Programmazione

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

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.

Overloading of operator[]:
A const version to read data and a non-const to modify it
Goal: write only a version of the method to avoid code duplication

```
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.

Overloading of operator[]:
A const version to read data and a non-const to modify it
Goal: write only a version of the method to avoid code duplication

```
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

- B. Stroustrup, "C++, guida essenziale per programmatori" - pp. 161
- L.J. Aguilar, "Fondamenti di programmazione in C++. Algoritmi, strutture dati e oggetti" - pp. 125-128
- D.S. Malik, "Programmazione in C++" - pp. 43-45
- Thinking in C++, 2nd ed. Volume 1, pp. 181-186

Reading material

- M. Bertini, "Programmazione Object-Oriented in C++", cap. 3 - pp. 85-88

Credits

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