Overloading and Debugging

CS A150 - C++ Programming 1

Overloading a Function

- C++ allows functions of the same name to be defined, as long as they have different **signatures**
  - The **signature** of a function is
    the **name** of the function and the **parameter list**
  - Must be "unique" for each function definition
- Same function name
- Different parameter lists
- Two separate function definitions
- Allows same task performed on different data

Overloading a Function (cont.)

- When an overloaded function is called,

  the *compiler* selects the proper function by looking at the
  - **Number** of **parameters**
  - **Type** of **parameters**
  - **Order** of **parameter**

Overloading Example: Average

- Function computes average of 2 numbers:

```
double average(double n1, double n2)
{
        //some code here…
}
```

- Now compute average of 3 numbers:

```
double average(double n1, double n2, double n3)
{
         //some code here…
}
```

- Same name, two different functions

Overloading Example (cont.)

- Which function gets called?
- Depends on function call itself:
  - `avg = average(5.2, 6.7);`
    - Calls "two-parameter average()"
  - `avg = average(6.5, 8.5, 4.2);`
    - Calls "three-parameter average()"
- Compiler resolves invocation based on *signature* of function call
  - "Matches" call with appropriate function
  - Each considered separate function

Example

- Example 1: Overloading a Function Name

Overloading Pitfall

- **Only** overload functions that perform the same task
  - An **`average( )`** function should ***always*** perform the same tasks in all overloads
- C++ function call resolution:
  - Compiler looks for ***exact*** signature
  - If exact signature **not** found, compiler looks for "compatible" signature
    - Careful! Possible loss of data

Function Call Resolution

- Given the following function:

```
void func( int n, double m );
```

- Possible calls:

```
func ( 2, 3 );       // converts 3 to a double
func ( 2.1, 3);      // truncates 2.1 to 2
func ( 2.1, 3.1 );   // truncates 2.1 to 2
func ( 2, 3.1);      // correct call
```

Function Call Resolution (cont.)

- To improve **readability**, add "**.0**" to a number that has no decimals **but** will be treated as a double.
- Example:

Given the function:

```
void func( int n, double m );
```

Avoid this:

```
func ( 2, 3);
```

Write this instead:

```
func ( 2, 3.0);
```

Default Arguments

- Allows omitting some arguments specified in function declaration/prototype
- Function given:

```
void showVolume( double length,
                 double width = 1.0,
                 double height = 1.0 );
```

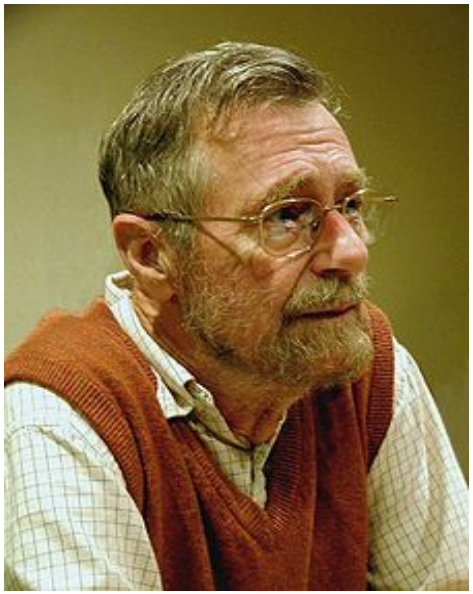
*last two arguments are **defaulted***

- Possible calls:

```
showVolume (2.0, 4.5, 6.2); //all arguments supplied
showVolume (2.0, 4.5); //height defaulted to 1.0
showVolume (2.0); //width and height defaulted to 1.0
```

Debugging

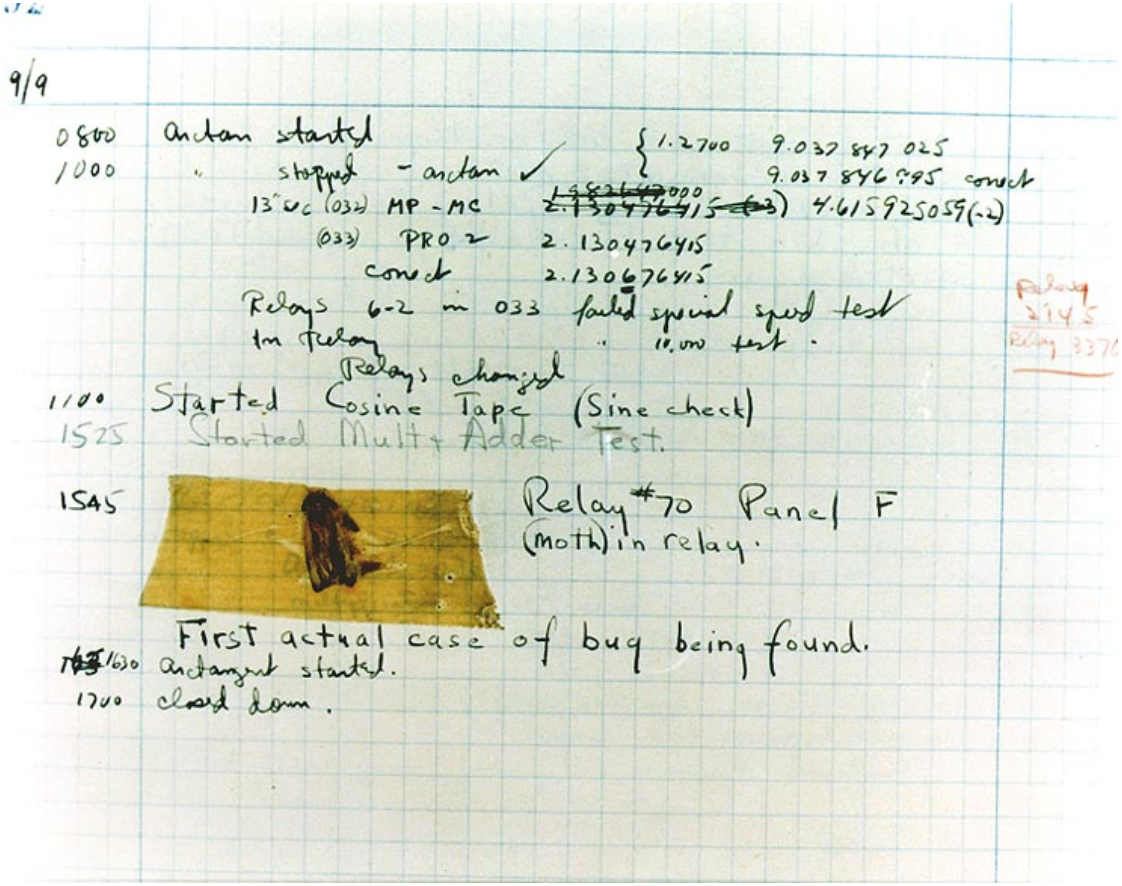
- **Edsger Dijkstra**
  - Famous Dutch computer scientist (1930-2002)
  - *"Testing can only reveal the presence of bugs, not their absence."*



Debugging

- **Grace Murray Hopper**
  - Computer scientist who led to the development of the programming language **COBOL**
  - Worked on the Harvard University **Mark II** (a primitive computer). In 1947, she coined the word "debugging" after one of the machines was not working and a **moth** was found in a system.

THE "BUG"



9/9

0800 Arctan started
1000 " stopped - arctan ✓
13"uc (032) MP - MC
(033) PRO 2
correct
1.2700 9.037 847 025
9.037 846 995 correct
4.615925059(-2)
2.13047615
2.13067615
Relays 6-2 in 033 failed special speed test
tm relay
Relays changed
10.00 test.
1100 Started Cosine Tape (Sine check)
1525 Started Multy Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1700 closed down.

Facts about Bugs

- Every program you write that is more than trivial will contain bugs.
- Many programs that you write will contain bugs even after you think you have fully tested them.
- Program bugs can remain hidden in a program that is apparently operating correctly—sometimes for years.
- Programs beyond a certain size and complexity always contain bugs, no matter how much time and effort you put in testing them.

Broad Strategies

- Strategies to make debugging as painless as possible:
  - Don't re-invent the wheel; understand and use the library.
  - Develop and test your code incrementally by testing each class and function individually.
  - Include debugging code that checks and validate data and conditions in your program.

Who Originates Bugs?

Who Originates Bugs?

you

(the programmer)

Type of Errors

- **Syntactic errors**
  - These are errors that result from statements that are not of the correct form (missing a semicolon, use a colon when you should have a comma). These are **easy to fix**, because the **compiler will alert you**.
- **Semantic** (or **logical**) **errors**
  - These are errors where the code is syntactically correct, but it does not do what you intended. You may get an indication that your program has a semantic error because it terminates abnormally or the output is not as expected. The **compiler cannot recognize these types of errors**.

Most Common Errors

- Failure to **initialize a variable**
- **Exceeding** integer type **range**
- **Loop condition** error
- **Infinite loop**
- **Omitting break** in a ***switch*** statement
- Error in **allocating size of array**
- Confusing **assignment operator** (=) with **comparison operator** (= =)
- Failure to process **unexpected user input** properly
- **Invalid** pointer or **reference**

Testing and Debugging Functions

- Many methods:
  - Lots of `cout` statements
    - In calls and definitions
    - Used to "trace" execution
  - **Compiler debugger**
    - Environment-dependent
    - MS Visual Studio has a powerful debugger
  - `assert` macro
    - Early termination as needed
  - **Stubs** and **drivers**
    - Incremental development

THE assert MACRO

- Assertion: a **true** or **false** statement
- Used to document and check **correctness**
  - Syntax:

```
assert( <assert_condition> );
```

    - No return value
    - Evaluates ***assert_condition***
    - Terminates if false, continues if true
- Predefined in library **<cassert>**
  - Macros used similarly as functions

THE assert MACRO - EXAMPLE

- Given the function declaration:

```
void computeCoin ( int coinValue,
                   int& number,
                   int& amountLeft );
```

- We want to make sure that

```
0 < coinValue < 100
0 <= amountLeft <= 100
```

- So we check

```
assert (( 0 < coinValue ) && ( coinValue < 100));
assert (( 0 <= amountLeft) && (amountLeft <= 100);
```

If this is not satisfied, then the program execution terminates.

assert On/Off

- No need to delete all assert statements
- Simply add "**#define NDEBUG**" *before* **#include** to turn OFF all assertions
- Remove "**#define**" line (or comment out) to turn assertions back on.

```
#define NDEBUG
#include <cassert>
```

Stubs and Drivers

- **Separate** compilation units
  - Each function **designed**, **coded**, **tested** separately.
  - Ensures validity of each unit.
  - **Divide & Conquer**
    - Transforms one big task into smaller, manageable tasks.
- But how to test independently?
  - Driver programs.

Example

- Example 2: Driver Program

Stubs

- Develop ***incrementally***
- Write "big-picture" functions first
  - Low-level functions last
  - "Stub-out" functions until implementation
  - Example:

```
double unitPrice( int diameter, double price )
{
      return (9.99); // not valid, but noticeably
                     // a "temporary" value
}
```

  - Calls to function will still "work"

Bottom Line

- You MUST test your programs for
  - Validity of data
  - Coding errors
  - Possible unexpected input

    - **BUT**, in this course we will ***always*** assume that the user enters correct data
    - **SO**, you ***only*** need to test
      - your code and
      - different values within the acceptable range

Questions?

(Overloading and Debugging)