C++ CASTS

C++ CASTS

ARE WAY MORE SOPHISTICATED THAN C-CASTS.

IN C, YOU CAN CAST PRETTY MUCH ANYTHING TO ANYTHING

```
int  *a;
char *b = (char *) a;
```

IN C++, ON THE OTHER HAND, THERE ARE 4 SPECIFIC TYPES OF CASTS, AND ALL DO DIFFERENT THINGS

IN C, YOU CAN CAST PRETTY MUCH ANYTHING TO ANYTHING

```
int  *a;
char *b = (char *) a;
```

C-STYLE CASTING IS CALLED TYPE CASTING,
AND YOU SHOULD AVOID IT IN C++!

C++ CASTS

ARE WAY MORE SOPHISTICATED THAN C-CASTS.

IN C, YOU CAN CAST PRETTY MUCH ANYTHING TO ANYTHING

C-STYLE CASTING IS CALLED TYPE CASTING, AND YOU SHOULD AVOID IT IN C++!

```
int  *a;
char *b = (char *) a;
```

IN C++, ON THE OTHER HAND, THERE ARE 4 SPECIFIC TYPES OF CASTS, AND ALL DO DIFFERENT THINGS

IN C++, ON THE OTHER HAND, THERE ARE 4 SPECIFIC TYPES OF CASTS, AND ALL DO DIFFERENT THINGS

`const_cast`

`static_cast`

`dynamic_cast`

`reinterpret_cast`

REINTERPRET_CAST IS BASICALLY A HALF-BAKED TYPECAST - VERY ARCANE, SO FORGET ABOUT IT - ALMOST NEVER USED

IN C++, ON THE OTHER HAND, THERE ARE 4 SPECIFIC TYPES OF CASTS, AND ALL DO DIFFERENT THINGS

`const_cast`

GET RID OF CONST-NESS

`static_cast`

GENERAL CONVERSIONS, INCLUDING THOSE MARKED EXPLICIT

`dynamic_cast`

"DOWNCAST" (SHAPE TO RECTANGLE IN A SAFE MANNER)

`reinterpret_cast`

REINTERPRET_CAST IS BASICALLY A HALF-BAKED TYPECAST - VERY ARCANE, SO FORGET ABOUT IT - ALMOST NEVER USED

IN C++, ON THE OTHER HAND, THERE ARE 4 SPECIFIC TYPES OF CASTS, AND ALL DO DIFFERENT THINGS

const cast

COMPILE-TIME CHECK

GET RID OF CONST-NESS

static cast

COMPILE-TIME CHECK

GENERAL CONVERSIONS, INCLUDING THOSE MARKED EXPLICIT

dynamic cast

RUN-TIME CHECK

"DOWNCAST" (SHAPE TO RECTANGLE IN A SAFE MANNER)

reinterpret_cast

REINTERPRET_CAST IS VERY ARCANE, SO FORGET ABOUT IT - ALMOST NEVER USED

EXAMPLE 71: UNDERSTAND

const_cast

const_cast

COMPILE-TIME CHECK

GET RID OF CONST-NESS

A NON-CONST REFERENCE CAN'T REFER TO A CONST OBJECT

```
const Student studentOne(name);
Student& studentTwo = studentOne;
```

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example71.cpp
Example71.cpp:45:12: error: binding of reference to type 'Student' to a value of type 'const Student' drops qualifiers
  Student& studentTwo = studentOne;
           ^            ~~~~~~~~~~
1 error generated.
```

A NON-CONST REFERENCE CAN'T REFER TO A CONST OBJECT

```
const Student studentOne(name);
Student& studentTwo = studentOne;
```

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example71.cpp
Example71.cpp:45:12: error: binding of reference to type 'Student' to a value of type 'const Student' drops qualifiers
  Student& studentTwo = studentOne;
           ^            ~~~~~~~~~~
1 error generated.
```

BUT CONST_CAST ELIMINATES THE COMPILER ERROR!

```
const Student studentOne(name);
Student& studentThree = const_cast<Student &>(studentOne);
```

BUT CONST_CAST ELIMINATES THE COMPILER ERROR!

```
const Student studentOne(name);
Student& studentThree = const_cast<Student &>(studentOne);
```

NOTICE THAT TEMPLATE PARAMETER!

BUT CONST_CAST ELIMINATES THE COMPILER ERROR!

```
const Student studentOne(name);
Student& studentThree = const_cast<Student &>(studentOne);
```

NOTICE THAT TEMPLATE PARAMETER!

THIS IS THE TYPE TO CONVERT TO

BUT CONST_CAST ELIMINATES THE COMPILER ERROR!

```
const Student studentOne(name);
Student& studentThree = const_cast<Student &>(studentOne);
```

NOTICE THAT TEMPLATE PARAMETER!

(THIS IS THE TEMPLATE PARAMETER FOR ALL C++ CASTS)

THIS IS THE TYPE TO CONVERT TO

BUT CONST_CAST ELIMINATES THE COMPILER ERROR!

```
const Student studentOne(name);
Student& studentThree = const_cast<Student &>(studentOne);
```

NOTICE THAT TEMPLATE PARAMETER!

EXAMPLE 72: UNDERSTAND

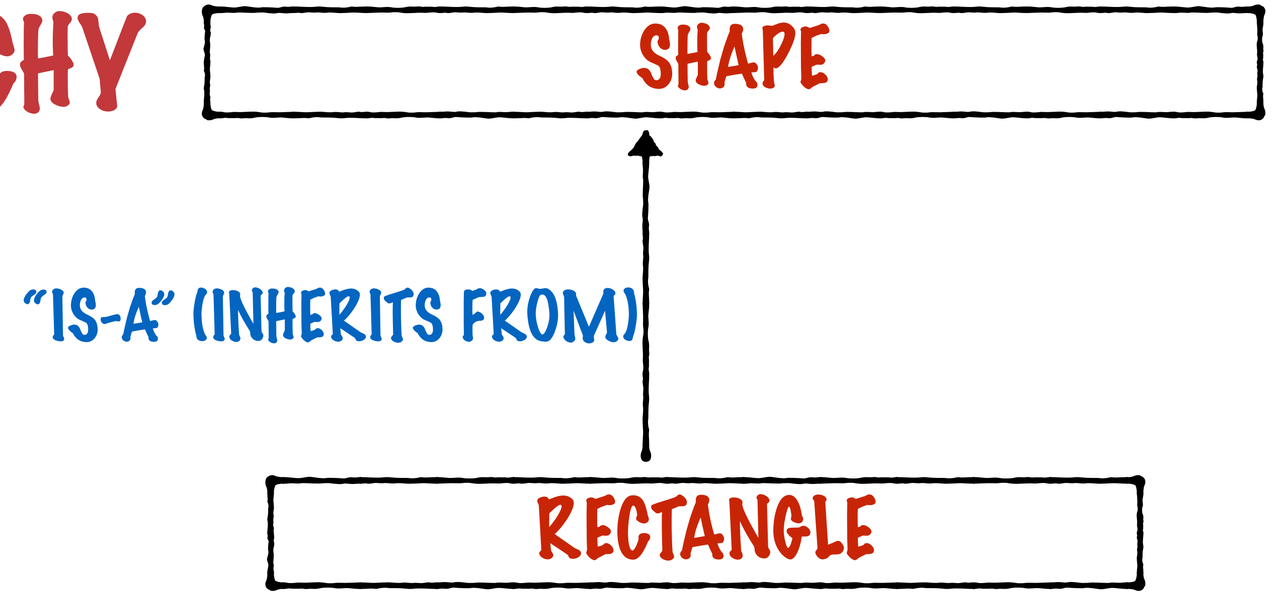
`dynamic_cast`

dynamic cast

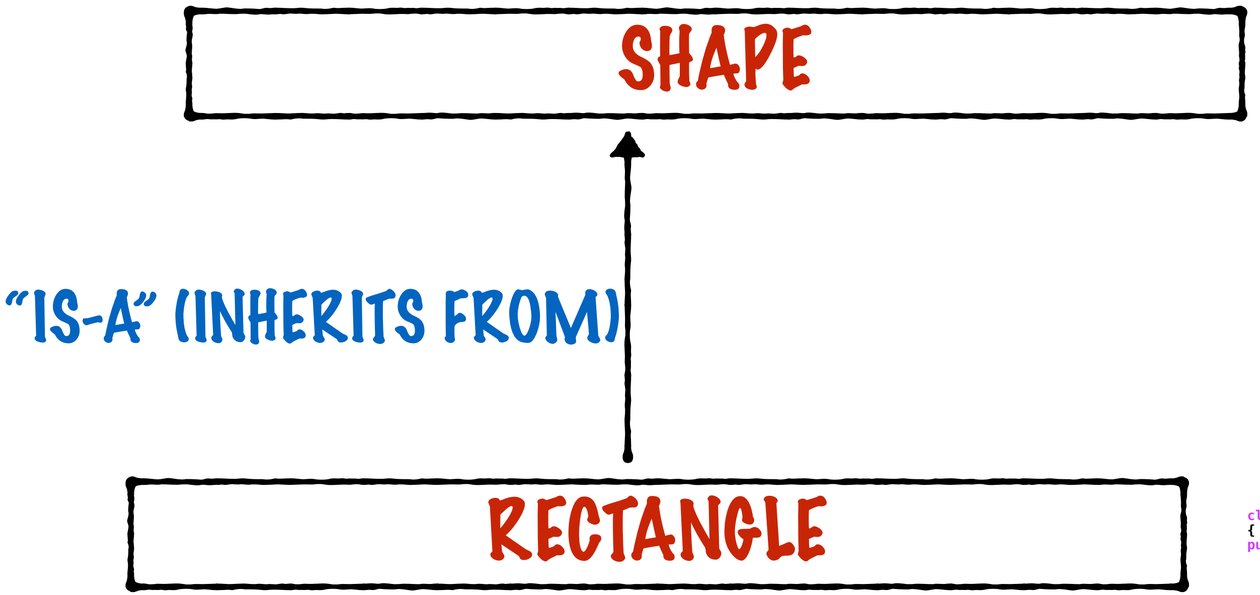
RUN-TIME CHECK

"DOWNCAST" (SHAPE TO RECTANGLE IN A SAFE MANNER)

SAY WE HAVE OUR FAMILIAR INHERITANCE HIERARCHY



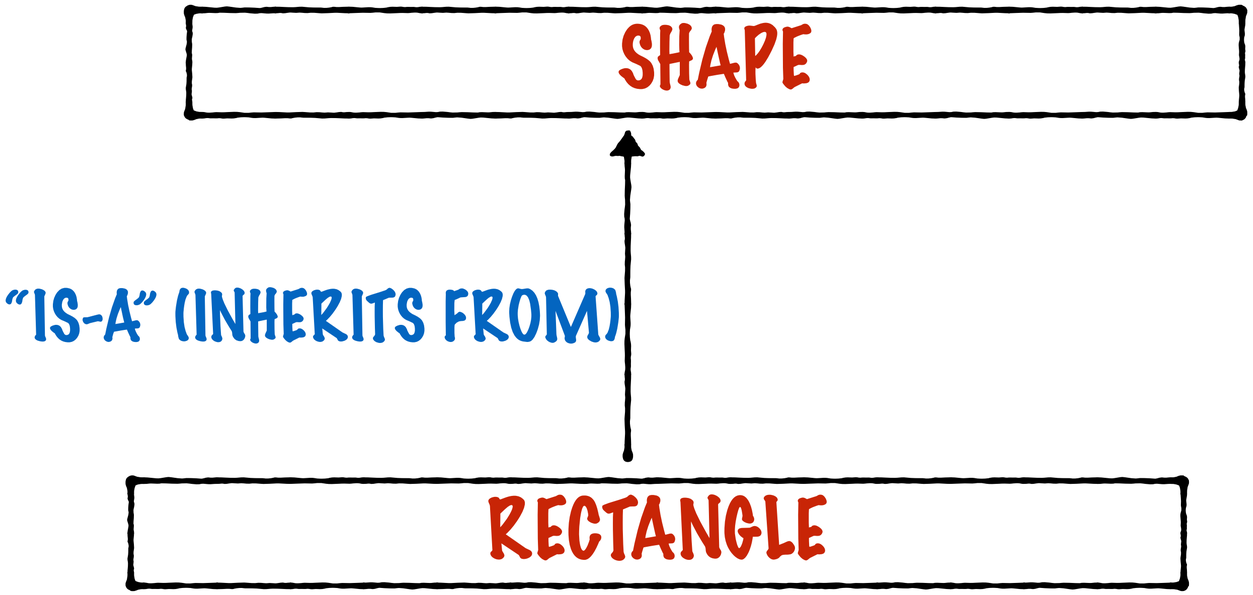
SAY WE HAVE OUR FAMILIAR INHERITANCE HIERARCHY




```cpp
class Shape
{
private:
public:
  string shapeType;
  void print()
  {
    cout << "I am a shape " << endl;
  }
  Shape()
  {
    shapeType = "Unknown";
    cout << "Inside the Shape constructor" << endl;
  }

  virtual ~Shape()
  {
    cout << "Inside the Shape destructor" << endl;
  }
};
```

```cpp
class Rectangle : public Shape
{
public:
  int rectangle_length;
  int rectangle_breadth;
  void print()
  {
    cout << "I am a rectangle " << endl;
  }
  Rectangle()
  {
    cout << "Inside the Rectangle constructor" << endl;
  }
  ~Rectangle()
  {
    cout << "Inside the Rectangle destructor" << endl;
  }
};
```

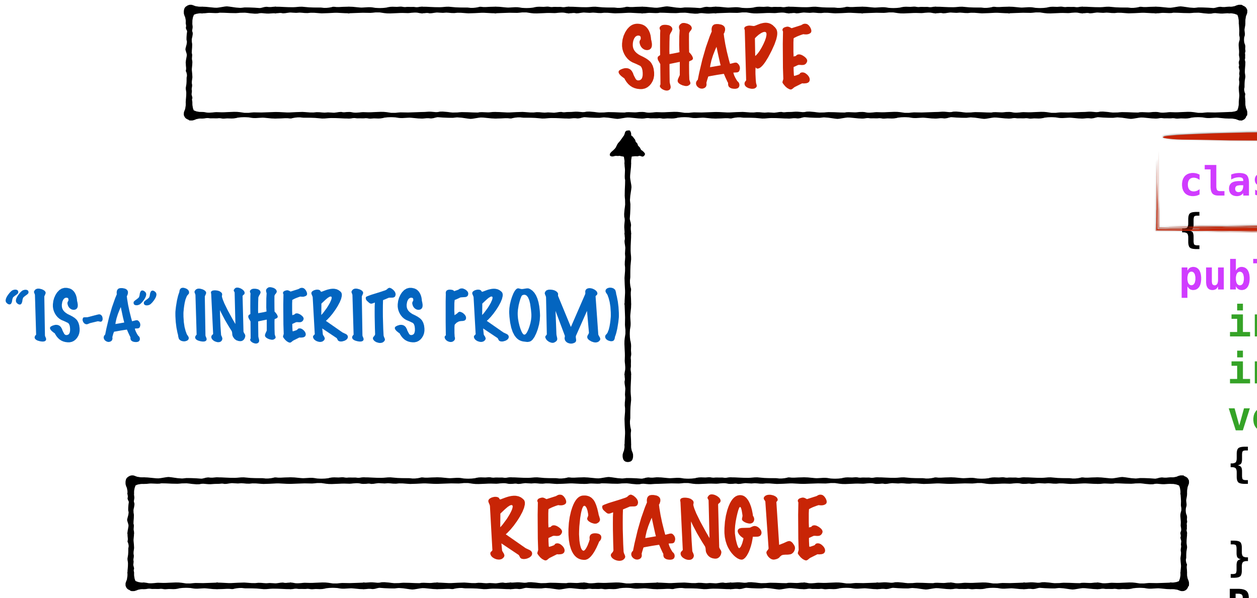
SAY WE HAVE OUR FAMILIAR INHERITANCE HIERARCHY




```cpp
class Rectangle : public Shape
{
public:
  int rectangle_length;
  int rectangle_breadth;
  void print()
  {
    cout << "I am a rectangle " << endl;
  }
  Rectangle()
  {
    cout << "Inside the Rectangle constructor" << endl;
  }
  ~Rectangle()
  {
    cout << "Inside the Rectangle destructor" << endl;
  }
};
```

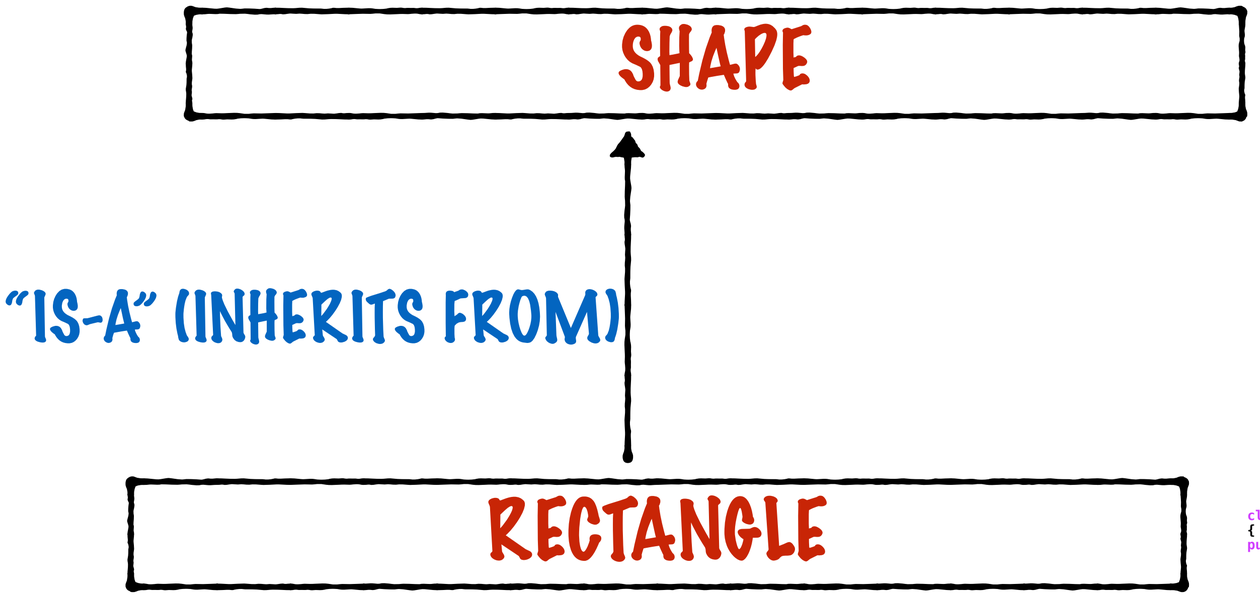
```cpp
class Shape
{
private:
public:
  string shapeType;
  void print()
  {
    cout << "I am a shape " << endl;
  }
  Shape()
  {
    shapeType = "Unknown";
    cout << "Inside the Shape constructor" << endl;
  }

  virtual ~Shape()
  {
    cout << "Inside the Shape destructor" << endl;
  }
};
```

SAY WE HAVE OUR FAMILIAR INHERITANCE HIERARCHY




```
class Rectangle : public Shape
{
public:
   int rectangle_length;
   int rectangle_breadth;
   void print()
   {
      cout << "I am a rectangle " << endl;
   }
   Rectangle()
   {
      cout << "Inside the Rectangle constructor" << endl;
   }
   ~Rectangle()
   {
      cout << "Inside the Rectangle destructor" << endl;
   }
};
```

SAY WE HAVE OUR FAMILIAR INHERITANCE HIERARCHY




```cpp
class Shape
{
private:
public:
  string shapeType;
  void print()
  {
    cout << "I am a shape " << endl;
  }
  Shape()
  {
    shapeType = "Unknown";
    cout << "Inside the Shape constructor" << endl;
  }

  virtual ~Shape()
  {
    cout << "Inside the Shape destructor" << endl;
  }
};
```

```cpp
class Rectangle : public Shape
{
public:
  int rectangle_length;
  int rectangle_breadth;
  void print()
  {
    cout << "I am a rectangle " << endl;
  }
  Rectangle()
  {
    cout << "Inside the Rectangle constructor" << endl;
  }
  ~Rectangle()
  {
    cout << "Inside the Rectangle destructor" << endl;
  }
};
```

NOW WE HAVE 2 POINTERS...

```
Shape * shape = new Shape();
Shape * shape2 = new Rectangle();
```

AND 2 CASTS

```
// dynamic_cast #1: successful - cast is possible, so return value is a valid pointer
Rectangle * rectangle = dynamic_cast<Rectangle*>(shape2);

// dynamic_cast #2: unsuccessful - cast is no possible, so return value is NULL
Rectangle * rectangle2 = dynamic_cast<Rectangle*>(shape);
```

NOW WE HAVE 2 POINTERS...

```
Shape * shape = new Shape();
Shape * shape2 = new Rectangle();
```

AND 2 CASTS

BOTH CASTS ARE DOWNCASTS - THEY TRY AND CONVERT A "SHAPE" OBJECT TO A "RECTANGLE" OBJECT

```
// dynamic_cast #1: successful - cast is possible, so return value is a valid pointer
Rectangle * rectangle = dynamic_cast<Rectangle*>(shape2);

// dynamic_cast #2: unsuccessful - cast is no possible, so return value is NULL
Rectangle * rectangle2 = dynamic_cast<Rectangle*>(shape);
```

NOW WE HAVE 2 POINTERS...

```
Shape * shape = new Shape();
Shape * shape2 = new Rectangle();
```

ONE OF THEM SHOULD SUCCEED, SINCE THAT SHAPE REALLY IS A RECTANGLE

AND 2 CASTS

```
// dynamic_cast #1: successful - cast is possible, so return value is a valid pointer
Rectangle * rectangle = dynamic_cast<Rectangle*>(shape2);

// dynamic_cast #2: unsuccessful - cast is no possible, so return value is NULL
Rectangle * rectangle2 = dynamic_cast<Rectangle*>(shape);
```

NOW WE HAVE 2 POINTERS...

```
Shape * shape = new Shape();
Shape * shape2 = new Rectangle();
```

AND 2 CASTS

ONE OF THEM SHOULD SUCCEED, SINCE THAT SHAPE REALLY IS A RECTANGLE

```
// dynamic_cast #1: successful - cast is possible, so return value is a valid pointer
Rectangle * rectangle = dynamic_cast<Rectangle*>(shape2);
```

```
// dynamic_cast #2: unsuccessful - cast is no possible, so return value is NULL
Rectangle * rectangle2 = dynamic_cast<Rectangle*>(shape);
```

NOW WE HAVE 2 POINTERS...

```
Shape * shape = new Shape();
Shape * shape2 = new Rectangle();
```

ONE OF THEM SHOULD SUCCEED, SINCE THAT SHAPE REALLY IS A RECTANGLE

AND 2 CASTS

```
// dynamic_cast #1: successful - cast is possible, so return value is a valid pointer
Rectangle * rectangle = dynamic_cast<Rectangle*>(shape2);

// dynamic_cast #2: unsuccessful - cast is no possible, so return value is NULL
Rectangle * rectangle2 = dynamic_cast<Rectangle*>(shape);
```

NOW WE HAVE 2 POINTERS...

```
Shape * shape = new Shape();
Shape * shape2 = new Rectangle();
```

BUT THE OTHER ONE SHOULD NOT SUCCEED, SINCE THAT SHAPE IS NOT A RECTANGLE!

AND 2 CASTS

```
// dynamic_cast #1: successful - cast is possible, so return value is a valid pointer
Rectangle * rectangle = dynamic_cast<Rectangle*>(shape2);
```

```
// dynamic_cast #2: unsuccessful - cast is no possible, so return value is NULL
Rectangle * rectangle2 = dynamic_cast<Rectangle*>(shape);
```

NOW WE HAVE 2 POINTERS...

```
Shape * shape = new Shape();
Shape * shape2 = new Rectangle();
```

AND 2 CASTS

BUT THE OTHER ONE SHOULD NOT SUCCEED, SINCE THAT SHAPE IS NOT A RECTANGLE!

```
// dynamic_cast #1: successful - cast is possible, so return value is a valid pointer
Rectangle * rectangle = dynamic_cast<Rectangle*>(shape2);
```

```
// dynamic_cast #2: unsuccessful - cast is no possible, so return value is NULL
Rectangle * rectangle2 = dynamic_cast<Rectangle*>(shape);
```

```
// dynamic_cast #1: successful - cast is possible, so return value is a valid pointer
Rectangle * rectangle = dynamic_cast<Rectangle*>(shape2);
```

```
// dynamic_cast #2: unsuccessful - cast is no possible, so return value is NULL
Rectangle * rectangle2 = dynamic_cast<Rectangle*>(shape);
```

AND THAT'S WHAT DYNAMIC CAST DOES

IT USES SOMETHING CALLED RTTI (REALTIME TYPE IDENTIFICATION) TO ENSURE - AT RUNTIME - THAT A CAST IS OK

```
// dynamic_cast #1: successful - cast is possible, so return value is a valid pointer
Rectangle * rectangle = dynamic_cast<Rectangle*>(shape2);
```

```
// dynamic_cast #2: unsuccessful - cast is no possible, so return value is NULL
Rectangle * rectangle2 = dynamic_cast<Rectangle*>(shape);
```

AND THAT'S WHAT DYNAMIC CAST DOES

IT USES SOMETHING CALLED RTTI (REALTIME TYPE IDENTIFICATION) TO ENSURE - AT RUNTIME - THAT A CAST IS OK

AN UNSUCCESSFUL CAST RETURNS NULL, WHICH IS EASY TO CHECK FOR

DYNAMIC CAST

IT USES SOMETHING CALLED RTTI (REALTIME TYPE IDENTIFICATION) TO ENSURE - AT RUNTIME - THAT A CAST IS OK

AN UNSUCCESSFUL CAST RETURNS NULL, WHICH IS EASY TO CHECK FOR

HAD WE DONE THIS SAME OPERATION USING STATIC_CAST - WHICH IS COMPILE-TIME - BEHAVIOUR WOULD BE UNDEFINED

HAD WE DONE THIS SAME OPERATION USING STATIC_CAST - WHICH IS COMPILE-TIME - BEHAVIOUR WOULD BE UNDEFINED

HAD WE DONE THIS SAME OPERATION USING STATIC_CAST - WHICH IS COMPILE-TIME - BEHAVIOUR WOULD BE UNDEFINED

HAD WE DONE THIS SAME OPERATION USING STATIC_CAST - WHICH IS COMPILE-TIME - BEHAVIOUR WOULD BE UNDEFINED

USE ONLY DYNAMIC CAST FOR DOWN CASTING - NEVER USE STATIC CAST

DYNAMIC CAST

IT USES SOMETHING CALLED RTTI (REALTIME TYPE IDENTIFICATION) TO ENSURE - AT RUNTIME - THAT A CAST IS OK

THE OTHER INTERESTING THING YOU CAN DO WITH RTTI IS: PRINT THE NAME OF A TYPE USING `typeid`

```
cout << "Using RTTI to print the type of an object :";
cout << typeid(*rectangle2).name() << endl;
```

RTTI (REALTIME TYPE IDENTIFICATION)

THE OTHER INTERESTING THING YOU CAN DO WITH RTTI IS: PRINT THE NAME OF A TYPE USING `typeid`

```
cout << "Using RTTI to print the type of an object :";
cout << typeid(*rectangle2).name() << endl;
```

```
Using RTTI to print the type of an object :9Rectangle
```

EXAMPLE 73: UNDERSTAND

`static_cast`

EXAMPLE 73: UNDERSTAND

`static_cast`

SAY WE HAVE 2 CLASSES TO REPRESENT A COMPLEX NUMBER

`ComplexNumber` `ComplexNumber_Polar`

SAY WE HAVE 2 CLASSES TO REPRESENT A COMPLEX NUMBER

ComplexNumber **ComplexNumber_Polar**

NOW EACH CLASS HAS A COPY CONSTRUCTOR TO CREATE AN OBJECT FROM AN OBJECT OF THE OTHER..

```
explicit ComplexNumber(const ComplexNumber_Polar& polar)
{
  realPart = polar.modulus * cos(polar.argument * 3.1415/180);
  complexPart = polar.modulus * sin(polar.argument * 3.1415/180);

}
```

SAY WE HAVE 2 CLASSES TO REPRESENT A COMPLEX NUMBER

ComplexNumber **ComplexNumber_Polar**

NOW EACH CLASS HAS A COPY CONSTRUCTOR TO CREATE AN OBJECT FROM AN OBJECT OF THE OTHER..

```
explicit ComplexNumber(const ComplexNumber_Polar& polar)
{
  realPart = polar.modulus * cos(polar.argument * 3.1415/180);
  complexPart = polar.modulus * sin(polar.argument * 3.1415/180);

}
```

WAIT A MINUTE? WHAT'S THAT `explicit` KEYWORD THERE FOR?

WAIT A MINUTE? WHAT'S THAT `explicit` KEYWORD THERE FOR?

ITS A WAY TO PREVENT IMPLICIT (AND POSSIBLY WRONG!) CONVERSIONS FROM ONE OBJECT TO ANOTHER

IT APPLIES TO CONSTRUCTORS, AND HELPS MAKE SURE THAT COPY CONSTRUCTORS ARE ONLY USED WHEN THE PROGRAMMER REALLY INTENDS THEM TO

```
ComplexNumber_Polar c(7.0,45);
ComplexNumber f1 = c;
ComplexNumber f = ComplexNumber(c);
```

X

```
ComplexNumber_Polar c(7.0,45);
ComplexNumber f1 = c;
ComplexNumber f = ComplexNumber(c);
```

WON'T WORK IF THE CONSTRUCTOR IS MARKED EXPLICIT

```
Example73.cpp:64:17: error: no viable conversion from 'ComplexNumber_Polar' to 'ComplexNumber'
  ComplexNumber f1 = c;
                ^    ~
Example73.cpp:27:7: note: candidate constructor (the implicit copy constructor) not viable: no known conversion from
      'ComplexNumber_Polar' to 'const ComplexNumber &' for 1st argument
class ComplexNumber
```

```
ComplexNumber_Polar c(7.0,45);
ComplexNumber f1 = c;
ComplexNumber f = ComplexNumber(c);
```

WILL WORK EVEN IF THE CONSTRUCTOR IS EXPLICIT!

SAY WE HAVE 2 CLASSES TO REPRESENT A COMPLEX NUMBER

ComplexNumber **ComplexNumber_Polar**

NOW EACH CLASS HAS A COPY CONSTRUCTOR TO CREATE AN OBJECT FROM AN OBJECT OF THE OTHER..

```
explicit ComplexNumber(const ComplexNumber_Polar& polar)
{
  realPart = polar.modulus * cos(polar.argument * 3.1415/180);
  complexPart = polar.modulus * sin(polar.argument * 3.1415/180);

}
```

WAIT A MINUTE? WHAT'S THAT `explicit` KEYWORD THERE FOR?

NOW, WOULD THIS CODE WORK?

```
void printComplexNumber(ComplexNumber c)
{
  c.print();
}
ComplexNumber_Polar c(7.0,45);
printComplexNumber(c);
```

X

WON'T WORK IF THE CONSTRUCTOR IS MARKED EXPLICIT

```
Example73.cpp:69:3: error: no matching function for call to 'printComplexNumber'
  printComplexNumber(c);
  ^~~~~~~~~~~~~~~~~~
Example73.cpp:56:6: note: candidate function not viable: no known conversion from 'ComplexNumber_Polar' to 'ComplexNumber' for 1st
      argument
void printComplexNumber(ComplexNumber c)
     ^
2 errors generated.
```

NOW, WOULD THIS CODE WORK?

```
void printComplexNumber(ComplexNumber c)
{
  c.print();
}
ComplexNumber_Polar c(7.0,45);
printComplexNumber(c);
```

X

WON'T WORK IF THE CONSTRUCTOR IS MARKED EXPLICIT

```
Example73.cpp:69:3: error: no matching function for call to 'printComplexNumber'
  printComplexNumber(c);
  ^~~~~~~~~~~~~~~~~~
Example73.cpp:56:6: note: candidate function not viable: no known conversion from 'ComplexNumber_Polar' to 'ComplexNumber' for 1st
      argument
void printComplexNumber(ComplexNumber c)
     ^
2 errors generated.
```

NOW, WOULD THIS CODE WORK?

```
void printComplexNumber(ComplexNumber c)
{
  c.print();
}
ComplexNumber_Polar c(7.0,45);
printComplexNumber(c);
```

X

WON'T WORK IF THE CONSTRUCTOR IS MARKED EXPLICIT

```
Example73.cpp:69:3: error: no matching function for call to 'printComplexNumber'
  printComplexNumber(c);
  ^~~~~~~~~~~~~~~~~~
Example73.cpp:56:6: note: candidate function not viable: no known conversion from 'ComplexNumber_Polar' to 'ComplexNumber' for 1st
      argument
void printComplexNumber(ComplexNumber c)
     ^
2 errors generated.
```

NOW, WOULD THIS CODE WORK?

```
void printComplexNumber(ComplexNumber c)
{
  c.print();
}
ComplexNumber_Polar c(7.0,45);
printComplexNumber(c);
```

X

WON'T WORK IF THE CONSTRUCTOR IS MARKED EXPLICIT

```
Example73.cpp:69:3: error: no matching function for call to 'printComplexNumber'
  printComplexNumber(c);
  ^~~~~~~~~~~~~~~~~~
Example73.cpp:56:6: note: candidate function not viable: no known conversion from 'ComplexNumber_Polar' to 'ComplexNumber' for 1st
      argument
void printComplexNumber(ComplexNumber c)
     ^
2 errors generated.
```

NOW, WOULD THIS CODE WORK?

```
void printComplexNumber(ComplexNumber c)
{
  c.print();
}
ComplexNumber_Polar c(7.0,45);
printComplexNumber(c);
```

X

```
Example73.cpp:69:3: error: no matching function for call to 'printComplexNumber'
  printComplexNumber(c);
  ^~~~~~~~~~~~~~~~~~
Example73.cpp:56:6: note: candidate function not viable: no known conversion from 'ComplexNumber_Polar' to 'ComplexNumber' for 1st
      argument
void printComplexNumber(ComplexNumber c)
     ^
2 errors generated.
```

static_cast

TO THE RESCUE!

WON'T WORK IF THE CONSTRUCTOR IS MARKED EXPLICIT

NOW, WOULD THIS CODE WORK?

```
void printComplexNumber(ComplexNumber c)
{
  c.print();
}
ComplexNumber_Polar c(7.0,45);
printComplexNumber(c);
```

X

WON'T WORK IF THE CONSTRUCTOR IS MARKED EXPLICIT

```
Example73.cpp:69:3: error: no matching function for call to 'printComplexNumber'
  printComplexNumber(c);
  ^~~~~~~~~~~~~~~~~~
Example73.cpp:56:6: note: candidate function not viable: no known conversion from 'ComplexNumber_Polar' to 'ComplexNumber' for 1st
      argument
void printComplexNumber(ComplexNumber c)
     ^
2 errors generated.
```

NOW, WOULD THIS CODE WORK?

```
void printComplexNumber(ComplexNumber c)
{
  c.print();
}
ComplexNumber_Polar c(7.0,45);
printComplexNumber(c);
printComplexNumber(static_cast<ComplexNumber>(c));
```

STATIC_CAST WILL FIGURE OUT ANY LEGAL WAY TO MAKE THE CONVERSION HAPPEN.

THIS INCLUDES CALLING EXPLICIT CONSTRUCTORS, OPERATORS, WHATEVER IT TAKES.

JUST REMEMBER THAT STATIC_CAST'S MAGIC IS RESTRICTED TO COMPILE TIME THOUGH (UNLIKE DYNAMIC CAST)