Looping and Counting

Lecture 3

Hartmut Kaiser

hkaiser@cct.lsu.edu

http://www.cct.lsu.edu/~hkaiser/spring_2015/csc1254.html
Abstract

• First we’ll discuss types and type safety.
• Then we will modify the program we developed last time (Framing a Name) to make it more flexible. We will touch on arithmetic expressions, looping constructs and conditions. In addition we will talk about loop invariants and counting.
Types

- C++ provides a set of types
  - E.g. bool, char, int, double
  - Called “built-in types”

- C++ programmers can define new types
  - Called “user-defined types”
  - We'll get to that eventually

- The C++ standard library provides a set of types
  - E.g. string, vector, complex
  - Technically, these are user-defined types
    - they are built using only facilities available to every user
Types and Literals

- Built-in types
  - Boolean type
    - bool
  - Character types
    - char
  - Integer types
    - int
      - and short and long
  - Floating-point types
    - double
      - and float
- Standard-library types
  - std::string
  - std::complex<Scalar>

- Boolean literals
  - true, false
- Character literals
  - 'a', 'x', '4', '\n', '$'
- Integer literals
  - 0, 1, 123, -6, 0x34, 0xa3
- Floating-point literals
  - 1.2, 13.345, .3, -0.54, 1.2e3F, 3F, .3F
- String literals: "asdf",
  "Howdy, all y'all!"
- Complex literals
  - std::complex<double>(12.3,99)
  - std::complex<float>(1.3F)
Types and Value Ranges

- int
- short
- long
- double
- float
- char
- string

- 4 bytes: $-2^{31}...+2^{31}-1$
- 2 bytes: $-2^{15}...+2^{15}-1$
- 8 bytes: $-2^{63}...+2^{63}-1$
- 8 bytes: $-1.8e+308 ... 1.8e+308$, 15 digits
- 4 bytes: $-3.4e+38 ... 3.4e+38$, 6 digits
- 1 byte: $-2^7...+2^7-1$
- Arbitrary length character sequence
Types and Objects

• A type
  • Defines a set of values and a set of operations (for an object)

• An object
  • Is some memory that holds a value of a given type

• A value
  • Is a set of bits in memory interpreted according to a type

• A variable
  • Is a named object

• A declaration
  • Is a statement giving a name to an object

• A definition
  • Is a declaration that sets aside memory for an object
Definition and Initialization

```cpp
int a = 7;

int b = 9;

char c = 'a';

double x = 1.2;

std::string s1 = "Hello";

std::string s2 = "1.2";
```
Objects

• An object is some memory that can hold a value (instance) of a given type
• A variable is a named object
• A declaration names an object

int a = 7;
char c = 'x';
complex<double> z(1.0, 2.0);
string s = "qwerty";
Type safety

- Language rule: type safety
  - Every object will be used only according to its type
    - A variable will be used only after it has been initialized
    - Only operations defined for the variable's declared type will be applied
    - Every operation defined for a variable leaves the variable with a valid value

- Ideal: static type safety
  - A program that violates type safety will not compile
    - The compiler reports every violation (in an ideal system)

- Ideal: dynamic type safety
  - If you write a program that violates type safety it will be detected at run time
    - Some code (typically "the run-time system") detects every violation not found by the compiler (in an ideal system)
Type safety

• Type safety is a very big deal
  • Try very hard not to violate it
  • “when you program, the compiler is your best friend”
    • But it won’t feel like that when it rejects code you’re sure is correct

• C++ is not (completely) statically type safe
  • No widely-used language is (completely) statically type safe
  • Being completely statically type safe may interfere with your ability to express ideas

• C++ is not (completely) dynamically type safe
  • Many languages are dynamically type safe
  • Being completely dynamically type safe may interfere with the ability to express ideas and often makes generated code bigger and/or slower

• Most of what you’ll be taught here is type safe
  • We’ll specifically mention anything that is not
The Problem

• Given the following interaction:

  Please enter your name: John

• We would like to print:

  *******************
  *                 *
  *   Hello John!   *
  *                 *
  *******************
The Problem

- Problems with our Solution
  - Each printed line has its own variable associated
  - Even simple change to format requires rewrite

- Solution to generate each character separately
  - No need to store strings
Overall Structure

// Ask a persons name, greet the person
#include <iostream>
#include <string>

int main()
{

    // ask for the persons name
    std::cout << "Please enter your first name: ";

    // read into 'first_name'
    std::string first_name;
    std::cin >> first_name;

    // build the message we intend to write
    std::string const greeting = "Hello, " + first_name + "!";

    // we have to rewrite this part...

    return 0;
}
Overall Structure

• Writing an unknown number of rows

```cpp
int const line_pad = 1;       // number of blank lines surrounding greeting
int const column_pad = 2;    // number of blank columns surrounding greeting

// total number of rows to write
int const rows = line_pad * 2 + 3;

// write 'rows' rows of output
int r = 0;

// invariant: we have written 'r' rows of output
while (r != rows) {
    // ... write a row of output ...
    std::cout << std::endl;
    ++r; // increment r, equivalent to r = r + 1
}
```
The while Statement

• The while statement

```
while (condition)
    statement
```

• If `condition` is false, `statement` will not be executed
• If `condition` is true, `statement` will be executed once, after which `condition` is re-checked
• The `statement` is either a single statement or a block (`{...}`) of several statements
Understanding while Statements

• After while finished the condition must be false
  • i.e. $r == \text{rows}$ (because $r != \text{rows}$ is false)

• Loop invariant
  • A property of the loop to be true each time the condition is about to be checked
  • Write program to make sure invariant holds all the time
The Loop Invariant

// invariant: we have written 'r' rows so far

// write 'rows' rows of output
int r = 0; // makes the invariant true

// invariant: we have written 'r' rows of output
while (r != rows) {
    // we can assume that the invariant is true here

    // write a row of output, makes invariant false
    std::cout << std::endl;

    ++r; // increment r, makes invariant true again
}

// we can conclude that the invariant is true here
Writing a Row

• All rows have the same length:
  
  ```
  std::string::size_type const cols =
      greeting.size() + column_pad * 2 + 2;
  ```

• Writing a row:
  
  ```
  // write 'cols' columns of output
  std::string::size_type c = 0;

  // invariant: we have written 'c' columns of output
  while (c != cols) {
      // write one or more characters
      // adjust the value of c
  }
  ```
Writing Border Characters

// write 'cols' columns of output
std::string::size_type c = 0;

// invariant: we have written 'c' columns of
//            output
while (c != cols) {
    if (r == 0 || r == rows-1 || c == 0 || c == cols-1) {
        std::cout << '*';
        ++c;
    } else {
        // write one or more non-border characters
        // adjust the value of c
    }
}
The if Statement

• The if statement
  
  ```java
  if (condition)
  statement1
  ```

• Or
  
  ```java
  if (condition)
  statement1
  else
  statement2
  ```

• If `condition` is true, `statement1` will be executed, otherwise `statement2` (or nothing)
Logical Operators

- **Condition**

  \[ r == 0 \text{ || } r == \text{rows}-1 \text{ || } c == 0 \text{ || } c == \text{cols}-1 \]

  - is true if either \((r == 0)\) or \((r == \text{rows}-1)\) or \((c == 0)\) or \((c == \text{cols}-1)\) is true

- **Left associative**
  - Evaluated left to right

- **Short cutting**
  - Evaluation stops as soon as one test is true
Writing Non-Border Characters

• Writing greeting is special, the rest are spaces:

```cpp
if (r == line_pad + 1 && c == column_pad + 1) {
    std::cout << greeting;
    c += greeting.size();
}
else {
    std::cout << ' ';
    ++c;
}
```
Being more concise

• The while statement:

```java
int r = 0;

while (r != rows) {
    // stuff that doesn't change value of 'r'
    ++r;
}
```

• Can be written as:

```java
for (int r = 0; r != rows; ++r) {
    // stuff that doesn't change value of 'r'
}
```
The for Statement

- The for statement

\[
\text{for (init-statement condition; expression) }
\text{statement}
\]

- The for statement starts executing the \textit{init-statement} (done once)
- The \textit{condition} is evaluated before \textit{statement} is executed, including the first time
- The \textit{statement} is executed only if \textit{condition} evaluated to true
- The \textit{expression} is executed after \textit{statement}
The for Statement

- The for Statement is equivalent to:

```
{  
    init-statement
    while (condition) {
        statement
        expression;
    }
}
```
The for Statement
Abbreviating use of std:::

• Writing std:: everywhere is tedious
• Either use
  
  ```
  using namespace std;
  ```
  • Imports all names from std:: into current scope
  • But never do that in a header file!

• Or import individual names
  
  ```
  using std::cout;
  ```
  • Imports std::cout from std:: into current scope

• Using declaration is in effect for the scope it is used in
Collapsing the Tests

• Reordering simplifies code:

```cpp
// test for greeting
using std::cout;
if (r == line_pad + 1 && c == column_pad + 1) {
    cout << greeting;
    c += greeting.size();
} else {
    // test for border
    if (r == 0 || r == rows-1 || c == 0 || c == cols-1)
        cout << '*';
    else
        cout << ' ';
    ++c;
}
```
Counting

• We’ve used:
  
  ```java
  for (int r = 0; r != rows; ++r) {
      // write a row
  }
  ```

• What about:
  
  ```java
  for (int r = 1; r <= rows; ++r) {
      // write a row
  }
  ```
Counting

• Why counting from zero?
  • Open range \([0, n)\) easier to understand than closed range \([1, n]\)
    • Use of operator\(!=\) vs. operator\(<=\)
    • Open range \([n, m)\) has \(m - n\) elements
    • Closed range \([n, m]\) has \(m - n - 1\) elements
  • Consider empty range: \([n, n)\) vs. \([n, n-1]\)
  • Consider loop invariances
    • What does it mean to have \(r \in [1, \text{rows}]\)
      • “\(r\) is the number of rows to be written”?
      • “\(r-1\) is the number of rows written”? 
Conclusions

• We have seen C++ has a rich set of operators, flexible input and output, and how to extend the core language by defining what it means to apply build-in operators to user defined types.