# Programming in Objective-C

**Data Types and Expressions**

**Program Looping**

+
## Data Types and Expressions

- You have already encountered the Objective-C basic data type int.
- As you will recall, a variable declared to be of type int can be used to contain integral values only—that is, values that do not contain decimal digits.
- The Objective-C programming language provides three other basic data types: float, double, and char.
- A variable declared to be of type float can be used for storing floating-point numbers (values containing decimal digits).

## Data Types and Expressions

- The double type is the same as type float, typically with roughly twice the range.
- The char data type can be used to store a single character, such as the letter a, the digit character 6, or a semicolon
- In Objective-C, any literal number, single character, or character string is known as a *constant.* For example, the number 58 represents a constant integer value.
- The string @"Programming in Objective-C is fun." is an example of a constant character string object.”

## Type int

- An integer constant consists of a sequence of one or more digits. A minus sign preceding the sequence indicates that the value is negative.
- The values 158, –10, and 0 are all valid examples of integer constants. No embedded spaces are permitted between the digits, and commas can't be used.
- So, the value 12,000 is not a valid integer constant and must be written as 12000.
- Every value, whether it's a character, an integer, or a floating-point number, has a *range* of values associated with it.

## Type int

- This range has to do with the amount of storage allocated to store a particular type of data.
- In general, that amount is not defined in the language; it typically depends on the computer you're running on and is therefore called *implementation* or *machine dependant.*
- For example, an integer variable might take 32 bits on your computer, or perhaps it might be stored in 64.
- If 64 bits were used, much larger numbers can be stored inside integer variables than if 32 bits were used instead.

+ 
## Type float

- You can use a variable declared to be of type float to store values containing decimal digits.
- A floating-point constant is distinguished by the presence of a decimal point.
- The values 3., 125.8, and -.0001 are all valid examples of floating-point constants.
- To display a floating- point value, the NSLog conversion characters %f or %g can be used.
- Floating-point constants can also be expressed in so-called *scientific notation.*
- As noted, the double type is the same as type float, only with roughly twice the range.

+

## Type char

- You can use a char variable to store a single character.
- A character constant is formed by enclosing the character within a pair of single quotation marks. So 'a', ';', and '0' are all valid examples of character constants.
- The first constant represents the letter a, the second is a semicolon, and the third is the character zero—which is not the same as the number zero.
- Do not confuse a character constant, which is a single character enclosed in single quotes, with a C-style character string, which is any number of characters enclosed in double quotes.
- The format characters %c can be used in an NSLog call to display the value of a char variable.

## Our First Programming Example

```
#import <Foundation/Foundation.h>

int main (int argc, char * argv[])
{
   @autoreleasepool {
      int    integerVar = 100;
      float  floatingVar = 331.79;
      double doubleVar = 8.44e+11;
      char   charVar = 'W';

     NSLog (@"integerVar = %i", integerVar);
     NSLog (@"floatingVar = %f", floatingVar);
     NSLog (@"doubleVar = %e", doubleVar);
     NSLog (@"doubleVar = %g", doubleVar);
     NSLog (@"charVar = %c", charVar);
   }
   return 0;
}
```

+ 
## Explanation

- In the second line of the program's output, notice that the value of 331.79, which is assigned to floatingVar, is actually displayed as 331.790009.
- The reason for this inaccuracy is the particular way in which numbers are internally represented inside the computer.
- You have probably come across the same type of inaccuracy when dealing with numbers on your calculator.
- If you divide 1 by 3 on your calculator, you get the result . 33333333, with perhaps some additional 3s tacked on at the end.

## Explanation Continued

- The string of 3s is the calculator’s approximation to one third. Theoretically, there should be an infinite number of 3s.
- But the calculator can hold only so many digits, thus the inherent inaccuracy of the machine.
- The same type of inaccuracy applies here: Certain floating-point values cannot be exactly represented inside the computer’s memory.

## Qualifiers: long, long long, short, unsigned, and signed

- If the qualifier long is placed directly before the int declaration, the declared integer variable is of extended range on some computer systems.
- An example of a long int declaration might be this:
- **long int factorial;**
- This declares the variable factorial to be a long integer variable.
- As with floats and doubles, the particular range of a long variable depends on your particular computer system.

## Qualifiers: long, long long, short, unsigned, and signed

- To display the value of a long int using NSLog, the letter *l* is used as a modifier before the integer format characters.
- This means that the format characters %li can be used to display the value of a long int in decimal format.
- You can also have a long long int variable, or even a long double variable to hold a floating point number with greater range.
- The qualifier short, when placed in front of the int declaration, tells the Objective-C compiler that the particular variable being declared is used to store fairly small integer values.

## Qualifiers: long, long long, short, unsigned, and signed

- The motivation for using short variables is primarily one of conserving memory space, which can be an issue when the program needs a lot of memory and the amount of available memory is limited.
- The final qualifier that can be placed in front of an int variable is used when an integer variable will be used to store only positive numbers.
- The following declares to the compiler that the variable counter is used to contain only positive values:
- unsigned int counter;

+

## Type id

- The id data type is used to store an object of any type. In a sense, it is a generic object type.
- For example, this line declares graphicObject to be a variable of type id:
- **id graphicObject;**
- Methods can be declared to return values of type id, like so:
- **-(id) newObject: (int) type;**
- This declares an instance method called newObject that takes a single integer argument called type and returns a value of type id.

+
## Type id

- The id data type is an important data type used often in this course.
- The id type is the basis for very important features in Objective-C known as *polymorphism* and *dynamic binding*

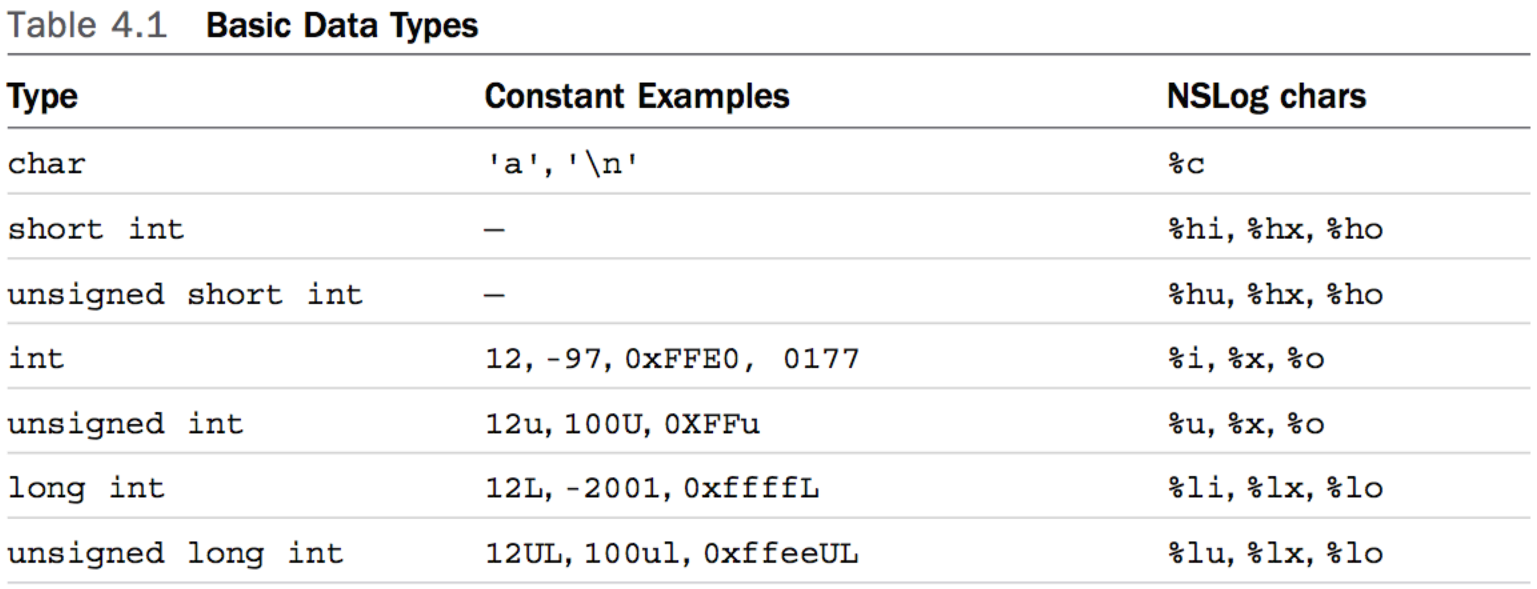


Table 4.1 **Basic Data Types**

| Type | Constant Examples | NSLog chars |
|---|---|---|
| `char` | `'a', '\n'` | `%c` |
| `short int` | — | `%hi, %hx, %ho` |
| `unsigned short int` | — | `%hu, %hx, %ho` |
| `int` | `12, -97, 0xFFE0, 0177` | `%i, %x, %o` |
| `unsigned int` | `12u, 100U, 0XFFu` | `%u, %x, %o` |
| `long int` | `12L, -2001, 0xffffL` | `%li, %lx, %lo` |
| `unsigned long int` | `12UL, 100ul, 0xffeeUL` | `%lu, %lx, %lo` |

+
# Basic Data-types Continued

| Type | Constant Examples | NSLog chars |
|---|---|---|
| long long int | 0xe5e5e5e5LL, 500ll | %lli, %llx, &llo |
| unsigned long long int | 12ull, 0xffeeULL | %llu, %llx, %llo |
| float | 12.34f, 3.1e-5f, 0x1.5p10, 0x1P-1 | %f, %e, %g, %a |
| double | 12.34, 3.1e-5, 0x.1p3 | %f, %e, %g, %a |
| long double | 12.34L, 3.1e-5l | %Lf, $Le, %Lg |
| id | nil | %p |

+

## Arithmetic Expressions

- In Objective-C, just as in nearly all programming languages, the plus sign (+) is used to add two values, the minus sign (-) is used to subtract two values, the asterisk (*) is used to multiply two values, and the slash (/) is used to divide two values.

- These operators are known as binary arithmetic operators because they operate on two values or terms.

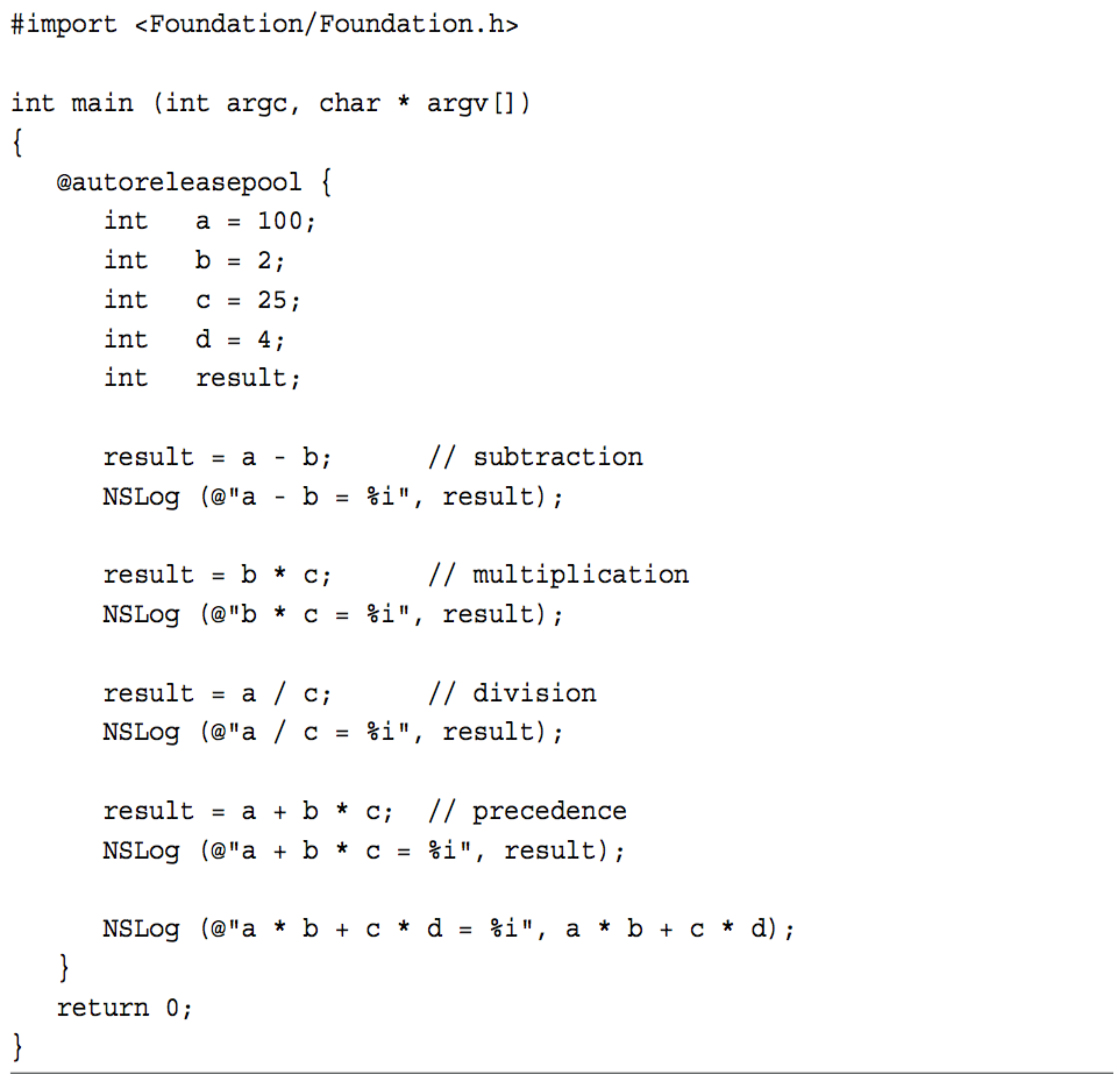
+
## Operator Precedence

- You have seen how a simple operation such as addition can be performed in Objective-C.
- The following program further illustrates the operations of subtraction, multiplication, and division.
- The last two operations performed in the program introduce the notion that one operator can have a higher priority, or precedence, over another operator.
- In fact, each operator in Objective-C has a precedence associated with it.

## Operator Precedence

- This precedence is used to determine how an expression that has more than one operator is evaluated: The operator with the higher precedence is evaluated first.
- Expressions containing operators of the same precedence are evaluated either from left to right or from right to left, depending on the operator.
- This is known as the *associative* property of an operator.
- We'll program a second example of this on the next slide.

+ ## Operator Precedence



```
#import <Foundation/Foundation.h>

int main (int argc, char * argv[])
{
   @autoreleasepool {
      int   a = 100;
      int   b = 2;
      int   c = 25;
      int   d = 4;
      int   result;

      result = a - b;      // subtraction
      NSLog (@"a - b = %i", result);

      result = b * c;      // multiplication
      NSLog (@"b * c = %i", result);

      result = a / c;      // division
      NSLog (@"a / c = %i", result);

      result = a + b * c;  // precedence
      NSLog (@"a + b * c = %i", result);

      NSLog (@"a * b + c * d = %i", a * b + c * d);
   }
   return 0;
}
```

+
## Operator Precedence

- After declaring the integer variables a, b, c, d, and result, the program assigns the result of subtracting b from a to result and then displays its value with an appropriate NSLog call.
- The next statement has the effect of multiplying the value of b by the value of c and storing the product in result:
- result = b * c;
- The result of the multiplication is then displayed using an NSLog call that should be familiar to you by now.
- The next program statement introduces the division operator, the slash. The NSLog statement displays the result of 4, obtained by dividing 100 by 25, immediately following the division of a by c.

+

# Operator Precedence

- Attempting to divide an integer by zero results in abnormal termination or an exception when the division is attempted.
- Even if the program does not terminate abnormally, the results obtained by such a division will be meaningless
- If the divisor is determined to be zero, an appropriate action can be taken and the division operation can be averted.
- This expression does not produce the result of 2550 (102 × 25);
- instead, the result displayed by the corresponding NSLog statement is shown as 150:
- **a+b* c**

+
# Operator Precedence

- This is because Objective-C, like most other programming languages, has rules for the order of evaluating multiple operations or terms in an expression.
- Evaluation of an expression generally proceeds from left to right. However, the operations of multiplication and division are given precedence over the operations of addition and subtraction. Therefore, the system evaluates the expression
- **a+b* c**
- as follows:
- **a + (b * c)**

+
## Operator Precedence

- If you want to alter the order of evaluation of terms inside an expression, you can use parentheses.
- In fact, the expression listed previously is a perfectly valid Objective-C expression.
- Thus, the following statement could have been substituted in Program 4.2 to achieve identical results:
- result = a + (b * c);
- However, if this expression were used instead, the value assigned to result would be 2550: result = (a + b) * c;

## Operator Precedence

- This is because the value of a (100) would be added to the value of b (2) before multiplication by the value of c (25) would take place.
- Parentheses can also be nested, in which case evaluation of the expression proceeds outward from the innermost set of parentheses.
- Just be sure to have as many closed parentheses as you have open ones.

## Operator Precedence

- Notice from the last statement in Program 4.2 that it is perfectly valid to give an expression as an argument to NSLog without having to first assign the result of the expression evaluation to a variable.
- The expression
- a*b+c* d
- is evaluated according to the rules stated previously as
- (a * b) + (c * d)
- or
- (100 * 2) + (25 * 4)
  The result of 300 is handed to the NSLog routine.

+

## Integer Arithmetic and the Unary Minus Operator

```
// More arithmetic expressions

#import <Foundation/Foundation.h>

int main (int argc, char * argv[])
{
   @autoreleasepool {
      int   a = 25;
      int   b = 2;
      float c = 25.0;
      float d = 2.0;

      NSLog (@"6 + a / 5 * b = %i", 6 + a / 5 * b);
      NSLog (@"a / b * b = %i", a / b * b);
      NSLog (@"c / d * d = %f", c / d * d);
      NSLog (@"-a = %i", -a);
   }
   return 0;
}
```

# The Modulus Operator

- The last arithmetic operator to be presented in this chapter is the modulus operator, which is symbolized by the percent sign (%).

```
#import <Foundation/Foundation.h>

int main (int argc, char * argv[])
{
   @autoreleasepool {
      int a = 25, b = 5, c = 10, d = 7;

      NSLog (@"a %% b = %i", a % b);
      NSLog (@"a %% c = %i", a % c);
      NSLog (@"a %% d = %i", a % d);
      NSLog (@"a / d * d + a %% d = %i", a / d * d + a % d);
   }
   return 0;
}
```

+
## Integer and Floating-Point Conversions

- To effectively develop Objective-C programs, you must understand the rules used for the implicit conversion of floating-point and integer values in Objective-C.

```
// Basic conversions in Objective-C

#import <Foundation/Foundation.h>

int main (int argc, char * argv[])
{
   @autoreleasepool {
      float  f1 = 123.125, f2;
      int    i1, i2 = -150;

      i1 = f1;   // floating to integer conversion
```

+

## Integer and Floating-Point Conversions

```
        NSLog (@"%f assigned to an int produces %i", f1, i1);

        f1 = i2;    // integer to floating conversion
        NSLog (@"%i assigned to a float produces %f", i2, f1);

        f1 = i2 / 100;    // integer divided by integer
        NSLog (@"%i divided by 100 produces %f", i2, f1);

        f2 = i2 / 100.0;    // integer divided by a float
        NSLog (@"%i divided by 100.0 produces %f", i2, f2);

        f2 = (float) i2 / 100;    // type cast operator
        NSLog (@"(float) %i divided by 100 produces %f", i2, f2);
    }
    return 0;
}
```

## Integer and Floating-Point Conversions

- Whenever a floating-point value is assigned to an integer variable in Objective-C, the decimal portion of the number gets truncated.
- So, when the value of f1 is assigned to i1 in the previous program, the number 123.125 is *truncated,* which means that only its integer portion, or 123, is stored in i1. The first line of the program's output verifies that this is the case.
- Assigning an integer variable to a floating variable does not cause any change in the value of the number; the system simply converts the value and stores it in the floating variable. The second line of the program's output verifies that the value of i2 (–150) was correctly converted and stored in the float variable f1.

+
## Calculator Class

```
#import <Foundation/Foundation.h>

@interface Calculator: NSObject

// accumulator methods
-(void)   setAccumulator: (double) value;
-(void)   clear;
-(double) accumulator;

// arithmetic methods
-(void) add: (double) value;
-(void) subtract: (double) value;
-(void) multiply: (double) value;
-(void) divide: (double) value;
@end

@implementation Calculator
{
    double accumulator;
}

-(void) setAccumulator: (double) value
{
    accumulator = value;
}
```

## Calculator Class Continued

```
-(void) clear
{
    accumulator = 0;
}

-(double) accumulator
{
    return accumulator;
}

-(void) add: (double) value
{
    accumulator += value;
}

-(void) subtract: (double) value
{
    accumulator -= value;
}
```

+

# Calculator Class Continued

```
-(void) multiply: (double) value
{
    accumulator *= value;
}

-(void) divide: (double) value
{
    accumulator /= value;
}
@end

int main (int argc, char * argv[])
{
   @autoreleasepool {
      Calculator *deskCalc  = [[Calculator alloc] init];

      [deskCalc setAccumulator: 100.0];
      [deskCalc add: 200.];
      [deskCalc divide: 15.0];
      [deskCalc subtract: 10.0];
      [deskCalc multiply: 5];
      NSLog (@"The result is %g", [deskCalc accumulator]);
   }
   return 0;
}
```

+ 
## Program Looping

- In Objective-C, you can repeatedly execute a sequence of code in several ways. These looping capabilities are the subject of this chapter, and they consist of the following:
- The for statement
- The while statement
- The do statement
- We start with a simple example: counting numbers.

+
## Program Looping

- The first row of the triangle contains one marble, the second row contains two marbles, and so on.
- In general, the number of marbles required to form a triangle containing *n* rows would be the sum of the integers from 1 through *n.*
- This sum is known as a *triangular number.*
- If you started at 1, the fourth triangular number would be the sum of the consecutive integers 1 through 4 (1 + 2 + 3 + 4), or 10.

## Program Looping

- Suppose you wanted to write a program that calculated and displayed the value of the eighth triangular number.
- Obviously, you could easily calculate this number in your head, but for the sake of argument, let’s assume you wanted to write a program in Objective-C to perform this task.
- Seen on the next slide.

+ 
## Program Looping

```
#import <Foundation/Foundation.h>

// Program to calculate the eighth triangular number

int main (int argc, char * argv[])
{
   @autoreleasepool {
      int triangularNumber;

      triangularNumber = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8;

      NSLog (@"The eighth triangular number is %i", triangularNumber);
   }

   return 0;
}
```

+
## The for Statement

- Let’s take a look at a program that uses the for statement. The purpose of Program 5.2 is to calculate the 200th triangular number.

```
#import <Foundation/Foundation.h>

int main (int argc, char * argv[])
{
   @autoreleasepool {
      int n, triangularNumber;

      triangularNumber = 0;

      for ( n = 1; n <= 200; n = n + 1 )
         triangularNumber += n;

      NSLog (@"The 200th triangular number is %i", triangularNumber);
   }

   return 0;
}
```

+ Exercises

Write a program that converts 27° from degrees Fahrenheit (F) to degrees Celsius (C) using the following formula:

```
C = (F - 32) / 1.8
```

Note that you don't need to define a class to perform this calculation. Simply evaluating the expression will suffice.

What output would you expect from the following program?

```
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
   @autoreleasepool {
      char c, d;

      c = 'd';
      d = c;
      NSLog (@"d = %c", d);
   }
   return 0;
}
```

+ 
## Exercises

Suppose you are developing a library of routines to manipulate graphical objects. Start by defining a new class called `Rectangle`. For now, just keep track of the rectangle's width and height. Develop methods to set the rectangle's width and height, retrieve these values, and calculate the rectangle's area and perimeter. Assume that these rectangle objects describe rectangles on an integral grid, such as a computer screen. In that case, assume that the width and height of the rectangle are integer values.

Here is the `@interface` section for the `Rectangle` class:

```
@interface Rectangle: NSObject
-(void) setWidth: (int) w;
-(void) setHeight: (int) h;
-(int) width;
-(int) height;
-(int) area;
-(int) perimeter;
@end
```

Write the implementation section and a test program to test your new class and methods.