Type-Conversion Operators

Here is the list of **type-conversions**.

1. **a = dynamic\_cast<T\*>(p)**  
   Try to convert **p** into a **T\***. It may return **0**
2. **a = dynamic\_cast<T&>(\*p)**  
   Try to convert **\*p** into a **T&**. It may throw **bad\_cast**
3. **a = static\_cast<T>(p)**  
   Convert **p** into a **T** if a **T** can be conterted into **p**'s type.
4. **a = reinterpret\_cast<T>(p)**  
   Convert **p** into a **T** represented by the same bit pattern.
5. **a =const\_cast<T>(p)**  
   Convert **p** into a **T** by adding or subtracting **const**.
6. **a = (T)v**  
   C-style cast.
7. **a = T(v)**  
   Functional cast.
8. struct Foo{};
9. struct Bar{};
10. int main(int argc, char\*\* argv)
11. {
12. Foo\* f = new Foo;
13. Bar\* b1 = f; // (1)
14. Bar\* b2 = static\_cast<Bar\*>(f); // (2)
15. Bar\* b3 = dynamic\_cast<Bar\*>(f); // (3)
16. Bar\* b4 = reinterpret\_cast<Bar\*>(f); // (4)
17. Bar\* b5 = const\_cast<Bar\*>(f); // (5)
18. return 0;
19. }

Answer is only line (4) compiles without any complain. Only **reinterpret\_cast** can be used to convert a pointer to an object to a pointer to an unrelated object type. The **dynamic\_cast** would fail at run-time, however on most compilers it will also fail to compile because there are no virtual functions in the class of the pointer being casted.

const\_cast

**const\_cast** is typically used to cast away the constness of objects. It is the only C++ style that can do this.

The syntax is:

const\_cast < type-name > (expression)

The reason for this operator is that we may have a need for a value that is constant most of the time but that can be changed occasionally. In such a case, we can declare the value as **const** and use **const\_cast** when we need to alter the value.

Here is a simple example:

#include <iostream>

#include <cstring>

using namespace std;

int main () {

string str("A123456789");

const char \*cstr = str.c\_str();

char \*nonconst\_cstr = const\_cast<char \*> (cstr) ;

\*nonconst\_cstr ='B';

cout << nonconst\_cstr << endl;

return 0;

}

Because we casted away the constness of the string, we were able to modify the string from "A123456789" to "B1234567889".

reinterpret\_cast

**reinterpret\_cast** is intended for low-level casts that yield implementation-dependent and it would not be portable.

This cast is used for reinterpreting a bit pattern. It is not guaranteed to be portable. In fact, it is best to assume that **reinterpret\_cast** is not portable at all. A typical example is an **int**-to-**pointer** to get a machine address into a program:

Register\* a = reinterpret\_cast<Register\*>(0xfa);

This example is the typical use of a **reinterpret\_cast**. We are telling the compiler that the part of memory starting with **0xfa** is to be considered a **Register**.

It converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

We can cast a pointer type to an integer type that's large enough to hold the pointer representation, but we can't cast a pointer to a smaller integer type or to a floating-point type. The format in which this integer value represents a pointer is platform-specific. We can't cast a function pointer to a data pointer or vice versa.

The syntax is:

reinterpret\_cast < type-name > (expression)

Here is an example code:

#include <iostream>

using namespace std;

struct data {

short a;

short b;

};

int main () {

long value = 0xA2345678;

data\* pdata = reinterpret\_cast<data\*> (&value;);

cout << pdata->a << endl;

return 0;

}

Output on my machine: 22136 which is 2 bytes of value.

Another example might be:

class A {};

class B {};

int main()

{

A \* pA = new A;

B \* pB = reinterpret\_cast<B\*>(pA);

}

In the following code (b1-b5), only **Bar\* b5 = reinterpret\_cast<Bar\*>** compiles.

struct Foo {};

struct Bar {};

int main()

{

Foo\* f = new Foo;

Bar\* b1 = f;

Bar\* b2 = static\_cast<Bar\*>(f);

Bar\* b3 = dynamic\_cast<Bar\*>(f);

Bar\* b4 = const\_cast<Bar\*>(f);

Bar\* b5 = reinterpret\_cast<Bar\*>(f);

return 0;

}

Only **reinterpret\_cast** can be used to convert a pointer to an object to a pointer to an unrelated object type. The dynamic cast would fail at run-time, however on most compilers it will also fail to compile because there are no virtual functions in the class of the pointer being casted.

Pointer comparison

Normally, the pointer comparison between different types are undefined. Here is the example of using **reinterpret\_cast** for pointer comparison.

#include <iostream>

class A{};

class B{};

class C:public A, public B{};

int main()

{

C d;

A \*a = &d;

B \*b = &d;

bool bool1 = reinterpret\_cast<char\*>(a) == reinterpret\_cast<char\*>(&d;);

bool bool2 = b == &d;

bool bool3 = reinterpret\_cast<char\*>(a) == reinterpret\_cast<char\*>(b);

std::cout << bool1 << bool2 << bool3 << std::endl; // 110

return 0;

}

Base pointer (**a** and **b**) and the address of the derived object **&d;** are equal but **a** and **b** are different as expected.

static\_cast

**static\_cast** can be used to force implicit conversions such as **non-const** object to **const**, **int** to **double**. It can be also be used to perform the reverse of many conversions such as **void\*** pointers to **typed** pointers, **base** pointers to **derived** pointers. But it cannot cast from **const** to **non-const** object. This can only be done by **const\_cast** operator.

The syntax is:

static\_cast < type-name > (expression)

It's valid only if **type\_name** can be converted implicitly to the same type that **expression** has, or vise versa. Otherwise, the type cast is an error.

class Base {};

class Derived : public Base {};

class UnrelatedClass {};

int main()

{

Base base;

Derived derived;

// #1: valid upcast

Base \*pBase = static\_cast<Base \*>(&derived;);

// #2: valid downcast

Derived \*pDerived = static\_cast<Derived \*> (&base;);

// #3: invalid, between unrelated classes

UnrelatedClass \*pUnrelated

= static\_cast<UnrelatedClass \*> (&derived;);

}

In the example, the conversion from **Base** to **Derived** and **Derived** to **Base** are valid, but a conversion from **Derived** to **UnrelatedClass** is disallowed.

The #1 conversion here is valid because an upcast can be done explicitly. The #2 conversion, from a base-class pointer to a derived-class pointer, can't be done without an explicit type conversion. But because the type cast in the other direction can be made without a type cast, it's valid to use **static\_cast** for a downcast. However, **pDerivded** would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

As we saw in the example, **static\_cast** can perform conversions between pointers to **related** classes, not only from the derived class to its base, but also from a base class to its derived.

This ensures that at least the classes are **compatible** if the proper object is converted, but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type.

Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, the overhead of the type-safety checks of **dynamic\_cast** is avoided.

dynamic\_cast

**dynamic\_cast** is used to perform **safe downcasting**, i.e., to determine whether an object is of a particular type in an inheritance hierarchy. It is the only cast that may have a significant runtime cost.

Look at the [dynamic\_cast](https://www.bogotobogo.com/cplusplus/dynamic_cast.php) of C++ Tutorial - dynamic\_cast.

Casting - pros and cons

This is from [Google C++ Style Guide](http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Casting).

C++ introduced a different cast system from C that distinguishes the types of cast operations. Use C++ casts like **static\_cast<>()**. Do not use other cast formats like **int y = (int)x;** or **int y = int(x);**.

1. **Pros**  
   The problem with C casts is the ambiguity of the operation; sometimes we are doing a conversion (e.g., **(int)3.5**) and sometimes we are doing a cast (e.g., **(int)"hello"**); C++ casts avoid this. Additionally C++ casts are more visible when searching for them.
2. **Cons**  
   The syntax is nasty.
3. **Decision**  
   Do not use C-style casts. Instead, use these C++-style casts.
   1. Use **static\_cast** as the equivalent of a C-style cast that does value conversion, or when we need to explicitly up-cast a pointer from a class to its superclass.
   2. Use **const\_cast** to remove the const qualifier.
   3. Use **reinterpret\_cast** to do unsafe conversions of pointer types to and from integer and other pointer types. Use this only if we know what we are doing and we understand the aliasing issues.
   4. For **dynamic\_cast**, visit [RTTI - pros and cons](http://www.bogotobogo.com/cplusplus/dynamic_cast.php#rtti_pros_cons).( **RTTI** allows a programmer to query the C++ class of an object at run time.)