11. Input Validation, Arithmetic Operators, Assignment Operators

CPSC 120: Introduction to Programming
Pratishtha Soni~ CSU Fullerton

Agenda

0. Sign-in sheet
1. Technical Q&A
2. Input Validation
3. Arithmetic Operators
4. Assignment Operators

1. Technical Q&A

Technical Q&A

Let’s hear your noted questions about...

- This week’s Lab
- Linux
- Any other technical issues

Reminder: write these questions in your notebook during lab

1. Input Validation

Happy Paths and Sad Paths

- **Path:** sequence of statements executed for a specific input
- **Happy path:** everything works as programmer expected
- **Sad path:** something went wrong
- So far: our programs assume happy path
- **Defensive programming:** writing code that anticipates and handles problems
- Expected by users
    - They need command errors to debug their own problems

Extraction Failure

- Recall: extraction operator >> **may fail**
- Happens when typed-in characters do not match data type

Example: Extraction Failure

```
int main(int argc, char* argv[]) {

double price{0.0};
std::cout << "Enter price: ";
std::cin >> price;

int servings{0};
std::cout << "Enter servings: ";
std::cin >> servings;

std::cout << "Price per serving: "
          << price / servings << "\n";

return 0;
}
```

Happy path:

```
$ ./a.out
Enter price: 6.99
Enter servings: 12
Price per serving: 0.5825
```

Sad path:

```
$ ./a.out
Enter price: free
Enter servings: Price per serving: nan
```

Review: Syntax: cin Expression

expression:

std::cin *extract-expression...*

extract-expression:

>> *variable*

In left-to-right order, for each *variable:*

- If cin already **failed**: do nothing
- Otherwise:
  - Skip whitespace, read characters from standard input
  - If they represent an object of *variable*'s type: store that object in *variable*
  - Otherwise: cin is **failed**; leave *variable* unchanged

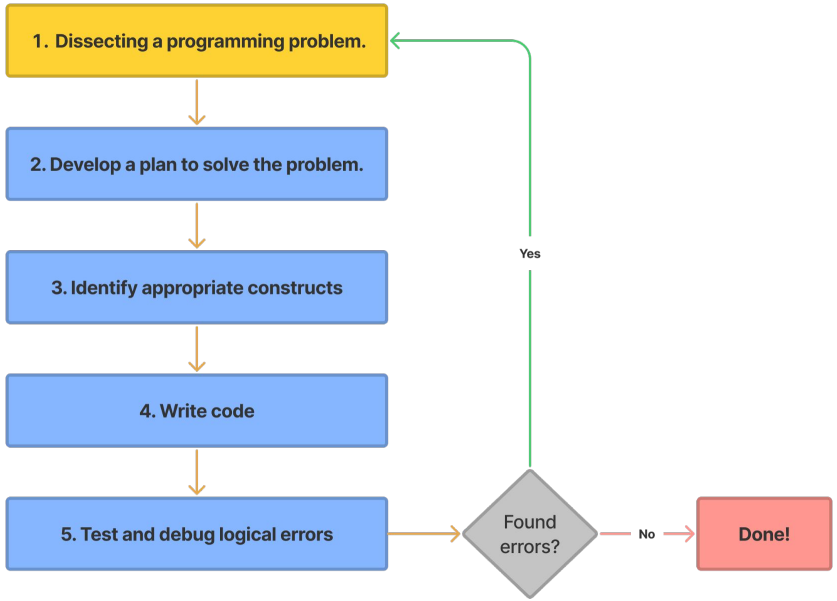
cin expression in expression statements:

```
int year{ 0 };

std::cout << “Enter year: “;

std::cin >> year;
```

Input Validation

- **Valid input:** user input
  - exists
  - proper data type
  - proper value
- **Invalid input:** not valid
- **Input validation:**
  - program checks if input is valid
  - when invalid, provides command error and exit code
- Two kinds of invalid input:
  - Extraction failure
  - Range errors

Steps for Solving a Programming Problem



Extraction Validation Algorithm

1. *Read input*
2. if *cin is in the failed state*:
    a. *Print command error*
    b. *Communicate error exit code to operating system*

Review: Syntax: if statement

statement:

if (*condition-expr* **)** *true-statement*
else-clause(optional)

else-clause:

else *false-statement*

Semantics:

1. Evaluate *condition-expr* and convert result to `bool`
2. If result is true: execute *true-statement*
3. Otherwise, execute *false-statement* if it exists

Examples:

```
if (lives == 0)
  std::cout << “Game over”;
```

```
if (age >= 18)
  std::cout << “adult”;
else
  std::cout << “minor”;
```

Review: Conversion to `bool`

Data Type	`bool` Conversion Semantics
int	Non-zero is true, zero is false
double	Non-zero is true, zero is false
cin	good is `true`; failed is `false`
string	Not available

Syntax: static_cast function call

expression:

static_cast<*target-type*>(*expression*)

Semantics:

1. Returns a value of type *target-type*

Example:

```
double a{2.3};
int b{5};
std::cout << static_cast<int>(a * b);
```

Output:

11

(not 11.5)

Review: Relational Operators

Operator	**Semantics**	**Example** (x and y are same type)
==	Equal to	`x == y`
!=	Not equal to	`x != y`
<	Less than	`x < y`
>	Greater than	`x > y`
<=	Less than or equal to	`x <= y`
>=	Greater than or equal to	`x >= y`

return statement

statement:

return *expression*(optional)**;**

Semantics:

- **Stop** executing the current function
- Use *expression* as **return value**
- *expression* is
  - omitted for `void` functions
  - required for non-`void`
  - mismatch is compile error

Pattern: Validating Extraction

statements:

```
std::cin >> variable ;
if ( static_cast<bool>(std::cin) == false ) {
 std::cout << "command-error";
 return 1;
}
```

Example:

```
double price{0.0};
std::cout << "Enter price: ";
std::cin >> price;
if (static_cast<bool>(std::cin) == false) {
  std::cout << "error: unrecognized input\n";
  return 1;
}
```

Input/Output:

```
Enter price: nothing
error: unrecognized input
```

Range Errors

- Each piece of input has a **valid range** of values
- Examples:
  - pizza price must be positive
  - pizza radius must be positive

Range validation algorithm:

1. *Read value*
2. if *value is **outside** valid range*:
   a. *Print command error*
   b. *Communicate error exit code to operating system*

Pattern: Validating Input Range

statements:

std::cin >> *variable*;
if (*variable has invalid value*) {
 std::cout << “*command-error*”;
 return 1;
}

Example:

```
double price{0.0};
std::cout << "Enter price: ";
std::cin >> price;
if (static_cast<bool>(std::cin) == false) {
  std::cout << "error: unrecognized input\n";
  return 1;
}
if (price <= 0.0) {
  std::cout << "error: price must be positive\n";
  return 1;
}
```

2. Arithmetic Operators

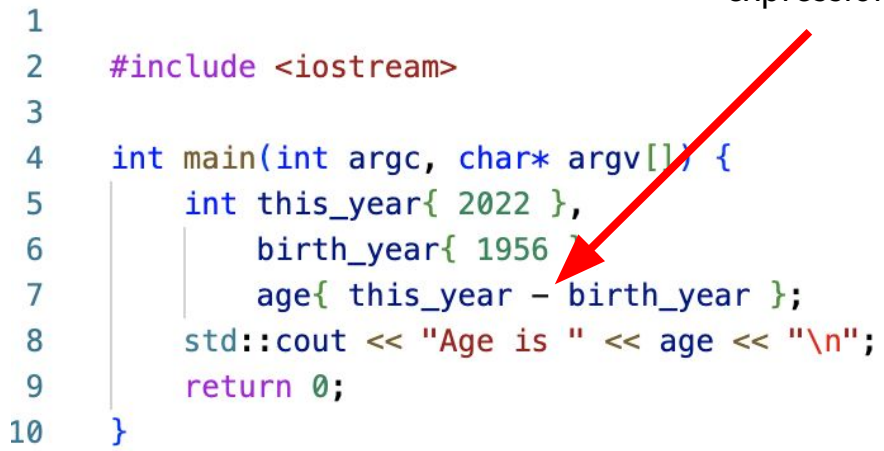
Syntax: Binary Operator Expression

expression:

left-expression operator right-expression

Semantics:

- Evaluate *left-expression* and *right-expression*
- Apply *operator* to produce a value

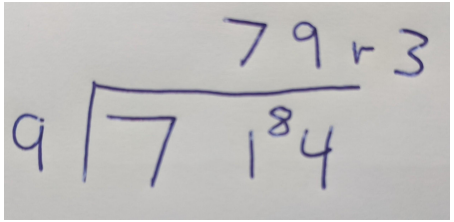


Binary Arithmetic Operators

Operator	Semantics	Example
`+`	add	`x + 3`
`-`	subtract	`i - 1`
`*`	multiply	`price * 1.1`
`/`	divide	`total / 2`
`%`	modulus (remainder)	`total % 10`

Integer Division

- arithmetic is **closed**:
  - operating on two `int`s always produces an `int`
  - operating on two `double`s always produces a `double`



- What about `int` division?
- If left-expression and right-expression are both integers:

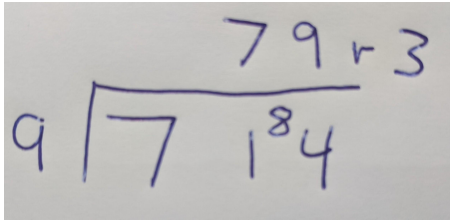
left-expression **/** *right-expression*

produces the **quotient** of *left-expression* divided by *right-expression*

- Equivalent: divide normally, then round down any fraction
- 714 / 9 produces 79

Modulus %

- *Modulus*: remainder of long division (“mod”)
- Example:
  714 % 9 produces 3
- Only available for integer types
  - `double` gives compile error
- Later: surprisingly, modulo is useful!



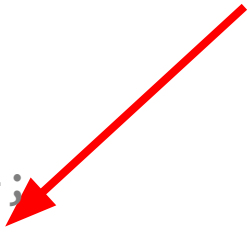
Syntax: Unary Operator Expression

expression:

operator right-expression

Example:

```
double temp{ 98.6 };
double neg_temp{ -temp };
```



Semantics:

- Evaluate *right-expression*
- Apply *operator* to produce a value

Unary Arithmetic Operator

Operator	Semantics	Example
-	negate (multiply by -1)	`-x`

Compound Expressions

- **Compound**: made of multiple parts
- **Compound expression**: expression with multiple operators
- Binary operator expression:

left-expression operator right-expression

- Can fill-in-the-blank *left-expression* with **another** binary operator expression
  - Same goes for *right-expression*
- Example:

```
3 * x + 4
```

Operator Precedence

- Recall: algorithm is **clear** about what to do
- Compound expressions could be unclear
- Does `3 * x + 4` mean...
- `(3 * x) + 4`, or
- `3 * (x + 4)`?
- **Operator precedence**: rule for which operator precedes the other
- Similar to math order of operations

Operator Precedence

Based on **PEMDAS**:

1. **P**arenthesis
2. ~~**E**xponents~~ (C++ does not have)
3. **M**ultiply
4. **D**ivide
5. **A**dd
6. **S**ubtract

Ties: **left-to-right** order

Full list: C++ Operator Precedence (cppreference.com)

Parenthesis

- Parenthesis are allowed in expressions
- Need to be balanced: every (has a matching)
- Example:

```
std::cout << ((x + 1) * 2) / 5;
```

- **Pitfall**: imbalanced parenthesis
  - Compile error
  - Count open paren (
  - Count close paren)
  - Correct the mismatch

Best Practice: Use Parenthesis Liberally

- Difficult for people to memorize the entire precedence table
- Typing ( ) is fast
- Becoming confused is a time sink
- Best practice: **add parenthesis to a compound expression to make order-of-evaluation crystal clear**, even if the parenthesis are technically unnecessary

Worse	Better
`3 * x + 4`	`(3 * x) + 4`

3. Assignment Operators

Code Variables Can Change

- In math, variables do not change, so
  x = 5
  x = 6
  are an invalid contradiction

- In programming, **variables can change over time**
- So this is fine:
  int x { 5 };
  x = 6; // change x to store 6

Side Effects

- Some expressions have side effects
- **Side effect:** a change that is a consequence of evaluating an expression

Kind of Operator	Side Effect	Example
Arithmetic	None	`this_year - birth_year`
stream insertion <<	Print value	`std::cout << year`
stream extraction >>	Store input in variable	`std::cin >> year`
Assignment (next slide)	Store expression in variable	`age = this_year - birth_year`

Assignment Operator

expression:

left **=** *expr*

Semantics:

- *left* must be a variable (or other *lvalue*)
- Evaluate *expr* to produce an object
- Side effect: *left* now stores the new object

left is **changed**
(unlike = in math)

Examples:

```
int score{ 0 };
std::cout << score << “\n”; // prints 0
score = 5;
std::cout << score << “\n”; // prints 5
score = -1;
std::cout << score << “\n”; //prints -1
```

Pitfall: Backwards Assignment

- Pattern:

  left **=** *expr*

- **Changes *left*** to become *expr*
- Pitfalls:
  - Expression on left side
  - Destination on left side

```
int a{ 3 }, b { 9 };
4 = a;  // compile error

// intend to change a to hold b's value
b = a; // backwards, should be a = b;
```

Review: Expression Statement

statement:

expr `;`

Example:

```
std::cout << “Hi” << “ there”;
```

Semantics:

- Evaluate *expr*
- Any object produced by *expr* is discarded
- (That’s all)

Pitfall: Ineffectual Expression Statement

- **Ineffectual**: has no effect
- Recall: object produced in an expression statement is discarded
- An expression statement with no side effects is ineffectual
  - Accomplishes nothing
  - Programmer may be confused
  - Delete the statement

Example:

```
int score { 0 };
score + 1;  // ineffectual
std::cout << score << “\n”;  //prints 0
```

Programmer intended:

```
int score { 0 };
score = score + 1;
std::cout << score << “\n”;  //prints 1
```

Arithmetic Assignment Operators

- Pattern: assign a variable to a new version of itself
- **Arithmetic assignment operator:** combination of = and an arithmetic operator
- Syntax:

*left op***=** *expr*

- Semantics: equivalent to

left = left op expr

Arithmetic Assignment	Equivalent To
`count += 1;`	`count = count + 1;`
`radius *= s;`	`radius = radius * s;`
`width /= 2;`	`width = width / 2;`
`roll %= sides;`	`roll = roll % sides;`

Pre-Increment And Pre-Decrement

- **Increment**: increase by one
- **Decrement**: decrease by one
- Common operation (ex. counting things)

Operator	Semantics	Example
`++`*var*	increment *var*	`++count;`
`--`*var*	decrement *var*	`--lives;`

Post-Increment Operators

- **Post-increment**: increments *var* **after producing the original value**
- **Post-decrement**: decrements *var* **after producing the original value**
- write operator **after** *var*

Operator	Semantics	Example
var`++`	produce current value of *var*, and then increment *var*	`count++;`
var`--`	produce current value of *var*, and then decrement *var*	`lives--;`

- Easter egg: C++ “one-ups” C

Example: Increment

```
#include <iostream>
int main(int argc, char* argv[]) {

  int a{ 5 };
  std::cout << "a is " << a << “\n”;
  a++;
  std::cout << "a is " << a << “\n”;
  std::cout << "a is " << ++a << “\n”;
  std::cout << "a is " << a << “\n”;
  std::cout << "a is " << a++ << “\n”;
  std::cout << "a is " << a << “\n”;

  return 0;
}
```

```
$ ./a.out
a is 5
a is 6
a is 7
a is 7
a is 7
a is 8
```

Comma Operator

expression:

left , *right*

Semantics:

1. Evaluate *left* and discard the result
2. Evaluate *right* and produce that value

Issues

- Confusing
- Almost entirely pointless

Example:

```
int a{ 5 }, b{ 1 }, c{ 0 };
c = a + 1, b + 1;  // discards a + 1
std::cout << c << “\n”; // prints 2
```

Why does the comma operator exist?

- Misguided attempt to make increment statements more concise

```
++i, ++j;
```

- Confusing; readability more important
- Style guide says to just write two separate statements

```
++i;
++j;
```

- **Never use the comma operator**