LEARN C++

Skill up with our free tutorials

10.20 — Void pointers

ALEX AUGUST 12, 2021

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:

```
#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 knows to 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;
}
```

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::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

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

Quiz answers

1. Show Solution

Next lesson
10.21 Pointers to pointers and dynamic multidimensional arrays

Back to table of contents

Previous lesson
10.19 For-each loops

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ co

Name*

Email*

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

Notify me about replies: 🔔 **POST COMMENT**

DP N N FOUT

Newest ▾

X

X