How to retrieve the data type of a variable?

Asked 9 years, 9 months ago Modified 1 year ago Viewed 76k times

13

I am using Arduino and I would like to know if there is a function that returns the data type of a variable. That is, I would like to run something as like the following:

```
// Note: 'typeof' is a sample function that should return the data type.
Serial.println(typeof(myVar));
```

serial variables data-type

Share Improve this question Follow

asked Jun 25, 2014 at 14:33
user502052
429 ● 1 ● 4 ● 7

6 Why do you need to do this? the type of the variable should be known since you declared it – sachleen Jun 25, 2014 at 14:44

@sachleen Not true. Arduino Reference info is infamous for not telling you what function return types are. I have to define as auto, then use a bunch of function overloads to find out. RTTI would be simpler to find out. – brewmanz Jun 7, 2022 at 2:01

5 Answers

Sorted by: Highest score (default)

20

In a typical C++ program, you would use the `typeid` operator, like this:

```
std::cout << typeid(myVar).name();
```

However, that requires a compiler feature called Runtime Type Information (RTTI). It's disabled in the Arduino IDE, presumably because it tends to increase the runtime memory requirements of the program.

You can get more information about the resource cost here: https://stackoverflow.com/questions/579887/how-expensive-is-rtti

Any fully-featured C++ compiler will definitely support RTTI though. If you want to try using a 3rd party IDE (such as Eclipse with the Arduino plugin) then you could easily enable it. It's probably not worth the hassle just for that though.

Alternative
A higher-performing (but less flexible) solution would be to use a traits class approach. This involves some funky template meta-programming:

```
// Generic catch-all implementation.
template <typename T_ty> struct TypeInfo { static const char * name; };
template <typename T_ty> const char * TypeInfo<T_ty>::name = "unknown";

// Handy macro to make querying stuff easier.
#define TYPE_NAME(var) TypeInfo< typeof(var) >::name

// Handy macro to make defining stuff easier.
#define MAKE_TYPE_INFO(type)  template <> const char * TypeInfo<type>::name =
#type;

// Type-specific implementations.
MAKE_TYPE_INFO( int )
MAKE_TYPE_INFO( float )
MAKE_TYPE_INFO( short )
```

You can add `MAKE_TYPE_INFO(..)` lines for any type you want, including the names of custom classes. You could then use it like this:

```
int myVar = 17;
Serial.println( TYPE_NAME(myVar) );
```

Anything you don't define using `MAKE_TYPE_INFO(..)` will show up as `"unknown"`.

That's some fairly advanced stuff in there, so I won't try to explain how it all works here. There are various tutorials on the web though about C++ template programming if you're interested.

EDIT: It's worth noting that the `typeof` operator isn't standard C++, but is supported by a few compilers, such as GCC. It's basically an older equivalent of `decltype`, which appears in the C++11 standard.

Share Improve this answer Follow

edited May 23, 2017 at 12:37
Community Bot
1

answered Jun 25, 2014 at 16:01
Peter Bloomfield
10.9k ●9 ●47 ●87

3 very informative and thorough answer! would appreciate more of this kind – Omer Jun 26, 2014 at 11:21

19

I use a simple stupid approach...

```
// serial print variable type
void types(String a) { Serial.println("it's a String"); }
void types(int a) { Serial.println("it's an int"); }
void types(char *a) { Serial.println("it's a char*"); }
void types(float a) { Serial.println("it's a float"); }
void types(bool a) { Serial.println("it's a bool"); }
```

This is the concept of **function overloading** where **multiple functions** with different parameter types are created but with the **same function name**. During run time, the function matching the right number of arguments and argument type(s) will get called. Hope this explanation helps.

Share Improve this answer Follow

edited Mar 10, 2023 at 14:54
Community Bot
1

answered May 31, 2017 at 22:13
SnakeNET
199 ●1 ●2

Could you expand your answer *a little*, briefly explaining how it works? – Greenonline Jun 1, 2017 at 4:45

i had to rewrite it by hand because copy paste led to issues i didn't understand. After retyping it by hand just as it is to the sketch it worked like a charm. Quite usefull... – novski Apr 1, 2018 at 9:38

Simple, easier to understand without knowledge of templates. Could be expanded to return enum of type. – voidPointer Jul 28, 2021 at 9:44

3

Building on @snakeNET's answer (which I regard as function **overload** rather than **polymorphism**)...

A more generic way would be to pass in a pointer to Print, allowing any Print related class to be used (not just Serial); also pass objects by reference e.g.

```
void types(Print* p, const String&) { p->print("it's a String"); } // for each type
...
Serial.print("Type is "); types(&Serial, myStr); Serial.println();
```

Note that I has changed 'println' to 'print'; again, making it more generic.

Share Improve this answer Follow

answered Jul 25, 2021 at 2:07
brewmanz
155 ● 4

1 I think the meaning of the term **polymorphism** has morphed somewhat over the decades to now include function overloading *outside* of OOP classes as well as overloading of member functions *within* classes. When I started learning programming, it only referred to overriding of virtual class methods in descendant classes. – tim Jul 27, 2021 at 21:58

1

I tested the template form, the long-hand form, and the `__PRETTY_FUNCTION__` macro and got different results. See the last two rows of the table for the differences.

Template Form 1

Attribution: Peter Bloomfield

```
MAKE_TYPE_INFO( bool )
MAKE_TYPE_INFO( bool* )
MAKE_TYPE_INFO( char )
MAKE_TYPE_INFO( char* )
MAKE_TYPE_INFO( double )
MAKE_TYPE_INFO( double* )
MAKE_TYPE_INFO( float )
MAKE_TYPE_INFO( float* )
MAKE_TYPE_INFO( int )
MAKE_TYPE_INFO( int* )
MAKE_TYPE_INFO( short )
MAKE_TYPE_INFO( short* )
MAKE_TYPE_INFO( String )
MAKE_TYPE_INFO( String* )
MAKE_TYPE_INFO( char[5] )
MAKE_TYPE_INFO( char(*)[5] )
```

Long-Hand Form

Attribution:

1. SnakeNET
2. brewmanz
3. Tweaked by me to be more generic.

```
const char* TypeOf(const bool&)   { static const char type[] = "bool";    return
type; }
const char* TypeOf(const bool*)   { static const char type[] = "bool*";   return
type; }
const char* TypeOf(const char&)   { static const char type[] = "char";    return
type; }
const char* TypeOf(const char*)   { static const char type[] = "char*";   return
type; }
const char* TypeOf(const double&) { static const char type[] = "double";  return
type; }
const char* TypeOf(const double*) { static const char type[] = "double*"; return
type; }
const char* TypeOf(const float&)  { static const char type[] = "float";   return
type; }
const char* TypeOf(const float*)  { static const char type[] = "float*";  return
type; }
const char* TypeOf(const int&)    { static const char type[] = "int";     return
type; }
const char* TypeOf(const int*)    { static const char type[] = "int*";    return
type; }
const char* TypeOf(const String&) { static const char type[] = "String";  return
type; }
const char* TypeOf(const String*) { static const char type[] = "String*"; return
type; }
//const char* const TypeOf(const char[5])    { static const char* type =
"char[5]";    return type; }  // Decays to char* so generates compiler
redefinition warning.
const char* const TypeOf(const char(*)[5]) { static const char* type =
"char(*)[5]"; return type; }
```

Template Form 2: `__PRETTY_FUNCTION__` Macro

Attribution:

1. Original `String` version by Vladimir Talybin
2. Tweaked `char` array version by me.

Using `String`

```
template <class T>
String type_name(const T&)
```

```
{
    String s = __PRETTY_FUNCTION__;
    int start = s.indexOf("[with T = ") + 10;
    int stop = s.lastIndexOf(']');
    return s.substring(start, stop);
}
```

Using `char` array

This variation has lower memory requirements.

```
template <typename T>
char* type_name(const T&)
{
    char* pf = __PRETTY_FUNCTION__;
    char* begin = strstr(pf, "[with T = ") + 10;
    char* end = strrchr(pf, ']');
    *end = 0;
    return begin;
}
```

Test Code

```
bool b = false;
char c = 'a';
double d = 1.2;
float f = 3.4;
int i = 5;
String s = "6789";
char chars[] = "1234";

Serial.println(F("TypeOf() Test"));
Serial.println(TypeOf(b));
Serial.println(TypeOf(&b));
Serial.println(TypeOf(c));
Serial.println(TypeOf(&c));
Serial.println(TypeOf(d));
Serial.println(TypeOf(&d));
Serial.println(TypeOf(f));
Serial.println(TypeOf(&f));
Serial.println(TypeOf(i));
Serial.println(TypeOf(&i));
Serial.println(TypeOf(s));
Serial.println(TypeOf(&s));
Serial.println(TypeOf(chars));
Serial.println(TypeOf(&chars));

Serial.println(F("TYPE_NAME() Test"));
Serial.println(TYPE_NAME(b));
Serial.println(TYPE_NAME(&b));
Serial.println(TYPE_NAME(c));
Serial.println(TYPE_NAME(&c));
Serial.println(TYPE_NAME(d));
Serial.println(TYPE_NAME(&d));
Serial.println(TYPE_NAME(f));
Serial.println(TYPE_NAME(&f));
Serial.println(TYPE_NAME(i));
Serial.println(TYPE_NAME(&i));
Serial.println(TYPE_NAME(s));
Serial.println(TYPE_NAME(&s));
Serial.println(TYPE_NAME(chars));
Serial.println(TYPE_NAME(&chars));

Serial.println(F("type_name() Test"));
Serial.println(type_name(b));
Serial.println(type_name(&b));
Serial.println(type_name(c));
Serial.println(type_name(&c));
Serial.println(type_name(d));
Serial.println(type_name(&d));
Serial.println(type_name(f));
Serial.println(type_name(&f));
Serial.println(type_name(i));
Serial.println(type_name(&i));
Serial.println(type_name(s));
Serial.println(type_name(&s));
Serial.println(type_name(chars));
Serial.println(type_name(&chars));
```

Results

TypeOf()	TYPE_NAME()	type_name()

bool	bool	bool
bool*	bool*	bool*
char	char	char
char*	char*	char*
double	double	double
double*	double*	double*
float	float	float
float*	float*	float*
int	int	int
int*	int*	int*
String	String	String
String*	String*	String*
char*	char[5]	char [5]
char(*)[5]	char(*)[5]	char (*)[5]

Some Conclusions

1. `TypeOf()` has an issue with a decaying pointer.
2. `TypeOf()` and `TYPE_NAME()` have to have individual definitions with long-hand form or `MAKE_TYPE_INFO`.
3. `type_name()` uses the `__PRETTY_FUNCTION__` macro to automatically get the type name within the template function, therefore there is no need to individually define the different types.
4. Template Form 1 has the lowest memory requirements.

Questions

1. Since Template Form 2 has higher memory requirements than Template Form 1, how could it be converted from Form 2 to Form 1?
2. How would a `static const char *` be initialised from a substring of a `char` array obtained from the `__PRETTY_FUNCTION__` macro and passed as a template argument at compile-time?

Something like this pseudo code?

```
static const char * name = substr(__PRETTY_FUNCTION__, begin, end);
```

Perhaps this constant expression which can be assigned to a template struct member at compile-time? (I used a `for` loop because `strstr` and `strrchr` are not `constexpr`.)

```
template <typename T>
constexpr char* type_name(const T&)
{
    char* pf = __PRETTY_FUNCTION__;
    char* begin;
    char* end;
    begin = end = pf;
    bool found_open_square_bracket = false;
    for (; *pf != 0; pf++)
    {
      if (!found_open_square_bracket && *pf == '[')
      {
        begin = pf + 10;
        found_open_square_bracket = true;
      }
      if (*pf == ']')
      {
        end = pf;
      }
    }
    *end = 0;
    return begin;
}
```

Share Improve this answer Follow

edited Jul 31, 2021 at 16:27

answered Jul 27, 2021 at 23:02
tim
699 ● 6 ● 15

1 Note that: 1. When you pass an array as a function argument, it decays to a pointer to its first element. 2. `char*[5]` is an array of pointers, whereas `&chars` is a pointer to an array. – Edgar Bonet Jul 28, 2021 at 9:09

1 @Juraj, Sorry. I thought the first paragraph covered it in the context of this thread. I'll make it clearer. – tim Jul 31, 2021 at 15:57

1 It is more a limitation of C++ version in Arduino. On PC I would return `std::string_view` in this case. On Arduino you may return a struct of `const char*` and `size_t`. This will solve you mem issue. – Vladimir Talybin Jul 31, 2021 at 19:46

1 I would also split the template part and parsing part in 2 functions to make compiler generate less code for each requested type. – Vladimir Talybin Jul 31, 2021 at 19:48

1

There also a trick using compiler predefined macro. The advantage is it can print any type.

```
template <class T>
String type_name(const T&)
{
    String s = __PRETTY_FUNCTION__;

    int start = s.indexOf("[with T = ") + 10;
    int stop = s.lastIndexOf(']');

    return s.substring(start, stop);
}
```

Use it like this

```
double pi = 3.14;
const char* str = "test";

Serial.println(type_name(pi));
Serial.println(type_name(str));
```

EDIT:

Some improvements based on tim's analyze.

First I have a comment on variation that has lower memory requirements. `__PRETTY_FUNCTION__` is a literal string (a constant) that should not be changed. So my const version that do not use heap is following.

```
template <size_t N>
const char* extract_type(const char (&signature)[N])
{
    const char* beg = signature;
    while (*beg++ != '=');
    ++beg;

    const char* end = signature + N;
    for (; *end != ']'; --end);

    static char buf[N];
    char* it = buf;

    for (; beg != end; ++beg, ++it)
        *it = *beg;
    *it = 0;

    return buf;
}

template <class T>
const char* type_name(const T&)
{
    return extract_type(__PRETTY_FUNCTION__);
}
```

A less safe but more efficient version using the fact that offset to type we want extract is a constant (always the same). If function signature changes (like put it in a namespace) the offset will change too. But come on, set it once, write a comment and it will work forever :).

```
template <size_t N>
const char* extract_type(const char (&signature)[N])
{
    const char* beg = signature + 42;
    const char* end = signature + N - 2;

    static char buf[N - 43];
    char* it = buf;

    for (; beg != end; ++beg, ++it)
        *it = *beg;
    *it = 0;

    return buf;
}

template <class T>
const char* type_name(const T&)
{
    return extract_type(__PRETTY_FUNCTION__);
}
```

While above is ok ofcourse it can be improved further. Here is a my version of constexpr function mentioned by @tim. It uses constant offset as above to keep it short. This code generates only extracted string of type, the proof is here.

```
#include <utility>

template <class T, std::size_t... I>
const char* type_name(std::index_sequence<I...>)
{
    static constexpr char name[] = { __PRETTY_FUNCTION__[I + 60]..., 0 };
    return name;
}

template <class T>
const char* type_name(const T&)
{
    return type_name<T>(
        std::make_index_sequence<sizeof(__PRETTY_FUNCTION__) - 44>());
}
```

The minimum requirement for this is C++14 because of sequence traits. These can be re-implemented (see definition here) to adapt it to C++11.

Share Improve this answer Follow edited Aug 1, 2021 at 19:18 answered Jul 29, 2021 at 22:00
Vladimir Talybin
121 ● 3

That's great, but `int stop = s.lastIndexOf(']');` is better because some types have `[]` in them. – tim Jul 31, 2021 at 10:29

@tim True, thank you! Will change it. – Vladimir Talybin Jul 31, 2021 at 11:40