Type conversions
- Implicit conversion
- Implicit conversions with classes
- Keyword explicit
- Type casting
- dynamic_cast
- static_cast
- reinterpret_cast
- const_cast
- typeid

Type conversions

Implicit conversion

Implicit conversions are automatically performed when a value is copied to a compatible type. For example:

当值被复制到兼容类型时，隐式转换将自动执行。例如:

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

Here, the value of `a` is promoted from `short` to `int` without the need of any explicit operator. This is known as a *standard conversion*. Standard conversions affect fundamental data types, and allow the conversions between numerical types (`short` to `int`, `int` to `float`, `double` to `int`...), to or from `bool`, and some pointer conversions.

在这里，a的值从short提升为int，而不需要任何显式操作符。这就是所谓的标准转换。标准转换影响基本数据类型，并允许数值类型之间的转换(short到int、int到float、double到int...)、到bool或从bool类型之间的转换，以及一些指针转换。

Converting to `int` from some smaller integer type, or to `double` from `float` is known as *promotion*, and is guaranteed to produce the exact same value in the destination type. Other conversions between arithmetic types may not always be able to represent the same value exactly:

从某个较小的整数类型转换为int，或者从float类型转换为double，称为提升，并保证在目标类型中生成完全相同的值。算术类型之间的其他转换可能并不总是能够精确地表示相同的值:

- If a negative integer value is converted to an unsigned type, the resulting value corresponds to its 2's complement bitwise representation (i.e., `-1` becomes the largest value representable by the type, `-2` the second largest, ...).

如果将负整型转换为无符号类型，则得到的值对应于其二进制补码位表示(即-1成为该类型可表示的最大值，-2成为第二大值，...)。

- The conversions from/to `bool` consider `false` equivalent to *zero* (for numeric types) and to *null pointer* (for pointer types); `true` is equivalent to all other values and is converted to the equivalent of `1`.

  从/到bool的转换认为false等价于零(对于数字类型)和空指针(对于指针类型);True等价于所有其他值，并被转换为等价于1的值

- If the conversion is from a floating-point type to an integer type, the value is truncated (the decimal part is removed). If the result lies outside the range of representable values by the type, the conversion causes *undefined behavior*.

  如果从浮点类型转换为整数类型，则该值将被截断(小数部分被删除)。如果结果超出了类型可表示值的范围，则转换将导致未定义的行为。

- Otherwise, if the conversion is between numeric types of the same kind (integer-to-integer or floating-to-floating), the conversion is valid, but the value is *implementation-specific* (and may not be portable).

  否则，如果转换是在相同类型的数字类型(整数到整数或浮点到浮点)之间进行的，则转换是有效的，但值是特定于实现的(并且可能不可移植)。

Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This warning can be avoided with an explicit conversion.

其中一些转换可能意味着精度的损失，编译器可以发出警告。可以通过显式转换来避免此警告。

For non-fundamental types, arrays and functions implicitly convert to pointers, and pointers in general allow the following conversions:

对于非基本类型，数组和函数隐式转换为指针，指针通常允许以下转换:

- *Null pointers* can be converted to pointers of any type

  空指针可以转换为任何类型的指针

- Pointers to any type can be converted to `void` pointers.

  指向任何类型的指针都可以转换为空指针。

- Pointer *upcast*: pointers to a derived class can be converted to a pointer of an *accessible* and *unambiguous* base class, without modifying its `const` or `volatile` qualification.

  指针上转换:指向派生类的指针可以转换为可访问且无二义性的基类的指针，而无需修改其const或volatile限定。

Implicit conversions with classes

In the world of classes, implicit conversions can be controlled by means of three member functions:

在类的世界里，隐式转换可以通过三个成员函数来控制:

- **Single-argument constructors:** allow implicit conversion from a particular type to initialize an object.

  单参数构造函数:允许从特定类型隐式转换来初始化对象。
- **Assignment operator:** allow implicit conversion from a particular type on assignments.

  赋值操作符:允许在赋值时从特定类型隐式转换。
- **Type-cast operator:** allow implicit conversion to a particular type.

  类型强制转换操作符:允许隐式转换到特定类型。

For example:

```
// implicit conversion of classes:
#include <iostream>
using namespace std;

class A {};

class B {
public:
  // conversion from A (constructor):
  B (const A& x) {}
  // conversion from A (assignment):
  B& operator= (const A& x) {return *this;}
  // conversion to A (type-cast operator)
  operator A() {return A();}
};

int main ()
{
  A foo;
  B bar = foo;     // calls constructor
  bar = foo;       // calls assignment
  foo = bar;       // calls type-cast operator
  return 0;
}
```

The type-cast operator uses a particular syntax: it uses the `operator` keyword followed by the destination type and an empty set of parentheses. Notice that the return type is the destination type and thus is not specified before the `operator` keyword.

类型强制转换操作符使用一种特殊的语法:它使用操作符关键字，后面跟着目标类型和一组空括号。注意，返回类型是目标类型，因此没有在operator关键字之前指定。

Keyword explicit

On a function call, C++ allows one implicit conversion to happen for each argument. This may be somewhat problematic for classes, because it is not always what is intended. For example, if we add the following function to the last example:

在函数调用中，c++允许对每个参数进行隐式转换。这对类来说可能有些问题，因为它并不总是预期的结果。例如，如果我们在最后一个例子中添加以下函数:

```
void fn (B arg) {}
```

This function takes an argument of type `B`, but it could as well be called with an object of type `A` as argument:

这个函数接受类型B的实参，但也可以用类型A的对象作为实参来调用它:

```
fn (foo);
```

This may or may not be what was intended. But, in any case, it can be prevented by marking the affected constructor with the `explicit` keyword:

这可能是，也可能不是我们的本意。但是，在任何情况下，它都可以通过显式关键字标记受影响的构造函数来避免:

```
// explicit:
#include <iostream>
using namespace std;

class A {};

class B {
public:
  explicit B (const A& x) {}
  B& operator= (const A& x) {return *this;}
  operator A() {return A();}
};

void fn (B x) {}

int main ()
{
  A foo;
  B bar (foo);
  bar = foo;
  foo = bar;

//  fn (foo);  // not allowed for explicit ctor.
  fn (bar);

  return 0;
```

```
}
```

Additionally, constructors marked with `explicit` cannot be called with the assignment-like syntax; In the above example, `bar` could not have been constructed with:

另外，用explicit标记的构造函数不能用类似赋值的语法调用;在上面的例子中，bar不能被构造为:

```
B bar = foo;
```

Type-cast member functions (those described in the previous section) can also be specified as `explicit`. This prevents implicit conversions in the same way as `explicit`-specified constructors do for the destination type.

类型转换成员函数(上一节中描述的)也可以显式指定。这可以防止隐式转换，就像显式指定的构造函数对目标类型所做的一样。

Type casting

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion, known in C++ as *type-casting*. There exist two main syntaxes for generic type-casting: *functional* and *c-like*:

c++是一种强类型语言。许多转换，特别是那些意味着对值有不同解释的转换，都需要显式转换，在c++中称为类型转换。泛型类型强制转换主要有两种语法:函数式和类c语法:

```
double x = 10.3;
int y;
y = int (x);    // functional notation
y = (int) x;    // c-like cast notation
```

The functionality of these generic forms of type-casting 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 compiles without errors:

这些类型转换的通用形式的功能足以满足大多数基本数据类型的需求。然而，这些操作符可以不加区别地应用于类和指向类的指针，这可能导致代码——尽管语法正确——但可能导致运行时错误。例如，下面的代码编译时没有错误:

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

class Dummy {
    double i,j;
};

class Addition {
    int x,y;
```

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

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

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

该程序声明了一个指向Addition的指针，但随后通过显式类型转换将另一个不相关类型对象的引用赋给它:

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

Unrestricted 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 some other unexpected results.

不受限制的显式类型转换允许将任何指针转换为任何其他指针类型，而不依赖于它们所指向的类型。对成员result的后续调用将产生运行时错误或其他一些意外结果。

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.

为了控制类之间的这些转换类型，我们有四个特定的强制转换操作符:dynamic_cast、reinterpret_cast、static_cast和const_cast。它们的格式是紧跟在尖括号(<>)之间的新类型之后，紧接在要在括号之间转换的表达式之后。

`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) expressionnew_type (expression)`
but each one with its own special characteristics:

dynamic_cast

`dynamic_cast` can only be used with pointers and references to classes (or with `void*`). Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type.

Dynamic_cast只能与类的指针和引用一起使用(或与void*一起使用)。它的目的是确保类型转换的结果指向目标指针类型的有效完整对象。

This naturally includes *pointer upcast* (converting from pointer-to-derived to pointer-to-base), in the same way as allowed as an *implicit conversion*.

这自然包括指针上转换(从指针到派生指针到指针到基指针的转换)，其方式与隐式转换相同。

But `dynamic_cast` can also *downcast* (convert from pointer-to-base to pointer-to-derived) polymorphic classes (those with virtual members) if -and only if- the pointed object is a valid complete object of the target type. For example:

但是dynamic_cast也可以向下强制转换(从指针到基类转换为指针到派生类)多态类(具有虚成员的类)，当且仅当所指向的对象是目标类型的有效完整对象时。例如:

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

class Base { virtual void dummy() {} };
class Derived: public Base { int a; };

int main () {
  try {
    Base * pba = new Derived;
    Base * pbb = new Base;
    Derived * pd;

    //补充：这个是执行不了的
    pd = dynamic_cast<Derived*>(pba);
    if (pd==0) cout << "Null pointer on first type-cast.\n";

    pd = dynamic_cast<Derived*>(pbb);
    if (pd==0) cout << "Null pointer on second type-cast.\n";

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

Compatibility note: This type of `dynamic_cast` requires *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 needs to be enabled for runtime type checking using `dynamic_cast` to work properly with these types.

兼容性注意:这种类型的dynamic_cast需要运行时类型信息(RTTI)来跟踪动态类型。一些编译器将此特性作为默认禁用的选项来支持。需要使用dynamic_cast启用运行时类型检查，才能正确地处理这些类型。

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

上面的代码尝试执行两个从Base*类型的指针对象(pba和pbb)到Derived*类型的指针对象的动态强制转换，但只有第一个成功。注意它们各自的初始化:

```
Base * pba = new Derived;
Base * pbb = new Base;
```

Even though both are pointers of type `Base*`, `pba` actually points to an object of type `Derived`, while `pbb` points to an object of type `Base`. Therefore, when their respective type-casts are performed using `dynamic_cast`, `pba` is pointing to a full object of class `Derived`, whereas `pbb` is pointing to an object of class `Base`, which is an incomplete object of class `Derived`.

尽管两者都是Base*类型的指针，但pba实际上指向的是Derived类型的对象，而pbb指向的是Base类型的对象。因此，当使用dynamic_cast执行它们各自的类型强制转换时，pba指向的是Derived类的完整对象，而pbb指向的是Base类的对象，后者是Derived类的不完整对象。

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_cast` is thrown instead.

当dynamic_cast不能强制转换指针，因为它不是必需类的完整对象时(如上例中的第二个转换)，它返回一个空指针以指示失败。如果使用dynamic_cast转换为引用类型且转换不可能，则会抛出bad_cast类型的异常。

`dynamic_cast` can also perform the other implicit casts allowed on pointers: casting null pointers between pointers types (even between unrelated classes), and casting any pointer of any type to a `void*` pointer.

Dynamic_cast还可以对指针执行其他允许的隐式强制转换:在指针类型之间强制转换空指针(甚至在不相关的类之间)，并将任意类型的指针强制转换为void*指针。

static_cast

`static_cast` can perform conversions between pointers to related classes, not only *upcasts* (from pointer-to-derived to pointer-to-base), but also *downcasts* (from pointer-to-base to pointer-to-derived). No checks are performed during runtime to guarantee that 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, it does not incur the overhead of the type-safety checks of `dynamic_cast`.

static_cast可以执行指向相关类的指针之间的转换，不仅可以执行向上转换(从指针到派生类到指针到基类)，还可以执行向下转换(从指针到基类到指针到派生类)。在运行时不执行任何检查，以确保正在转换的对象实际上是目标类型的完整对象。因此，要由程序员来确保转换是安全的。另一方面，它不会引起dynamic_cast的类型安全检查的开销。

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

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

这将是有效的代码，尽管b将指向一个不完整的类对象，如果解引用可能会导致运行时错误。

Therefore, `static_cast` is able to perform with pointers to classes not only the conversions allowed implicitly, but also their opposite conversions.

因此，static_cast不仅可以对指向类的指针执行隐式允许的转换，还可以执行相反的转换。

`static_cast` is also able to perform all conversions allowed implicitly (not only those with pointers to classes), and is also able to perform the opposite of these. It can:

Static_cast还能够执行隐式允许的所有转换(不仅是那些带有指向类的指针的转换)，也能够执行与之相反的转换。它可以:

- Convert from `void*` to any pointer type. In this case, it guarantees that if the `void*` value was obtained by converting from that same pointer type, the resulting pointer value is the same.

  从void*转换为任何指针类型。在这种情况下，它保证如果void*值是从相同的指针类型转换而来的，则得到的指针值是相同的。

- Convert integers, floating-point values and enum types to enum types.

  将整数、浮点值和枚举类型转换为枚举类型。

Additionally, `static_cast` can also perform the following:

此外，static_cast还可以执行以下操作:

- Explicitly call a single-argument constructor or a conversion operator.

  显式调用单参数构造函数或转换操作符。

- Convert to *rvalue references*.

  转换为右值引用。

- Convert `enum class` values into integers or floating-point values.

  将枚举类值转换为整数或浮点值。

- Convert any type to `void`, evaluating and discarding the value.

  将任何类型转换为void，求值并丢弃该值。

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.

Reinterpret_cast将任何指针类型转换为任何其他指针类型，即使是不相关的类。操作结果是值从一个指针到另一个指针的简单二进制拷贝。所有指针转换都是允许的:既不检查指针指向的内容，也不检查指针类型本身。

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 (such as `intptr_t`), is guaranteed to be able to be cast back to a valid pointer.

它还可以强制转换指向或来自整型类型的指针。这个整数值表示指针的格式是特定于平台的。唯一的保证是，将指针转换为大到足以完全包含它的整数类型(如intptr_t)的指针，保证能够被转换回有效的指针。

The conversions that can be performed by `reinterpret_cast` but not by `static_cast` are low-level operations based on reinterpreting the binary representations of the types, which on most cases results in code which is system-specific, and thus non-portable. For example:

可以通过reinterpret_cast执行但不能通过static_cast执行的转换是基于重新解释类型的二进制表示的低级操作，在大多数情况下，这会导致特定于系统的代码，因此不可移植。例如:

```
class A { /* ... */ };
class B { /* ... */ };
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

This code compiles, although it does not make much sense, since now `b` points to an object of a totally unrelated and likely incompatible class. Dereferencing `b` is unsafe.

这段代码进行编译，尽管它没有多大意义，因为现在' b '指向一个完全不相关且可能不兼容的类的对象。解除引用' b '是不安全的。

const_cast

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

这种类型的强制转换操作指针所指向的对象，可以是被设置的，也可以是被删除的。例如，要向需要非const实参的函数传递const指针:

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

void print (char * str)
{
  cout << str << '\n';
}

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

The example above is guaranteed to work because function `print` does not write to the pointed object. Note though, that removing the constness of a pointed object to actually write to it causes *undefined behavior*.

上面的示例保证可以工作，因为函数print不写入所指向的对象。但是请注意，删除指向对象的constness以实际向其写入会导致未定义的行为。

typeid

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

Typeid允许检查表达式的类型:

`typeid (expression)`
This operator returns a reference to a constant object of type `type_info` that is defined in the standard header `<typeinfo>`. A value returned by `typeid` can be compared with another value returned by `typeid` 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.

该操作符返回对type_info类型的常量对象的引用，该对象在标准头文件中定义。由typeid返回的值可以使用操作符==和!=与由typeid返回的另一个值进行比较，或者使用其name()成员获取以空结尾的表示数据类型或类名的字符序列。

```
// 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;
}
```

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:

当typeid应用于类时，typeid使用RTTI来跟踪动态对象的类型。当typeid应用于其类型是多态类的表达式时，结果是派生最完整的对象的类型:

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

class Base { virtual void f(){} };
class Derived : public Base {};

int main () {
  try {
    Base* a = new Base;
    Base* b = new Derived;
    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() << '\n'; }
  return 0;
}
```

Note: The string returned by member `name` of `type_info` depends on the specific implementation of your compiler and library. It is not necessarily a simple string with its typical type name, like in the compiler used to produce this output.

注意:type_info成员名返回的字符串取决于编译器和库的具体实现。它不一定是一个带有典型类型名的简单字符串，就像用于生成输出的编译器中那样。

Notice how the type that `typeid` considers for pointers is the pointer type itself (both `a` and `b` are of type `class Base *`). 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).

注意，typeid为指针考虑的类型是指针类型本身(a和b都是类基*类型*)。*然而，当typeid被应用到对象(如*a和*b)时，typeid会产生它们的动态类型(即它们最派生的完整对象的类型)。

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.

如果类型typeid的值前面是一个解引用操作符(*)的指针，并且这个指针有一个空值，那么typeid将抛出一个bad_typeid异常。