# 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 isValidLenght;
};
```

```
size_t TextBlock::length()
const {
 if(!isValidLengt) {
  _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 later)

# C++ and casting

# C++ casting

- C++ casts are more restricted than C style casts
- In general the lesser we cast the better: C++ is a type safe language and casts subvert this behaviour
  - e.g. `const_cast` can be used to eliminate code duplication: the benefits are worth the risk

# C and C++ casts

- C style casts, to cast an expression to be of type T:
  - (T) expression
  - T(expression)
- C++ style casts:
  - static_cast<T>(expression)
  - dynamic_cast<T>(expression)
  - const_cast<T>(expression)
  - reinterpret_cast<T>(expression)

# static_cast

- `static_cast` forces implicit conversions, such as non-const objects to const objects (as seen in const/non-const methods), int to double, `void*` to typed pointers, pointer-to-base to pointer-to-derived (but no runtime check).
- it's the most useful C++ style cast

```
int j = 41;
int v = 4;
float m = j/v; // m = 10
float d = static_cast<float>(j)/v; // d = 10.25

BaseClass* a = new DerivedClass();
static_cast<DerivedClass*>(a)->derivedClassMethod();
```

# static_cast - cont.

- Prefer `static_cast` over C style cast, because we get the type safe conversion of C++:

```
class MyClass : public MyBase {...};
class MyOtherStuff {...} ;
MyBase  *pSomething; // filled somewhere
MyClass *pMyObject;
pMyObject = static_cast<MyClass*>(pSomething);
// Safe, as long as we checked
pMyObject = (MyClass*)(pSomething); // Same as static_cast<>
// Safe; as long as we checked but harder to read
MyOtherStuff *pOther;
pOther = static_cast<MyOtherStuff*>(pSomething);
// Compiler error: Can't convert
pOther = (MyOtherStuff*)(pSomething); // No compiler error.
            // Same as reiterpret_cast<> and it's wrong!!!
```

# dynamic_cast

- dynamic_cast performs safe (runtime check) downcasting: i.e. determines if an object is of a particular type in an inheritance hierarchy.
  - it has a runtime cost depending on the compiler implementation

```
class Window { //... };
class SpecialWindow :
public Window {
public:
 void blink();
};
```

```
Window* pW;
//...pW may point to whatever object
// in Window hierarchy

if( SpecialWindow*
pSW=dynamic_cast<SpecialWindow*>pw )
  pSW->blink();
```

# const_cast

- `const_cast` is used to cast away the const*ness* of an object
- It's the only cast that can do it

# `const` member functions

- Let's review again how to avoid code duplication between const and non-const member functions...
  - the non-const method calls the const method and then cast away its constancy with `const_cast`

# const member functions - cont.

```
class TextBlock {
public:
 const char& operator[](size_t pos) const {
   //... checks over boundaries, etc.
   //...
   return text[pos];
 }
 char& operator[](size_t pos) {
   return
     const_cast<char&>( // take away constancy
       static_cast<const TextBlock&>(*this)[pos] // add constancy
     );
 }
 //...
};
```

# const member functions - cont.

```
class TextBlock {
public:
 const char& operator[](size_t pos) const {
   //... checks over boundaries, etc.
   //...
   return text[pos];
 }
 char& operator[](size_t pos) {
   return
     const_cast<char&>( // take away constancy
       static_cast<const TextBlock&>(*this)[pos] // add constancy
     );
 }
 //...
};
```

Don’t panic: **first** cast to const, to call the const method, **then** remove const-ness

# reinterpret_cast

- `reinterpret_cast` is used for low-level casts, e.g. to perform conversions between unrelated types, like conversion between unrelated pointers and references or conversion between an integer and a pointer.
- It produces a value of a new type that has the same bit pattern as its argument. It is useful to cast pointers of a particular type into a `void*` and subsequently back to the original type.
  - may be perilous: we are asking the compiler to trust us...

# Reading material

- 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
- Thinking in C++, 2nd ed. Volume 1, pp. 165-167, 181-186

# Credits

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