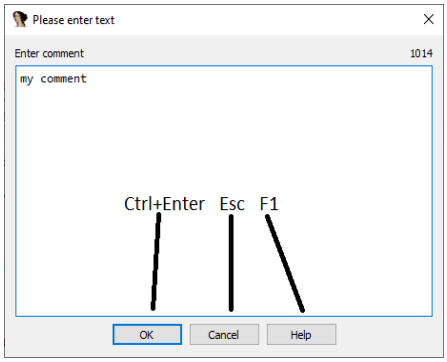
#01: Lesser-known keyboard shortcuts in IDA

07 Aug 2020

https://hex-rays.com/blog/igor-tip-of-the-week-01-lesser-known-keyboard-shortcuts-in-ida/

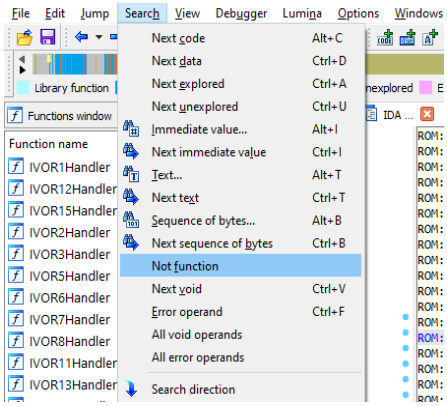
This week's tip will be about using the keyboard in IDA. Nowadays, while most actions can be carried out using the mouse, it can still be much faster and more efficient to use the keyboard. IDA first started as a DOS program, long before GUI and mouse became common, which is why you can still do most of the work without touching the mouse! While most of common shortcuts can be found in the cheat sheet (HTML[1], PDF[2]), there remains some which are less obvious, but incredibly useful!

Text input dialog boxes (e.g. Enter Comment or Edit Local Type)



You can use `Ctrl-Enter` to confirm (OK) or `Esc` to dismiss (Cancel) the dialog. This works regardless of the button arrangement (which can differ depending on the platform and/or theme used).

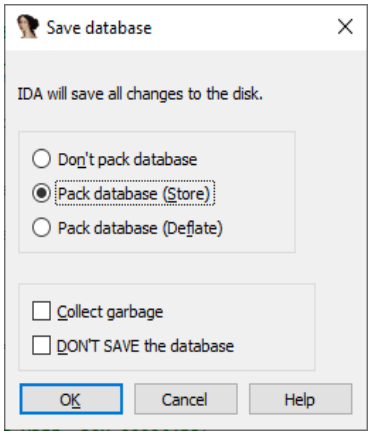
Quick menu navigation



If you hold down `Alt` on Windows (or enable a system option), you should see underlines under the menu item names.

You can press the underlined letter (also known as "accelerator") while holding down `Alt` to open that menu, and then press the underlined letter of the specific menu item to trigger it. The second step will work even if you release `Alt` . For example, to execute "Search > Not function" (which has no default hotkey), you can press `Alt-H, F` . Although there may be no underlines on Linux or Mac, the same key sequence should still work. If you don't have access to a Windows IDA and don't want to brute-force accelerator keys manually, you can check the `cfg/idagui.cfg` file which describes IDA's default menu layout and all assigned accelerators (prefixed with &).

Dialog box navigation



In addition to OK/Cancel buttons, many of IDA's dialog boxes have checkboxes, radio buttons or edit fields. You can use the standard `Tab` key to navigate between them and `Space` bar to toggle, however, similarly to the menus, most dialog box controls in IDA have accelerator shortcuts. You can use `Alt` on Windows to reveal them but, unlike menus, they work even *without* `Alt`. For example. to quickly exit IDA discarding any changes made since opening the database, use this key sequence:

`Alt-X` (or `Alt-F4`) to show the "Save database" dialog
`D` to toggle the "DON'T SAVE the database" checkbox
`Enter` or `Alt-K` (or `K`) to confirm (OK)

NOTE: a few dialogs are excluded from this feature, for example the Options-General... dialog, also Script Command (`Shift-F2`) or other dialogs with a text edit box. In such dialogs you have to hold down Alt to use accelerators.

[1] https://hex-rays.com/wp-content/static/products/ida/idapro_cheatsheet.html
[2] https://hex-rays.com/wp-content/static/products/ida/support/freefiles/IDA_Pro_Shortcuts.pdf

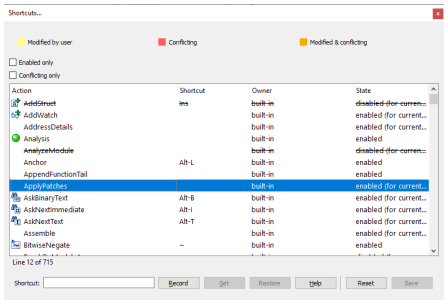
#02: IDA UI actions and where to find them

14 Aug 2020

https://hex-rays.com/blog/igor-tip-of-the-week-02-ida-ui-actions-and-where-to-find-them/

In the previous post we described how to quickly invoke some of IDA's commands using the keyboard. However, sometimes you may need to perform a specific action many times and if it doesn't have a default hotkey assigned it can be tedious to click through the menus. Even the accelerator keys help only so much. Or it may be difficult to discover or find a specific action in the first place (some actions do not even have menu items). There are **two IDA features** that would help here:

The shortcut editor

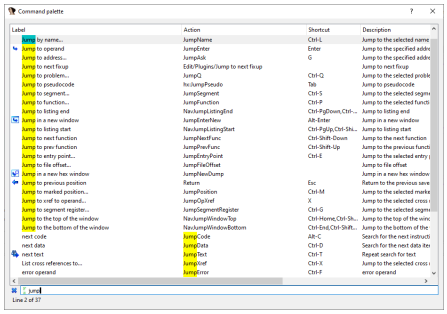


The editor is invoked via Options > Shortcuts... and allows you to see, add, and modify shortcuts for almost all UI actions available in IDA.

The dialog is non-modal and shows which actions are available for the current view (currently disabled ones are struck out) so you can try clicking around IDA and see how the set of available actions changes depending on the context.

To assign a shortcut, select the action in the list then type the key combination in the "Shortcut:" field (on Windows you can also click the "Record" button and press the desired shortcut), then click "Set" to save the new shortcut for this and all future IDA sessions. Use "Restore" to restore just this action, or "Reset" to reset all actions to the default state (as described in `idagui.cfg`).

The command palette



Command palette (default shortcut is `Ctrl-Shift-P`) is similar to the Shortcut editor in that it shows the list of all IDA actions but instead of changing shortcuts you can simply invoke the action.

The filter box at the bottom filters the actions that contain the typed text with fuzzy matching and is focused when the palette is opened so you can just type the approximate name of an action and press `Enter` to invoke the best match.

#03: Selection in IDA

21 Aug 2020

https://hex-rays.com/blog/igor-tip-of-the-week-03-selection-in-ida/

This week's post is about selecting items in IDA and what you can do with the selection.

As a small change from the previous posts with mainly keyboard usage, we'll also use the mouse this time!

Actions and what they are applied to

When an action is performed in IDA, by default it is applied only to the item under the cursor or to the current address (depending on the action). However, sometimes you might want to perform the action on more items or to an address range, for example to:

- undefine a range of instructions;
- convert a range of undefined bytes to a string literal if IDA can't do it automatically (e.g. string is not null-terminated);
- create a function from a range of instructions with some data in the middle (e.g. when you get the dreaded "The function has unde fined instruction/data at the specified address" error);
- export disassembly or decompilation of only selected functions instead of the whole file;
- copy to clipboard a selected fragment of the disassembly.

Selecting in IDA

The simplest ways to select something in IDA are the same as in any text editor:

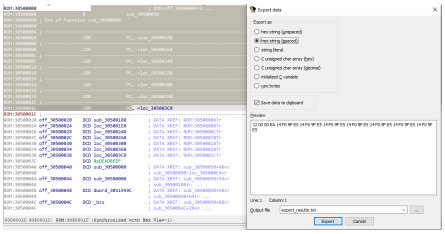
- click and drag with the mouse (you can also scroll with the wheel while keeping the left button pressed);
- hold down `Shift` and use the cursor navigation keys (← ↑ → ↓ `PgUp` `PgDn` `Home` `End` etc.).

However, this can quickly become tiring if you need to select a huge block of the listing (e.g.several screenfuls). In that case, the anchor selection will be of great use.

Using the anchor selection

1. Move to the start of the intended selection and select Edit > Begin selection (or use the `Alt-L` shortcut).
2. Navigate to the other end of the selection using any means (cursor keys, Jump actions, Functions window, Navigation bar etc.).
3. Perform the action (via context menu, keyboard shortcut, or global menu). It will be applied to the selection from the anchor point to the current position.

Using the anchor selection



Some of the actions that use selection:

- Commands in the File > Produce file submenu (create .ASM, .LST, HTML or .C file)
- Edit > Export data (`Shift-E`)

Some more complicated actions requiring selection will be discussed in the forthcoming posts. Stay tuned and see you next Friday!

#04: More selection!

28 Aug 2020

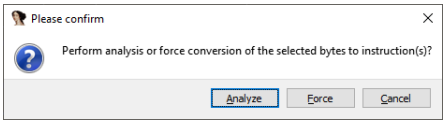
https://hex-rays.com/blog/igor-tip-of-the-week-04-more-selection/

In the previous post we talked about the basic usage of selection in IDA. This week we'll describe a few more examples of actions affected by selection.

Firmware/raw binary analysis

When disassembling a raw binary, IDA is not always able to detect code fragments and you may have to resort to trial & error for finding the code among the whole loaded range which can be a time-consuming process. In such situation the following simple approach may work for initial reconnaissance:

1. Go to the start of the database (`Ctrl-PgUp`);
2. Start selection (`Alt-L`);
3. Go to the end (`Ctrl-PgDn`). You can also go to a specific point that you think may be the end of code region (e.g. just before a big chunk of zeroes or FF bytes);
4. Select Edit > Code or press `C`. You'll get a dialog asking what specific action to perform:



5. Click "Force" if you're certain there are mostly instructions in the selected range, or "Analyze" if there may be data between instructions.
6. IDA will go through the selected range and try to convert any undefined bytes to instructions. If there is indeed valid code in the selected area, you might see fu nctions being added to the Functions window (probably including some false positives).

Structure offsets

Another useful application of selection is applying structure offsets to multiple instructions. For example, let's consider this function from a UEFI module:

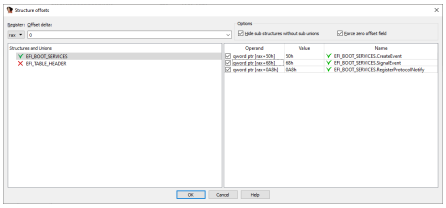
```
.text:0000000000001A64 sub_1A64        proc near               ; CODE XREF: sub_15A4+EB↑p
.text:0000000000001A64                                         ; sub_15A4+10E↑p
.text:0000000000001A64
.text:0000000000001A64 var_28          = qword ptr -28h
.text:0000000000001A64 var_18          = qword ptr -18h
.text:0000000000001A64 arg_20          = qword ptr  28h
.text:0000000000001A64
.text:0000000000001A64                 push    rbx
.text:0000000000001A66                 sub     rsp, 40h
.text:0000000000001A6A                 lea     rax, [rsp+48h+var_18]
.text:0000000000001A6F                 xor     r9d, r9d
.text:0000000000001A72                 mov     rbx, rcx
.text:0000000000001A75                 mov     [rsp+48h+var_28], rax
.text:0000000000001A7A                 mov     rax, cs:gBS
.text:0000000000001A81                 lea     edx, [r9+8]
.text:0000000000001A85                 mov     ecx, 200h
.text:0000000000001A8A                 call    qword ptr [rax+50h]
.text:0000000000001A8D                 mov     rax, cs:gBS
.text:0000000000001A94                 mov     r8, [rsp+48h+arg_20]
.text:0000000000001A99                 mov     rdx, [rsp+48h+var_18]
.text:0000000000001A9E                 mov     rcx, rbx
.text:0000000000001AA1                 call    qword ptr [rax+0A8h]
.text:0000000000001AA7                 mov     rax, cs:gBS
.text:0000000000001AAE                 mov     rcx, [rsp+48h+var_18]
.text:0000000000001AB3                 call    qword ptr [rax+68h]
.text:0000000000001AB6                 mov     rax, [rsp+48h+var_18]
.text:0000000000001ABB                 add     rsp, 40h
.text:0000000000001ABF                 pop     rbx
.text:0000000000001AC0                 retn
.text:0000000000001AC0 sub_1A64        endp
```

#04: More selection!

28 Aug 2020

https://hex-rays.com/blog/igor-tip-of-the-week-04-more-selection/

If we know that `gBS` is a pointer to `EFI_BOOT_SERVICES`, we can convert accesses to it (in the call instructions) to structure offsets. It can be done for each access manually but is tedious. In such situation the selection can be helpful. If we select the instructions accessing the structure and press `T` (structure offset), a new dialog pops up:



You can select which register is used as the base, which structure to apply and even select which specific instructions you want to convert.

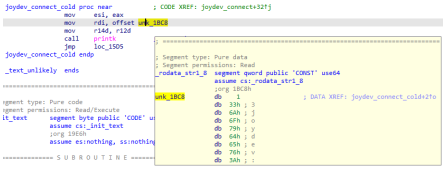
After selecting `rax` and `EFI_BOOT_SERVICES`, we get a nice-looking listing:

```
.text:0000000000001A64 sub_1A64        proc near               ; CODE XREF: sub_15A4+EB↑p
.text:0000000000001A64                                         ; sub_15A4+10E↑p
.text:0000000000001A64
.text:0000000000001A64 Event           = qword ptr -28h
.text:0000000000001A64 var_18          = qword ptr -18h
.text:0000000000001A64 Registration    = qword ptr  28h
.text:0000000000001A64
.text:0000000000001A64                 push    rbx
.text:0000000000001A66                 sub     rsp, 40h
.text:0000000000001A6A                 lea     rax, [rsp+48h+var_18]
.text:0000000000001A6F                 xor     r9d, r9d        ; NotifyContext
.text:0000000000001A72                 mov     rbx, rcx
.text:0000000000001A75                 mov     [rsp+48h+Event], rax ; Event
.text:0000000000001A7A                 mov     rax, cs:gBS
.text:0000000000001A81                 lea     edx, [r9+8]     ; NotifyTpl
.text:0000000000001A85                 mov     ecx, 200h       ; Type
.text:0000000000001A8A                 call    [rax+EFI_BOOT_SERVICES.CreateEvent]
.text:0000000000001A8D                 mov     rax, cs:gBS
.text:0000000000001A94                 mov     r8, [rsp+48h+Registration] ; Registration
.text:0000000000001A99                 mov     rdx, [rsp+48h+var_18] ; Event
.text:0000000000001A9E                 mov     rcx, rbx        ; Protocol
.text:0000000000001AA1                 call    [rax+EFI_BOOT_SERVICES.RegisterProtocolNotify]
.text:0000000000001AA7                 mov     rax, cs:gBS
.text:0000000000001AAE                 mov     rcx, [rsp+48h+var_18] ; Event
.text:0000000000001AB3                 call    [rax+EFI_BOOT_SERVICES.SignalEvent]
.text:0000000000001AB6                 mov     rax, [rsp+48h+var_18]
.text:0000000000001ABB                 add     rsp, 40h
.text:0000000000001ABF                 pop     rbx
.text:0000000000001AC0                 retn
.text:0000000000001AC0 sub_1A64        endp
```

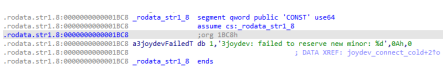
Forced string literals

When some code is referencing a string, IDA is usually smart enough to detect it and convert referenced bytes to a literal item. However, in some cases the automatic conversion does not work, for example:

- string contains non-ASCII characters
- string is not null-terminated



A common example of the former is Linux kernel which uses a special byte sequence to mark different categories of kernel messages. For example, consider this function from the `joydev.ko` module:



IDA did not automatically create a string at 1BC8 because it starts with a non-ASCII character. However, if we select the string's bytes and press `A` (Convert to string), a string is created anyway:

#04: More selection!

28 Aug 2020

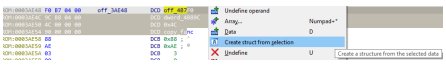
https://hex-rays.com/blog/igor-tip-of-the-week-04-more-selection/

Creating structures from data

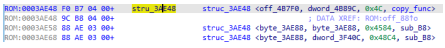
This action is useful when dealing with structured data in binaries. Let's consider a table with approximately this layout of entries:

```
struct copyentry {
 void *source;
 void *dest;
 int size;
 void* copyfunc;
};
```

While such a structure can always be created manually in the Structures window, often it's easier to format the data first then create a structure which describes it. After creating the four data items, select them and from the context menu, choose "Create struct from selection":



IDA will create a structure representing the selected data items which can then be used to format other entries in the program or in disassembly to better understand the code working with this data.



#05: Highlight

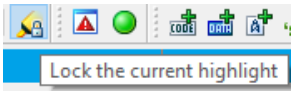
04 Sep 2020

https://hex-rays.com/blog/igor-tip-of-the-week-05-highlight/

In IDA, **highlight** is the dynamic coloring of a word or number under the cursor as well as all matching substrings on the screen. In the default color scheme, a yellow background color is used for the highlight.

Highlight is updated when you click on a non-whitespace location in the listing or move the cursor with the arrow keys. Highlight is not updated (remains the same) when:

- moving the cursor with `PgUp`, `PgDn`, `Home`, `End`;
- scrolling the listing with mouse wheel or scroll bar;
- using Jump commands or clicking in the navigation band (unless the cursor happens to land on a word at the new location);
- highlight is locked by the LockHighlight action (it is one of the handful of actions which are only available as a toolbar button by default).



Register highlight

```
movzx   eax, word ptr [r10+r8*2]

shr     ax, 3
and     eax, 1
cmp     cl, 33h ; '3'
ja      short loc_274BD
mov     r14, rbx
shr     r14, cl
mov     rcx, r14
and     ecx, 1
or      eax, ecx

                        ; CODE XREF:
lea     ecx, [r9+1]
test    al, al
jnz     short loc_27480
cmp     dl, 2Fh ; '/'
jnz     short loc_2748B
cmp     ecx, 3
jz      short loc_27520
add     rsi, 1
mov     byte ptr [rdi], 2Fh ; '/'
lea     eax, [r9+2]
add     rdi, 1
movzx   edx, byte ptr [rsi]
mov     r9d, ecx
mov     ecx, eax
test    dl, dl
```

For some processors, highlighted registers are treated in a special way: not only is the same register highlighted but also any register which contains it or is a part of it. For example, on x86_x64, if `ax` is selected, then `al`, `ah`, `eax` and `rax` get highlighted too.

Manual highlight

```
var_C0          = qword ptr -0C0h
var_B4          = dword ptr -0B4h
var_B0          = qword ptr -0B0h
var_A8          = byte ptr -0A8h
var_60          = qword ptr -60h
var_58          = qword ptr -58h
stack_end       = qword ptr  8

; __unwind {
                endbr64
                push    r15
                xor     eax, eax
                push    r14
                push    r13
                push    r12
                push    rbp
                push    rbx
                mov     rbx, rcx
                sub     rsp, 98h

                mov     [rsp+0C8h+var_C0], rdx

                mov     rdx, cs:_dl_starting_up_ptr

                mov     [rsp+0C8h+var_B0], rdi

                mov     rdi, r9         ; lpfunc
                mov     [rsp+0C8h+var_B4], esi
```

In addition to the automatic highlight by clicking on a word/number, you can also select an arbitrary substring using mouse or keyboard and it will be used to highlight all matching sequences on the screen. For manual highlight, only exactly matching substrings are highlighted – there is no special handling for the registers.

Manual highlight

You can quickly jump between highlighted matches using `Alt-Up` and `Alt-Down`. This works even if the closest match is not on screen – IDA will look for next match in the selected direction.

Highlight is available not only in the disassembly listing but in most text-based IDA subviews: Pseudocode, Hex View, Structures and Enums.

#06: IDA Release notes

11 Sep 2020

https://hex-rays.com/blog/igor-tip-of-the-week-06-release-notes/

With every IDA release[1], we publish detailed release notes describing various new features, improvements and bugfixes. While some of the additions are highlighted and therefore quite visible, others are not so obvious and may require careful reading. Having a closer look at these release notes, you will be surprised to see many small but useful features added through different IDA versions.

A couple of good examples can be:

Text input dialog boxes (e.g. Enter Comment or Edit Local Type)

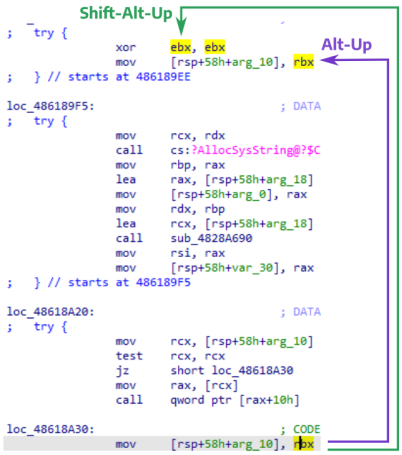
Added in IDA 7.5, these actions allow you to quickly jump between various uses of a register.

> • UI: added actions to search for register definition or register use (Shift+Alt+Up, Shift+Alt+Down)
> From: What's new in IDA 7.5[2]

`Shift-Alt-Up` : find the previous location where the selected register is **defined** (written to).

`Shift-Alt-Down` : find the next location where the selected register is **used** (read from or partially overwritten).

These actions are especially useful in big functions compiled with high optimization level where the distance between definition and use can be quite big so tracking registers visually using standard highlight[3] is not always feasible.



In the above screenshot, you can see that `Alt-Up` jumps to the closest highlight substring match while `Shift-Alt-Up` finds where `rbx` was changed (`ebx` is the low part of `rbx` so the `xor` instruction changes `rbx`).

These actions are currently implemented for a limited number of processors (x86/x64, ARM, MIPS), but may be extended to others if we get more requests.

Jump to previous or next function

> + ui: added shortcuts Ctrl+Shift+Up/Ctrl+Shift+Down to jump to the start of the previous/next function
> From: What's new in IDA 7.2[4]

Added in IDA 7.2, these are minor but very useful shortcuts, especially in large binaries with many big functions.

By the way, if standard shortcuts are tricky to use, you can always set custom ones using a key combination you prefer.

[1] https://hex-rays.com/products/ida/news/
[2] https://hex-rays.com/products/ida/news/7_5/
[3] https://hex-rays.com/products/ida/news/7_2/
[4] https://hex-rays.com/blog/igor-tip-of-the-week-02-ida-ui-actions-and-where-to-find-them/

#07: IDA command-line options cheatsheet

18 Sep 2020

https://hex-rays.com/blog/igor-tip-of-the-week-07-ida-command-line-options-cheatsheet/

Most IDA users probably run IDA as a stand-alone application and use the UI to configure various options. However, it is possible to pass command-line options to it to automate some parts of the process. The full set of options[1] is quite long so we'll cover the more common and useful ones.

ℹ In the examples below, `ida` can be replaced by `ida64` for 64-bit files, or `idat` (`idat64`) for console (text-mode) UI.

Simply open a file in IDA

```
ida <filename>
```

`<filename>` can be a new file that you want to disassemble or an existing database. This usage is basically the same as using File > Open or dropping the file onto IDA's icon. You still need to manually confirm the options in the Load File dialog or any other prompts that IDA displays, but the initial splash screen is skipped.

ℹ If you use any additional command-line options, make sure to put them before the filename or they'll be ignored.

Open a file and auto-select a loader

```
ida -T<prefix> <filename>
```

Where `<prefix>` is a unique prefix of the loader description shown in the Load file dialog. For example, when loading a .NET executable, IDA proposes the following options:

• Microsoft.Net assembly
• Portable executable for AMD64 (PE)
• MS-DOS executable (EXE)
• Binary file

For each of them, the corresponding-T option could be:

- `-TMicrosoft`
- `-TPortable`
- `-TMS`
- `-TBinary`

When the prefix contains a space, use quotes. For example, to load the first slice from a fat Mach-O file:

```
ida "-TFat Mach-O File, 1" file.macho
```

In case of archive formats like ZIP, you can specify the archive member to load after a colon (and additional loader names nested as needed). For example, to load the main dex file from an .apk (which is a zip file):

```
ida -TZIP:classes.dex:Android file.apk
```

However, it is usually better to pick the APK loader at the top level (especially in the case of multi-dex files)

```
ida -TAPK file.apk
```

When `-T` is specified, the initial load dialog is skipped and IDA proceeds directly to loading the file using the specified loader (but any additional prompts may still be shown).

Auto-accept any prompts, informational messages or warnings

Sometimes you just want to load the file and simply accept all default settings. In such case you can use the -A switch:

```
ida -A <filename>
```

This will load the file using autonomous, or batch, mode, where IDA will not display any dialog but accept the default answer in all cases.

[1] https://hex-rays.com/products/ida/support/idadoc/417.shtml.html

18 Sep 2020
https://hex-rays.com/blog/igor-tip-of-the-week-07-ida-command-line-options-cheatsheet/

In this mode **no** interactive dialogs will show up after loading is finished (e.g not even “Rename” or “Add comment”). To restore interactivity, execute `batch(0)`[2] statement in the IDC or Python console at the bottom of IDA’s window.

Batch disassembly

This is an extension of the previous section and is invoked using the -B switch:

```
ida -B <filename>
```

IDA will load the file using all default options, wait for the end of auto-analysis, output the disassembly to `<filename>.asm` and exit after saving the database.

Binary file options

When loading raw binary files, IDA cannot use any of the metadata that is present in higher-level file formats like ELF, PE or Mach-O. In particular, the *processor type* and *loading* address cannot be deduced from the file and have to be provided by the user. To speed up your workflow, you can specify them on the command line:

```
ida -p<processor> -B<base> <filename>
```

`<processor>` is one of the processor types[3] supported by IDA. Some processors also support options after a colon.

`<base>` is the hexadecimal load base in paragraphs (16-byte quantities). In practice, it means that you should remove the last zero from the full address.

For example, to load a big-endian MIPS firmware at linear address 0xBFC00000:

```
ida -pmipsb -bBFC0000 firmware.bin
```

A Cortex-M3 firmware mapped at 0x4000:

```
ida -parm:ARMv7-M -b400 firmware.bin
```

Logging

When IDA is running autonomously, you may miss the messages that are usually printed in the Output window but they may contain important informational messages, errors, or warnings. To keep a copy of the messages you can use the `-L` switch:

```
ida -B -Lida_batch.log <filename>
```

[2] https://hex-rays.com/products/ida/support/idadoc/287.shtml
[3] https://hex-rays.com/products/ida/support/idadoc/618.shtml

#08: Batch mode under the hood

25 Sep 2020

https://hex-rays.com/blog/igor-tip-of-the-week-08-batch-mode-under-the-hood/

We've briefly covered batch mode last time but the basic functionality is not always enough so let's discuss how to customize it.

Basic usage

To recap, the batch mode can be invoked with this command line:

```
ida -B -Lida.log <other switches> <filename>
```

IDA will load the file, wait for the end of analysis, and write the full disassembly to `<filename>.asm`

How it works

In fact, `-B` is a shorthand for `-A -Sanalysis.idc`:

- `-A`: enable autonomous mode (answer all queries with the default choice).
- `-Sanalysis.idc`: run the script `analysis.idc` after loading the file.

You can find `analysis.idc` in the `idc` subdirectory of IDA install. In IDA 7.5 it looks as follows:

```
static main()
{
        // turn on coagulation of data in the final pass of analysis
        set_inf_attr(INF_AF, get_inf_attr(INF_AF) | AF_DODATA | AF_FINAL);
        // .. and plan the entire address space for the final pass
        auto_mark_range(0, BADADDR, AU_FINAL);
        msg("Waiting for the end of the auto analysis...\n");
        auto_wait();
        msg("\n\n------ Creating the output file.... --------\n");
        auto file = get_idb_path()[0:-4] + ".asm";
        auto fhandle = fopen(file, "w");
        gen_file(OFILE_ASM, fhandle, 0, BADADDR, 0); // create the assembler
        file
        msg("All done, exiting...\n");
        qexit(0); // exit to OS, error code 0 - success
}
```

Thus, to modify the behavior of the batch mode you can:

- Either modify the standard `analysis.idc`
- Or specify a different script using `-S<myscript.idc>`

For example, to output an LST file (it includes address prefixes), change the gen_file[1] call:

```
gen_file(OFILE_LST, fhandle, 0, BADADDR, 0);
```

Batch decompilation

If you have the decompiler[2] for the target file's architecture, you can also run it in batch mode[3]. For example, to decompile the whole file:

```
ida -Ohexrays:outfile.c:ALL -A <filename>
```

To decompile only the function `main`:

```
ida -Ohexrays:outfile.c:main -A <filename>
```

This uses the functionality built-in into the decompiler plugin which works similarly to the `analysis.idc` script (wait for the end of autoanalysis, then decompile the specified functions to `outfile.c`).

[1] https://hex-rays.com/products/ida/support/idadoc/244.shtml
[2] https://hex-rays.com/decompiler/
[3] https://hex-rays.com/products/decompiler/manual/batch.shtml

#08: Batch mode under the hood

25 Sep 2020

https://hex-rays.com/blog/igor-tip-of-the-week-08-batch-mode-under-the-hood/

Customizing batch decompilation

If the default functionality is not enough, you could write a plugin to drive the decompiler via its C++ API[4]. However, for scripting it's probably more convenient to use Python. Similarly to IDC, Python scripts can be used with the `-S` switch to be run automatically after the file is loaded.

A sample script is attached to this post. Use it as follows:

```
ida -A -Sdecompile_entry_points.py -Llogfile.txt <filename>
```

Speeding up batch processing

In the examples so far we've been using the `ida` executable which is the full GUI version of IDA. Even though the UI is not actually displayed in batch mode, it still has to load and initialize all the dependent UI libraries which can take non-negligible time. This is why it is often better to use the text-mode executable (`idat`) which uses lightweight text-mode UI. However, it still needs a terminal even in batch mode. In case you need to run it in a situation without a terminal (e.g. run it in background or from a daemon), you can use the following approach:

1. set environment variable `TVHEADLESS=1`
2. redirect output

For example:

```
TVHEADLESS=1 idat -A -Smyscript.idc file.bin >/dev/null &
```

Downloads
decompile_entry_points.py[5]

[4] https://hex-rays.com/products/decompiler/sdk/
[5] https://hex-rays.com/wp-content/uploads/2020/09/decompile_entry_points.py

#09: Reanalysis

02 Oct 2020

https://hex-rays.com/blog/igor-tip-of-the-week-09-reanalysis/

While working in IDA, sometimes you may need to reanalyze some parts of your database, for example:

- after changing a prototype of an external function (especially calling convention, number of purged bytes, or “Does not return” flag);
- after fixing up incorrectly detected ARM/Thumb or MIPS32/MIPS16 regions;
- after changing global processor options (e.g. setting `$gp` value in MIPS or TOC in PPC);
- other situations (analyzing switches, etc.)

Reanalyzing individual instructions

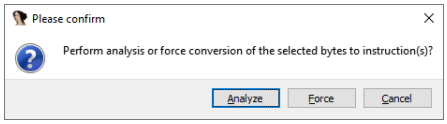
To reanalyze an instruction, position the cursor in it and press `C` (convert to code). Even if the instruction is already code, this action is not a no-op: it asks the IDA kernel to:

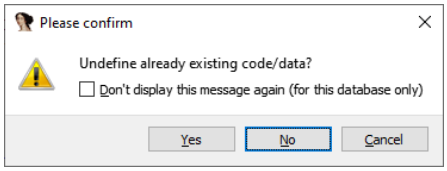
1. delete cross-references from the current address;
2. have the processor module reanalyze the instruction; normally this should result in (re-)creation of cross-references, including the flow cross-reference to the following instruction (unless the current instruction stops the code flow).

Reanalyzing a function

All of the function’s instructions are reanalyzed when any of the function’s parameters are changed (e.g.. in case stack variables need to be recreated). So, the following key sequence causes the whole function to be reanalyzed: `Alt-P`(Edit function), `Enter`(confirm dialog).

Reanalyzing a bigger range of instructions

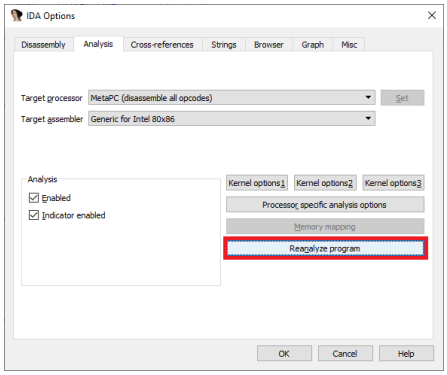


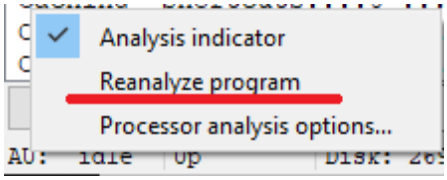


For this we can use the trick covered in the post on selection[1].

1. go to start of the range;
2. press (start selection);
3. go to the end of selection;
4. press (convert to code). Pick “Analyze” in the first prompt and “No” in the second.

Reanalyzing whole database





If you need to reanalyze everything but don’t want to go through the hassle of selecting all the code, there is a dedicated command which can be invoked in two ways:

1. Menu Options > General..., Analysis Tab, Reanalyze program button;
2. Right-click the status bar at the bottom of IDA’s window, Reanalyze program

[1] https://hex-rays.com/blog/igor-tip-of-the-week-04-more-selection/

#10: Working with arrays

09 Oct 2020

https://hex-rays.com/blog/igor-tip-of-the-week-10-working-with-arrays/

Arrays are used in IDA to represent a sequence of multiple items of the same type: basic types (byte, word, dword etc.) or complex ones (e.g. structures).

Creating an array

To create an array:

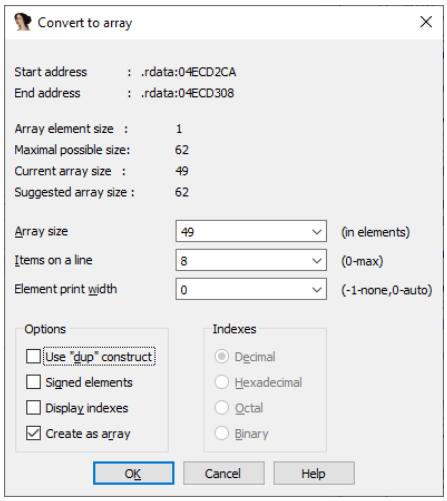
1. Create the first item;
2. Choose “Array...” from the context menu , or press * ;
3. Fill in at least the *Array size* field and click OK.

Step 1 is optional; if no data item exists at the current location, a byte array will be created.

Hint: if you select a range before pressing *, *Array size* will be pre-filled with the number of items which fits into the selected range.

Quick menu navigation

Array parameters affect how the array is displayed in the listing and can be set at the time the array is first created or any time later by pressing *.



- Array size: total number of elements in the array;
- Items on a line: how many items (at most) to print on one line. 0 means to print the maximum number which fits into the disassembly line;
- Element print width: how many characters to use for each element. Together with the previous parameter can be used for formatting arrays into nice-looking tables. For example: 8 items per line, print width **-1**:

```
db 1, 2, 3, 4, 5, 6, 7, 8
db 9, 10, 11, 12, 13, 14, 15, 16
db 17, 18, 19, 20, 21, 22, 23, 24
db 25, 255, 255, 255, 255, 255, 255, 26
db 27, 28, 29, 30, 31, 32, 33, 34
db 35, 36, 37, 38, 39, 40, 41, 42
```

print width **0**:

```
db   1,  2,  3,  4,  5,  6,  7,  8
db   9, 10, 11, 12, 13, 14, 15, 16
db  17, 18, 19, 20, 21, 22, 23, 24
db  25,255,255,255,255,255,255, 26
db  27, 28, 29, 30, 31, 32, 33, 34
db  35, 36, 37, 38, 39, 40, 41, 42
```

print width **5**:

```
db     1,    2,    3,    4,    5,    6,    7,    8
db     9,   10,   11,   12,   13,   14,   15,   16
db    17,   18,   19,   20,   21,   22,   23,   24
db    25,  255,  255,  255,  255,  255,  255,   26
db    27,   28,   29,   30,   31,   32,   33,   34
db    35,   36,   37,   38,   39,   40,   41,   42
```

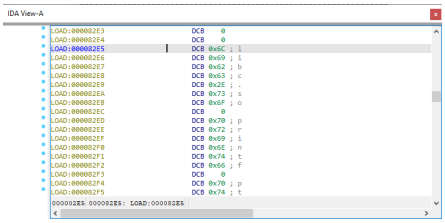
- Use “dup” construct: for assemblers that support it, repeated items with the same value will be collapsed into a dup expression instead of printing each item separately;
  dup off: `db 0FFh, 0FFh, 0FFh, 0FFh, 0FFh, 0FFh`
  dup on: `db 6 dup(0FFh)`
- Signed elements: integer items will be treated as signed numbers;
- Display indexes: for each line, first item’s array index will be printed in a comment.
- Create as array: if unchecked, IDA will convert the array into separate items.

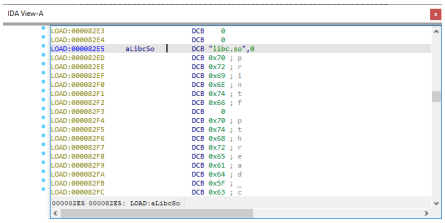
#10: Working with arrays

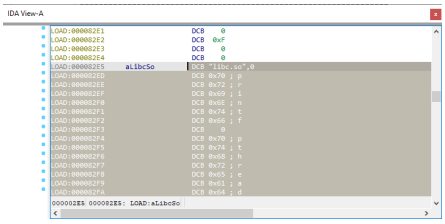
09 Oct 2020

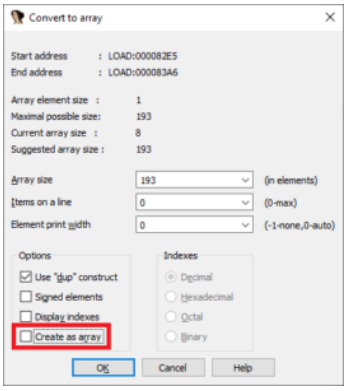
https://hex-rays.com/blog/igor-tip-of-the-week-10-working-with-arrays/

Creating multiple string literals









The last option in array parameters dialog can be useful when dealing with multiple string literals packed together. For example, if we have a string table like this:

First, create one string.

Then, select it and all the following strings using one of the methods described before[1].

Invoke Edit > Array... or press *. The array size will be set to the total length of the selection. In the dialog, **uncheck** “Create as array”. Click OK.

We get a nicely formatted string table!

This approach works also with Unicode (UTF-16) strings.

[1] https://www.hex-rays.com/blog/igor-tip-of-the-week-04-more-selection/

#11: Quickly creating structures

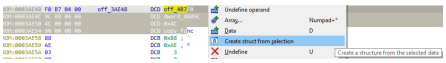
16 Oct 2020

https://hex-rays.com/blog/igor-tip-of-the-week-11-quickly-creating-structures/

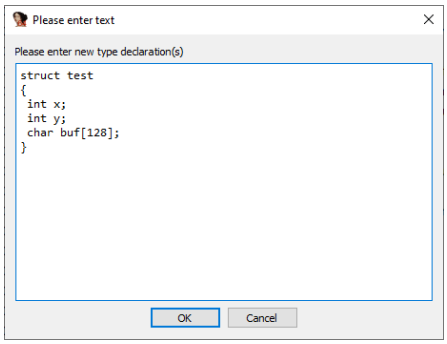
When reverse engineering a big program, you often run into information stored in structures. The standard way of doing it involves using the Structures window and adding fields one by one, similar to the way you format data items in disassembly. But are there other options? Let's look at some of them.

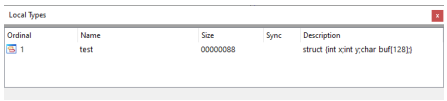
Using already formatted data

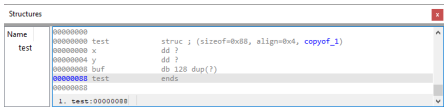
This was mentioned briefly in the post on selection[1] but is worth repeating. If you happen to have some formatted data in your disassembly and want to group it into a structure, just select it and choose "Create struct from selection" in the context menu.

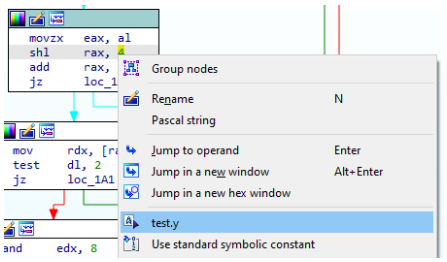


Using Local Types









The Local Types view shows the *high level* or *C level* types used in the database such as structs, enums and typedefs. It is most useful with the decompiler but can still be used for the assembler level types such as Structures and Enums. For example, open the Local Types (`Shift-F1` or View > Open subviews > Local Types), then press `Ins` (or pick Insert.. from the context menu). In the new dialog enter a C syntax structure definition and click OK.

The structure appears in the list but cannot yet be used in disassembly.

To make it available, double-click it and answer "Yes".

Now that a corresponding assembler level type has been created in the Structures view, it can be used in the disassembly.

For more info about using Local Types and two kinds of types check this IDA Help topic[2].

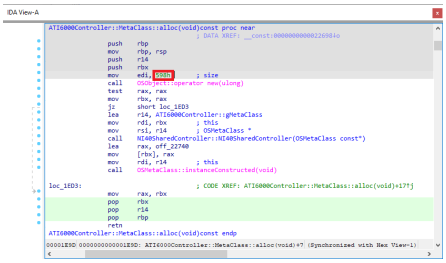
[1] https://hex-rays.com/blog/igor-tip-of-the-week-03-selection-in-ida/
[2] https://www.hex-rays.com/products/ida/support/idadoc/1042.shtml

#12: Creating structures with known size

23 Oct 2020

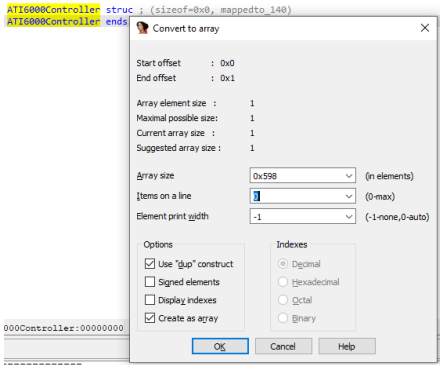
https://hex-rays.com/blog/igor-tip-of-the-week-12-creating-structures-with-known-size/ v

Sometimes you know the structure size but not the actual layout yet. For example, when the size of memory being allocated for the structure is fixed:



In such cases, you can quickly make a dummy structure and then modify it as you analyze code which works with it. There are several approaches which can be used here.

Fixed-size structure 1: single array

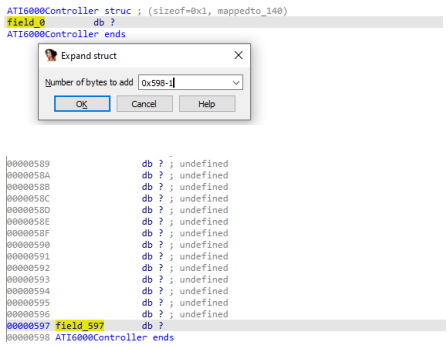


This is the fastest option but makes struct modification a little awkward.

1. create the struct (go to Structures view, press `Ins` and specify a name);
2. create the array (position cursor at the start of the struct, press * and enter the size (decimal or hex)

When you need to create a field in the middle, press * to resize the array so it ends before the field, create the field, then create another array after it to pad the struct to the full size again.

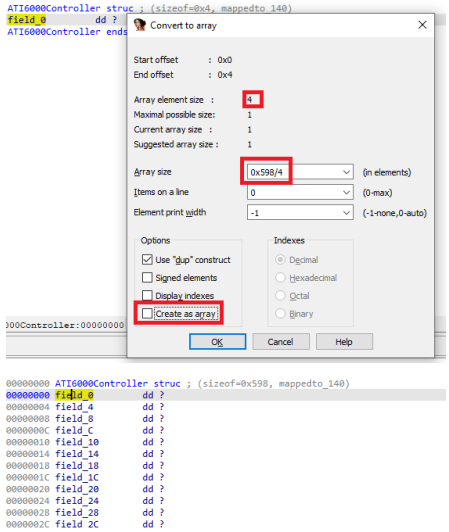
Fixed-size structure 2: big gap in the middle



1. create the struct (go to Structures view, press `Ins` and specify a name);
2. create a byte field (press `D`);
3. add a gap (`Ctrl-E` or “Expand struct type..” in context menu) and enter the size minus 1;
4. (optional but recommended) On `field_0` which is now at the end of the struct, press `N`, `Del`, `Enter`. This will reset the name to match the actual offset and will not hinder creation of another `field_0` at offset 0 if needed.

To create fields in the middle of the gap, go to the specific offset in the struct (`G` can be used for big structs).

Fixed-size structure 3: fill with dummy fields



1. create the struct (go to Structures view, press `Ins` and specify a name);
2. create one dummy field (e.g. a dword);
3. press * and enter the size (divided by the field size if different from byte). Uncheck “Create as array” and click OK.

#12: Creating structures with known size

23 Oct 2020

https://hex-rays.com/blog/igor-tip-of-the-week-12-creating-structures-with-known-size/ v

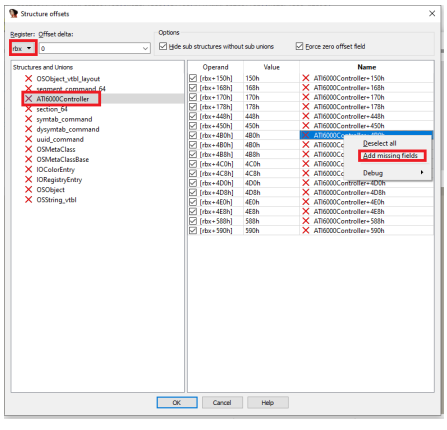
Fixed-size structure 1: single array

Using a structure with a gap in the middle (option 2 above) is especially useful when analyzing functions that work with it using a fixed register base. For example, this function uses `rbx` as the base for the structure:

```
ATI6000Controller::initializeProjectDependentResources(void) proc near
  push    rbp
  mov     rbp, rsp
  push    rbx
  sub     rsp, 8
  mov     rbx, rdi
  lea     rax, `vtable for'NI40SharedController
  mov     rdi, rbx        ; this
  call    qword ptr [rax+0C30h]
  test    eax, eax
  jnz     loc_25CD
  mov     rax, [rbx+168h]
  mov     [rbx+4B8h], rax
  mov     rax, [rbx+178h]
  mov     [rbx+4C0h], rax
  mov     rax, [rbx+150h]
  mov     [rbx+4C8h], rax
  mov     [rbx+4B0h], rbx
  mov     rax, [rbx+448h]
  mov     [rbx+4D0h], rax
  mov     rcx, [rbx+170h]
  mov     [rbx+4D8h], rcx
  mov     rcx, [rax]
  mov     [rbx+4E0h], rcx
  mov     eax, [rax+8]
  mov     [rbx+4E8h], rax
  call    NI40PowerPlayManager::createPowerPlayManager(void)
  mov     [rbx+450h], rax
  test    rax, rax
  jnz     short loc_2585
  mov     eax, 0E00002BDh
  jmp     short loc_25CD
-----------------------------------------------------------------

loc_2585:
  mov     rcx, [rax]
  lea     rsi, [rbx+4B0h]
  ...
```

To automatically create fields for all rbx-based accesses:



1. select all instructions using `rbx`;
2. from context menu, choose “Structure offset” (or press `T`);
3. in the dialog, make sure Register is set to `rbx`, select the created struct (a red cross simply means that it has no fields at the matching offsets currently);
4. from the right pane’s context menu, choose “Add missing fields”.

You can then repeat this for all other functions working with the structure to create other missing fields.

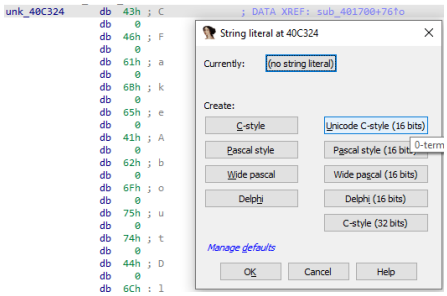
#13: String literals and custom encodings

30 Oct 2020

https://hex-rays.com/blog/igor-tip-of-the-week-13-string-literals-and-custom-encodings/

Most of IDA users probably analyze software that uses English or another Latin-based alphabet. Thus the defaults used for string literals – the OS system encoding on Windows and UTF-8 on Linux or macOS – are usually good enough. However, occasionally you may encounter a program which does use another language.

Unicode strings



In case the program uses wide strings, it is usually enough to use the corresponding "Unicode C-style" option when creating a string literal:

In general, Windows programs tend to use 16-bit wide strings (wchar_t is 16-bit) while Linux and Mac use 32-bit ones (`wchar_t is 32-bit`). That said, exceptions happen and you can use either one depending on a specific binary you're analyzing.

Hint: you can use accelerators to quickly create specific string types, for example **`Alt-A`**, **`U`** for Unicode 16-bits.

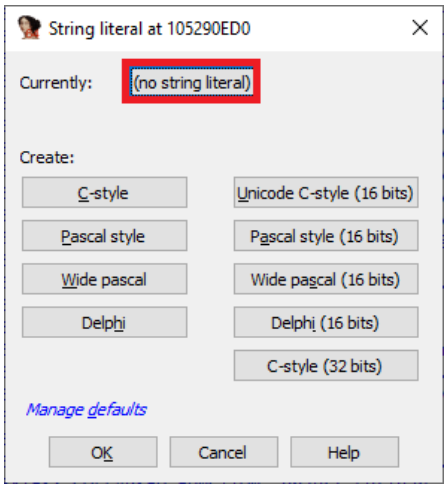
Custom encodings

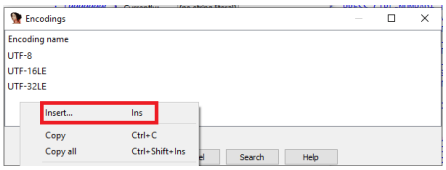
There may be situations when the binary being analyzed uses an encoding different from the one picked by IDA, or even multiple mutually incompatible encodings in the same file. In that case you can set the encoding separately for individual string literals, or globally for all new strings.

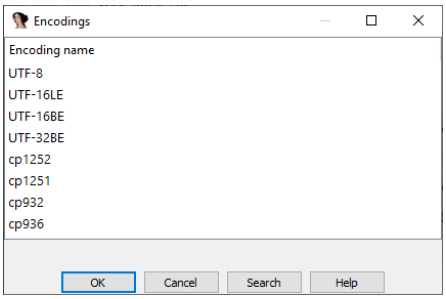
Add a new encoding

To add a custom encoding to the default list (usually UTF-8, UTF-16LE and UTF-32LE):

1. Options > String literals... (**`Alt-A`**);
2. Click the button next to "Currently:";
3. In context menu, "Insert..." (Ins);
4. Specify the encoding name.







For the encoding name you can use:
- Windows codepages (e.g. 866, CP932, windows-1251)
- Well-known charset names (e.g. Shift-JIS, UTF-8, Big5)

On Linux or macOS, run iconv -l to see the available encodings.

Note: some encodings are not supported on all systems so your IDB may become system-specific.

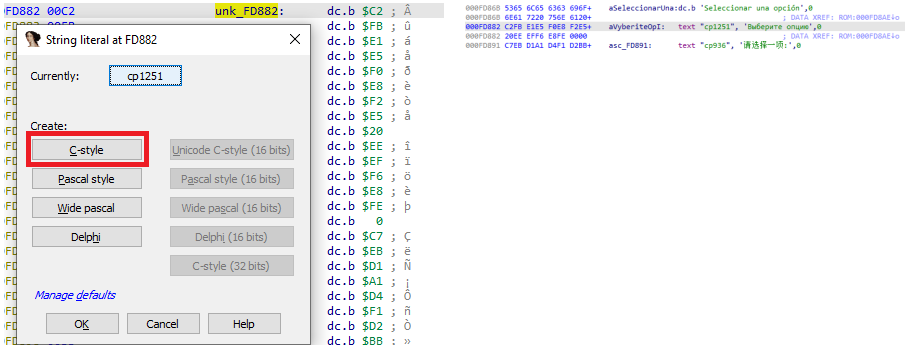
#13: String literals and custom encodings

30 Oct 2020

https://hex-rays.com/blog/igor-tip-of-the-week-13-string-literals-and-custom-encodings/

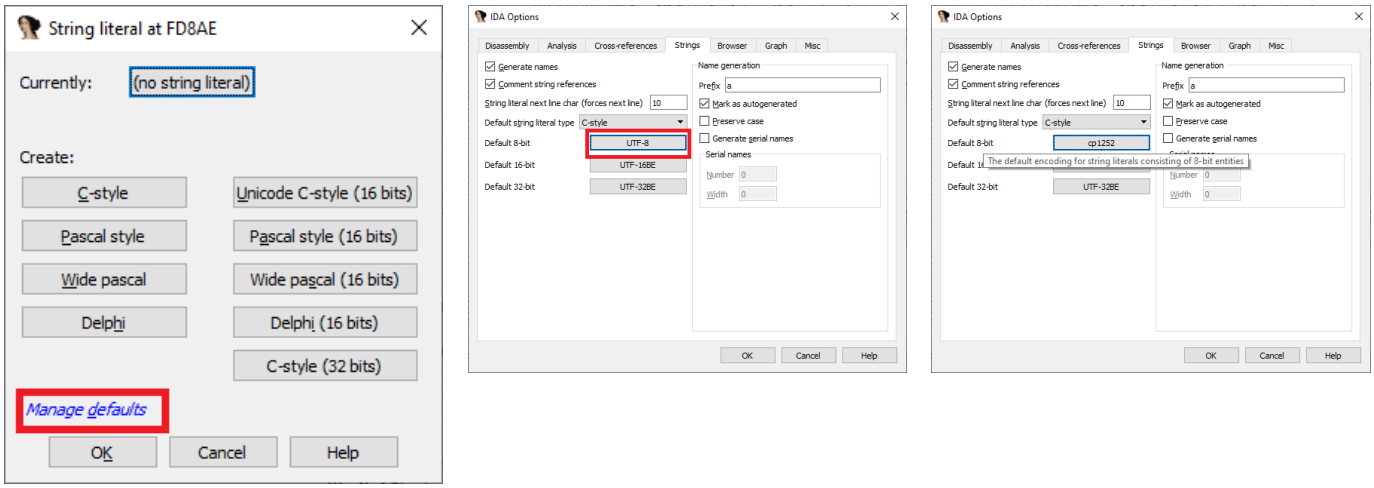
Use the encoding for a specific string literal

1. Invoke Options > String literals... (`Alt-A`);
2. Click the button next to “Currently:”;
3. Select the encoding to use;
4. Click the specific string button (e.g. C-Style) if creating a new literal or just OK if modifying an existing one.



Set an encoding as default for all new string literals

1. Invoke Options > String literals... (`Alt-A`);
2. Click “Manage defaults”;
3. Click the button next to “Default 8-bit” and select the encoding to use.



From now on, the `A` shortcut will create string literals with the new default encoding, but you can still override it on a case-by-case basis, as described above.

#14: Comments in IDA

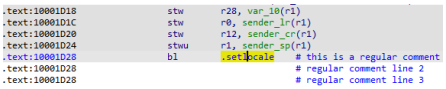
06 Nov 2020

https://hex-rays.com/blog/igor-tip-of-the-week-14-comments-in-ida/

The “I” in IDA stands for interactive, and one of the most common interactive actions you can perform is adding comments to the disassembly listing (or decompiler pseudocode). There are different types of comments you can add or see in IDA.

Regular comments

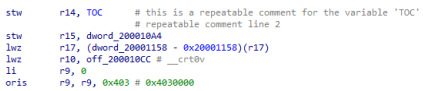
These comments are placed at the end of the disassembly line, delimited by an assembler-specific comment character (semicolon, hash, at-sign etc.). A multi-line comment shifts the following listing lines down and is printed aligned with the first line which is why they can also be called indented comments



Shortcut: `:` (colon)

Repeatable comments

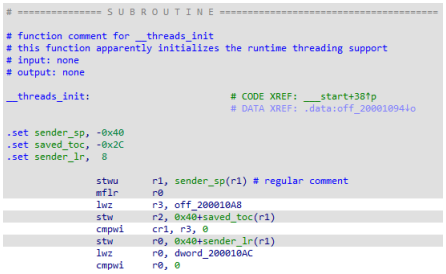
Basically equivalent to regular comments with one small distinction: they are repeated in any location which refers to the original comment location. For example, if you add a repeatable comment to a global variable, it will be printed at any place the variable is referenced.



Shortcut: `;` (semicolon)

Function comments

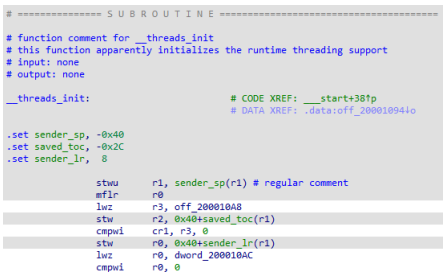
A repeatable comment added at the first instruction of a function is considered a function comment. It is printed before the function header and – since it’s a repeatable comment – at any place the function is called from. They’re good for describing what the function does in more detail than can be inferred from the function’s name.



Shortcut: `;` (semicolon)

Anterior and posterior comments

These are printed before (anterior) or after (posterior) the current address as separate lines of text, shifting all other listing lines. They are suitable for extended explanations, ASCII art and other freestanding text. Unlike regular comments, no assembler comment characters are added automatically.

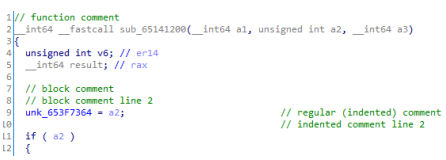


Shortcuts: `Ins`, `Shift-Ins` (`I` and `Shift-I` on Mac)

Trivia: the comment with file details that is usually added at the beginning of the listing is an anterior comment so you can use to edit it.

Pseudocode comments

In the decompiler pseudocode you can also add *indented*[1] comments using the shortcut `/` (slash) and *block*[2] comments using `Ins` (`I` on Mac). They are stored separately from the disassembly comments, however function comments are shared with those in disassembly.



[1] https://www.hex-rays.com/products/decompiler/manual/cmd_comments.shtml
[2] https://www.hex-rays.com/products/decompiler/manual/cmd_block_cmts.shtml

#14: Comments in IDA

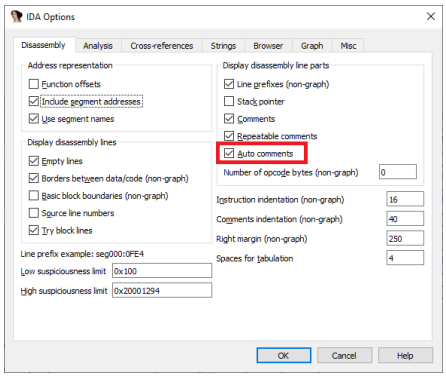
06 Nov 2020

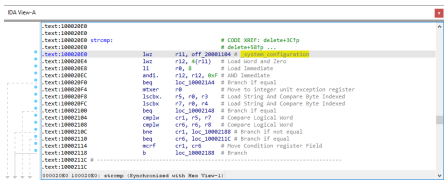
https://hex-rays.com/blog/igor-tip-of-the-week-14-comments-in-ida/

Automatic comments

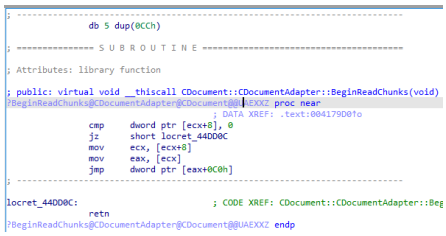
In some situations IDA itself can add comments to disassembly. A few examples:

“Auto comments” in Option > General.., Disassembly tab enables **instruction comments**.

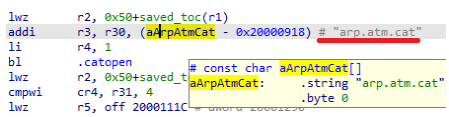




Demangled names are shown as auto comments by default. Use the Options > Demangled names... dialog if you prefer to replace the mangled symbol directly in the listing.



String literals work similarly to repeatable comments: the string contents shows up as a comment in the place it’s referenced from.



#15: Comments in structures and enums

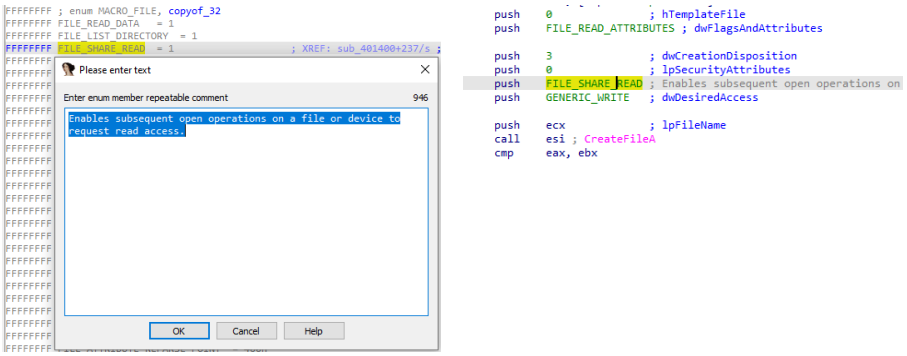
13 Nov 2020

https://hex-rays.com/blog/igor-tip-of-the-week-15-comments-in-structures-and-enums/

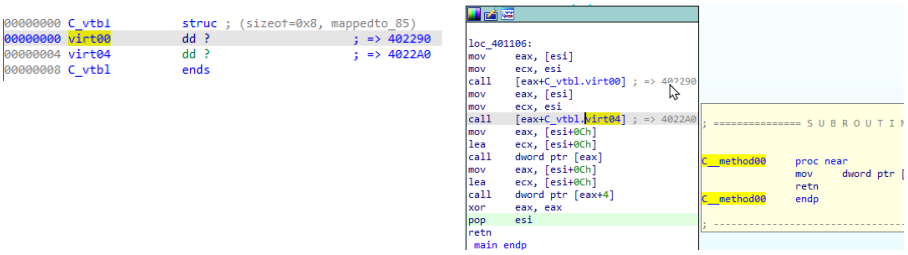
Last week we've discussed various kinds of comments[1] in IDA's disassembly and pseudocode views.

In fact, the comments are also available for Structures and Enums. You can add them both for the struct/enum as a whole and for individual members. Similar to the disassembly, regular and repeatable comments are supported.

Repeatable comments are duplicated in the listing when the enum or structure member is used.



One interesting use of this is for C++ class vtables (or any struct with pointers): if you add the comment with the method's address in the vtable structure, it will be printed in disassembly and you can double-click it to jump to the implementation or hover over it to see a hint window with disassembly.



[1] https://hex-rays.com/blog/igor-tip-of-the-week-14-comments-in-ida/

#16: Cross-references

20 Nov 2020

https://hex-rays.com/blog/igor-tip-of-the-week-16-cross-references/

cross-reference, n.
A reference or direction in one place in a book or other source of information to information at another place in the same work
(from Wiktionary[1])

To help you during analysis, IDA keeps track of cross-references (or xrefs for short) between different parts of the program. You can inspect them, navigate them or even add your own to augment the analysis and help IDA or the decompiler.

Types of cross-references

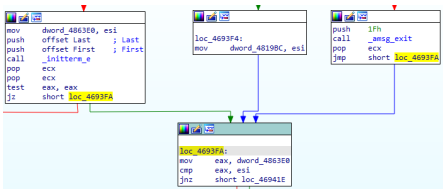
There are two groups of cross-references:

1. **code** cross-references indicate a relationship between two areas of code:
    1. **jump** cross-reference indicates conditional or unconditional transfer of execution to another location.
    2. **call** cross-reference indicates a function or procedure call with implied return to the address following the call instruction.
    3. **flow** cross-reference indicates normal execution flow from current instruction to the next. This xref type is rarely shown explicitly in IDA but is used extensively by the analysis engine and plugin/script writers need to be aware of it.
2. **data** cross-references are used for references to data, either from code or from other data items:
    1. **read** cross-reference indicates that the data at the address is being read from.
    2. **write** cross-reference indicates that the data at the address is being written to.
    3. **offset** cross-reference indicates that the address the of the item is taken but not explicitly read or written.
    4. **structure** cross-references are added when a structure is used in the disassembly or embedded into another structure.

The cross-reference types may be denoted by single-letter codes which are described in IDA's help topic "Cross reference attributes".

Quick menu navigation

In the graph view, code cross-references are shown as edges (arrows) between code blocks. You can navigate by following the arrows visually or double-clicking.



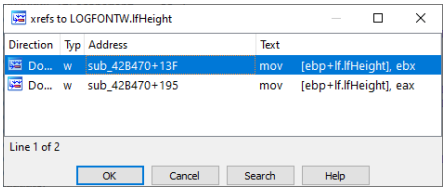
In text mode, cross-references to the current address are printed as comments at the end of the line. By default, maximum two references are printed; if there are more, ellipsis (...) is shown. You can increase the amount of printed cross-references in Options > General... Cross-references tab.

Only explicit references are shown in comments; flow cross-references are not displayed in text mode. However, the absence of a flow cross-reference (end of code execution flow) is shown by a dashed line; usually it's seen after unconditional jumps or returns but can also appear after calls to non-returning functions.

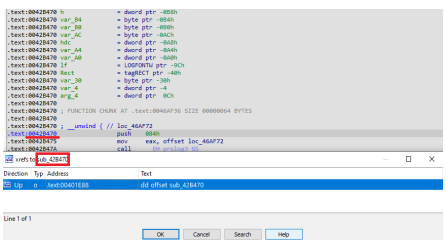
To navigate to the source of the cross-reference, double-click or press Enter on the address in the comment.

Shortcuts

`X` is probably the most common and useful shortcut: press it to see the list of cross-references to the **identifier under cursor**. Pick an item from the list to jump to it. The shortcut works not only for disassembly addresses but also for **stack variables** (in a function) as well as **structure** and **enum members**.



`Ctrl-X` works similarly but shows the list of cross-references to the **current address**, regardless of where the cursor is in the line. For example, it is useful when you need to check the list of callers of the current function while being positioned on its first instruction.



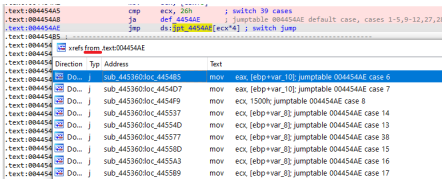
[1] https://en.wiktionary.org/wiki/cross-reference

#16: Cross-references

20 Nov 2020

https://hex-rays.com/blog/igor-tip-of-the-week-16-cross-references/

`Ctrl-J`, on the other hand, shows a list of cross-references **from the current address**. Having multiple cross-references from a single location to multiple others is a somewhat rare situation but one case where it's useful is **switches** (table jumps): using this shortcut on the indirect jump instructions allows you to quickly see and jump to any of the switch cases.



If you forget the shortcuts or simply prefer using the mouse, you can find the corresponding menu items in the Jump menu (and sometimes in the context menu).

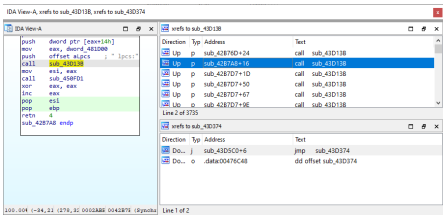
#17: Cross-references 2

27 Nov 2020

https://hex-rays.com/blog/igor-tip-of-the-week-17-cross-references-2/

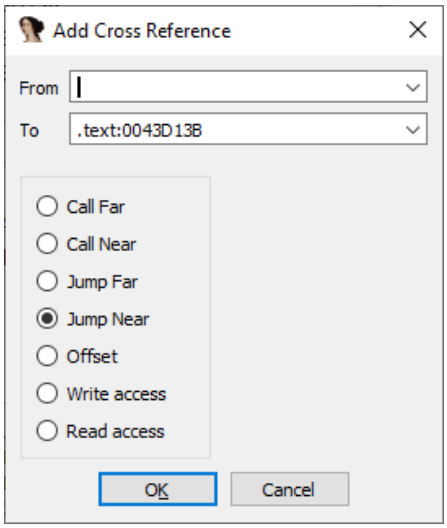
Cross references view

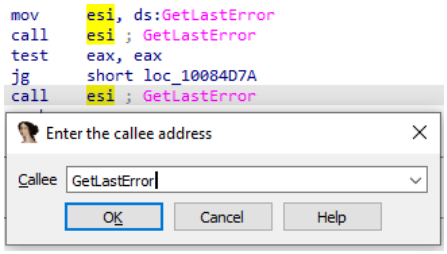
The jump to xref[1] actions are good enough when you have a handful of cross-references but what if you have hundreds or thousands? For such cases, the Cross references view may be useful. You can open it using the corresponding item in the View > Open Subviews menu. IDA will gather cross-references to the current disassembly address and show them in a separate tab. It's even possible to open several such views at the same time (for different addresses).



Adding cross-references

In some cases you may need to add a manual cross-reference, for example to fix up an obfuscated function's control flow graph or add a call cross-reference from an indirect call instruction discovered by debugging. There are several ways to do it.





• In the Cross references view, choose "Add cross-reference..." from the context menu or press `Ins`. In the dialog, enter source and destination addresses and the xref type.

• For **indirect calls** in binaries for **PC** (x86/x64), **ARM**, or **MIPS** processors, you can use Edit > Plugins > Set callee address (`Alt-F11`).

• To add cross-references **programmatically**, use IDC or IDAPython functions `add_cref` and `add_dref`[2]. Use the `XREF_USER` flag together with the xref type to ensure that your cross-reference is not deleted by IDA on reanalysis:

```
add_cref(0x100897E8, 0x100907C0, fl_CN|XREF_USER)
add_dref(0x100A65CC, 0x100897E0, dr_O|XREF_USER)
```

[1] https://hex-rays.com/blog/igor-tip-of-the-week-16-cross-references/
[2] https://hex-rays.com/products/ida/support/idadoc/313.shtml

#18: Decompiler and global cross-references

03 Dec 2020

https://hex-rays.com/blog/igors-tip-of-the-week-18-decompiler-and-global-cross-references/

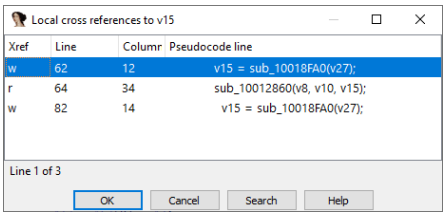
Previously we've covered cross-references[1] in the disassembly view but in fact you can also consult them in the decompiler (pseudocode) view.

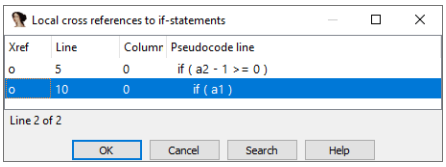
Local cross-references

The most common shortcut (**X**) works similarly to disassembly: you can use it on labels, variables (local and global), function names, but there are some differences and additions:

• for local variables, the list of cross-references shows *pseudocode* lines instead of disassembly snippets.

• if you press **X** on an C statement keyword (e.g. `if`, `while`, `return`), all statements of the same type in the current function will be shown



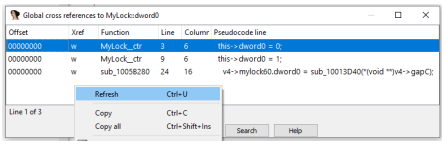


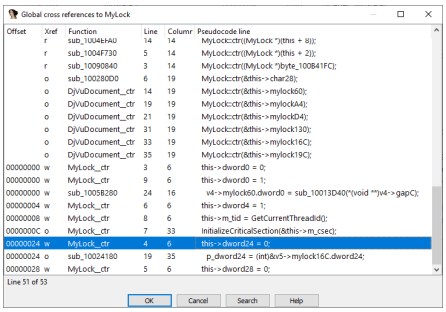
Global cross-references

If you have a well-analyzed database with custom types used by the program and properly set up function prototypes, you can ask the decompiler to analyze all functions and build a list of cross-references to a structure field, an enum member or a whole local type. The default hotkey is **`Ctrl-Alt-X`**.

When you use it for the first time, the list may be empty or include only recently decompiled functions.

To cover all functions, refresh the list from the context menu or by pressing **`Ctrl-U`**. This will decompile all functions in the database and gather the complete list. The decompilation results are cached so next time you use the feature it will be faster.





[1] https://hex-rays.com/blog/igor-tip-of-the-week-16-cross-references/

#19: Function calls

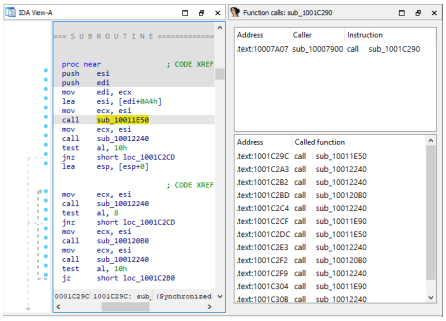
10 Dec 2020

https://hex-rays.com/blog/igor-tip-of-the-week-19-function-calls/

When dealing with big programs or huge functions, you may want to know how various functions interact, for example where the current function is called from and what other functions it calls itself. While for the former you can use “Cross-references to”, for the latter you have to go through all instructions of the function and look for calls to other functions. Is there a better way?

Function calls view

This view, available via View > Open subviews > Function calls, offers a quick overview of calls to and from the current function. It is dynamic and updates as you navigate to different functions so it can be useful to dock it next to the listing to be always visible. Double-click any line in the caller or called list to jump to the corresponding address.



#20: Going places

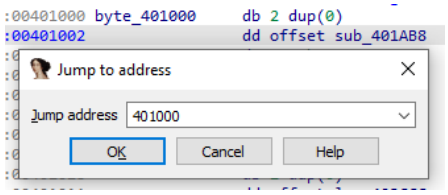
17 Dec 2020

https://hex-rays.com/blog/igors-tip-of-the-week-20-going-places/

Even if you prefer to move around IDA by clicking, the `G` shortcut should be the one to remember. The action behind it is called simply "Jump to address" but it can do many more things than what can be guessed from the name.

Jump to address

First up is the actual jumping to an address: enter an address value to jump to. You can prefix it with `0x` to denote hexadecimal notation but this is optional: in the absence of a prefix, the entered string is parsed as a hexadecimal number.



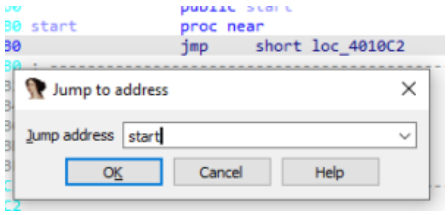
In architectures with segmented architecture (e.g. 16-bit x86), a segment:offset syntax can be used. Segment can a be symbolic name (`seg001`, `dseg`) or hexadecimal (`F000`); the offset should be hexadecimal. If the current database contains both segmented and linear (flat) addressed segments (e.g. a legacy 16-bit bootloader with 32-bit protected mode OS image in high memory), a "segment" `0` can be used to force the usage of linear address (`0:1000000`).

Jump relative to current location

If the entered value is prefixed with `+` or `-`, it is treated as relative offset from the cursor's position. Once again, the `0x` prefix is optional: `+100` jumps 256 bytes forward and `-10000` goes 64KiB(65536 bytes) backwards.

Jump relative to current location

A name (function or global variable name, or a label) in the program can be entered to jump directly to it. Note that the raw name should be entered as it's used in the program with any possible special symbols, for example `_main` for `main()` or `??2@YAPEAX_K@Z` for `operator new()`.



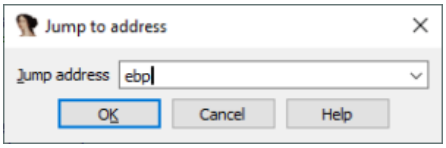
Jump to an expression

A C syntax expression can be used instead of a bare address or a name. Just like in C, the hexadecimal numbers must use the `0x` prefix – otherwise decimal is assumed. Names or the special keyword `here` can be used (and are resolved to their address). Some examples:

- `here + 32*4`: skip 32 dwords. Equivalent to `+80`
- `_main - 0x10`: jump to a position 0x10 bytes before the function main()
- `f2 + (f4-f3)`: multiple symbols can be used for complicated situations

Using registers

During debugging, you can use register names as variables, similarly to names in preceding examples. For example, you can jump to `EAX`, `RSP`, `ds:si`(16-bit x86), `X0+0x20`(ARM64) and so on. This works both in disassembly and the hex view.



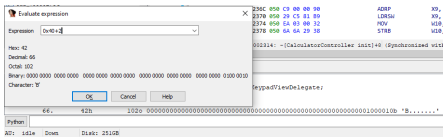
#21: Calculator and expression evaluation feature in IDA

08 Jan 2021

https://hex-rays.com/blog/igors-tip-of-the-week-21-calculator-and-expression-evaluation-feature-in-ida/

When reverse-engineering, sometimes you need to perform some simple calculations. While you can always use an external calculator program, IDA has a built-in one. You can invoke it by pressing ? or via View > Calculator.

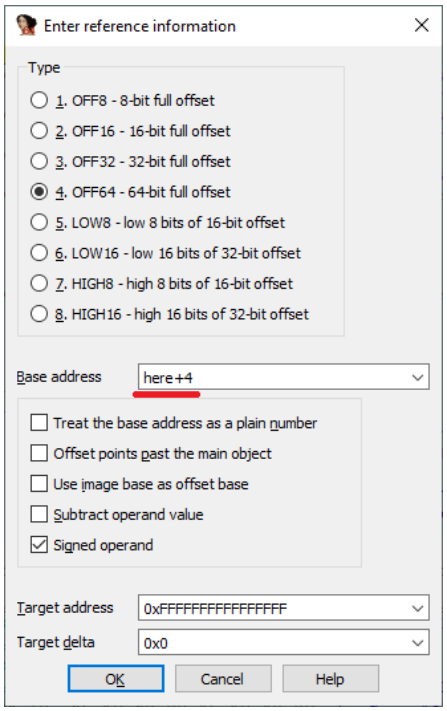
The calculator shows the result in hex, decimal, octal, binary and as a character constant. This information is also duplicated in the Output window in case you need to copy it to somewhere else.



In addition to plain numbers, you can use names from the database, as well as register values during debugging similarly to the "Jump to address" dialog from the previous tip[1].

By the way, the number, address, or identifier under cursor is picked up automatically when you press ? so there's no need to copy or type it manually.

In fact, the expression evaluation feature is provided by the IDC language[2] interpreter built-in into IDA. You can use expressions in almost any place in IDA that accepts numbers: Jump to address, Make array, User-defined offset and so on.



You can also use any of the available IDC functions[3]. For example, expressions like the following are possible during debugging:

```
get_qword(__security_cookie)^RSP
```

[1] https://www.hex-rays.com/blog/igors-tip-of-the-week-20-going-places/
[2] https://www.hex-rays.com/products/ida/support/idadoc/157.shtml
[3] https://www.hex-rays.com/products/ida/support/idadoc/162.shtml

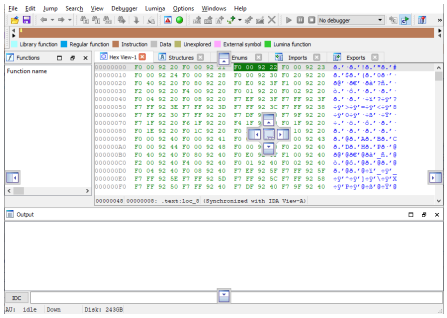
#22: IDA desktop layouts

15 Jan 2021

https://hex-rays.com/blog/igors-tip-of-the-week-22-ida-desktop-layouts/

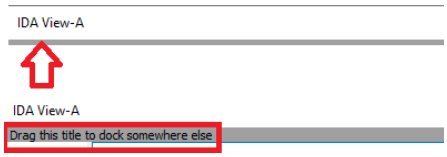
IDA's default windows layout is sufficient to perform most standard analysis tasks, however it may not always be the best fit for all situations. For example, you may prefer to open additional views or to modify existing ones depending on your monitor size, specific tasks, or the binary being analyzed.

Rearranging windows



The standard operation is mostly intuitive – click and drag the window title to dock the window elsewhere. While dragging, you will see the drop markers which can be used to dock the window next to another or as a tab. You can also release the mouse without picking any marker to make the window float independently.

Docking a floating window



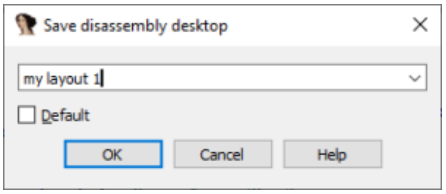
Once a window is floating, you can't dock it again by dragging the title. Instead, hover the mouse just below to expose the drag handle which can be used to dock it again.

Reset layout

If you want to start over, use Windows > Reset desktop to go back to the default layout.

Saving and using custom layouts

The layout is saved automatically in the database, but if you want to reuse it later with a different one, use Windows > Save desktop... to save it under a custom name and later Windows > Load desktop... to apply it in another database or session. Alternatively, check the "Default" checkbox to make this layout default for all new databases.

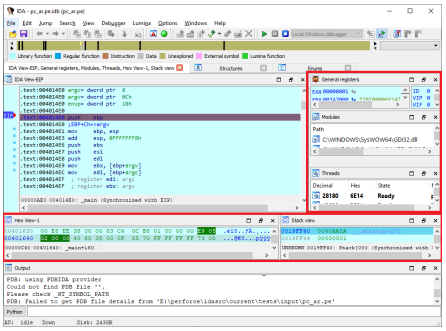


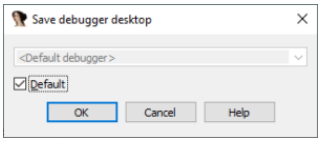
Debugger desktop

When debugging, the windows layout changes to add views which are useful for the debugger (e.g. debug registers, Modules, Threads). This can lead to crowded display on small monitors so rearranging them can become a frequent task.

This layout is separate from the disassembly-time one so if you want to persist a custom debugger layout, you need to save it during the debug session.

More info: Desktops[1] in the IDA Help.





[1] https://hex-rays.com/products/ida/support/idadoc/1418.shtml

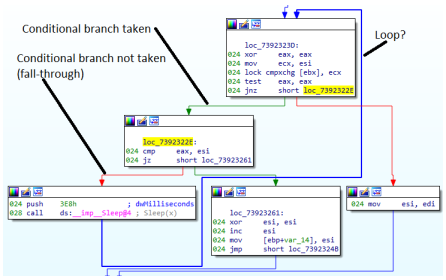
#23: Graph view

22 Jan 2021

https://hex-rays.com/blog/igors-tip-of-the-week-23-graph-view/

Graph view is the default disassembly representation in IDA GUI and is probably what most IDA users use every day. However, it has some lesser-known features that can improve your workflow.

Parts of the graph



The graph consists of nodes (blocks) and edges (arrows between blocks). Each node roughly corresponds to a basic block.

a **basic block** is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit.
(from Wikipedia[1])

Edges indicate code flow between nodes and their color changes depending on the type of code flow:

- conditional jumps/branches have two outgoing edges: green for branch taken and red for branch not taken (i.e. fall through to next address);
- other kind of edges are blue;
- edges which go backwards in the graph (which usually means they're part of a loop) are thicker in width.

Keyboard controls

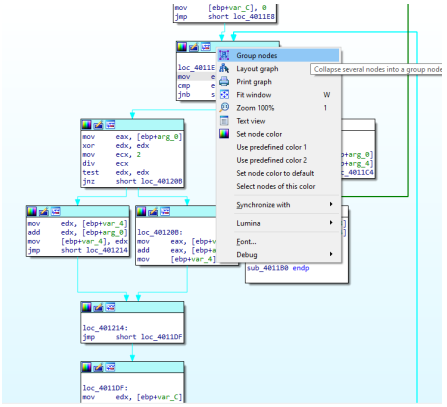
- `W` to zoom out so the whole graph fits in the visible window area;
- `1` to zoom back to 100%;
- `Ctrl-Up` moves to the parent node;
- `Ctrl-Down` moves to the child node
  (if there are several candidates in either case, a selector is displayed)

Mouse controls

Besides the usual clicking around, a few less obvious mouse actions are possible:

- double-click an edge to jump to the other side of it or hover to preview the target (source) node;
- click and drag the background to pan the whole graph in any directions;
- use the mouse wheel to scroll the graph vertically (up/down);
- `Alt`+wheel to scroll horizontally (left/right);
- `Ctrl`+wheel to zoom in/out

Rearranging and grouping the nodes



If necessary, you can move some nodes around by dragging their titles. Edges can also be moved by dragging their bending points. Use "Layout graph" from the context menu to go back to the initial layout.

Big graphs can be simplified by grouping:

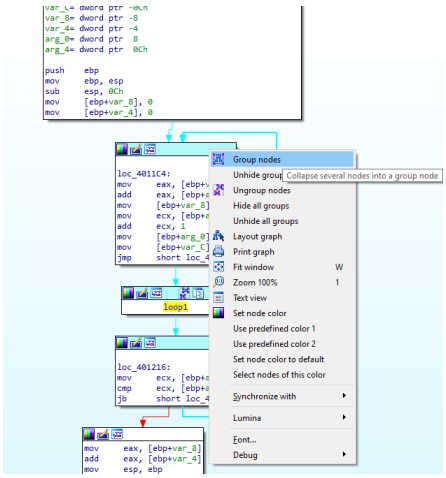
1. Select several nodes by holding down `Ctrl` and clicking the titles of multiple nodes or by click-dragging a selection box. The selected nodes will have a different color from others (cyan in default color scheme);
2. Select "Group nodes" from the context menu and enter the text for the new node. IDA will replace selected nodes with the new one and rearrange the graph;
3. You can repeat the process as many times as necessary, including grouping already-grouped nodes;
4. Created groups can be expanded again temporarily or ungrouped completely, going back to separate nodes. Use the context menu or new icons in the group node's title bar for this.

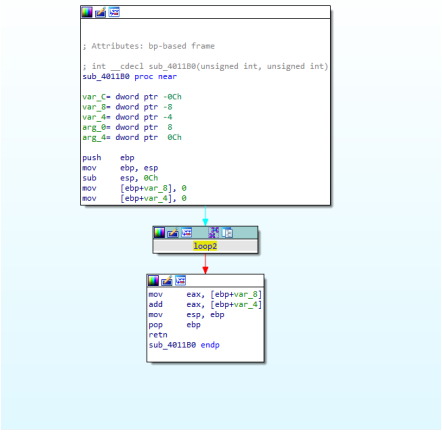
[1] https://en.wikipedia.org/wiki/Basic_block

#23: Graph view

22 Jan 2021

https://hex-rays.com/blog/igors-tip-of-the-week-23-graph-view/





More info: Graph view in IDA Help[2] (also available via `F1` in IDA).

[2] https://www.hex-rays.com/products/ida/support/idadoc/42.shtml

#24: Renaming registers

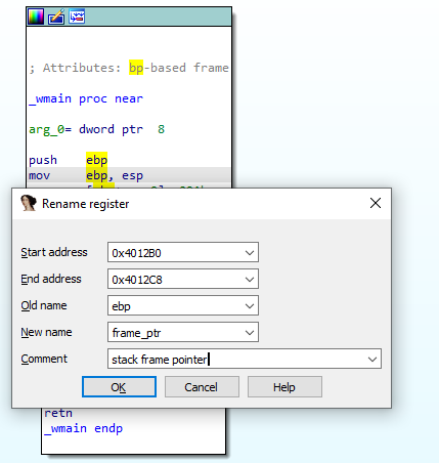
29 Jan 2021

https://hex-rays.com/blog/igors-tip-of-the-week-24-renaming-registers/

While register highlighting can help tracking how a register is used in the code, sometimes it's not quite sufficient, especially if multiple registers are used by a complicated piece of code. In such situation you can try *register renaming*.

To rename a register:

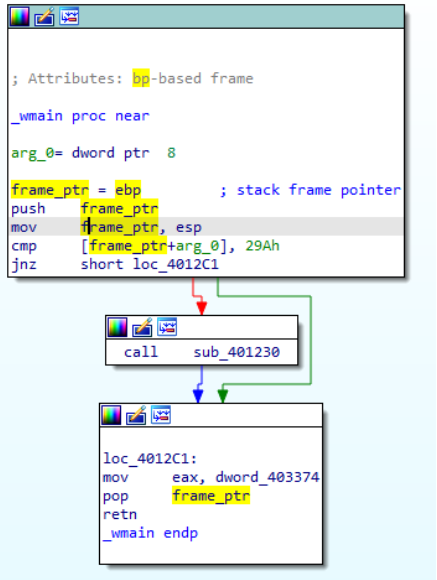
- place the cursor on it and press `N` or `Enter`, or
- double-click it



A dialog appears where you can specify:

- new name to be used in the disassembly;
- comment to be shown at the place of the new name's definition;
- range of addresses where to use the name.

The address range defaults to the current function boundaries but you can either edit them manually or select a range before renaming (this can be tricky since the cursor needs to be on the register). The new range cannot cross function boundaries (registers can be renamed only inside a function). The new name and the comment are printed at the start of the specified range.



Even if you don't rename registers yourself, you may encounter them in your databases. For example, the DWARF plugin can use the information available in the DWARF debug info to rename and comment registers used for storing local variables or function arguments.

To undo renaming and revert back to the canonical register name, rename it to an empty string.

See also: Rename register[1] in the IDA Help.

```
; void vman5_01AddDriverClass(t_DeviceType driver, t_VMAL_InitDriv
EXPORT vman5_01AddDriverClass
vman5_01AddDriverClass

fp_init= -0x20
fp_readpartitiontable_0= 0
fp_writepartitiontable_0= 4
fp_getdevicecharateristics_0= 8

driver = R0 ; t_DeviceType
fp_init_0 = R1 ; t_VMAL_InitDriver
fp_registervolume = R2 ; t_VMAL_RegisterVolume
fp_unregistervolume = R3; t_VMAL_UnregisterVolume
PUSH {driver-R7,LR}
MOVS R1, fp_unregistervolume
fp_unregistervolume = R1; t_VMAL_UnregisterVolume
fp_readpartitiontable = R4; t_VMAL_ReadPartitionTable
fp_writepartitiontable = R5; t_VMAL_WritePartitionTable
fp_getdevicecharateristics = R6; t_VMAL_GetDeviceCharacteristics
MOVS R3, #0x1C
ADD fp_readpartitiontable, SP, #0x24+fp_readpartitiont
MULS R3, driver
LDR R7, =drv_classes
LDM fp_readpartitiontable, {fp_readpartitiontable-fp_g
STRB driver, [R7,R3]
LDR R0, [SP,#0x24+fp_init]
ADDS R3, R3, R7
STR R0, [R3,#4]
MOVS R0, R3
ADDS R0, #0xC
STM R0!, {fp_unregistervolume,fp_readpartitiontable-fp
STR fp_registervolume, [R3,#8]
POP {R0-R7,PC}
; End of function vman5_01AddDriverClass
```

[1] https://www.hex-rays.com/products/ida/support/idadoc/1346.shtml

#25: Disassembly options

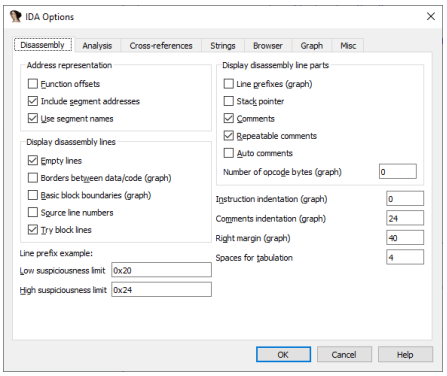
05 Feb 2021

https://hex-rays.com/blog/igors-tip-of-the-week-25-disassembly-options/

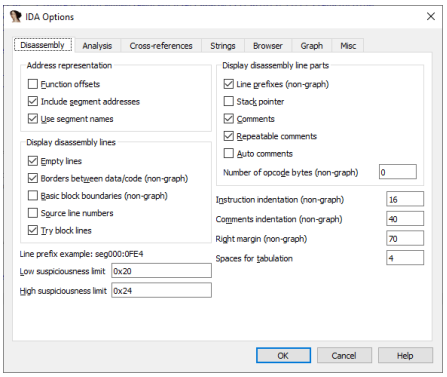
By default IDA's disassembly listing shows the most essential information: disassembled instructions with operands, comments, labels. However, the layout of this information can be tuned, as well as additional information added. This can be done via the Disassembly Options tab available via Options > General... menu (or `Alt-0`, `G`).

Text and Graph views options

If you open the options dialog in graph mode, you should have something like the following:



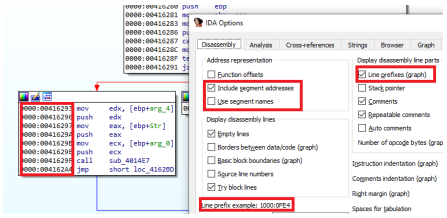
And if you do it in text mode (use `Space` to switch), it will be different:



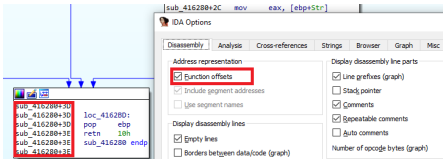
As you may notice, some options are annotated with (graph) or (non-graph), denoting the fact that IDA keeps two sets of options for different modes of disassembly. To make the graphs look nicer, the defaults are tuned so that the nodes are relatively narrow, while the text mode can use the full width of the window and is spaced out more. However, you can still tweak the options of either mode to your preference and even save them as a named or default desktop layout[1].

Line prefixes

One example of a setting which is different in text and graph modes is "Line prefixes" (enabled in text mode, disabled in graph mode). Prefix is the initial part of the disassembly line which indicates its address (e.g. `.text:00416280`). For example, you can enable it in the graph too or disable display of the segment name to save space.



Or you can show offsets from start of the function instead of full addresses:



This can be convenient because you always know which function you're currently analyzing.

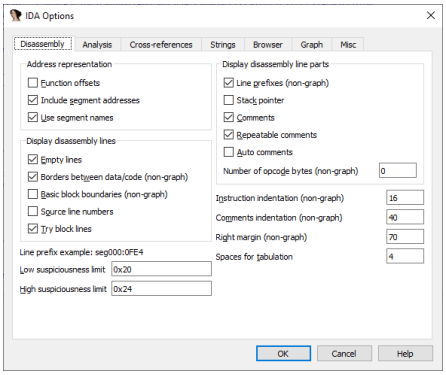
[1] https://www.hex-rays.com/blog/igors-tip-of-the-week-22-ida-desktop-layouts/

#26: Disassembly options 2

12 Feb 2021

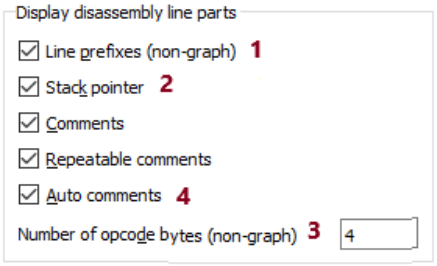
https://hex-rays.com/blog/igors-tip-of-the-week-26-disassembly-options-2/

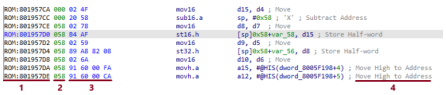
Continuing from last week[1], let's discuss other disassembly options you may want to change. Here's the options page again:



Disassembly line parts

This group is for options which control the content of the main line itself. Here is an example of a line with all options enabled:



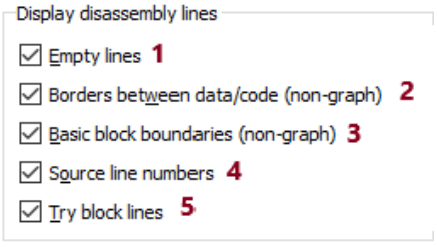


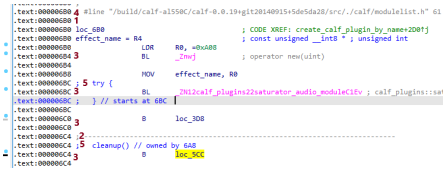
The marked up parts are:

1. The line prefix (address of the line).
2. The stack pointer value or delta (relative to the value at the entry point). Enabling this can be useful when debugging problems like "sp-analysis failed", "positive sp v value has been detected", or "call analysis failed".
3. Opcode bytes. The number entered in the "Number of opcode bytes" specifies the number displayed on a single line at most. If the instruction is longer, the rest is printed on the second line. If you prefer to truncate the extra bytes, enter a negative number (e.g. -4 will display 4 bytes at most, the rest will be truncated).
4. Comments for instructions with a short description of what the instruction is doing (may not be available for all processors or all instructions).

Display disassembly lines

This group of options control display of lines other than the actual line of the disassembly for a given address (main line).





1. Empty lines: this prints additional empty lines to make disassembly more readable, especially in text mode (e.g. between functions or before labels). Turn it off to fit more code on screen.
2. Borders between data/code: displays the border line (`; ----------`) whenever there is a stop in the execution flow (e.g. after an unconditional jump or a call to a non-returning function).
3. Basic block boundaries: adds one more empty line at the end of each basic block (i.e. after a call or a branch).
4. Source line numbers: displays source file name and line number if this information is available in the database (e.g. imported from the DWARF debug information).
5. Try block lines: enables or disables display of information about exception handling recovered by parsing the exception handling metadata in the binary.

[1] https://www.hex-rays.com/blog/igors-tip-of-the-week-25-disassembly-options/

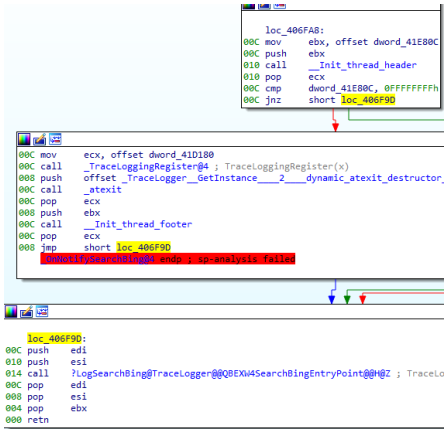
#27: Fixing the stack pointer

19 Feb 2021

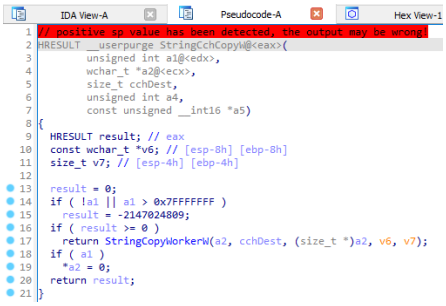
https://hex-rays.com/blog/igors-tip-of-the-week-27-fixing-the-stack-pointer/

As explained in Simplex method in IDA Pro[1], having correct stack change information is essential for correct analysis. This is especially important for good and correct decompilation. While IDA tries its best to give good and correct results (and we've made even more improvements since 2006), sometimes it can still fail (often due to wrong or conflicting information). In this post we'll show you how to detect and fix problems such as:

"sp-analysis failed"



"positive sp value has been detected"

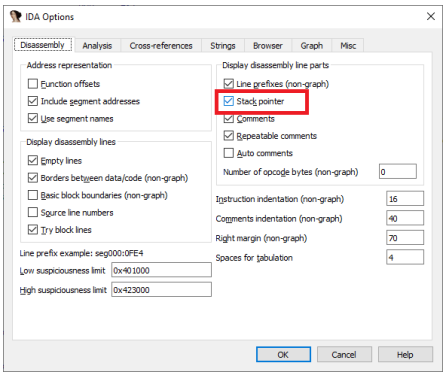


Both examples are from the 32-bit build of notepad.exe from Windows 10 (version 10.0.17763.475) with PDB symbols from Microsoft's public symbol server applied.

Note: in many cases the decompiler will try to recover and still produce reasonable decompilation but if you need to be 100%

Detecting the source of the problem

The first steps to resolve them are usually:



1. Switch to the disassembly view (if you were in the decompiler);
2. Enable "Stack pointer" under "Disassembly, Disassembly line parts" in Options > General...;
3. Look for unusual or unexpected changes in the SP value (actually it's the SP delta value) now added before each instruction.

To detect "unusual changes" we first need to know what is "usual". Here are some examples:

- push instructions should increase the SP delta by the number of pushed bytes (e.g. `push eax` by 4 and `push rbp` by 8)
- conversely, pop instructions decrease it by the same amount
- call instructions usually either decrease SP to account for the pushed arguments (`__stdcall` or `__thiscall` functions on x86), or leave it unchanged to be decreased later by a separate instruction
- the values on both ends of a jump (conditional or unconditional) should be the same
- the value at the function entry and return instructions should be 0
- between prolog and epilog the SP delta should remain the same with the exception of small areas around calls where it can increase by pushing arguments but then should return back to "neutral" before the end of the basic block.

In the first example, we can see that `loc_406F9D` has the SP delta of `00C` and the first jump to it is also `00C`, however the second one is `008`. So the problem is likely in that second block. Here it is separately:

[1] https://www.hex-rays.com/blog/simplex-method-in-ida-pro/

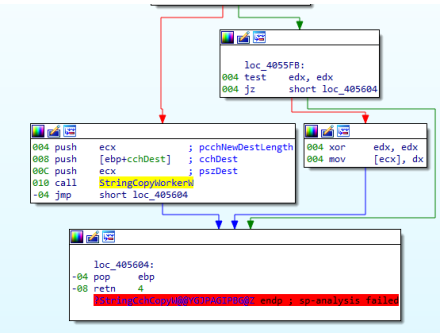
#27: Fixing the stack pointer

19 Feb 2021

https://hex-rays.com/blog/igors-tip-of-the-week-27-fixing-the-stack-pointer/

```
00C mov    ecx, offset dword_41D180
00C call   _TraceLoggingRegister@4 ; TraceLoggingRegister(x)
008 push   offset _TraceLogger__GetInstance___2___dynamic_atexit_destructor_for__s_instance__ ; void (__cdecl *)()
00C call   _atexit
00C pop    ecx
008 push   ebx
00C call   __Init_thread_footer
00C pop    ecx
008 jmp    short loc_406F9D
```

We can see that `00C` changes to 008 after the call to `_TraceLoggingRegister@4`. On the first glance it makes sense because the `@4` suffix denotes `__stdcall function`[2] with 4 bytes of arguments (which means it removes 4 bytes from the stack). However, if you actually go inside and analyze it, you'll see that it does not use stack arguments but the register `ecx`. Probably the file has been compiled with `Link-time Code Generation`[3] which converted __stdcall to __fastcall to speed up the code.



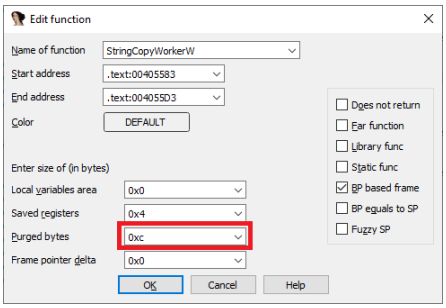
In the second case the disassembly looks like following:

Here, the problem is immediately obvious: the delta becomes negative after the call. It seems IDA decided that the function is subtracting 0x14 bytes from the stack while there are only three pushes (3*4 = 12 or 0xC). You can also go inside `StringCopy-WorkerW` and observe that it ends with `retn 0Ch` – a certain indicator that this is the correct number.

Fixing wrong stack deltas

How to actually fix the wrong delta depends on the specific situation but generally there are two approaches:

1. Fix just the place(s) where things go wrong. For this, press **`Alt-K`** (Edit > Functions > Change stack pointer...) and enter the correct amount of the SP change. In the first example it should be 0 (since the function is not using any stack arguments) and in the second 12 or 0xc. Often this is the only option for indirect calls.
2. If the same function called from multiple places causes stack unbalance issues, edit the function's properties (**`Alt-P`** or Edit > Functions > Edit function...) and change the "Purged bytes" value.



This simple example shows that even having debug symbols does not guarantee 100% correct results and why giving override options to the user is important.

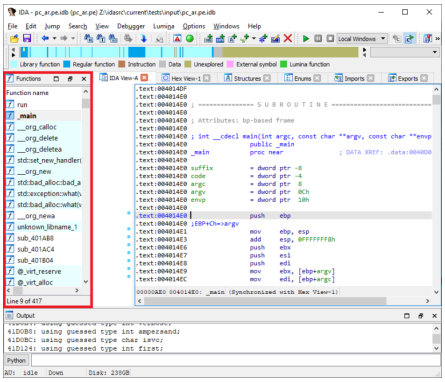
[2] https://docs.microsoft.com/en-us/cpp/cpp/stdcall
[3] https://docs.microsoft.com/en-us/cpp/build/reference/ltcg-link-time-code-generation

#28: Functions list

26 Feb 2021

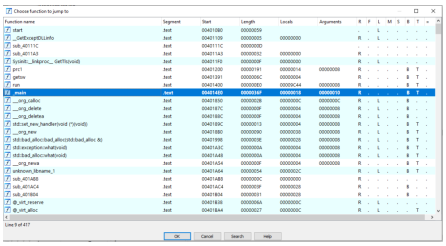
https://hex-rays.com/blog/igors-tip-of-the-week-28-functions-list/

The Functions list is probably one of the most familiar features of IDA's default desktop layout. But even if you use it every day, there are things you may not be aware of.



Modal version

Available via Jump > Jump to function... menu, or the `Ctrl-P` shortcut, the modal dialog lets you see the full width of the list as well as do some quick navigation, for example:



1. To jump to the current function's start, use `Ctrl-P`, `Enter` ;
2. To jump to the previous function, use `Ctrl-P`, `Up`, `Enter` (also available as JumpPrevFunc action: default shortcut is `Ctrl-Shift-Up`);
3. To jump to the next function, use `Ctrl-P`, `Down`, `Enter` (also available as JumpNextFunc action: default shortcut is `Ctrl-Shift-Down`).

Columns

As can be seen on the second screenshot, the Functions list has many more columns than Function name which is often the only one visible. They are described in the corresponding help topic[1]. By clicking on a column you can ask IDA to sort the whole list on that column. For example, you can sort the functions by size to look for largest ones – the bigger the function, the more chance it has a bug; or you may look for a function with the biggest Locals area since it may have many buffers on the stack which means potential overflows.

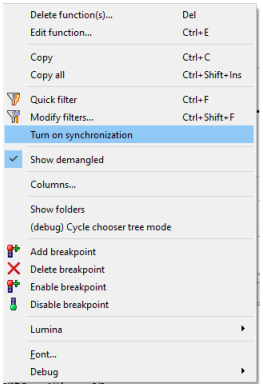
If you sort or filter the list, you may see the following message in the Output window:

```
Caching 'Functions window'... ok
```

Because sorting requires the whole list, IDA has to fetch it and re-sort on almost any change in the database since it may change the list. On big databases this can become quite slow so once you don't need sorting anymore, it's a good idea to use "Unsort" from the context menu.

Synchronization

The list can be synchronized with the disassembly by selecting "Turn on synchronization" from the context menu. Once enabled, the list will scroll to the current function as you navigate in the database. You can also turn it off if you prefer to see a specific function in the list no matter where you are in the listing.



[1] https://www.hex-rays.com/products/ida/support/idadoc/586.shtml

#28: Functions list

26 Feb 2021

https://hex-rays.com/blog/igors-tip-of-the-week-28-functions-list/

Folders

Since IDA 7.5, folders can be used to organize your functions. To enable, select “Show folders” in the context menu, then “Create folder with items...” to group selected items into a folder.

Colors & styles

Some functions in the list may be colored. In most cases the colors match the legend in the navigation bar:

- Cyan: Library function (i.e. a function recognized by a FLIRT signature[2] as a compiler runtime library function)
- Magenta/Fuchsia: an external function thunk, i.e. a function implemented in an external module (often a DLL or a shared object)
- Lime green: a function with metadata retrieved from the Lumina database[3]

But there are also others:

- Light green: function marked as decompiled[4]
- Other: function with manually set color (via Edit function... or a plugin/script)

You may also see functions marked **in bold**. These are functions which have a defined prototype (i.e types of arguments, return value and calling convention). The prototype may be defined by the user (Y hotkey[5]), or set by the loader or a plugin (e.g. from the DWARF or PDB debug information).

Multi-selection

By selecting multiple items you can perform some operations on all of them, for example:

- Delete function(s)...: deletes the selected functions by removing the function info (name, bounds) from the database. The instructions previously belonging to the functions remain so this can be useful, for example, for combining incorrectly split functions.
- Add breakpoint: adds a breakpoint to the first instruction of all selected functions. This can be useful for discovering which functions are executed when you trigger a specific functionality in the program being debugged.
- Lumina: you can push or pull metadata only for selected functions.

[2] https://www.hex-rays.com/products/ida/tech/flirt/
[3] https://www.hex-rays.com/products/ida/lumina/
[4] https://www.hex-rays.com/products/decompiler/manual/cmd_mark.shtml
[5] https://www.hex-rays.com/products/ida/support/idadoc/1361.shtml

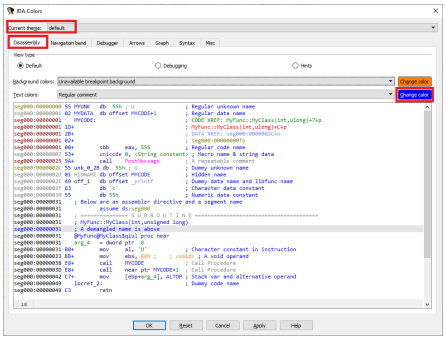
#29: Color up your IDA

05 Mar 2021

https://hex-rays.com/blog/igors-tip-of-the-week-29-color-up-your-ida/

For better readability, IDA highlights various parts of the disassembly listing using different colors; however these are not set in stone and you can modify most of them to suit your taste or situation. Let's have a look at the different options available for changing colors in IDA.

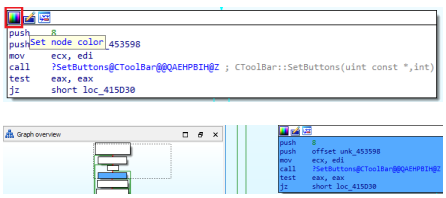
Themes



In case you are not aware, IDA supports changing the color scheme used for the UI (windows, controls, views and listings). The default theme uses light background but there are also two dark themes available. You can change the theme used via Options > Colors... ("Current theme" selector). Each theme then can be customized further by editing the colors in the tabs below. In the Disassembly tab, you can either select items from the dropdown, or click on them in the listing, then change the color by clicking the corresponding button.

If you prefer editing color values directly, you can update many of them at once or even create a complete custom theme by following the directions on the "CSS-based styling" page.

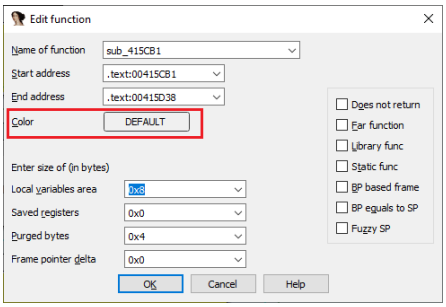
Coloring graph nodes



In the Graph View, you can color whole nodes (basic blocks) by clicking the first icon (Set node color) in the node's header.

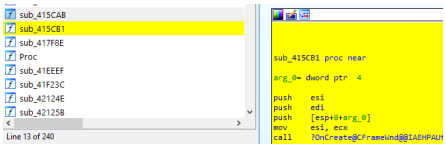
After choosing the color, all instructions in the block will be colored and it will also be shown with the corresponding color in the graph overview.

Coloring functions



Instead of (or in addition to) marking up single instructions or basic blocks you can also color whole functions. This can be done in the Edit Function (+) dialog by clicking the corresponding button.

Changing the color of a function colors all instructions contained in it (except those colored individually), as well as its entry in the Functions list.



#30: Quick views

12 Mar 2021

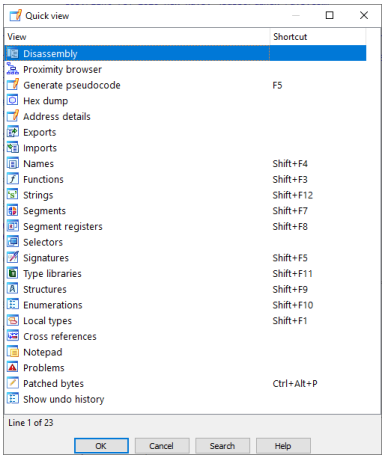
https://hex-rays.com/blog/igors-tip-of-the-week-30-quick-views/

IDA has three shortcuts as an alternative to some menus which could be cumbersome to navigate.

Quick view

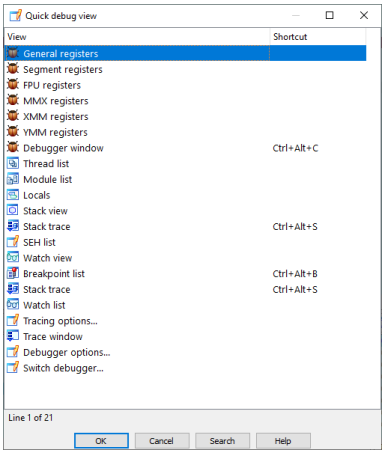
Probably the most commonly used, it is triggered by the shortcut `Ctrl+1` and shows the items under the View > Open subviews menu.

It can be especially useful for opening views which have no dedicated shortcut such as Notepad (although you can always assign a custom one via the Shortcut editor[1]).



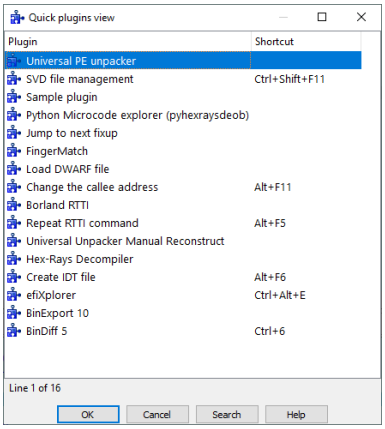
Quick debug view

Most useful during a debugging session, this one allows you to bypass navigating to the Debugger > Debugger windows menu by simply pressing `Ctrl+2`.



Quick debug view

Last but not least, `Ctrl+3` opens the list of plugin menu items listed under the Edit > Plugins menu, allowing you to quickly invoke a specific plugin. Please note that this list does not necessarily include all installed plugins; some plugins add menu items elsewhere or may not have a menu item at all and work in an automatic fashion.



[1] https://www.hex-rays.com/blog/igor-tip-of-the-week-02-ida-ui-actions-and-where-to-find-them/

#31: Hiding and Collapsing

19 Mar 2021

https://hex-rays.com/blog/igors-tip-of-the-week-31-hiding-and-collapsing/

You may have come across the menu items View > Hide, Unhide but possibly never used them.

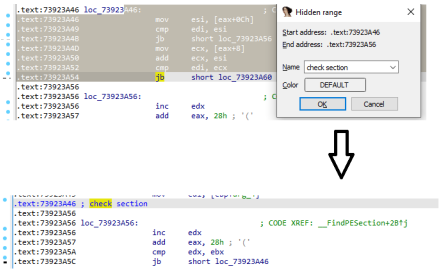
These commands allow you to hide, or collapse and unhide/uncollapse parts of IDA's output. They can be used in the following situations:

Hiding instructions or data items

To make your database more compact and reduce clutter, you can opt to hide or replace some parts of the listing by short text:

1. Select some instructions or data items
2. Invoke View > Hide (or press `Ctrl+Numpad-`)
3. Enter the text with which to replace the selected area (and optionally pick a color)

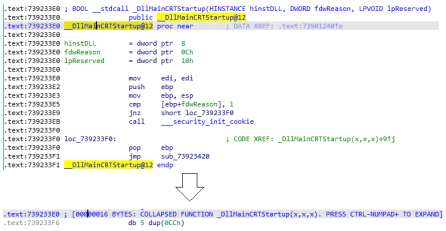
The instructions/data are replaced by the entered text but are not removed from the database; you can reveal them using View > Unhide (or `Ctrl+Numpad+`).



Hiding whole functions

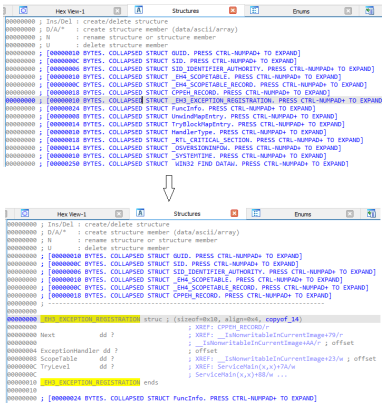
You can also hide or collapse whole functions by using the Hide command while the cursor is on the function's name:

You may have already seen the "COLLAPSED FUNCTION" text for library functions detected by the FLIRT signatures (colored cyan in the function list and navigation bar). The actual implementation of library functions is rarely important for analyzing the program's code so IDA collapses them to not distract the user.



Hiding structures and enums

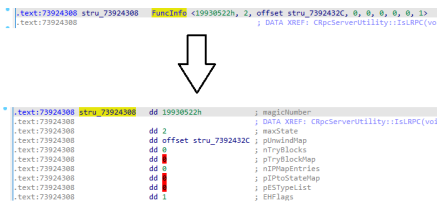
Structure or enum definitions can be collapsed and uncollapsed similarly to functions.



Terse struct representation

When defining structure instances in data, IDA will by default try to display them in terse form, with everything on one line. By using **Unhide**, you can have it printed in full, or verbose form, with each field on separate line and a comment with the field name.

Conversely, you can use **Hide** to collapse a structure instance into a terse form (this may not work in some cases due to the specific structure's layout).



#31: Hiding and Collapsing

19 Mar 2021

https://hex-rays.com/blog/igors-tip-of-the-week-31-hiding-and-collapsing/

Collapsing blocks in decompiler

The decompiler also has similar but separate pair of actions. They are available in the context menu or via the `Numpad-` and `Numpad+` hotkeys. You can collapse compound operators, as well as the variable declaration block at the start of the function.

More info:

Hide[1] and Unhide[2] (IDA)

Collapse/uncollapse item[3] (Decompiler)

[1] https://hex-rays.com/products/ida/support/idadoc/599.shtml
[2] https://hex-rays.com/products/ida/support/idadoc/600.shtml
[3] https://www.hex-rays.com/products/decompiler/manual/cmd_collapse.shtml

#32: Running scripts

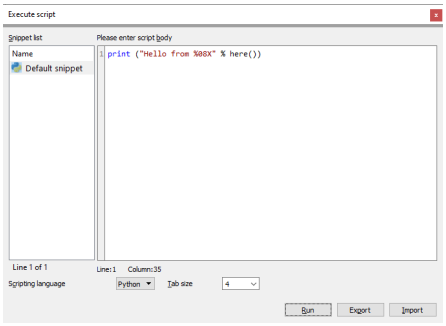
26 Mar 2021

https://hex-rays.com/blog/igors-tip-of-the-week-32-running-scripts/

Scripting allows you to automate tasks in IDA which can be repetitive or take a long time to do manually. We previously covered[1] how to run them in batch (headless) mode, but how can they be used interactively?

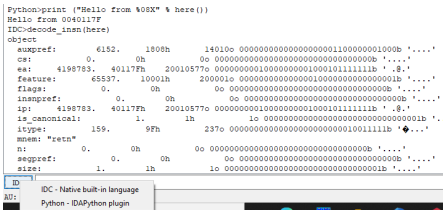
Script snippets

File > Script Command... (`Shift+F2`)



Although this dialog is mainly intended for quick prototyping and database-specific snippets, you can save and load scripts from external files via the “Export” and “Import” buttons. There is some basic syntax highlighting but it’s not a replacement for a full-blown IDE. Another useful feature is that the currently selected snippet can be executed using the `Ctrl+Shift+X` shortcut (“SnippetsRunCurrent” action) even

Command Line Interface (CLI)



While somewhat awkward to use for bigger tasks, it has a couple of unique features:

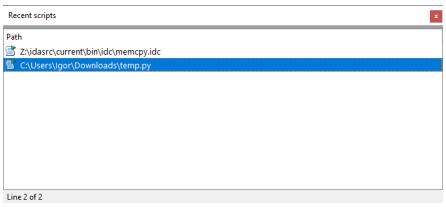
- the result of entered expression is printed in the Output Window (unless inhibited with a semicolon). In case of IDC, values are printed in multiple numeric bases and objects are pretty-printed recursively.
- It supports limited Tab completion[2].

Command Line Interface (CLI)

If you already have a stand-alone script file and simply want to run it, File > Script file.. (`Alt+F7`) is probably the best and quickest solution. It supports both IDC and Python scripts.

Recent scripts

The scripts which were executed through the “Script file...” command are remembered by IDA and can be executed again via the Recent Scripts list (View > Recent scripts, or `Alt+F9`). You can also invoke an external editor (configured in Options > General..., Misc tab) to edit the script before running.



Command Line Interface (CLI)

IDA ships with some example scripts which can be found in “idc” directory for IDC and “python/examples” for IDAPython. There are also some user-contributed scripts in the download area[3].

[1] https://www.hex-rays.com/blog/igor-tip-of-the-week-08-batch-mode-under-the-hood/
[2] https://www.hex-rays.com/blog/implementing-command-completion-for-idapython/
[3] https://hex-rays.com/products/ida/support/download/

#33: IDA's user directory (IDAUSR)

02 Apr 2021

https://hex-rays.com/blog/igors-tip-of-the-week-33-idas-user-directory-idausr/

The user directory is a location where IDA stores some of the global settings and which can be used for some additional customization.

Default location

On Windows: %APPDATA%/Hex-Rays/IDA Pro

On Linux and Mac: $HOME/.idapro

For brevity, we'll refer to this path as `$IDAUSR` in the following text.

Contents/settings

The directory is used to store the processor module caches (`proccache.lst` and `proccache64.lst`) as well as the trusted database caches (`trusted_i64_list.bin` and `trusted_idb_list.bin`). Trusted databases are those that were authorized by the user to be run under debugger. The cache is used to prevent accidental execution of unknown binaries (for example, a database provided by a third party can contain a malicious executable path so it's not run without confirmation by default).

On Linux and Mac, the user directory also contains the pseudo registry file `ida.reg`. It holds global IDA settings which are stored in the registry on Windows (for example, the custom desktop layouts[1]).

If you modify or add shortcuts[2], modifications are stored in `shortcuts.cfg` in this directory.

Plugins

The user directory (more specifically, `$IDAUSR/plugins`) can be used for installing plugins instead of IDA's installation directory. This has several advantages:

1. No need for administrative permissions on Windows;
2. The plugins can be shared by multiple IDA installs or versions, so there's no need to reinstall plugins in new location when installing a new IDA version;
3. plugins in the user directory can override plugins with the same name in IDA's directory so this feature can be used to replace plugins shipped with IDA.

Both native (C++) and scripted (Python/IDC) plugins can be used this way.

Config files

To change some default options, you sometimes need to edit configuration files in IDA's `cfg` subdirectory (for example, `ida.cfg` or `hexrays.cfg`). Instead of editing them in-place, you can extract only the options you need to change and put them into the same-named file in `$IDAUSR/cfg`. Unlike the plugins, the config files don't override IDA's files completely but are applied additionally. For example, to enable synchronization and split view[3] for the decompiler, put the following lines in `$IDAUSR/cfg/hexrays.cfg`:

```
//--
PSEUDOCODE_SYNCED=YES
PSEUDOCODE_DOCKPOS=DP_RIGHT
//--
```

Other addons

The user directory can also be used to provide additional loaders, processor modules, type libraries and signatures. IDA will scan the following directories for them:

```
$IDAUSR/loaders
$IDAUSR/procs
$IDAUSR/til/{processor}
$IDAUSR/sig/{processor}
```

[1] https://www.hex-rays.com/blog/igors-tip-of-the-week-22-ida-desktop-layouts/
[2] https://www.hex-rays.com/blog/igor-tip-of-the-week-02-ida-ui-actions-and-where-to-find-them/
[3] https://twitter.com/HexRaysSA/status/1341745224037634049

#33: IDA's user directory (IDAUSR)

02 Apr 2021

https://hex-rays.com/blog/igors-tip-of-the-week-33-idas-user-directory-idausr/

IDAPython

If a file named `idapythonrc.py` is present in the user directory, it will be parsed and executed at the end of IDAPython's initialization. This allows you, for example, to add custom IDAPython functions, preload some commonly used scripts, or do any other customization that's more convenient to do in Python code.

Overriding the user directory location

If you prefer to use a custom location for user settings or need several sets of such directories, you can set the `IDAUSR` environment variable to another path (or even a set of paths) before running IDA.

Overriding the user directory location

If you copied files to the correct location but IDA does not seem to pick them up, you can use the `-z` commandline switch[4] to confirm that it's finding your file. For example, the following command line enables debug output of processing of all types of customizations (plugins, processor modules, loaders, FLIRT signatures, config files) and also copies the debug output to a log file:

```
ida -zFC -Lida.log file.bin
```

Among the output, you should see lines similar to following:

```
Scanning plugins directory C:\Users\Igor\AppData\Roaming\Hex-Rays\IDA Pro\plugins, for *.dll.
Scanning plugins directory C:\Users\Igor\AppData\Roaming\Hex-Rays\IDA Pro\plugins, for *.idc.
Scanning plugins directory C:\Program Files\IDA Pro 7.6\plugins, for *.dll.
Scanning plugins directory C:\Program Files\IDA Pro 7.6\plugins, for *.idc.
<...>
Scanning directory 'C:\Users\Igor\AppData\Roaming\Hex-Rays\IDA Pro\loaders' for loaders
```

So you can verify whether IDA is looking in the expected location.

For even more details on this feature, please check Environment variables[5] (IDAUSR section).

[4] https://www.hex-rays.com/blog/igor-tip-of-the-week-07-ida-command-line-options-cheatsheet/
[5] https://www.hex-rays.com/products/ida/support/idadoc/1375.shtml

#34: Dummy names

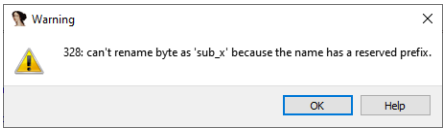
09 Apr 2021

https://hex-rays.com/blog/igors-tip-of-the-week-34-dummy-names/

In IDA's disassembly, you may have often observed names that may look strange and cryptic on first sight: `sub_73906D75`, `loc_40721B`, `off_40A27C` and more. In IDA's terminology, they're called dummy names. They are used when a name is required by the assembly syntax but there is nothing suitable available, for example the input file has no debug information (i.e. it has been stripped), or when referring to a location not present in the debug info. These names are not actually stored in the database but are generated by IDA on the fly, when printing the listing.

Dummy name prefixes

The dummy name consists of a type-dependent prefix and a unique suffix which is usually address-dependent. The following prefixes are used in IDA:



- `sub_` instruction, subroutine(function) start
- `locret_` a return instruction
- `loc_` other kind of instruction
- `off_` data, contains an offset(pointer) value
- `seg_` data, contains a segment address value
- `asc_` data, start of a string literal
- `byte_` data, byte
- `word_` data, 16-bit
- `dword_` data, 32-bit
- `qword_` data, 64-bit
- `byte3_` data, 3-byte
- `xmmword_` data, 128-bit
- `ymmword_` data, 256-bit
- `packreal_` data, packed real
- `flt_` floating point data, 32-bit
- `dbl_` floating point data, 64-bit
- `tbyte_` floating point data, 80-bit
- `stru_` structure
- `custdata_` custom data type
- `align_` alignment directive
- `unk_` unexplored (undefined, unknown) byte

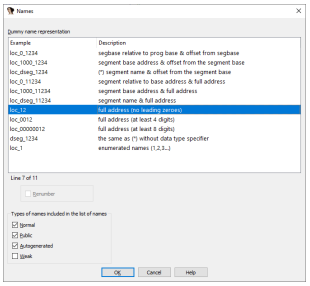
Because the prefixes are treated in a special way by IDA, they're reserved and cannot be used in user-defined names. If you try to use such a name, you'll get an error from IDA:

Warning 328: can't rename byte as 'sub_x' because the name has a reserved prefix.
Warning: can't rename byte because the name has a reserved prefix

A possible workaround is to add an underscore at the start so the prefix is different. But if you want to get rid of an existing name and have IDA use a dummy name again, just delete it (rename to an empty string).

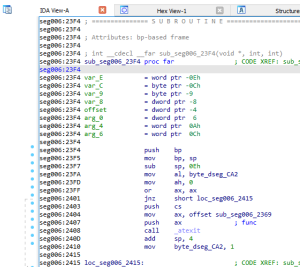
Name suffixes

The default suffix is the linear (aka effective) address of the item to which the dummy name is attached. However, this is not the only possibility. By using the Options > Name representation... dialog, you can choose something different.



Dummy name representation dialog

The options from the first half can be especially useful when dealing with segmented programs such as 16-bit DOS software; instead of a global linear address you can see the segment and the offset inside it so, for example, it is evident when the destination is in another segment.



DOS program when using "segment name & offset from the segment base" representation

#34: Dummy names

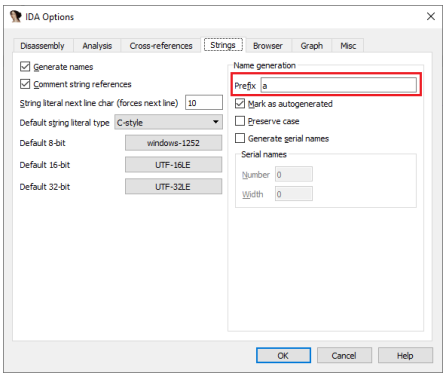
09 Apr 2021

https://hex-rays.com/blog/igors-tip-of-the-week-34-dummy-names/

Other prefixes

In addition to dummy names, there are two other kinds of autogenerated names that are used in IDA:

Stack variables (`var_`) and arguments (`arg_`).
String literal names generated from their text (e.g. `aException` for “exception”)
The stack prefixes are hardcoded and not configurable but the latter can be configured in Options > General..., Strings tab.



Strings options

Unlike the dummy names, these names are stored in the database marked as autogenerated so their prefixes are not considered reserved and you can use them in custom names.

#35: Demangled names

16 Apr 2021

https://hex-rays.com/blog/igors-tip-of-the-week-35-demangled-names/

Name mangling (also called **name decoration**) is a technique used by compilers to implement some of the features required by the language. For example, in C++ it is used to distinguish functions with the same name but different arguments (function overloading), as well as to support namespaces, templates, and other purposes.

Mangled names often end up in the final binary and, depending on the compiler, may be non-trivial to understand for a human (a simple example: "operator new" could be encoded as `??2@YAPAXI@Z` or `_Znwm`). While these cryptic strings can be decoded by a compiler-provided utility such as undname (MSVC) or `c++filt` (GCC/Clang), it's much better if the disassembler does it for you (especially if you don't have the compiler installed). This process of decoding back to a human-readable form is called **demangling**. IDA has out-of-box support for demangling names for the following compilers and languages:

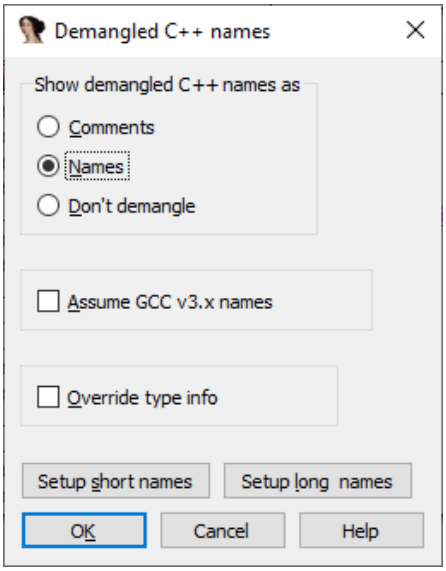
- Microsoft (Visual C++)
- Borland (C++, Pascal, C++ Builder, Delphi)
- Watcom (C++)
- Visual Age (C++)
- DMD (D language)
- GNU mangling (GCC, Clang, some commercial compilers)
- Swift

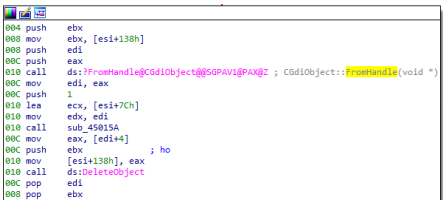
You do not need to pick the compiler manually; IDA will detect it from the name format and apply the corresponding demangler automatically.

Demangled name options

By default, IDA uses a comment to show the result of demangling, meaning that every time a mangled name is used, IDA will print a comment with the result of demangling. For example, `?FromHandle@CGdiObject@@SGPAV1@PAX@Z` demangles to `CGdiObject::FromHandle(void *)`, which is printed as a comment:

If you prefer, you can show the demangled result in place of the mangled name instead of just a comment. This can be done in the Options > Demangled names... dialog:





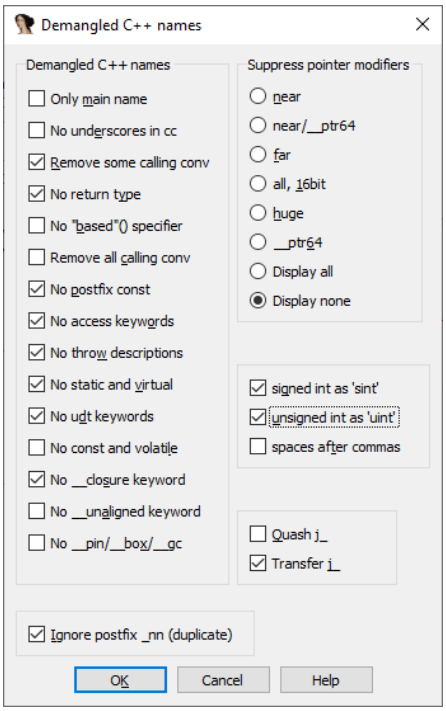
#35: Demangled names

16 Apr 2021

https://hex-rays.com/blog/igors-tip-of-the-week-35-demangled-names/

Short and long names

The buttons “Setup short names” and “Setup long names” allow you to modify the behavior of the built-in demangler in two common situations. The “short” names are used in contexts where space is at premium: references in disassembly, lists of functions and so on. “Long” names are used in other situations, for example when printing a comment at the start of the function. By using the additional options dialog, you can select what parts of the demangled name to show, hide, or shorten to make it either more compact or more verbose.



Name simplification

Some deceptively simple-looking names may end up very complicated after compilation, especially when templates are involved. For example, a simple `std::string`[1] from STL actually expands to

```
std::basic_string<char,std::char_traits<char>,std::allocator<char>>
```

To ensure interoperability, the compiler has to preserve these details in the mangled name, so they reappear on demangling; however, such implementation details are usually not interesting to a human reader who would prefer to see a simple std::string again. This is why IDA implements name simplification as a post-processing step. Using the rules in the file cfg/goodname.cfg, IDA applies them to transform a name like

```
std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > & __thiscall std::ba-
sic_string<char,struct std::char_traits<char>,class std::allocator<char> >::erase(unsigned int,unsigned int)
```

into

```
std::string & std::string::erase(unsigned int,unsigned int)
```

which is much easier to read and understand.

IDA ships with rules for most standard STL classes but you can add custom ones too. Read the comments inside `goodname.cfg` for the description of how to do it.

More info: Demangled names in IDA Help.

[1] https://en.cppreference.com/w/cpp/string
[2] https://www.hex-rays.com/products/ida/support/idadoc/611.shtml

#36: Working with list views in IDA

23 Apr 2021

https://hex-rays.com/blog/igors-tip-of-the-week-36-working-with-list-views-in-ida/

List views (also called choosers or table views) are used in many places in IDA to show lists of different kind of information. For example, the Function list[1] we've covered previously is an example of a list view. Many windows opened via the View > Open subviews menu are list views:

- Exports
- Imports
- Names
- Strings
- Segments
- Segment registers
- Selectors
- Signatures
- Type libraries
- Local types
- Problems
- Patched bytes

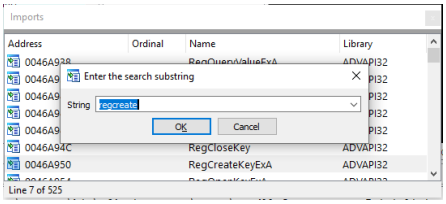
Many modal dialogs from the Jump menu (such as those for listing Cross references[2]) are also examples of list views. Because they are often used to select or choose one entry among many, they may also be called choosers.

List view can also be part of another dialog or widget, for example the shortcut list in the Shortcut editor[3]. These are called "embedded choosers" in the IDA SDK.

All list views share common features which we discuss below.

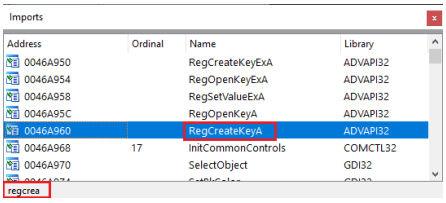
Text search

You can search for arbitrary text in the contents of the list view by using `Alt-T` to specify the search string and `Ctrl-T` to find the next occurrence.



Incremental search

Simply start typing to navigate to the closest item which starts with the typed text. The text will appear in the status bar. Use `Backspace` to erase incorrectly typed letters and `Ctrl-Enter` to jump to the next occurrence of the same prefix (if any).



Columns

Each list view has column headers at the top. In most (not all) of them, you can hide specific columns by using "Hide column" or "Columns..." from the context menu.

Similarly to the standard list views in most OSes, you can resize columns by dragging the delimiters between them or auto-size the column to fit the longest string in it by double-clicking the right delimiter.

[1] https://www.hex-rays.com/blog/igors-tip-of-the-week-28-functions-list/
[2] https://www.hex-rays.com/blog/igor-tip-of-the-week-16-cross-references/
[3] https://www.hex-rays.com/blog/igor-tip-of-the-week-02-ida-ui-actions-and-where-to-find-them/

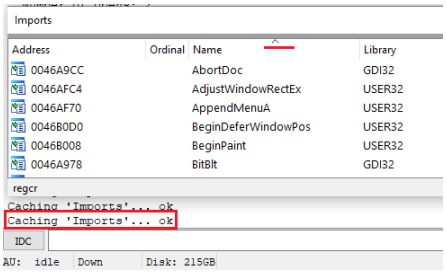
#36: Working with list views in IDA

23 Apr 2021

https://hex-rays.com/blog/igors-tip-of-the-week-36-working-with-list-views-in-ida/

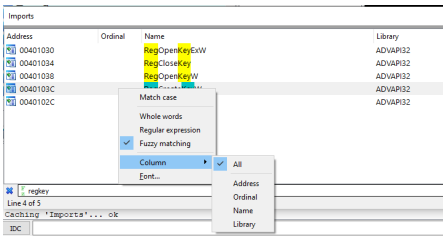
Sorting

The list view can be sorted by clicking on a column's header. The sorting indicator shows the direction of sorting (click it again to switch the direction). Because IDA needs to fetch the whole list of items to sort them, this can be slow in big lists so a reminder with the text "Caching <window>..." is printed in the Output window each time the list is updated and re-sorted. To improve the performance, you can disable sorting by using "Unsort" from the context menu.

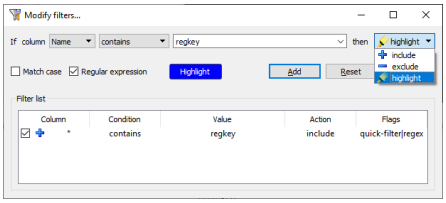


Filtering

A quick filter box can be opened by pressing `Ctrl-F`. Type some text in it to only show items which include the typed substring. By default it performs case-insensitive match on all columns, however you can modify some options from the context menu, such as:



- enable case-sensitive matching
- match only whole words instead of any substring
- enable fuzzy matching
- interpret the entered string as a regular expression
- pick a column on which to perform the matching



Instead of a quick filter, you can also use more complicated filtering ("Modify Filters" from context menu, or `Ctrl-Shift-F`). In this dialog you can not only include matching items, but also exclude or simply highlight them with a custom color.

Similarly to sorting, filtering requires fetching of the whole list which can slow down IDA, especially during autoanalysis. To remove any filters, choose "Reset filters" from the context menu.

See also: How To Use List Viewers in IDA[4]

[4] https://www.hex-rays.com/products/ida/support/idadoc/427.shtml

#37: Patching

30 Apr 2021

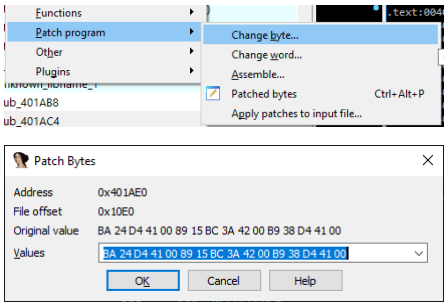
https://hex-rays.com/blog/igors-tip-of-the-week-37-patching/

Although IDA is mostly intended to be used for static analysis, i.e. simply looking at unaltered binaries, there are times you do need to make some changes. For example, you can use it to fix up some obfuscated instructions to clean up the code flow or decompiler output, or change some constants used in the program.

Patching bytes

Individual byte values can be patched via the Edit > Patch program > Change byte... command.

You can change up to 16 bytes at a time but you don't have to enter all sixteen – the remaining ones will remain unchanged.

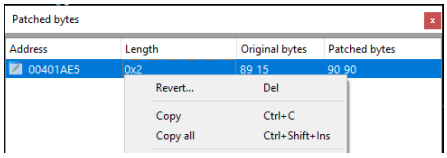


Assembling instructions

Edit > Patch program > Assemble... is available only for the x86 processor and currently only supports a subset of 32-bit x86 but it still may be useful in simple situations. For example, the `nop` instruction is the same in all processor mode so you can still use it to patch out unnecessary instructions.

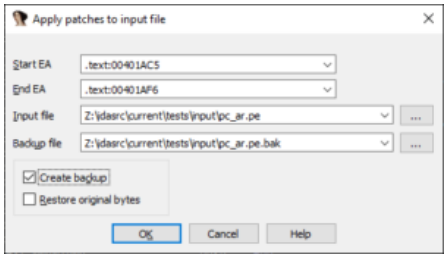
Patched bytes view

Available either under Edit > Patch program or in View > Open subviews submenus, this list view shows the list of the patched locations in the database and allows you to revert changes in any of them.



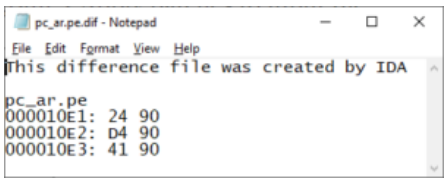
Patching the input file

All the patch commands only affect the contents of the database. The input file always remains unaffected by any change in the database. But in the rare case when you do need to update the input file on disk, you can use Edit > Patch program > Apply patches to input file...



Creating a difference file

File > Produce file > Create DIF File... outputs a list of patched location into a simple text file which can then be used to patch the input file manually in a hex editor or using a third party tool.



Patching during debugging

During debugging, patching still does not affect the input file, however it does affect the program memory if the location being patched belong to a currently mapped memory area. So you can, for example, change instructions or data to see how the program behaves in such situation.

30 Apr 2021
https://hex-rays.com/blog/igors-tip-of-the-week-37-patching/

Third party solutions

If the basic patching features do not quite meet your requirements, you can try the following third party plugins:

- IDA Patcher[1] by Peter Kacherginsky, a submission to our 2014 plugin contest[2]
- KeyPatch[3] by the Keystone Engine project, a winner of the 2016 contest[4]

See also: IDA Help: Edit|Patch core submenu[5]

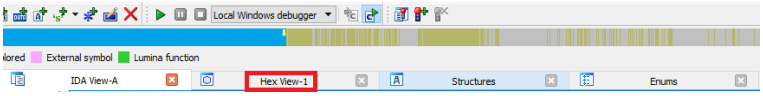
[1] https://github.com/iphelix/ida-patcher
[2] https://www.hex-rays.com/contests_details/contest2014/
[3] https://www.keystone-engine.org/keypatch/
[4] https://www.hex-rays.com/contests_details/contest2016/
[5] https://www.hex-rays.com/products/ida/support/idadoc/526.shtml

#38: Hex view

07 May 2021

https://hex-rays.com/blog/igors-tip-of-the-week-38-hex-view/

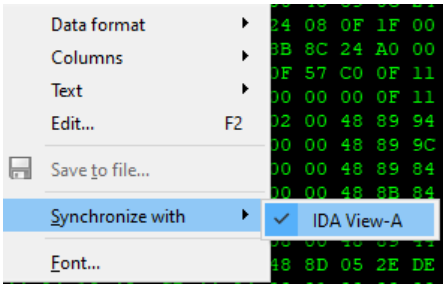
In addition to the disassembly and decompilation (Pseudocode) views, IDA also allows you to see the actual, raw bytes behind the program's instructions and data. This is possible using the Hex view, one of the views opened by default (or available in the View > Open subviews menu).
Even if you've used it before, there may be features you are not aware of.



Synchronization

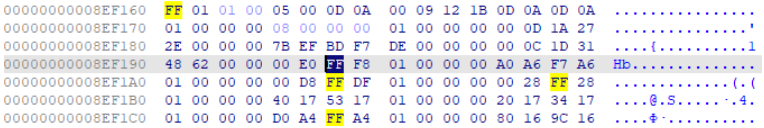
Hex view can be synchronized with the disassembly view (IDA View) or Pseudocode (decompiler) view. This option is available in the context menu under "Synchronize with".

Synchronization can also be enabled or disabled in the opposite direction (i.e. from IDA View or Pseudocode window). When it is on, the views' cursors move in lockstep: changing the position in one view updates it in the other.

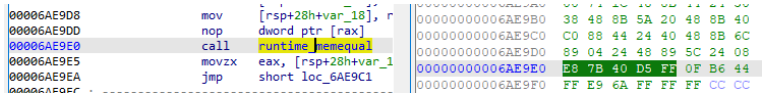


Highlight

There are two types of highlight available in the Hex view.
1. the text match highlight is similar to the one we've seen in the disassembly listing[1] and shows matches of the selected text anywhere on the screen.

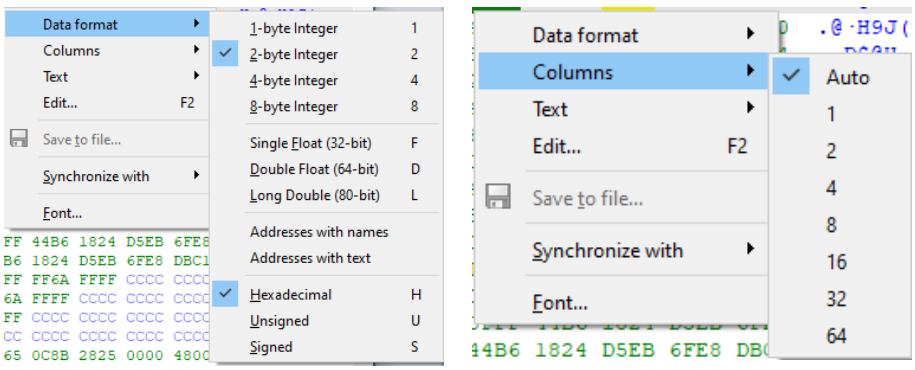


2. current item highlight shows the group of bytes that constitutes the current item (i.e. an instruction or a piece of data). This can be an alternative way to track the instruction's opcode bytes instead of the disassembly option.



Layout and data format

The default settings use the classic 16-byte lines with text on the right. You can change the format of individual items as well as the amount of items per line (either a fixed count or auto-fit).



[1] https://www.hex-rays.com/blog/igors-tip-of-the-week-26-disassembly-options-2/

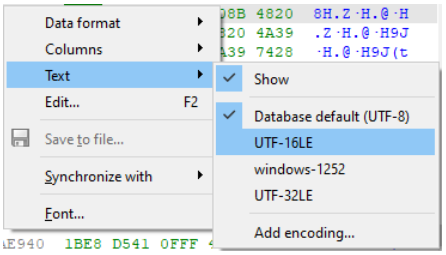
#38: Hex view

07 May 2021

https://hex-rays.com/blog/igors-tip-of-the-week-38-hex-view/

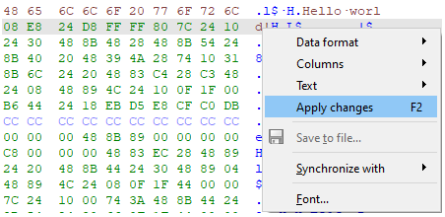
Text options

Text area at the right of the hex dump can be hidden or switched to another encoding if necessary.



Editing (patching)

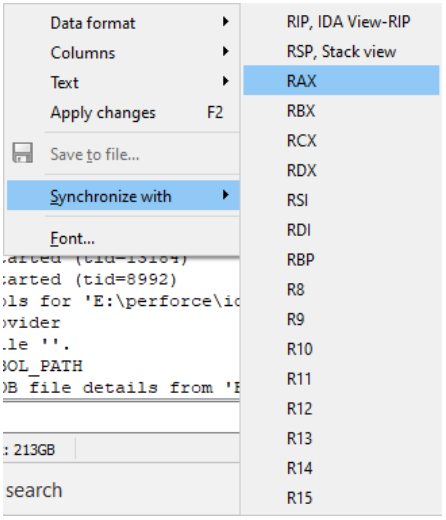
Hex view can be used as an alternative to the Patch program menu[2]. To start patching, simply press `F2`, enter new values and press `F2` again to commit changes (`Esc` to cancel editing). An additional advantage is that you can edit values in their native format (e.g. decimal or floating-point), or type text in the text area.

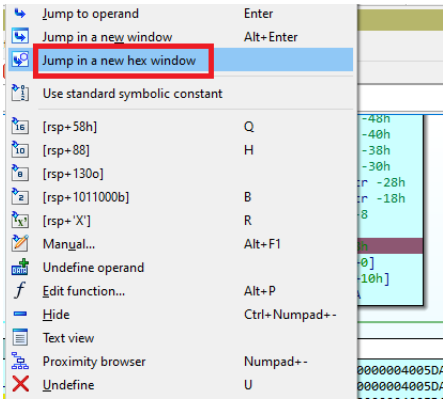


Debugging

Default debugging desktop has two Hex Views, one for a generic memory view and one for the stack view (synchronized to the stack pointer). Both are variants of the standard hex view and so the above-described functionality is available but there are a few additional features available only during debugging:

1. Synchronization is possible not only with other views but also with a value of a register. Whenever the register changes, the position in the hex view will be updated to match (as long as it is a valid address).
2. A new command in the disassembly view's context menu allows to open a hex view at the address of the operand under cursor.





[2] https://www.hex-rays.com/blog/igors-tip-of-the-week-37-patching/

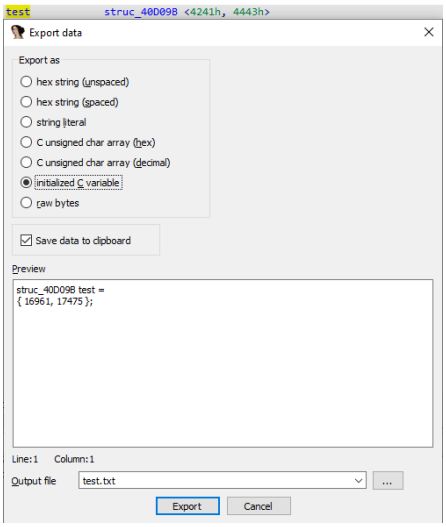
#39: Export Data

14 May 2021

https://hex-rays.com/blog/igors-tip-of-the-week-39-export-data/

The Edit > Export Data command (`Shift+E`) offers you several formats for extracting the selected data from the database:

- hex string (unspaced): 4142434400
- hex string (spaced): 41 42 43 44 00
- string literal: ABCD
- C unsigned char array (hex): unsigned char aAbcd[] = { 0x41, 0x42, 0x43, 0x44, 0x00 };
- C unsigned char array (decimal): unsigned char aAbcd[] = { 65, 66, 67, 68, 0 };
- initialized C variable: struc_40D09B test = { 16961, 17475 }; NB: this option is valid only in some cases, such as for structure instances or items with type information.
- raw bytes [can be only saved to file]



Data in the selected format is shown in the preview text box which can be copied to the clipboard or saved to a file for further processing.

#40: Decompiler basics

21 May 2021

https://hex-rays.com/blog/igors-tip-of-the-week-40-decompiler-basics/

The Hex-Rays decompiler is one of the most powerful add-ons available for IDA. While it's quite intuitive once you get used to it, it may be non-obvious how to start using it.

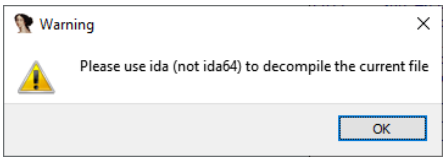
Basic information

As of the time of writing (May 2021), the decompiler is not included with the standard IDA Pro license; some editions of IDA Home and IDA Free include a cloud decompiler, but the offline version requires IDA Pro and must be purchased separately.
The following decompilers are currently available:

- x86 (32-bit)
- x64 (64-bit)
- ARM (32-bit)
- ARM64 (64-bit)
- PPC (32-bit)
- PPC64 (64-bit)
- MIPS (32-bit)

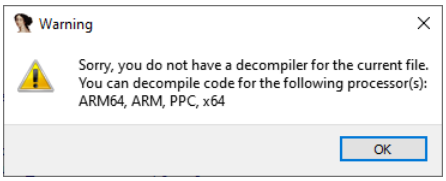
Pick the matching IDA

The decompiler must be used with the matching IDA: 32-bit decompilers only work with 32-bit IDA (e.g. `ida.exe`) while 64-bit ones require ida64. If you open a 32-binary in `IDA64` and press **F5**, you'll get a warning:



Warning: Please use ida (not ida64) to decompile the current file

If you try to decompile a file for which you do not have a decompiler, a different error is displayed:



Invoking the decompiler

The decompiler can be invoked in the following ways:

1. View > Open subviews > Generate pseudocode (or simply **F5**). This always opens a new pseudocode view (up to 26);
2. **Tab** switches to the last active pseudocode view and decompiles current function. If there are none, a new view is opened just like with **F5**.
Tab can also be used to switch from pseudocode back to the disassembly. Whenever possible, it tries to jump to the corresponding location in the other view.
3. Full decompilation of the whole database can be requested via File > Produce file > Create C file... (hotkey **Ctrl+F5**). This command decompiles selected or all functions in the database (besides those marked as library functions) and writes the result to a text file.

Changing options

Because of its origins as a standalone plugin, the decompiler's options are not currently present in the Options menu but are accessed via Edit > Plugins > Hex-Rays Decompiler.

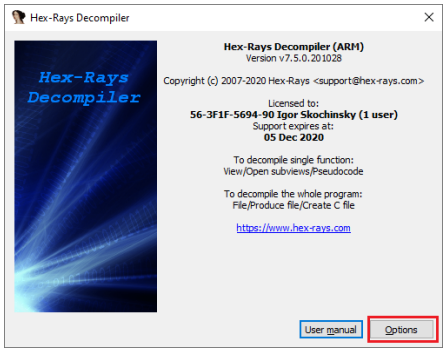
This dialog changes options for the current database. To change them for all future files, edit `cfg/hexrays.cfg`. Instead of editing the file in IDA's directory, you can create one with only changed options in the user directory[1]. The available options are explained in the manual[2].

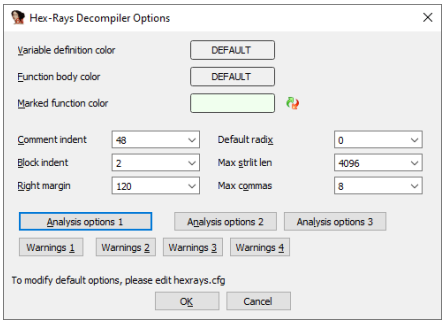
[1] https://www.hex-rays.com/blog/igors-tip-of-the-week-33-idas-user-directory-idausr/
[2] https://www.hex-rays.com/products/decompiler/manual/config.shtml

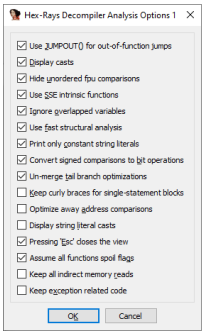
#40: Decompiler basics

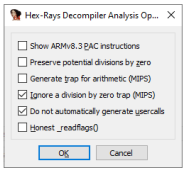
21 May 2021

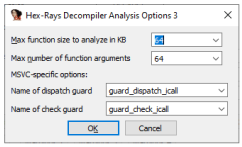
https://hex-rays.com/blog/igors-tip-of-the-week-40-decompiler-basics/











#41: Binary file loader

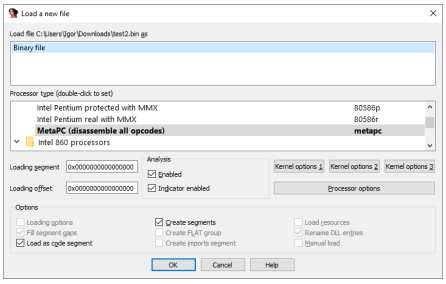
28 May 2021

https://hex-rays.com/blog/igors-tip-of-the-week-41-binary-file-loader/

IDA supports more than 40 file formats out of box. Most of them are structured file formats – with defined headers and meta-data – so they're recognized and handled automatically by IDA. However, there are times when all you have is just a piece of a code without any headers (e.g. shellcode or raw firmware) which you want to analyze in IDA. In that case, you can use the binary loader. It is always available even if the file is recognized as another file format.

Processor selection

Since raw binaries do not have metadata, IDA does not know which processor module to use for it, so you should pick the correct one. By default, the **metapc** (responsible for x86 and x64 disassembly) is selected, but you can choose another one from the list (double-click to change).

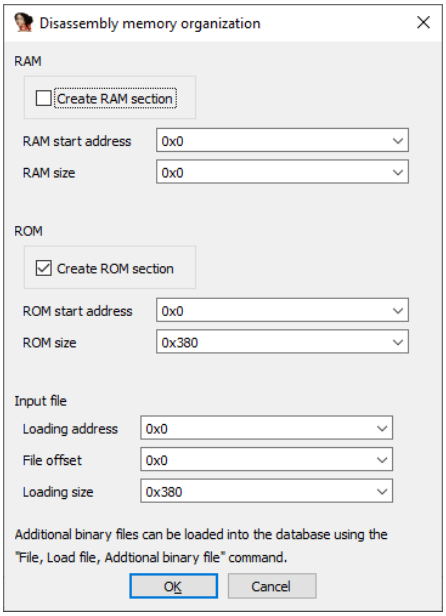


Memory loading address

Without metadata, IDA also does not know at which address to place the loaded data, so you may need to help it. The *Loading segment* and *Loading offset* fields are valid for the x86 family only. If the code being loaded uses a flat memory model (such as 32-bit protected mode or 64-bit long mode), Loading segment should be left at 0 and the address specified in the Loading offset field.

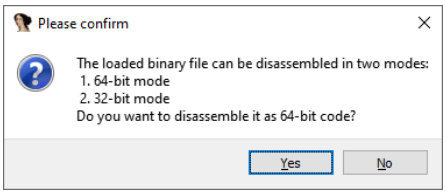
Other processors such as ARM, MIPS, or PPC, do not use these fields but prompt for memory layout after you confirm the initial selection.

In this dialog you can specify where to place the data and whether to create an additional RAM section. By default the whole file is placed at address 0 in the ROM segment but you can specify a different one or load only a part of the file by changing the file offset and loading size.



Code bitness

For processors where instruction decoding changes depending on current mode, such as PC (16-bit mode, 32-bit protected mode, or 64-bit long mode) or ARM (AArch32 or AArch64), you may get one more additional question.



#41: Binary file loader

28 May 2021

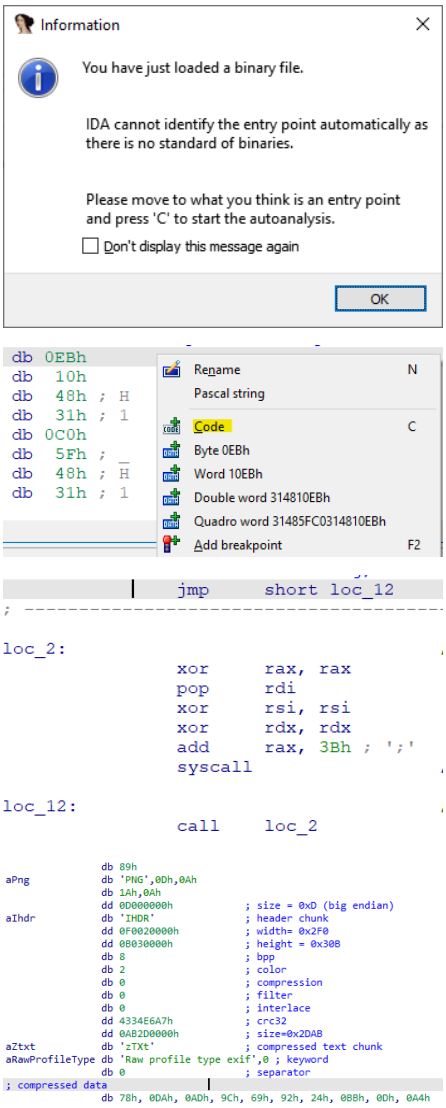
https://hex-rays.com/blog/igors-tip-of-the-week-41-binary-file-loader/

Start disassembling

Finally, the file is loaded, but IDA can't decide how to disassemble it on its own.

As suggested by the dialog, you can use `C` (make code) to try decoding at locations which look like valid instructions. Typically, shellcode will have valid instructions at the beginning, and firmware for most processors either starts at the lowest address or uses a vector table (a list of addresses) pointing to code.

In addition to shellcode or firmware, the binary file loader can be used to analyze other kinds of files using IDA's powerful features for marking up and labeling data and code. For example, here's a PNG file labeled and commented in IDA:



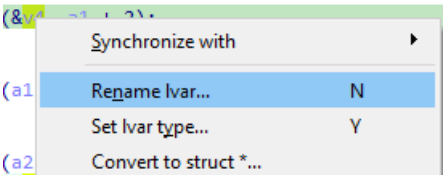
#42: Renaming and retyping in the decompiler

04 Jun 2021

https://hex-rays.com/blog/igors-tip-of-the-week-42-renaming-and-retyping-in-the-decompiler/

Previously we've covered how to start using the decompiler[1], but unmodified decompiler output is not always easy to read, especially if the binary doesn't have symbols or debug information. However, with just a few small amendments you can improve the results substantially. Let's look at some basic interactive operations available in the pseudocode view.

Text input dialog boxes (e.g. Enter Comment or Edit Local Type)



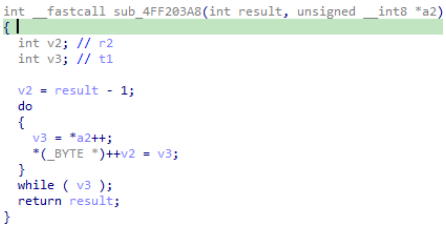
Although it sounds trivial, renaming can dramatically improve readability. Even something simple like renaming of `v3` to `counter` can bring immediate clarity to what's going on in a function. Coupled with the auto-renaming feature added in IDA 7.6[2], this can help you propagate nice names through pseudocode as you analyze it. The following items can be renamed directly in the pseudocode view:

- local variables
- function arguments
- function names
- global variables (data items)
- structure members

Renaming is very simple: put the cursor on the item to rename and press `N` – the same shortcut as the one used in the disassembly listing. Of course, the command is also available in the context menu.

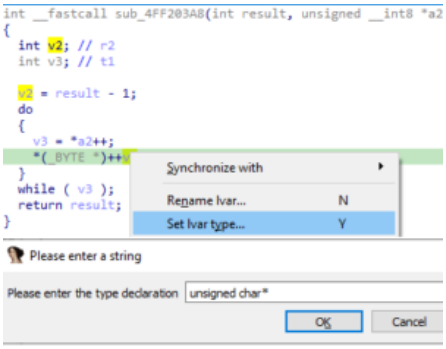
You can also choose to do your renaming in the disassembly view instead of pseudocode. This can be useful if you plan to rename many items in a big function and don't want to wait for decompilation to finish every time. Once you finished renaming, press `F5` to refresh the pseudocode and see all the new names. Note that register-allocated local variables cannot be renamed in the disassembly; they can only be

Retyping



```
int __fastcall sub_4FF203A8(int result, unsigned __int8 *a2)
{
  int v2; // r2
  int v3; // t1

  v2 = result - 1;
  do
  {
    v3 = *a2++;
    *(_BYTE *)++v2 = v3;
  }
  while ( v3 );
  return result;
}
```



Type recovery is one of the hardest problems in decompilation. Once the code is converted to machine instructions, there are no more types but just bits which are being shuffled around. There are some guesses the decompiler can make nevertheless, such as a size of the data being processed, and in some cases whether it's being treated as a signed value or not, but in general the high-level type recovery remains a challenge in which a human brain can be of great help.

For example, consider this small ARM function:

```
sub_4FF203A8
  SUB R2, R0, #1
loc_4FF203AC
  LDRB R3, [R1],#1
  CMP R3, #0
  STRB R3, [R2,#1]!
  BNE loc_4FF203AC
  BX LR
```

Its initial decompilation looks like this:

We see that the decompiler could guess the type of the second argument (`a2`, passed in `R1`) because it is used in the `LDRB` instruction (load byte). However, `v2` remains a simple `int` because the first operation done on it is a simple arithmetic `SUB` (subtraction). Now, after some thinking it is pretty obvious that both `v2` and `result` are also byte pointers and the subtraction is simply pointer math (since pointers are just numbers on the CPU level).

We can fix things by changing the type of both variables to the same `unsigned __int8 *` (or the equivalent `unsigned char *`). To do this, put cursor on the variable and press `Y`, or use "Set lvar type" from the context menu.

[1] https://hex-rays.com/blog/igors-tip-of-the-week-40-decompiler-basics/
[2] https://hex-rays.com/products/ida/news/7_6/

#42: Renaming and retyping in the decompiler

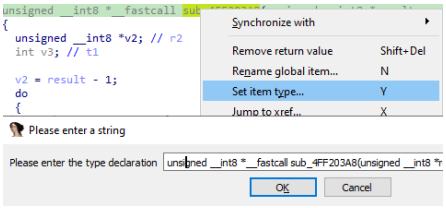
04 Jun 2021

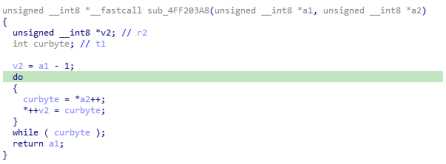
https://hex-rays.com/blog/igors-tip-of-the-week-42-renaming-and-retyping-in-the-decompiler/

Alternatively, instead of fixing the local variable and then the argument, you can directly edit the function prototype by using the shortcut on the function's name in the first line.

In that case, first argument's type will be automatically propagated into the local variable and you won't need to change it manually (user-provided types have priority over guessed ones).

In the final version there are no more casts and it's clearer what's happening. We'll solve the mystery of the function's purpose next week, stay tuned!





#43: Annotating the decompiler output

11 Jun 2021

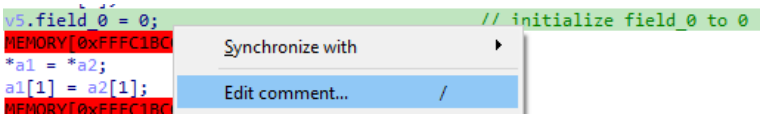
https://hex-rays.com/blog/igors-tip-of-the-week-43-annotating-the-decompiler-output/

Last week[1] we started improving decompilation of a simple function. While you can go quite far with renaming and retyping, some things need more explanation than a simple renamng could provide.

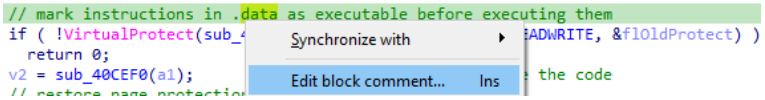
Comments

When you can't come up with a good name for a variable or a function, you can add a comment with an explanation or a theory about what's going on. The following comment types are available in the pseudocode:

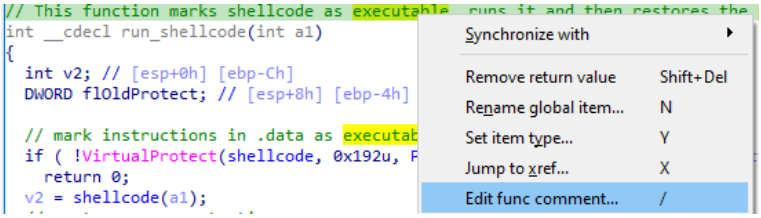
1. Regular end-of-line comments. Use `/` to add or edit them (easy to remember because in C++ `//` is used for comments).



2. Block comments. Similarly to anterior comments[2] in the disassembly view, the `Ins` shortcut is used (`I` on Mac). The comment is added before the current statement (not necessarily the current line).

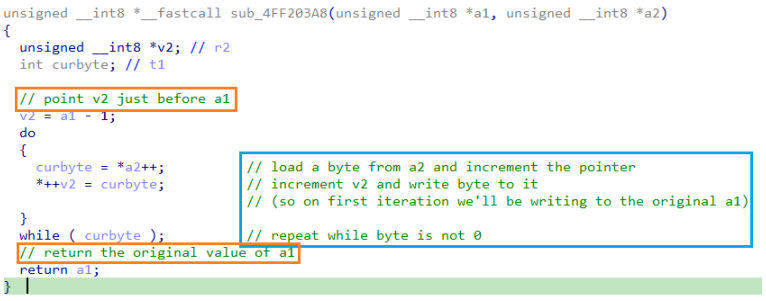


3. Function comment is added when you use `/` on the first line of the function.



Due to limitations of the implementation[3], the first two types can move around or even end up as orphan comments when the pseudo-code changes. The function comment is attached to the function itself and is visible also in the disassembly view.

Using the comments, we can annotate the function from the previous post[4] to clarify what is going on. On the screenshot below, regular comments are highlighted in blue while block comments are outlined in orange.



In the end, the function seems to be copying bytes from a2 to a1, stopping at the first zero byte. If you know libc, you'll quickly realize that it's actually a trivial implementation of `strcpy`[5]. We can now rename the function and arguments to the canonical names and add a function comment explaining the purpose of the function.

[1] https://hex-rays.com/blog/igors-tip-of-the-week-42-renaming-and-retyping-in-the-decompiler/
[2] https://hex-rays.com/blog/igor-tip-of-the-week-14-comments-in-ida/
[3] https://hex-rays.com/blog/coordinate-system-for-hex-rays/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-42-renaming-and-retyping-in-the-decompiler/
[5] https://en.cppreference.com/w/c/string/byte/strcpy

#43: Annotating the decompiler output

11 Jun 2021

https://hex-rays.com/blog/igors-tip-of-the-week-43-annotating-the-decompiler-output/

```
// Copies the null-terminated byte string pointed to by src,
// including the null terminator, to the character array
// whose first element is pointed to by dest.
char *__fastcall strcpy(char *dest, const char *src)
{
  char *v2; // r2
  int curbyte; // t1

  // point v2 just before a1
  v2 = dest - 1;
  do
  {
    curbyte = *(unsigned __int8 *)src++;// load a byte from a2 and increment the pointer
    *++v2 = curbyte;          // increment v2 and write byte to it
                              // (so on first iteration we'll be writing to the original a1)
  }
  while ( curbyte );          // repeat while byte is not 0
  // return the original value of a1
  return dest;
}
```

Alas, the existing comments are not updated automatically, so references to `a1` and `a2` would have to be fixed manually.

Empty lines

To improve the readability of pseudocode even further, you can add empty lines either manually or automatically. For manual lines, press **`Enter`** after or before a statement. For example, here's the same function with extra empty lines added:

```
char *__fastcall strcpy(char *dest, const char *src)
{
  char *v2; // r2
  int curbyte; // t1

  // point v2 just before a1
  v2 = dest - 1;
  do
  {
    curbyte = *(unsigned __int8 *)src++;// load a byte from a2 and increment the pointer

    *++v2 = curbyte;          // increment v2 and write byte to it
                              // (so on first iteration we'll be writing to the original a1)
  }
  while ( curbyte );          // repeat while byte is not 0

  // return the original value of a1
  return dest;
}
```

To remove the manual empty lines, edit the anterior comment (**`Ins`** or **`I`** on Mac) and remove the empty lines from the comment.

To add automatic empty lines, set `GENERATE_EMPTY_LINES = YES` in `hexrays.cfg`. This will cause the decompiler to add empty lines between compound statements as well as before labels. This improves readability of long or complex functions. For example, here's a decompilation of the same function with both settings. You can see that the second one reads easier thanks to extra spacing.

```
    }
  }
  if ( v23 == 61 )
  {
    ++v22;
    *v28 = 0;
  }
 }
 v19 = v22 + 1;
 *v22 = 0;
 if ( !nvars || drop_var_from_set(v13, nvars, (unsigned __int8 **)dest)
   hdelete_r(v13, htab, flag);
 }
LABEL_23:
 if ( v19 >= enva || !*v19 )
  break;
LABEL_25:
 v13 = v19;
}
v37 = 0;
free(v12);
v38 = (char *)dest - 4;
while ( v37 < nvars )
```

Default

```
  if ( v23 == 61 )
  {
    ++v22;
    *v28 = 0;
  }
 }

 v19 = v22 + 1;
 *v22 = 0;
 if ( !nvars || drop_var_from_set(v13, nvars, (unsigned __int8 **)dest)
   hdelete_r(v13, htab, flag);
 }

LABEL_23:
 if ( v19 >= enva || !*v19 )
   break;

LABEL_25:
 v13 = v19;
}

v37 = 0;
free(v12);
```

GENERATE_EMPTY_LINES = YES

[1] https://hex-rays.com/wp-content/static/products/ida/idapro_cheatsheet.html
[2] https://hex-rays.com/wp-content/static/products/ida/support/freefiles/IDA_Pro_Shortcuts.pdf

#44: Hex dump loader

18 Jun 2021

https://hex-rays.com/blog/igors-tip-of-the-week-44-hex-dump-loader/

IDA has a file loader named 'hex' which mainly supports loading of text-based file formats such as Intel Hex[1] or Motorola S-Record[2]. These formats contain records with addresses and data in hexadecimal encoding.

For example, here's a fragment of an Intel Hex file:

```
:18000000008F9603008FD801008FDC01008FE001008FE401008FE80190
:20004000008FEC01008FF001008FF401008FF801008FFC01008F0002008F0402008F08024D
:20006000008F0C02008F1002008F1402008F1802008F1C02008F2002008F2402008F280228
:14008000008F2C02008F3002008F3402008F3802008F3C0293
:1000A000008F4002008F4402008F4802008F4C02F4
:20010000008F5002008F5402008F5802008F5C02008F6002008F6402008F680243204C694C
:200120006272617279204 36F70797269676874202843292031393932048492D5445434818
```

or an S-Record

```
S0030000FC
S1230100810F0016490F0016816F8A0A0F0000009830 0016B2310016BC3300168E0D0016A7
S1230108280F00169A2900168A00F0018660000804000000182300167922001 60C00000032
S12301109800E00182A09E0B8000C2012A380016080000 00EA3100163A380016FA310016CA
S1230118FF250016BE2100160000000 00182200169A0100169C330016F9C010010D000000D7
```

However, you may also have a simple unformatted hex dump, with or without addresses:

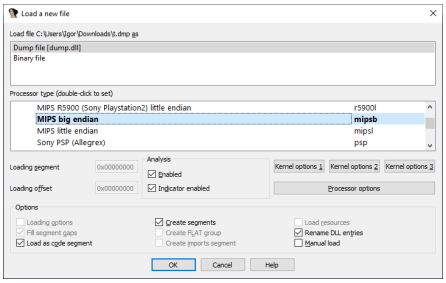
```
0020: 59 69 74 54 55 B6 3E F7 D6 B9 C9 B9 45 E6 A4 52
1000: 12 23 34 56 78
0100: 31 C7 1D AF 32 04 1E 32 05 1E 3C 32 07 1E 21 D9
12 23 34 56 78
```

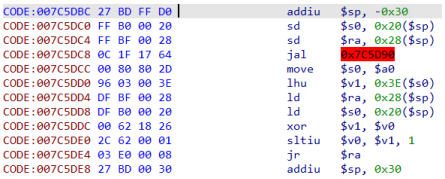
Such files are recognized and handled by another loader called 'dump'. Since, like raw binaries, they do not carry information about the processor used, it has to be selected by the user.

For example, a hex dump of some MIPS code:

```
007C5DBC 27 BD FF D0
007C5DC0 FF B0 00 20
007C5DC4 FF BF 00 28
007C5DC8 0C 1F 17 64
007C5DCC 00 80 80 2D
007C5DD0 96 03 00 3E
007C5DD4 DF BF 00 28
007C5DD8 DF B0 00 20
007C5DDC 00 62 18 26
007C5DE0 2C 62 00 01
007C5DE4 03 E0 00 08
007C5DE8 27 BD 00 30
```

can be loaded into IDA without having to convert it to binary or a structured format like ELF.





This feature could be useful when working with shellcode or exchanging data with other software. As we described before, IDA also supports exporting data from database[3] as hexadecimal dump.

[1] https://en.wikipedia.org/wiki/Intel_HEX
[2] https://en.wikipedia.org/wiki/SREC_(file_format)
[3] https://hex-rays.com/blog/igors-tip-of-the-week-39-export-data/

#45: Decompiler types

25 Jun 2021

https://hex-rays.com/blog/igors-tip-of-the-week-45-decompiler-types/

In one of the previous posts, we've discussed how to edit types of functions and variables[1] used in the pseudocode. In most cases, you can use the standard C types: `char`, `int`, `long` and so on. However, there may be situations where you need a more specific type. Decompiler may also generate such types itself so recognizing them is useful. The following custom types may appear in the pseudocode or used in variable and function types:

Explicitly-sized integer types

- `__int8` – 1-byte integer (8 bits)
- `__int16` – 2-byte integer (16 bits
- `__int32` – 4-byte integer (32 bits)
- `__int64` – 8-byte integer (64 bits)
- `__int128` – 16-byte integer (128 bits)

Explicitly-sized boolean types

- `_BOOL1` – boolean type with explicit size specification (1 byte)
- `_BOOL2` – boolean type with explicit size specification (2 bytes)
- `_BOOL4` – boolean type with explicit size specification (4 bytes)

Regardless of size, values of these types are treated in the same way: 0 is considered `false` and all other values `true`.

Unknown types

- `_BYTE` – unknown type; the only known info is its size: 1 byte
- `_WORD` – unknown type; the only known info is its size: 2 bytes
- `_DWORD` – unknown type; the only known info is its size: 4 bytes
- `_QWORD` – unknown type; the only known info is its size: 8 bytes
- `_OWORD` – unknown type; the only known info is its size: 16 bytes
- `_TBYTE` – 10-byte floating point (x87 extended precision 80-bit value)
- `_UNKNOWN` – no info is available about type or size (usually only appears in pointers)

Please note that these types are not equivalent to the similarly-looking Windows data types[2] and may appear in non-Windows programs.

More info: Set function/item type[3] in IDA Help.

[1] https://hex-rays.com/blog/igors-tip-of-the-week-42-renaming-and-retyping-in-the-decompiler/
[2] https://docs.microsoft.com/en-us/windows/win32/winprog/windows-data-types
[3] https://hex-rays.com/products/ida/support/idadoc/1361.shtml

#46: Disassembly operand representation

02 Jul 2021

https://hex-rays.com/blog/igors-tip-of-the-week-46-disassembly-operand-representation/

As we've mentioned before, the I in IDA stands for interactive, and we already covered some of the disassembly view's interactive features like renaming[1] or commenting[2]. However, other changes are possible too. For example, you can change the *operand representation* (sometimes called operand type in documentation). What is it about?

Most assemblers (and disassemblers) represent machine instructions using a mnemonic (which denotes the basic function of the instruction) and operands on which it acts (commonly delimited by commas). As an example, let's consider the most common x86 instruction `mov`, which copies data between two of its operands. A few examples:

`mov rsp, r11` – copy the value of `r11` to `rsp`

`mov rcx, [rbx+8]` – copy a 64-bit value from the address equal to value of the register `rbx` plus 8 to `rcx` (C-like equivalent: `rcx = *(int64*)(rbx+8);`)

`mov [rbp+390h+var_380], 2000000h` – copy the value `2000000h` (0x2000000 in C notation) to the stack variable `var_380`

The first example uses two registers as operands, the second a register and an indirect memory operand with base register and displacement, the third – another memory operand as well as an immediate (a constant value encoded directly in the instruction's opcode).

The last two examples are interesting because they involve numbers (displacements and immediates), and the same number can be represented in multiple ways. For example, consider the following instructions:

```
mov eax, 64h
mov eax, 100
mov eax, 144o
mov eax, 1100100b
mov eax, 'd'
mov eax, offset byte_64
mov eax, mystruct.field_64
```

All of them have exactly the same byte sequence (machine code) on the binary level: `B8 64 00 00 00`. So, while picking another operand representation may change the visual aspect, the underlying value and the program behavior **does not change**. This allows you to choose the best variant which represents the intent behind the code without having to add a long explanation in comments.

The following representations are available in IDA for numerical operands (some of them may only make sense in specific situations):

1. Default number representation (aka **void)**: used when there is no specific override applied on the operand (either by the user or IDA's autoanalyzer or the processor module). The actually used representation depends on the processor module but the most common fallback is hexadecimal. Uses **orange color** in the default color scheme. For values which match a printable character in the current encoding, a comment with the character could be displayed (depends on the processor module). Hotkey: `#` (hash sign).

```
mov     eax, 0C8h ; 'È'
mov     eax, 0C8h ; 'È'
mov     eax, 64h ; 'd'
mov     eax, 0C8h ; 'È'
```

2. Decimal: shows the operand as a decimal number. Hotkey is `H`.
3. Hexadecimal: explicitly show the operand as hexadecimal. Hotkey is `Q`.
4. Binary: shows the operand as a binary number. Hotkey is `B`.
5. Octal: shows the operand as an octal number. No default hotkey but can be picked from the context menu or the "Operand type" toolbar.
6. Character: shows the operand as a character constant if possible. Hotkey: `R`.
7. Structure offset: replaces the numerical operand with a reference to a structure member with a matching offset. Hotkey: `T`.
8. Enumeration (symbolic constant): the number is replaced by a symbolic constant with the same value. Hotkey: `M`.
9. Stack variable: the number is replaced by a symbolic reference into the current function's stack frame. Usually only makes sense for instructions involving stack pointer or frame pointer. Hotkey: `Kt`.

[1] https://hex-rays.com/blog/igors-tip-of-the-week-24-renaming-registers/
[2] https://hex-rays.com/blog/igor-tip-of-the-week-14-comments-in-ida/

#46: Disassembly operand representation

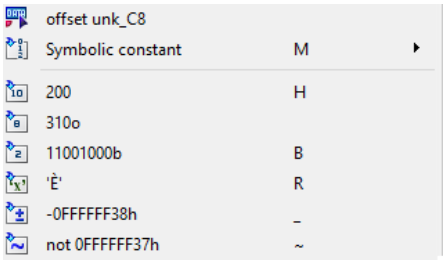
02 Jul 2021

https://hex-rays.com/blog/igors-tip-of-the-week-46-disassembly-operand-representation/

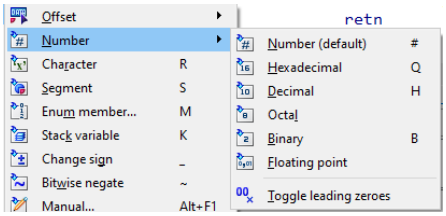
10. Floating-point constant: only works in some cases and for some processors. For example, `3F000000h`(`0x3F000000`) is actually an IEEE-754 encoding of the number `0.5`. There is no default hotkey but the conversion can be performed via the toolbar or main menu.
11. Offset operand: replace the number by an expression involving one or more addresses in the program. Hotkeys: **`O`**, **`Ctrl-O`** or **`Ctrl-R`** (for complex offsets).

All hotkeys revert to the default representation if applied twice.

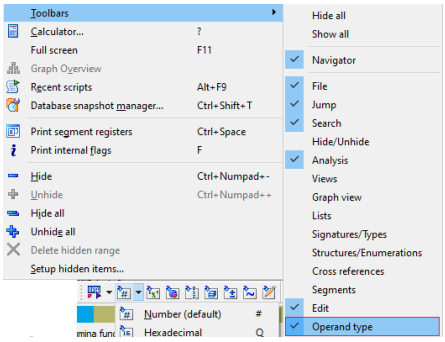
In addition to the hotkeys, the most common conversions can be done via the context menu:



The full list is available in the main menu (Edit > Operand Type):



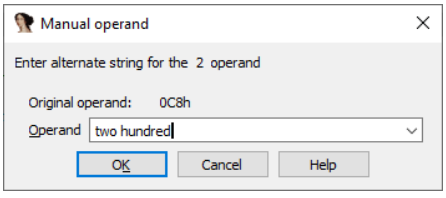
as well as the “Operand Type” toolbar:



Two more transformations can be applied to an operand on top of changing its numerical base:

1. Negation. Hotkey _(underscore). Can be used, for example, to show `-8` instead of `0FFFFFFF8h` (two representations of the same binary value).
2. Bitwise negation (aka inversion or binary NOT). Hotkey: ~(tilde). For example, `0FFFFFFF8h` is considered to be the same as `not 7`.

Finally, if you want to see something completely custom which is not covered by the existing conversions, you can use a manual operand. This allows you to replace the operand by an arbitrary text; it is not checked by IDA so it’s up to you to ensure that the new representation matches the original value. Hotkey: **`Alt-F1`**.



#47: Hints in IDA

09 Jul 2021

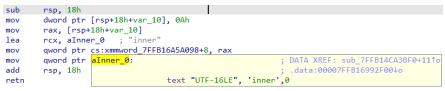
https://hex-rays.com/blog/igors-tip-of-the-week-47-hints-in-ida/

Hints (aka tooltips) are popup windows with text which appear when you hover the mouse cursor over a particular item in IDA. They are available in many situations.

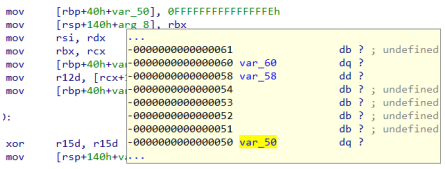
Disassembly hints

In the disassembly view, hints can be shown in the following cases:

1. When hovering over names or addresses, a fragment of disassembly at the destination is shown.

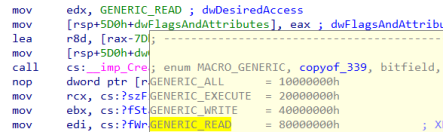


2. When hovering over stack variables, a fragment of the stack frame layout is shown

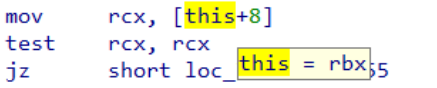


3. When hovering over structure offset operands, the fragment of the struct definition.

4. For enum operands – the enum with the definition.



5. For renamed registers[1], the hint shows the original register name

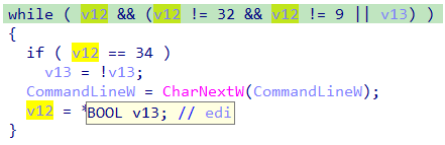


All these hints except the last one can be expanded or shrunk using the **mouse wheel**.

Decompiler hints

In the pseudocode, the hints are shown for:

1. Local variables and current function arguments: type and location (register or stack).



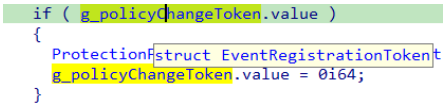
[1] https://hex-rays.com/blog/igors-tip-of-the-week-24-renaming-registers/

#47: Hints in IDA

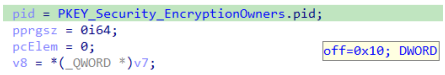
09 Jul 2021

https://hex-rays.com/blog/igors-tip-of-the-week-47-hints-in-ida/

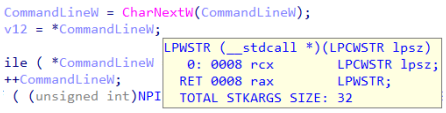
2. global variables: type.



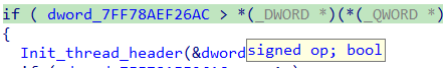
3. structure or union members: member type and offset.



4. function calls: prototype and information about arguments and return value.



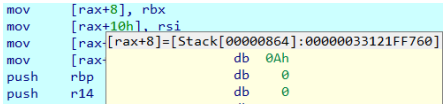
5. other expressions and operators: type, signedness, etc.



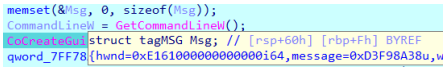
Debugger hints

During debugging, the hints behave mostly in the same way but with addition of dynamic information:

1. In the disassembly view, hovering on instruction operands shows a hint with their values and, if the value resolves to a valid address, a fragment of memory at that address.

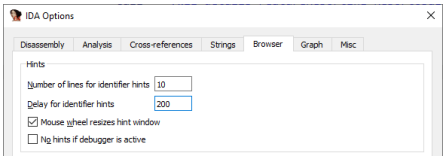


2. In pseudocode, values of variables are shown in hints.



Configuring hints

The way hints work can be configured via Options > General..., Browser tab. You can set how many lines are displayed by default and the delay before the hint is shown. The hints can be disabled completely by setting the number of lines to 0, or only disabled during the debugging (showing the hint during debugging may lead to memory reads which can be slow in some situations).



#48: Searching in IDA

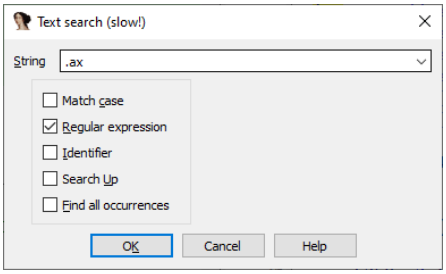
16 Jul 2021

https://hex-rays.com/blog/igors-tip-of-the-week-48-searching-in-ida/

We covered how to search for things in choosers (list views)[1], but what if you need to look for something elsewhere in IDA?

Text search

When searching for textual content, the same shortcut pair (**Alt**-T to start, **Ctrl-T** to continue) works almost anywhere IDA shows text:



- Disassembly (IDA View)
- Hex View
- Decompiler output (Pseudocode)
- Output window
- Structures and Enums windows
- Choosers (list views)

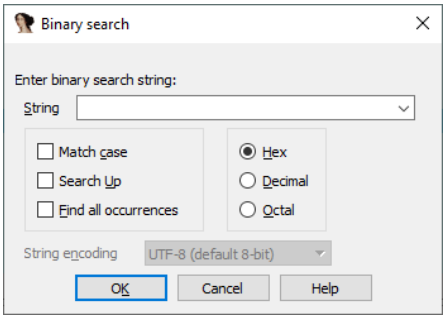
This search matches text anywhere in the current view, for example both the instructions and comments, if present.

For the main windows, the action is also accessible via the Search > Text... menu.

The notice "(slow!)" refers to the fact that for text searching, IDA has to render all text lines in the range being searched, which can get quite slow, especially for big binaries. However, if you need the features like regexp matching, or searching for text in comments, the wait could be worth it.

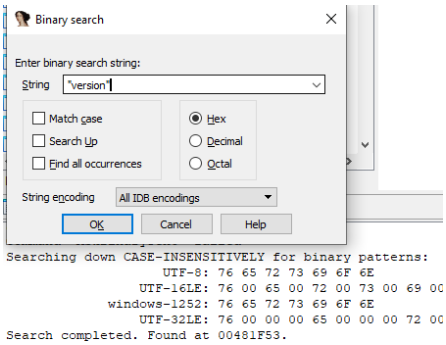
Binary search

Available as the shortcut pair **Alt-B**/**Ctrl-B**, or Search > Sequence of bytes..., this feature allows searching for byte sequences (including string literals) and patterns in the database (including process memory during debugging).



The input line accepts the following inputs:

1. byte sequence (space-delimited): `01 02 03 04`
2. byte sequence with wildcard bytes represented by question marks: `68 ? ? ? 0` will match both `68 C4 1A 48 00` and `68 D8 1A 48 00`.
3. one or more numbers in the selected radix (hexadecimal, decimal or octal). The number will be converted to the minimal necessary number of bytes according to the current processor endianness. For example, ² will be converted to `E0 69 44` on x86 (a little-endian processor). This feature is useful for finding values in data areas or embedded in instructions (immediates).
4. Quoted string literals, for example `"Error"`. The string will be converted to bytes using the encoding specified in the encoding selector. If "All Encodings" is selected, search will be performed using all configured encodings[2].
5. Wide-character string constant (e.g. `L"test"`). Only UTF-16 is used convert such strings to raw bytes.



[1] https://hex-rays.com/blog/igors-tip-of-the-week-36-working-with-list-views-in-ida/
[2] https://hex-rays.com/blog/igor-tip-of-the-week-13-string-literals-and-custom-encodings/

#48: Searching in IDA

16 Jul 2021

https://hex-rays.com/blog/igors-tip-of-the-week-48-searching-in-ida/

Immediate search

As mentioned previously, the same instruction operand can be represented in different ways[3] in IDA. For example, an instruction like

```
test dword ptr [eax], 10000h
```

can be also displayed as

```
test dword ptr [eax], 65536
```

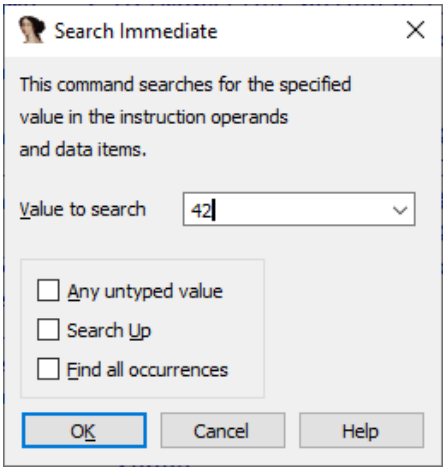
or even

```
test dword ptr [eax], AW_HIDE
```

So if you do the text search for `10000h`, IDA will find the first variation but not the other two. On x86, you can use binary search for `10000` hex (will be converted to byte sequence `00 00 01`), but this will not work for processors which use instruction encodings on non-byte boundary, or may give many false positives if unrelated instructions happen to match the byte sequence. So here's why the immediate search is preferable:

1. it only checks instructions with numerical operands or data items, improving search speed and reducing false positives;
2. it compares the **numerical value** of the operand, so any change in representation does not prevent the match, meaning it will find any of the three variations above

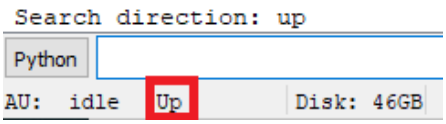
Available as the shortcut pair **`Alt-I`**/**`Ctrl-I`**, or Search > Immediate value...



The value can be entered in any numerical base using the C syntax (decimal, hex, octal).

Search direction

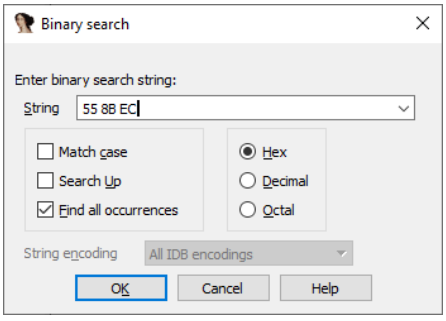
By default, all searches are performed "down" from the current position, i.e. toward increasing addresses. You can change it by checking "Search Up" in the individual search dialogs or beforehand via Search > Search direction. The currently set value is displayed in the menu item as well as IDA's status bar.

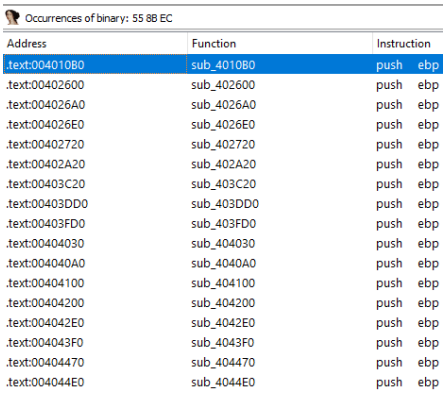


The "search next" commands and shortcuts (**`Ctrl-T`**, **`Ctrl-B`**, **`Ctrl-I`**) also use this setting.

Find all occurrences

This checkbox allows you to get results of the search over whole database or view in a list which you can then inspect at your leisure instead of looking at every search hit one by one.





[3] https://hex-rays.com/blog/igors-tip-of-the-week-46-disassembly-operand-representation/

#48: Searching in IDA

16 Jul 2021

https://hex-rays.com/blog/igors-tip-of-the-week-48-searching-in-ida/

Picking the search type

This is not a definitive guide but here are some suggestions:

1. text (e.g. prompt or error message) displayed by the program: binary search for the quoted substring (NB: this will not work if the string is not hardcoded but is in an external file or resource stream not loaded by IDA).
2. magic constant or error code: immediate search (in some cases binary search for the value can work too).
3. an address to which there are no apparent cross references: binary search for the address value (will only succeed if the reference actually uses the value directly without calculating it in some way).
4. specific instruction opcode pattern: binary search for byte sequence (possibly with wildcard bytes).
i5. nstruction not having a fixed encoding: text search for mnemonic and/or operands (possibly as regexp).

More info: Search submenu[4]

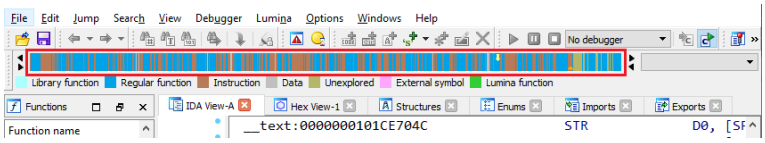
[4] https://hex-rays.com/products/ida/support/idadoc/568.shtml

#49: Navigation band

23 Jul 2021

https://hex-rays.com/blog/igors-tip-of-the-week-49-navigation-band/

Navigation band, also sometimes called the navigator, or navbar, is the UI element shown by default at the top of IDA's window, in the toolbar area.



It shows the global overview of the program being analyzed and allows to see at a quick glance how well has the program been analyzed and what areas may need attention.

Colors

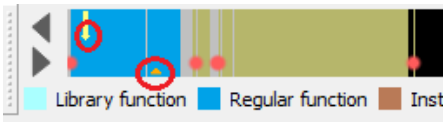
The colors are explained in the legend; the default color scheme uses the following colors:

- Cyan/turquose: Library functions, i.e. functions which have been recognized by a FLIRT signature. Usually such functions cone from the compiler or third party libraries and not the code written by the programmer, so they can often be ignored as a known quantity;
- Blue: Regular functions, i.e. functions not recognized by FLIRT or Lumina. These could contain the custom functionality, specific to the program;
- Maroon/brown: instructions(code) not belonging to any functions. These could appear when IDA did not detect or misdetected function boundaries, or hint at code obfuscation being employed which could prevent proper function creation. It could also be data incorrectly being treated as code.
- Gray: data. This color is used for all defined data items (string literals, arrays, individual variables).
- Olive: unexplored bytes, i.e. areas not yet converted to either code or data.
- Magenta: used to mark functions or data imported from other modules (including wrapper thunks for imported functions).
- Lime green: functions recognized by Lumina. They could be either library functions, or custom functions seen previously in other binaries and uploaded by users to the public Lumina server.

  Colors can be changed when changing the color scheme, or individually in Options > Colors... , Navigation band.

Indicators

In addition to the colors, there may be additional indicators on the navigation band. The yellow arrow is the current cursor position in the disassembly (IDA View), while the small orange triangle on the opposite side shows the current autoanalysis location (it is only visible while autoanalysis is in progress).

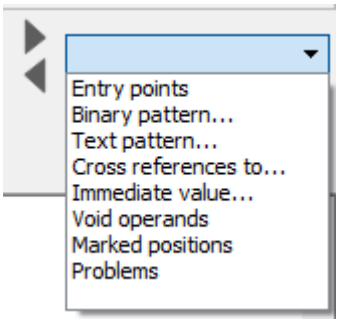


Additional display

The combobox (dropdown) at the right of the navigation band allows you to add some additional markers to it. For example, you can show:

- Entry points (exported functions);
- Binary or text pattern search results[1];
- immediate search[1] results;
- cross references[1] to a specific address;
- bookmarked positions;
- etc.

The markers show up as red circles and can be clicked to navigate.



[1] https://hex-rays.com/blog/igors-tip-of-the-week-48-searching-in-ida/

#49: Navigation band

23 Jul 2021

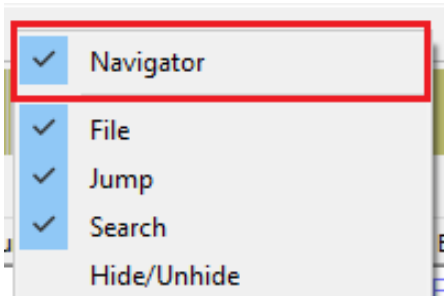
https://hex-rays.com/blog/igors-tip-of-the-week-49-navigation-band/

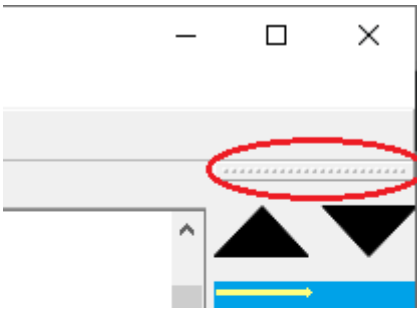
Configuration

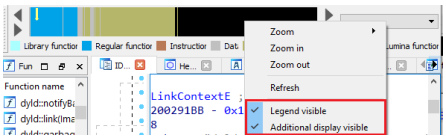
The control can be hidden or shown via View > Toolbars > Navigator, or the same item in the toolbar's context menu.

It can be placed at any of the four sides of IDA's window by using the drag handle.

In the horizontal position, you can show or hide the legend and the additional display combobox from the context menu.



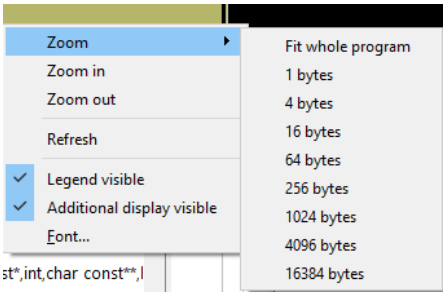




Navigation and zooming

By default, the navigation band shows the complete program, however you can zoom in to see a more detailed view of a specific part. Zooming can be done by `Ctrl` + mouse wheel, or from the context menu. The numerical options specify how many bytes of the program are represented by one pixel on the band.

Once zoomed in, the visible part can be scrolled with the mouse wheel or by clicking the arrow buttons at either end of the band. You can click into any part of the band to navigate there in the disassembly view.

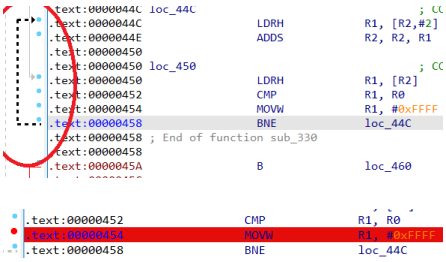


#50: Execution flow arrows

30 Jul 2021

https://hex-rays.com/blog/igors-tip-of-the-week-50-execution-flow-arrows/

Although nowadays most IDA users probably use the graph view, the text view can still be useful in certain situations. In case you haven't noticed, it has a UI element which can help you visualize code flow even without the full graph and even outside of functions (the graph view is available only for functions). This element is shown on the left of the disassembly listing:



The arrows represent code flow (cross-references) and the following types may be present:

- Solid lines represent unconditional jumps/branches, dashed lines – conditional ones;
- Thick arrows are used for jumps back to lower addresses (they indicate potential loops);
- The current arrow is highlighted in black;
- Red arrows are used when target and/or destination lies outside of the function boundaries

In addition to arrows, the blue dots indicate potential breakpoint location, so the breakpoint can be added by clicking on the dot, which will highlight the whole line red to indicate an active breakpoint.

#51: Custom calling conventions

06 Aug 2021

https://hex-rays.com/blog/igors-tip-of-the-week-51-custom-calling-conventions/

The Hex-Rays decompiler was originally created to deal with code produced by standard C compilers. In that world, everything is (mostly) nice and orderly: the calling conventions[1] are known and standardized and the arguments are passed to function according to the ABI[2].

However, the real life is not that simple: even in code coming from standard compilers there may be helper functions accepting arguments in non-standard locations, code written in assembly, or whole program optimization[3] causing compiler to use custom calling conventions for often-used functions. And code created with non-C/C++ compilers may use completely different calling conventions (a notable example is Go).

Thus a need arose to specify custom calling conventions so that the decompiler can provide readable output when they're used. For this, ability to specify custom calling conventions has been added to IDA and decompiler.

Text input dialog boxes (e.g. Enter Comment or Edit Local

The most commonly used custom calling convention is specified using the keyword `__usercall`. The basic syntax is as follows:

```
{return type} __usercall funcname@<return argloc>({type} arg1, {type} arg2@<argloc>, ...);
```

where `arglocis` one of the following:

- a processor register name, e.g. `eax`, `ebx`, `esi` etc. In some cases flag registers (`zf, sf, cf` etc.) may be accepted too.
- a register pair delimited with a colon, e.g. <edx:eax>.

The register size should match the argument or return type (if the function returns `void`, return argloc must be omitted). Arguments without location specifiers are assumed to be passed on stack according to usual rules.

Scattered argument locations

In complicated situations a large argument (such as a structure instance) may be passed in multiple registers and/or stack slots. In such case the following descriptors can be used:

- a partial register location: `argoff:register^regoff.size`.
- a partial stack location: `argoff:^stkoff.size`.
- a list of partial register and/or stack locations covering the whole argument delimited with a comma.

Where:

- `argoff` – offset within the argument
- `stkoff` – offset in the stack frame (the first stack argument is at offset 0)
- `register` – register name used to pass part of the argument
- `regoff` – offset within the register
- `size` – number of bytes for this portion of the argument

`regoff` and `size` can be omitted if there is no ambiguity (i.e. whole register is used).

For example, a 12-byte structure passed in `RDI` and `RSI` could be specified like this:

```
void __usercall myfunc(struc_1 s@<0:rdi, 8:rsi.4>);
```

Userpurge

The `__userpurge` calling convention is equivalent to `__usercall` except it is assumed that the callee adjusts the stack to account for arguments passed on stack (this is similar to how `__cdecl` differs from `__stdcall` on x86).

[1] https://docs.microsoft.com/en-us/cpp/cpp/calling-conventions
[2] https://en.wikipedia.org/wiki/Application_binary_interface
[3] https://docs.microsoft.com/en-us/cpp/build/reference/gl-whole-program-optimization

#51: Custom calling conventions

06 Aug 2021

https://hex-rays.com/blog/igors-tip-of-the-week-51-custom-calling-conventions/

Spoiled registers

The compiler or OS ABI also usually specifies which registers are caller-saved, i.e. may be spoiled (or clobbered) by a function call. In general, any register which can be used for argument passing or return value is considered potentially spoiled because the called function could in turn call other functions. For example, on x86, `EAX`, `ECX`, and `EDX` are by default considered spoiled and their values after the call are considered undefined by the decompiler. If this is not the case, you can help the decompiler by using the `__spoils<{reglist}>` specifier. For example, if the function does not clobber any registers, you can use the following prototype:

```
void __spoils<> func();
```

If a custom memcpy implementation uses esi and edi without saving and restoring them, you can add them to the spoiled list:

```
void* __spoils<esi, edi> memcpy(void*, void*, int);
```

The __spoils attribute can also be combined with __usercall:

```
int __usercall __spoils<> g@<esi>();
```

See also: Set function/item type[4] and Scattered argument locations[5] in IDA Help.

[4] https://hex-rays.com/products/ida/support/idadoc/1361.shtml
[5] https://hex-rays.com/products/ida/support/idadoc/1492.shtml

#52: Special type attributes

13 Aug 2021

https://hex-rays.com/blog/igors-tip-of-the-week-52-special-attributes/

IDA uses mostly standard C (and basic C++) syntax, but it also supports some extensions, in particular to represent low-level details which are not necessary for "standard" C code but are helpful for real-life binary code analysis. We've already covered custom types[1] and calling conventions[2], but there are more extensions you may use or encounter.

Function attributes

The following attributes may be used in function prototypes:

- `__pure` : a pure function (always returns the same result for same inputs and does not affect memory in a visible way);
- `__noreturn`: function does not return to the caller;
- `__usercall` or `__userpurge`: user-defined calling convention (see previous post[3]);
- `__spoils`: explicit spoiled registers specification (see previous post[3]);
- `v__attribute__((format(printf,n1,n2)))`: variadic function with a printf-style format string in argument at position n1 and variad-

Argument attributes

These attributes can often appear when IDA lowers a user-provided prototype to represent the actual low-level details of argument passing.

- `__hidden`: the argument was not present in source code (for example the implicit `this` pointer in C++ class methods).
- `__return_ptr`: hidden argument used for the return value (implies `__hidden`);
- `__struct_ptr`: argument was originally a structure value;
- `__array_ptr`: argument was originally an array (arrays ;
- `__unused`: unused function argument.

For example, if s1 is a structure of 16 bytes, then the following prototype:

```
struct s1 func();
```

will be lowered by IDA to:

```
struct s1 *__cdecl func(struct s1 *__return_ptr __struct_ptr retstr);
```

Other attributes

- `__cppobj`: used for structures representing C++ objects; some layout details change if this attribute is used (e.g. treatment of empty structs or reuse of end-of-struct padding in inheritance);
- `__ptr32`, `__ptr64`: explicitly-sized pointers;
- `__shifted`: a pointer which points not at the start of an object but some location inside or before it.

See also: Set function/item type[4] in IDA Help.

[1] https://hex-rays.com/blog/igors-tip-of-the-week-45-decompiler-types/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-51-custom-calling-conventions/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-51-custom-calling-conventions/
[4] https://hex-rays.com/products/ida/support/idadoc/1361.shtml