# EXAMPLE 53: UNDERSTAND WHAT VIRTUAL FUNCTIONS ARE

# VIRTUAL FUNCTIONS

ARE AN INCREDIBLY IMPORTANT IDEA IN PROGRAMMING
- ALMOST AS IMPORTANT AS OBJECTS AND CLASSES

YOU SHOULD KNOW - UPFRONT - THAT MOST FUNCTIONS IN C# AND JAVA ARE, BY-DEFAULT, VIRTUAL

THIS IS DIFFERENT FROM C++, WHERE FUNCTIONS ARE VIRTUAL ONLY IF EXPLICITLY MARKED AS SUCH.

# VIRTUAL FUNCTIONS

## SAY YOU HAVE A BASE CLASS (SHAPE) AND A DERIVED CLASS (CIRCLE)

```
class Shape
{
public:
  virtual void print()
  {
    cout << "I am a shape, not sure of what type" << endl;
  }
};

class Circle : public Shape
{
public:
  void print()
  {
    cout << "I am a circle" << endl;
  }
};
```

# SAY YOU HAVE A BASE CLASS (SHAPE) AND A DERIVED CLASS (CIRCLE)

```
class Shape
{
public:
  virtual void print()
  {
    cout << "I am a shape, not sure of what type" << endl;
  }
};

class Circle : public Shape
{
public:
  void print()
  {
    cout << "I am a circle" << endl;
  }
};
```

# SAY YOU HAVE A BASE CLASS (SHAPE) AND A DERIVED CLASS (CIRCLE)

```
class Shape
{
public:
  virtual void print()
  {
    cout << "I am a shape, not sure of what type" << endl;
  }
};
```

```
class Circle : public Shape
{
public:
  void print()
  {
    cout << "I am a circle" << endl;
  }
};
```

THE CLASS CIRCLE INHERITS FROM SHAPE

# SAY YOU HAVE A BASE CLASS (SHAPE) AND A DERIVED CLASS (CIRCLE)

```
class Shape
{
public:
  virtual void print()
  {
    cout << "I am a shape, not sure of what type" << endl;
  }
};

class Circle : public Shape
{
public:
  void print()
  {
    cout << "I am a circle" << endl;
  }
};
```

BOTH CLASSES HAVE A FUNCTION WITH THE SAME NAME AND SIGNATURE..

# SAY YOU HAVE A BASE CLASS (SHAPE) AND A DERIVED CLASS (CIRCLE)

```
class Shape
{
public:
  virtual void print()
  {
    cout << "I am a shape, not sure of what type" << endl;
  }
};

class Circle : public Shape
{
public:
  void print()
  {
    cout << "I am a circle" << endl;
  }
};
```

THE ONLY DIFFERENCE IS THAT THE BASE CLASS VERSION IS MARKED VIRTUAL

HERE DOES NOT MATTER IF THE DERIVED CLASS VERSION IS MARKED VIRTUAL OR NOT)

# SAY YOU HAVE A BASE CLASS (SHAPE) AND A DERIVED CLASS (CIRCLE)

```
class Shape
{
public:
  virtual void print()
  {
    cout << "I am a shape, not sure of what type" << endl;
  }
};

class Circle : public Shape
{
public:
  void print()
  {
    cout << "I am a circle" << endl;
  }
};
```

THE FUNCTIONS PRINT DIFFERENT MESSAGES..

SAY YOU HAVE A BASE CLASS (SHAPE) AND A DERIVED CLASS (CIRCLE)

NOW, LET'S SAY YOU HAVE A SHAPE POINTER (OR REFERENCE) VARIABLE..

..CONTAINING A POINTER (OR REFERENCE) TO A CIRCLE

..AND YOU CALL THAT VIRTUAL METHOD ON THIS VARIABLE

SAY YOU HAVE A BASE CLASS (SHAPE) AND A DERIVED CLASS (CIRCLE)

NOW, LET'S SAY YOU HAVE A SHAPE POINTER (OR REFERENCE) VARIABLE..

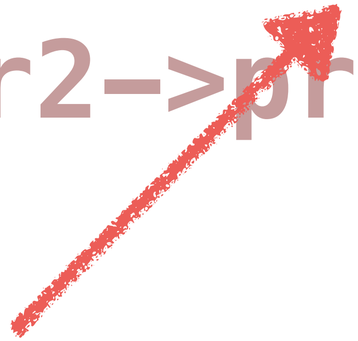
..CONTAINING A POINTER (OR REFERENCE) TO A CIRCLE

..AND YOU CALL THAT VIRTUAL METHOD ON THIS VARIABLE

```
Shape *sPtr2 = new Circle();
sPtr2->print();
```

# WHICH VERSION OF THE FUNCTION WILL BE CALLED?

```
Shape *sPtr2 = new Circle();
sPtr2->print();
```



A BASE CLASS POINTER

POINTING TO A DERIVED CLASS OBJECT

```
Shape *sPtr2 = new Circle();
sPtr2->print();
```

THIS IS TOTALLY VALID IN C++

IN FACT THIS IS A GOOD WAY TO WRITE CODE BECAUSE YOU CAN HAVE THE SAME POINTER POINT TO ANY SHAPE

# WHICH VERSION OF THE FUNCTION WILL BE CALLED?

```
Shape *sPtr2 = new Circle();
sPtr2->print();
```

```
class Shape
{
public:
  virtual void print()
  {
    cout << "I am a shape, not sure of what type" << endl;
  }
};

class Circle : public Shape
{
public:
  void print()
  {
    cout << "I am a circle" << endl;
  }
};
```

# WHICH VERSION OF THE FUNCTION WILL BE CALLED?

```
class Circle : public Shape
{
public:
   void print()
   {
      cout << "I am a circle" << endl;
   }
};
```

```
Shape *sPtr2 = new Circle();
sPtr2->print();
```

ANSWER: THE DERIVED CLASS VERSION - ONLY BECAUSE THE FUNCTION WAS MARKED VIRTUAL

# WHICH VERSION OF THE FUNCTION WILL BE CALLED?

```
Shape *sPtr2 = new Circle();
sPtr2->print();
```

"I am a circle"

## ANSWER: THE DERIVED CLASS VERSION - ONLY BECAUSE THE FUNCTION WAS MARKED VIRTUAL

# WHICH VERSION OF THE FUNCTION WILL BE CALLED?

```
Shape *sPtr2 = new Circle();
sPtr2->print();
```

"I am a circle"

ANSWER: THE DERIVED CLASS VERSION - ONLY BECAUSE THE FUNCTION WAS MARKED VIRTUAL

ERR..OKEY-DOKEY..

REMIND ME WHY I SHOULD BE EXCITED ABOUT THIS, PLEASE?

REMIND ME WHY I SHOULD BE EXCITED ABOUT THIS, PLEASE?

THIS IS INCREDIBLY EXCITING!

BECAUSE!

THE C++ COMPILER FIGURED OUT WHICH VERSION TO CALL AT RUNTIME, NOT AT COMPILE-TIME

# THE C++ COMPILER FIGURED OUT WHICH VERSION TO CALL AT RUNTIME, NOT AT COMPILE-TIME

```
Shape *sPtr2 = new Circle();
sPtr2->print();
```

"I am a circle"

SO FAR, ALL FUNCTION CALLS IN C/C++ WE HAVE SEEN HAVE BEEN STATICALLY BOUND

THIS CALL IS DYNAMICALLY BOUND

REMIND ME WHY I SHOULD BE EXCITED ABOUT THIS, PLEASE?

REMIND ME WHY I SHOULD BE EXCITED ABOUT THIS, PLEASE?

# STATICALLY BOUND

FUNCTION CALLS ARE "RESOLVED" AT COMPILE-TIME

ALL NON-VIRTUAL CALLS IN C/C++ ARE STATICALLY BOUND

# DYNAMICALLY BOUND

FUNCTION CALLS ARE "RESOLVED" AT RUN-TIME

ONLY VIRTUAL CALLS IN C++ ARE DYNAMICALLY BOUND

ALMOST ALL PUBLIC FUNCTION CALLS IN JAVA ARE DYNAMICALLY BOUND

REMIND ME WHY I SHOULD BE EXCITED ABOUT THIS, PLEASE?

DYNAMIC BINDING HAS A DRAMATIC EFFECT ON THE WAY SOFTWARE IS DESIGNED AND WRITTEN

IT ALLOWS SOMETHING KNOWN AS RUNTIME POLYMORPHISM

YOU WRITE CODE WITH A LOT OF VARIABLES THAT ARE POINTERS TO BASE CLASSES -

AND KEEP ASSIGNING THOSE POINTERS TO OBJECTS OF THE RIGHT DERIVED CLASSES..

AND "IT JUST WORKS" BECAUSE OF RUNTIME POLYMORPHISM

# RUNTIME POLYMORPHISM

REMIND ME WHY I SHOULD BE EXCITED ABOUT THIS, PLEASE?

IS ONLY POSSIBLE BECAUSE OF

## VIRTUAL FUNCTIONS

```
  {
    cout << "I am a shape, not sure of what type" << endl;
  }
};

class Circle : public Shape
{
public:
  void print()
  {
    cout << "I am a circle" << endl;
  }
};
```

WHICH VERSION OF THE FUNCTION WILL BE CALLED?

REMIND ME WHY I SHOULD BE EXCITED ABOUT THIS, PLEASE?

ANSWER: THE DERIVED CLASS VERSION - ONLY BECAUSE THE FUNCTION WAS MARKED VIRTUAL

ERR..OKEY-DOKEY..

THAT'S WHY :-)

SO - WILL THIS DYNAMIC BINDING HAPPEN ONLY FOR BASE CLASS POINTERS HOLDING DERIVED CLASS OBJECTS?

YEP! POINTERS, AND REFERENCES BOTH. BUT NOT ORDINARY VARIABLES.

DOES THAT NOT EXCLUDE MOST VARIABLES THAT WE ACTUALLY USE?

ACTUALLY NOT - MOST PRODUCTION C++ CODE WILL USE POINTERS AND REFERENCES

# WHY IS VIRTUAL THE DEFAULT IN JAVA, BUT NOT IN C++?

VIRTUAL FUNCTIONS INCUR A PERFORMANCE HIT

THIS HIT IS BOTH IN SPEED AND IN MEMORY USAGE

LET'S SEE WHY.

# VIRTUAL FUNCTIONS INCUR A PERFORMANCE HIT

BOTH IN SPEED AND IN MEMORY USAGE

THE C++ COMPILER ADDS A HIDDEN MEMBER VARIABLE TO ANY OBJECT WITH ANY VIRTUAL FUNCTIONS

THIS HIDDEN MEMBER VARIABLE (1 PER OBJECT) IS CALLED THE `_vptr`

AND IT POINTS TO A TABLE (1 PER CLASS) CALLED THE `vTable` THAT HOLDS THE CORRECT VERSION OF EACH VIRTUAL FUNCTIONS TO CALL

# VIRTUAL FUNCTIONS INCUR A PERFORMANCE HIT

## BOTH IN SPEED AND IN MEMORY USAGE

THE SPEED PERFORMANCE HIT IS BECAUSE THE LOOKUP FROM `_vptr` TO `vTable` TO FUNCTION HAPPENS AT RUNTIME

THE MEMORY USAGE PERFORMANCE HIT IS BECAUSE EACH OBJECT CARRIES AROUND THIS ADDITIONAL MEMBER VARIABLE (WE WILL VERIFY THIS IN A BIT:-))

# 4 CLASSES, 2 BASE, 2 DERIVED. ONLY ONE BASE HAS A VIRTUAL FUNCTION

```
class Shape_NonVirtual
{
private:
  string shapeType;
public:
  void print()
  {
    cout << "I am a shape, not sure of what type" << endl;
  }
};
```

```
class Shape_Virtual
{
private:
  string shapeType;
public:
  virtual void print()
  {
    cout << "I am a shape, not sure of what type" << endl;
  }
};
```

```
class Circle_NonVirtual : public Shape_NonVirtual
{
public:
  void print()
  {
    cout << "I am a circle" << endl;
  }
};
```

```
class Circle_Virtual : public Shape_Virtual
{
public:
  void print()
  {
    cout << "I am a circle" << endl;
  }
};
```

# OTHERWISE IDENTICAL

# 4 CLASSES, 2 BASE, 2 DERIVED. ONLY ONE BASE HAS A VIRTUAL FUNCTION

```
class Shape_NonVirtual
{
private:
  string shapeType;
public:
  void print()
  {
    cout << "I am a shape, not sure of what type" << endl;
  }
};
```

```
class Shape_Virtual
{
private:
  string shapeType;
public:
  virtual void print()
  {
    cout << "I am a shape, not sure of what type" << endl;
  }
};
```

```
class Circle_NonVirtual : public Shape_NonVirtual
{
public:
  void print()
  {
    cout << "I am a circle" << endl;
  }
};
```

```
class Circle_Virtual : public Shape_Virtual
{
public:
  void print()
  {
    cout << "I am a circle" << endl;
  }
};
```

# 4 CLASSES, 2 BASE, 2 DERIVED. ONLY ONE BASE HAS A VIRTUAL FUNCTION

```
class Shape_NonVirtual
{
private:
  string shapeType;
public:
  void print()
  {
    cout << "I am a shape, not sure what type" << endl;
  }
};
```

```
class Shape_Virtual
{
private:
  string shapeType;
public:
  virtual void print()
  {
    cout << "I am a shape, not sure what type" << endl;
  }
};
```

LET'S DO A BUNCH OF LITTLE DRILLS TO MAKE SURE WE UNDERSTAND THIS OH-SO-EXCITING CONCEPT

```
class Circle_NonVirtual : public Shape_NonVirtual
{
public:
  void print()
  {
    cout << "I am a circle" << endl;
  }
};
```

```
class Circle_Virtual : public Shape_Virtual
{
public:
  void print()
  {
    cout << "I am a circle" << endl;
  }
};
```

# WHAT'S THE OUTPUT?

```
Shape_Virtual s1;
Shape_NonVirtual s2;

cout << "Size of object with vPtr is: " << sizeof(s1) << endl;
cout << "Size of object without vPtr is: " << sizeof(s2) << endl;
```

```
Size of object with vPtr is: 32
Size of object without vPtr is: 24
```

# THE DIFFERENCE OF 8 BYTES IS BECAUSE OF THE `_vptr`

# WHAT'S THE OUTPUT?

```
Circle_Virtual c1;
Circle_NonVirtual c2;

cout << "Size of object with vPtr is: " << sizeof(c1) << endl;
cout << "Size of object without vPtr is: " << sizeof(c2) << endl;
```

```
Size of object with vPtr is: 32
Size of object without vPtr is: 24
```

AGAIN, THE DIFFERENCE OF 8 BYTES IS BECAUSE OF THE `_vptr`

# WHAT'S THE OUTPUT?

```
Circle_Virtual c1;
Circle_NonVirtual c2;
cout << "Will the circles know their types?" << endl;
cout << "Circle with the virtual function says:" << endl;
c1.print();
cout << "Circle without the virtual function says:" << endl;
c2.print();
```

```
Will the circles know their types?
Circle with the virtual function says:
I am a circle
Circle without the virtual function says:
I am a circle
```

WITH ORDINARY (NON-POINTER, NON-REFERENCE) VARIABLES, NO DYNAMIC DISPATCH BUT THEY KNOW THEIR TYPES EVEN VIA STATIC DISPATCH

# WHAT'S THE OUTPUT?

```
cout << "Now assign the circles to their base class types." << endl;
Shape_Virtual s1 = c1;
Shape_NonVirtual s2 = c2;

cout << "NOW Will the circles know their types?" << endl;
s1.print();
s2.print();
```

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Now assign the circles to their base class types.
NOW Will the circles know their types?
I am a shape, not sure of what type
I am a shape, not sure of what type
```

WITH ORDINARY VARIABLES (NON-POINTER, NON-REFERENCE),NO VIRTUAL FUNCTION EVER WILL BE CALLED

# WHAT'S THE OUTPUT?

```
Shape_Virtual *sPtr1 = new Circle_Virtual();
Shape_NonVirtual *sPtr2 = new Circle_NonVirtual();

cout << "NOW Will the circles know their types?" << endl;
sPtr1->print();
sPtr2->print();
```

```
NOW Will the circles know their types?
I am a circle
I am a shape, not sure of what type
```

# AHA! FINALLY! VIRTUAL FUNCTION MAGIC AT WORK!

# WHAT'S THE OUTPUT?

```
Shape_Virtual& sRef1 = *sPtr1;
Shape_NonVirtual& sRef2 = *sPtr2;

cout << "NOW Will the circles know their types?" << endl;
sRef1.print();
sRef2.print();
```

```
NOW Will the circles know their types?
I am a circle
I am a shape, not sure of what type
```

# REFERENCES ARE POINTERS TOO - VIRTUAL FUNCTION MAGIC!