Operators

Class 5

Section 2.11

- we will not cover section 2.11
- you will not be tested on it
- feel free to read it on your own

Data Types

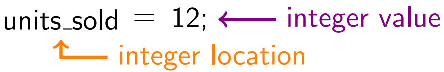
Data Type

A data type is a set of values and a set of operations defined on those values.

- in class 4 we talked about the values of various data types
- in this class we will focus on some of the operations

Assignment

- perhaps the most important operator of all is assignment $=$
- assignment is a binary operator
- it has a left-hand side (lhs) and a right-hand side (rhs)
- the lhs must resolve to an lvalue: an address where a value can be stored
- the rhs must resolve to an rvalue: a value that can be stored in the lhs address



units_sold = 12;

integer value

integer location

- the value 12 is copied into the memory storage location denoted by label (variable) `units_sold`

An Illegal Assignment

```
12 = units_sold;
```

- 12 is not an lvalue; it is not a storage location (it is a literal)

Initialization

- assignment can be combined with variable declaration
- this is called initialization

```
unsigned units_sold = 12;
```

- this is much more concise than declaration-then-assignment

```
unsigned units_sold;
units_sold = 12;
```

- when you can initialize, you should
- Gaddis is sloppy about this

Scope

Scope

A variable's scope is the region of the program in which the variable exists and in which its name can be legally used.

- a variable's scope extends from declaration to the end of the closest containing block

```
int main()
{
 int value = 100;← start of scope
 cout << value << endl;
 return 0;
}← end of scope
```

Scope

- you cannot reference a variable outside of its scope

```
int main()
{
  cout << value << endl;

  int value = 100;
  return 0;
}
```

illegal: out of scope

scope

Re-Declaring a Variable

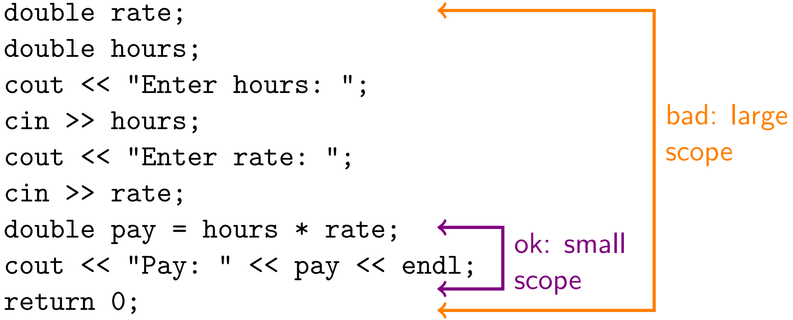
- the scope concept means that it is illegal to re-declare a variable in the same scope

```
int main()
{
 int value = 100; ←—— declaration
 cout << value << endl;
 int value = 200; ←—— illegal: redeclared in same scope
 return 0;
}
```

- you can re-assign a variable repeatedly, but not re-declare it

Declaration Location

- best practice in coding dictates that a variable's scope should be as small as possible
- this means declaring a variable close to the place where it is first used
- the style guide also has this rule



```
double rate;
double hours;
cout << "Enter hours: ";
cin >> hours;
cout << "Enter rate: ";
cin >> rate;
double pay = hours * rate;
cout << "Pay: " << pay << endl;
return 0;
```

bad: large scope

ok: small scope

Unary Minus

- C++ has one unary arithmetic operator, the minus sign
- it can be used with signed integer and with floating point types

```
int x = 5;
int y = -x;
double a = -12.34;
```

Binary Operators

- C++ has several binary arithmetic operators
- can be used with all integer and floating point types
- in chapter 2, we assume both lhs and rhs of these operators are of exactly the same type
- in chapter 3 we will look at mixing types

Operator	**Purpose**
+	addition
-	subtraction
*	multiplication
/	division
%	modulus (integer only)

Division

- all the operators are straightforward except division
- if both operands are floating point types, the result is similar to the results on a calculator

```
double x = 10.0 / 4.0;
```

← x becomes 2.5

- if both operands are integer types, the result is an integer quotient with no fractional part or remainder

```
int x = 10 / 4;
```

← x becomes 2

Division

- all the operators are straightforward except division
- if both operands are floating point types, the result is similar to the results on a calculator

```
double x = 10.0 / 4.0;
```

← x becomes 2.5

- if both operands are integer types, the result is an integer quotient with no fractional part or remainder

```
int x = 10 / 4;
```

← x becomes 2

- remember, in this chapter, we do not allow mixing types

```
int x = 10.0 / 4;
```

← mixed types not allowed

Modulus

- applies only to integer types
- not defined for floating point types (Python allows this, which is just weird)
- used to give the remainder after division completes

$$5 \div 2 = \underset{\substack{\nearrow \\ \text{quotient}}}{2} \text{ r } \underset{\substack{\nwarrow \\ \text{remainder}}}{1}$$

Modulus

- when using modulus, the dividend can be positive, zero, or negative
- the divisor should always be positive (just like in elementary school)
- a negative divisor is legal in C++, but mathematically very controversial, so don't do it

Modulus

- an extremely useful operator
- two big uses:
  1. determine if a number is even or odd
  2. (combined with division) determine specific digits in a base-10 number

Even or Odd

- a number is even if its remainder when divided by 2 is 0
- 156 % 2 is 0, so 156 is an even number

- a number is odd if its remainder when divided by 2 is 1
- 157 % 2 is 1, so 157 is an odd number

Digits in a Number

- imagine I have: `unsigned x = 12345;`
- I need to extract the hundreds digit (in this case 3) from the value
- `unsigned hundreds_digit = (x / 100) % 10;`

Digits in a Number

- imagine I have: `unsigned x = 12345;`
- I need to extract the hundreds digit (in this case 3) from the value
- `unsigned hundreds_digit = (x / 100) % 10;`
- how would you get the thousands digit?

Digits in a Number

- imagine I have: `unsigned x = 12345;`
- I need to extract the hundreds digit (in this case 3) from the value
- `unsigned hundreds_digit = (x / 100) % 10;`
- how would you get the thousands digit?
- an example use: convert 255 minutes into minutes and hours

Digits in a Number

- imagine I have: `unsigned x = 12345;`
- I need to extract the hundreds digit (in this case 3) from the value
- `unsigned hundreds_digit = (x / 100) % 10;`
- how would you get the thousands digit?

- an example use: convert 255 minutes into minutes and hours

```
unsigned total_minutes = ...;
unsigned hours = total_minutes / 60;
unsigned minutes = total_minutes % 60;

unsigned check = hours * 60 + minutes;
```

Parentheses

- just like in algebra, we use parentheses to override precedence
- `x = 1 + 2 * 3;` multiplication has higher precedence than addition, so this yields 7
- `x = (1 + 2) * 3;` parentheses override precedence, so this yields 9

Parentheses

- just like in algebra, we use parentheses to override precedence
- `x = 1 + 2 * 3;` multiplication has higher precedence than addition, so this yields 7
- `x = (1 + 2) * 3;` parentheses override precedence, so this yields 9

- however, only use parentheses either 1) when they are necessary or 2) when they improve readability
- `x = 1 * (2 * 3);` bad: irrelevant because multiplication is commutative
- `x = (1 + 2 + 3);` bad: unnecessary, and thus confusing
- `x = (1 + 2) / (3 + 4);` good: mathematically necessary
- `x = (2 * a) + (b * 4 / c);` good: not mathematically necessary, but improves readability

Comments

- there are three types of comments
  1. here-to-end-of-line comments, denoted by double-slashes //
  2. multiline comments delimited by /* and */
  3. Javadoc comments beginning with /** and ending with */
- here-to-end comments are limited to a single editor line
- multiline comments can span multiple lines
- Javadoc comments are used specifically to explain functions (explained later in the semester)

- comment symbols that appear inside of strings are not comments

  ```
  cout << "Hello // world" << endl;
  ```

Comments

- what kind and how many comments to write is largely a matter of style
- in this class, we will recognize three kinds of comments:
  1. a header comment at the beginning of a file, telling the purpose of the file and the name of the author
  2. a header comment at the beginning of a function (Javadoc, explained later in the course)
  3. in-code comments that explain the purpose of this "paragraph" of code

Comments

- what kind and how many comments to write is largely a matter of style
- in this class, we will recognize three kinds of comments:
  1. a header comment at the beginning of a file, telling the purpose of the file and the name of the author
  2. a header comment at the beginning of a function (Javadoc, explained later in the course)
  3. in-code comments that explain the purpose of this “paragraph” of code
- make sure your comments actually add information

```
//Enter Radius ⟵ does this add information?
cout << "Please enter the radius of a circle:  ";
cin >> radius;
```

Variable Name Comments

- Gaddis discusses a fourth kind
- a comment that documents the purpose of a variable

  ```
  double pay_rate; // holds the hourly pay rate
  ```

Variable Name Comments

- Gaddis discusses a fourth kind
- a comment that documents the purpose of a variable
  ```
  double pay_rate; // holds the hourly pay rate
  ```
- this is very definitely not allowed in this course
- if you have to explain the name, you chose a poor name
- you should create self-documenting variable names
  ```
  double hourly_pay_rate;
  ```
- that way, you won't have to look back at the declaration to see if the variable is holding an hourly or weekly pay rate
- just looking at the variable itself tells you that it's an hourly pay rate

Named Constants

- consider the following statement in a finance program dealing with loans

  ```
  amount = balance * 0.069;
  ```

- there are two potential problems with this code
  1. 0.069 is an anonymous amount; there is no hint why it is in the program. Is this interest? A fee? A tax rate? A one-time adjustment? The reader doesn't know.

Named Constants

- consider the following statement in a finance program dealing with loans

  ```
  amount = balance * 0.069;
  ```
- there are two potential problems with this code
  1. 0.069 is an anonymous amount; there is no hint why it is in the program. Is this interest? A fee? A tax rate? A one-time adjustment? The reader doesn't know.
  2. Think of what happens if this multiplier changes in the future. If this is an interest rate, what if the rate changes?

Magic Numbers

- in programming, an anonymous value is called a magic number
- it is a literal that appears in code with no hint of its purpose
- 0.069 is a magic number
- if instead we wrote:

  ```
  amount = balance * interest_rate;
  ```

  it would be obvious what was going on
- the style guide forbids the use of magic numbers

Repeated Use

- in a banking program, there may be many places where the interest rate is involved in calculations
- if 0.069 is used 37 times over 25 pages of code, two bad things can happen
  1. what if the interest rate changes to, say, 0.066? all 37 occurrences of 0.069 over 25 pages must be found and changed!
  2. even if the rate doesn't change, what if one of the 37 occurrences is mis-typed as 0.068? the chances of noticing it are slim

Repeated Use

- in a banking program, there may be many places where the interest rate is involved in calculations
- if 0.069 is used 37 times over 25 pages of code, two bad things can happen
  1. what if the interest rate changes to, say, 0.066? all 37 occurrences of 0.069 over 25 pages must be found and changed!
  2. even if the rate doesn't change, what if one of the 37 occurrences is mis-typed as 0.068? the chances of noticing it are slim
- to avoid both anonymous magic numbers and repeated use of a literal, we use named constants

Named Constants

```
const double INTEREST_RATE = 0.069;
...
blah = blah blah * INTEREST_RATE;
...
yada yada yada INTEREST_RATE yada yada;
```

- if the interest rate changes, you need only change the code in one place
- interest rate is always identical; no chance of a typo
- since the interest rate never changes in one run of the program, it is a constant
- like a variable, but cannot be re-assigned
- style calls for constant identifiers to be in ALL_UPPER_CASE