83. Casting

Objectives

- Static casting
- Constant casting
- Reinterpret cast
- Dynamic cast

Casting

Recall that from CISS240, you can typecast an `int` to a `double`, a `double` to an `int`, a `char` to an `int`, an `int` to a `char`, etc.:

```
int i = 42;
double d = 3.14;
char c = 'A';
int flag = 1;

double j = double(i); // or (double) i
int e = int(d); // or (int) d
int f = int(c); // or (int) c
bool flag2 = bool(flag);

// ...
```

Recall from the notes on pointers, you cannot assign pointer values to a pointer variable if the pointer types are not the same:

```
int i = 42;
int * p = &i; // OK
double * q = &i; // BAD!!!
```

But in fact you can … if you perform an explicit typecast:

```
int i = 42;
int * p = &i; // OK
double * q = (double *)i; // OK ... but probably a
                          // bad idea ...
```

In general, it's a good idea to think carefully before you typecast.

The above typecast has been around since C (the parent language of C++ and also the ancestor language of many programming languages) and these are available in C++. C++ has also introduced it's own casting mechanisms through four different casting operators:

- `static_cast`
- `const_cast`
- `reinterpret_cast`
- `dynamic_cast`

static_cast

The **`static_cast`** is most similar to the basic typecasting of types which are very similar or only result in loss of precision, i.e., casting such as

```
int i = 42;
double d = 3.14;
char c = 'A';
bool flag = true;

double j = double(i);   // or (double) i
int e = int(d);         // or (int) i
int d = int(c);         // or (int) c
int flag2 = bool(flag); // or (int) flag

// ...
```

All the above can be executed using **`static_cast`**:

```
int i = 42;
double d = 3.14;
char c = 'A';
bool flag = true;

double i1 = static_cast< double >(i);
int d1 = static_cast< int >(d);
int c1 = static_cast< int >(c);
bool flag1 = static_cast< bool> (flag);

// ...
```

In general static cast is allowed when automatic typecasting is allowed. For instance

```
int i = 42;
double i1 = i;                        // automatic
double i2 = double(i);                // OK
double i3 = static_cast< double >(i); // OK
```

However this is not allowed:

```
int i = 42;
double * p = &i;
```

Therefore this will not compile:

```
int i = 42;
double * p = static_cast< double *>(&i); // BAD!!!
```

Of course C-style casting allows you to do this:

```
int i = 42;
double * p = (double *)(&i);
```

And this is **the whole point of C++ typecasting**. It's to subdivide C style castings into various kinds of C++ type casting so as to provide finer control and to provide more warnings.

That being said, note that in many areas of C++ programming, as long as you know what you are doing, i.e., you are only typecasting when automatic typecasting is allowed (and not crazy typecasting between pointers of different types), then you'll see that C-style typecasting is still used when writing C++ code.

For instance when you look OpenCV programs (CV = computer vision, OpenCV is a very famous open source CV library written in C/C++), it's very common to see C-style typecasting rather than C++ style typecasting. Also, OpenGL C++ program (for computer graphics) tend to use C-style typecasting.

const_cast

Look at this program:

```
void g(int & y)
{}

void f(const int & x)
{
    g(x);
}

int main{}
{
    f(42);
    return 0;
}
```

Will it compile? (I'll give you 5 seconds.) Next, try to compile it.

Of course it won't.

If you did not answer the above question correctly, you should review notes on references and also on pointers right away because the above is similar to

```
void g(int * y)
{}

void f(const int * x)
{
    g(x);
}

int main{}
{
    int i = 42;
    f(&i);
    return 0;
}
```

In particular look at constantness of pointers (and of references).

This will not compile

```
void g(int & y)
{}

void f(const int & x)
{
    g(x);
}

int main{}
{
    int i = 42;
    f(i);
    return 0;
}
```

because the y of `g()` references the x of `f()`, but y is a reference while x is a reference to a constant. This means that **x cannot change the value it is referencing** and yet x wants to **allow y to reference that value with the power to change it**. That's **not possible**.

This is similar to this issue (for references):

```
int i = 42;
const int & x = i; // x is weak ... change change i
int & y = x;       // y wants to be strong and wants
                   // to change i, through x. NOPE!!!
```

which has the same issue as this (for pointers):

```
int i = 42;
const int * x = &i;
int * y = x;
```

Going back to

```
void g(int & y)
{}

void f(const int & x)
{
    g(x);
}

int main{}
{
    f(42);
    return 0;
}
```

The issue is the x in `g(x)` **cannot** be referencing a constant `int`, i.e., you want to **remove the constantness** of the x:

```
void g(int & y)
{}

void f(const int & x)
{
    g(x);
}

int main{}
{
    f(42);
    return 0;
}
```

Of course one solution is this:

```
void g(const int & y)
{}

void f(const int & x)
{
    g(x);
}

int main{}
{
    f(42);
    return 0;
}
```

Then everything works. But if the function `g()` was written by someone else and you cannot change the source code, then what are you doing to do? (This does happen. For instance `g()` might be a function in a library that someone else wrote and you only have the header file and the compiled binary code, but not the source file. This means that although you can change the prototype in the header file it won't help because you cannot change `g()` in the cpp file.)

You can remove the constantness of the `x` in `f()`:

```
void g(int & y)
{}

void f(~~const~~ int & x)
{
    g(x);
}

int main{}
{
    f(42);
    return 0;
}
```

But what if the function `f()` should not change the value that `x` references? For instance what if `f()` is a printing function? Then `x` should be a constant reference … **except when calling `g()`**. In other words, you only want to remove the constantness of `x` **temporarily**. Then you can use the constant cast:

```
void g(int & y)
{}

void f(const int & x)
{
    g(const_cast< int & >(x));
}

int main{}
{
    f(42);
    return 0;
}
```

Go ahead and compile it now.

NOTE: This is (in my opinion) a wrong name. Because it makes you think that you are making something constant. In fact the point of this cast is to **remove** the constantness. A better name would be "remove_const_cast"!!!

This can happen in the following scenario. Suppose in your `Date` class, you forgot to make the `get_year()` method constant. Then this will not compile:

```
class Date
{
public:
    ...
    int get_yyyy() // OOPS ... forgot to make method
                   // constant!!!
    {
        return yyyy_;
    }
    ...
};

std::ostream & operator<<(std::ostream & cout,
                          const Date & date)
{
    cout << date.get_yyyy()
         << ...
    return cout;
}

int main{}
{
    f(42);
    return 0;
}
```

Exercise. Make sure you complete the above example to see the compilation error message from your compiler!!! The correct thing is to make the `get_yyyy()` constant. But suppose you can't. Then you would have to resort to this:

```
class Date
{
public:
    ...
    int get_yyyy() // OOPS ... forgot to make method
                   // constant!!!
    {
        return yyyy_;
    }
    ...
};

std::ostream & operator<<(std::ostream & cout,
                          const Date & date)
{
    cout << const_cast< Date & >(date).get_yyyy()
         << ...
    return cout;
}

int main{}
{
    f(42);
    return 0;
}
```

Get it?

In general, use of `const_cast` indicates an issue with the design of the code (just like the above scenario) – the issue is really that `get_yyyy()` should be made constant. But there are times when you have no choice when you don't have access to the source cpp file and the definition of the function/method is in the cpp file and not in the header file.

For the analogous pointer case, you can also use constant cast to resolve the issue:

```
void g(int * x)
{}

void f(const int * x)
{
    g(const_cast< int * >(x));
}

int main()
{
    int i = 42;
    f(&i);

    return 0;
}
```

reinterpret_cast and void *

Recall that while you can do C-style casting for pointer types:

```
int i = 42;
float * p = (float *)(&i);
```

Using C++ typecasting, the above is done using the `reinterpret_cast`. Try this:

```
int i = 42;
float * p = reinterpret_cast< float * >(&i);
```

You should try to do the above with **`static_cast`** and **`const_cast`** to check that neither of them works.

You should try this:

```
#include <iostream>

int main()
{
    int i = 42;
    float * p = reinterpret_cast< float * >(&i);
    *p = 0.0f;
    std::cout << i << '\n'; // you get 0 because the
                            // bit pattern of 0.0f
                            // happens to be the same
                            // bit pattern of 0.

    *p = 3.14f;
    std::cout << i << '\n'; // you do NOT get 3
                            // because the bit
                            // pattern of 3.14f is
                            // NOT 3.

    return 0;
}
```

The second output is `1078523331`. Why? Because the bit pattern for `3.14f` happens to be the bit pattern of the integer `1078523331`.

In general you should probably not use **`reinterpret_cast`** of pointers until you have taken CISS360 where you will learn more about bit representations and C/C++ bit programming.

Note that in the above p points to a `float` (the smaller version of `double`). So C++ will treat `*p` is a `float`. Sometimes (in the future), you might simply **want an address** and you **don't really want to interpret the value at that address** as an `int` or `double` or `float` or `char` or `bool` or anything at all. In that case, the pointer type you want is called

`void *`

Of course if q is a `void *`, you can print the address stored in q, you can give that address value to another pointer variable. But you should not work with `*q` since C++ does not know how to interpret the bits at that address. Run this:

```
int i = 42;
float * p = reinterpret_cast< float * >(&i);
*p = 0.0f;
std::cout << i << '\n';
*p = 3.14f;
std::cout << i << '\n';

void * q = reinterpret_cast< float * >(&i);
std::cout << q << '\n';    // OK
std::cout << (*q) << '\n'; // ARGH!!!! WRONG!!!
```

Run this too where you store an `int*` value in a `void*` pointer, pass the value of the pointer to another `void*` pointer, cast the `void*` value to a `double*`:

```
int * p0 = new int;
void * p1 = p0;
void * p2 = reinterpret_cast< void * >(p0);
double * p3 = reinterpret_cast< double * >(p2);
std::cout << p0 << ' ' << p1 << ' ' << p2 << ' '
          << p3 << '\n';
```

As I mentioned earlier, in the area of OpenCV and OpenGL, you'll see that many software engineers do not use the C++ style casting. For instance one function in OpenGL is

```
void glVertexAttribPointer(GLuint index,
                           GLint size,
                           GLenum type,
                           GLboolean normalized,
                           GLsizei stride,
                           const void * pointer);
```

and you'll see that most OpenGL programmers will simply use the simple C/C++ cast like this to call the above function:

```
glVertexAttribPointer(0,
                      3,
                      GL_FLOAT,
                      GL_FALSE,
                      5 * sizeof(GLfloat),
                      (GLvoid*) 0);
```

(note: `GLvoid` is just a typedef of `void.`)

dynamic_cast

Run the following **very important** example and read the code very slowly and very carefully:

```
#include <iostream>

class C4
{
public:
    virtual ~C4() {}
    virtual void f() { std::cout << "C4::f()\n"; }
    void g() { std::cout << "C4::g()\n"; }
};

class C3: public C4
{
public:
    virtual void f() { std::cout << "C3::f()\n"; }
    void g() { std::cout << "C3::g()\n"; }
};

class C2: public C3
{
public:
    virtual void f() { std::cout << "C2::f()\n"; }
    void g() { std::cout << "C2::g()\n"; }
};

class C1: public C2
{
public:
    virtual void f() { std::cout << "C1::f()\n"; }
    void g() { std::cout << "C1::g()\n"; }
};

class C0: public C1
{
public:
    virtual void f() { std::cout << "C0::f()\n"; }
    void g() { std::cout << "C0::g()\n"; }
};

class X {}; // X is NOT in the inheritance hierarchy C0,..,C4

int main()
{
    C3 * p = new C1;
    C3 * q = dynamic_cast< C3 * >(p);
    C2 * r = dynamic_cast< C2 * >(p);
    C1 * t = dynamic_cast< C1 * >(p);
    C0 * u = dynamic_cast< C0 * >(p);
    C4 * v = dynamic_cast< C4 * >(p);
    X * w = dynamic_cast< X * >(p);

    std::cout << "p:" << p << '\n'; p->f(); p->g();
    std::cout << "q:" << q << '\n'; q->f(); q->g();
    std::cout << "r:" << r << '\n'; r->f(); r->g();
    std::cout << "t:" << t << '\n'; t->f(); t->g();
    std::cout << "u:" << u << '\n';
    std::cout << "v:" << v << '\n'; v->f(); v->g();
    std::cout << "w:" << w << '\n';

    delete p;
    return 0;
}
```

Here's my output

```
p:0x605eb0
C1::f()
C3::g()
q:0x605eb0
C1::f()
C3::g()
r:0x605eb0
C1::f()
C2::g()
t:0x605eb0
C1::f()
C1::g()
u:0
v:0x605eb0
C1::f()
C4::g()
w:0
```

Note that some of the dynamic casts resulting in **`NULL`** **pointers**. That's when **dynamic casting fails**.

Note that p was declared like this:

```
C3 * p = new C1;
```

You can cast p as pointer to the classes from `C0` to `C4`. However casting to `C0` (i.e., below `C1`) will give you a `NULL` pointer. This allows you to access methods from `C1` to `C4`, although if the method is **virtual**, the search for this method starts at the "bottom" at `C1`. This is the case for `f()`. However `g()` is **not virtual**. Therefore

```
dynamic_cast< C2 * >(p).g();
```

will execute the `C2::g()` method.

Also, note that you can try to cast to a pointer to a class that is **not in the inheritance hierarchy** (look at the class X), but in that case you'll get **`NULL`**.

The **main reason** for dynamic cast is to **typecast a pointer from a class to a subclass**. Dynamic cast will give you a NULL if the typecast attempts to change the point type to a non-subclass type. So if you have

```
Z * p = new X;
dynamic_cast< Y * >(p).m();
```

where X, …, Y, …, Z is an inheritance hierarchy, then

- If **`m` is virtual**, the search for `m` **starts at X** and proceeds up the hierarchy until it's found.
- If **`m` is nonvirtual**, the search for `m` **starts at Y** and proceeds up the hierarchy until it's found.

Note that if `m` is virtual, then the dynamic cast is actually redundant (i.e. `Y` is ignored since the search for `m` starts at `X` anyway),

And if `Y` is a **subclass of X (or even further below)** or if `Y` is **not between X and Z** (in the inheritance hierarchy, the dynamic cast **fails** (i.e., it gives you `NULL`).

Here's another very important thing: in the above example, try to make all the `f()` non-virtual. When you compile, you'll get an error. In other words, **dynamic casting is allowed only when there is at least one virtual method** in the classes involved. (I'm not going to explain this other than the fact that when the classes involved are polymorphic with at least one virtual method, some extra information is included in the classes involved which is needed for dynamic casting to work.)

In general, the benefit of this cast is this: If you intend to cast a pointer along a class hierachy, then by doing

```
dynamic_cast< Y * >(p)
```

will remind you that `Y` is in the class hierarchy (usually) below the type used to declare `p`, i.e., the declaration of `p` is

```
Z * p;
```

and `Z` is an ancestor class of `Y`. If you accidentally change the `Y` to a class that is **not** under `Z`:

```
dynamic_cast< A * >(p) // ooops ...
```

then the resulting pointer is `NULL` and will probably trigger an error later on when you dereference `p`. Of course if you are cautious and you know what you are doing (i.e., please don't cast to `A *`), then the dynamic cast is not necessary.

Now, if you need to down cast (i.e., cast to a lower level subclass) to execute a non-virtual method in that class, then you might as well make that method virtual and use polymorphism to detect that method. So frequently (but not always), dynamic cast is a sign of bad design. However there are cases where it cannot be avoided. In any case, C++ is a language that gives you maximum freedom: you can use polymorphism to choose which virtual method to execute, or you can manually cast to the right class and then execute the non-virtual method.