IS2203: Object Oriented Programming

Exception Handling

Viraj Welgama

What is an Error?

- **Error** is a fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways.
- Some of the errors inhibit the program from getting compiled or executed.
- Programming errors often remain undetected until the program is compiled or executed.
- Errors may have subtle effects or cause the program to crash or freeze the computer.

Terminology!

```
public:
    int add(int x, int y) {
        return x-y;
    }
```

- A mistake in the code is called Error.
- Error found by tester is called defect.
- Defect accepted by development team is called Bug.
- And the program does not meet the requirements then it is Failure.

why errors in a computer program called bugs?

The term "**bug**" for errors in a computer program dates back to early computing history. The most famous origin story involves **Grace Hopper**, a pioneer of computer programming, who discovered a literal **moth** stuck in a relay of the Harvard Mark II computer in 1947. The moth was causing a malfunction, and when it was removed, the team jokingly recorded the incident as "debugging" the system.

However, the term "**bug**" was already in use before that. Engineers and inventors, including **Thomas Edison**, used "bug" to refer to mechanical or electrical faults as early as the late 19th century. It became widely adopted in computing due to Hopper's popularization of the term.

Today, "**debugging**" remains the standard term for finding and fixing errors in software and hardware.

why errors in a computer program called bugs?

The term "bug" to describe errors or glitches in a computer program has an interesting history. It is often attributed to an incident involving early computer pioneer Grace Hopper in 1947. While working on the Harvard Mark II computer, her team discovered a malfunction caused by a moth trapped in a relay. They removed the moth and taped it into their logbook, noting it as the "first actual case of a bug being found."

However, the use of the word "bug" to describe technical issues predates this event. Thomas Edison, for example, used the term in the 1870s to describe problems with his inventions. The term "bug" had been used in engineering and other fields for decades to refer to small defects or malfunctions.

In the context of computer programming, "bug" became a popular term to describe errors in code, whether they were caused by hardware issues (like the moth) or software logic mistakes. Today, "debugging" refers to the process of finding and fixing these errors in programs.

why errors in a computer program called bugs?

The term "bug" in computer programming has a rather interesting origin story! Here's the gist of it:

- **It wasn't always about software:** The term "bug" to describe a technical problem actually predates computers. It was used in engineering for issues with all sorts of machines. Thomas Edison, the famous inventor, used the term "bug" to describe problems with his inventions way back in the late 19th century.
- **The moth in the machine:** However, the story that really popularized the term "bug" in the computing world involves an actual bug – a moth! In 1947, Grace Hopper, a pioneering computer scientist, was working on the Harvard Mark II computer when they found a moth trapped in a relay, causing the machine to malfunction.
- **"First actual case of bug being found":** Hopper and her team taped the moth into the computer's logbook with the note "First actual case of bug being found." This incident became a famous anecdote and helped solidify the term "bug" for any kind of error or glitch in a computer program.

So, while the term "bug" was already around, it was this moth incident that really made it stick in the world of computer programming!

Type of Errors

1. Syntax Errors
   - Just like human languages, computer languages have grammar rules. But while humans are able to communicate with less-than-perfect grammar, computers can't ignore mistakes, i.e. syntax errors.

```
14      MyMaths m
15      cout<<m.add(12, 10);
16
```

Compiler (2) | Resources | Compile Log | Debug | Find Results | Close

Line	Col	File	Message
		C:\Users\UCSC\Google Drive\Courses\1st Year - SCS1...	**In function 'int main()':**
15	2	C:\Users\UCSC\Google Drive\Courses\1st Year - SCS1209...	[Error] expected initializer before 'cout'

   - As your proficiency with programming language increases, you will make syntax errors less frequently.
   - Many text editors or IDEs are coming with the ability to warn you about syntax errors at the time of writing.

Type of Errors

2. Logic Errors
   - Logic errors can be the hardest to track down. Everything looks like it is working; you have just programmed the computer to do the wrong thing.
   - Technically the program is correct, but the results won't be what you expected.

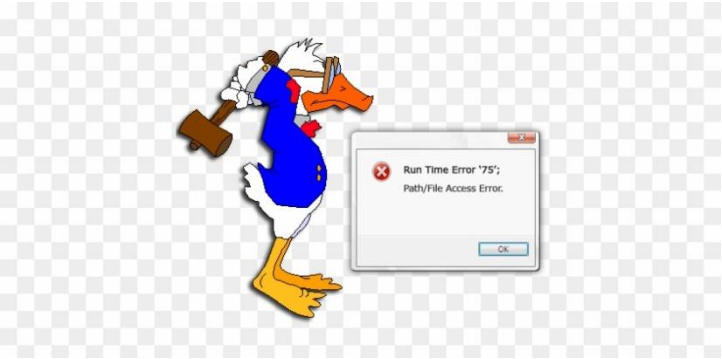
```
int findMax(int x, int y) {
    return (x < y)? x : y;
}
```

   - in 1999 NASA lost a spacecraft due to miscalculations between English and American units. The software was coded one way but needed to work another.

Type of Errors

3. Compilation Errors
   - Some programming languages require a compilation step. Compilation is where your high-level language converts into a lower-level language that the computer can understand better. A compilation or compile-time error happens when the compiler doesn't know how to turn your code into the lower-level code.
   - Compilation errors are occurred due to syntax errors.
   - If there is a compile-time error in your software, you won't be able to get it tested or launched

Type of Errors



4. Runtime Errors
   - Runtime errors happen as a user is executing your program. The code might work correctly on your machine, but there may be an issue with some resources like in a file.
   - Runtime errors are particularly annoying because they directly impact your end user.
   - A lot of these other errors will happen when you're at your computer working on the code. These errors occur when the system is running and can stop someone from doing what they need to do.
   - having a good error reporting in place to capture any runtime errors and let automatically open up new bugs in your program can be good solutions.
   - Learning from every bug report will prevent the same error happening again in the future.
   - use of frameworks and community maintained code are excellent ways of minimizing these types of errors.

Type of Errors

5. Arithmetic Errors
   - An arithmetic error is a type of logic error but involves mathematics.
   - A typical example when performing a division by zero
   - Arithmetic errors can generate logic errors or even run-time errors like in the case of divide by zero.
   - Having functional tests that always include edge-cases like zero, or negative numbers is an excellent way to stop these arithmetic errors in their tracks.

Type of Errors

6. Resource Errors
   - The computer that your program is on will allocate a fixed amount of resources to the running of it. If something in your code forces the computer to try and allocate more resources than it has, it can create a resource error.
   - If you accidentally wrote a loop that your code could never exit from, you would eventually run out of resources. In this example, the while loop will keep on adding new elements to an array. Eventually, you will run out of memory.

```
list = [1, 2, "a", "c", 3.4]
while (1):
    list.append("Hi")
```

   - Resource errors are an example of a type of error in programming that might be something for the operations team to fix rather than developers.

Type of Errors

7. Interface Errors
   - Interface errors occur when there is a disconnect between how you meant your program to be used and how it is actually used.
   - Most things in software follow standards. If input your program receives doesn't conform to the standards, you might get an interface error.

Detecting Errors

Defensive Programming

- Write specifications for functions
- Modularize programs
- Check conditions of inputs/outputs

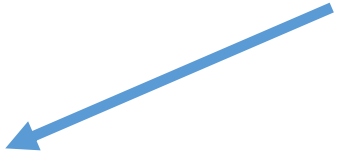
Detecting Errors

Defensive Programming

- Write specifications for functions
- Modularize programs
- Check conditions of inputs/outputs

Testing/Validation

- Compare input/output pairs to specification
- “It’s not working!”
- “How can I break my program?”



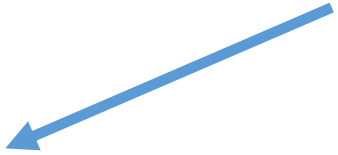
Detecting Errors

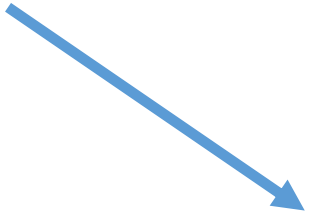
Defensive Programming

- Write specifications for functions
- Modularize programs
- Check conditions of inputs/outputs

Testing/Validation

- Compare input/output pairs to specification
- “It’s not working!”
- “How can I break my program?”





Debugging

- Study events leading up to an error
- “Why is it not working?”
- “How can I fix my program?”

Avoiding Errors

1. Syntax errors -
2. Logic errors -
3. Compilation errors -
4. Run time errors -
5. Arithmetic Errors -
6. Resource Errors -
7. Interface Errors -

Avoiding Errors

1. Syntax errors - IDEs will prompt the errors
2. Logic errors - Black box, white box testing
3. Compilation errors - Compiler will prompt the errors
4. Run time errors - ?
5. Arithmetic Errors - Black box, white box testing
6. Resource Errors - Stress Testing
7. Interface Errors - Component Testing

Avoiding Errors

1. Syntax errors - IDEs will prompt the errors
2. Logic errors - Black box, white box testing
3. Compilation errors - Compiler will prompt the errors
4. Run time errors - Defensive Programming
5. Arithmetic Errors - Black box, white box testing
6. Resource Errors - Stress Testing
7. Interface Errors - Component Testing

How do we handle this in C++?

- C++ supports to handle such issues through **Exception Handling**
  - It is one of the advantages of C++ over C

What is Exception?

- An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

- In other wards, Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution.

- An exception is a C++ object that represents an error.
- When the program raises an exception, it must either handle the exception immediately otherwise it would terminate and come out.

Why Exception Handling?

- **Separation of Error Handling code from Normal Code:**
  - In traditional error handling codes, there are always if else conditions to handle errors.
  - These conditions and the code to handle errors get mixed up with the normal flow.
  - This makes the code less readable and maintainable.
  - With try catch blocks, the code for error handling becomes separate from the normal flow.

Why Exception Handling?

- **Functions/Methods can handle any exceptions they choose:**
  - A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller.
  - If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.
  - In C++, a function can specify the exceptions that it throws using the throw keyword.
  - The caller of this function must handle the exception in some way (either by specifying it again or catching it)

Why Exception Handling?

- **Grouping of Error Types:**
  - In C++, both basic types and objects can be thrown as exception.
  - We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

Specialized Keywords

- **try:**
  - represents a block of code that can throw an exception.
- **catch:**
  - represents a block of code that is executed when a particular exception is thrown.
- **throw:**
  - used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

Example:

This is a simple example to show exception handling in C++.

The output of program explains flow of execution of try/catch blocks.

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

int main() {
   int x = -1;

   // Some code
   cout << "Before try \n";
   try {
      cout << "Inside try \n";
      if (x < 0)
      {
         throw x;
         cout << "After throw (Never executed) \n";
      }
   }
   catch (int x ) {
      cout << "Exception Caught \n";
   }

   cout << "After catch (Will be executed) \n";
   return 0;

}
```

catch(...)

- There is a special catch block called '**catch all**' catch(...) that can be used to catch all types of exceptions.
- For example, in this program, an int is thrown as an exception, but there is no catch block for int, so catch(...) block will be executed.

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

int main()
{
    try  {
       throw 10;
    }
    catch (char *excp)  {
        cout << "Caught " << excp;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
```

No Implicit Conversion in Try-Catch

Implicit type conversion doesn't happen for primitive types.

For example, in this program 'a' is not implicitly converted to int

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

int main()
{
    try  {
       throw 'a';
    }
    catch (int x)  {
        cout << "Caught " << x;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
```

If the exception did not catch…?

- If an exception is thrown and not caught anywhere, the program terminates abnormally.
- For example, in this program, an int is thrown, but there is no catch block to catch an int.

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

int main() {

   int x = -1;

   try {
        throw x;
        cout << "After throw (Never executed) \n";
     }

   catch (char x) {
      cout << "Exception Caught \n";
   }

   cout << "After catch (Will be executed) \n";
   return 0;

}
```

Throwing Objects as Exceptions

- You can throw not only the primitive the data type, but also any objects as exceptions
- In this program, it throws an object of class A.

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

class A {
    public:
    A() {
        cout<<"constructor of class A is called...\n";
    }
    void anyError() {
        cout<<"Sorry, there is an error!\n";
    }
};
int main() {

   A a; // creating an object of class A

   try {
        throw a;
     }

   catch (A aa) {
     aa.anyError();
   }

   cout << "Error handled. \n";
   return 0;

}
```

Exceptions with Inheritance

- If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class.
- If we put base class first then the derived class catch block will never be reached.

Exceptions with Inheritance

- If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class.
- If we put base class first then the derived class catch block will never be reached.

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

class Base {};
class Derived: public Base {};

int main() {
   Derived d;
   // some other stuff
   try {
      // Some monitored code
      throw d;
   }
   catch(Base b) {
       cout<<"Caught Base Exception";
   }
   catch(Derived d) {  //This catch block is NEVER executed
       cout<<"Caught Derived Exception";
   }
   return 0;
}
```

Exceptions with Inheritance

- If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class.
- If we put base class first then the derived class catch block will never be reached.

- if we change the order of catch statements of this program, then both catch statements become reachable.
- In Java, catching a base class exception before derived is not allowed by the compiler itself. In C++, compiler might give warning about it, but compiles the code.

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

class Base {};
class Derived: public Base {};

int main() {
   Derived d;
   // some other stuff
   try {
      // Some monitored code
      throw d;
   }
   catch(Base b) {
       cout<<"Caught Base Exception";
   }
   catch(Derived d) {  //This catch block is NEVER executed
       cout<<"Caught Derived Exception";
   }
   return 0;
}
```

Standard **Exception** Class

- Like Java, C++ library has a standard exception class which is base class for all standard exceptions which is defined in the <exception> header.
  - http://www.cplusplus.com/reference/exception/exception/
- It is the root of the exception hierarchy, allowing custom exceptions to be derived from it.
- All objects thrown by components of the standard library are derived from this class.
- Therefore, all standard exceptions can be caught by catching this type.
- C++ provides derived classes like **std::runtime_error**, **std::logic_error**, etc., for common error types.
- **what():** A virtual member function that returns a C-style string describing the error. Derived classes can override this to provide specific error messages.

Unchecked Exceptions

- Unlike Java, in C++, all exceptions are unchecked.
- Compiler doesn’t check whether an exception is caught or not.
- For example, in C++, it is not necessary to specify all uncaught exceptions in a function declaration.
  - Although it’s a recommended practice to do so.

Example:

- This function signature is fine by the compiler, but not recommended.
- Ideally, the function should specify all uncaught exceptions in the signature.

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

void fun(int *ptr, int x)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}

int main()
{
    int y = 0;
    try {
       fun(NULL, y);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

Example:

- This function signature is fine by the compiler, but not recommended.
- Ideally, the function should specify all uncaught exceptions in the signature.
- So, the recommended signature would be;

```
void fun(int *ptr, int x) throw (int *, int)
```

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

void fun(int *ptr, int x)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}

int main()
{
    int y = 0;
    try {
       fun(NULL, y);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

Nested Catch Blocks

- In C++, try-catch blocks can be nested.

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

int main() {

    try {
        try {
            throw 20;
        }
        catch (int n) {
            cout << "Handle Partially ";
        }
        throw 'a';
    }
    catch (char n) {
        cout << "Handle remaining ";
    }
    return 0;
}
```

Re-throwing Exceptions

- A function can also re-throw an exception using same “throw; “.
- One function can handle a part and can ask the caller to handle the remaining.

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

int main() {
    try {
        try {
            throw 20;
        }
        catch (int n) {
            cout << "Handle Partially ";
            throw; // Re-throwing an exception
        }
    }
    catch (int n) {
        cout << "Handle remaining ";
    }
    return 0;
}
```

Calling Destructors

- When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block.

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

class Test {
public:
    Test() { cout << "Constructor of Test " << endl; }
    ~Test() { cout << "Destructor of Test " << endl; }
};

int main() {
    try {
        Test t1;
        throw 10;
    }
    catch (int i) {
        cout << "Caught " << i << endl;
    }
}
```

Some Important points…

- A single try statement can have multiple catch statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic catch clause (`catch(…)`), which handles any exception.
- You can throw anything as an exception.
- If you are not supposed to handle a particular exception with a function, you can re-throw that exception out from that function to be handled by the caller.

Special Note:

- In C++, exceptions are not automatically triggered for things like division by zero or out-of-bounds array access (unlike Java).
  - However, you can manually trigger exceptions using the throw keyword inside a try block.