19.5 — Void pointers

ALEX[1] SEPTEMBER 29, 2023

The **void pointer**, also known as the generic pointer, is a special type of pointer that can be pointed at objects of any data type! A void pointer is declared like a normal pointer, using the void keyword as the pointer's type:

```cpp
void* ptr {}; // ptr is a void pointer
```

A void pointer can point to objects of any data type:

```cpp
int nValue {};
float fValue {};

struct Something
{
    int n;
    float f;
};

Something sValue {};

void* ptr {};
ptr = &nValue; // valid
ptr = &fValue; // valid
ptr = &sValue; // valid
```

However, because the void pointer does not know what type of object it is pointing to, dereferencing a void pointer is illegal. Instead, the void pointer must first be cast to another pointer type before the dereference can be performed.

```cpp
int value{ 5 };
void* voidPtr{ &value };

// std::cout << *voidPtr << '\n'; // illegal: dereference of void pointer

int* intPtr{ static_cast<int*>(voidPtr) }; // however, if we cast our void pointer to an int pointer...

std::cout << *intPtr << '\n'; // then we can dereference the result
```

This prints:

```
5
```

The next obvious question is: If a void pointer doesn't know what it's pointing to, how do we know what to cast it to? Ultimately, that is up to you to keep track of.

Here's an example of a void pointer in use:

```cpp
#include <cassert>
#include <iostream>

enum class Type
{
    tInt, // note: we can't use "int" here because it's a keyword, so we'll use "tInt" instead
    tFloat,
    tCString
};

void printValue(void* ptr, Type type)
{
    switch (type)
    {
    case Type::tInt:
        std::cout << *static_cast<int*>(ptr) << '\n'; // cast to int pointer and perform indirection
        break;
    case Type::tFloat:
        std::cout << *static_cast<float*>(ptr) << '\n'; // cast to float pointer and perform indirection
        break;
    case Type::tCString:
        std::cout << static_cast<char*>(ptr) << '\n'; // cast to char pointer (no indirection)
        // std::cout will treat char* as a C-style string
        // if we were to perform indirection through the result, then we'd just print the single char that ptr is pointing to
        break;
    default:
        std::cerr << "printValue(): invalid type provided\n";
        assert(false && "type not found");
        break;
    }
}

int main()
{
    int nValue{ 5 };
    float fValue{ 7.5f };
    char szValue[]{ "Mollie" };

    printValue(&nValue, Type::tInt);
    printValue(&fValue, Type::tFloat);
    printValue(szValue, Type::tCString);

    return 0;
}
```

This program prints:

```
5
7.5
Mollie
```

Void pointer miscellany

Void pointers can be set to a null value:

```cpp
void* ptr{ nullptr }; // ptr is a void pointer that is currently a null pointer
```

Because a void pointer does not know what type of object it is pointing to, deleting a void pointer will result in undefined behavior. If you need to delete a void pointer, `static_cast` it back to the appropriate type first.

It is not possible to do pointer arithmetic on a void pointer. This is because pointer arithmetic requires the pointer to know what size object it is pointing to, so it can increment or decrement the pointer appropriately.

Note that there is no such thing as a void reference. This is because a void reference would be of type void &, and would not know what type of value it referenced.

Conclusion

In general, it is a good idea to avoid using void pointers unless absolutely necessary, as they effectively allow you to avoid type checking. This allows you to inadvertently do things that make no sense, and the compiler won't complain about it. For example, the following would be valid:

```
int nValue{ 5 };
printValue(&nValue, Type::tCString);
```

But who knows what the result would actually be!

Although the above function seems like a neat way to make a single function handle multiple data types, C++ actually offers a much better way to do the same thing (via function overloading) that retains type checking to help prevent misuse. Many other places where void pointers would once be used to handle multiple data types are now better done using templates, which also offer strong type checking.

However, very occasionally, you may still find a reasonable use for the void pointer. Just make sure there isn't a better (safer) way to do the same thing using other language mechanisms first!

Quiz time

Question #1

What's the difference between a void pointer and a null pointer?

Show Solution (javascript:void(0))[2]

Next lesson
20.1 Function Pointers

3

Back to table of contents

4

Previous lesson
19.4 Pointers to pointers and dynamic multidimensional arrays

5

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[9] are connected to your provided email address.

176 COMMENTS Newest

PooperShooter
May 22, 2025 3:52 pm PDT

Void pointers are everywhere when working with windows API

0 Reply

johnooooooooo
Reply to PooperShooter[10] July 1, 2025 12:46 pm PDT

Think because C does not have templates so there would have been no easier way for functions to accept various pointer types without writing a tonne of repeating functions

0 Reply

Guy
May 17, 2025 12:53 pm PDT

if you don't need to change the type later, couldn't you just use auto* to do that?

0 Reply

jyc
March 13, 2025 6:16 am PDT

Just wanted to add that one should avoid doing something like this:

```cpp
char someVal = 'a';
void* ptr = &someVal;
int* integerCast = static_cast<int*>(ptr); //don't do this
std::cout << *integerCast; //this may crash depending on the target
```

As SEI CERT (https://wiki.sei.cmu.edu/confluence/display/c/EXP36-C.+Do+not+cast+pointers+into+more+strictly+aligned+pointer+types)[11] states: "Do not convert a pointer value to a pointer type that is more strictly aligned than the referenced type. Different alignments are possible for different types of objects. ... If the misaligned pointer is dereferenced, the program may terminate abnormally."

1 Reply

Lijus TorTorvalds
September 15, 2024 3:40 am PDT

Ex-Google, Ex-Facebook, Ex-Airbnb, Ex-CIA employee here. Never use templates, use void pointers instead. C++ will generate 100 bazillion functions when declaring a template, while a void pointer will only need ONE. 100x SPEEDUP and 10^(-5)% of the bloat, THANK ME LATER

8 Reply

NordicCat
Reply to Lijus TorTorvalds[12] January 10, 2025 2:47 am PST

agreed!

0 Reply

EmtyC
Reply to Lijus TorTorvalds[12] December 6, 2024 8:16 am PST

Ex-CIA ? name every confidential fact

2 Reply

Md Nishad
Reply to Lijus TorTorvalds[12] September 28, 2024 8:14 am PDT

Why isn't there a dislike button

1 Reply

Alex Author
Reply to Md Nishad[13] September 30, 2024 5:10 pm PDT

So people like you can't use it. :) He's ex-CIA, he'd probably stage a coup in your country for disagreeing.

I don't agree with his conclusion, but templates can definitely bloat your code.

29 Reply

WinkyWonka
November 8, 2023 2:57 am PST

Some fun template:

```
template<typename T>
void printVoidPtr(void* ptr)
{
    std::cout << "Void pointer is = " << ptr << " = " << &ptr << " = " <<
*static_cast<T*>(ptr) << " \n";
}
```

and for char

```
template<>
void printVoidPtr<char>(void* ptr)
{
    std::cout << "Void pointer is = " << ptr << " = " << &ptr << " = " <<
static_cast<char*>(ptr) << " \n";

}
```

Unsure how to do a template with the struct something thought.

```
template<typename T, typename U>
void printVoidPtr<Something>(void* ptr)
{
    std::cout << "Void pointer is = " << ptr << " = " << &ptr << " = " <<
*static_cast<Something<T,U>*>(ptr) << " \n";
}
```

as this doesn't work.

Dear Alex, do you have any advice regarding the template deduction for a struct?

0 Reply

Tomáš Nadrchal
Reply to WinkyWonka 14 February 27, 2025 6:27 am PST

You do not need specialized templates. It will work as templated class is just a type. You just need in this case overloading ostream operator << for that class.

```cpp
#include <iostream>
#include <utility>

template<typename T>
auto printVoidPtr(void* ptr) -> void
{
    std::cout << "Void pointer is = " << ptr << " = " << &ptr << " = " <<
*static_cast<T*>(ptr) << " \n";
}

template <typename T>
struct Type
{
    T value;
    friend std::ostream& operator << (std::ostream& out, Type& type)
    {
        std::cout << type.value;
        return out;
    }
};

auto main () -> int
{
    Type<int> c {1};
    using TypeC = decltype(c);
    Type<int>* cPtr = &c;
    printVoidPtr<TypeC>(cPtr);

    return 0;
}
```

This just works and print correctly 1. I used decltype to show that it can easily used to complicated types. But it is still bad way to use as there is no type checking.

0 Reply

Alex Author

Reply to WinkyWonka 14 November 10, 2023 2:01 pm PST

I think the issue here is that Something is a class template itself, so you have nested templates. My template-fu isn't strong enough to advise here.

1 Reply

WinkyWonka

Reply to WinkyWonka 14 November 8, 2023 2:57 am PST

The CTAD is also added

```cpp
template<typename T, typename U>
Something(T, U) -> Something<T, U>;
```

0 Reply

VexedBannister

September 28, 2023 1:11 pm PDT

> "Although some compilers allow deleting a void pointer that points to dynamically allocated memory, doing so should be avoided, as it can result in undefined behavior."

You probably omitted explaining this further for a reason. But, if you don't mind me asking, how?

```
void* ptr { new int };

delete ptr;
```

Does `delete` need the right pointer type to know how much memory to release?

Last edited 1 year ago by VexedBannister

0 Reply

Alex Author

Reply to VexedBannister 15 September 29, 2023 12:11 pm PDT

The technically correct answer is because the C++ standard says so.

> Does delete need the right pointer type to know how much memory to release?

Generally, yes. And also to call the correct destructor for class types.

1 Reply

Timo

August 17, 2023 11:28 am PDT

```
#include <cassert>
#include <iostream>

enum class Type
{
    tInt, // note: we can't use "int" here because it's a keyword, so we'll use "tInt"
instead
    tFloat,
    tCString
};

void printValue(void* ptr, Type type)
{
    switch (type)
    {
    case Type::tInt:
        std::cout << *static_cast<int*>(ptr) << '\n'; // cast to int pointer and
perform indirection
        break;
    case Type::tFloat:
        std::cout << *static_cast<float*>(ptr) << '\n'; // cast to float pointer and
perform indirection
        break;
    case Type::tCString:
        std::cout << static_cast<char*>(ptr) << '\n'; // cast to char pointer (no
indirection)
        // std::cout will treat char* as a C-style string
        // if we were to perform indirection through the result, then we'd just print
the single char that ptr is pointing to
        break;
    default:
        std::cerr << "printValue(): invalid type provided\n";
        assert(false && "type not found");
        break;
    }
}

int main()
{
    int nValue{ 5 };
    float fValue{ 7.5f };
    char szValue[]{ "Mollie" };

    printValue(&nValue, Type::tInt);
    printValue(&fValue, Type::tFloat);
    printValue(szValue, Type::tCString);

    return 0;
}
```

You could also just assign auto to the pointer parameter object instead of void and then dereference based on the received pointer datatype and do some error handling based if ptr is void. I guess it was just as example but the code seems complex :D

0 Reply

Alex Author

Reply to Timo 16 August 17, 2023 9:55 pm PDT

Yeah, templates are nice, aren't they?

`void*` was how this kind of stuff used to be done in C, where templates don't exist.

4 Reply

Krishnakumar
July 25, 2023 11:15 am PDT

> Although some compilers allow deleting a void pointer that points to dynamically allocated memory, doing so should be avoided, as it can result in undefined behavior.

Therefore, it would be important to do something like this, no?

```
if (ptr) {
    delete ptr;
}
```

Last edited 1 year ago by Krishnakumar

0 Reply

Alex Author
Reply to Krishnakumar [17] July 28, 2023 6:21 pm PDT

No, that's doing a null pointer check, not a void pointer check.

If you want to delete the object a void pointer is pointing to, cast the void pointer back to a pointer to the object's type and then delete that.

2 Reply

Suku_go
June 22, 2023 1:19 am PDT

Which one should I prefer to use in a switch-case statement, Break or Return if the results are both the same ?
The code below achieved the similar result when using 'return' instead of 'break'. And I didn't see any bad side effect.

```
#include <iostream>
#include <cassert>

enum class Type
{
    tInt, // note: we can't use "int" here because it's a keyword, so we'll use "tInt" instead
    tFloat,
    tCString
};

void printValue(void* ptr, Type type)
{
    switch (type)
    {
    case Type::tInt:
        std::cout << *static_cast<int*>(ptr) << '\n'; // cast to int pointer and perform indirection
        return;
    case Type::tFloat:
        std::cout << *static_cast<float*>(ptr) << '\n'; // cast to float pointer and perform indirection
        return;
    case Type::tCString:
        std::cout << static_cast<char*>(ptr) << '\n'; // cast to char pointer (no indirection)
        // std::cout will treat char* as a C-style string
        // if we were to perform indirection through the result, then we'd just print the single char that ptr is pointing to
        return;
    default:
        assert(false && "type not found");
        return;
    }
}

int main()
{
    int nValue{ 5 };
    float fValue{ 7.5f };
    char szValue[]{ "Mollie" };

    printValue(&nValue, Type::tInt);
    printValue(&fValue, Type::tFloat);
    printValue(szValue, Type::tCString);

    return 0;
}
```

Last edited 2 years ago by Suku_go

0 Reply

Alex Author

Reply to Suku_go 18 June 23, 2023 9:35 pm PDT

If a case requires the function be exited (e.g. because it is an error case), use return. Otherwise, favor break. That way if you need to add code below the switch block, you don't have to rewrite your switch block.

1 Reply

Diamindleague

April 18, 2023 12:33 pm PDT

```
#include <iostream>
#include <cassert>

enum class Type
{
tInt, // note: we can't use "int" here because it's a keyword, so we'll use "tInt" instead
tFloat,
tCString
};

void printValue(void* ptr, Type type)
{
switch (type)
{
case Type::tInt:
std::cout << static_cast<int>(ptr) << '\n'; // cast to int pointer and perform indirection
break;
case Type::tFloat:
std::cout << static_cast<float>(ptr) << '\n'; // cast to float pointer and perform indirection
break;
case Type::tCString:
std::cout << static_cast<char*>(ptr) << '\n'; // cast to char pointer (no indirection)
// std::cout will treat char* as a C-style string
// if we were to perform indirection through the result, then we'd just print the single char that ptr is
pointing to
break;
default:
assert(false && "type not found");
break;
}
}

int main()
{
int nValue{ 5 };
float fValue{ 7.5f };
char szValue[]{ "Mollie" };

printValue(&nValue, Type::tInt);
printValue(&fValue, Type::tFloat);
printValue(szValue, Type::tCString);

return 0;

}
```

Hey nascardriver and Alex I'm having trouble interpreting this code please explain to me step by step if you can thank you in advance.

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. javascript:void(0)
3. https://www.learncpp.com/cpp-tutorial/function-pointers/
4. https://www.learncpp.com/
5. https://www.learncpp.com/cpp-tutorial/pointers-to-pointers/
6. https://www.learncpp.com/void-pointers/
7. https://www.learncpp.com/cpp-tutorial/function-parameters-and-arguments/
8. https://www.learncpp.com/cpp-tutorial/passing-arguments-by-value/
9. https://gravatar.com/
10. https://www.learncpp.com/cpp-tutorial/void-pointers/#comment-610318
11. https://wiki.sei.cmu.edu/confluence/display/c/EXP36-C.+Do+not+cast+pointers+into+more+strictly+aligned+pointer+types
12. https://www.learncpp.com/cpp-tutorial/void-pointers/#comment-601986
13. https://www.learncpp.com/cpp-tutorial/void-pointers/#comment-602449
14. https://www.learncpp.com/cpp-tutorial/void-pointers/#comment-589529
15. https://www.learncpp.com/cpp-tutorial/void-pointers/#comment-587923
16. https://www.learncpp.com/cpp-tutorial/void-pointers/#comment-585681
17. https://www.learncpp.com/cpp-tutorial/void-pointers/#comment-584471
18. https://www.learncpp.com/cpp-tutorial/void-pointers/#comment-582257
19. https://g.ezoic.net/privacy/learncpp.com