COMPUTER SCIENCE 1: STARTING COMPUTING CSCI 1300

The image part with relationship ID rId3 was not found in the file.

Ioana Fleming / Vipra Gupta
Spring 2018
Lecture 7

Agenda

- C++
  - Data Types. Typecasting
  - Operators / Operator precedence
  - Conditional Statements – if-else

Data Types:
Simple Types (1 of 2)

Display 1.2 Simple Types

TYPE NAME	MEMORY USED	SIZE RANGE	PRECISION
`short` (also called `short int`)	2 bytes	−32,768 to 32,767	Not applicable
`int`	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`long` (also called `long int`)	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`float`	4 bytes	approximately 10^{-38} to 10^{38}	7 digits
`double`	8 bytes	approximately 10^{-308} to 10^{308}	15 digits

Data Types: Simple Types (2 of 2)

`long double`	10 bytes	approximately 10^{-4932} to 10^{4932}	19 digits
`char`	1 byte	All ASCII characters (Can also be used as an integer type, although we do not recommend doing so.)	Not applicable
`bool`	1 byte	`true`, `false`	Not applicable

The values listed here are only sample values to give you a general idea of how the types differ. The values for any of these entries may be different on your system. *Precision* refers to the number of meaningful digits, including digits in front of the decimal point. The ranges for the types `float`, `double`, and `long double` are the ranges for positive numbers. Negative numbers have a similar range, but with a negative sign in front of each number.

Assigning Data - recap

- **Declaring** a variable:
  - Does not need to have a value, but MUST have a type
    - int myValue;
- **Initializing** a variable in the declaration statement
  - Results "undefined" if you don't!
    - int myValue = 0;
- **Assigning** data during execution
  - Lvalues (left-side) & Rvalues (right-side)
    - Lvalues must be variables (variable names)
    - Rvalues can be any expression
    - Example:
      distance = rate * time;
      Lvalue: "distance"
      Rvalue: "rate * time"

Assigning Data: Shorthand Notations

EXAMPLE	EQUIVALENT TO
`count += 2;`	`count = count + 2;`
`total -= discount;`	`total = total - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rushFactor;`	`time = time/rushFactor;`
`change %= 100;`	`change = change % 100;`
`amount *= cnt1 + cnt2;`	`amount = amount * (cnt1 + cnt2);`

Data Assignment Rules

- Compatibility of Data Assignments
  - Type mismatches
    - General Rule: Cannot place value of one type into variable of another type

```
intVar = 2.99;    //2 is assigned to intVar!
```

    - Only integer part "fits", so that's all that goes
    - Called "implicit" or "automatic type conversion"
  - Literals
    - 2, 5.75, "Z", "Hello World"
    - Considered "constants"

```
char symbol = 'Z';
```

Literal Data

- Literals
  - Examples:
    - 2 // Literal constant int
    - 5.75 // Literal constant double
    - "Z" // Literal constant char
    - "Hello World" // Literal constant string
- Cannot change values during execution
- Called "literals" because you "literally typed" them in your program!

Constants

- Naming your constants
  - Literal constants are "OK", but provide little meaning
    - e.g., seeing 24 in a program, tells nothing about what it represents
- Style alert: **No magic numbers!**

Use named constants instead

- Meaningful name to represent data

```
const double TAXRATE = 0.05; //5% sales tax
```

```
(remember from the totalCost.cpp example)
```

- Called a "declared constant" or "named constant"
  - Now use it's name wherever needed in program
  - Added benefit: changes to value result in one fix

Arithmetic Precision

- Precision of Calculations
  - VERY important consideration!
    - Expressions in C++ might not evaluate as you'd "expect"!
  - "Highest-order operand" determines type of arithmetic "precision" performed
  - Common pitfall!

Arithmetic Precision Examples

- Examples:
  - 17 / 5 evaluates to 3 in C++!
    - Both operands are integers
    - Integer division is performed!
  - 17.0 / 5 equals 3.4 in C++!
    - Highest-order operand is “double” type
    - Double "precision" division is performed!
  - int intVar1 =1, intVar2=2;
    intVar1 / intVar2;
    - Performs integer division!
    - Result: 0!

Individual Arithmetic Precision

- Calculations done "one-by-one"
  - 1 / 2 / 3.0 / 4 performs 3 separate divisions.
    - First→ 1 / 2   equals 0
    - Then→ 0 / 3.0 equals 0.0
    - Then→ 0.0 / 4 equals 0.0!
- So not necessarily sufficient to change just "one operand" in a large expression
  - Must keep in mind all individual calculations that will be performed during evaluation!

Type Casting

- Two types
  - Implicit—also called "Automatic"
    - Done FOR you, automatically
      17 / 5.5
      This expression causes an "implicit type cast" to take place, casting the 17 → 17.0
  - Explicit type conversion
    - Programmer specifies conversion with cast operator
      (double)17 / 5.5
      Same expression as above, using explicit cast
      (double)myInt / myDouble
      More typical use; cast operator on variable

Cloud9 example: *types_variables.cpp*

Shorthand Operators

Increment & Decrement Operators

1. Increment operator, ++

```
intVar++;  //is equivalent to
intVar = intVar + 1;
```

2. Decrement operator, --

```
intVar--;   //is equivalent to
intVar = intVar - 1;
```

Shorthand Operators: Two Options

- **Post**-Increment: `intVar++`
  - Uses current value of variable, THEN increments it
- **Pre**-Increment: `++intVar`
  - Increments variable first, THEN uses new value

- "Use" is defined as whatever "context" variable is currently in

- No difference if "alone" in statement: intVar++; and ++intVar; → identical result

Post-Increment in Action

- Post-Increment in Expressions:

```
int      n = 2, valueProduced;
valueProduced = 2 * (n++);
cout << valueProduced << endl;
cout << n << endl;
```

This code segment produces the output:
4
3
, since post-increment was used.

Pre-Increment in Action

- Now using Pre-increment:

```
int      n = 2, valueProduced;
valueProduced = 2 * (++n);
cout << valueProduced << endl;
cout << n << endl;
```

This code segment produces the output:
6
3
, because pre-increment was used

Cloud9 example: *pre_post_increment.cpp*

Control Flow: Learning Objectives

- Boolean Expressions
  - Building, Evaluating & Precedence Rules
- Branching Mechanisms
  - if-else
  - switch
  - Nesting if-else
- Loops
  - While, do-while, for
  - Nesting loops

Boolean Expressions

- Data type bool
  - Returns true or false
  - true, false are predefined library consts

Boolean Expressions: Comparison Operators

1. Comparison Operators: ==, <, >, !=, <=, >=
2. Logical Operators
   - Logical AND (&&)
   - Logical OR (||)

Display 2.1 Comparison Operators

MATH SYMBOL	ENGLISH	C++ NOTATION	C++ SAMPLE	MATH EQUIVALENT
=	Equal to	`==`	`x + 7 == 2*y`	$x + 7 = 2y$
≠	Not equal to	`!=`	`ans != 'n'`	*ans* ≠ *'n'*
<	Less than	`<`	`count < m + 3`	*count* < m + 3
≤	Less than or equal to	`<=`	`time <= limit`	*time* ≤ *limit*
>	Greater than	`>`	`time > limit`	*time* > *limit*
≥	Greater than or equal to	`>=`	`age >= 21`	*age* ≥ 21

Evaluating Boolean Expressions: Truth Tables

Display 2.2 **Truth Tables**

AND

Exp_1	*Exp_2*	*Exp_1 && Exp_2*
true	true	true
true	false	false
false	true	false
false	false	false

OR

Exp_1	*Exp_2*	*Exp_1 || Exp_2*
true	true	true
true	false	true
false	true	true
false	false	false

NOT

Exp	!(*Exp*)
true	false
false	true

Precedence Examples

- Arithmetic before logical
  - x + 1 > 2 || x + 1 < -3 means:
    - (x + 1) > 2 || (x + 1) < -3
- Short-circuit evaluation
  - (x >= 0) && (y > 1)
  - Be careful with increment operators!
    - (x > 1) && (y++)
- Integers as boolean values
  - All non-zero values → true
  - Zero value → false

Branching Mechanisms

- if-else statements
  - Choice of two alternate statements based on condition expression
  - Example:

```
if (hrs > 40)
 grossPay = rate*40 + 1.5*rate*(hrs-40);
else
 grossPay = rate*hrs;
```

if-else Statement Syntax

- Formal syntax:
```
if (<boolean_expression>)
 <yes_statement>
else
 <no_statement>
```
- Note each alternative is only ONE statement!
- To have multiple statements execute in either branch → use compound statement and { }

Compound/Block Statement

- Only "get" one statement per branch
- Must use compound statement { } for multiples
  - Also called a "block" statement
- Each block should have block statement
  - Even if just one statement
  - Enhances readability

Compound Statement in Action

- Note indenting in this example:

```
if (myScore > yourScore)
{
 cout << "I win!\n";
 wager = wager + 100;
}
else
{
 cout << "I wish these were golf scores.\n";
 wager = 0;
}
```

Common Pitfalls

- Operator "=" vs. operator "=="
- One means "assignment" (=)
- One means "equality" (==)
  - VERY different in C++!
  - Example:

```
if (x = 12)  ←Note operator used!
    Do_Something
else
    Do_Something_Else
```

The Optional else

- else clause is optional
  - If, in the false branch (else), you want "nothing" to happen, leave it out
  - Example:

```
if (sales >= minimum)
     salary = salary + bonus;
cout << "Salary = " << salary;
```

  - Note: nothing to do for false condition, so there is no else clause!
  - Execution continues with cout statement