使用调试帮助程序

结构化数据（如 、 或 类型的对象）作为树的一部分显示在"**局部变量**"和"**表达式**"视图中。要访问对象的子结构，请展开树节点。子结构按其内存中的顺序显示，除非从上下文菜单中选择了"**按字母顺序对类和结构的成员进行排序**"选项。classstructunion

同样，指针显示为树项，其中单个子项表示指针的目标。如果选择了上下文菜单项"**自动取消引用指针**"，则指针和目标将合并为一个条目，显示指针的名称和类型以及目标的值。

这种标准表示对于简单结构的检查来说已经足够好了，但它通常不能对更复杂的结构（例如或关联容器）提供足够的见解。这些项在内部由指针的复杂排列表示，这些指针通常经过高度优化，部分数据既不能通过子结构也不能通过指针直接访问。QObjects

为了让用户也能轻松访问这些项目，Qt Creator 使用了称为*调试帮助程序*的 Python 脚本。始终自动使用调试帮助程序。若要强制以类似 C 的纯形式显示结构，请依次选择"**编辑**>**首选项**>**调试器**>"**局部变量和表达式**"，然后取消选中"**使用调试帮助程序**"复选框。这仍将使用Python脚本，但会生成更基本的输出。要强制对单个对象或给定类型的所有对象进行普通显示，请从上下文菜单中选择相应的选项。

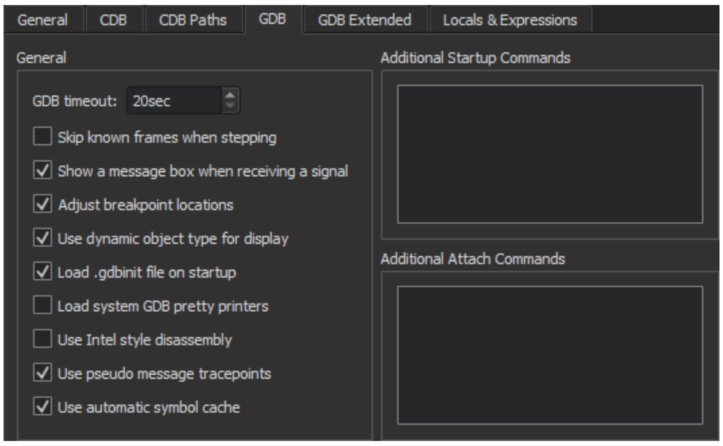
Qt Creator 附带了 200 多种最流行的 Qt 类、标准C++容器和智能指针的调试助手，满足了开箱即用C++应用程序开发人员的通常需求。

扩展调试帮助程序

Qt Creator 使用 Python 脚本将原始内存内容和键入来自本机调试器后端（当前支持 GDB、LLDB 和 CDB）的信息数据转换为在"**局部变量**"和"**表达式**"视图中呈现给用户的形式。

与GDB漂亮的打印机和LLDB的数据格式化程序不同，Qt Creator的调试助手独立于本机调试后端。也就是说，相同的代码可以与 Linux 上的 GDB、macOS 上的 LLDB 和 Windows 上的 CDB 一起使用，或者其他任何平台上，其中三个受支持的后端中至少有一个可用。

要使用安装在系统中或链接到应用程序使用的库的默认 GDB 漂亮打印机，请选择"**首选项**">**调试器**"> GDB >**加载系统 GDB 漂亮打印机**。有关更多信息，请参阅指定 GDB 设置。


General | CDB | CDB Paths | GDB | GDB Extended | Locals & Expressions

General

GDB timeout: 20sec

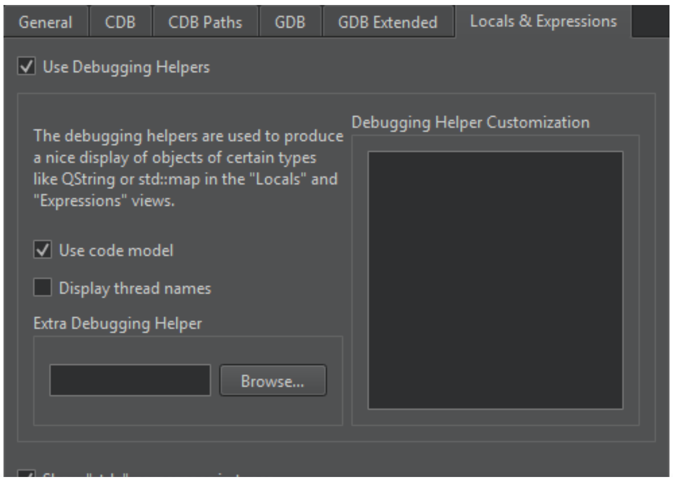
- [ ] Skip known frames when stepping
- [x] Show a message box when receiving a signal
- [x] Adjust breakpoint locations
- [x] Use dynamic object type for display
- [x] Load .gdbinit file on startup
- [ ] Load system GDB pretty printers
- [ ] Use Intel style disassembly
- [x] Use pseudo message tracepoints
- [x] Use automatic symbol cache

Additional Startup Commands

Additional Attach Commands

自定义内置调试帮助程序

您可以在加载并完全初始化内置调试帮助程序后执行命令。若要加载其他调试帮助程序或修改现有调试帮助程序，请选择"**编辑**>**首选项**">"**调试器**">"**局部变量**"和"**表达式**"，然后在"**调试帮助程序自定义**"字段中输入命令。


General | CDB | CDB Paths | GDB | GDB Extended | Locals & Expressions

- [x] Use Debugging Helpers

The debugging helpers are used to produce a nice display of objects of certain types like QString or std::map in the "Locals" and "Expressions" views.

- [x] Use code model
- [ ] Display thread names

Extra Debugging Helper

Browse...

Debugging Helper Customization

☑ Show QObject names if available

Maximum string length: 10000

Display string length: 100

如果在使用 GDB 时收到有关接收信号的错误消息，则可以指定 GDB 命令来处理信号。例如，如果收到以下错误消息，则可以告诉 GDB 忽略该信号：。SIGSTOPThe inferior stopped because it received a signal from the operating system. Signal name: SIGSTOP

To stop GDB from handling the signal, add the following commands to the **Debugging Helper Customization** field:SIGSTOP

```
handle SIGSTOP nopass
handle SIGSTOP nostop
```

To display a message box as soon as your application receives a signal during debugging, select **Edit** > **Preferences** > **Debugger** > **GDB** > **Show a message box when receiving a signal**.

Adding Custom Debugging Helpers

To add debugging helpers for your own types, no compilation is required, just adding a few lines of Python. The scripts can address multiple versions of Qt, or of your own library, at the same time.

To add debugging helpers for custom types, add debugging helper implementations to the startup file of the native debuggers (for example, or) or specify them directly in the **Additional Startup Commands** in **Edit** > **Preferences** > **Debugger** > **GDB**.~/.gdbinit~/.lldbinit

To get started with implementing debugging helpers for your own data types, you can put their implementation into the file in your Qt installation or stand-alone Qt Creator installation. On macOS, the file is bundled into the Qt Creator application package, where it is located in the folder.share/qtcreator/debugger/personaltypes.pyContents/resources/debugger

The file contains one example implementation:personaltypes.py

```
# def qdump__MapNode(d, value):
#    d.putValue("This is the value column contents")
#    d.putExpandable()
#    if d.isExpanded():
#        with Children(d):
#            # Compact simple case.
#            d.putSubItem("key", value["key"])
#            # Same effect, with more customization possibilities.
#            with SubItem(d, "data")
#                d.putItem("data", value["data"])
```

To add debugging helpers:

1. Open the file for editing. For example, if your Qt installation is located in the directory on Windows, look in . On macOS, look in .share/qtcreator/debugger/personaltypes.pyQt5C:\Qt5\Tools\QtCreator\share\qtcreator\debuggerQt5/Qt Creator.app/Contents/resources/debugger
2. Add your dumper implementation to the end of the file. For more information about implementing debugging helpers, see the following sections.personaltypes.py
3. To prevent from being overwritten when you update your Qt Creator installation (when updating your Qt installation, for example), copy it to a safe location outside the Qt Creator installation in your file system and specify the location in **Edit** > **Preferences** > **Debugger** > **Locals & Expressions** > **Extra Debugging Helper**.personaltypes.py

The custom debugging helpers will be automatically picked up from when you start a debugging session in Qt Creator or select **Reload Debugging Helpers** from the context menu of the **Debugger Log** view.personaltypes.py

Debugging Helper Overview

The implementation of a debugging helper typically consists of a single Python function, which needs to be named , where is the class or class template to be examined. Note that the scope resolution operator is replaced by double underscores: . Nested namespaces are possible. Template arguments are not used for the construction of the function name.qdump__NS__FooNS::Foo::__

Examples:

- The name for the function implementing a debugging helper for the type is .namespace Project { template<typename T> struct Foo {... } }qdump__Project__Foo
- The name for the function implementing a debugging helper for the type is .std::__1::vector<T>::iteratorqdump__std____1__vector__iterator

Qt Creator's debugger plugin calls this function whenever you want to display an object of this type. The function is passed the following parameters:

- d of type , an object containing the current settings and providing facilities to build up an object representing a part of the **Locals** and **Expressions** views.Dumper
- value of type , wrapping either a gdb.Value or an lldb.SBValue.Value

The function has to feed the Dumper object with certain information that is used to build up the object and its children's display in the **Locals** and **Expressions** views.qdump__*

Example:

```
def qdump__QFiniteStack(d, value):
        d.putItemCount(size)
        if d.isExpanded():
            d.putArrayData(value["_array"], size, value.type[0])
```

Note: To create dumper functions usable with both LLDB and GDB backends, avoid direct access to the and namespaces and use the functions of the class instead.gdb.*lldb.*Dumper

To get to the base instance of the object in the debugging helper, use the function or the following example code:value.base()

```
def qdump__A(d, value):
   t = value.members(True)[0].type
   dptr, base_v = value.split('p{%s}' % t.name)
   d.putItem(base_v)
```

Debugging helpers can be set up to be called whenever a type name matches a regular expression. To do so, the debugging helper's function name must begin with (with two underscore characters). In addition, the function needs to have a third parameter called with a default value that specifies the regular expression that the type name should match.qdump__regex

For example, the Nim 0.12 compiler assigns artificial names, such as and , to all generic sequences it compiles. To visualize these in Qt Creator, the following debugging helper may be used:TY1TY2

```
def qdump__NimGenericSequence__(d, value, regex = "^TY.*$"):
    size = value["Sup"]["len"]
    base = value["data"].dereference()
    typeobj = base.dereference().type
    d.putArrayData(base, size, typeobj)
```

Debugging Helper Implementation

A debugging helper creates a description of the displayed data item in a format that is similar to GDB/MI and JSON.

For each line in the **Locals** and **Expressions** views, a string like the following needs to be created and channeled to the debugger plugin.

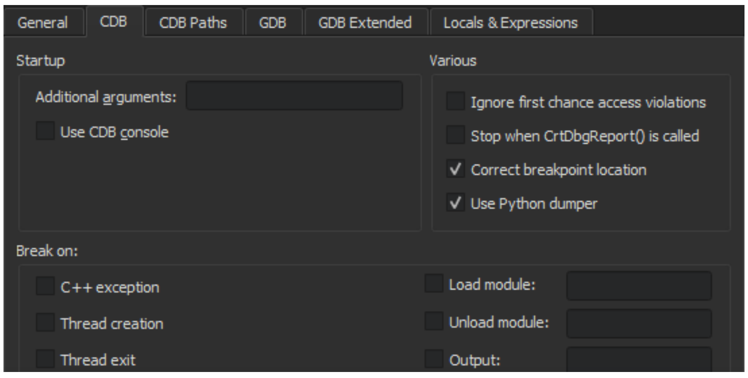
```
{ iname='some internal name',           # optional
  address='object address in memory',   # optional
  name='contents of the name column',   # optional
  value='contents of the value column',
  type='contents of the type column',
  numchild='number of children',        # zero/nonzero is sufficient
  children=[              # only needed if item is expanded in view
     {iname='internal name of first child',
       },
     {iname='internal name of second child',
       },

  ]}
```

The value of the field is the internal name of the object, which consists of a dot-separated list of identifiers, corresponding to the position of the object's representation in the view. If it is not present, it is generated by concatenating the parent object's , a dot, and a sequential number.inameiname

The value of the field is displayed in the **Name** column of the view. If it is not specified, a simple number in brackets is used instead.name

As the format is not guaranteed to be stable, it is strongly recommended not to generate the wire format directly, but to use the abstraction layer provided by the Python Dumper classes, specifically the class itself, and the and abstractions. These provide a complete framework to take care of the and fields, to handle children of simple types, references, pointers, enums, and known and unknown structs, as well as some convenience functions to handle common situations.DumperDumper:ValueDumper:Typeinameaddr

When using CDB as debugger backend, you can enable the Python dumper by selecting **Edit** > **Preferences** > **Debugger** > **CDB** > **Use Python dumper**.



Additional arguments:
☐ Use CDB console
☐ Ignore first chance access violations
☐ Stop when CrtDbgReport() is called
✓ Correct breakpoint location
✓ Use Python dumper
☐ C++ exception
☐ Thread creation
☐ Thread exit
☐ Load module:
☐ Unload module:
☐ Output:

✓ First chance exceptions ✓ Second chance exceptions

Dumper Class

The class contains generic bookkeeping, low-level, and convenience functions.`Dumper`

The member functions of the class are the following:`Dumper`

- `putItem(self, value)` - The *master function* that handles basic types, references, pointers, and enums directly, iterates over base classes and class members of compound types, and calls functions when appropriate.`qdump__*`
- `putIntItem(self, name, value)` - Equivalent to:

```
with SubItem(self, name):
    self.putValue(value)
    self.putType("int")
```

- `putBoolItem(self, name, value)` - Equivalent to:

```
with SubItem(self, name):
    self.putValue(value)
    self.putType("bool")
```

- `putCallItem(self, name, value, func, *args)` - Uses the native debugger backend to place the function call on the value specified by *value* and output the resulting item.`func`

  Native calls are extremely powerful and can leverage existing debugging or logging facilities in the debugged process, for example. However, they should only be used in a controlled environment, and only if there is no other way to access the data, for the following reasons:

  - Direct execution of code is dangerous. It runs native code with the privileges of the debugged process, with the potential to not only corrupt the debugged process, but also to access the disk and network.
  - Calls cannot be executed when inspecting a core file.
  - Calls are expensive to set up and execute in the debugger.

- `putArrayData(self, address, itemCount, type)` - Creates the number of children specified by of the type of an array-like object located at .`itemCounttypeaddress`
- `putSubItem(self, component, value)` - Equivalent to:

```
with SubItem(self, component):
    self.putItem(value)
```

  Exceptions raised by nested function calls are caught and all output produced by is replaced by the output of:`putItem`

```
except RuntimeError:
    d.put('value="<invalid>",type="<unknown>",numchild="0",')
```

- `put(self, value)` - A low-level function to directly append to the output string. That is also the fastest way to append output.
- `putField(self, name, value)` - Appends a field.`name='value'`
- `childRange(self)` - Returns the range of children specified in the current scope.`Children`
- `putItemCount(self, count)` - Appends the field to the output.`value='<%d items>'`
- `putName(self, name)` - Appends the field.`name=''`
- `putType(self, type, priority=0)` - Appends the field , unless the *type* coincides with the parent's default child type or was already called for the current item with a higher value of .`type=''putTypepriority`
- `putBetterType(self, type)` - Overrides the last recorded .`type`
- `putExpandable(self)` - Announces the existence of child items for the current value. The default are no children.
- `putNumChild(self, numchild)` - Announces the existence (> 0) or non-existence of child items for the current value.`numchild`
- `putValue(self, value, encoding = None)` - Appends the file , optionally followed by the field . The needs to be convertible to a string entirely consisting of alphanumerical values. The parameter can be used to specify the encoding in case the real value had to be encoded in some way to meet the alphanumerical-only requirement. The parameter is either a string of the form where is any of , , , , , or . gives the size of the basic component of the object if it is not implied by and specifies whether or not the value should be surrounded by quotes in the display.`value=''valueencoding=''valueencodingencodingcodec:itemsize:quotecodeclatin1utf8utf16ucs4intfloatitemsizecodecquote`

  Example:

```
# Safe transport of quirky data. Put quotes around the result.
d.putValue(d.hexencode("ABC\"DEF"), "utf8:1:1")
```

- `putByteArrayValue(self, value)` - Encodes a QByteArray and calls with the correct setting.`putValueencoding`
- `isExpanded(self)` - Checks whether the current item is expanded in the view.
- `createType(self, pattern, size = None)` - Creates a object. The exact operation depends on .`Dumper.Typepattern`
  - If matches the name of a well-known type, a object describing this type is returned.`patternDumper.Type`
  - If is the name of a type known to the native backend, the returned type describes the native type.`pattern`
  - Otherwise, is used to construct a type description by interpreting a sequence of items describing the field of a structure as follows. Field descriptions consist of one or more characters as follows:`pattern`
    - `q` - Signed 8-byte integral value
    - `Q` - Unsigned 8-byte integral value
    - `i` - Signed 4-byte integral value
    - `I` - Unsigned 4-byte integral value
    - `h` - Signed 2-byte integral value
    - `H` - Unsigned 2-byte integral value
    - `b` - Signed 1-byte integral value
    - `B` - Unsigned 1-byte integral value
    - `d` - 8-byte IEEE 754 floating point value
    - `f` - 4-byte IEEE 754 floating point value
    - `p` - A pointer, that is, an unsigned integral value of suitable size according to the target architecture
    - `@` - Suitable padding. The size is determined by the preceding and following field and the target architecture
    - `<n>s` - A blob of <n> bytes, with implied alignment of 1
    - `<typename>` - A blob of suitable size and suitable alignment determined by a with the name `Dumper.Typetypename`

Dumper.Type Class

The class describes the type of a piece of data, typically a C++ class or struct, a pointer to a struct, or a primitive type, such as an integral or floating point type.`Dumper.Type`

Type objects, that is, instances of the class, can be created by native debugger backends, usually by evaluating Debug Information built into or shipped alongside the debugged binary, or created on-the-fly by the debugging helper.`Dumper.Type`

Qt Creator uses the possibility to provide type information on-the-fly for most Qt classes, obliterating the need to use *Debug* builds of Qt for the purpose of object introspection.

The member functions of the class are the following:`Dumper.Type`

- `name` - The name of this type as a string, or if the type is anonymous.`None`
- `size(self)` - Returns the size of an object of this type in bytes.
- `bitsize(self)` - Returns the size of an object of this type in bits.
- `alignment(self)` - Returns the required alignment for objects of this type in bytes.
- `deference(self)` - Returns the dereferences type for pointer type, otherwise.`None`
- `pointer(self)` - Returns a pointer type that can be dereferenced to this type.
- `target(self)` - A convenience function that returns the item type for array types and the dereferenced type for pointers and references.
- `stripTypedefs(self)` - Returns the underlying type if this type is an alias.
- `templateArgument(self, position, numeric = False)` - Returns the template parameter located at if this is a templated type. If is , returns the parameter as an integral value.`positionnumericTrue`
- `fields(self)` - Returns a list of describing the base classes and data members of this type.`Dumper:Fields`

Dumper.Field Class

The class describes a base class or a data member of a type object.`Dumper.Field`

The member function and properties of the class are the following:`Dumper.Field`

- `isBaseClass` - Distinguishes between base classes and data members.
- `fieldType(self)` - Returns the type of this base class or data member.
- `parentType(self)` - Returns the owning type.
- `bitsize(self)` - Returns the size of this field in bits.
- `bitpos(self)` - Returns the offset of this field in the owning type in bits.

Dumper.Value Class

The class describes a piece of data, such as instances of C++ classes or primitive data types. It can also be used to describe artificial items that have no direct representation in memory, such as file contents, non-contiguous objects, or collections.`Dumper.Value`

A has always an associated . The two main representations of the value's actual data are:`Dumper.ValueDumper.Type`

- Python object following the Python buffer protocol, such as a Python , or a object. The should match the size of this value's type.`memoryviewbytessize()`
- An integral value representing a pointer to the begin of the object in the current address space. The size of the object is given by its type's .`size()`

Knowledge of the internal representation of a is typically not required when creating a debugger helper for it.`Dumper.Value`

- `pointer(self)` - Returns an interpretation of this value as a pointer in the current address space.
- `members(self, includeBases)` - Returns a list of objects representing the base objects and data members of this value.`Dumper.Value`
- `dereference(self)` - For values describing pointers, returns the dereferenced value, and otherwise.None
- `cast(self, type)` - Returns a value that has the same data as this value, but the type .`type`
- `address(self)` - Returns the address of this value if it consists of a contiguous region in the current address space, and otherwise.None
- `data(self)` - Returns the data of this value as a Python object.`bytes`
- `split(self, pattern)` - Returns a list of values created according to from this value's data. Acceptable patterns are the same as for .`patternDumper.createType`
- `dynamicTypeName(self)` - Tries to retrieve the name of the dynamic type of this value if this is a base class object. Returns if that is not possible.None

Children and SubItem Class

The attempt to create child items might lead to errors if data is uninitialized or corrupted. To gracefully recover in such situations, use and *Context Managers* to create the nested items.`ChildrenSubItem`

The constructor uses one mandatory argument and several optional arguments. The mandatory argument refers to the current object. The optional arguments can be used to specify the number of children, with type and grandchildren each. If is specified, only that many children are displayed. This should be used when dumping container contents that might take overly long otherwise. The parameters and can be used to reduce the amount of data produced by the child dumpers. Address printing for the *n*th child item will be suppressed if its address equals with *addrBase + n * addrStep*.`Children__init__(self, dumper, numChild = 1, childType = None, childNumChild = None, maxNumChild = None, addrBase = None, addrStep = None)DumpernumChildchildType_childNumChild_maxNumChildaddrBaseaddrStep`

Example:

```
if d.isExpanded():
    with Children(d):
        with SubItem(d):
            d.putName("key")
            d.putItem(key)
        with SubItem(d):
            d.putName("value")
            d.putItem(value)
```

Note that this can be written more conveniently as:

```
d.putNumChild(2)
if d.isExpanded():
    with Children(d):
        d.putSubItem("key", key)
        d.putSubItem("value", value)
```

‹ Interacting with the Debugger   Debugging Qt Quick Projects ›

Qt DOCUMENTATION

Customer Success

Community

Contribute to Qt
Forum
Wiki
Downloads
Marketplace

Feedback Sign In