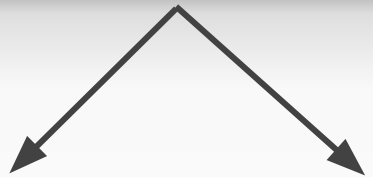
C++ constant

Joost Akkermans, Serge Knecht, Randy Paredis, David Schotmans

constexpr vs const



constexpr

Compile time evaluation

Value must be known at compilation

Integral type(bool, char, integer)
Floating-point types
Enumerator

const

Expresses immutability in interfaces
(implies a guarantee that, once initialized, the value of that object won't change)

Can be initialized at compilation time or at run time

Compiler Optimizations

constexpr

Constant expression

Available since C++ 11

Declaration that it is possible to evaluate the value of a function or variable at compile time

```
const double PI1 = 3.141592653589793;
constexpr double PI2 = 3.141592653589793;
```

may be initialized at compile time or run time
must be initialized at compile time

```
constexpr double PI3 = PI1;
constexpr double PI4 = PI2;
```

NOT ALLOWED
ALLOWED

constexpr function

Unlike **constexpr** can also be applied to functions and class constructors.

Example:

```
constexpr int MeaningOfLife ( int a, int b ) { return a * b; }
constexpr int meaningOfLife = MeaningOfLife( 6, 7 );
```

Function MeaningOfLife is calculating the constant values at compile time

Limits on its definition:

- Only operators are allowed that don't modify state (e.g. +, -,?:, [])
- Therefore not allowed: =, ++

const modifies type restricting its usage

Pointer to constant

```
void f1(char* p) {
char s[] = “Gorm”;
const char* pc = s;
pc[3] = ‘g’;
pc=p;
}
```

Constant pointer

```
void f1(char* p) {
char s[] = “Gorm”;
char *const pc = s;
pc[3] = ‘g’;
pc=p;
}
```

Constant pointer to constant

```
void f1(char* p) {
char s[] = “Gorm”;
const char *const pc = s;
pc[3] = ‘g’;
pc=p;
}
```

const modifies type restricting its usage

Pointer to constant

```
void f1(char* p) {
char s[] = “Gorm”;
const char* pc = s;
pc[3] = ‘g’;          //error
pc=p;
}
```

Constant pointer

```
void f1(char* p) {
char s[] = “Gorm”;
char *const pc = s;
pc[3] = ‘g’;
pc=p;                 //error
}
```

Constant pointer to constant

```
void f1(char* p) {
char s[] = “Gorm”;
const char *const pc = s;
pc[3] = ‘g’;          //error
pc=p;                 //error
}
```

const correctness

It means using the keyword `const` to prevent `const` objects from getting mutated.

- Type safety
  - Zegt tegen de compiler dat er niets mag worden aangepast.
  - Verschillende klassen (`const std::string` ↔ `std::string`)
- Wanneer toevoegen?
  - **Zo vroeg mogelijk!**
- Leesrichting = rechts naar links
- *The constness of a method must make sense to the object's users.*
- mutable / *const_cast*
  - **C++11:** `#include <type_traits> (std::add_const<T>, std::remove_const<T>)`

NOTE: there is an extremely unlikely error that can occur with const_cast. It only happens when three very rare things are combined at the same time: a data member that ought to be mutable (such as is discussed above), a compiler that doesn't support the mutable keyword and/or a programmer who doesn't use it, and an object that was originally defined to be const (as opposed to a normal, non-const object that is pointed to by a pointer-to-const). Although this combination is so rare that it may never happen to you, if it ever did happen, the code may not work (the Standard says the behavior is undefined).

const correctness

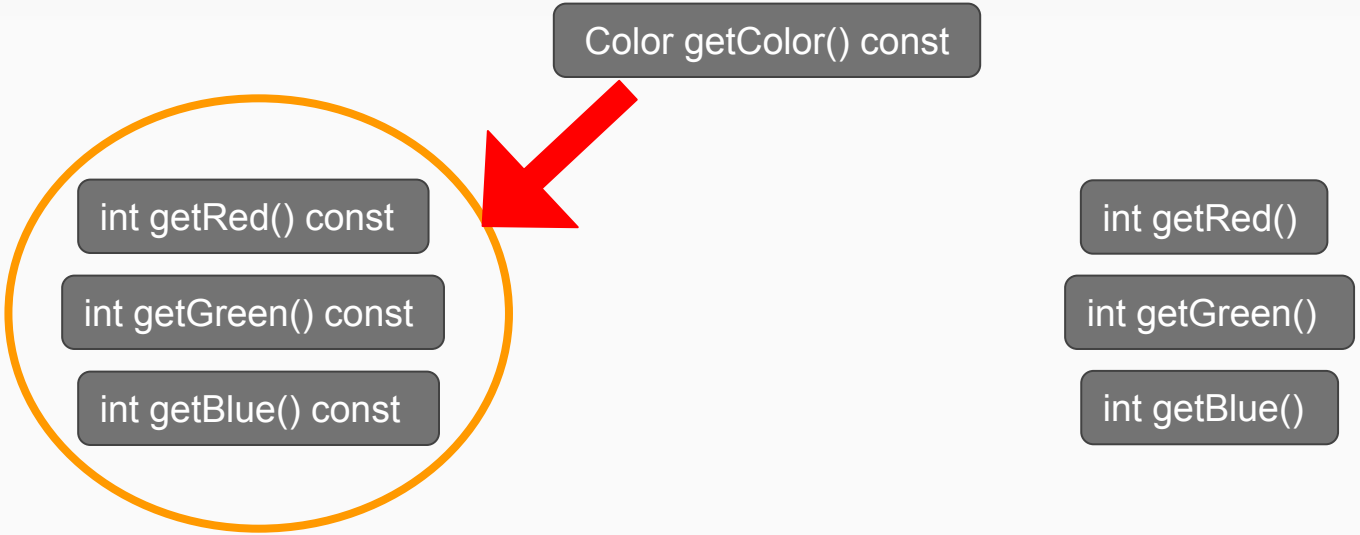
When passing parameters by reference to functions or constructors, be very careful about const correctness.

Pass by non-const reference ONLY if the function will modify the parameter and it is the intent to change the caller's copy of the data, otherwise pass by const reference.

Why is this so important?

There is a small clause in the C++ standard
that says that non-const references cannot bind to temporary objects.
A temporary object is an instance of an object that does not have a variable name.

const function only accessing const function path



Example

```
#include <iostream>

class A{
public:
    A(){}
    void f() const{
        std::cout << "Using f() const: value i = " << i << std::endl;
    }

    void f(){
        i++;
        std::cout << "Using f(): value i = " << i << std::endl;
    }

    int i = 0;
};
```

```
int main(){
    A a;
    a.f();

    const A b;
    b.f();
    return 0;
}
```

```
Using f(): value i = 1
Using f() const: value i = 0
```

const data members

Const data members kunnen met een verschillende waarde geïnitialiseerd worden per object (vs. static data members).

Const data members zorgen ervoor dat de ***impliciet gegenereerde* assignment operator** wordt **impliciet verwijderd** door de compiler

Const Data members moeten geïnitialiseerd worden in de *initializer list* van de constructor van de klasse waarin het const data member wordt gedeclareerd.

Initialization ≠ Assignment

const data members

```cpp
#include <iostream>

using namespace std;

class Test {
public:
    Test(int val) : i(val) {

    }

    Test(const Test& that) : i(that.i) {

    }

    // Test& operator=(const Test & that) {
    //  return *this;
    // }

    int getValue() const {
        return i;
    }
private:
    const int i;
};
```

```cpp
int main() {
    Test t(3);

    Test b(2);

    b = t;

    cout << "b.i = " << b.getValue() << "\n";

    return 0;
}
```

Assignment operator impliciet verwijderd

```
constness.cpp: In function 'int main()':
constness.cpp:35:4: error: use of deleted function 'Test& Test::operator=(const Test&)'
  b = t;
    ^
constness.cpp:5:7: note: 'Test& Test::operator=(const Test&)' is implicitly deleted because
the default definition would be ill-formed:
 class Test {
       ^
constness.cpp:5:7: error: non-static const member 'const int Test::i', can't use default ass
ignment operator
```

const data member beperking is omzeilbaar

```
5   class Test {
6   public:
7       Test(int val) : i(val) {
8
9       }
10
11      Test(const Test& that) : i(that.i) {
12
13      }
14
15      Test& operator=(const Test & that) {
16          int * var = const_cast<int*>(&i);
17          *var = that.i;
18          return *this;
19      }
20
21      int getValue() const {
22          return i;
23      }
24  private:
25      const int i;
26  };
```

Definieer een custom assignment-operator => constness van data member toch omzeilbaar

```
29  int main() {
30      Test t(3);
31
32      Test b(2);
33
34      cout << "b.i = " << b.getValue() << "\n";
35
36      b = t;
37
38      cout << "b.i = " << b.getValue() << "\n";
39
40      return 0;
41  }
```

```
b.i = 2
b.i = 3
```

const_cast<type*>

Undefined behaviour

```
#include <iostream>
using namespace std;

int main(){
    const int a = 3;
    int* b = const_cast<int*>(&a);
    *b = 5;
    cout << "Value of a: " << a << endl;
    cout << "Value of *b: " << *b << endl;
    return 0;
}
```

```
Value of a: 3
Value of *b: 5
```

MIPS Assembly versie → code optimalisatie

```
.cprestore      16
li      $2,3            # 0x3
sw      $2,28($fp)
addiu   $2,$fp,28
sw      $2,24($fp)
lw      $2,24($fp)
li      $3,5
sw      $3,0($2)
```

Wegschrijven 3 naar A

Wegschrijven 5 naar B zonder const_cast

```
$LC0:
        .ascii  "Value of a: \000"
        .align  2
$LC1:
        .ascii  "Value of *b: \000"
        .text
        .align  2
```

Compiler optimaliseert code