EXAMPLE 57: UNDERSTAND AND FEAR OBJECT SLICING - AND PASS FUNCTION PARAMETERS BY CONST REFERENCE, NOT BY VALUE

EXAMPLE 37

PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

RECAP

EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

WHEN AN OBJECT IS PASSED BY VALUE TO A FUNCTION AS A FUNCTION ARGUMENT, A TEMPORARY VARIABLE IS CREATED

THIS TEMPORARY VARIABLE IS CONSTRUCTED (USING THE COPY CONSTRUCTOR), AND THEN MUST BE DESTRUCTED TOO (VIA THE DESTRUCTOR)

RECAP

EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

WHEN AN OBJECT IS PASSED BY VALUE TO A FUNCTION AS A FUNCTION ARGUMENT, A TEMPORARY VARIABLE IS CREATED

THIS TEMPORARY VARIABLE IS CONSTRUCTED (USING THE COPY CONSTRUCTOR), AND THEN MUST BE DESTRUCTED TOO (VIA THE DESTRUCTOR)

ALL OF THESE CONSTRUCTOR AND DESTRUCTOR CALLS ADD UP (REMEMBER THEY WILL CASCADE INTO THE MEMBER VARIABLES OF THAT OBJECT!)

RECAP

EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

WHEN AN OBJECT IS PASSED BY VALUE TO A FUNCTION AS A FUNCTION ARGUMENT, A TEMPORARY VARIABLE IS CREATED

THIS TEMPORARY VARIABLE IS CONSTRUCTED (USING THE COPY CONSTRUCTOR), AND THEN MUST BE DESTRUCTED TOO (VIA THE DESTRUCTOR)

ALL OF THESE CONSTRUCTOR AND DESTRUCTOR CALLS ADD UP (REMEMBER THEY WILL CASCADE INTO THE MEMBER VARIABLES OF THAT OBJECT!)

IN A PERFORMANCE-SENSITIVE APPLICATION THIS WILL BECOME UNACCEPTABLE

RECAP

EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

IN A PERFORMANCE-SENSITIVE APPLICATION
THIS WILL BECOME UNACCEPTABLE

SLICING

THERE IS ANOTHER REASON TO PREFER PASSING FUNCTION ARGUMENTS AS CONST REFERENCES RATHER THAN BY VALUE, CALLED **SLICING**

RECAP

EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

IN A PERFORMANCE-SENSITIVE APPLICATION
THIS WILL BECOME UNACCEPTABLE

SLICING

THERE IS ANOTHER REASON TO PREFER PASSING FUNCTION ARGUMENTS
AS CONST REFERENCES RATHER THAN BY VALUE, CALLED SLICING

SLICING HAS TO DO WITH INHERITANCE, SO WE WILL GET TO IT LATER

RECAP

SLICING

IF A FUNCTION HAS PARAMETER OF TYPE SHAPE (BASE CLASS)

```
void callPrint_Sliced(Shape s)
{
  cout << endl << "Slicing is about to occur" << endl;
  s.print();
}
```

SLICING

IF A FUNCTION HAS PARAMETER OF TYPE SHAPE (BASE CLASS)

```
void callPrint_Sliced(Shape s)
{
  cout << endl << "Slicing is about to occur" << endl;
  s.print();
}
```

NOTE! THIS PARAMETER IS NOT A POINTER OR A REFERENCE, I.E. IT IS PASSED BY VALUE

SLICING

IF A FUNCTION HAS PARAMETER OF TYPE SHAPE (BASE CLASS)

```
void callPrint_Sliced(Shape s)
{
    cout << endl << "Slicing is about to occur" << endl;
    s.Print();
}
```

NOTE! THIS PARAMETER IS NOT A POINTER OR A REFERENCE, I.E. IT IS PASSED BY VALUE

AND YOU PASS IN A RECTANGLE (DERIVED CLASS OBJECT)

SLICING

IF A FUNCTION HAS PARAMETER OF TYPE SHAPE (BASE CLASS)

AND YOU PASS IN A RECTANGLE (DERIVED CLASS OBJECT)

THE C++ COMPILER WILL SLICE YOUR RECTANGLE INTO A SHAPE BEFORE COPYING (PASSING BY VALUE) TO THE FUNCTION

SLICING

IF A FUNCTION HAS PARAMETER OF TYPE SHAPE (BASE CLASS)

AND YOU PASS IN A RECTANGLE (DERIVED CLASS OBJECT)

THE C++ COMPILER WILL SLICE YOUR RECTANGLE INTO A SHAPE BEFORE COPYING (PASSING BY VALUE) TO THE FUNCTION

IT SOUNDS PAINFUL - AND IT IS - BECAUSE THE SLICED OBJECT WILL NOT DO AS YOU EXPECTED IT TO!

SLICING

IF A FUNCTION HAS PARAMETER OF TYPE SHAPE (BASE CLASS)

AND YOU PASS IN A RECTANGLE (DERIVED CLASS OBJECT)

THE C++ COMPILER WILL SLICE YOUR RECTANGLE INTO A SHAPE BEFORE COPYING (PASSING BY VALUE) TO THE FUNCTION

IT SOUNDS PAINFUL - AND IT IS - BECAUSE THE SLICED OBJECT WILL NOT DO AS YOU EXPECTED IT TO!

TO AVOID SLICING, PASS FUNCTION ARGUMENTS AS CONST REFERENCES!

TO AVOID SLICING, PASS FUNCTION ARGUMENTS AS CONST REFERENCES!

BASE CLASS - SHAPE

```
class Shape
{
private:
public:
  string shapeType;
  Shape()
  {
    shapeType = "Unknown";
    cout << "Inside the Shape constructor" << endl;
  }

  ~Shape()
  {
    cout << "Inside the Shape destructor" << endl;
  }
  virtual void print() const
  {
    cout << "I am a shape, and my size is " << sizeof(*this) << " bytes " << endl;
  }
};
```

DERIVED CLASS - RECTANGLE

```
class Rectangle : public Shape
{
public:
  int rectangle_length;
  int rectangle_breadth;
  Rectangle()
  {
    cout << "Inside the Rectangle constructor" << endl;
  }
  ~Rectangle()
  {
    cout << "Inside the Rectangle destructor" << endl;
  }
  virtual void print() const
  {
    cout << "I am a rectangle,, and my size is " << sizeof(*this) << " bytes " << endl;
  }
};
```

BAD FUNCTION - SLICING HAPPENS

```
void callPrint_Sliced(Shape s)
{
  cout << endl << "Slicing is about to occur" << endl;
  s.print();
}
```

GOOD FUNCTION - NO SLICING HAPPENS

```
void callPrint_NonSliced(const Shape& s)
{
  cout << endl << "No slicing can occur here" << endl;
  s.print();
}
```

TO AVOID SLICING, PASS FUNCTION ARGUMENTS AS CONST REFERENCES!

BAD FUNCTION - SLICING HAPPENS

```
void callPrint_Sliced(Shape s)
{
  cout << endl << "Slicing is about to occur" << endl;
  s.print();
}
```

GOOD FUNCTION - NO SLICING HAPPENS

```
void callPrint_NonSliced(const Shape& s)
{
  cout << endl << "No slicing can occur here" << endl;
  s.print();
}
```

CALL THE FUNCTIONS, AND SEE WHAT HAPPENS!

```
Rectangle r;
callPrint_Sliced(r);
callPrint_NonSliced(r);
```

CALL THE FUNCTIONS, AND SEE WHAT HAPPENS!

```
Rectangle r;
callPrint_Sliced(r);
callPrint_NonSliced(r);
```

```
Slicing is about to occur
I am a shape, and my size is 32 bytes
Inside the Shape destructor

No slicing can occur here
I am a rectangle,, and my size is 40 bytes
Inside the Rectangle destructor
Inside the Shape destructor
```

CALL THE FUNCTIONS, AND SEE WHAT HAPPENS!

```
Rectangle r;
callPrint_Sliced(r);
callPrint_NonSliced(r);
```

```
Slicing is about to occur
I am a shape, and my size is 32 bytes
Inside the Shape destructor
```

THE SLICED OBJECT LOSES TRACK OF ITS SIZE, AND OF ALL VIRTUAL FUNCTIONS

CALL THE FUNCTIONS, AND SEE WHAT HAPPENS!

```
Rectangle r;
callPrint_Sliced(r);
callPrint_NonSliced(r);
```

```
Slicing is about to occur
I am a shape, and my size is 32 bytes
Inside the Shape destructor
```

AND WHAT'S WORSE - ONLY THE SHAPE DESTRUCTOR IS CALLED ON THE COPY THAT WAS CREATED TO PASS-BY-VALUE

CALL THE FUNCTIONS, AND SEE WHAT HAPPENS!

```
Rectangle r;
callPrint_Sliced(r);
callPrint_NonSliced(r);
```

```
Slicing is about to occur
I am a shape, and my size is 32 bytes
Inside the Shape destructor
```

THIS IS TERRIBLE! MEMORY LEAK OF ALL RESOURCES IN THE RECTANGLE PORTION!

CALL THE FUNCTIONS, AND SEE WHAT HAPPENS!

```
Rectangle r;
callPrint_Sliced(r);
callPrint_NonSliced(r);
```

NONE OF THESE ISSUES OCCUR WITH THE NON-SLICED VERSION

```
No slicing can occur here
I am a rectangle,, and my size is 40 bytes
Inside the Rectangle destructor
Inside the Shape destructor
```