C++ type-casting
Real-Time Type Information

- Version 1: Dr. Ofir Pele
- Version 2: Dr. Miri Ben-Nissan
- Version 3: Dr. Erel Segal-Halevi

C style casting:

- ```
  double d = 3.0;
  int i = (int) d;
  ```

- ```
  const int *cP = &i;
  int *ncP = (int*)cp;
  ```

- ```
  double * dP = (double*)ncP;
  ```

- ```
  Shape *s1 = new Circle;
  Shape *s2 = new Shape;
  Circle *s1c = (Circle*) s1;  s1c->setradius(1);
  Circle *s2c = (Circle*) s2;  s2c->setradius(1);
  ```

C++ style casting

- `const_cast<type>(expression)`
- `reinterpret_cast<type>(expression)`
- `static_cast<type>(expression)`
- `dynamic_cast<type>(expression)`

The ‘const_cast‘ operator

```
const_cast<type>(expression)
```

Is used to remove const-ness:

```
void g(C* cp);// programmer forgot “const”
void f(C const* cp)
{
   g(const_cast<C *>(cp));
}
```

- Usually, you should design your variables/methods such that you won’t have to use `const_cast`.
- Compile time operator
- Can cause serious trouble

‘reinterpret_cast‘ operator

```
reinterpret_cast<type>(expression)
```

- Reinterpret byte patterns.
- Circumvents type checking.
- Implementation-dependent.
- (Should be) used rarely.
- Very dangerous! (folder 4).
- Usage example: writing an image file (folder 5).

‘reinterpret_cast‘ operator

```
reinterpret_cast<type>(expression)
```

The ‘static_cast’ operator (folder 4)

```
static_cast<type>(expression)
```

When conversion method is known during compilation:

- double→int, int→double, etc.
- Conversion operator / conversion constructor.
- up-cast – Circle→Shape, Circle*→Shape*.

Safer that “old-style” casts

- e.g. won’t cast `int*` to `float*`

Failure causes a compiler error

- No dynamic checking is done

static_cast vs reinterpret_cast

reinterpret_cast does not do anything at runtime.

static_cast does at runtime a conversion determined at compile time.

Copy&paste into godbolt.org to see.

```
int main() {
    int i = 5;
    double d;
    d = (double)i;
    d =
static_cast<double>(i);
    int& ir = i;
    double& dr0 =
(double&)ir;
    double& dr =
reinterpret_cast<double&>(
ir);
}
```

Run Time Type Information (RTTI)

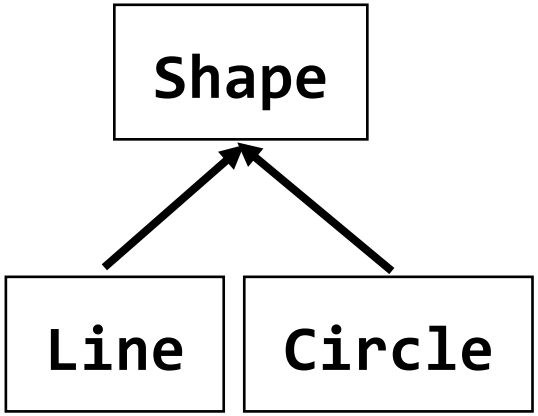
Run Time Type Information (RTTI) – why?

Problem:

Up-casting works fine.

- Treating sub-class as base class:

```
Shape * s = new Circle();
```



What about **down-casting**?

- might not be safe!
- correctness cannot be determined by the compiler.

```
Circle * c = (Circle*) s;
```

RTTI

RTTI – Run Time Type Information

Mechanisms for RTTI:

1. `dynamic_cast` operator
2. `typeid` operator (and type_info class)

The ‘dynamic_cast‘ operator

```
dynamic_cast<T>(expression)
```

Enables run-time type checking:

When `expression` is a **pointer**:

- Returns a valid pointer if `expression` really points to type `T`
- null pointer value otherwise

The ‘dynamic_cast‘ operator

```
dynamic_cast<T>(expression)
```

Enables run-time type checking:

When `expression` is a **reference**:

- Returns a valid reference if `expression` is really of type `T`
- Throws an exception when it fails (“`bad_cast`”)

dynamic_cast : example

```
Shape* s = container.pop();
Circle* c = dynamic_cast<Circle*>(s);
if (c != nullptr) {// c is a circle
   c->setRadius(42);
} else {
   ...
}
```

dynamic_cast - more

```
dynamic_cast<T>(expression)
```

Note:

- Used only on **pointer** or **reference** types.
- Can be used for:
  - **up-cast** – useless – done automatically.
  - **down-cast** – most useful.
  - **cross-cast** – useful with multiple inheritance.
- **Only for types with virtual-functions** (“Polymorphic types”) - These object have a space for information about type: the virtual function table.

dynamic_cast: only for polymorphics

```
class Circle : public Shape
{
   virtual void draw();
}
class Date : public Time
{
    // Time has no virtual functions
}
void foo(Shape * s, Time * t)
{
   Circle * c =
    dynamic_cast<Circle*>( s ); //ok
   Date * date =
    dynamic_cast<Date*>( t ); //compilation error
}
```

RTTI : typeid operator

RTTI : typeid operator (folder 6)

- Obtains info about an object/expression

  usage: `typeid( obj )` (like "`sizeof`")

Example:

```
Animal& d = *new Dog{}; // virt. class
Animal& c = *new Cat{};
cout <<   "d is a " << typeid(d).name()
     << ", c is a " << typeid(c).name()


     <<endl;
```

Output (might be):

d is a Dog, c is a Cat

RTTI : typeid operator

- Obtains info about an object/expression

  usage: `typeid( obj )` (like "`sizeof`")

Example:

```
Animal& d = *new Dog{}; // virt. class
Animal& c = *new Cat{};
cout <<   "d is a " << typeid(d).name()
     << ", c is a " << typeid(c).name()
     <<endl;
```

Output (becase it does not have to be the name of the type that the programmer used, it might also be):

d is a 3Dog, c is a 3Cat

RTTI **misuse**

```
void rotate( shape const& s )
{
   if (typeid(s) == typeid(Circle) )
   //do nothing
else if (typeid(s) == typeid(Triangle) )
   //rotate Triangle
else if (typeid(s) == typeid(Rectangle) )
   //rotate Rectangle
}
```

- **Use virtual functions when you can !**
- **Use RTTI only you cannot use virtual functions (e.g. the source code of Shape is unavailable).**

Cast comparison

	Compile-time	Run-time
const_cast	Check that source, target are the same except const	nothing
reinterpret_cast	Check that source, target are pointers / refs	nothing
static_cast	Check that there is a conversion source → target	Fixed conversion
dynamic_cast	Check that the class is polymorphic	Expensive check