C++ Programming I

Casting Operators in C++

C++ Programming
May 3, 2018

Dr. P. Arnold
Bern University of Applied Sciences

Agenda

- **Casting Operators**
  - `static_cast`
  - `dynamic_cast`
  - `reinterpret_cast`
  - `const_cast`
  - Acceptance Casting Operators

Casting Operators

The need of Casting Operators in C++

Intro

- In a perfectly type-safe and type-strong world comprising well-written C++ applications, there should be no need for casting and for casting operators!
- In real world however, different people, vendors using different environments have to work together!
- Therefore, the compilers have to be **forced** to interpret data in way that it can compile
- For example the native type `bool` did not exist years back in C. Libraries made for C compilers rely on an integral type holding boolean data
- A `bool` was for example defined as
  `typedef unsigned short BOOL`
- In order to use such a C-Style library one has to provide a way to make the C++ -compiler type `bool` and the library type `BOOL` work together.
- This happens by using casts:

```cpp
// C-Style library function returning BOOL
BOOL IsX();

bool Result = (bool)IsX(); // C-Style cast
```

The need of Casting Operators in C++

Intro

- The evolution of C++ saw the emergence of new C++ casting operators to meet the requirements for strong type safety!
- Most C++ compilers will rightfully not compile:

```
char* staticStr = "Hello World!";
int* intArray = staticStr; // error: cannot convert char* to int*
```

The need of Casting Operators in C++
Intro

- The evolution of C++ saw the emergence of new C++ casting operators to meet the requirements for strong type safety!
- Most C++ compilers will rightfully not compile:

```
char* staticStr = "Hello World!";
int* intArray = staticStr; // error: cannot convert char* to int*
```

- To be backward compliant (old and legacy code still compiles) following syntax is automatically allowed

```
// Cast one problem away, create another
int* intArray = (int*)staticStr;
```

- This C-style cast actually **forces** the compiler to interpret the destination as a type chosen by the programmer, thus **not type safe!**

Casting Operators

supplied by C++

- C++ supplies four casting operators:
  1. `static_cast`
  2. `dynamic_cast`
  3. `reinterpret_cast`
  4. `const_cast`

Casting Operators

supplied by C++

- C++ supplies four casting operators:
  1. `static_cast`
  2. `dynamic_cast`
  3. `reinterpret_cast`
  4. `const_cast`
- The usage syntax of the casting operators is consistent:

```
destType result = cast_operator<destType>(objToCast);

// For example a simple type conversion
const float PI = 3.14519;

// C++ cast
int intPI = static_cast<int>(PI);

// C-style cast
int intPI = (int)PI;

// or functional notation
int intPI = int(PI);
```

- Same result for all!

static_cast

Pointer Conversion & Explicit Standard Type Conversion

- `static_cast` is used in two scenarios:
  1. Convert pointers between related types (up- & downcasting)
  2. Perform explicit type conversions for standard data types that would otherwise happen automatically or implicitly
- `static_cast` implements a basic compile-time check to ensure that the pointer is being cast to a related type (improvement)
- C-style casts allow casting a pointer to one object to a absolutely unrelated object!

```cpp
Base* objBase = new Derived();
Derived* objDer = static_cast<Derived*>(objBase); // ok!

// class Unrelated is not related to Base
Unrelated* notRelated = static_cast<Unrelated*>(objBase); //Error

// C-Style cast works!
Unrelated* notRelated = (Unrelated*)objBase; //ok
```

static_cast
Up- & Down Casting

- Casting a `Derived*` to a `Base*` is called **upcasting** and can be done without any explicit casting operator
- Casting a `Base*` to a `Derived*` is called **downcasting** and cannot be done without usage of explicit casting operators

```
Derived objDerived;
Base* objBase = &objDerived; // Upcast -> ok!
Derived* objDer = objBase; // Error: Downcast needs explicit cast
```

static_cast

Up- & Down Casting

- Casting a `Derived*` to a `Base*` is called **upcasting** and can be done without any explicit casting operator
- Casting a `Base*` to a `Derived*` is called **downcasting** and cannot be done without usage of explicit casting operators

```
Derived objDerived;
Base* objBase = &objDerived; // Upcast -> ok!
Derived* objDer = objBase; // Error: Downcast needs explicit cast
```

- However, `static_cast` verifies only that the pointer types are related and does not perform a run-time check!
- You can still compile the following bug:

```
Base* objBase = new Base();
Derived* objDer = static_cast<Derived*>(objBase); // No error!

// Ups!
objDer->DerivedFunction();
```

static_cast

Explicit static_cast

- Make implicit casts explicit

```
destType result = cast_operator<destType>(objToCast);

// For example a simple type conversion
const float PI = 3.14519;

// C++ cast
int intPI = static_cast<int>(PI);

// C-style cast
int intPI = (int)PI;

// or functional notation
int intPI = int(PI);
```

- Using a `static_cast` brings the nature of conversion to the attention of the reader

dynamic_cast

Runtime Casting

- Dynamic casting is the opposite of static casting and actually executes the cast at runtime
- The result of a `dynamic_cast` operation can be checked to see whether the attempt at casting succeeded

```
Base* objBase = new Derived();

// Perform a downcast
Derived* objDer = dynamic_cast<Derived*>(objBase);

if(objDer) // Check for success of the cast
    objDer->CallDerivedFunction();
```

dynamic_cast

Runtime Casting

- Dynamic casting is the opposite of static casting and actually executes the cast at runtime
- The result of a `dynamic_cast` operation can be checked to see whether the attempt at casting succeeded

```
Base* objBase = new Derived();

// Perform a downcast
Derived* objDer = dynamic_cast<Derived*>(objBase);

if(objDer) // Check for success of the cast
    objDer->CallDerivedFunction();
```

- `dynamic_cast` cat determine the type at runtime and use a casted pointer when it is safe to do so!
- This mechanism of identifying the type of the object at runtime is called *runtime type identification* (RTTI)
- Check the next example

dynamic_cast
Runtime Type identification RTTI

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

class Fish
{
public:
    virtual void swim(){cout << "Fish swims" << endl;}
    virtual ~Fish(){} // must be virtual
};

class Tuna: public Fish
{
public:
    void swim(){cout << "Tuna swims real fast" << endl;}

    void becomeDinner(){cout << "Tuna became Sushi" << endl;}
};

class Carp: public Fish
{
public:
    void swim(){cout << "Carp swims real slow" << endl;}

    void talk(){cout << "Carp talked carp!" << endl;}
};
```

- Note: Next to implement `swim()`, `Tuna` and `Carp` contain a specific function each!

dynamic_cast

Runtime Type identification RTTI

```
void detectFishType(Fish* objFish)
{
    Tuna* objTuna = dynamic_cast<Tuna*>(objFish);
    if (objTuna)
    {
        cout << "Detected Tuna. Making Tuna dinner: " << endl;
        objTuna->becomeDinner();   // calling Tuna::BecomeDinner
    }

    Carp* objCarp = dynamic_cast<Carp*>(objFish);
    if(objCarp)
    {
        cout << "Detected Carp. Making carp talk: " << endl;
        objCarp->talk();  // calling Carp::Talk
    }

    cout << "Verifying type using virtual Fish::Swim: " << endl;
    objFish->swim(); // calling virtual function Swim
}
```

- Given an instance of the base class `Fish*` , you are able to dynamically detect whether that pointer points to a `Tuna` or a `Carp`!
- The return value of a `dynamic_cast` operation should be checked for validity - it's `nullptr` when the cast fails

Casting Operators

dynamic_cast

Runtime Type identification RTTI

```
int main()
{
    Carp myLunch;
    Tuna myDinner;

    detectFishType(&myDinner);
    detectFishType(&myLunch);

    return 0;
}

// Output

// Detected Tuna. Making Tuna dinner:
// Tuna became Sushi
// Verifying type using virtual Fish::Swim:
// Tuna swims real fast

// Detected Carp. Making carp talk:
// Carp talked Carp!
// Verifying type using virtual Fish::Swim:
// Carp swims real slow
```

- Note: The same functionality can and probably should be implemented using virtual functions!

`reinterpret_cast`

The closest C-Style Cast

- `reinterpret_cast` is the closest a CC++ casting operator gets to the C-style cast
- The following code compiles, *i.e.* forces the compiler to accept the situation!

```
Base* objBase = new Base();
Unrelated* notRelated = reinterpret_cast<Unrelated*>(objBase);
```

reinterpret_cast
The closest C-Style Cast

- `reinterpret_cast` is the closest a CC++ casting operator gets to the C-style cast
- The following code compiles, *i.e.* forces the compiler to accept the situation!

```
Base* objBase = new Base();
Unrelated* notRelated = reinterpret_cast<Unrelated*>(objBase);
```

- The usage should be restricted to certain low-level applications (*e.g* drivers) where data has to be converted to a simple type to be accepted by the API - Application Programming Interface:

```
PrintCommand* object = new PrintCommand();
// Need to send the object as a byte-stream...
char* bytesFoAPI = reinterpret_cast<char*>(object);
```

Note:

Avoid `reinterpret_cast` as far as possible

const_cast

Changing const specifier

- const_cast enables you to turn off the const access modifier

const_cast

Changing const specifier

- const_cast enables you to turn off the const access modifier
- Use case: You're given a class DisplayClass you can't change

```
class DisplayClass // third-party library
{
public:
    // ...
    void displayMembers(); //problem-display function not const
};
```

- You implement a displayAllData function and obviously you don't want your object to be changed when displayed
- But if fails to compile:

```
void DisplayAllData(const DisplayClass& object)
{
    object.displayMembers(); // Compile failure
    // reason: call to a non-const member using a const reference
}
```

const_cast

Changing const specifier

- const_cast enables you to turn off the const access modifier
- Use case: You're given a class DisplayClass you can't change

```
class DisplayClass // third-party library
{
public:
    // ...
    void displayMembers(); //problem-display function not const
};
```

- You implement a displayAllData function and obviously you don't want your object to be changed when displayed
- But if fails to compile:

```
void DisplayAllData(const DisplayClass& object)
{
    object.displayMembers(); // Compile failure
    // reason: call to a non-const member using a const reference
}
```

- As a last resort you cast away your constness:

```
void DisplayAllData(const DisplayClass& object)
{
    DisplayClass& refData = const_cast<DisplayClass&>(object);
    refData.displayMembers(); // Allowed!
}
```

Casting Operators
Problems with the C++ Casting Operators

- Because of the syntax and being redundant the use of the C++ casting operators is discussed controversially:

```cpp
double Pi = 3.14159265;

int num = static_cast<int>(Pi); // C++ style cast: static_cast

int num2 = (int)Pi; // C-style cast

int num3 = Pi; // leave casting to the compiler
```

- In all cases the same result is achieved
- Practically, the second one is the most used version followed by the third!
- The advantage of using `static_cast` is often overshadowed by the clumsiness of its syntax
- C++ casting operators other than `dynamic_cast` are avoidable in modern C++ applications
- It's matter of taste!
- Most important:
  1. Avoid casting as far as possible (design!)
  2. If you do, you should know what happens behind the scenes

Thank You

Questions

???