C++ Casting

CSE 390C

C-Style Casting

```
// lhs = (new_type) rhs;
int var = (int) 3.90;
```

- Simple syntax that we've seen in Java and C
- Converts primitive data types to another (ex. `double` to `int`)
- Can also be used to convert between pointers of different types
  - The data being pointed to will just be interpreted differently
- Still valid in C++, but bad style…

C++ Casting

- Four different types of casts:
  - `static_cast<type>(expression)`
  - `dynamic_cast<type>(expression)`
  - `const_cast<type>(expression)`
  - `reinterpret_cast<type>(expression)`
- Makes your intent clearer for readers of your code!

static_cast

- Used to:
  - Convert class pointers to a related type (within the same class hierarchy)
    - Be careful casting down a class hierarchy
  - Convert between different primitive types
- `static_casts` are checked at compile time

```
class unrelated { };

class base { };

class derived : public base { };
```

```
// non-pointer conversion
int var = static_cast<int>(3.90);

base b;
derived d;

// invalid type conversion
unrelated* unrelated_ptr = static_cast<unrelated*>(&b);
// valid static cast
base* base_ptr = static_cast<base*>(&d);
// valid static cast, but dangerous
derived* derived_ptr = static_cast<derived*>(&b);
```

dynamic_cast

- Used to:
  - Convert pointers/references of related types (within the same class hierarchy)
- At *compile* time, verify that the pointers have related types
- At *run time,* if casting from base class to derived class, verify that object is actually an instance of the derived class.
- Used mainly for handling polymorphism/inheritance

```
base b;
derived d;

// unrelated type conversion (error caught at compile time)
unrelated* unrelated_ptr = dynamic_cast<unrelated*>(&b);
// valid dynamic casts
base* base_ptr = dynamic_cast<base*>(&d);
derived* derived_ptr = dynamic_cast<derived*>(base_ptr);
// compiles but will fail during run time (will return nullptr)
base_ptr = &b;
derived_ptr = dynamic_cast<derived*>(base_ptr);
```

const_cast

- Used to add or strip "const-ness" of a pointer or reference.
- Often, regarded as bad style (and even dangerous), since violates the intent and const-ness of the variable
- Striping const-ness:
  - If variable being pointed to was defined as const, this behavior is undefined!
  - Otherwise, can modify underlying data through this pointer/reference

```
// undefined behavior!
const int x = 123;
int& y = const_cast<int&>(x);
y++;
```

```
// strips const-ness away
int a = 10;
const int& b = a;
int& c = const_cast<int&>(b);
c++;
b++;
```

const_cast cont.

- Adding const-ness:
  - Only gives this pointer/reference const-ness, but others can still modify the data this points to/references

```
// adds const-ness
int u = 143;
int& v = u;
const int& w = const_cast<const int&>(v);
v++;
w++;
```

reinterpret_cast

- Used to convert between incompatible types
  - Ex. `int` to a pointer or vice-versa
  - Also, can convert between incompatible pointer types
- Low-level bit pattern is kept the same, just interpreted differently
- Dangerous!!!

```
// sets str to point to x, but treat x as a string???
int x = 345;
std::string* str = reinterpret_cast<std::string*>(&x);

// does not print "345", instead undefined behavior
std::cout<< *str << std::endl;
```

Implicit Conversion

```
int x = 65.5;   // double -> int
char c = x;     // int -> char, ASCII conversion
```

- When a variable is used in the context of a different type (without an explicit cast)
  - The compiler will then attempt to infer the correct conversion
  - The compiler can also take advantage of single parameter class constructors
    - ex. (`const char*`) -> `string`

```
void foo(std::string str);
```

```
foo("this a string literal");  // const char* -> string
```