C++ Casting

CSE 390C

Contents

Casting

Implicit Type Conversion

- When a value/variable is given in the context of another type, the compiler automatically performs the conversion

```C++
double a = 1; // int 1 converted to double 1.0
double b{2}; // int 2 converted to double 2.0
int func() { return 3.0;}// double 3.0 converted to int 3
double c = 10 / 3; // int division, takes value 3.0
if (c) {} // c converted to type bool
void param(double d) {}
param(3); // int 3 converted to double
```

C-style Casting

```C++
int x{10}, y{4};
double z = (double) x / y; // casts x to double
```

- C++ also offers another style of C-style cast, called **function-style cast**

```C++
double a = double(x) / y;
```

- Should be avoided as it is not clear which cast will be performed

Named Casts

C++ supports these cast options, known as **named casts**:

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

Makes your casting intent clear!

`static_cast`

- Used when explicitly converting a value of one type to another

```cpp
char c = 'c';
cout << static_cast<int>(c) << endl; // prints 97 instead of 'c'
```

- `static_cast<type>(t)` returns a temp `type` object having value of `t` converted to `type`.
- Checked at compile time
- Makes narrowing conversions explicit

dynamic_cast

- We've seen implicit conversion of a derived class pointer into base class (upcasting), with `dynamic_cast` we can do the opposite (downcasting)

```C++
class Person {...};
class Student : public Person {...};
Person * p1 {new Student()}; // implicit conversion
Student * s1 {dynamic_cast<Student*>(p1)}; // convert Base class into Derived class pointer
```

dynamic_cast

- `dynamic_cast` fails at compile time due to unrelated types

```C++
class Animal {};
Animal * a1{new Animal()};
Student * s2{dynamic_cast<Animal*>}; // Animal and Student are unrelated
```

`dynamic_cast`

- If it fails at runtime, the result of the conversion will be a null pointer.

```C++
class Employee : public Person {};
Employee * e1 {dynamic_cast<Student*>(s1)};
```

- Always ensure any `dynamic_cast` calls succeeded by checking for `nullptr`:

```C++
if (e1) {
  // any operations on e1
}
```

`const_cast`

- Used to add/remove `const` of a pointer or reference
- Considered bad style to use
- Adding `const`

```cpp
int a = 1;
int & b = a;
const int & c = const_cast<const int &>(b)
b++; // Note: b is still not const
```

`const_cast`

- Removing `const`

```C++
int a = 3;
const int & b = a;
int & c = const_cast<int &>(b);
c++;
b++; // won't work, b is const
```

`const_cast`

- Removing `const` can lead to undefined behavior

```C++
1  const int x = 10;
2  int & y = const_cast<int &>(x);
3  y++;
```

`reinterpret_cast`

- Reinterprets the bit-level representation of one type as if it were another type

```cpp
int i = 0x4048F5C3; // bit value of 3.14 as a float
float *f = reinterpret_cast<float *>(&i);
int *j = reinterpret_cast<int *>(&i);
cout << *f << " " << *j << endl;
// outputs 1078523331 3.14
```

Bonus

C-style cast

- C-style cast don't make it clear what cast was done
- It works by trying casts in this order:
  - `const_cast`
  - `static_cast`
  - `static_cast`, followed by `const_cast`
  - `reinterpret_cast`
  - `reinterpret_cast`, followed by `const_cast`
- Since most of these are unsafe, C-style casting is also unsafe