Data Structures and Algorithms in C++

Class Meeting

Robot and Smart Systems
Manara University

Fall 2022
Instructor: Iyad Hatem

- Some of you may have discovered this while programming the solution to Assignment #1.

The C++ Division Operator

- If both operands of the / operator are integer constants or variables, then the result will be integer.
  - Any fractional amount is truncated (not rounded).
  - Examples: `7/3` ➔ `2` and `1/2` ➔ `0`
- If one or both operands are double constants or variables, then the result will be double.
  - Examples: `7/3.0` ➔ `2.333`... and `1.0/2.0` ➔ `0.5`

$$\pi = \frac{12}{\sqrt{190}} ln[(2\sqrt{2} + \sqrt{10})(3 + \sqrt{10})]$$

Assignment #1: Sample Solution

```
void Ramanujan_0()
{
    double const1 = 12/sqrt(190);
    double const2 = 2*sqrt(2) + sqrt(10);
    double const3 = 3 + sqrt(10);

    double pi = const1*log(const2*const3);
    cout << "  Estimate:  " << pi << endl;
}
```

- The built-in square root `sqrt` and the natural logarithm `log` functions are from the `cmath` library:

```
#include <cmath>
```

$$\frac{4}{\pi} = \frac{1}{882}\sum_{n=0}^{\infty} \frac{(-1)^n (4n)!}{(4^n n!)^4} \frac{1123 + 21460n}{882^{2n}}$$

Assignment #1: Sample Solution, *cont'd*

- What does the $(-1)^n$ factor do?
  - Whenever n is <u>odd</u>, the factor equals -1.
    - Example: $(-1)^3 = (-1)(-1)(-1) = -1$
  - Whenever n is <u>even</u>, the factor equals +1.
    - Example: $(-1)^4 = (-1)(-1)(-1)(-1) = +1$
- Therefore, the factor <u>alternates</u> between <u>adding and subtracting</u> the term it multiplies.

$$\frac{4}{\pi} = \frac{1}{882} \sum_{n=0}^{\infty} \frac{(-1)^n (4n)!}{(4^n n!)^4} \frac{1123 + 21460n}{882^{2n}}$$

Assignment #1: Sample Solution, *cont'd*

- It is inefficient to use the built-in power function for this purpose:
  - Use a Boolean variable instead that alternates between true and false.

```
pow(-1, n)
```

- Copying a mathematical formula directly can lead to inefficient or erroneous code.
  - A formula that is not designed for computation can accumulate roundoff errors when it is used inside of a loop. It can also have overflow errors.

See: https://www.amazon.com/Java-Number-Cruncher-Programmers-Numerical/dp/0130460419/ref=sr_1_1?dchild=1&keywords=java+number+cruncher&qid=1598936278&s=books&sr=1-1

$$\frac{4}{\pi} = \frac{1}{882} \sum_{n=0}^{\infty} \frac{(-1)^n (4n)!}{(4^n n!)^4} \frac{1123 + 21460n}{882^{2n}}$$

Assignment #1: Sample Solution, *cont'd*

```
void Ramanujan_2()
{
    cout << "  Iteration  Estimate" << endl;

    double four_over_pi;
    double factor0 = ((double) 1)/882.0;
    bool negate = false;

    double sum = 0.0;
    double prev = 0.0;
    double diff = 0.0;

    int n = 0;
```

$$\frac{4}{\pi} = \frac{1}{882}\sum_{n=0}^{\infty}\frac{(-1)^n(4n)!}{(4^n n!)^4}\,\frac{1123 + 21460n}{882^{2n}}$$

Assignment #1: Sample Solution, *cont'd*

```
    do
    {
        double factor1 = factorial(4*n)/pow((pow(4.0, n)*factorial(n)), 4);
        double factor2 = (1123 + 21460*n)/pow(882.0, 2*n);

        if (negate) factor1 = -factor1;
        sum += factor1*factor2;

        four_over_pi = factor0*sum;
        cout << setw(11) << n+1 << "  " << 4.0/four_over_pi << endl;

        diff = abs(prev - four_over_pi);
        prev = four_over_pi;
        negate = !negate;
        n++;
    } while ((diff > TOLERANCE) && (n <= MAX_ITERATIONS));
}
```

$$\frac{1}{\pi} = 12 \sum_{n=0}^{\infty} \frac{(-1)^n (6n)!}{(3n)!\,(n!)^3} \frac{13591409 + 545140134n}{(640320^3)^{\left(n+\frac{1}{2}\right)}}$$

Assignment #1: Sample Solution, *cont'd*

```
void Chudnovsky()
{
    double one_over_pi;
    double sum = 0.0;
    double prev = 0.0;
    double diff = 0.0;

    bool negate = false;
    int n = 0;
```

$$\frac{1}{\pi} = 12 \sum_{n=0}^{\infty} \frac{(-1)^n (6n)!}{(3n)!\,(n!)^3} \frac{13591409 + 545140134n}{(640320^3)^{\left(n+\frac{1}{2}\right)}}$$

Assignment #1: Sample Solution, *cont'd*

```
    do
    {
        double factor1 = factorial(6*n)/(factorial(3*n)*pow(factorial(n), 3));
        double factor2 = (13591409 + 545140134*n)/pow(640320, 3*n + 1.5);

        if (negate) factor1 = -factor1;
        sum += factor1*factor2;

        one_over_pi = 12*sum;
        cout << setw(11) << n+1 << "  " << 1.0/one_over_pi << endl;

        diff = abs(prev - one_over_pi);
        prev = one_over_pi;
        negate = !negate;
        n++;
    } while ((diff > TOLERANCE) && (n <= MAX_ITERATIONS));
}
```

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots \quad \frac{\pi}{4} = 4\arctan\frac{1}{5} - arctan\frac{1}{239}$$

Assignment #1: Sample Solution, *cont'd*

```
double arctangent(double x)
{
    double arctan = x;
    bool addsub = false;
    double numerator = x;
    double x_squared = x*x;
    double term;
    int odd = 3;

    do
    {
        numerator *= x_squared;
        term = numerator/odd;

        if (addsub) arctan += term;
        else        arctan -= term;

        odd += 2;
        addsub = !addsub;
    } while ((term > TOLERANCE) && (odd <= MAX_ITERATIONS));

    return arctan;
}
```

Predefined Functions

- C++ includes predefined functions.
  - AKA built-in functions
  - Example: Math function **`sqrt`**
- Predefined functions are stored in libraries.
  - Your program will need to include the appropriate library header files to enable the compiler to recognize the names of the predefined functions.
  - Example: **`#include <cmath>`**
    in order to use predefined math functions like **`sqrt`**

Savitch_ch_04.ppt: slides 8 – 12, 72

Prede

Some Predefined Functions

Name	Description	Type of Arguments	Type of Value Returned	Example	Value	Library Header
sqrt	square root	*double*	*double*	sqrt(4.0)	2.0	cmath
pow	powers	*double*	*double*	pow(2.0,3.0)	8.0	cmath
abs	absolute value for *int*	*int*	*int*	abs(-7) abs(7)	7 7	cstdlib
labs	absolute value for *long*	*long*	*long*	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	absolute value for *double*	*double*	*double*	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath
ceil	ceiling (round up)	*double*	*double*	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	floor (round down)	*double*	*double*	floor(3.2) floor(3.9)	3.0 3.0	cmath

Random Numbers

- To generate (pseudo-) random numbers using the predefined functions, first include two library header files:

```
#include <cstdlib>
#include <ctime>
```

- "Seed" the random

```
srand(time(0));
```

  - If you don't seed, you'll always get the same "random" sequence (which may be useful for debugging).

Random Numbers, *cont'd*

- Each subsequent call `rand();` returns a “random” number ≥ 0 and ≤ **RAND_MAX**.
  - **RAND_MAX** is library-dependent but is guaranteed to be at least 32,767.

- Use **+** and **%** to scale to a desired number range.
  - Example: Each execution of the expression

```
rand()%6 + 1
```

returns a random number
with the value 1, 2, 3, 4, 5, or 6.

Type Casting

- Suppose integer variables `i` and `j` are initialized to 5 and 2, respectively.

- What is the value of the division `i/j` ?

- What if we wanted to have a quotient of type double?
  - We want to keep the fraction.

Type Casting, *cont'd*

- One way is to convert one of the operands (say `i`) to double.
  - Then the quotient will be type double.

```
double quotient = static_cast<double>(i)/j;
```

- Why won't the following work?

```
double quotient = static_cast<double>(i/j);
```

Programmer-Defined Functions

- In addition to using the predefined functions, you can write your own functions.
- Programmer-defined functions are critical for good program design.
- In your C++ program, you can call a programmer-defined function only after the function has been declared or defined.

Function Declarations

- A function declaration specifies:
  - The function name.
  - The number, order, and data types of its formal parameters.
  - The data type of its return value.

- Example:

```
double total_cost(double unit_cost, int count);
```

Function Definitions, *cont'd*

- After you've declared a function, you must define it.
  - Write the code that is executed whenever the function is called.
  - A **return** statement terminates execution of the function and returns a value to the caller.

- Example:

```
double total_cost(double unit_cost, int count)
{
    double total = count*unit_cost;
    return total;
}
```

- Call a function that you wrote just as you would call a predefined function.

Function Calls

- Example:

```
int how_many;
double how_much;
double spent;

how_many = 5;
how_much = 29.99;
spent = total_cost(how_much, how_many);
```

Void Functions

- A void function performs some task but does not return a value.

- Therefore, its **return** statement terminates the function execution but does not include a value.
  - A return statement is not necessary for a void function if the function terminates “naturally”
    after it finishes executing the last statement.

- Example void function definition:

```
void print_TF(bool b)
{
    if (b) cout << "T";
    else   cout << "F";
}
```

Void Functions, *cont'd*

- A call to a void function cannot be part of an expression, since the function doesn't return
  a value.

- Instead, call a void function as a statement by itself.

- Example:

```
bool flag = true;
print_TF(flag);
```

Coding Convention with Functions

- First declare all your functions.
- Document each declaration with a comment that describes:
  - What the function does.
  - What is each function parameter.
  - What is the return value.
- Code the main function.
- Define the functions.
  - Don’t repeat the declaration’s comment.
  - Only document each function’s internal operations.

Coding Convention with Functions, *cont'd*

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

/**
 * Add two integers and return their sum.
 * @param n1 the first integer
 * @param n2 the second integer
 * @return their sum.
 */
int make_sum(int n1, int n2);

/**
 * Print an integer value;
 * @param n the value to print.
 */
void print(int n);
```

The declarations tell you <u>what</u> the functions will do and provide the overall structure of the program without all the details.

```
int main()
{
    int i = 5, j = 7;
    int sum = make_sum(i, j);
    print(sum);
}

int make_sum(int n1, int n2)
{
    return n1 + n2;  // return their sum
}

void print(int n)
{
    cout << "The value is " << n << endl;
}
```

Function definitions.

Break

Top-Down Design

- Top-down design is an important software engineering principle.

- Start with the topmost subproblem of a programming problem.
  - Write a function for solving the topmost subproblem.

- Break each subproblem into smaller subproblems.
  - Write a function to solve each subproblem.
  - This process is called stepwise refinement.

Top-Down Design*, cont’d*

- The result is a hierarchical decomposition of the problem.

- AKA functional decomposition

Top-Down Design Example

- Write a program that inputs from the user that are positive integer values less than 1000.

- Translate the value into words.

- Example:
  - The user enters **`482`**
  - The program writes **`four hundred eighty-two`**

- Repeat until the user enters a value ≤ 0.

Top-Down Design Example, *cont'd*

- What is the topmost problem?
  - Read numbers entered by the user until the user enters a value ≤ 0.
  - Translate each number to words.
- This is a high-level description of what the program is supposed to do.

Refinement 1

- Loop to read and print the numbers.

translator1.cpp

- Call a `translate` function,
  but it doesn't do anything yet.

Refinement 2

- How to translate a number into words?
  - Break the number into separate digits.
  - Translate the digits into words such as *one*, *two*, ..., *ten*, *eleven*, *twelve*, ..., *twenty*, *thirty*, etc.

- Refine the translate function to handle some simple cases:
  - **`translate_ones`**: 1 through 9
  - **`translate_teens`**: 11 through 19

translator2.cpp

- The translate function takes a 3-digit number and separates out the hundreds digit.

Refinement 3

- Translate the hundreds digit.
  - **`translate_hundreds`**
  - Do this simply by translating the hundreds digits as we did a ones digit. Then append the word **`hundred`**.

Refinement 3, *cont'd*

- Translate the last two digits:
  - We can already translate a teens number.
  - Otherwise, break apart the two digits into a tens digit and a ones digit.
    - **`translateTens`**: 10, 20, 30, ..., 90
    - We can already translate a ones digit.

translator3.cpp

Refinement 4

- Add a hyphen between *twenty*, *thirty*, etc. and a ones word.
  - Example: **`twenty-one`**

Refinement 5

- Break a 6-digit number into a 3-digit first part and a 3-digit second part.

- Translate the first part and
  then append the word **`thousand`**.

- Translate the second part.

Refinement 6? 7?

```
Number? 300010
300010 : three hundred  thousand ten
```

Extra space!

- Insert commas into numbers?
  - Example: `12,345`

Scope and Local Variables

- Any variable declared inside a function is local to that function.
  - The scope of the variable is that function.
  - The variable is not accessible from outside the function.
  - A variable with the same name declared inside another function is a different variable.
- The same is true for any variable declared inside the main function.

Block Scope

- You can declare variables inside of a block.
  - A block of code is delimited by **{** and **}**.

- The variables are local to the block.
  - Example:

```
if (x < y)
{
     int i;
     ...
}
```

Global Constants and Variables

- If a constant or a variable is declared outside of and before the main and the function definitions, then that constant or variable is global and accessible by the main and any function.
- Global variables are not recommended.
  - If a function modifies a global variable, that can affect other functions.
  - Such "side effects" of a function can make a program error-prone and difficult to maintain.
- Global constants are OK.

Overloading Function Names

- A function is characterized by both its name and its parameters.
  - A function's signature includes the number, order, and data types of the formal parameters.
- You can overload a function name by defining another function with the same name but with a different signature.
  - When you call a function with a shared name, the arguments of the call determine which function you mean.

Overloading Function Names, *cont'd*

- Example declarations:

```
double average(double n1, double n2);
double average(double n1, double n2, double n3);
```

- Example calls:

```
double avg2 = average(x, y);
double avg3 = average(x, y, z);
```

- Be careful with automatic type conversions of arguments when overloading function names.
  - See the Savitch text and slides.

Pass by Value

- By default, arguments to a function are passed by value.
  - AKA call by value

- A copy of the argument's value is passed to the function.

- Any changes that the function makes to the parameters do not affect the calling arguments.
  - Example: The faulty swap function.

Pass by Value, *cont'd*

```
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

swaps.cpp

- Why doesn't this function do what was intended?

Demo

Pass by Reference

- If you want the function to be able to change the value of the caller's arguments, you must use pass by reference.
  - AKA call by reference

- The address of the actual argument is passed to the function.
  - Example: The proper exchange function.

Pass by Reference, *cont'd*

```
void exchange(int& a, int& b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

swaps.cpp

- Why is this code better?

Demo

Procedural Abstraction

- Design your function such that the caller does not need to know how you implemented it.

- The function is a “black box”.

Procedural Abstraction, *cont'd*

- The function's name, its formal parameters, and your comments should be sufficient for the caller.
- Preconditions: What must be true when the function is called.
- Postconditions: What will be true after the function completes its execution.

Testing and Debugging Functions

- There are various techniques
to test and debug functions.

- You can add temporary **`cout`** statements in your functions to print the values of local variables to help you determine what the function is doing.

- With the Eclipse or the NetBeans IDE,
you can set breakpoints, watch variables, etc.

assert

- Use the **assert** macro during development to check that a function's preconditions hold.
  - You must first **#include <cassert>**
  - Example:

```
assert(y != 0);
quotient = x/y;
```

- Later, when you are sure that your program is debugged and you are going into production, you can logically remove all the asserts by defining **NDEBUG** before the include:

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

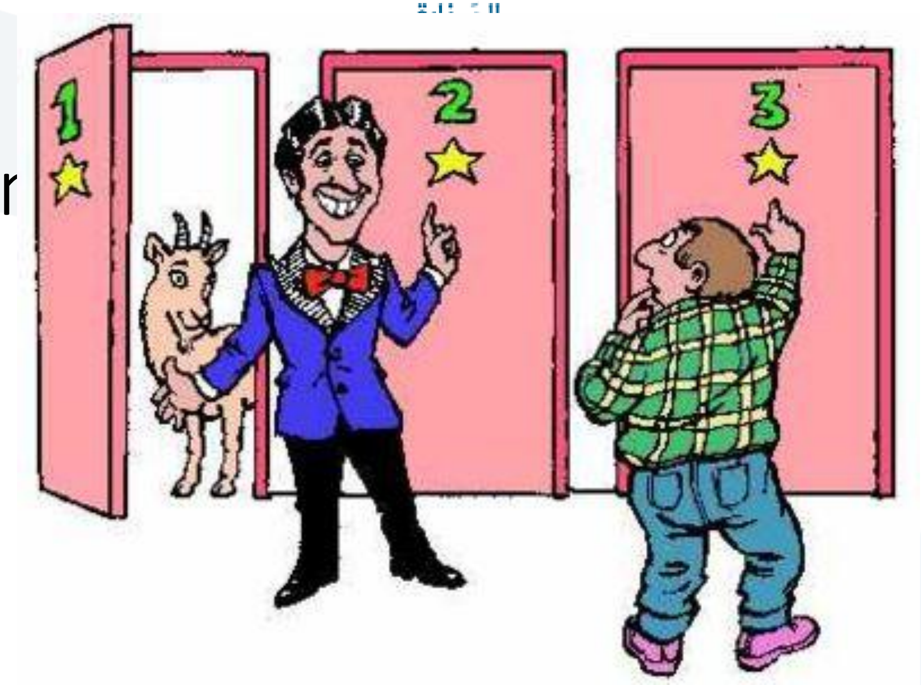
assert, *cont'd*

assert.cpp

```
#include <iostream>

//#define NDEBUG
#include <cassert>

using namespace std;

/**
 * Print a positive value.
 * @param n the value which must be > 0.
 */
void print_positive(int n);

int main()
{
    print_positive(-3);
    return 0;
}

void print_positive(int n)
{
    assert(n > 0);
    cout << "n = " << n << endl;
}
```

Demo

Assignmer



- Behind one door is a new car.
- Behind the other two doors are goats.
- Can you pick the right door?

Assignment #2: Monty Hall Problem, *cont'd*

- Do a hierarchical decomposition.
  - Iteratively add new functionality to code that works.
  - Choose good function names.
  - Use parameters wisely.
- You will need to generate random numbers.
  - Use the same seed value if you always want the same sequence of random numbers for testing.
- Your final program should have correct output and be easy to read.