C++

Const

Const

- The keyword **const** has different semantics in C++ depending on its context
- **const** in variable declarations means that the value of the variable can not be changed: const float PI = 3.14156; // PI is a constant
- In C, **#define** was used to define constant. In C++, it is much better to use const**.**
- Using **const** for declaring constant variables provides additional safety (type checking)
- Trying to change a const object results in compile time error.

Example

```
#include <iostream>

const float PI = 3.14156;
const int i; // compile time error: const must be initialized

int main() {
  std::cout << “Pi is: “ << PI << std::endl;
  PI = 3.25; // trying to redefine PI <- compile time error
}
```

Example 2

```
class Date {
 public:
  Date(int m, int d, int y) {
    month=m; day=d; year=y;
  }
  void inc_day() {day++;}
  int process (const Date& d) {
   // d is const and can not be modified
   // in process()
  }
 private:
  int month, day, year;
};
```

```
int main () {
 const Date d1(1, 15, 2010);
 d1.inc_day(); // compile-time error

 Date d2(1, 16, 2010);
 int temp = d1.process(d2); //
compile-time error
}
```

Pointer to a const

- “const int* pi” is a **pointer to a const**
  - pi can either be a const or not
  - pi can be changed
  - *pi cannot be changed (it cannot be used in an assignment)

```
const int i = 1; int j = 2;
const int* pi;
pi = &i; pi = &j; // ok
pi = &i; *pi = 3; // compile-time error
pi = &j; *pi = 4; // compile-time error
int* pi2; pi2 = &i; // compile-time error
pi2 = &j; *pi2 = 5; // ok
```

Const pointer

- It is also possible to have a pointer that is constant (we cannot change the address it points to). This implies nothing about the object being pointed to

```
int a = 5;
int* const pa = &a; // const pointer so must be assigned at initialization
*pa = 10;           // ok now *pa and a are 10
int b = 2;
pa = &b;            // compile-time error: pa is a constant
```

Const pointer to a constant

- Both the pointer and object pointed to are constant.

```
const int a = 1;
const int* const pa = &a;
*pa = 2; // compile-time error
int b = 2; pa = &b; // compile-time error
```

- Note: a const pointer to a const can still point to a non const but it can not change it:

```
int a = 1;
const int* const pa = &a; // ok
*pa = 2;  // compile-time error
```

Const and pointer: summary

- We saw three cases:
  1. const int* pa; // a pointer to a constant
  2. int* const pa = &a; // a const pointer to an int
  3. const int* const pa = &a; // a const pointer to a constant int
- Three important points:
  1. Read the declarations from right to left to figure what is constant
  2. A constant object can not be changed
  3. If p is a pointer to a const then *p can not be assigned to

References to const

- Similar to pointers, references can refer to const:

```
int a = 1;
const int& ra = a;
a = 2;  // ok
ra = 3; // compile-time error
```

- The key is that **ra** is a reference to a constant therefore **ra** cannot be assigned to (same logic as for pointer)
- Note: a const reference makes no sense because a reference cannot be reseated so is by definition const !

References to const as function arguments

- We saw that it is preferable in C++ to pass arguments by reference
- If you intend to have your function leave the argument(s) unchanged, then you need to express this intention by using the **const** keyword:

```
int my_func(const Obj& o);
```

References to const as function arguments

- Accidentally modifying the argument will be caught by the compiler:

```
void func(Obj& o) {
  o.speed += 50;  // ok
}

void func_2(const Obj& o) {
  o.speed += 50;  // compile-time error
}
```

References to const as function arguments

- Note: a function with a const reference argument can be called with either a const or a non-const argument. While a function without a const reference can be called only with a non-const argument.

```
void f1(const Obj& o) { // … }
void f2(Obj& o) {// …}

int main() {
  Obj o1;
  const Obj o2;
  f1(o1); // ok
  f1(o2); // ok
  f2(o1); // ok
  f2(o2); // compile-time error
}
```

Const member function

- A member function that does not modify an object should indicate so by using the **const** keyword after its argument list.

```
class A{
public:
  int get_day() const { return day; } // const
member function
  void set_day(int d) { day = d; } // non-const
member function
 // …
};
```

`mutable`

- A member of a class can be defined as `mutable`. It means that it can be modified, even in a `const` object
- Solve the problem of logical vs physical constness
- Logical constness: from a user point of view, a function does not appear to change the state of an object, but some implementation details (not directly observable) need to be updated

mutable

```
class Date {
 public:
   string repr() const; // string representation
 private:
   mutable bool cache_valid;
   mutable string cache;
   void compute_cache() const;
};
```

mutable

```
string Date::repr() const {
 if (!cache_valid) {
   compute_cache(); // fill mutable cache
   cache_valid = true;
 }
 return cache;
}
```

`mutable`

- A similar effect can be obtained by placing the changing data in a separate object and accessing it via a pointer
- `const` does not apply to objects accessed via pointers (or references)

Usage of const

- Programs that compile correctly (and work correctly) can be produced without using the `const` keyword at all
- However as a good software engineering practice, the `const` keyword is used to help track (or prevent) potential bugs as early as possible (at compile-time)

Usages of `const`

- We saw three usages of `const`:
  - For object that cannot change
  - For arguments to function that should not be changed by the function
  - For member functions that should not change the object's state