Subject: Doubt with Buffer<> of a trivially destructible type
Posted by koldo on Thu, 12 Dec 2024 11:58:14 GMT
View Forum Message <> Reply to Message

Hello all

Compiling this code with CLANG, I get the error: "object expression of non-scalar type 'double[3]' cannot be used in a pseudo-destructor expression" in the line with Buffer.
However, vector3_destructible gets true.
I wanted to ask you how to use Buffer for such a data type, or if there is a better dynamic container.

```
typedef double Vector3[3];

bool vector3_destructible = std::is_trivially_destructible<Vector3>::value;

Buffer<Vector3> vector3_data(10);
```

Subject: Re: Doubt with Buffer<> of a trivially destructible type
Posted by Oblivion on Thu, 12 Dec 2024 13:15:23 GMT
View Forum Message <> Reply to Message

Hello Iñaki,

Quote:
However, vector3_destructible gets true.

Yes, because std::is_trivially_destructible removes all extents (dimensions) of the object in question before its type gets evaluated. So, basically it is returning true for double type.

However, Buffer<Vector3> doesn't. So it expands into Buffer<double[3]>, hence the error.

I think you should instead prefer -if possible- a structure or Buffer<double> mybuffer(count * 3).

Then again, I don't know the exact requirements of your code.

Best regards,
Oblivion

Subject: Re: Doubt with Buffer<> of a trivially destructible type
Posted by Oblivion on Thu, 12 Dec 2024 13:40:45 GMT
View Forum Message <> Reply to Message

However, this should work too:

```
CONSOLE_APP_MAIN
{
 typedef std::array<double, 3> Vector3;

 Buffer<Vector3> vector3_data(10, {1.0, 2.0, 3.0});

 for(int i = 0; i < 10; i++) {
  auto a = vector3_data[i];
  Cout() << a[0] << " "<<  a[1] << " " << a[2] << "\r\n";
 }

}
```

Or, C++17 style:

```
 typedef std::array<double, 3> Vector3;

 Vector<Vector3> vector3_data(10, {1.0, 2.0, 3.0});

 for(auto& [a, b, c] : vector3_data) {
  Cout() << a << " "<<  b << " " << c << "\r\n";
 }
```

Best regards,
Oblivion

Subject: Re: Doubt with Buffer<> of a trivially destructible type
Posted by koldo on Thu, 12 Dec 2024 15:05:45 GMT
View Forum Message <> Reply to Message

Thank you Oblivion

Yes, there are simple solutions. This case is to alloc data to be filled by a DLL made in Delphi, with particular alignment requirements.

I have reviewed deeper in how U++ handles this, and it takes care of that in run time:

```
void Free() {
 if(ptr) {
```

```
  if(std::is_trivially_destructible<T>::value)  // <<========
   MemoryFree(ptr);
  else {
   void *p = (byte *)ptr - 16;
   size_t size = *(size_t *)p;
   Destroy(ptr, ptr + size);            // <<======== Destroy() is called only if there is a destructor
   MemoryFree(p);
  }
 }
}
```

However the compiler detects this:

```
template <class T>
inline void Destruct(T *t)
{
 t->~T();                       // error: object expression of non-scalar type 'double[3]' cannot be used in
a pseudo-destructor expression
}

template <class T>
inline void Destroy(T *t, const T *end)
{
 while(t != end)
  Destruct(t++);
}
```

Subject: Re: Doubt with Buffer<> of a trivially destructible type
Posted by Lance on Sat, 14 Dec 2024 18:19:10 GMT
View Forum Message <> Reply to Message

This is a valid use case, and something need to be addressed in u++ libary directly instead of circling around.

Here is a simple utility we can use to fix it from with u++ libary

```
template <typename T, std::size_t...>
constexpr auto object_count(T& t)
{
 return 1u;
}

template <std::size_t ... Ns, typename T, std::size_t n>
constexpr auto object_count(T (&arr)[n] )
{
 return n * object_count(arr[0]);
}
```

```
// eg, with
 double d;
 double a1[5];
 double a2[5][3];
 double a3[5][4][2];

 // then
 static_assert(object_count(d)==1,"?");
 static_assert(object_count(a1)==5,"?");
 static_assert(object_count(a2)==15,"?");
 static_assert(object_count(d3)==40,"?);
```

With above utility, we can easily modify u++ Vector to accomodate c style array.

basially, if T is trivially relocatible, then
any c array of T is alos trivially relocatible,
Upp::Vector don't care any detail of c array, except the total number of T objects in the array to properly construct, move and destruct it, with T::~T() properly defined of course.

Oh, for

```
// some type T
T d3[3][2][5];
```

a simple

```
sizeof(d3)/sizeof(T)
```

will do the job

Subject: Re: Doubt with Buffer<> of a trivially destructible type
Posted by koldo on Sat, 14 Dec 2024 19:45:35 GMT
View Forum Message <> Reply to Message

Hi Lance

Thank you for your response.
I have to say that std::vector has the same problem, but it is also true that we like to do things better than the standard library.

One difficulty we have with CLANG is that although in U++ we try to handle the problem at run time, CLANG detects the problem at compile time.
Therefore, the solution has to work at compile time.

Subject: Re: Doubt with Buffer<> of a trivially destructible type
Posted by Lance on Sat, 14 Dec 2024 19:53:22 GMT
View Forum Message <> Reply to Message

I believe this is something that should and can be addressed of at compile time by making Upp::Vector more robust.

I am surprised std::vector is guilty of the same problem. I will take a look to see if more recent standard library has fixed it.

Subject: Re: Doubt with Buffer<> of a trivially destructible type
Posted by Lance on Sat, 14 Dec 2024 20:02:51 GMT
View Forum Message <> Reply to Message

Hello Koldo,

It should not matter whether T is trivially destructible. As long as T is Upp::Moeavable, a type of fixed sized c array of T should be Vector friendly.

For example ,

typedef Upp::String S[3];

Upp::Vector<S3> s3s;

should compile and behave fine with a c array-awared Vector.

Subject: Re: Doubt with Buffer<> of a trivially destructible type
Posted by Lance on Sat, 14 Dec 2024 20:37:51 GMT
View Forum Message <> Reply to Message

My bad. I was wrong. a c array cannot be returned from a function; this disqualifies a c-array be used directly in a vector type (std::vector or Upp::Vector).

And I just noticed that I am completely off topic

Subject: Re: Doubt with Buffer<> of a trivially destructible type
Posted by koldo on Sun, 15 Dec 2024 11:25:39 GMT
View Forum Message <> Reply to Message

Lance wrote on Sat, 14 December 2024 20:53I believe this is something that should and can be addressed of at compile time by making Upp::Vector more robust.

I am surprised std::vector is guilty of the same problem. I will take a look to see if more recent standard library has fixed it.
Yes. This code gives the same error:

```
CONSOLE_APP_MAIN
{
 typedef double Vector3[3];

 std::vector<Vector3> vector3_data(10);
}
```

Because of this in file clang\include\c++\v1__memory\allocator.h:

```
_LIBCPP_DEPRECATED_IN_CXX17 _LIBCPP_HIDE_FROM_ABI void destroy(pointer __p) {
__p->~_Tp(); }
```

Subject: Re: Doubt with Buffer<> of a trivially destructible type
Posted by koldo on Sun, 15 Dec 2024 11:45:27 GMT
View Forum Message <> Reply to Message

The solution for U++ could be any of these, applied to class Buffer<> functions Malloc() and Free():

- For C++ 17, just insert constexpr in the std::is_trivially_destructible check, to force it to work in compile time:

```
if constexpr (std::is_trivially_destructible<T>::value)
```

- In C++ it is a little more complicated as it requires a little of SFINAE. This way the std::is_trivially_destructible check works in compile time:

```
 template <typename U = T>
 typename std::enable_if<!std::is_trivially_destructible<U>::value>::type
 Malloc(size_t size) {
  void* p = MemoryAlloc(size * sizeof(U) + 16);
  *(size_t*)p = size;
  ptr = (U*)((byte*)p + 16);
 }
 template <typename U = T>
 typename std::enable_if<std::is_trivially_destructible<U>::value>::type
 Malloc(size_t size) {
  ptr = (U*)MemoryAlloc(size * sizeof(U));
 }

 template <typename U = T>
 typename std::enable_if<!std::is_trivially_destructible<U>::value>::type
 Free() {
  if (ptr) {
   void* p = (byte*)ptr - 16;
   size_t size = *(size_t*)p;
   Destroy(ptr, ptr + size);
```

```
  MemoryFree(p);
 }
}
template <typename U = T>
typename std::enable_if<std::is_trivially_destructible<U>::value>::type
Free() {
 if (ptr) {
  MemoryFree(ptr);
 }
}
```

Subject: Re: Doubt with Buffer<> of a trivially destructible type
Posted by Lance on Sun, 15 Dec 2024 13:29:13 GMT
View Forum Message <> Reply to Message

The inablility for Vector to desctruct T[n] is a consequence of inherited inability for a function to return a c array.

For example

```
T Vector::pop();
```

requires the housed type T be returnable.

There is no reason for c++ to systematically disallow it other than c says so, and there is liitle or no incentive for c++ to do otherwise, as it can be easily circled around.

We can have our Vector to work with T[n] if it's really really desirable, by encapuslating it in an intermediate struct Vector::ArrayWrapper automatically.
ArrayWrapper is contructible from, assignable from, convertible to the array type.

There is some work involved. And the requirement is less common to make it worthy.

Subject: Re: Doubt with Buffer<> of a trivially destructible type
Posted by mirek on Thu, 26 Dec 2024 17:47:34 GMT
View Forum Message <> Reply to Message

- For C++ 17, just insert constexpr in the std::is_trivially_destructible check, to force it to work in compile time:
if constexpr (std::is_trivially_destructible<T>::value)

We are now C++17 -> used this one

Subject: Re: Doubt with Buffer<> of a trivially destructible type
Posted by koldo on Fri, 27 Dec 2024 17:34:01 GMT
View Forum Message <> Reply to Message

mirek wrote on Thu, 26 December 2024 18:47- For C++ 17, just insert constexpr in the std::is_trivially_destructible check, to force it to work in compile time:
if constexpr (std::is_trivially_destructible<T>::value)

We are now C++17 -> used this oneThank you Mirek. Problem solved!