4.13 — Const, constexpr, and symbolic constants

BY ALEX ON FEBRUARY 23RD, 2015 | LAST MODIFIED BY NASCARDRIVER ON DECEMBER 29TH, 2019

Const variables

So far, all of the variables we've seen have been non-constant -- that is, their values can be changed at any time. For example:

```
int x { 4 }; // initialize x with the value of 4
x = 5; // change value of x to 5
```

However, it's sometimes useful to define variables with values that can not be changed. For example, consider the gravity of Earth (near the surface): 9.8 meters/second^2. This isn't likely to change any time soon (and if it does, you've likely got bigger problems than learning C++). Defining this value as a constant helps ensure that this value isn't accidentally changed.

To make a variable constant, simply put the const keyword either before or after the variable type, like so:

```
const double gravity { 9.8 }; // preferred use of const before type
int const sidesInSquare { 4 }; // okay, but not preferred
```

Although C++ will accept const either before or after the type, we recommend using *const* before the type because it better follows standard English language convention where modifiers come before the object being modified (e.g. a "green ball", not a "ball green").

Const variables *must* be initialized when you define them, and then that value can not be changed via assignment.

Declaring a variable as const prevents us from inadvertently changing its value:

```
const double gravity { 9.8 };
gravity = 9.9; // not allowed, this will cause a compile error
```

Defining a const variable without initializing it will also cause a compile error:

```
const double gravity; // compiler error, must be initialized upon definition
```

Note that const variables can be initialized from other variables (including non-const ones):

```
std::cout << "Enter your age: ";
int age;
std::cin >> age;

const int usersAge { age }; // usersAge can not be changed
```

Const is often used with function parameters:

```
void printInteger(const int myValue)
{
    std::cout << myValue;
}
```

Making a function parameter const does two things. First, it tells the person calling the function that the function will not change the value of myValue. Second, it ensures that the function doesn't change the value of myValue.

When arguments are passed by value, we generally don't care if the function changes the value of the parameter (since it's just a copy that will be destroyed at the end of the function anyway). For this reason, we usually don't const parameters passed by value. But later on, we'll talk about other kinds of function parameters (where

changing the value of the parameter will change the value of the argument passed in). For these types of parameters, judicious use of const is important.

Runtime vs compile time constants

C++ actually has two different kinds of constants.

Runtime constants are those whose initialization values can only be resolved at runtime (when your program is running). Variables such as *usersAge* and *myValue* in the snippets above are runtime constants, because the compiler can't determine their initial values at compile time. *usersAge* relies on user input (which can only be given at runtime) and *myValue* depends on the value passed into the function (which is only known at runtime). However, once initialized, the value of these constants can't be changed.

Compile-time constants are those whose initialization values can be resolved at compile-time (when your program is compiling). Variable *gravity* above is an example of a compile-time constant. Compile-time constants enable the compiler to perform optimizations that aren't available with runtime constants. For example, whenever *gravity* is used, the compiler can simply substitute the identifier *gravity* with the literal double *9.8*.

When you declare a const variable, the compiler will implicitly keep track of whether it's a runtime or compile-time constant.

In most cases, this doesn't matter, but there are a few odd cases where C++ requires a compile-time constant instead of a run-time constant (such as when defining the length of a fixed-size array -- we'll cover this later), or to declare a `std::bitset`.

```
#include <iostream>
#include <bitset>
#include <cstddef>

std::size_t getNumberOfBits()
{
    return 3;
}

int main()
{
    const std::size_t numberOfBits{ 3 }; // Compile-time constant

    std::bitset<numberOfBits> b{};

    const std::size_t otherNumberOfBits{ getNumberOfBits() }; // Run-time constant

    std::bitset<otherNumberOfBits> b2{}; // Error

    return 0;
}
```

`otherNumberOfBits` is a run-time constant, because `getNumberOfBits` might do something that requires the program to be running, eg. using `std::cin`.

constexpr

To help provide more specificity, C++11 introduced new keyword **constexpr**, which ensures that a constant must be a compile-time constant:

```
constexpr double gravity { 9.8 }; // ok, the value of 9.8 can be resolved at compile-time
constexpr int sum { 4 + 5 }; // ok, the value of 4 + 5 can be resolved at compile-time

std::cout << "Enter your age: ";
```

```
int age;
std::cin >> age;
constexpr int myAge { age }; // not okay, age can not be resolved at compile-time
```

`constexpr` variables are `const`. This will get important when we talk about other effects of `const` in upcoming lessons.

Best practice

Any variable that should not be modifiable after initialization and whose initializer is known at compile-time should be declared as constexpr.
Any variable that should not be modifiable after initialization and whose initializer is not known at compile-time should be declared as const.

Naming your const variables

Some programmers prefer to use all upper-case names for const variables. Others use normal variable names with a 'k' prefix. However, we will use normal variable naming conventions, which is more common. Const variables act exactly like normal variables in every case except that they can not be assigned to, so there's no particular reason they need to be denoted as special.

Symbolic constants

In the previous lesson **4.12 -- Literals**, we discussed "magic numbers", which are literals used in a program to represent a constant value. Since magic numbers are bad, what should you do instead? The answer is: use symbolic constants! A **symbolic constant** is a name given to a constant literal value. There are two ways to declare symbolic constants in C++. One of them is good, and one of them is not. We'll show you both.

Bad: Using object-like macros with a substitution parameter as symbolic constants

We're going to show you the less desirable way to define a symbolic constant first. This method was commonly used in a lot of older code, so you may still see it.

In lesson **2.10 -- Introduction to the preprocessor**, you learned that object-like macros have two forms -- one that doesn't take a substitution parameter (generally used for conditional compilation), and one that does have a substitution parameter. We'll talk about the case with the substitution parameter here. That takes the form:

```
#define identifier substitution_text
```

Whenever the preprocessor encounters this directive, any further occurrence of *identifier* is replaced by *substitution_text*. The identifier is traditionally typed in all capital letters, using underscores to represent spaces.

Consider the following snippet:

```
#define MAX_STUDENTS_PER_CLASS 30
int max_students { numClassrooms * MAX_STUDENTS_PER_CLASS };
```

When you compile your code, the preprocessor replaces all instances of MAX_STUDENTS_PER_CLASS with the literal value 30, which is then compiled into your executable.

You'll likely agree that this is much more intuitive than using a magic number for a couple of reasons. MAX_STUDENTS_PER_CLASS provides context for what the program is trying to do, even without a comment.

Second, if the number of max students per classroom changes, we only need to change the value of MAX_STUDENTS_PER_CLASS in one place, and all instances of MAX_STUDENTS_PER_CLASS will be replaced by the new literal value at the next compilation.

Consider our second example, using #define symbolic constants:

```
#define MAX_STUDENTS_PER_CLASS 30
#define MAX_NAME_LENGTH 30

int max_students { numClassrooms * MAX_STUDENTS_PER_CLASS };
setMax(MAX_NAME_LENGTH);
```

In this case, it's clear that MAX_STUDENTS_PER_CLASS and MAX_NAME_LENGTH are intended to be independent values, even though they happen to share the same value (30). That way, if we need to update our classroom size, we won't accidentally change the name length too.

So why not use #define to make symbolic constants? There are (at least) three major problems.

First, because macros are resolved by the preprocessor, which replaces the symbolic name with the defined value, #defined symbolic constants do not show up in the debugger (which shows you your actual code). So although the compiler would compile `int max_students { numClassrooms * 30 };`, in the debugger you'd see `int max_students { numClassrooms * MAX_STUDENTS_PER_CLASS };`, and MAX_STUDENTS_PER_CLASS would not be watchable. You'd have to go find the definition of MAX_STUDENTS_PER_CLASS in order to know what the actual value was. This can make your programs harder to debug.

Second, macros can conflict with normal code. For example:

```
#include "someheader.h"
#include <iostream>

int main()
{
    int beta { 5 };
    std::cout << beta;

    return 0;
}
```

If someheader.h happened to #define a macro named beta, this simple program would break, as the preprocessor would replace the int variable beta's name with whatever the macro's value was.

Thirdly, macros don't follow normal scoping rules, which means in rare cases a macro defined in one part of a program can conflict with code written in another part of the program that it wasn't supposed to interact with.

Warning

Avoid using #define to create symbolic constants macros.

A better solution: Use constexpr variables

A better way to create symbolic constants is through use of constexpr variables:

```
constexpr int maxStudentsPerClass { 30 };
constexpr int maxNameLength { 30 };
```

Because these are just normal variables, they are watchable in the debugger, have normal scoping, and avoid other weird behaviors.

Best practice

Use constexpr variables to provide a name and context for your magic numbers.

Using symbolic constants throughout a multi-file program

In many applications, a given symbolic constant needs to be used throughout your code (not just in one location). These can include physics or mathematical constants that don't change (e.g. pi or avogadro's number), or application-specific "tuning" values (e.g. friction or gravity coefficients). Instead of redefining these every time they are needed, it's better to declare them once in a central location and use them wherever needed. That way, if you ever need to change them, you only need to change them in one place.

There are multiple ways to facilitate this within C++, but the following is probably easiest:

1) Create a header file to hold these constants
2) Inside this header file, declare a namespace (we'll talk more about this in lesson **6.2 -- User-defined namespaces**)
3) Add all your constants inside the namespace (make sure they're *constexpr* in C++11/14, or *inline constexpr* in C++17 or newer)
4) #include the header file wherever you need it

For example:

constants.h (C++11/14):

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

// define your own namespace to hold constants
namespace constants
{
    constexpr double pi { 3.14159 };
    constexpr double avogadro { 6.0221413e23 };
    constexpr double my_gravity { 9.2 }; // m/s^2 -- gravity is light on this planet
    // ... other related constants
}
#endif
```

In C++17, prefer "inline constexpr" instead:

constants.h (C++17 or newer):

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

// define your own namespace to hold constants
namespace constants
{
    inline constexpr double pi { 3.14159 }; // inline constexpr is C++17 or newer only
    inline constexpr double avogadro { 6.0221413e23 };
    inline constexpr double my_gravity { 9.2 }; // m/s^2 -- gravity is light on this planet
    // ... other related constants
}
#endif
```

Use the scope resolution operator (::) to access your constants in .cpp files:

main.cpp:

```
#include "constants.h"
#include <iostream>

int main
{
    std::cout << "Enter a radius: ";
    int radius{};
    std::cin >> radius;

    double circumference { 2 * radius * constants::pi };
    std::cout << "The circumference is: " << circumference;

    return 0;
}
```

If you have both physics constants and per-application tuning values, you may opt to use two sets of files -- one for the physics values that will never change, and one for your per-program tuning values that are specific to your program. That way you can reuse the physics values in any program.

4.x -- Chapter 4 summary and quiz

Index

4.12 -- Literals

C++ TUTORIAL | PRINT THIS POST

206 comments to 4.13 — Const, constexpr, and symbolic constants

« Older Comments 1 2 3

HZ

December 28, 2019 at 10:05 am · Reply

A hopefully useful detail: review "For example, whenever gravity is used, the compiler can simply substitute the identifier gravity for the literal double 9.8.", where I suppose you mean "substitute with". It appears to me as if it now reads exactly the reverse of what you intend to explain.
Cheers!

nascardriver
December 29, 2019 at 2:44 am · Reply

Yep,
substitute old with new
substitute new for old
Lesson amended, thanks!

HZ
December 29, 2019 at 1:36 pm · Reply

:-) I am quite fond of English, however, it isn't my native language, so I had to verify my hunch before writing the comment. Your old/new new/old "template" (to stay in tune with our subject matter here) is quite a nice one to check which way around gives the correct sentence!
HZ

HolzstockG
December 26, 2019 at 1:05 pm · Reply

Why do we put constants inside of header file and not in source code file which we could add? Is there even a possiblity to create forward declaration for variables?

Raton
January 31, 2020 at 11:35 pm · Reply

I think it's recommended not to #include .cpp files inside .cpp files. A .cpp file is supposed to contain normal statements (not just declarations), which you probably don't want to #include in other .cpp files. And a header file is supposed, I think, to contain declarations, which is exactly what is needed here.

And I think you can forward declare variables using the extern keyword but I'm not sure.

nascardriver
February 1, 2020 at 3:07 am · Reply

Everything correct.

HolzstockG
December 26, 2019 at 8:50 am · Reply

You can add here as an reminder from previous lesson that one of the cases of having to use constant during compile-time is defining std::bitset<> next to the arrays.

nascardriver
December 28, 2019 at 4:38 am · Reply

I added an example with `std::bitset` in the run-time/compile-time section. Thank you for the suggestion!

`constexpr` variables cannot be forward declared, they have to be initialized at their declaration. The `inline` prevents the variables from being defined multiple times.
Other variables can be forward declared, this is covered later.

HolzstockG
December 28, 2019 at 10:05 pm · Reply

Thank you

Eshita Rastogi
November 24, 2019 at 6:31 pm · Reply

Hello
while compiling main.cpp and constants.h, I am getting an error in the constants in the following line

double circumference { 2 * radius * constants::pi };

The error is as follows
name followed by '::' must be a class or namespace name

Can anyone explain what is the meaning of this error?

Alex
November 25, 2019 at 10:20 pm · Reply

It's not recognizing "constants" as a namespace. That either means you have a typo somewhere (e.g. in constants.h you named your namespace "constant" or "costnants" or something, or you forgot to #include "constants.h" from main.cpp.

BooGDaaN
November 16, 2019 at 5:33 am · Reply

At the end of this setion, you need to include also:

```
1 #include <iostream>
```

Great practice and very useful!

kiwi
October 8, 2019 at 5:18 am · Reply

Under the "Runtime vs compile time constants" header, the third sentence states: "Variables such as usersAge and myValue in the snippets above above are runtime constants..."

I think you meant to type shown above instead of above above. Just wanted to let you know so someone else doesn't get confused.

Thanks for the comprehensive tutorials!

Alex
October 11, 2019 at 12:55 pm · Reply

Fixed. Thanks!

Mike
October 2, 2019 at 10:08 am · Reply

From Ch 4.13
"When you declare a const variable, the compiler will implicitly keep track of whether it's a runtime or compile-time constant.

In most cases, this doesn't matter, but there are a few odd cases where C++ requires a compile-time constant instead of a run-time constant..."

When I tested this using a run-time constant in Visual Studio, I could never get it to compile. The operator>> before the variable name would always show a red squiggly line. The compiler would return many errors starting with error C2593: 'operator >>' is ambiguous

Now I understand, const means it can't be changed, but why doesn't the compiler just state that to the user. Instead, it's making the previous operator/command in the statement look like it's the culprit, thus resulting in like 10 or 15 errors.

The lesson touched on run-time constants, but I don't believe it elaborated on how to actually do/code it to work.

I know the code below is not recommended, but shouldn't it still work, as Alex stated above regarding run-time constants: "In most cases, this doesn't matter"?

```
1 const int range{};
2 std::cout << "Enter your max range: ";
3 std::cin >> range;  // I get the red squiggly on >>, though if I remove const from @range, n
```

nascardriver
October 3, 2019 at 12:11 am · Reply

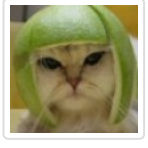
> I know the code below is not recommended, but shouldn't it still work
No, it shouldn't work. It doesn't matter is `range` is a compile-time constant. It's `const` and can't be changed, so `std::cin >>` can't assign to it.

The error is shown on `>>` because `>>` is a function (You'll learn that later). There are multiple versions of this function (You'll learn that later), but none of them takes a `const int` as an argument. So you're trying to use a function that doesn't exist, hence the error on the `>>`.

Jon
September 26, 2019 at 1:46 am · Reply

That last main.cpp could use

```
1 #include <iostream>
```



Alex
September 26, 2019 at 3:13 pm · Reply

Added. Thanks!

Marie Bethell
September 17, 2019 at 4:21 pm · Reply

Hi!
What does inline do?
Thanks.

nascardriver
September 18, 2019 at 11:57 pm · Reply

It means that the variable can be defined multiple times (in different files), ie. you don't need a separate source file for the definition. This will make more sense after you read Chapter S.

nascardriver
September 19, 2019 at 12:30 am · Reply

Sorry that was wrong, you don't need `inline` for that.
If you don't declare the variable `inline`, the variable will be created in every file that includes the header. It will have the same value, so it appears to be the same variable, but in reality it's not the same. Without `inline`, you're wasting memory.
I don't know why not all `constexpr` variables are `inline` by default, maybe Alex can help out.

Alex
September 19, 2019 at 4:00 pm · Reply

> I don't know why not all `constexpr` variables are `inline` by default, maybe Alex can help out.

Good answers here: https://stackoverflow.com/questions/51051787/why-are-non-member-static-constexpr-variables-not-implicitly-inline

nascardriver
September 19, 2019 at 10:31 pm · Reply

Thanks!

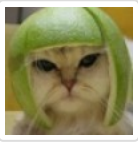
Marie Bethell
September 20, 2019 at 11:16 am · Reply

Thanks, that was helpful.

Georgii
June 29, 2019 at 7:01 am · Reply

Warning. This comment has nothing to do with programming.
In the second paragraph you say: "For example, consider the value of gravity on Earth: 9.8 meters/second^2."
Typical programmer language :p)))
The value of gravity... Well I'd say it's pretty darn precious, because I really like being able to lie down and relax)

A better option would have been "Gravitational acceleration on Earth" or "Freefall acceleration on Earth".
Awesome tutorials btw.
Thanks



Alex
June 30, 2019 at 4:39 pm · Reply

Heh. I changed it to "Gravity of Earth" rather than "Gravity on Earth", which Wikipedia defines as, "the net acceleration that is imparted to objects due to the combined effect of gravitation (from distribution of mass within Earth) and the centrifugal force (from the Earth's rotation)".

Nguyen
June 25, 2019 at 7:06 pm · Reply

Hi,

Sometime I see const placed before the function's return type and/or parameter's type as shown below:

Example 1

```
const int printInteger(int myValue)
{
    myValue = 123;
    return myValue;
}
```

Example 2

```
const int printInteger(const int myValue)
{
    myValue = 123;
    return myValue;
}
```

Could you please explain const in these examples?

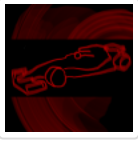
Thanks

nascardriver
June 26, 2019 at 8:17 am · Reply

Those don't make a lot of sense yet. `const` is mostly used in combination with pointers or references, which are covered in chapter 6.

M.J Aslam
June 17, 2019 at 9:54 pm · Reply

I believe when we use const with pointers then the meaning of using them before or after the declaration type does change. May be you can explain that as well.



nascardriver
June 19, 2019 at 3:04 am · Reply

It's covered in chapter 6. There's no point in doing it here, because pointers haven't been covered yet.

BP
June 5, 2019 at 4:17 am · Reply

Hey!
A quick question, is there a good reason for:

```
1 double circumference = 2 * radius * constants::pi;
```

This is done in the last example and I don't really know why you wouldn't use uniform initialization.
Is there a reason?
Thanks!

Alex
June 12, 2019 at 11:48 am · Reply

I just missed updating that one from copy initialization to uniform initialization when I updated the lesson. It's updated now. Thanks for pointing that out.

BP
June 12, 2019 at 11:57 am · Reply

Thanks for replying

Anders
May 6, 2019 at 1:20 am · Reply

Could you write something about the use of

```
1 constexpr const
```

together.
As a reader you almost get the idea that you would use either constexpr OR const.

Alex
May 6, 2019 at 3:30 pm · Reply

For objects, "constexpr const" is redundant when both keywords refer to the same object. In such a case, you can use constexpr.

Sammy
May 1, 2019 at 6:50 pm · Reply

I did some research and found out:
const can use on both runtime and compile-time, while constexpr can use on compile-time only.

Why don't we just use const, so that we don't have to think which time are we on?
Could you explain more why constexpr is the best practice?
The above lesson is not clear enough for me. Thanks.

nascardriver
May 2, 2019 at 4:09 am · Reply

If you can do computations at compile-time, you don't have to do them every time the program runs. const will calculate on every run, constexpr will calculate once during compilation.
There are also constexpr and consteval functions, which can do even more computations at compile-time and they won't work with const.

Ganesh Koli
September 25, 2019 at 10:34 am · Reply

Hello nascardriver,
Before constexpr as a part of C++11, Do compiler optimize the code to do computation at compile time , which later added in C++ standard.

I believe most of the compiler do optimization with respecting the C++ standard(i mean without violating C++ standard) like copy ellision, run time optimization.

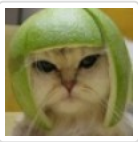
Thanks to clear my view. and provide more details.

nascardriver
September 26, 2019 at 3:48 am · Reply

Compilers did and still do computations at compile time if they think it improves the program. `constexpr` is only a hint for compilers so it's easier for them to find functions which can be optimized. They don't have to run `constexpr` functions at compile-time.

Conor
April 26, 2019 at 4:42 am · Reply

Can you use constexpr in C?



Alex
April 28, 2019 at 4:28 pm · Reply

No. Constexpr is a C++11 concept. Maybe someday they'll back-port it to C.

Gejsi
March 1, 2019 at 9:18 pm · Reply

Can I name the namespace header file the exact same word as the namespace identifier, or is it a bad practice and gives a compile error?

IE:

```
#ifndef CONSTANTS
#define CONSTANTS

namespace constants
{
constexpr int x(2);
constexpr int y(5);
constexpr int z(x * y);
}
#endif
```

nascardriver
March 3, 2019 at 8:55 am · Reply

* Line 6, 7, 8: Initialize your variables with brace initializers. You used direct initialization.

If the name that describes the contents of the file and the one that describes the contents of the namespace are the same, then you should name the file and namespace the same or similar. There won't be any errors.

Jeroen P. Broks
February 17, 2019 at 4:00 am · Reply

In the "#define myconstant 2" vs "const int myconstatant = 2" discussion, I do wonder about one thing. I don't expect to often have to deal with it, but I recently wrote a small tool in C and I had to change my entire code when I wanted to make it MS-DOS compatible due to things not being fully supported. If I would ever code a tool which I intend to make runnable in DOS (either on real MS-DOS or DOSBox), should that 'force' me to use the "#define" method or is const also present in C++ compilers that age?

nascardriver
February 17, 2019 at 7:43 am · Reply

MS-DOS is an OS, C++ is a language, they're unrelated to each other.
The closest guess I have is that you're running a compiler in MS-DOS that doesn't know about most of C++'s revisions (It probably only knows C++98). If that's the case, you can use the clang++ status site the see the changes in each version of C++ (https://clang.llvm.org/cxx_status.html).
const has always been there.
If you're talking about something else, please let me know.

Okorie Emmanuella
February 1, 2019 at 5:55 pm · Reply

Hello Nascardriver!

```
1  double circumference = 2 * radius * constants::pi;
```

is 2 not a magic number in the above snippet?

nascardriver
February 2, 2019 at 4:29 am · Reply

Hi!

No. The formula for the circumference is well known (At least in combination with the variable name "circumference"). Also, there's no good name you could give to 2 in this case.
Magic numbers occur when you don't name numbers that you made up yourself, like the height of a tower, a maximum score, etc..

Okorie Emmanuella
February 4, 2019 at 5:44 am · Reply

Oh. i understand it better now. thank you!

Louis Cloete
January 29, 2019 at 12:49 pm · Reply

@Alex, if I find that I repeat a block of code similar to this:

```
SomeStruct *s;

// From here to the end is repeated identically four times.
s = doSomething(x,y); // dynamically allocate memory.
if (s)
{
    if (s->a == Enum::SOMETHING)
    {
        delete s;
        return true;
    }
delete s;
}
```

Would it make sense to define it as an object-like macro, then #undef'ing the macro after the fourth time? Just to be clear, it can not be inside a loop, because I change variables x and y between each time this sequence is executed.

nascardriver
January 30, 2019 at 6:45 am · Reply

Don't use macros unless you have to.
You give little context about your code, here's my suggestion.

```
enum class Enum
{
  NOTHING,
  SOMETHING,
};

struct SomeStruct
{
  Enum a{ Enum::NOTHING };
};

SomeStruct *doSomething(int x, int y)
{
  return new SomeStruct{};
}

bool wrap(int x, int y)
{
  if (SomeStruct * s{ doSomething(x, y) })
  {
    if (s->a == Enum::SOMETHING)
    {
      delete s;
      return true;
    }

    delete s;
  }

  return false;
}

```

```
bool fn(void)
{
  constexpr int k_iCalls{ 4 };

  constexpr int x[k_iCalls]{ 8, 12, 3, 99 };
  constexpr int y[k_iCalls]{ 1, 1, 9, 0 };

  for (int i{ 0 }; i < k_iCalls; ++i)
  {
    if (wrap(x[i], y[i]))
    {
      return true;
    }
  }

  // ...

  return false;
}

int main(void)
{
  fn();

  return 0;
}
```

Louis Cloete
January 30, 2019 at 7:56 am · Reply

The context is that I am writing a PvP chess game. The function takes a board coordinate as input and figures out if a Knight threatens a given square. Here it is:

```
enum class Type
{
    ROOK,
    KNIGHT,
    BISHOP,
    QUEEN,
    KING,
    PAWN,
    UNKNOWN
};

struct Square;

struct Piece
{
    Square *pos;
    Type type;
    bool isWhite;
};

struct Square
{
    Piece *piece;
    char background;
};

using uint_t = unsigned int;

```

```
// A chess board is 8x8.
constexpr uint_t g_sideLength { 8 };

using boardArray_t = std::array<std::array<Square, g_sideLength>, g_sideLength>;

struct Coordinate
{
    uint_t file;
    uint_t rank;
};

Coordinate * findPiece(const Coordinate &direction, const Coordinate &startPos,
                  const boardArray_t &board, int distance=0)
{
    int counter { 1 };
    Coordinate *searchPos { new Coordinate };
    searchPos ->rank = startPos.rank + direction.rank;
    searchPos ->file = startPos.file + direction.file;

    // Ensure indexes stay within bounds.
    while (searchPos->rank < g_sideLength && searchPos->file < g_sideLength)
    {
        // If the square contains a piece, return its coordinate.
        if (bord.at(searchPos->rank).at(searchPos->file).piece)
            return searchPos;

        // Search for a limited amount of iterations if specified by distance pa
        if (distance != 0 && counter == distance)
                break;

        ++counter;
    }

    delete searchPos;
    return nullptr;
}

bool isKnightsThreat(bool isWhite, const Coordinate &coord, const boardArray_t &
{
    int rank;
    Coordinate *pos;
    Piece *piece;

    for (int file { -2 }; file <= 2; ++file)
    {
        if (file < static_cast<int>(g_sideLength))
        {
            if (file == 0)
                continue;

            // If file == +/- 2
            if (file == -2 || file == 2)
            {
#define PIECE_HOSTILE_KNIGHT pos = findPiece({ static_cast<uint_t>(rank),\
                    static_cast<uint_t>(file) }, coord, board, 1);\
                if (pos)\
                {\
                    piece = board.at(pos->rank).at(pos->file).piece;\
                    if (piece->isWhite == !isWhite && piece->type == Type::KNIGH
                    {\
                        delete pos;\
                        return true;\
```

```
                    }\
                }

                rank = -1;
                PIECE_HOSTILE_KNIGHT

                rank = 1;
                PIECE_HOSTILE_KNIGHT
            }
            else
            {
                rank = -2;
                PIECE_HOSTILE_KNIGHT

                rank = 2;
                PIECE_HOSTILE_KNIGHT
#undef PIECE_HOSTILE_KNIGHT
            }
        }
    }
    return false;
}
```

It is a bit ugly. It also has a lot of unsigned ints, because of std::array's sizeType. The gist is that if the change in one direction is +/- 2, the change in the other direction must be +/- 1, and vice versa. You gave me an idea, anyway. I can post when I implemented it if you want to.

nascardriver
January 30, 2019 at 8:24 am · Reply

The size type of @std::array doesn't have to be an unsigned int. Use boardArray_t::size_type instead.
@searchPos in @findPiece never changes. There was also a spelling error in a variable name that prevented compilation. Make sure to test your program often during development to prevent getting overrun by problems in the end.
My untested solution (Write a solution yourself first, post that)

SPOILER SPOILER SPOILER

```
class Coordinate
{
public:
  uint_t file;
  uint_t rank;

public:
  bool valid(void) const
  {
    return ((file >= 0) && (file < g_sideLength) &&
            (rank >= 0) && (rank < g_sideLength));
  }
};

bool isKnightsThreat(bool isWhite, const Coordinate &coord, const boardArray_
{
  // @coord could get attacked from 8 positions.
  // Note: Subtraction will cause a roll-around to UINT_MAX. Use signed ints.
  // The roll around doesn't matter, because @isValid will detect in, but it'
  // not a good thing to happen.
  // You could probably get rid of this array by finding a pattern in the
```

```
    // offsets.
    Coordinate vCoordAttackerLocations[]{
      { coord.file + 2, coord.rank + 1 },
      { coord.file + 2, coord.rank - 1 },
      { coord.file - 2, coord.rank + 1 },
      { coord.file - 2, coord.rank - 1 },
      { coord.file + 1, coord.rank + 2 },
      { coord.file + 1, coord.rank - 2 },
      { coord.file - 1, coord.rank + 2 },
      { coord.file - 1, coord.rank - 2 },
    };

    for (const auto &coordAttacker : vCoordAttackerLocations)
    {
      if (coordAttacker.valid())
      {
        // Note: @findPiece is leaking @searchPos if the while-loop terminates.

        const auto &square{ board.at(coordAttacker.rank).at(coordAttacker.file)

        if (square.piece->isWhite != isWhite)
        {
          if (square.piece->type == Type::KNIGHT)
          {
            // The knight can attack @coordAttacker
            return true;
          }
        }
      }
    }

    return false;
  }
```

Louis Cloete
January 30, 2019 at 2:39 pm · Reply

I decided against using boardArray_t::sizeType, because I have an array of Pieces too, which is technically another type, and I want to use the same loop counter to access both sometimes. std::array::sizeType is an alias for unsigned int, so I use it. I would prefer to use ints, but the compiler doesn't like it. Maybe I should just remove the -Werror flag and use ints.

Thanks for pointing out that searchPos never changes. I missed that.

The spelling error was because I wrote the code with Afrikaans variable names. I copied it into the comment editing box and did a manual search and replace with English names. @board was originally @bord. I did compile just prior to asking my original question and didn't change anything since then until I copied the code into the comment editing box.

My code:
Note: I didn't use classes, since I started this project before I knew how to use them. I decided to finish it without using my own classes, then refactor it into classes as applicable. I also didn't translate again. You should be able to follow the gist of the code after the previous translated version.

```
Koordinaat* soekStuk(const Koordinaat &rigting, const Koordinaat &beginpc
                     const bordArray_t &bord, int afstand=0)
{
    int teller { 1 };
```

```
    Koordinaat *soekPos { new Koordinaat };
    soekPos->gelid = beginpos.gelid + rigting.gelid;
    soekPos->ry = beginpos.ry + rigting.ry;

    // Sorg dat die indekse nie buite die omvang van die array gaan nie.
    while (soekPos->gelid < g_sylengte && soekPos->ry < g_sylengte)
    {
        // As daar 'n stuk is op die posisie, return die blokkie.
        if (bord.at(soekPos->gelid).at(soekPos->ry).stuk)
            return soekPos;

        // Soek slegs vir 'n beperkte aantal kere as so gespesifiseer is.
        if (afstand != 0 && teller == afstand)
                break;

        // Skuif die soek
        if (rigting.gelid)
            soekPos->gelid += rigting.gelid;
        if (rigting.ry)
            soekPos->ry += rigting.ry;

        ++teller;
    }

    delete soekPos;
    return nullptr;
}

bool isRuitersBedreiging(bool isWit, const Koordinaat &koord, const bordA
{
    Koordinaat *pos;
    Stuk *stuk;

    for (int ryVerpl { -2 }; ryVerpl <= 2; ++ryVerpl)
    {
        if (ryVerpl == 0)
            continue;

        int vermenigvuldiger { ryVerpl % 2 ? 1 : 2 };

        if (koord.ry + static_cast<uint_t>(ryVerpl) < g_sylengte)
        {
            for (int gelidVerpl { -1 }; gelidVerpl <= 1; gelidVerpl += 2)
            {
                pos = soekStuk({ static_cast<uint_t>(gelidVerpl * vermeni
                    static_cast<uint_t>(ryVerpl) }, koord, bord, 1);
                if (pos)
                {
                    stuk = bord.at(pos->gelid).at(pos->ry).stuk;
                    if (stuk->isWit == !isWit && stuk->soort == Soort::RU
                    {
                        delete pos;
                        return true;
                    }

                    delete pos;
                }
            }
        }
    }
    return false;
}
```

nascardriver
January 31, 2019 at 6:09 am · Reply

* Line 5, 35, 36: Initialize your variables with uniform initialization.
* Line 33, 49: Limit your lines to 80 characters in length for better readability on small displays.

> Maybe I should just remove the -Werror flag and use ints.
No. Error and warnings mean that you're doing something wrong. Disabling them will lead to trouble sooner or later.

> I wrote the code with Afrikaans
Program in Dutch and dutch people will help you. Program in English and the world will help you.

* @soekStuk would be more efficient to use if it returned the Square it found, as any caller will most likely want to access the piece.
* @pos should be declared in line 49
* @stuk should be declared in line 53
* @soekStuk can return by value. This will make your code easier and less prone to memory leaks. This won't slow down your code. Compilers have to elide the copy.

Louis Cloete
February 4, 2019 at 9:54 am · Reply

Hi, @nascardriver!

I have been afc for a while. I think you misunderstood my intention with the -Werror flag. Of course you shouldn't disable warnings. It is just that the choice to use unsigned ints for std::array's size type generates a lot of sign conversion warnings. Some of them are legit, and should be dealt with, while some of them are no problem. I can see that my int loop counter will never go negative. I am just counting from 0 up to 7 in a lot of cases. To fix that, I have to either static_cast to unsigned int (ugly) or use unsigned ints for loop counters (not always possible in my program for other reasons). Disabling -Werror will still give the warnings, just not stop compilation on them. This will allow me to "fix" the legit ones without bloating my code fixing bogus errors.

About coding in English, I didn't think I would share this code with anyone except for personal programmer friends speaking Afrikaans. I always code in English if I intend to share / ask a wider audience.

About the last asterisks: I return a pointer because you can do this with a pointer:

```
1 if (ptr)
2 {
3     // Do something dereferencing the pointer
4 }
```

It is also a very handy way to encode the information if an actual piece was found in the search to return nullptr for a blank search. What should I do to make it clear to the caller that no legal Piece was found in the current search if I return by value?

I agree that it is better to return the Square, so I made the @soekStuk return a Square*

nascardriver
February 5, 2019 at 6:45 am · Reply

> Werror
If you remove -Werror and ignore the warnings about signed/unsigned mismatches you'll miss other warnings that might not be negligible, because you get used to seeing warnings. You might not like @static_cast now, but after using it for a while you'll no longer care about it.

> @soekStuk
3 options I can think of right now:

```
#include <optional>
#include <tuple>

class CSquare
{
public:
  int iData{ 0 };
};

CSquare findSquare(void)
{
  return { 123 };
}

std::optional<CSquare> findSquareOptional(void)
{
  // Not found
  return {};

  // Found. Note: We have to explicitly call @CSquare::CSqua
  return CSquare{ 123 };
}

bool findSquareReference(CSquare &s)
{
  // Not found
  return false;

  // Found
  s = { 123 };
  return true;
}

std::tuple<bool, CSquare> findSquareTuple(void)
{
  // Not found
  return { false, {} };

  // Found
  return { true, { 123 } };
}

int main(void)
{
  // Can't test if a square was found.
  CSquare s1{ findSquare() };

  if (auto s2{ findSquareOptional() })
```

```
49      {
50        // A square was found.
51        // Access the square as if it was a pointer
52        s2->iData;
53      }
54
55      if (CSquare s3{}; findSquareReference(s3))
56      {
57        // A square was found
58        s3.iData;
59      }
60
61      if (auto s4{ findSquareTuple() }; std::get<bool>(s4))
62      {
63        // A square was found
64        std::get<CSquare>(s4).iData;
65      }
66
67      return 0;
68  }
```

Jules
January 25, 2019 at 7:25 am · Reply

#include "constants.h"
double circumference = 2 * radius * constants::pi;

Instead can I use :
double circumference = 2 * radius * constants.pi;

what is the use of the "." operator? is it discussed later?

And whenever "std::cout" or "std::cin" is used, are cin and cout functions being accessed from the standard library using the scope resolution operator? I am confused because cin and cout dont seem to be constants

nascardriver
January 25, 2019 at 7:36 am · Reply

:: is used to access members without objects
. is used to access members of an object

This will make more sense later.

> And whenever "std::cout" or "std::cin" is used, are cin and cout functions being accessed from the standard library?
Yes

Jules
January 25, 2019 at 7:45 am · Reply

Thank you!
So "members" include constants and functions

nascardriver
January 25, 2019 at 7:49 am · Reply

Yep

HUE Saki
January 24, 2019 at 10:59 pm · Reply

PLEASE HELP ME!

error: invalid operands of types 'int' and 'double()' to binary 'operator*'

why above error occur when I try to compile below program in my CODE::BLOCK

CONSTANT.H

```
ifndef CONSTANTS_H
#define CONSTANTS_H

// define your own namespace to hold constants
namespace constants
{
   constexpr double pi(3.14159);
   constexpr double avogadro(6.0221413e23);
   constexpr double my_gravity(9.2); // m/s^2 -- gravity is light on this planet
   // ... other related constants
}
#endif
```

MAIN.cpp

```
#include <iostream>
#include "CONSTANT.H"

int main()
{

double circumference = 2.0 * 7.0 * constants::pi;
return 0;}
```

hailinh0708
January 26, 2019 at 11:20 pm · Reply

1. You forget the "#" sign before ifndef.
Aside from that, it compiles fine for me.
Note: The compiler error(s) that appears do not relate to the error you encountered.

« Older Comments 1 2 3