Const Qualifier in C

The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed (Which depends upon where const variables are stored, we may change value of const variable by using pointer). The result is implementation-defined if an attempt is made to change a const (See this forum topic).

1) Pointer to variable.

```
int *ptr;
```

Run on IDE

We can change the value of ptr and we can also change the value of object ptr pointing to. Pointer and value pointed by pointer both are stored in read-write area. See the following code fragment.

```
#include <stdio.h>
int main(void)
{
    int i = 10;
    int j = 20;
    int *ptr = &i;        /* pointer to integer */
    printf("*ptr: %d\n", *ptr);

    /* pointer is pointing to another variable */
    ptr = &j;
    printf("*ptr: %d\n", *ptr);

    /* we can change value stored by pointer */
    *ptr = 100;
    printf("*ptr: %d\n", *ptr);

    return 0;
}
```

Run on IDE

Output:

```
*ptr: 10
*ptr: 20
*ptr: 100
```

2) Pointer to constant.

Pointer to constant can be declared in following two ways.

```
const int *ptr;
```

Run on IDE

or

```
int const *ptr;
```

Run on IDE

We can change pointer to point to any other integer variable, but cannot change value of object (entity) pointed using pointer ptr. Pointer is stored in read-write area (stack in present case). Object pointed may be in read only or read write area. Let us see following examples.

```
#include <stdio.h>
int main(void)
{
    int i = 10;
    int j = 20;
    const int *ptr = &i;    /* ptr is pointer to constant */

    printf("ptr: %d\n", *ptr);
    *ptr = 100;        /* error: object pointed cannot be modified
                     using the pointer ptr */

    ptr = &j;          /* valid */
    printf("ptr: %d\n", *ptr);

    return 0;
}
```

Run on IDE

Output:

```
error: assignment of read-only location '*ptr'
```

Following is another example where variable i itself is constant.

```
#include <stdio.h>

int main(void)
{
    int const i = 10;    /* i is stored in read only area*/
    int j = 20;

    int const *ptr = &i;        /* pointer to integer constant. Here i
                                 is of type "const int", and &i is of
                                 type "const int *".  And p is of type
                                "const int", types are matching no issue */

    printf("ptr: %d\n", *ptr);

    *ptr = 100;        /* error */

    ptr = &j;          /* valid. We call it as up qualification. In
                         C/C++, the type of "int *" is allowed to up
                         qualify to the type "const int *". The type of
                         &j is "int *" and is implicitly up qualified by
                         the compiler to "cons tint *" */

    printf("ptr: %d\n", *ptr);

    return 0;
}
```

Run on IDE

Output:

```
error: assignment of read-only location ‘*ptr’
```

Down qualification is not allowed in C++ and may cause warnings in C. Following is another example with down qualification.

```
#include <stdio.h>

int main(void)
{
    int i = 10;
    int const j = 20;

    /* ptr is pointing an integer object */
    int *ptr = &i;

    printf("*ptr: %d\n", *ptr);

    /* The below assignment is invalid in C++, results in error
       In C, the compiler *may* throw a warning, but casting is
       implicitly allowed */
    ptr = &j;

    /* In C++, it is called 'down qualification'. The type of expression
       &j is "const int *" and the type of ptr is "int *". The
       assignment "ptr = &j" causes to implicitly remove const-ness
       from the expression &j. C++ being more type restrictive, will not
       allow implicit down qualification. However, C++ allows implicit
       up qualification. The reason being, const qualified identifiers
       are bound to be placed in read-only memory (but not always). If
       C++ allows above kind of assignment (ptr = &j), we can use 'ptr'
       to modify value of j which is in read-only memory. The
       consequences are implementation dependent, the program may fail
       at runtime. So strict type checking helps clean code. */

    printf("*ptr: %d\n", *ptr);

    return 0;
}

// Reference http://www.dansaks.com/articles/1999-02%20const%20T%20vs%20T%20const.pdf

// More interesting stuff on C/C++ @ http://www.dansaks.com/articles.htm
```

Run on IDE

3) Constant pointer to variable.

```
int *const ptr;
```

Run on IDE

Above declaration is constant pointer to integer variable, means we can change value of object pointed by pointer, but cannot change the pointer to point another variable.

```
#include <stdio.h>

int main(void)
{
   int i = 10;
   int j = 20;
   int *const ptr = &i;    /* constant pointer to integer */

   printf("ptr: %d\n", *ptr);

   *ptr = 100;    /* valid */
   printf("ptr: %d\n", *ptr);

   ptr = &j;        /* error */
```

```
    return 0;
}
```

Run on IDE

Output:

```
error: assignment of read-only variable ‘ptr’
```

4) constant pointer to constant

```
const int *const ptr;
```

Run on IDE

Above declaration is constant pointer to constant variable which means we cannot change value pointed by pointer as well as we cannot point the pointer to other variable. Let us see with example.

```
#include <stdio.h>

int main(void)
{
    int i = 10;
    int j = 20;
    const int *const ptr = &i;        /* constant pointer to constant integer */

    printf("ptr: %d\n", *ptr);

    ptr = &j;            /* error */
    *ptr = 100;        /* error */

    return 0;
}
```

Run on IDE

Output:

```
      error: assignment of read-only variable ‘ptr’
      error: assignment of read-only location ‘*ptr’
```

This article is compiled by “**Narendra Kangralkar**“. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Practice Tags : C

Article Tags : C C-Storage Classes and Type Qualifiers pointer

Login to Improve this Article

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Recommended Posts: