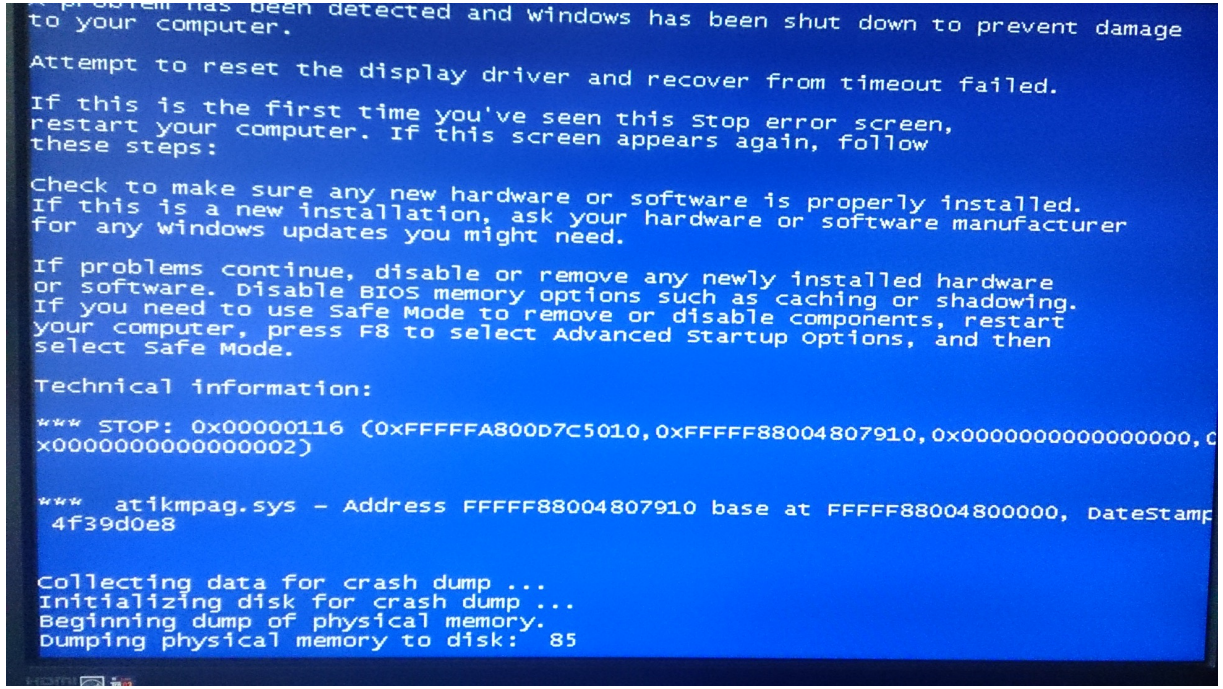
Exceptions & Errors

CMSC 240 Software Systems Development

Today

- Errors
- Exceptions
- Debugging



RICHMOND

Today

- Inheritance
- Polymorphism
- Virtual functions
- Pure virtual functions and abstract classes

RICHMOND

Errors

- When we write programs, errors are natural and unavoidable; the question is, how do we deal with them?
  - Organize software to minimize errors
  - Eliminate existing errors with
    - Debugging
    - Testing

- "Avoiding, finding, and correcting errors is **95%** or more of the effort for serious software development." – Bjarne Stroustrup

Sources of Errors

- Poor specification
  - “What’s this supposed to do?”
- Unexpected arguments
  - “but sqrt() isn’t supposed to be called with -1 as its argument”
- Unexpected input
  - “but the user was supposed to input an integer”

- **Code that simply doesn’t do what it was supposed to do**

Kinds of Errors

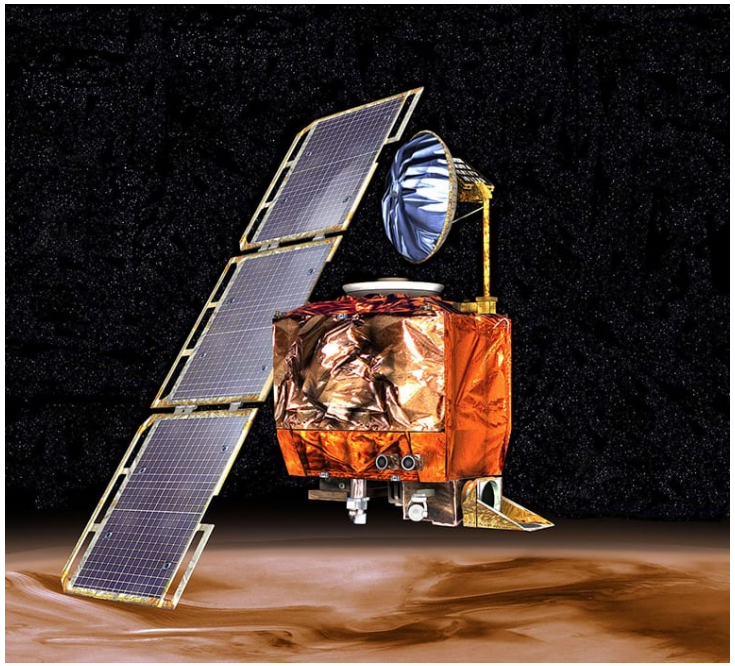
- Compile-time errors
  - Syntax errors
  - Type errors
- Link-time errors
- Run-time errors
  - Detected by computer (crash)
  - Detected by library (exceptions)
  - Detected by user code
- Logic errors
  - Detected by programmer (code runs, but produces incorrect output)

Check Your Inputs

- Before trying to use an input value, check that it meets your expectations/requirements
  - Function arguments
  - Data from input (istream, fstream)

Mars Climate Orbiter

The navigation team at the Jet Propulsion Laboratory (JPL) used the metric system in its calculations, while Lockheed Martin Astronautics in Denver, Colorado, which designed and built the spacecraft, provided crucial acceleration data in the English system.



```
1   int area(int length, int width)
2   {
3       return length * width;
4   }
5
6   int main()
7   {
8       int result1 = area(7);          // error: wrong number of arguments
9
10      int result2 = area("seven", 2); // error: 1st argument has a wrong type
11
12      int result3 = area(7, 10);      // ok
13
14      int result4 = area(7.5, 10);    // ok, but dangerous: 7.5 truncated to 7
15
16      int result5 = area(10, -7);     // ok, but this is a difficult case:
17                                      // the types are correct,
18                                      // but the values make no sense
19
20      return 0;
21  }
```

Bad Function Arguments

- So, how about:

```
int result = area(10, -7);
```

- Alternatives:
  - Just don’t’ do that
    - Hard to control all situations
  - The caller should check
    - Get’s messy, and is hard to accomplish systematically

```
 9  int error(string message)
10  {
11      cerr << "Error: " << message << endl;
12      exit(1);
13  }
14
15  int main()
16  {
17      int x, y;
18      cout << "Enter values for x and y:" << endl;
19      cin >> x >> y;
20
21      // Caller validates the inputs
22      if (x <= 0)
23          error("Non-positive x value.");
24
25      if (y <= 0)
26          error("Non-positive y value.");
27
28      int result = area(x, y);
29      cout << "Area == " << result << endl;
30
31      return 0;
32  }
```

How to Report an Error

- Return an "error value" (not general, problematic)

```
1    int area(int length, int width)    // returns a negative value for bad input
2    {
3        if(length <= 0 || width <= 0)
4            return -1;
5
6        return length * width;
7    }
```

- So, let the caller beware

```
42       int result = area(x, y);
43       if (result < 0)
44           error("Bad area computation.");
```

alue?

- For some functions there isn't a "bad value" to return (e.g., max())

```cpp
#include <iostream>
using namespace std;

class Parent
{
public:
    Parent()
    {
        cout << "1. Parent class under construction." << endl;
    }
};

class Child : public Parent   // Child inherits from the Parent
{
public:
    Child()
    {
        cout << "2. Child class under construction." << endl;
    }
};

int main()
{
    // Create a new instance of the child class.
    Child childInstance;
}
```

```cpp
#include <iostream>
using namespace std;

class Parent
{
public:
    Parent()
    {
        cout << "1. Parent class under construction." << endl;
    }
};

class Child : public Parent   // Child inherits from the Parent
{
public:
    Child()
    {
        cout << "2. Child class under construction." << endl;
    }
};

int main()
{
    // Create a new instance of the child class.
    Child childInstance;
}
```

```
1. Parent class under construction.
2. Child class under construction.
```

Ask a question

Today

- ~~Inheritance~~
- Polymorphism
- Virtual functions
- Pure virtual functions and abstract classes

RICHMOND

Polymorphism

- Polymorphism is a foundational concept in object-oriented programming that enables objects of different classes to be treated as objects of a common super class

- The term "polymorphism" is derived from Greek and means "having multiple forms"

- At its core, polymorphism allows one interface to represent **many different types** of objects or methods

Polymorphism

- Remember, a class defines a type
- A type defined by a subclass is called a subtype, and a type defined by its superclass is called a supertype
- For example
  - **`Dog`** is a subtype of **`Animal`**, and
  - **`Animal`** is a supertype for **`Cat`**
- Polymorphism means that a variable of a supertype can refer to a subtype object
  - For example, an **`Animal`** could be used to refer to a **`Cat`** or **`Dog`**

Polymorphism

- An object of a subtype can be used wherever its supertype value is required

For example: the **`animals`** vector is a list of pointers to **`Animal`** types. But we load it with **`Dog`** and **`Cat`** types.

```
// Create a dog and a cat.
Dog woofer{"Woofer", 3, 36.4};
Cat cheddar{"Cheddar", 5, 3.1};

// Create a vector of animal pointers.
vector<Animal*> animals;

// Add addresses to a dog and a cat.
animals.push_back(&woofer);
animals.push_back(&cheddar);
```

Polymorphism

- An object of a subtype can be used wherever its supertype value is required

Actual types

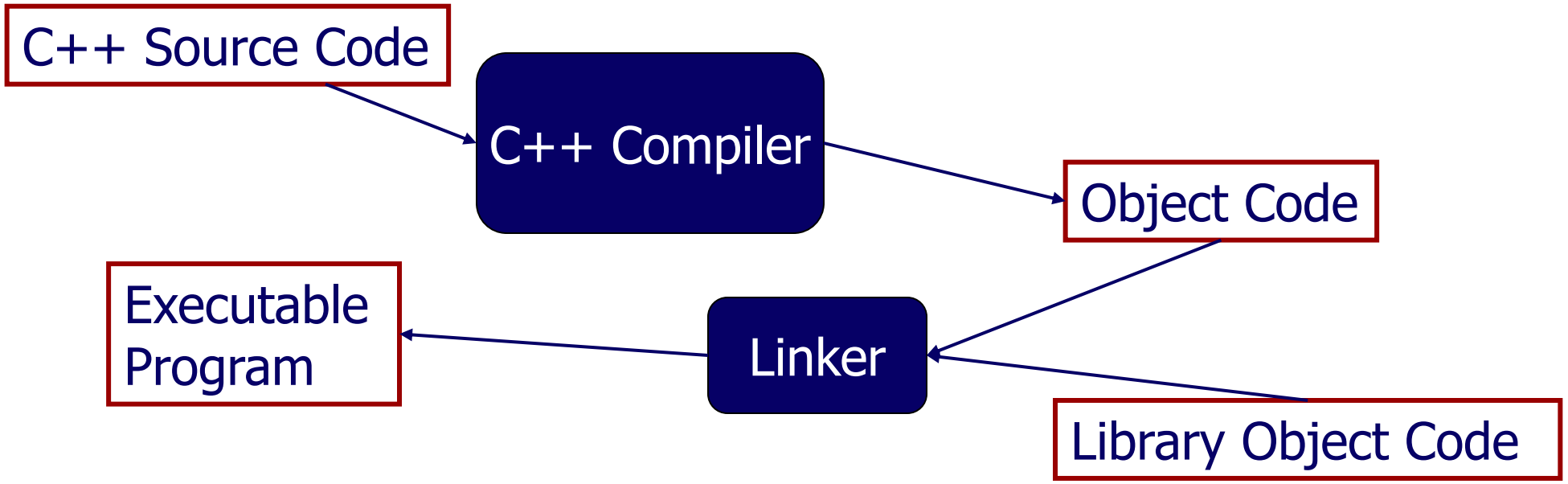
Declared type

```
// Create a dog and a cat.
Dog woofer{"Woofer", 3, 36.4};
Cat cheddar{"Cheddar", 5, 3.1};

// Create a vector of animal pointers.
vector<Animal*> animals;

// Add addresses to a dog and a cat.
animals.push_back(&woofer);
animals.push_back(&cheddar);
```

Today

- ~~Inheritance~~
- ~~Polymorphism~~
- Virtual functions
- Pure virtual functions and abstract classes

RICHMOND


C++ Source Code
C++ Compiler
Object Code
Executable Program
Linker
Library Object Code

```
1   #ifndef ANIMAL_H
2   #define ANIMAL_H
3   #include <string>
4
5   class Animal
6   {
7   public:
8       Animal(std::string name, int age);
9       void eat();
10      void setFavoriteFood(std::string favorite);
11      virtual void speak();
12  private:
13      std::string name;
14      int age;
15      std::string favoriteFood;
16      void sleep();
17  };
18
19  #endif
```

Today

- ~~Inheritance~~
- ~~Polymorphism~~
- ~~Virtual functions~~
- Pure virtual functions and abstract classes

RICHMOND

Pure Virtual Function

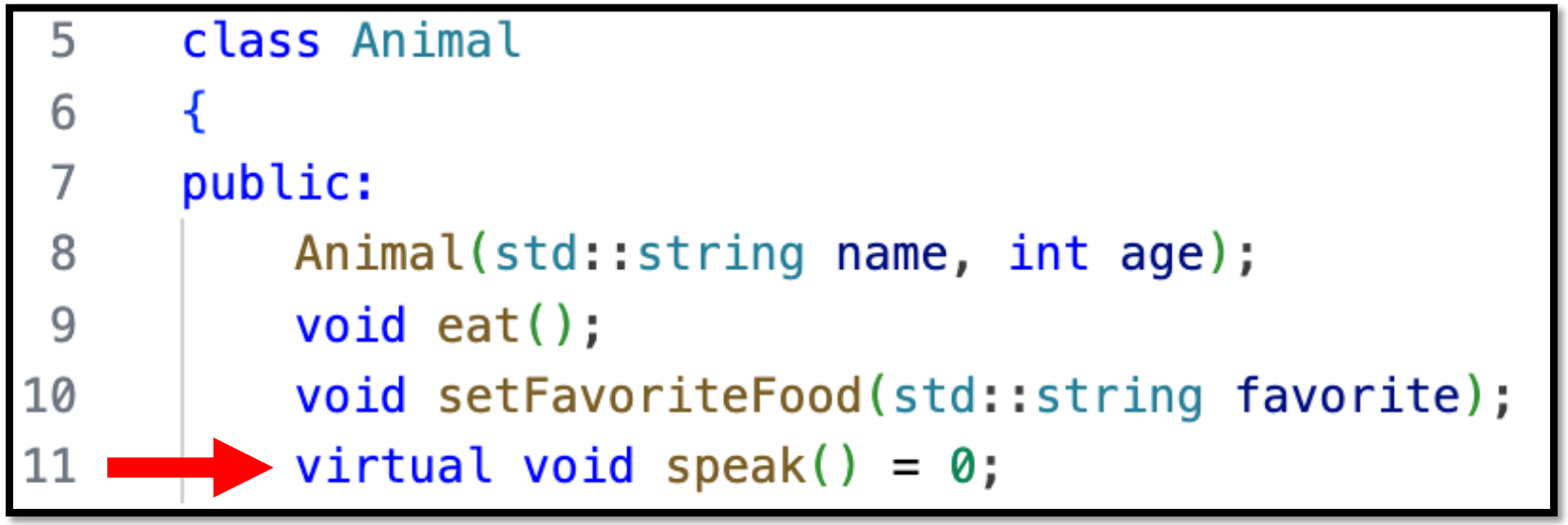
pure virtual function

- a virtual function with an `= 0` assignment
- indicating that there is no implementation for that function
- any concrete derived class must provide an implementation

```
 5    class Animal
 6    {
 7    public:
 8        Animal(std::string name, int age);
 9        void eat();
10        void setFavoriteFood(std::string favorite);
11        virtual void speak() = 0;
```

Abstract Class

- An **abstract class** is a class that either defines or inherits at least one function for that is pure virtual
- **You can not create an instance of an abstract class**



Acknowledgements

- BSOD image source Wikimedia
- Satellite image source Wikimedia

Ask a question