module06

Jump to bottom

qingqingqingli edited this page on Jan 28, 2021 · 3 revisions

Table of content

Introduction

- Cast is essentially conversion, which allows us to transform the bits of one type to another type. For instance, int and double are saved differently in bits. To go from one type to the other, the bits need to be converted.
- With `identity conversion`, bits after conversion are not re-ordered. It's called `reinterpretation`, which allows us to work on more generic, more accurate types of addresses. Reinterpretation include: downcast, upcast, type qualifier
- C offers 2 types of casts (don't use them for C++):
  - implicit cast
  - explicit cast

- C++ offers 5 types of casts:
  - `implicit cast` : only conversion of simple values and upcast
  - `static_cast` : with downcast and upcast, we know what we want and where we are going from the inheritance tree. This will not prevent crosscast, but gives issues at run time. But it will prevent cast from classes of two different inheritance trees.
  - `dynamic_cast` : only cast that happens at runtime. It adds certain performance overheads to your program. It takes advantage of `rtti` (run-time type information). The class must have one virtual member function. It will check if the transform from one form to another is realistic base on the hierarchy. It is one of the base principles, hidden behind the notion of plugin
  - `const_cast` : will not be used often. Consider if it's a design flaw when you need to use this cast
  - `reinterpret_cast` : the most open cast. No semantics check. The most suitable case is to change the type of some raw data, or you want to convert the type of one data to another type that is usable by your program.

```
// -----------------+---------+----------+---------+------------+-------------- //
//     Cast         |  Conv . |  Reint . |  Upcast |  Downcast  |  Type qual .  //
// -----------------+---------+----------+---------+------------+-------------- //
// implicit         |   YES   |          |   YES   |            |               //
// static_cast      |   YES   |          |   YES   |    YES     |               //
// dynamic_cast     |         |          |   YES   |    YES     |               //
// const_cast       |         |          |         |            |      YES      //
// reinterpret_cast |         |   YES    |   YES   |    YES     |               //
// -----------------+---------+----------+---------+------------+-------------- //
// legacy C cast    |   YES   |   YES    |   YES   |    YES     |      YES      //
// -----------------+---------+----------+---------+------------+-------------- //

// -----------------+--------------------+----------------------+--------------- //
//     Cast         |  Semantics check   |   Reliable at run    | tested at run  //
// -----------------+--------------------+----------------------+--------------- //
// implicit         |        YES         |        YES           |                //
// static_cast      |        YES         |                      |                //
// dynamic_cast     |        YES         |        YES           |       YES      //
// const_cast       |                    |                      |                //
// reinterpret_cast |                    |                      |                //
// -----------------+--------------------+----------------------+--------------- //
// legacy C cast    |                    |                      |                //
// -----------------+--------------------+----------------------+--------------- //
```

From C type conversion

- **Implicit conversion**: the compiler will cast for you
- **Explicit conversion**: you specify the type to cast

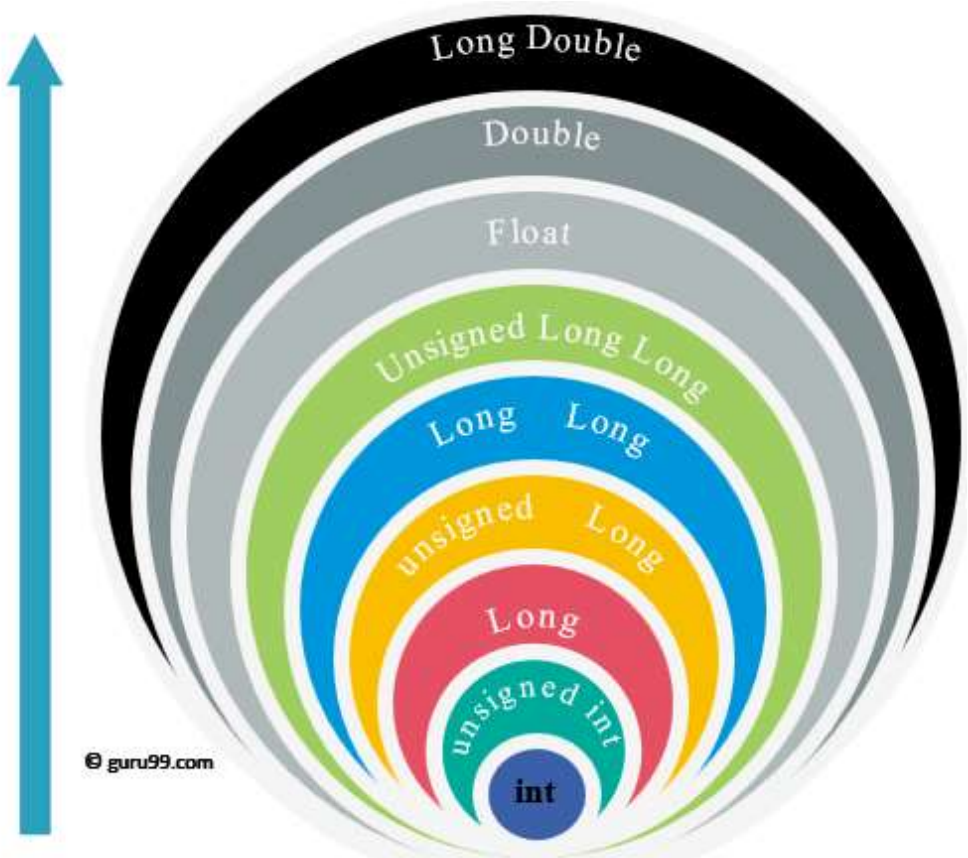
- Double and floats are bigger and more accurate than integers. They are saved differently in bits
- Need to mindful with demotion. You always want to use an explicit cast to tell the compiler to convert correctly and do not lose any precision

```
int main(void)
{
        int a = 42;

        double b = a; // implicit conversion cast
        double c = (double)a; // explicit conversion cast

        double d = a; // implicit promotion -> okay (no problem to move to a more generic t
        int e = d; // implicit demotion -> not really okay
        int f = (int)d; // explicit demotion -> okay
}
```

> conversion hierarchy



From C type reinterpretation

- Reinterpretation is about identity conversion. It doesn't reorganise the bits but interpret it in a different way.
- There are hierarchies among the accuracy of the different data types: float > (more accurate than) int > (more accurate than) > void

```
int main(void)
{
        float a = 420.042f; // reference value

        void * b = &a; // implicit reinterpretation cast
        void * c =  (void *) &a; // explicit reinterpretation cast

        void * d = &a; // implicit promotion -> okay (no problem to move to a more generic
        int * e = d; // implicit demotion -> not really okay
        int * f = (int *) d; // explicit demotion -> okay
}
```

From C type qualifier reinterpretation

- There is no bit transformation
- Qualifiers include: const, auto. They add hierarchy to the data types.
- We can easily move from a mutable type to a const type (move up), but move down can cause problems (compilers will not allow without an explicit demotion from const to non-const).

```
int main(void)
{
        int a = 42; // reference value

        int const * b = &a; // implicit type qualifier cast
        int const * c = (int const *) &a; // explicit type qualifier cast

        int const * d = &a; // implicit promotion -> okay
        int * e = d; // implicit demotion -> compiler will give error
        int * f = (int *) d; // explicit demotion -> okay
}
```

Upcast & downcast

- **Downcast**: From a generic type to a more specific type
- **Upcast**: From a more specific type to a generic type

```
class Parent {};
class Child1: public Parent {}; // Parent class is a more generic type
class Child2: public Parent {}; // Child1 & Child2 are more specific classes

int main(void)
{
        Child1 a; // reference value

        Parent * b = &a; // implicit reinterpretation cast
```

```
        Parent * c = (Parent *) &a; // explicit reinterpretation cast

        Parent * d = &a; // implicit upcast -> ok
        Child1 * e = d; // implicit downcast -> no!
        Child2 * f = (Child2 *) d; // explicit downcast _> ok but really?
}
```

1st C++ cast: static_cast

- All cast actions are conversions, but some of the conversions have interesting properties:
  - Conversion
  - Reinterpretation (identity conversion)
  - Type qualifier interpretation (a specific type of reinterpretation)
  - Downcast
  - Upcast
- `static_cast<type_to_convert_to>(expression)` . `static_cast` keyword means that the cast is checked statically (when the code is compiled). It will allow us to make simple conversions from direct values
- The effect of the cast is to `convert the value` that results from evaluating expression to the type that you specify between the angle brackets. The expression can be anything from a single variable to a complex expression involving lots of nested parentheses
- Generally, the need for explicit casts should be rare, particularly with basic types of data. If you have to include a lot of explicit conversion in your code, it's often a sign that you could choose more suitable types for your variables.

> example: simple conversion of values to avoid loss of precision

```
int main(void)
{
        int a = 42; // reference value

        double b = a; // implicit promotion -> ok
        int c = b; // implicit demotion -> no!
        int d = static_cast<int>(b); // explicit demotion -> ok

        return 0;
}
```

> example: downcast

```
class Parent {};
class Child1: public Parent {}; // Parent is more generic
class Child2: public Parent {}; // Child1 & Child2 more accurate
```

```
class Unrelated {}; // detached from the hierarchy of the inheritance tree

int main(void)
{
        Child1 a; // reference value

        Parent * b = &a; // implicit upcast -> ok
        Child1 * c = b; // implicit downcast -> no!
        Child2 * d = static_cast<Child2 *>(b); // explicit downcast -> ok

        Unrelated * e = static_cast<Unrelated *>(&a); // explicit conversion -> no!
        // static_cast will make sure that the cast will happen within an inheritance tree
        return 0;
}
```

2nd C++ cast: dynamic_cast

- Safely converts pointers and references to classes up, down, and sideways along the inheritance hierarchy.
- `dynamic cast` means that the cast is **checked dynamically (when the program is executing / at run time)**. So the dynamic cast may fail at run time, so you have to handle the potential failures within your code.
- All other casts are made during compilation and it's done in a static way.
- **You can only apply this operator to pointers and references to `polymorphic class types`**, which are class types that contain at least one virtual function. The reason is that only pointers to polymorphic class types contain the information that the `dynamic_cast<>()` operator needs to check the validity of the conversion.
- When you use plugins, your code will be grouped under a class. You can make sure that you are dealing with the right type, otherwise you output an error message.

```
#include <iostream>
#include <typeinfo> // This header defines types used related to operators typeid and dynam
#include <exception>

class Parent {public: virtual ~Parent(void) {} };
class Child1: public Parent {};
class Child2: public Parent {};

int main(void)
{
        Child1 a; // reference value
        Parent * b = &a; // implicit upcast -> ok

    // explicit downcast
```

```
    Child1 * c = dynamic_cast<Child1 *>(b);
    if (c = NULL) {
        std::cout << "Conversion is NOT okay" << std::endl;
    }
    else {
            std::cout << "Conversion is okay" << std::endl;
    }

    // explicit downcast
    try {
            Child2 & d = dynamic_cast<Child2 &>(*b); // convert to a reference
            // it will fail because it casts to another
            // the reference can't be NULL by definition, so it needs another way
            // to handle the cast failure
        std::cout << "Conversion is okay" << std::endl;
    }
    catch (std::bad_cast &bc){
            std::cout << "Conversion is NOT okay: " << bc.what() << std::endl;
            return 0;
    }
    return 0;
}
```

3rd C++ cast: reinterpret_cast

- This cast will allow you to do reinterpretation, as well as downcast and upcast. It's a very open form of casting
- Reinterpret casts are only available in C++ and are the least safe form of cast, allowing the reinterpretation of the underlying bits of a value into another type. It should not be used to cast down a class hierarchy or to remove the const or volatile qualifiers.

```
int main(void)
{
        float a = 420.042f; // reference value

        void * b = &a; // implicit promotion -> ok
        int * c = reinterpret_cast<int *>(b); // explicit demotion -> ok
        // there will be no semantics checks, as the compiler will trust you
        // they will reinterpret any address as the specified other type
        int & d = reinterpret_cast<int &>(b); // explicit demotion -> ok

        return 0;
}
```

4th C++ cast: const_cast

- It deals with the type qualifiers transformation

- Very rarely, a function deals with a `const` object, either passed as an argument or the object pointed to by `this`, and it is necessary to make it `non-cast`. This maybe because you want to pass it as an argument to another function that has a non-const parameter.

- `const_cast<type>(expression)`. The type of expression must be either `const Type` or the same as `Type`. You should not use this operator to undermine the const-ness of an object. The only situations in which you should use it are those where you are sure the const nature of the object won't be violated as a result.

- Make sure that you have a good reason to do a const_cast conversion, and not because of a design flaw

```
int main(void)
{
        int a = 42; // reference value

        int const * b = &a; // implicit promotion -> ok
        // moving from a mutable value to a const is not a problem
        int * c = b; // explicit demotion -> no!
        int * d = const_cast<int *>(b); // explicit demotion -> ok
}
```

Type cast operators

- `operator type_name` declares an implicit conversion operator. In other words, this function is called when you are attempting to (implicitly) convert an object of your type to `type_name`

```
class Foo {

public:
        Foo(Float const v) : _v(v) {}
        float getV(void) {return this->_v;}

        operator float() {return this->_v;} // cast operator
        operator int() {return static_cast<int>(this->_V);} // cast operator

private:
    float _v;
};

int main(void)
{
        Foo a(420.024f);
        float b = a; // implicit cast from Foo to float
        int c = a; // implicit cast from Foo to int

        std::cout << a.getV() << std::endl;
    std::cout << b << std::endl;
    std::cout << c << std::endl;
```

```
    return 0;
}
```

Explicit keyword

```
Class A {};
Class B {};

Class C {
public:
        C(A const & _) {return ;}
    explicit C(B const & _) {return ;} // will prevent implicit conversion of your instance
};

void f(C const & _) {
        return ;
}

int main(void)
{
        f(A()); // implicit conversion okay
        f(B()); // implicit conversion not okay, constructor is explicit
}
```

Converting between pointers to class objects

- You can implicitly convert a pointer to a derived class to a pointer to a base class, and you can do this for both direct and indirect base classes.
- A pointer to a class type can only point to objects of that type, or to objects of a derived class type

The need for virtual destructors

- If the destructors are virtual, the destructor corresponding to the object type is called.
- So if a pointer points to a BrassPlus object, the BrassPlus destructor is called.And when a BrassPlus destructor finishes, it automatically calls the base-class constructor.Thus, using virtual destructors ensures that the correct sequence of destructors is called.

resources

- dynamic_cast

Pages 11

Find a page...

Clone this wiki locally

```
https://github.com/qingqingqingli/CPP.wiki.git
```