C++ Typecasting

Typecasting is the concept of converting the value of one type into another type. For example, you might have a float that you need to use in a function that requires an integer.

Implicit conversion

Almost every compiler makes use of what is called automatic typecasting. It automatically converts one type into another type. If the compiler converts a type it will normally give a warning. For example this warning: conversion from 'double' to 'int', possible loss of data.
The problem with this is, that you get a warning (normally you want to compile without warnings and errors) and you are not in control. With control we mean, you did not decide to convert to another type, the compiler did. Also the possible loss of data could be unwanted.

Explicit conversion

The C and C++ languages have ways to give you back control. This can be done with what is called an explicit conversion. Sure you may still lose data, but you decide when to convert to another type and you don't get any compiler warnings.
Let's take a look at an example that uses implicit and explicit conversion:

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

int main() {
	int a;
	double b=2.55;

	a = b;
	cout << a << endl;

	a = (int)b;
	cout << a << endl;

	a = int(b);
	cout << a << endl;

}
```

Note: the output of all cout statements is 2.
The first conversion is an implicit conversion (the compiler decides.) As explained before, the compiler should give a warning.
The second conversion is an explicit typecast, in this case the C style explicit typecast.
The third conversion is also explicit typecast, in this case the C++ style explicit typecast.

Four typecast operators

The C++ language has four typecast operators:

- static_cast
- reinterpret_cast
- const_cast
- dynamic_cast

Static_cast

Automatic conversions are common in every C++ program. You have:

- Standard conversion. For instance: from short to int or from int to float.
- User defined conversions (Class conversions.)
- Conversion from derived class to base class.

The static_cast can be used for all these types of conversion. Take a look at an example:

```
int a = 5;
int b = 2;
double out;
// typecast a to double
out = static_cast<double>(a)/b;
```

It may take some time to get used to the notation of the typecast statement. (The rumour goes that Bjarne Stroustrup made it difficult on purpose, to discourage the use of typecasting.) Between the angle brackets you place to which type the object should be casted. Between the parentheses you place the object that is casted.
It is not possible to use static_cast on const objects to non-const objects. For this you have to use const_cast. (Further down we take a look at const_cast.)
If an automatic conversion is valid (from enum to int for instance) then you can use static_cast to do the opposite (from int to enum.)
For instance:

```
enum my_numbers { a=10, c=100, e=1000 };
const my_numbers b = static_cast<my_numbers> (50);
const my_numbers d = static_cast<my_numbers> (500);
```

Note: We add some new values (b and d). These are type-cast from int to enum.

Reinterpret_cast

The reinterpret_cast is used for casts that are not safe:

- Between integers and pointers
- Between pointers and pointers
- Between function-pointers and function-pointers

For instance the typecast from an integer to a character pointer:

```
char *ptr_my = reinterpret_cast<char *>(0xb0000);
```

Note: the example above uses a fixed memory location.

If we use the reinterpret_cast on a null-pointer then we get a null-pointer of the asked type:

```
char *ptr_my = 0;
int *ptr_my_second = reinterpret_cast<int *>(ptr_my);
```

The reinterpret_cast is almost as dangerous as an "old fashion" cast. The only guaranty that you get is that if you cast an object back to the original data-type (before the first cast) then the original value is also restored (of course only if the data-type was big enough to hold the value.)
The only difference with an old fashion cast is that const is respected. This means that a reinterpret_cast can not be used to cast a const object to non-const object. For instance:

```
char *const MY = 0;
// This is not valid because MY is a const!!
int *ptr_my = reinterpret_cast<int *>( MY);
```

Const_cast

The only way to cast away the const properties of an object is to use const_cast. Take a look at an example:

```
void a(Person* b);
int main() {
	const Person *ptr_my = new Person("Joe");
	a( const_cast<Person *>(ptr_my) );
}
```

The use of const_cast on an object doesn't guarantee that the object can be used (after the const is cast away.) Because it is possible that the const-objects are put in read-only memory by the program.
The const_cast can not be used to cast to other data-types, as it is possible with the other cast functions. Take a look at the next example:

```
int a;
const char *ptr_my = "Hello";
a  = const_cast<int *>(ptr_my);
a = reinterpret_cast<const char*>(ptr_my);
a = reinterpret_cast<int *>(const_cast<char *>(ptr_my) );
```

Note: casting from const char * to int * isn't very good (not to say a very dirty trick.) Normally you won't do this.

The first statement (const_cast) will give an error, because the const_cast can't convert the type. The second statement (reinterpret_cast) will also give an error, because the reinterpret_cast can't cast the const away. The third statement will work (mind the note. It is a dirty trick, better not use it.)

Runtime Type Information (RTTI)

Runtime Type Information (RTTI) is the concept of determining the type of any variable during execution (runtime.) The RTTI mechanism contains:

- The operator dynamic_cast
- The operator typeid
- The struct type_info

RTTI can only be used with polymorphic types. This means that with each class you make, you must have at least one virtual function (either directly or through inheritance.)
Compatibility note: On some compilers you have to enable support of RTTI to keep track of dynamic types. So to make use of dynamic_cast (see next section) you have to enable this feature. See you compiler documentation for more detail.

Dynamic_cast

The dynamic_cast can only be used with pointers and references to objects. It makes sure that the result of the type conversion is valid and complete object of the requested class. This is way a dynamic_cast will always be successful if we use it to cast a class to one of its base classes. Take a look at the example:

```
class Base_Class { };
class Derived_Class: public Base_Class { };
Base_Class a; Base_Class * ptr_a;
Derived_Class b; Derived_Class * ptr_b;
ptr_a = dynamic_cast<Base_Class *>(&b);
ptr_b = dynamic_cast<Derived_Class *>(&a);
```

The first dynamic_cast statement will work because we cast from derived to base. The second dynamic_cast statement will produce a compilation error because base to derived conversion is not allowed with dynamic_cast unless the base class is polymorphic.
If a class is polymorphic then dynamic_cast will perform a special check during execution. This check ensures that the expression is a valid and complete object of the requested class. Take a look at the example:

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

class Base_Class { virtual void dummy() {} };
class Derived_Class: public Base_Class { int a; };

int main () {
  try {
       Base_Class * ptr_a = new Derived_Class;
```

```
        Base_Class * ptr_b = new Base_Class;
        Derived_Class * ptr_c;

        ptr_c = dynamic_cast< Derived_Class *>(ptr_a);
        if (ptr_c ==0) cout << "Null pointer on first type-cast" << endl;

        ptr_c = dynamic_cast< Derived_Class *>(ptr_b);
        if (ptr_c ==0) cout << "Null pointer on second type-cast" << endl;

        } catch (exception& my_ex) {cout << "Exception: " << my_ex.what();}
  return 0;
}
```

In the example we perform two dynamic_casts from pointer objects of type Base_Class* (namely ptr_a and ptr_b) to a pointer object of type Derived_Class*.
If everything goes well then the first one should be successful and the second one will fail. The pointers ptr_a and ptr_b are both of the type Base_Class. The pointer ptr_a points to an object of the type Derived_Class. The pointer ptr_b points to an object of the type Base_Class. So when the dynamic type cast is performed then ptr_a is pointing to a full object of class Derived_Class, but the pointer ptr_b points to an object of class Base_Class. This object is an incomplete object of class Derived_Class; thus this cast will fail!
Because this dynamic_cast fails a null pointer is returned to indicate a failure. When a reference type is converted with dynamic_cast and the conversion fails then there will be an exception thrown out instead of the null pointer. The exception will be of the type bad_cast.
With dynamic_cast it is also possible to cast null pointers even between the pointers of unrelated classes. Dynamic_cast can cast pointers of any type to void pointer(void*).

Typeid and typ_info

If a class hierarchy is used then the programmer doesn't have to worry (in most cases) about the data-type of a pointer or reference, because the polymorphic mechanism takes care of it. In some cases the programmer wants to know if an object of a derived class is used. Then the programmer can make use of dynamic_cast. (If the dynamic cast is successful, then the pointer will point to an object of a derived class or to a class that is derived from that derived class.) But there are circumstances that the programmer (not often) wants to know the prizes data-type. Then the programmer can use the typeid operator.

The typeid operator can be used with:

- Variables
- Expressions
- Data-types

Take a look at the typeid example:

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

int main()    {
       int *a, b;
       a = 0; b = 0;
       if (typeid(a) != typeid(b)){
              cout << "a and b are of different types:\n";
              cout << "a is: " << typeid(a).name() << '\n';
              cout << "b is: " << typeid(b).name() << '\n';
       }
       return 0;
}
```

Note: the extra header file typeinfo.

The result of a typeid is a const type_info&. The class type_info is part of the standard C++ library and contains information about data-types. (This information can be different. It all depends on how it is implemented.)

A bad_typeid exception is thrown by typeid, if the type that is evaluated by typeid is a pointer that is preceded by a dereference operator and that pointer has a null value.
That is all for this tutorial. In the next C++ programming tutorial we will take a look at pre-processors.

Динамическая информация о типе.

- *type_info*:
  Stores information about a type. An object of this class is returned by the *typeid* operator (as a const-qualified lvalue). Although its actual dynamic type may be of a derived class. It can be used to compare two types or to retrieve information identifying a type.
  typeid can be applied to any type or any expression that has a type.
  If applied to a reference type (lvalue), the *type_info* returned identifies the referenced type.
  Any *const* or *volatile* qualified type is identified as its unqualified equivalent.
  A *typedef* type is considered the same as its aliased type.
  When *typeid* is applied to a reference or dereferenced pointer to an object of a polymorphic class type (a class declaring or inheriting a virtual function), it considers its dynamic type (i.e., the type of the most derived object). This requires the RTTI (Run-time type information) to be available.
  When *typeid* is applied to a dereferenced null pointer, a *bad_typeid* exception is thrown.
  The lifetime of the object returned by *typeid* extends to the end of the program.
- *bad_typeid*:
  Type of the exceptions thrown by *typeid* when applied on a pointer to a polymorphic type which has a null pointer value.
  Its member what returns a null-terminated character sequence identifying the exception.
- *bad_cast*:
  Type of the exceptions thrown by *dynamic_cast* when it fails the run-time check performed on references to polymorphic class types.
  The run-time check fails if the object would be an incomplete object of the destination type.
  Its member what returns a null-terminated character sequence identifying the exception.
  Some functions in the standard library may also throw this exception to signal a type-casting error.

1. http://www.codingunit.com/c-tutorial-typecasting-part-1
2. http://www.codingunit.com/cplusplus-tutorial-typecasting-part-2-rtti-dynamic_cast-typeid-and-type_info
3. http://www.cplusplus.com/reference/typeinfo/