Programmazione

Prof. Marco Bertini
marco.bertini@unifi.it
http://www.micc.unifi.it/bertini/

Const correctness

What is const correctness ?

- It is a semantic constraint, enforced by the compiler, to avoid modification of a particular object marked as `const`
- `const` can be used in various scopes:
  - outside of classes at global/namespace scope:

```
const double AspectRatio = 1.653;
// much better than a C style define:
#define ASPECT_RATIO 1.653
```

Class constants

- It's usable for static objects at file, function and block level
- It's usable also for class specific constants, e.g. for static and non-static data members:

```
class VideoFrame {
private:
  static const int PALFrameRate;
  ...
};
const int VideoFrame::PALFrameRate = 25;
```

Pointers and constancy

- We can specify that a pointer is constant, that the data pointed to is constant, that both are constant (or neither):

```
char greeting[] = “Hello”;
char* p = greeting; // nothing is constant

const char* p = greeting; //non-const pointer
                          //    const data
char* const p = greeting; //    const pointer
                          // non-const data
const char* const p = greeting; // everything is const
```

Pointers and constancy - cont.

- If `const` appears to the left of * then what is pointed to is constant, if it's on the right then the pointer is constant:

`const char* const p` means that `p` is a constant pointer to constant chars

- according to this writing `char const* p` is the same of `const char* p`

References and constancy

- You can not change an alias, i.e. you can't reassign a reference to a different object, so:

`Fred& const x` makes no sense (it's the same thing of `Fred& x`), however:

`const Fred& x` is OK: you can't change the Fred object using the x reference.

Functions and constancy

- The most powerful use of const is its application to function declarations: we can refer to function return value, function parameters and (for member functions) to the function itself
- Helps in reducing errors, e.g. you are passing an object as parameter using a reference/pointer and do not want to have it modified:

```
void foo(const bar& b);
// b can't be modified
// use const params whenever possible
```

const return value

- Using a const return value reduces errors in client code, e.g.:

```
class Rational { //...};
const Rational operator*(
                        const Rational& lhs,
                      const Rational& rhs
                          );

Rational a,b,c;
// let's say we missed an =
// to make a comparison...
(a*b)=c; // it's now illegal thanks to
         // const return value !
```

const return value - cont.

- When returning a reference probably it's better to return it as constant or it may be used to modify the referenced object:

```
class Person {
public:
  string& badGetName() {
       return name;
  }
  //...
private:
  string name;
};

void myCode(Person& p) {
  p.badGetName() = “Igor”; // can change the name
                           // attribute of Person
}
```

const member functions

- The purpose of const member functions is to identify which functions can be invoked on const objects.
  These functions inspect and do not mutate an object.
- NOTE: it's possible to overload methods that change only in constancy !
  It's useful if you need a method to inspect and mutate with the same name

const member functions - cont.

```
class TextBlock {
public:
 const char& operator[](size_t pos) const {
   return text[pos];
 }
 char& operator[](size_t pos) { // has to be reference
   return text[pos];                // to be modifiable:
 }                                  // C++ returns by value !
private:
 string text;
};
```

- this is useful when dealing with objects that are passed as const references:

```
void print(const TextBlock& ctb, size_t pos) {
  cout << ctb[pos]; // calls the const version of []
};
```

`const` member functions - cont.

- C++ compilers implement bitwise constancy, but we are interested in logical constancy, e.g. the const reference return value seen before or we may need to modify some data member within a const method (declared `mutable`):

```
class TextBlock {
public:
  size_t length() const;
private:
  string text;
  mutable size_t length;
  mutable bool isValidLength;
};
```

```
size_t TextBlock::length()
const {
 if(!isValidLength) {
  length=text.size();
  isValidLength=true;
 }
 return length;
}
```

const member functions - cont.

- To avoid code duplication between const and non-const member functions that have the same behaviour can be solved:
  - putting common tasks in private methods called by the two versions of the const/ non-const methods
  - casting away constancy, with the non-const method calling the const method (see future lecture)

Reading material

- M. Bertini, "Programmazione Object-Oriented in C++" - pp. 46-49
- B. Stroustrup, "C++, guida essenziale per programmatori" - pp. 8-9
- B. Stroustrup, "C++, Linguaggio, libreria standard, principi di programmazione", pp. 42, 173-174, 281, 420
- L.J. Aguilar, "Fondamenti di programmazione in C++. Algoritmi, strutture dati e oggetti" - pp. 84, pp. 125-128
- D.S. Malik, "Programmazione in C++" - pp. 43-45, 47-48

Credits

- These slides are based on the material of:
  - Marshall Cline, C++ FAQ Lite
  - Scott Meyers, "Effective C++", 3rd edition, Addison-Wesley