

Java vs. C++

Part 3

+ Review

- Expressions
 - Boolean Expressions
 - Relational Operators
 - Logical Operators
 - Assignment as expression

+ Expressions

- Numeric constants and variables
E.g., 1, 1.23, x
- Value-returning functions
E.g., `getchar()`
- Expressions connected by an *operator*
E.g., `1 + 2, x * 2, getchar() - 1`
- All expressions have a type


+ Boolean Expressions

- C does not have type boolean
- False is represented by integer 0
- Any expression evaluates to non-zero is considered true
- True is typically represented by 1 however

+ Relational Expressions

- Equality/Inequality
 - `if (x == 1)`
 - `if (x != 1)`
- Relation
 - `if (x > 0)`
 - `if (x >= 0)`
 - `if (x < 0)`
 - `if (x <= 0)`

\neq
 $>$
 \geq
 $<$
 \leq



`==` (equality)
`=` (assignment)

The values are internally represented as integer.
true \rightarrow 1 (not 0), false \rightarrow 0

+ Complex Condition (Logical Operators)

- And
 - `if ((x > 0) && (x <= 10))` `0 < x <= 10`
- Or
 - `if ((x > 10) || (x < -10))` `|x| > 10`
- Negation
 - `if (!(x > 0))` `not (x > 0) \Leftrightarrow x \leq 0`

Beware that `&` and `|` are also C operators

+ Assignment as Expression

■ Assignment

- Assignments are expressions
- Evaluates to value being assigned

■ Example

```
int x = 1, y = 2, z = 3;
x = (y = z);
```

3 ← 3 ← 3
evaluates to 3

evaluates to 3 (true)
if (x = 3) {
...
}

+ if-else Statement

```
int main() {
    int choice;
    scanf("%d", &choice); //user input

    if (choice == 1) {
        printf("The choice was 1.\n");
    }
    else {
        printf("The choice wasn't 1.\n");
    }
    return 0;
}
```

+ Lazy Logical Operator Evaluation

- If the conditions are sufficient to evaluate the entire expression, the evaluation terminates at that point
=> **lazy**



• Examples

```
if ((x > 0) && (x <= 10))
    Terminates if (x > 0) fails
if ((x > 10) && (x < 20)) || (x < -10))
    Terminates if (x > 10) && (x < 20) succeeds
```

+ Use of Braces

```
if (choice == 1) {
    printf("1\n");
}
else {
    printf("Other\n");
}
```

```
if (choice == 1)
    printf("1\n");
else
    printf("Other\n");
```

When the operation is a single statement, '{' and '}' can be omitted. But, don't!!!

+ switch Statement

```
switch (integer expression) {
    case constant:
        statements
        break;
    case constant:
        statements
        break;
    possibly more cases
    default:
        statements
}
```

Multi-branching



+ break Fall Through

- Omitting **break** in a **switch** statement will cause program control to fall through to the next case
- Can be a very convenient feature
- Also generates very subtle bugs
- **switch** statements only test equality with integers

+ Example

```
int x, y, result = 0; cin >> x >> y;
switch(x) {
    case 1: break;
    case 2:
    case 3: result = 100;
    case 4:
        switch(y) {
            case 5: result += 200; break;
            default: result = -200; break;
        }
        break;
    default: result = 400; break;
}
```

+ while Loops

```
while (true) {
    /* some operation */
}
```

+ while and Character Input

- `std::char_traits<char>::eof()` is a constant defined in `iostream`
- `eof` - stands for End Of File

```
int main() {
    string read;
    while (!cin.fail()) {
        int next = cin.get();
        if(next == std::char_traits<char>::eof()) {
            break;
        }
        read += static_cast<char>(next); // Append character.
    }
    cout << " read: " << read << endl;
    return 0;
}
```

pg. 354

+ while and Character Input alternate

- no need for failure checking.

```
int main() {
    string read;
    char next;
    while (cin.get(next)) {
        read += next; // Append character.
    }
    cout << " read: " << read << endl;
    return 0;
}
```

pg. 355

+ while and Character Input alternate2

- no need for failure checking.
- uses `>>` operator

```
int main() {
    string read;
    char next;
    while (cin >> noskipws >> next) {
        read += next; // Append character.
    }
    cout << " read: " << read << endl;
    return 0;
}
```

+ while and Character Input alternate3

- use an assignment as expression.

```
int main() {
    string read;
    int next;
    while ((next = cin.get()) !=
        std::char_traits<char>::eof()) {
        read += static_cast<char>(next); // Append character.
    }
    cout << " read: " << read << endl;
    return 0;
}
```

+ Review: Assignment has value

- In C++, assignment expression has a value, which is the value of the lefthand side after assignment.
- Parens in `{next = cin.get()} != std::char_traits<char>::eof()` are necessary.
- `next = cin.get() != std::char_traits<char>::eof()` is equivalent to

```
next = (cin.get() !=
std::char_traits<char>::eof())
```
- `next` gets assigned 0 or 1.

+ Strings

Java	C++
<pre>public class StringTest { public static void main(String[] args) { String x = "hello"; String y = "world"; String z = x + ", " + y + "!\n"; System.out.print(z); } }</pre>	<pre>#include <string> #include <iostream> using namespace std; int main() { string x = "hello"; string y = "world"; string z = x + ", " + y + "!\n"; cout << z; }</pre>

+ C-Style Strings

■ `char* x = "hello";`

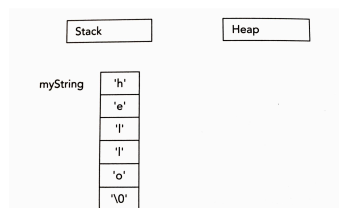


FIGURE 2-1

+ C-Style Strings

■ `char* x = "hello";`

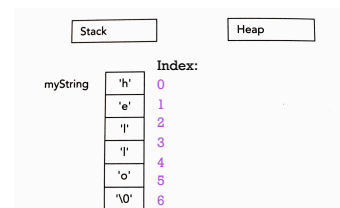


FIGURE 2-1

+ Streams

- | Java | C++ |
|---|---|
| Input <ul style="list-style-type: none"> System.in – the stream Scanner =
 <code>new Scanner(System.in);</code> (must use exceptions) Output <ul style="list-style-type: none"> System.out – the stream System.err – the stream | Input <ul style="list-style-type: none"> cin – the stream >> – the operator <ul style="list-style-type: none"> works on all primitive types default delimiter is whitespace Output <ul style="list-style-type: none"> cout – the stream cerr – the stream << – the operator <ul style="list-style-type: none"> works on all primitive types |

+ cin

- methods (pgs 354-357)
 - fail() – the stream has reached a bad state
 - get() – gets the next character in the stream
 - last char is `std::char_traits<char>::eof()`
 - unget() – puts the last character back on the stream
 - putback(char p) – puts p at the beginning of the stream
 - peek() – shows what next will give without removing it from the stream
 - getline(char[] buffer, int bufferSize) – takes up to bufferSize length line off of the stream and puts it in buffer

+ cin

- input manipulators (pg. 358)
 - used in conjunction with >> operator
 - boolalpha – reads false as false and anything else as true
 - noboolalpha* – reads 0 as false and anything else as true
 - hex, oct, dec* – specify the numeric base (16, 8, or 10).
 - skipws* – skip over all the whitespace in the stream
 - noskipws – don't skip all of the whitespace in the stream
 - ws – skip the current sequence of whitespace at the current loc.
 - get_money – reads money format
 - get_time – reads time format
 - quoted – reads a string in quotes
 - (* means default)

+ cout

- output manipulators (pg. 351)
 - most are complementary to cin
 - setprecision – number of decimal places to use
 - setw – the field width of numerical output
 - setfill – the character to pad the output of a number smaller than the value of setw
 - showpoint – show the decimal point of the floating point number
 - noshowpoint – don't show the decimal point of the floating point number with no fractional part.