# 【C++】Day 13(2)

| | |
|---|---|
| Class | C++ |
| Date | @December 1, 2021 |
| Material | |
| Series Number | |
| Summary | Explicit Type Conversion |

## 【Ch4】Explicit Conversions

### 4.11.3 Explicit Conversions

Sometimes we want to explicitly force an object to be converted to a different type.

For example, we might want to use floating-point divison in the following code:

```
int i, j;
double slop = i / j;
```

To do so, we'd need a way to explicitly convert i and/or j to double. We use a cast to request an explicit conversion.

*Warning: Although necessary at times, casts are inherently dangerous constructs.*

*Named Casts*

A named cast has the following form:

```
cast-name<type> (expression);
```

where `type` is the target type of the conversion, and `expression` is the value to be cast. If type is a reference, then the result is an lvalue.

The cast-name may be one of `static_cast`, `dynamic_cast`, `const_cast`, and `reinterpret_cast`.

### *static_cast*

Any well-defined type conversion, other than those involving low-level const, can be requested using a static_cast.

For example, we can force our expression to use floating-point divison by cating one of the operands to double:

```
//cast used to force floating-point divison
double slop = static_cast<double>(j) / i;
```

A `static_cast` is useful to perform a conversion that the compiler will not generate automatically.

For example, we can use a static_cast to retrieve a pointer value that was stored in a void* pointer:

```
void* p = &d; //ok: address of any nonconset object can be stroed in a void*
//ok: converts void* back to the original pointer type
double *dp = static_cast<double*>(p);
```

### *const_cast*

A `const_cast` changes only a low-level const in its operand:

```
const char *pc;
char *ptr = const_cast<char*>(pc); //ok: but writing through p is undefined
```

Once we have cast away the const of an object, the compiler will no longer prevent us from writing to that object.

*Note: If the object was originally not a const, using a cast to obtain write access is legal.*

Only a const_cast may be used to change the the constness of an expression. Trying to change whether an expression is const with any of the other forms of name cast is a compile-time error. Similarly, we cannot use a const_cast to change the type of an expression.

```
const char *cp;
//error: static_cast can't cast away const
char *q = static_cast<char*>(cp);
static_cast<string>(cp); //ok: conferts string literal to string
const_cast<string>(cp); //error: const_cast can only chagne constness
```

A const_cast is most useful in the context of overloaded functions.

Use of const_cast:

Imagine that we have a function that asks for a pointer.

```
void func(int* ptr);

const int* myPtr;

funcs(const_cast<int*> ptr);
```

*reinterpret_cast*

A `reinterpret_cast` generally performs a low-level reinterpretation of the bit pattern of its operands.

As an example, given the following cast:

```
int *ip;
char *pc = reinterpret_cast<char*>(ip);
```

we must never forget that the actual object addressed by pc is an int, not a character.

*Warning: A reinterpret_cast is inherently machine dependent. Safely using reinterpret_cast requires completely understanding the types involved as well as the details of how the compiler implements the cast.*