Compiling: Additional Topics

CSE 410 22wi

Lecture 05

Lecture Oveview

1. Review: compiling simple statements and procedures
2. Classes and objects
3. Class Hierarchies
4. Static Typing and Dispatch
5. Dynamic Dispatch
6. Compile Time Generation of Source Code
7. Generics / Parameterized Types

1. Review: Compiling a C Program

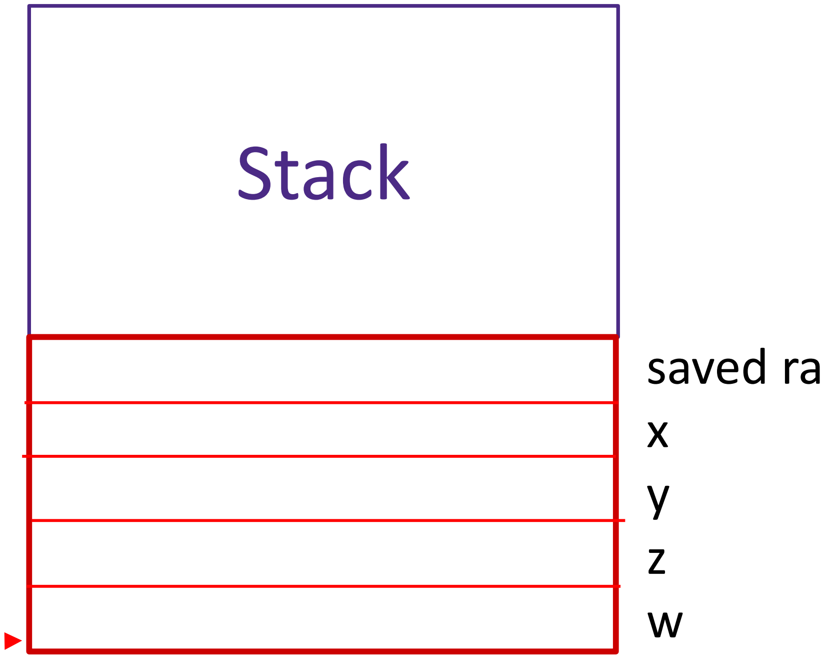
```c
int  val = 10238;
int main(int argc, char *argv[])
{
   int i;
   for (i = 2; i<=val/2; i++) {
      if ((val/i)*i == val){
         printf("%d\n", i);
      }
   }
   return 0;
}
```

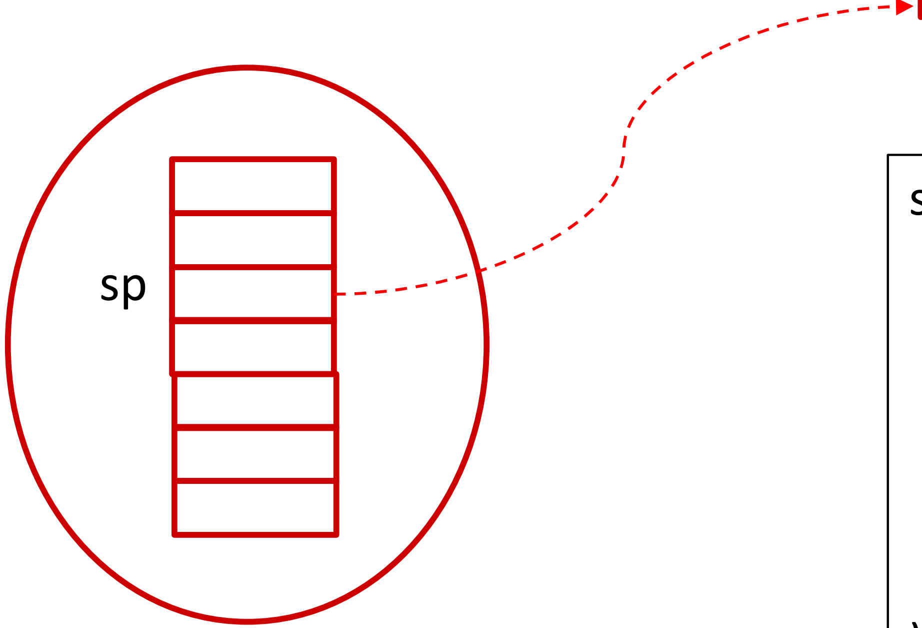
```
    lw    x7, val
    lw    x9, i
    div   x10, x9, x7
    mul   x10, x10, x9
    bne   x10, x7, else
     <body code>
else:
```

1. Review: Call / Return

```
val = sub(2);
```

```
int sub(int w) {
      int x, y, z;
       ...
       return x;
}
```





```
sub:     addi   sp, sp, -20
          sw       a0, 0(sp)
          sw       ra, -16(sp)
          ...
          lw     ra, -16(sp)
          addi sp, sp, 20
          jr       ra
)
```

1. Summary

- At compile time (statically), the compiler
  - Figures out where space will be allocated for variables
    - In the data section or on the stack
    - A variable's location is typically an offset from some base register
    - For example, local variables are at statically determined offsets from the stack pointer register
    - (There is an analogous pointer register that points at the data section, called register gp. When programs are bigger than one file, and multiple files contain .data sections, it isn't possible to determine offsets from gp as each file is compiled. Instead, there is an additional build step required, called *linking*.)
  - Turns expessions in the source language into a sequence of assembler instructions
    - This requires that the compiler make decisions about what to use registers for at each point in execution
  - Turns control flow into branches, including use of the jal instruction for the call-return semantics of subroutines

2. What About Code with Classes/Objects?

- How are machine instructions used to create classes and objects?
- Q: Is any additional functionality required of the ISA (hardware) to support object oriented programming?
  - A: No

- What about class hierarchies? How does that work?

2. Sample Object Oriented Program

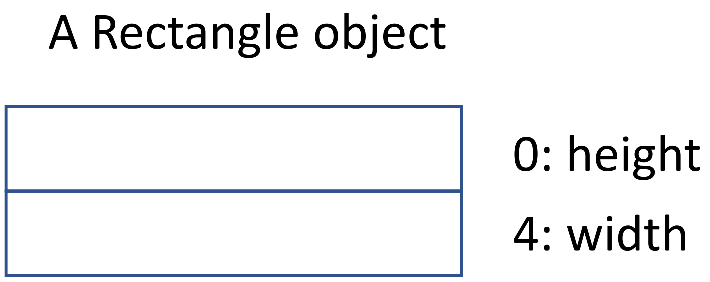
```
class Rectangle {
public:
  Rectangle(int height, int width);
  int getArea();
private:
  int   height;
  int   width;
};

Rectangle::Rectangle(int height, int width) {
  this->height = height;
  this->width = width;
}

int Rectangle::getArea() {
  return height * width;
}
```

```
int main(int argc, char *argv[]) {
  Rectangle plywood(4,8);

  cout << "The rectangle's area is "
         << plywood.getArea()
         << " sq. ft."
         << endl;

  return 0;
}
```

This is C++, but we mean to talk about general ideas rather than specifics of C++

2. What Are Objects?

```
class Rectangle {
public:
  Rectangle(int height, int width);
  int getArea();
private:
  int   height;
  int   width;
};
```



- Looking at the class declaration, the compiler understands that every rectangle object will take two words of storage at runtime
- It doesn’t know where in memory those two words will be allocated at run time for any object
- But it does decide (statically) how to use the two words, wherever they are: the first word is for instance variable “height” and the second for “width”

2. Object as Local Variable

```
int main(int argc, char *argv[]) {
  Rectangle plywood(4,8);

  cout << "The rectangle's area is "
         << plywood.getArea()
         << " sq. ft."
         << endl;

  return 0;
}
```

- plywood is a local variable
- So, the compiler includes its size (two ints == 8 bytes) when calculating how much stack space to allocate on entry to main
- The compiler chooses offset into that space to hold plywood.
- Suppose that offset were 24, meaning that plywood is the eight bytes located at 24(sp)
- Then plywood.height would be at 24(sp) (offset 0 in plywood) and plywood.width would be at 28(sp) (offset 4 in plywood)

This is somewhat C++ specific – Java operates differently

2. Constructing an Object

```
int main(int argc, char *argv[]) {
  Rectangle plywood(4,8);

  cout << "The rectangle's area is "
         << plywood.getArea()
         << " sq. ft."
         << endl;

  return 0;
}
```

- The language requires that a constructor be called when an object is created
- So, the compiler generates a call to the constructor method as part of the initial instructions executed on entry to main
- At the language level, the constructor is a class method
- In assembler, the constructor is just a procedure. The compiler generates a label for that procedure and puts a jal instruction into main to call it.
- At this level, there is nothing new about it – class methods are just subroutines

2. Static Type Checking

```
class Rectangle {
public:
  Rectangle(int height, int width);
  int getArea();
private:
  int   height;
  int   width;
};

Rectangle::Rectangle(int height, int width) {
  this->height = height;
  this->width = width;
}

int Rectangle::getArea() {
  return height * width;
}
```

```
int main(int argc, char *argv[]) {
  Rectangle plywood(4,8);

  plywood.print();

  return 0;
}
```

This is a type error. The compiler realizes that at compile time because it sees (a) plywood is a Rectangle, and (b) Rectangles don't have a print() method

3. Class Hierarchies

```
class Shape {
public:
 Shape();
 int getArea();
protected:
  int id;
};

class Rectangle : public Shape {
  Rectangle(height, width);
  int getArea();
private:
  int  height, width;
};

class Circle: public Shape {
   Circle(int diameter);
   int getArea(); // int return type for simplicity
private:
  int diameter;
};
```

- Here Shape is a base class
- Both Rectangle and Circle are subclasses
- A Rectangle *is* a Shape
- A Circle *is* a Shape

- A Rectangle object requires 3 words of memory: height, width, and id
- A Circle object requires two words of memory: diameter and id

4. Static Typing and Dispatch

```
int main(int argc, char *argv[]) {
  Rectangle r(2,4);
  Circle c(1);

   cout << r.getArea() << endl;
   cout << c.getArea() << endl;

  return 0;
}
```

- "Type checking" is verifying that the code "makes sense"
  - *Does variable r have a method named getArea()?*
- Type checking can be done statically
  - At compile time
- Advantages:
  - No run time overhead to determine or verify types
  - Compiler can generate code at compile time that does exactly what is needed
- Disadvantages:
  - Can be somewhat restrictive, as the language has to limit what the programmer can write to things that can be statically checked

4. Static Typing and Dispatch

```
int main(int argc, char *argv[]) {
  Rectangle r(2,4);
  Circle c(1);

   cout << r.getArea() << endl;
   cout << c.getArea() << endl;

  return 0;
}
```

- "dispatch" means transferring control to the proper subroutine
- How does compiler know to call Rectangle.getArea() for r and Circle.getArea() for c?
- Simple: the programmer told it r is a rectangle and c is a circle.

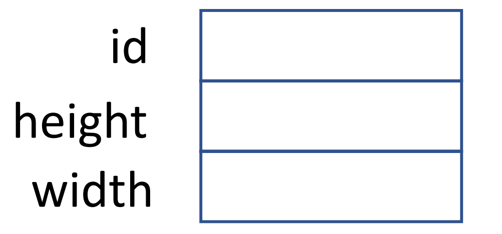
- The fact that two different methods are called "getArea" means the compiler cannot use that name for them in the assembler code. Instead it creates names something like Circle-getArea and Rectangle-getArea and uses those names in assembler

5. Dynamic Dispatch

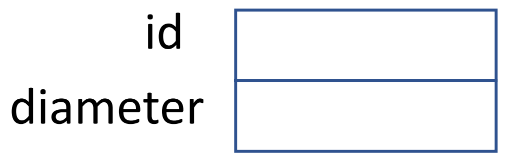
```
Shape larger(Shape A, Shape B) {
  if ( A.getArea() > B.getArea() )
      return A;
  return B;
}
```

- When you look at this code, can you decide whether A.getArea() should invoke Rectangle.getArea() or Circle.getArea()?
- Neither can the compiler
- The decision can’t be made statically, because we don’t know what types of objects will be passed as arguments
- Instead, the decision has to be made dynamically – at run time, when we have objects we can examine to determine their types

5. Dynamic Dispatch Implementation Strategy

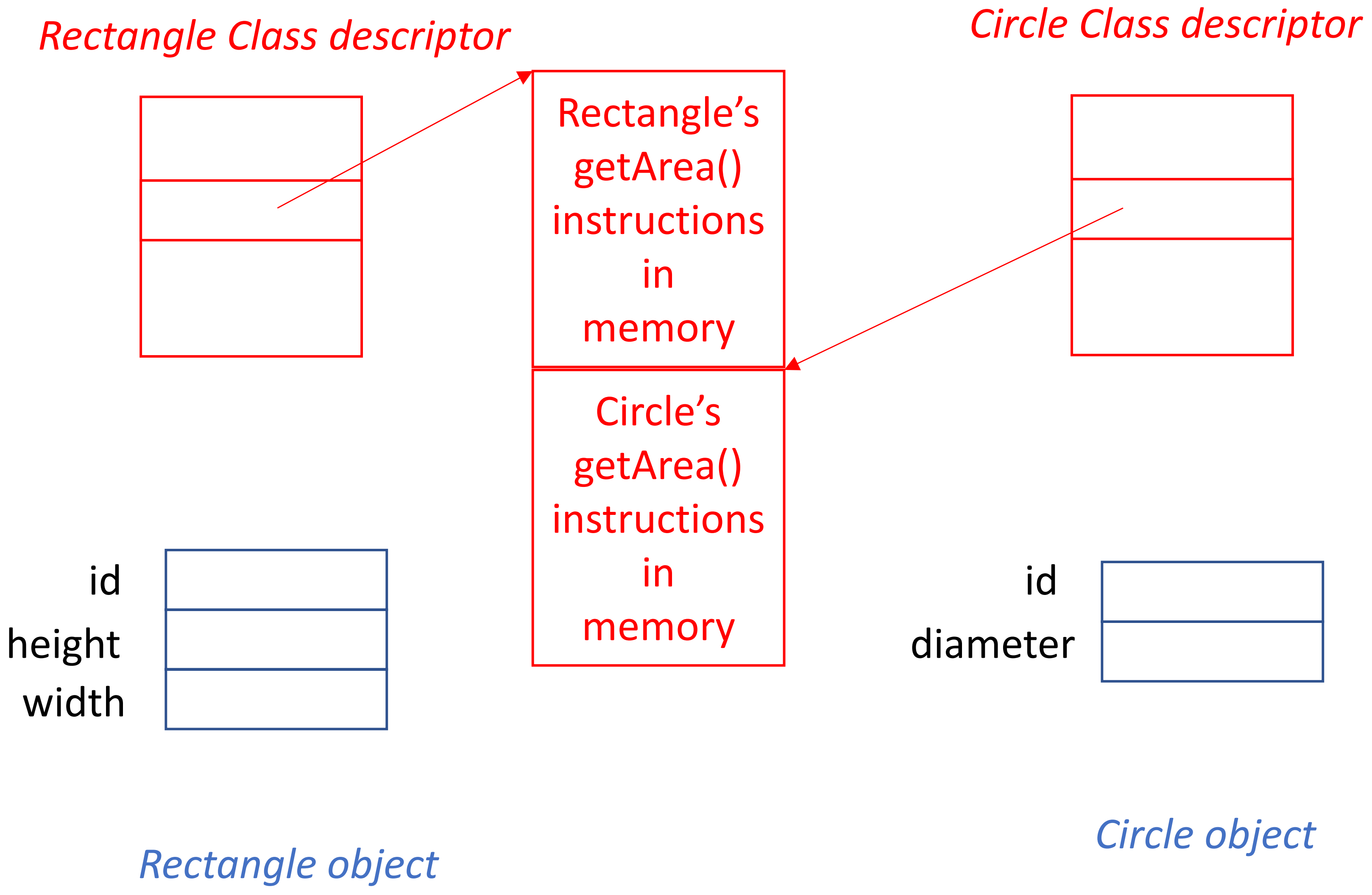


Rectangle object

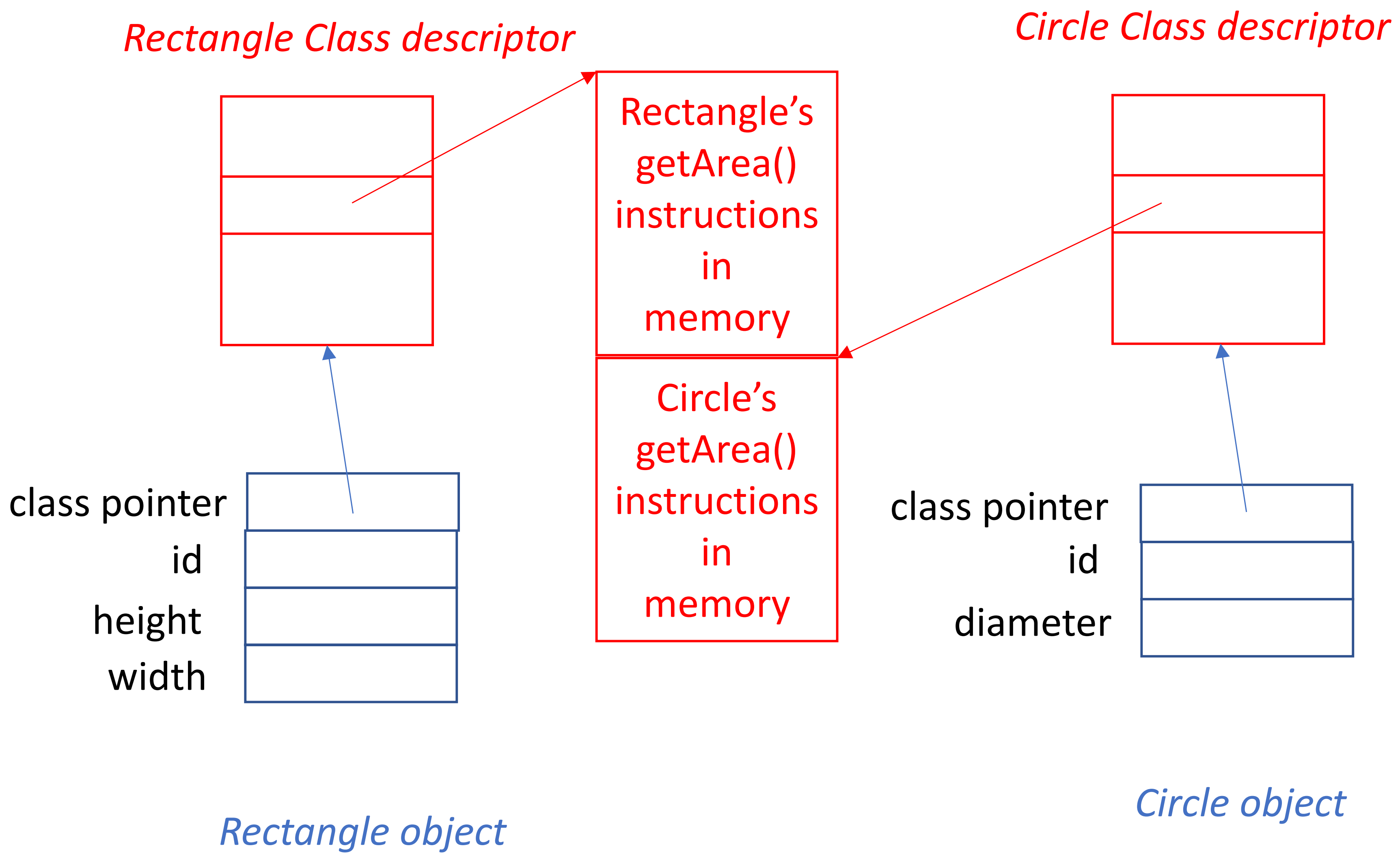


Circle object

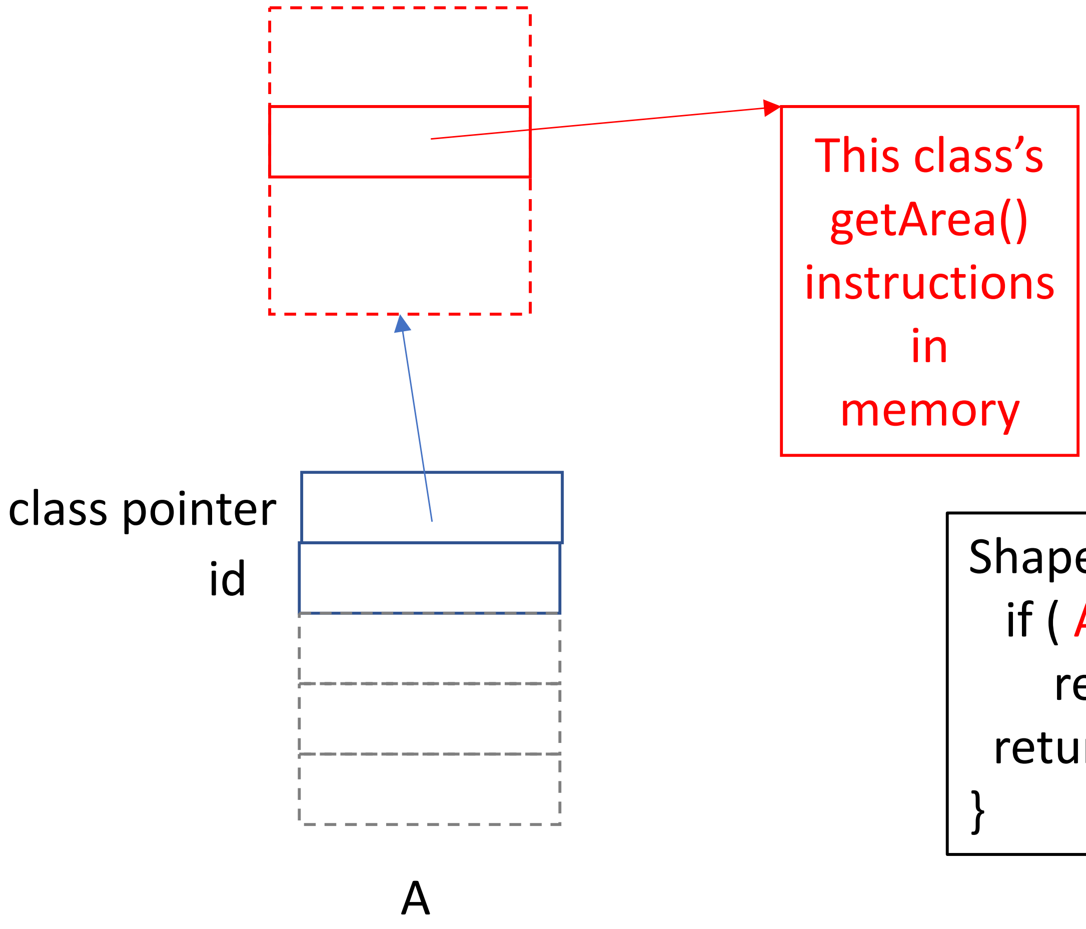
5. Dynamic Dispatch Implementation Strategy


Rectangle Class descriptor
Circle Class descriptor
Rectangle's getArea() instructions in memory
Circle's getArea() instructions in memory
id
height
width
id
diameter
Rectangle object
Circle object

5. Dynamic Dispatch Implementation Strategy


Rectangle Class descriptor
Circle Class descriptor
Rectangle's getArea() instructions in memory
Circle's getArea() instructions in memory
class pointer
id
height
width
class pointer
id
diameter
Rectangle object
Circle object

5. Dynamic Dispatch Implementation Strategy



```
Shape larger(Shape A, Shape B) {
  if ( A.getArea() > B.getArea() )
      return A;
  return B;
}
```

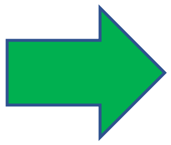
6. Compile Time Generation of Source Code

- Again, the syntax is language (C) specific, but the ideas are general

- Key idea: do source-to-source transformation at compile time
- In C’s case, perform “macro processing” at compile time
  - Do replacement of one string with another string everywhere in the source code before compiling
  - Compile what results from doing the substitution

6. Compile Time Generation of Source Code

```
#define N 100
int main(int argc, char *argv[]) {
   int i;
   int myArray[N};
   for (i=0; i<N; i++)
      myArray[i] = i;
   someSubroutine(myArray);
   return 0;
```



```

int main(int argc, char *argv[]) {
   int i;
   int myArray[100};
   for (i=0; i<100; i++)
      myArray[i] = i;
   someSubroutine(myArray);
   return 0;
```

- N is not a variable
- N is a compile time string (in this case with value “100”)
- Before compiling, the compiler replaces instances of “N” in the source with “100”

6. Compile Time Generation of Source Code

```
#define N oops
int main(int argc, char *argv[]) {
   int i;
   int myArray[N};
   for (i=0; i<N; i++)
      myArray[i] = i;
   someSubroutine(myArray);
   return 0;
```

$\Rightarrow$

```

int main(int argc, char *argv[]) {
   int i;
   int myArray[oops};
   for (i=0; i<oops; i++)
      myArray[i] = i;
   someSubroutine(myArray);
   return 0;
```

- This is happening on the characters in the program
- It’s as though the programmer had typed something else
- There is no type error in the code the programmer actually typed, because the “pre-processing” is just string-to-string
- The compiler will raise type errors in the produced code, though

7. Generics / Parameterized Types

```
int allEqual( int array[], int arrayLength) {
     int i;
    for (i=1; i<arrayLength; i++)
           if ( array[i] != array[0] ) return false;
    return true;
}
```

- Because the compiler wants to help the programmer by catching type errors at compile time, the programmer must specify the type of the elements of the array
- If I wanted this functionality for an array of floats, say, I'd need to write another version of the method
- And another for Rectangles and Circles and etc.
- But, the code is correct (almost) no matter what the type of the elements is
- Can I have a "generic" implementation (and still have type checking)?

7. Generics / Parameterized Types

```
boolean allEqual( int array[], int arrayLength) {
     int i;
    for (i=1; i<arrayLength; i++)
           if ( array[i] != array[0] ) return false;
    return true;
}
```

- Because the compiler wants to help the programmer by catching type errors at compile time, the programmer must specify the type of the elements of the array
- If I wanted this functionality for an array of floats, say, I'd need to write another version of the method
- And another for Rectangles and Circles and etc.
- But, the code is correct (almost) no matter what the type of the elements is
- Can I have a "generic" implementation (and still have type checking)?

7. Generics / Parameterized Types

- The solution is to write a type-independent code template (that's C++ terminology) that the compiler can use to generate type-specific implementations as needed

```
template <class T>
boolean allEqual( T array[], int arrayLength) {
     int i;
    for (i=1; i<arrayLength; i++)
          if ( array[i] != array[0] ) return false;
    return true;
}
```

7. Generics / Parameterized Types

```
template <class T>
boolean allEqual( T array[], int arrayLength)
{
     int i;
    for (i=1; i<arrayLength; i++)
          if ( array[i] != array[0] ) return false;
    return true;
}
```

```
…
Rectangle rectArray[20];
…
 if (allEqual(rectArray), 20) ) {
          ….
}
```

```
boolean allEqual( Rectangle array[], int arrayLength)
{
     int i;
    for (i=1; i<arrayLength; i++)
          if ( array[i] != array[0] ) return false;
    return true;
}
```

8. Dynamically Typed Languages

- Some (popular) languages check types at run time rather than compile time
- Advantages:
  - Generics are no problem
  - Language feels more powerful to programmer
- Disadvantages:
  - Code produced by compiler may be slower (than if statically typed)
  - Instead of a clear type error at compile time, you may have a very unclear run time error
- Example dynamically typed languages:
  - Python
  - Javascript
- (Java does as much type checking statically as it can, and then does some verification of type compatibility dynamically when it has no other choice)

8. Dynamically Typed Languages

Example python program (lect6.py)

```
dynamicallyTypedVariable = 10;
print('variable = {0}'.format(dynamicallyTypedVariable));

dynamicallyTypedVariable = "Hello";
print('variable = {0}'.format(dynamicallyTypedVariable));
```

Example execution (Note: no obvious compile step)

```
$ python lect6.py
variable = 10
variable = Hello
```

Lecture Summary

- Our attention has been on the hardware-software interface
  - The Instruction Set Architecture (ISA)
- The most important point to us is that the compiler is a translator from a program written in the source language to a program written to the ISA specification
  - We’ve concentrated on translation of simple, statically typed languages (like C)
- But, the compiler can do much more
  - E.g., help the programmer generate code (rather than having to type it all)
  - Can do some/many things dynamically rather than statically, which in some ways makes writing the program easier