Advanced Concepts:

22. Type Casting

Converting an expression of a given type into another type is known as *type-casting*. We have already seen some ways to type cast:

Implicit conversion

Implicit conversions do not require any operator. They are automatically performed when a value is copied to a compatible type. For example:

```
short a=2000;
int b;
b=a;
```

Here, the value of a has been promoted from `short` to `int` and we have not had to specify any type-casting operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow conversions such as the conversions between numerical types (`short` to `int`, `int` to `float`, `double` to `int`...), to or from `bool`, and some pointer conversions. Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This can be avoided with an explicit conversion.

Implicit conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions. For example:

```
class A {};
class B { public: B (A a) {} };

A a;
B b=a;
```

Here, a implicit conversion happened between objects of `class A` and `class B`, because B has a constructor that takes an object of class A as parameter. Therefore implicit conversions from A to B are allowed.

Explicit conversion

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion: functional and c-like casting:

```
short a=2000;
int b;
b = (int) a;    // c-like cast notation
b = int (a);    // functional notation
```

The functionality of these explicit conversion operators is enough for most needs with fundamental data types. However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause runtime errors. For example, the following code is syntactically correct:

```
// class type-casting
#include <iostream>
using namespace std;

class CDummy {
    float i,j;
};

class CAddition {
        int x,y;
  public:
        CAddition (int a, int b)
{ x=a; y=b; }
        int result() { return
x+y;}
};

int main () {
  CDummy d;
  CAddition * padd;
  padd = (CAddition*) &d;
  cout << padd->result();
  return 0;
}
```

The program declares a pointer to `CAddition`, but then it assigns to it a reference to an object of another incompatible type using explicit type-casting:

```
padd = (CAddition*) &d;
```

Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member `result` will produce either a run-time error or an unexpected result.

In order to control these types of conversions between classes, we have four specific casting operators: `dynamic_cast`, `reinterpret_cast`, `static_cast` and const_cast. Their format is to follow the new type enclosed between angle-brackets (<>) and immediately after, the expression to be converted between parentheses.

```
dynamic_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
static_cast <new_type> (expression)
const_cast <new_type> (expression)
```

The traditional type-casting equivalents to these expressions would be:

```
(new_type) expression
new_type (expression)
```

but each one with its own special characteristics:

dynamic_cast

dynamic_cast can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

Therefore, dynamic_cast is always successful when we cast a class to one of its base classes:

```
class CBase { };
class CDerived: public CBase { };

CBase b; CBase* pb;
CDerived d; CDerived* pd;

pb = dynamic_cast<CBase*>(&d);
   // ok: derived-to-base
pd = dynamic_cast<CDerived*>(&b);
   // wrong: base-to-derived
```

The second conversion in this piece of code would produce a compilation error since base-to-derived conversions are not allowed with dynamic_cast unless the base class is polymorphic.

When a class is polymorphic, dynamic_cast performs a special checking during runtime to ensure that the expression yields a valid complete object of the requested class:

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

class CBase { virtual void dummy()
{} };
class CDerived: public CBase { int
a; };
```

```
int main () {
  try {
    CBase * pba = new CDerived;
    CBase * pbb = new CBase;
    CDerived * pd;

    pd =
dynamic_cast<CDerived*>(pba);
    if (pd==0) cout << "Null
pointer on first type-cast" <<
endl;

    pd =
dynamic_cast<CDerived*>(pbb);
    if (pd==0) cout << "Null
pointer on second type-cast" <<
endl;

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

```
Null pointer on second type-cast
```

Compatibility note: dynamic_cast requires the Run-Time Type Information (RTTI) to keep track of dynamic types. Some compilers support this feature as an option which is disabled by default. This must be enabled for runtime type checking using dynamic_cast to work properly.

The code tries to perform two dynamic casts from pointer objects of type CBase* (pba and pbb) to a pointer object of type CDerived*, but only the first one is successful. Notice their respective initializations:

```
CBase * pba = new CDerived;
CBase * pbb = new CBase;
```

Even though both are pointers of type CBase*, pba points to an object of type CDerived, while pbb points to an object of type CBase. Thus, when their respective type-castings are performed using dynamic_cast, pba is pointing to a full object of class CDerived, whereas pbb is pointing to an object of class CBase, which is an incomplete object of class CDerived.

When dynamic_cast cannot cast a pointer because it is not a complete object of the required class -as in the second conversion in the previous example- it returns a null pointer to indicate the failure. If dynamic_cast is used to convert to a reference type and the conversion is not possible, an exception of type bad_alloc is thrown instead.

dynamic_cast can also cast null pointers even between pointers to unrelated classes, and can also cast pointers of any type to void pointers (void*).

static_cast

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.

```
class CBase {};
class CDerived: public CBase {};
CBase * a = new CBase;
CDerived * b = static_cast<CDerived*>(a);
```

This would be valid, although b would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

static_cast can also be used to perform any other non-pointer conversion that could also be performed implicitly, like for example standard conversion between fundamental types:

```
double d=3.14159265;
int i = static_cast<int>(d);
```

Or any conversion between classes with explicit constructors or operator functions as described in "implicit conversions" above.

reinterpret_cast

reinterpret_cast 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.

It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it, is granted to be able to be cast back to a valid pointer.

The conversions that can be performed by reinterpret_cast but not by static_cast have no specific uses in C++ are low-level operations, whose interpretation results in code which is generally system-specific, and thus non-portable. For example:

```
class A {};
class B {};
```

```
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

This is valid C++ code, although it does not make much sense, since now we have a pointer that points to an object of an incompatible class, and thus dereferencing it is unsafe.

const_cast

This type of casting manipulates the constness of an object, either to be set or to be removed. For example, in order to pass a const argument to a function that expects a non-constant parameter:

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

void print (char * str)
{
  cout << str << endl;
}

int main () {
  const char * c = "sample text";
  print ( const_cast<char *>
(c) );
  return 0;
}
```

```
sample text
```

typeid

`typeid` allows to check the type of an expression:

```
typeid (expression)
```

This operator returns a reference to a constant object of type `type_info` that is defined in the standard header file `<typeinfo>`. This returned value can be compared with another one using operators `==` and `!=` or can serve to obtain a null-terminated character sequence representing the data type or class name by using its `name()` member.

```
// typeid
#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;
}
```

```
a and b are of different types:
a is: int *
b is: int
```

When `typeid` is applied to classes `typeid` uses the RTTI to keep track of the type of dynamic objects. When typeid is applied to an expression whose type is a polymorphic class, the result is the type of the most derived complete object:

```
// typeid, polymorphic class
#include <iostream>
#include <typeinfo>
#include <exception>
using namespace std;

class CBase { virtual void
f(){} };
class CDerived : public CBase {};

int main () {
  try {
    CBase* a = new CBase;
    CBase* b = new CDerived;
    cout << "a is: " <<
typeid(a).name() << '\n';
    cout << "b is: " <<
typeid(b).name() << '\n';
    cout << "*a is: " <<
typeid(*a).name() << '\n';
    cout << "*b is: " <<
typeid(*b).name() << '\n';
  } catch (exception& e) { cout <<
"Exception: " << e.what() <<
endl; }
  return 0;
}
```

```
a is: class CBase *
b is: class CBase *
*a is: class CBase
*b is: class CDerived
```

Notice how the type that `typeid` considers for pointers is the pointer type itself (both `a` and `b` are of type `class CBase *`). However, when `typeid` is applied to objects (like `*a` and `*b`) `typeid` yields their dynamic type (i.e. the type of their most derived complete object).

If the type `typeid` evaluates is a pointer preceded by the dereference operator (`*`), and this pointer has a null value, `typeid` throws a `bad_typeid` exception.