More on classes, methods, functions

- we can provide the implementation of a method within the class definition
- we can create constructors that take parameters
- constructors can declare initializer lists - values to use when initializing the class fields
- we can identify functions/methods as “inline”: an optimization suggestion for the compiler

method defs within the class def

- places the code for the method directly inside the class definition instead of having it outside
- can mix & match: do some internally, some externally

```
// implementation external
class example {
   public:
       void hi();
};

void example::hi()
{
   cout << “Hi!”;
}
```

```
// implementation internal
class example {
   public:
       void hi() {
           cout << “Hi!”;
        }
};
```

Parameterized constructors

- constructors can have parameters, and can use default values
- caller passes the parameters when declaring/creating instance

```
class circle {
    private:
        int x, y, radius;
    public:
        example();
        example(int xv, int yv, int rv=1);
};

circle::circle(int xv, int yv, int rv)
{
   x = xv; y = yv; radius = rv;
}
```

```
int main()
{
   circle c1;  // uses default constructor
   circle c2(5,6);  // uses parameterized, default for rv
   circle c3(1,2,3);   // uses parameterized
   circle *cptr = new circle;  // uses default
   circle *cptr2 = new circle(2,4,6); // uses parameterized
   ...
}
```

As with overloading functions, we need to ensure there is no possible ambiguity about which constructor could be called.

Initializer lists

- constructors can be followed by an initializer list, identifying values to be used to initialize fields
- again need to be sure there is no possible ambiguity about which constructor version should be called

```
class circle {
    private:
        float x, y, radius;
    public:
         // example: initializer list and empty body
         circle(): x(0), y(0), radius(1) { }
};
```

Inlining methods/functions

- can suggest “inlining” a method/function as an optimization possibility to the compiler
- suggests replacing calls to the method/function with a direct substitution of the function body
- generally only done when body is simple/direct and the overhead of the function call would be much higher than the execution of the body

```
class example {
  private:
    int* ptr;
  public:
    inline void nullify() { ptr = NULL; }
};
```

```
int main()
{
    example x;
    ...
    x.nullify();
    // instead of method call it turns into x.ptr = NULL;
}
```