

C++ Casting

CSE 390C

Contents

Casting	3
Implicit Type Conversion	4
C-style Casting	5
Named Casts	6
static_cast	7
dynamic_cast	8
const_cast	11
reinterpret_cast	14
Bonus	15
C-style cast	16

Casting

Implicit Type Conversion

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

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



C-style Casting

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



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

```
1 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

```
1 char c = 'c';  
2 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)

```
1 class Person {...};  
2 class Student : public Person {...};  
3 Person * p1 {new Student()}; // implicit conversion  
4 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

```
1 class Animal {};  
2 Animal * a1{new Animal()};  
3 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.

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



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

```
1 if (e1) {  
2     // any operations on e1  
3 }
```



const_cast

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

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



const_cast

- Removing `const`

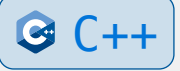
```
1 int a = 3;  
2 const int & b = a;  
3 int & c = const_cast<int &>(b);  
4 c++;  
5 b++; // won't work, b is const
```



const_cast

- Removing `const` can lead to undefined behavior

```
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

```
1 int i = 0x4048F5C3; // bit value of 3.14 as a float
2 float *f = reinterpret_cast<float *>(&i);
3 int *j = reinterpret_cast<int *>(&i);
4 cout << *f << " " << *j << endl;
5 // 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