# Const Qualifier

## Overview

Sometimes we want to define a variable whose value we know cannot be changed. For example, we might want to use a variable to refer to the size of a buffer size. Using a variable makes it easy for us to change the size of the buffer if we decided the original size wasn't what we needed. On the other hand, we'd also like to prevent code from inadvertently giving a new value to the variable we use to represent the buffer size. We can make a variable unchangeable by defining the variable's type as `const`.

```
const double pi = 3.14159;     // the value of pi is unchangable
const double e;                              // error: e is uninitialized const

// this whole block is invalid
cout << "enter number: ";
cin >> pi;
cout << pi;
// error!
```

## Initialization and const

When we use an object to initialize another object, it doesn't matter either or both of the object are `const`

```
int i = 42;
const int ci = i;      // value of i is now in ci
int j = ci;                    // value of ci is now in j
```

---

# References to const

We can bind a reference to an object of a `const` type. Unlike an ordinary reference, a reference to `const` cannot be used to change the object to which the reference is bound.

```
const int ci = 1024;
const int &r1 = ci;

// error
r1 = 42;
int &r2 = ci;
```

Because we cannot assign directly to `ci`, we also should not be able to use a reference to change `ci`. Therefore, the initialization of `r2` is an error. If this initialization were legal, we could use `r2` to change the value of its underlying object.

### A Reference to const May Refer to an Object That is Not const

Binding a reference to `const` to an object says nothing about whether the underlying object itself is `const`. Because underlying object might be non`const`, it might be change by other means

```
int i 42;
int &r1 = i;
const int &r2 = i;
r1 = 0;                 //valid because r is not const
r2 = 0;         // invalid becuae r2 is a const
```

Binding `r2` to the `int i` is legal. However, we cannot use `r2` to change `i`. Even so, the value in `i` still might change. We can change `i` by assigning to it directly, or by assigning to anther reference bound to `i`, such as `r1`.

---

## Pointer and const

We can define pointers that point to either `const` or non`const` types. A pointer to `const` may not be used to change the object to which the pointer points. We may store the address of a `const` only in a pointer to `const`

```
const double pi = 3.14 //unchangable
double *ptr = &pi;             //invalid
const double *cptr = &pi;      // valid
*cptr = 42;                    // invalid
```

A pointer to `const` says nothing about whether the object to which the pointer points is `const`. Defining a pointer as a pointer to `const` affects only what we can do with the pointer. It is important to remember that there is no guarantee that an object pointed to by a pointer to `const` won't change.

> **Tip**
>
> It may be helpful to think of pionters and references to `const` as pointers or references "that *think* they point or refer to `const`."

### const Pointers

Pointers are object therefore we can have a pointer that is itself `const`. A `const` pointer must be initialized, and once initialized, its value (address) may not be change. We indicate that the pointer is `const` by putting the `const` after the `*`. This placement indicates that it is the pointer, not the pointed-to type, that is `const`

```
int errNum = 0;
int *const curErr = &errNum;   //curErr will always point to errNum
const double pi = 3.14;
const double *const pip = &pi; //pip is a const pointer to a const object
```

---

## Top-Level const

We use the term **top-level** **`const`** to indicate that the pointer itself is a `const`. When a pointer can point to a `const` object, we refer to that `const` as a **Lowe-level** **`const`**.
A top-level `const` indicates that an object itself is `const`

**Top-level** **`const`**

- Pointer itself is a `const`
- can appear in any object type

**Low-level** **`const`**

- Pointer points to a `const` object
- appears as the base type of compound types such as pointer or references.

```
int i = 0;
int *const p1 = &i;     //top-level; cannot change the value of p1;
const int ci = 42;      //cannot change ci; top-level
const int *p2 = &ci     //can change p2; low-level
const int *const p3 = p2 //right-most const is top-level, left-most is not
const int &r = ci;      //const in reference type is always low-level
```

The distinction between top-level and low-level matter when we copy an object.

---

# constexpr and Constant Expressions

A constant expression is an expression whose values cannot change and that can be evaluated at compile time.

```
const int nice_number = 69;    //this is a constant expression
int sad_number = 17;           //not a constant expression
const int rand = random();     //rand is not a constant expression
```

In `rand` the `random` function should be a `constexpr`

## constexpr Variables

text can be widely separated. Under the new standard, we can ask the compiler to verify that a variable is a constant expression by declaring the variable in a `constexpr` declaration. Variables declared as `constexpr` are implicitly `const` and must be initialized by constant expressions:

```
constexpr int mf = 20; //20 is constant expression
constexpr int limit = mf + 1 // a constant expression
constexpr int sz = size()              // only if size is a constexpr function
```

---

# Pointers and constexpr

It is important to understand that when we define a pointer in `constexpr` declaration, the `const` specifier applies to the pointer, not the type to which the pointer points.

```
const int *p = nullptr //p is a pointer to a const int
constexpr int *q = nullptr     //q is a const pointer to int
```

**Note**

`p` is pointer to `const`, while `q` is a constant pointer

**Info**

`constexpr` imposes a top-level `const`