Document number: *P **[TO-BE-CONFIRMED]** R0*

Date: *2022-09-14*

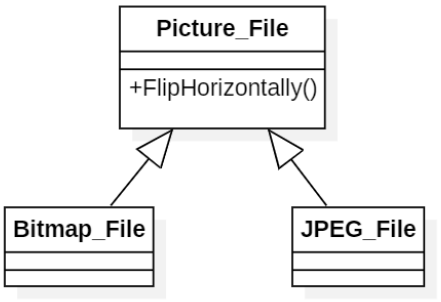
Audience: Library Evolution Working Group

Reply-to: *T. P. K. Healy <tho8ma8spkhe8aly at yah8oo dot co8m> Remove all 8's from address*

I. Table of Contents

II. Introduction

*Where all the types of an **std::variant** share a common base class, allow the **variant** object to be accessed as though it were a pointer to the base class. Taking the following class hierarchy as an example:*



This aim of this proposal is to allow programmers to write code as follows:

```
variant< specify_base<Picture_File>, Bitmap_File, JPEG_File > object;

object.emplace<Bitmap_File>("birthday_party.bmp");

object->FlipHorizontally();
```

III. Motivation and Scope

This proposed change to ***std::variant*** allows more simplistic management of a ***variant*** object whose types all share a common base class, allowing for more simple syntax. It is particularly useful for projects where there is a motivation to avoid the '***new***' operator, for example working on a microcontroller with 92 kilobytes of SRAM, or in a time-critical environment where heap manipulation is to be kept to an absolute minimum.

IV. Impact On the Standard

This proposed change to ***std::variant*** will have no consequences for the core language nor for any other classes in the standard library. Old code will not be affected. There is full backward compatibility.

V. Design Decisions

If a program decides a runtime which derived class to use, a global pointer to the common base class can be used as follows:

```
Picture_File *picfile = nullptr;

void Initialise_Imaging_Sytem(bool const compressed = false)
{
    picfile = compressed ? new JPEG_File : new Bitmap_File;
}

int main(void)
{
     Initialise_Imaging_Sytem(true);
     picfile->Open("birthday_cake.jpg");
}
```

The above code snippet uses dynamic memory allocation. To achieve this without dynamic memory allocation, an alternative in C++20 would be:

```
Picture_File *picfile = nullptr;

variant<monostate, Bitmap_File, JPEG_File> vobj;

void Initialise_Imaging_System(bool const compressed = false)
{
   picfile = compressed ? &vobj.emplace<JPEG_File>() : &vobj.emplace<Bitmap_File>();
}

int main(void)
{
    Initialise_Imaging_Sytem(true);
    picfile->Open("birthday_cake.jpg");
}
```

With the proposed change in this paper to the ***std::variant*** class, the above code snippet could be simplified to:

```
variant< specify_base<Picture_File>, Bitmap_File, JPEG_File > picfile;

void Initialise_Imaging_System(bool const compressed = false)
{
    if ( compressed ) picfile.emplace<JPEG_File>(); else picfile.emplace<Bitmap_File>();
}

int main(void)
{
   Initialise_Imaging_Sytem(true);
   picfile->Open("birthday_cake.jpg");
}
```

VI. Technical Specifications

The helper class "***std::specify_base***" shall be defined as follows:

```
#include <type_traits>  // remove_cvref_t, is_class_v

template<class Base>
struct specify_base {
   typedef std::remove_cvref_t<Base> type;
   static_assert( std::is_class_v<type> );
   static bool constexpr throws_bad_variant_access = true;
};
```

There shall also be an alternative helper class "***std::specify_base_nothrow***" which causes a ***nullptr*** to be returned instead of throwing "***std::bad_variant_access"***, defined as follows:

```
template<class Base>
struct specify_base_nothrow {
   typedef std::remove_cvref_t<Base> type;
   static_assert( std::is_class_v<type> );
   static bool constexpr throws_bad_variant_access = false;
};
```

The class "***std::variant***" shall be amended as follows:

```
#include <cstddef>      // size_t
#include <concepts>     // derived_from
#include <type_traits>  // is_class_v, remove_cvref_t
#include <tuple>        // tuple, tuple_element_t

namespace detail {

   template<class T, bool has_subtype>
   struct subtype_or_void {
      typedef void type;
   };

   template<class T>
   struct subtype_or_void<T, true> {
      typedef std::remove_cvref_t<typename T::type> type;
   };

   template<class T, bool has_subtype>
   using subtype_or_void_t = typename subtype_or_void<T, has_subtype>::type;

   template<class T>
   inline bool constexpr is_base_specified = false;

   template<class T>
   inline bool constexpr is_base_specified< specify_base<T> > = true;

   template<class T>
   inline bool constexpr is_base_specified< specify_base_nothrow<T> > = true;

   template<std::size_t N, typename... T, std::size_t... I>
   std::tuple< std::tuple_element_t< N+I, std::tuple<T...> >... > sub(std::index_sequence<I...>);

   template<std::size_t N, typename... T>
   using subpack = decltype(sub<N, T...>(std::make_index_sequence<sizeof...(T) - N>{}));
}  // close namespace 'detail'
```

```
template<class... Types>
class variant {

   // All of the original definition of std::variant goes here verbatim

private:

   using FirstType = std::tuple_element_t< 0u, std::tuple<Types...> >;

public:

   static bool constexpr is_common_base_specified = detail::is_base_specified<FirstType>;

   typedef detail::subtype_or_void_t<FirstType, is_common_base_specified> Base;

   typedef detail::subpack<1u, Types...> AllTypesExceptFirst;

   // The next line makes sure that all classes are derived from Base
   static_assert(    !is_common_base_specified
                  || []<typename... T>(std::type_identity< std::tuple<T...> >)
                  {
                     return (std::derived_from<T,Base> && ...);
                  }(std::type_identity<AllTypesExceptFirst>{}),
                  "All types must inherit publicly from Base");

private:

  static Base const volatile *detail_common_base(std::variant<Types...> const volatile *const arg)
   {
        static_assert(detail::is_base_specified<FirstType>,
                              "To invoke 'common_base()', the first type must "
                              "be a specialisation of 'specify_base'");

        std::variant<Types...> *const p = const_cast< std::variant<Types...> * >(arg);

        auto my_lambda = []<class U>(U &u) -> Base*
          {
                if constexpr ( detail::is_base_specified<U> )  // If no object is currently hosted
                {
                        if constexpr ( U::throws_bad_variant_access ) throw std::bad_variant_access();
                        return nullptr;
                }
                else
                {
                        static_assert( std::derived_from<U,Base>,
                                        "Base class specified to std::variant< std::specify_base<T>, "
                                        "MoreTypes... > is not a base class of the currently hosted object");

                                return &u;
                }
        };

        return std::visit<Base*>(my_lambda, *p);
   }
```

```
public:

#define CONST_CAST_COMMON_BASE(modifiers, name_of_method)            \
   Base modifiers *name_of_method(void) modifiers                    \
   {                                                                 \
      return const_cast<Base modifiers*>( detail_common_base(this) );\
   }

#define CONST_CAST_COMMON_BASE_FOUR(name_of_method)                  \
   CONST_CAST_COMMON_BASE(             , name_of_method )            \
   CONST_CAST_COMMON_BASE(       const, name_of_method )             \
   CONST_CAST_COMMON_BASE(    volatile, name_of_method )             \
   CONST_CAST_COMMON_BASE(const volatile, name_of_method )

   CONST_CAST_COMMON_BASE_FOUR(common_base)
   CONST_CAST_COMMON_BASE_FOUR(operator->)

#undef CONST_CAST_COMMON_BASE_FOUR
#undef CONST_CAST_COMMON_BASE

};
```

And here is a sample program to test it out:

```
#include <variant>     // variant, specify_base, specify_base_nothrow
#include <iostream>  // cout

struct Mammal { virtual void Speak(void) const volatile = 0; };
struct Dog : Mammal { void Speak(void) const volatile override { std::cout << "Dog\n"; } };
struct Cat : Mammal { void Speak(void) const volatile override { std::cout << "Cat\n"; } };
struct Fish {};  // not a mammal!

std::variant< std::specify_base<Mammal>, Dog, Cat > my_mammal;

int main(void)
{
   my_mammal.emplace<2u>();

   my_mammal->Speak();

   // On the next line we can make a const volatile object too
   std::variant< std::specify_base_nothrow<Mammal>, Dog, Cat > const volatile my_cvmammal;
   my_cvmammal->Speak();  // This will throw 'bad_variant_access'

   // The next line will fail to compile because a fish isn't a mammal
   std::variant< specify_base<Mammal>, Dog, Cat, Fish > volatile some_other_object;

   std::variant<int,double,long> abc;  // This is unaffected
}
```

VII. Acknowledgements **This is just as draft**

VIII. References **This is just as draft**