6.13 — Void pointers

BY ALEX ON JULY 19TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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:

```
void *ptr; // ptr is a void pointer
```

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

```
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, it cannot be dereferenced directly! Rather, the void pointer must first be explicitly cast to another pointer type before it is dereferenced.

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

// std::cout << *voidPtr << '\n'; // illegal: cannot dereference a void pointer

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

std::cout << *intPtr << '\n'; // then we can dereference it like normal
```

This prints:

5

The next obvious questions 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:

```
#include <iostream>

enum class Type
{
    INT,
    FLOAT,
    CSTRING
};

void printValue(void *ptr, Type type)
```

```
{
    switch (type)
    {
        case Type::INT:
            std::cout << *static_cast<int*>(ptr) << '\n'; // cast to int pointer and dereferen
            break;
        case Type::FLOAT:
            std::cout << *static_cast<float*>(ptr) << '\n'; // cast to float pointer and deref
            break;
        case Type::CSTRING:
            std::cout << static_cast<char*>(ptr) << '\n'; // cast to char pointer (no derefere
            // std::cout knows to treat char* as a C-style string
            // if we were to dereference the result, then we'd just print the single char that
            break;
    }
}

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

    printValue(&nValue, Type::INT);
    printValue(&fValue, Type::FLOAT);
    printValue(szValue, Type::CSTRING);

    return 0;
}
```

This program prints:

```
5
7.5
Mollie
```

Void pointer miscellany

Void pointers can be set to a null value:

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

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.

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::CSTRING);
```

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

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

Quiz answers

1) **Show Solution**

 **6.14 -- Pointers to pointers and dynamic multidimensional arrays**

 **Index**

 **6.12a -- For-each loops**

C++ TUTORIAL | PRINT THIS POST

86 comments to 6.13 — Void pointers

« Older Comments 1 2

Toshi

November 18, 2019 at 9:47 am · Reply

I find it hard to understand the casting part, any references for learning that would be greatly appreciated

nascardriver
November 19, 2019 at 1:17 am · Reply

Make sure you understood lesson S.4.4a (Explicit type conversion).
After that, it's quite simple. First, we cast the `void*` to another pointer type, eg. `int*`. Now that we have an `int*`, we can dereference it with the `*` operator and get the value that the pointer is pointing to.

```
1  void* p{ /*...*/ };
2
3  // Convert the void pointer to an int pointer
4  int* pI{ static_cast<int*>(p) };
5
6  // Use operator* to get the value
7  int i{ *pI };
8
9  // We can do it in one line without a pI
10 int i2{ *(static_cast<int*>(p)) };
11
12 // And without parentheses
13 int i3{ *static_cast<int*>(p) };
```

Miroslav Avramov
November 17, 2019 at 3:37 am · Reply

They're useful mainly when you want a generic way to point at something but will disambiguate what that is via some other mechanism. They were used more in the C world, as a way to do a primitive form of polymorphism. C++ has better mechanisms for such things (like templates), so they're not used much in C++.

Could you give examples to prove this statement? Void * and Typedef have been using from B.Kernigan & D. Ritchie book "The C" till now. Void * is great thing when system doesn't know what type of data user will send, I think. Typedef is use to create new data type name.

alfonso
October 23, 2019 at 11:43 pm · Reply

"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."

It is pretty much a similar situation with dereferencing a void pointer. You can't do it directly. But you can do it somehow.

```
1  #include <iostream>
2
3  int main () {
4      int array [] {4, 5, 6, 7};
5      void* ptr {array};
6
7      std::cout << *static_cast <int*> (ptr) << '\n';
8      std::cout << *(static_cast <int*> (ptr) + 1) << '\n';
9
```

```
10      return 0;
11  }
```

alfonso
October 23, 2019 at 1:45 am · Reply

I understand the purpose of the example code here. But in 'real life' isn't better to have three explicit functions like printInt (), printFloat () and printCstring ()? The function printValue () must iterate at every call just to find the functionality for the case that we already know at the call moment: "I have an int and I want to print an int." So why to search for a printInt () functionality through a list of functionalities?

nascardriver
October 23, 2019 at 3:31 am · Reply

You wouldn't write code like this in practice at all, as there are safer alternatives (eg. `std::variant`), but it's not easy to come up with a short example that requires the use of `void*`.
You're right, it'd be better to call the correct function right away if the type is known. You'll see code like this in cases where the type is unknown at compile-time. This happens when you load the values and a type id or type name from a database for example. You'll have a value and only an id of the type, but you can't know which type it is at compile-time.

hassan magaji
January 24, 2019 at 10:33 pm · Reply

hi everyone,
i understood the concept of void pointers quiet well but it seems to me like they're less useful, if you have to cast them everytime to a known data type why didnt you declared it to be of that data type in the first place.
I wish to know more about their significance if i missed one.

nice tutorials by the way.

Alex
January 27, 2019 at 3:31 pm · Reply

They're useful mainly when you want a generic way to point at something but will disambiguate what that is via some other mechanism. They were used more in the C world, as a way to do a primitive form of polymorphism. C++ has better mechanisms for such things (like templates), so they're not used much in C++.

Glory Pachnanda
January 21, 2019 at 5:30 pm · Reply

Your article clears my confusion on void pointer, it really helps me. Thank you so much to share this useful information.

Louis Cloete
January 18, 2019 at 9:12 pm · Reply

I have a slightly unrelated question. I have been wondering how to write my own function to truncate a floating point number. I used type float for the parameter, since I didn't want it to be

able to be too large to fit into an int. Here is my code:

```
// Truncates a 32-bit floating point number encoded according to the IEEE 754 standard.
// I assume the number is constructed as:
// 1 sign bit
// 8 exponent bits
// 23 mantissa bits
// The function truncates toward 0.
int trunc(float f)
{
    if (f > -1.0f && f < 1.0f)
        return 0;

    // The mantissa in a 32-bit IEEE 754 float is the 23 least significant bits.
    static constexpr int mantissaLength { 23 };

    // Copy the argument to retain all info even after manipulating the number and
    // reinterpret the 32 bits in memory as an int to enable bitshifts.
    unsigned int exp { *reinterpret_cast<unsigned int*>(&f) };

    // Shift the bits so that the exponent is in the eight least significant bits.
    exp >>= mantissaLength;

    // Mask out the 8 least signinficant digits of the number
    // to eliminate the sign bit, if any.
    exp &= 0xffu;

    // exp should be >= 127 by now, since we handled the cases of a negative exp
    // already by testing if -1 < f < 1. We can thus safely subtract 127 from it
    // without having overflow to get the actual positive value of the exponent.
    exp -= 127u;

    static constexpr int signPos { 31 };

    // sign is true if the number is negative.
    bool sign { *reinterpret_cast<unsigned int*>(&f) >> signPos };

    // If the exponent is zero, the truncated number must be +/- 1, since there
    // can only be one digit before the point and it must be non-zero. In
    // binary, that means it must be a 1. The number is also not shifted left
    // or right by a non-zero exponent. Thus the final answer must be +/- 1.
    if (exp == 0)
        return sign ? -1 : 1;

    // Copy the argument to retain all info even after manipulating the number and
    // reinterpret the 32 bits in memory as an int to enable bitshifts.
    unsigned int mantissa { *reinterpret_cast<unsigned int*>(&f) };

    // Extract the mantissa and flip the other bits to 0.
    mantissa &= 0x7fffffu;

    // Add the implied '1' before the decimal point back
    mantissa |= 0x800000u;

    // The number needs to be bitshifted exp places to the left and
    // mantissaLength places to the right to place the point at the proper pos.
    int shiftMagnitude { mantissaLength - static_cast<int>(exp) };

    // The sign of shiftMagnitude determines whether the direction of the shift
    shiftMagnitude > 0 ? mantissa >>= shiftMagnitude : mantissa <<= -shiftMagnitude;
    return sign ? -(static_cast<int>(mantissa)) : static_cast<int>(mantissa);
}
```

I used reinterpret casts, but upon reviewing the section on void pointers, I realised I could've used void pointers as well:

```
// Truncates a 32-bit floating point number encoded according to the IEEE 754 standard.
// I assume the number is constructed as:
// 1 sign bit
// 8 exponent bits
// 23 mantissa bits
// The function truncates toward 0.
int trunc(float f)
{
    if (f > -1.0f && f < 1.0f)
        return 0;

    // The mantissa in a 32-bit IEEE 754 float is the 23 least significant bits.
    static constexpr int mantissaLength { 23 };

    // Copy the argument to retain all info even after manipulating the number and
    // reinterpret the 32 bits in memory as an int to enable bitshifts.
    void *expPtr { static_cast<void*>(&f) };
    unsigned int exp { *static_cast<unsigned int*>(exp) };

    // Shift the bits so that the exponent is in the eight least significant bits.
    exp >>= mantissaLength;

    // Get the remainder of exp divided by 256 to eliminate the sign bit, if any.
    exp &= 0xffu;

    // exp should be >= 127 by now, since we handled the cases of a negative exp
    // already by testing if -1 < f < 1. We can thus safely subtract 127 from it
    // without having overflow to get the actual positive value of the exponent.
    exp -= 127u;

    static constexpr int signPos { 31 };

    // sign is true if the number is negative.
    void *signPtr { static_cast<void*>(&f) };
    bool sign { static_cast<bool>(*static_cast<unsigned int*>(sign) >> signPos) };

    // If the exponent is zero, the truncated number must be +/- 1, since there
    // can only be one digit before the point and it must be non-zero. In
    // binary, that means it must be a 1. The number is also not shifted left
    // or right by a non-zero exponent. Thus the final answer must be +/- 1.
    if (exp == 0)
        return sign ? -1 : 1;

    // Copy the argument to retain all info even after manipulating the number and
    // reinterpret the 32 bits in memory as an int to enable bitshifts.
    void *mantissaPtr { static_cast<void*>(&f) };
    unsigned int mantissa { *static_cast<unsigned int*>(mantissa) };

    // Extract the mantissa and flip the other bits to 0.
    mantissa &= 0x7fffffu;

    // Add the implied '1' before the decimal point back
    mantissa |= 0x800000u;

    // The number needs to be bitshifted exp places to the left and
    // mantissaLength places to the right to place the point at the proper pos.
    int shiftMagnitude { mantissaLength - static_cast<int>(exp) };

    // The sign of shiftMagnitude determines whether the direction of the shift
```

```
60      shiftMagnitude > 0 ? mantissa >>= shiftMagnitude : mantissa <<= -shiftMagnitude;
61      return sign ? -(static_cast<int>(mantissa)) : static_cast<int>(mantissa);
62  }
```

What is considered best programming style?

PS I may have made a mistake in the void pointer version as I changed it here in the comment and didn't compile and test.

nascardriver
January 19, 2019 at 9:03 am · Reply

Hi Louis!

First off, good job! Your code shows a firm understanding of casts, pointers, and bitwise operations.

> I may have made a mistake in the void pointer version
You used the "name" variables in places where the "namePtr" variables should've been used.

> What is considered best programming style?
Both, void* and @reinterpret_cast allow type-unsafe conversions. In that aspect, they're the same. Since you only need to new pointer to immediately read from it, you should use @reinterpret_cast. If you were to store the pointer, you should use void* (Or @std::uintptr_t, if you want it as a number).

Line 58 in the @reinterpret_cast version: Use the conditional operator only if you need the operations result. Otherwise, an if-statement is easier to read.

Bear in mind that the internal representation of floating pointer numbers is not standardized in C++. You can use @std::numeric_limits::is_iec559 to check if a number fulfills the requirements of the IEEE 754 standard.

https://en.cppreference.com/w/cpp/types/numeric_limits/is_iec559

Louis Cloete
January 19, 2019 at 11:48 am · Reply

Hi nascardriver! Thanks for the compliment. I have a few questions.

1. What is @std::uintptr_t? Is it a type that can be assigned from any type pointer, but interprets the number as an unsigned int when you dereference it? Can you do this with it: [EDIT]: You can't, but I can't figure out how you would use it. What is the use for it then? Does it store the address of a variable as an int? So I can't really use it in my trunc() function?

```
1   std::uintptr_t uintptr { &f };
2
3   unsigned int exp { ((uintptr >> mantissaLength) & 0xffu) - 127u };
4
5   // ...
6
7   bool sign { static_cast<bool>(uintptr >> signPos) };
8
9   // ...
10
11  unsigned int mantissa { (uintptr & 0x7fffffu) | 0x800000u };
```

I couldn't get a satisfactory answer on cppreference.com

PS I don't know why, but Code::Blocks doesn't autocomplete std::uintptr_t for you, but it does autocomplete uintptr_t. Similar story for all fixed width ints: it doesn't autocomplete the std:: version. Why is that? And why does the identifier without the std:: prefix compile at all? Why does it highlight the fixed width int types and uintptr_t like keywords, but not my own typedefs?

2. About the conditional operator: do you mean this instead?

```
    // The sign of shiftMagnitude determines the direction of the shift.
    if (shiftMagnitude > 0)
        mantissa >>= shiftMagnitude;
    else
        mantissa <<= -shiftMagnitude;
```

nascardriver
January 20, 2019 at 1:15 am · Reply

> What is @std::uintptr_t?
Update: Don't use @std::uintptr_t, it's an optional type (ie. compilers don't have to support it). Use @std::uint_fast32_t or @std::uint_fast64_t instead (Or their "least" counterparts).
It's an unsigned integer type guaranteed to be able to store any valid address. This is useful if you want to do arithmetic on the address instead of on a pointer.
Since you're not storing addresses in @truc, you don't need it. It's an alternative to a void*.

```
std::cout << std::hex;

int *p{ new int{ 3 } };

void *v{ static_cast<void *>(p) };
auto ui{ reinterpret_cast<std::uint_fast64_t>(p) };

std::cout << p << '\n'; // 2266E70
std::cout << (p + 1) << '\n'; // 2266E74
std::cout << (v + 1) << '\n'; // Illegal
std::cout << (ui + 1) << '\n'; // 2266E71
```

> Code::Blocks
I'm using (std::)int_fast32_t as a representative for all fixed width integer types.
@int_fast32_t is defined in @<stdint.h>
@std::int_fast32_t is defined int @<cstdint>
Whenever there's a <c*> and a <*.h> header, the <*.h> header is the C version, the <c*> header is the C++ version. C++ uses namespaces (In particular @std). C doesn't have namespaces, so it defines everything globally.
The C++ standard demands the C headers to be available in C++ (for backward compatibility).
I'm guessing a header you included (<iostream> tends to include a lot), included <stdint.h>.
This allows you to use int_fast32_t, but not @std::int_fast32_t.
If you can use @std::int_fast32_t but Code::Blocks doesn't suggest it, there's something wrong with Code::Blocks.
I'm experiencing the same syntax highlighting behavior in Visual Studio Code. My only guess is that this is done to differentiate between system- and user types.

> About the conditional operator: do you mean this instead?
Yes

Sivasankar
November 5, 2018 at 9:18 pm · Reply

You said that deleting a void pointer should be avoided. Do you mean, it(void pointer) should be converted to particular pointer type before deleting?

nascardriver

November 5, 2018 at 11:51 pm · Reply

It should be converted back to the type it was allocated as so you're deleting the right amount of memory.

Sivasankar
November 6, 2018 at 10:37 pm · Reply

Thanks for the clarification.

Blue
October 22, 2018 at 12:21 pm · Reply

Auto *ptr and Void *ptr are same?

Alex
October 22, 2018 at 1:49 pm · Reply

No. void *ptr is a void pointer. auto *ptr is a pointer to a type that is inferred based on context.

Baschti
September 6, 2018 at 3:18 pm · Reply

Why should you not dynamically allocated memory with a void pointer?

Alex
September 10, 2018 at 11:50 am · Reply

1) There's rarely a need to actually do so (unless maybe you're doing your own memory management)
2) Deleting memory pointed to by a void pointer can cause issues (namely, destructors won't be called)

« Older Comments 1 2