

Overview of C++ Execution Control

- Expressions vs. statements
- Arithmetic operators and expressions
* / % + -
- Relational operators and expressions
== <= >= < > !=
Watch out for the “future equivalence operator”, =
- Logical operators and expressions
! && ||
- Assignment operators and statements
= *= /= %= += -= &= |=
- Execution control in a nutshell
- Iteration: increment and decrement operators
- Loop control statements
for, while, do while, continue, break
- Conditional control statements
if, else, else if, operator ?:, switch

Expressions vs. Statements

- A statement can stand alone in a C++ program (**compilable**)
 - Or can be part of another statement
 - Examples include

```
i = 7;  
if (argc < 3) { usage(); }  
usage ();  
cout << "hello, world!" << endl;
```
- An expression cannot stand alone
 - But rather must be part of a statement
 - Examples include

```
argc < 3  
a + b - 7
```
- Some expressions are easily made into statements
 - By adding a semicolon, the return keyword, etc.
 - Examples include

```
i++ → i++;  
i < 3 → return i < 3;
```

Arithmetic Operators and Expressions

- Semantics (meaning) varies with the types of the operands
- Result type also depends on the types of the operands
- Operators * / % have higher precedence than (binary) + -
- See pp. 1058-1061 in textbook (Prata 5th Edition)
- Semantics are similar for some operators

* + -

- Watch out: over/under-flow, and some operators differ more /

- Modulus (remainder) operator can be very useful

```
if (a % 2 == 0) {  
    // even number  
} else {  
    // odd number  
}
```

Relation Operators and Expressions

- Relational operators:
== <= >= < > !=
- Very useful for control execution
- Test relationships between values of variables
- Return type is **bool**
- Can be used to build complex Boolean expressions
- Watch out for the “future equivalence operator”, =, when you meant ==

Logical Operators and Expressions

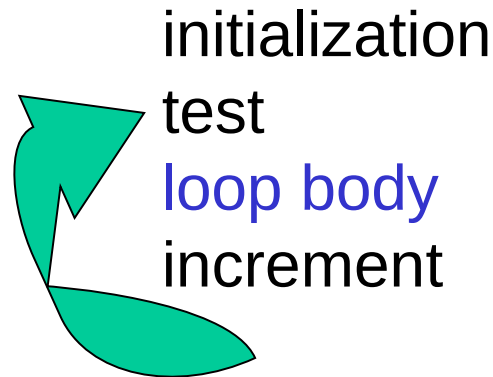
- Logical operators (high→low precedence)
! && ||
- Input and result types are both **bool**
- Often used with relational operators
- Help to build logical formulas
- Very useful for execution control

Assignment Operators and Statements

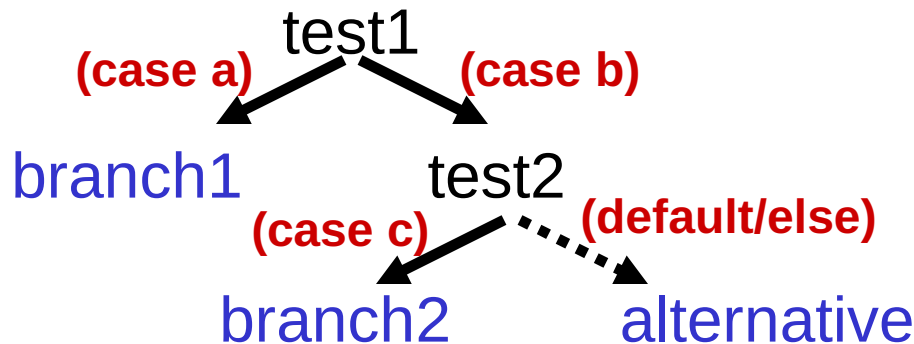
- Basic assignment operator is =
 - Statement made up of two parts (one on either side of the operator)
 - Left hand side is called an *l-value*
 - Must be mutable (non-const variable visible in scope)
 - Result of right hand side is called an *r-value*
 - Any well formed expression of type matching l-value
 - Example
 - a = b + 9; // b may be const, a cannot be.**
- Other assignment operators do some other operation on r-value, then assign the result
 - May be slightly more efficient than expanded form
 - May introduce or reduce operator side effects
 - Examples
 - *= /= %= += -= &= |=**

Execution Control in a Nutshell

- Loops: return control to top of loop, under a given condition
 - Overall structure (details differ between different kinds of loops):



- Conditionals: select from alternatives, under given conditions
 - Overall structure (details differ between different kinds of branches):



Iteration, Progress, Termination

- *Iteration* is fundamental to loop control
 - Loop is a (hopefully ;-)) bounded series of steps
- Iteration *progresses* through a *range* of steps
 - Loop is unbounded if no progress is made
 - For example if someone left off the increment: `++i;`
 - Loop is unbounded if range does *terminate*
 - For example, if test is badly formed: `while (c < 32767)`
- Three main questions to answer
 - How does the loop's range start
 - How does the loop's range stop
 - How to move from one step to the next
- Many kinds of ranges
 - Examples include integer, exponent, logical

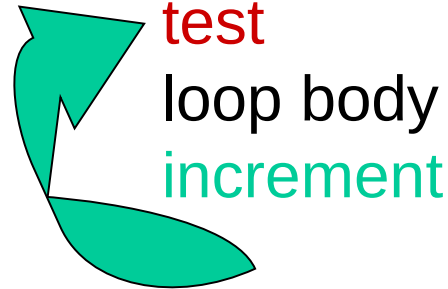
Increment and Decrement Operators

- Useful for iteration
 - Increase or decrease a pointer, index, or value
 - Steps through a logical “range” (more later on STL)
- A few details that we’ll revisit throughout the course
 - Increment (++) increases value/position/index
 - Decrement (--) decreases value/position/index
 - Prefix form is called with operator to left of operand
`++i` `--i`
 - Postfix form is called with operator to right of operand
`i++` `i--`
 - Prefix form takes no arguments when declared
`operator++ ()` `operator-- ()`
 - Postfix form takes an int argument when declared
`operator++ (int)` `operator-- (int)`
 - Prefix returns “after” value, postfix returns “before” value

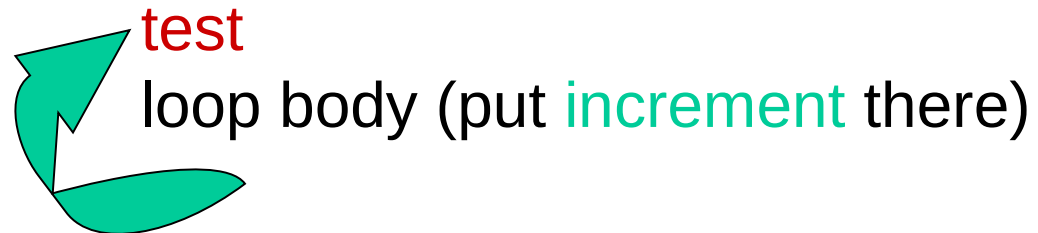
Loop Control Statements

- For loops

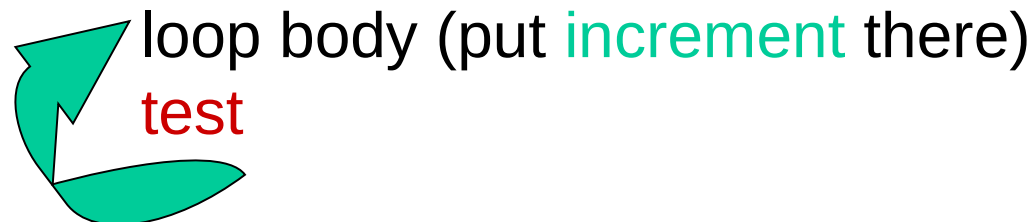
initialization



- While loops



- Do loops



For Loop with Simple Integer Range

```
for (int i = 0; i < 10; ++i) {  
    cout << i << endl;  
}
```

Output:

0
1
2
3
4
5
6
7
8
9

For Loop with Exponential Range

```
for (int i = 1; i < 1024; i *= 2 ) {  
    cout << i << endl;  
}
```

Output:

```
1  
2  
4  
8  
16  
32  
64  
128  
256  
512
```

While Loop with Logical Range

```
int balance = 2000;
while (balance > 0) {
    int withdrawal;
    cout << "your balance is now: "
         << balance << endl <<
         "how much would you like to withdraw? ";
    cin >> withdrawal;
    balance -= withdrawal;
}
```

Output:

```
your balance is now: 2000
how much would you like to withdraw? 1270
your balance is now: 730
how much would you like to withdraw? 700
your balance is now: 30
how much would you like to withdraw? 30
```

Do Loop with Logical Range

```
char c; // can be in any scope visible to loop
do {
    cout << "Enter Q or q to quit: " << endl;
    cin >> c;
} while (c != 'Q' && c != 'q');
```

Output:

Enter Q or q to quit:

a

Enter Q or q to quit:

B

Enter Q or q to quit:

q

Converting Between For and While Loops

```
for (int i = 1; i < 1024; i *= 2) {  
    cout << i << endl;  
}
```

```
int i = 1;  
while (i < 1024) {  
    cout << i << endl;  
    i *= 2;  
}
```

Converting Between Do and While Loops

```
char c; // simple declaration fine, could be inside  
do {  
    cout << "Enter Q or q to quit: " << endl;  
    cin >> c;  
} while (c != 'Q' && c != 'q');
```

```
char c = 0; // must initialize outside (why?)  
while (c != 'Q' && c != 'q') {  
    cout << "Enter Q or q to quit: " << endl;  
    cin >> c;  
}
```

Continue and Break Statements

- Two additional statements used for loop control
- Put them within body of loop to control execution
- Continue statement
 - Skips the rest of the loop
 - Control moves directly to loop's test statement
 - Useful when testing validity of user inputs, for example
 - Syntax:
`continue;`
- Break statement
 - Leaves the loop mid-way through its body
 - Control moves to the next statement after the loop
 - Useful when conditions can change mid-way through loop
 - Syntax:
`break;`

If, Else, Conditional Control Statements

- If / else if / else statements

```
if (a < b) {  
    cout << a << " < " b << endl;  
} else if (a > b) {  
    cout << a << " > " b << endl;  
} else {  
    cout << a << " == " b << endl;  
}
```

- Conditional operator

```
return (a < b) ? a : b;
```

Switch Statements

- Use when all cases are based on values of a single expression – notice default case, break statements

```
switch (num % 4) {  
    case 0:  
        cout << num <<  
            " is divisible by both 2 and 4" << endl;  
        break;  
    case 2:  
        cout << num <<  
            " is divisible by 2 but not by 4" << endl;  
        break;  
    default:  
        cout << num <<  
            " is not divisible by 2 or by 4" << endl;  
        break;  
}
```

Summary: Execution Control in C++

- Expressions
 - Built from operators and operands
- Statements
 - Built from expressions (operators, operands)
- Execution control determines order in which program statements are run
- Conditional control statements allow choices
- Loop control statements allow repetition
- Things to think about in every program
 - When is a branch or loop executed?
 - If it's a loop, when does it stop?
 - If it's a loop, is progress made each time?

For Next Time

- Topic: C++ functions
- Assigned reading
 - pages 279-340
 - pages 362-365