BLG 102E
Introduction to Scientific Computing and Engineering

SPRING 2025

WEEK 2

ISTANBUL TECHNICAL UNIVERSITY

Data Types

- every piece of data has a type
  - integer, real, character, string, boolean, …
- which values it can take
- which operations will be allowed on it

Literals

- value directly written into source code: literal
- value determines type

Numeric Literals

- digits, optional sign: integer
- digits, point, optional sign: floating point

value	**type**
`42`	integer
`-6`	integer
`3.14159`	floating point
`-1.5`	floating point

Text Literals

value	type
`"Hello, world!"`	string
`'t'`	character
`"t"`	string

- single quotes: character
- double quotes: string

value	type
`7`	integer
`'7'`	character
`"7"`	string

value	type
`""`	string
`''`	WRONG

Expression

- expression: a computation
  - gets evaluated
  - gives a result

- type of result determines type of expression
  - arithmetic, boolean, …

Arithmetic Operators

- addition:

```
5.8 + 3.12
```

- subtraction:

```
154.9 - 12.1
```

- multiplication:

```
6.25 * 7.0
```

- division:

```
126.244 / 7.1
```

- unary minus (sign):

```
-(-4.55)
```

Precedence

- precedence as in mathematics
  - unary minus
  - `*` and `/`
  - `+` and –
- use parentheses to change computation order:

```
10 + 4 * 3

(10 + 4) * 3
```

Rules of operator precedence

Highest

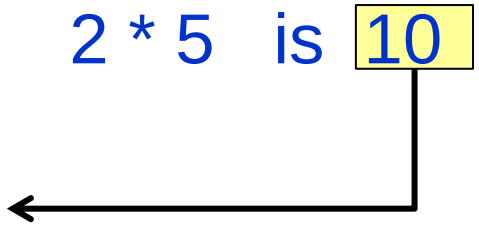
Lowest

Operators	Operations	Order of evaluation (precedence)
()	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
*, /, or %	Multiplication, Division, Modulus	Evaluated second. If there are several, they are evaluated left to right.
+ or -	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

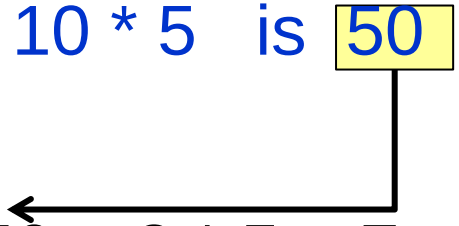
Arithmetic

- Calculate the result of following arithmetic expression.

Step1. y = 2 * 5 * 5 + 3 * 5 + 7; (Leftmost multiplication)

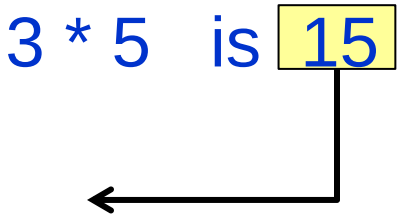


Step2. y = 10 * 5 + 3 * 5 + 7; (Leftmost multiplication)

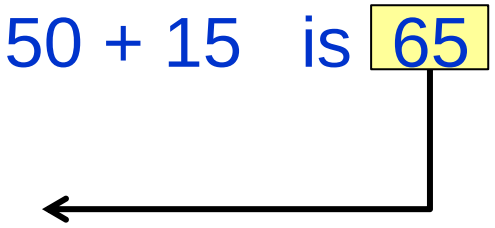


Step3. y = 50 + 3 * 5 + 7; (Multiplication before addition)

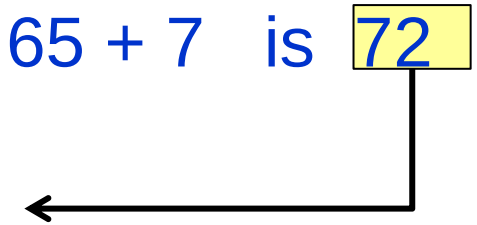
Arithmetic



Step4. y = 50 + 15 + 7; (Leftmost addition)



Step5. y = 65 + 7; (Last addition)



Step6. y = 72; (Assign 72 to y)

Operator Style

- one space each on both sides of operator
- except unary minus

```
5*1.28+3*17.32

// prefer
5 * 1.28 + 3 * 17.32
```

```
- (6 * 7)

// prefer
-(6 * 7)
```

Parenthesis Style

- no space inside parentheses

```
5 * ( 1.25 + 0.03 )

// prefer
5 * (1.25 + 0.03)
```

Integer

- regular:

  `int`

- narrower range:

  `short int`

- wider range:

  `long int`

- value ranges not standard
- unsigned variants:
  - `unsigned int`
  - `unsigned short int`
  - `unsigned long int`

Floating Point

- single precision:

  `float`

- double precision:

  `double`

- higher precision:

  `long double`

- prefer `double`

Variable

- variable:
  - a memory location (address)
  - associated with a name (identifier)
  - containing some information (value)

- different from mathematical variables
  - mathematical variables are abstract

Defining Variables

- defining a variable:
  - name
  - type
  - optionally, initial value (strongly recommended)

- syntax:

```
TYPE NAME = VALUE;
```

- without initial value:

```
TYPE NAME;
```

Variable Definition Examples

```
int weight = 65;
```

- **name:** `weight`
- **type:** `int`
- **initial value:** `65`

```
double bmi;
```

- **name:** `bmi`
- **type:** `double`
- no initial value

Variable Comments

- adding helpful comments is good style

```
int weight = 65;   // mass, in kg
```

Multiple Definitions

- multiple variables can be defined in the same statement
- all of the same type

```
TYPE NAME1 = VALUE1, NAME2 = VALUE2, ...;
```

Multiple Definition Example

```
double height = 1.74, bmi;
```

- prefer defining one variable per statement

```
double height = 1.74;   // in m
double bmi;   // body mass index
```

Using Variables

- variables can be used in expressions

```
x + 4
155 - x
x * y
-x
```

Using Variables

C operation	C arithmetic operator	Algebraic expression example	C expression example
Addition	+	*f* + *7*	f + 7
Subtraction	-	*p* – *c*	p – c
Multiplication	*	*b* . *r*	b * r
Division	/	*x* / *y*	x / y
Modulus	%	*r mod p*	r % p

Initial Values

- initial value is an expression
- previously defined variable can be used

```
int weight = 65;   // mass, in kg
double height = 1.74;   // in m
double bmi = weight / (height * height);
```

Initial Values

- Initialization: assigning an initial numerical value to a variable.
- Variables should normally be initialized to some value before being used in a program.
- An uninitialized variable contains an arbitrary value (the value last stored in the memory location reserved for that variable).

PROGRAM1

```
#include <stdio.h>
int main()
{
  int a; // Declaration without initialization
  printf("%d \n", a);
}
```

Program Output

```
3148880
```

Uninitialized variable.
Its value is arbitrary (random).

PROGRAM2

```c
#include <stdio.h>
int main()
{
  int a;   // Declaration
  a = 50;  // Initialization (assignment)

  printf("%d \n", a);
}
```

Program Output

```
50
```

PROGRAM3

```c
#include <stdio.h>
int main()
{
  // Declaration and initialization in one statement
  int a = 50;

  printf("%d \n", a);
}
```

Program Output

```
50
```

Assignment

- value of variable can be changed
- assignment: store new value in variable
- replaces previous value

```
VARIABLE = EXPRESSION;
```

Assignment Example

- change value of `weight` to `68`

```
weight = 68;
```

Definition and Assignment

- assignment modifies value of previously defined variable
- no type keyword in assignment

```
int weight = 65;   // definition
...
...
weight = 68;       // assignment
```

Assignment and Equality

- assignment is not equality

```
weight = weight - 1.5;
```

- evaluate expresion, store result in variable

variable ← *expression*

Assignment Semantics

```
int weight = 65;
weight = weight - 2;
```

- look up current value of `weight` (`65`)
- calculate `weight - 2` (`63`)
- store result in variable `weight`

Left Hand Side

- left hand side must be a variable

```
68 = weight;   // syntax error
```

Left Hand Side

Syntax:

Variable = Expression

- The + operator has two operands (num1 and num2).

```
sum = num1 + num2;
```

The **sum** variable gets the result of adding operation num1 + num2

- The left of the assignment operator (=) can not be an arithmetic expression.

```
num1 + num2 = sum;   // Compiler error!
```

Variable Names

- using descriptive names is good style
- `weight` is better than `w`

Name Rules

- only letters, digits, underscore
- must not start with digit
- reserved words aren't allowed

Name Examples

name	**validity**
`w`	valid
`weight1`	valid
`1weight`	invalid: starts with digit
`minor?`	invalid: contains question mark
`body mass index`	invalid: contains space
`body-mass-index`	invalid: contains dash
`return`	invalid: reserved word

Name Examples

- **VALID variable names:**

  OgrenciNum
  OgrNum
  Ogr_Num
  Ogr4,
  Sum , alfa , teta , aSquare , Pi

- **INVALID variable names:**

  ÖğrenciNum : Ö and ğ are invalid
  Ogr Num : Space is invalid
  Ogr-Num : Minus sign is invalid
  4.Ogr : First letter can not be a digit. Dot is invalid.
  ∑ , α , θ , a^2, π : Symbols are invalid

Naming Style

- when combining words, prefer underscores to capitalization

name	validity
`body_mass_index`	valid, recommended
`bodyMassIndex`	valid, but not common C style
`BodyMassIndex`	valid, but at least start with lowercase

Case Sensitivity

- lowercase and uppercase are significant

- `weight`, `Weight`, `WEIGHT`, `weIGhT` are all valid
- four different names

Printing Variables

- `printf` function can print values

  `printf(` *format-control-string* `,` *variable-arguments* `);`

- **Format control string:**
  - describes input/output format specificiers
  - each specification begins with a percent sign (%), ends with format specifier
  - parts other than format specifiers printed as is

- **Variable-arguments:**
  - variables correspond to each format specification in format-control-string
  - values follow order of their specifiers

Print Syntax

```
printf("... %SPEC1 ... %SPEC2 ...", VAL1, VAL2, ...);
```

- `%SPEC1` specifies `VAL1`, `%SPEC2` specifies `VAL2`, ...

Format Specifiers

type	specifier
int	%d
short	%hi
long	%ld
unsigned int	%u
unsigned short	%hu
unsigned long	%lu

type	specifier
float	%f
double	%lf

Body Mass Index

- measure for body leanness

- w : weight (mass, kg, integer)
- h : height (m, floating point)

$$bmi = \frac{w}{h^2}$$

BMI Table

category	**BMI range**
severe thinness	< 16
moderate thinness	16 - 18.5
normal	18.5 - 25
overweight	25 - 30
obese	> 30

BMI Program

```
/*
 * This program calculates and prints
 * the body mass index of a person
 * who weighs 65 kg and is 1.74 m tall.
 */

#include <stdio.h>   // printf

int main() {
    int weight = 65;   // mass, in kg
    double height = 1.74;   // in m
    double bmi = weight / (height * height);
    printf("Body mass index: %lf\n", bmi);
    return 0;
}
```

Formatting Output

- notation can be changed:
  - hexadecimal integer: `%x`
- size of output can be specified:
  - `M` digit integer: `%Md`
  - `M` digit integer with preceding 0s: `%0Md`
  - `N` digits after decimal point: `%.Nf`
  - `N` digits after decimal point, `M` in total (including point): `%M.Nf`
- value not changed, only formatted for display

Formatting Example

```
int main() {
    int weight = 65;   // mass, in kg
    double height = 1.74;   // in m
    double bmi = weight / (height * height);
    printf("Body mass index: %.1lf\n", bmi);

    return 0;
}
```

Printing Expressions

- format specifiers correspond to expressions, not variables
- result of calculation can be printed without assigning to variable

```
printf("Body mass index: %.1lf\n", weight / (height * height));
```

Data Input

- most programs will need to get data from the user
- working on fixed data: hard-coded

- get inputs
- process inputs and generate outputs
- print outputs

Input Function

- `scanf` function reads data from user
- and stores it in a variable
- variable has to be defined before input

- `&` in front of variable name

- format specification similar to `printf`
- also defined in `stdio.h`

Prompt

- `scanf` doesn't print a prompt
- use `printf` to print prompt first

Input Example

```
int weight;
printf("Enter weight (in kg): ");
scanf("%d", &weight);
```

BMI Program

```
/*
 * This program calculates and prints the body mass index of a person
 * whose weight and height are given by the user.
 */
#include <stdio.h>      // printf, scanf

int main() {
    int weight;    // mass, in kg
    printf("Enter weight (in kg): ");
    scanf("%d", &weight);
    double height;  // in m
    printf("Enter height (in m): ");
    scanf("%lf", &height);
    double bmi = weight / (height * height);
    printf("Body mass index: %.1lf\n", bmi);
    return 0;
}
```

Multiple Inputs

- multiple pieces of data can be read in a single input
- user can separate them with spaces or new lines

```
int weight;
double height;
printf("Enter weight (in kg) and height (in m): ");
scanf("%d %lf", &weight, &height);
```

Variable Definitions

- variables can be defined where they are first needed
- many programmers prefer to define all variables

  before first statement

Where Needed

```
int weight;  // mass, in kg
printf("Enter weight (in kg): ");
scanf("%d", &weight);

double height;  // in m

printf("Enter height (in m): ");
scanf("%lf", &height);

double bmi = weight / (height * height);
```

Before First Statement

```
int weight = 0;         // mass, in kg
double height = 0.0;    // in m
double bmi = 0.0;       // body mass index

printf("Enter weight (in kg): ");

scanf("%d", &weight);


printf("Enter height (in m): ");
scanf("%lf", &height);


bmi = weight / (height * height);
```

Data Sizes

```
int main()
{
    char    x;
    int     y;
    float   z;
    double  t;

    short int  x1;
    long  int  x2;

    unsigned char  y1;
    unsigned int   y2;
    unsigned short int z1;
    unsigned long  int z2;

    unsigned float y3;
}
```

Basic data types

Byte-size modifiers (default is long)

Sign modifiers (default is signed)

Compile-time error :
Floats can not be unsigned

Data Sizes

- getting data size: `sizeof`
  - size in bytes (long integer value)

```
printf("int: %ld bytes\n", sizeof(int));
printf("short int: %ld bytes\n", sizeof(short int));
printf("long int: %ld bytes\n", sizeof(long int));
printf("float: %ld bytes\n", sizeof(float));


printf("double: %ld bytes\n", sizeof(double));
printf("long double: %ld bytes\n", sizeof(long double));
```

Integer Limits

- defined in `limits.h`

- `INT_MIN`
- `SHRT_MIN`
- `LONG_MIN`
- `INT_MAX`
- `SHRT_MAX`
- `LONG_MAX`
- `UINT_MAX`
- `USHRT_MAX`
- `ULONG_MAX`

Data Type Ranges (Not standard)
(Signed)

Type name	Memory Size (Bytes)	Range of Values	
		Min	Max
char	1	-128	127
short int	2	-32,768	32,767
int, long int	4	-2,147,483,648	2,147,483,647

Data Type Ranges (Not standard) (Unsigned)

Type name	Memory Size (Bytes)	Range of Values	
		Min	Max
unsigned char	1	0	255
unsigned short int	2	0	65,535
unsigned int, unsigned long int	4	0	4,294,967,295

Data Type Ranges (Not standard)
(Fractional) (signed only)

Type name	Memory Size (Bytes)	Precision of floating point	Range of Values	
			Min	Max
float	4	Single-precision floating-point (7 fraction digits)	$-3.4 * 10^{-38}$	$3.4 * 10^{38}$
double	8	Double-precision floating-point (15 fraction digits)	$-1.7 * 10^{-308}$	$1.7 * 10^{308}$

Fixed-Width Integers

- sizes and ranges set by standard
- defined in `stdint.h`

- `int8_t`
- `int16_t`
- `int32_t`
- `int64_t`

- `uint8_t`
- `uint16_t`
- `uint32_t`
- `uint64_t`

Overflow

- assigned value not in variable type's range: overflow

```
short int price;
...
...
price = 100000;
```

Overflow

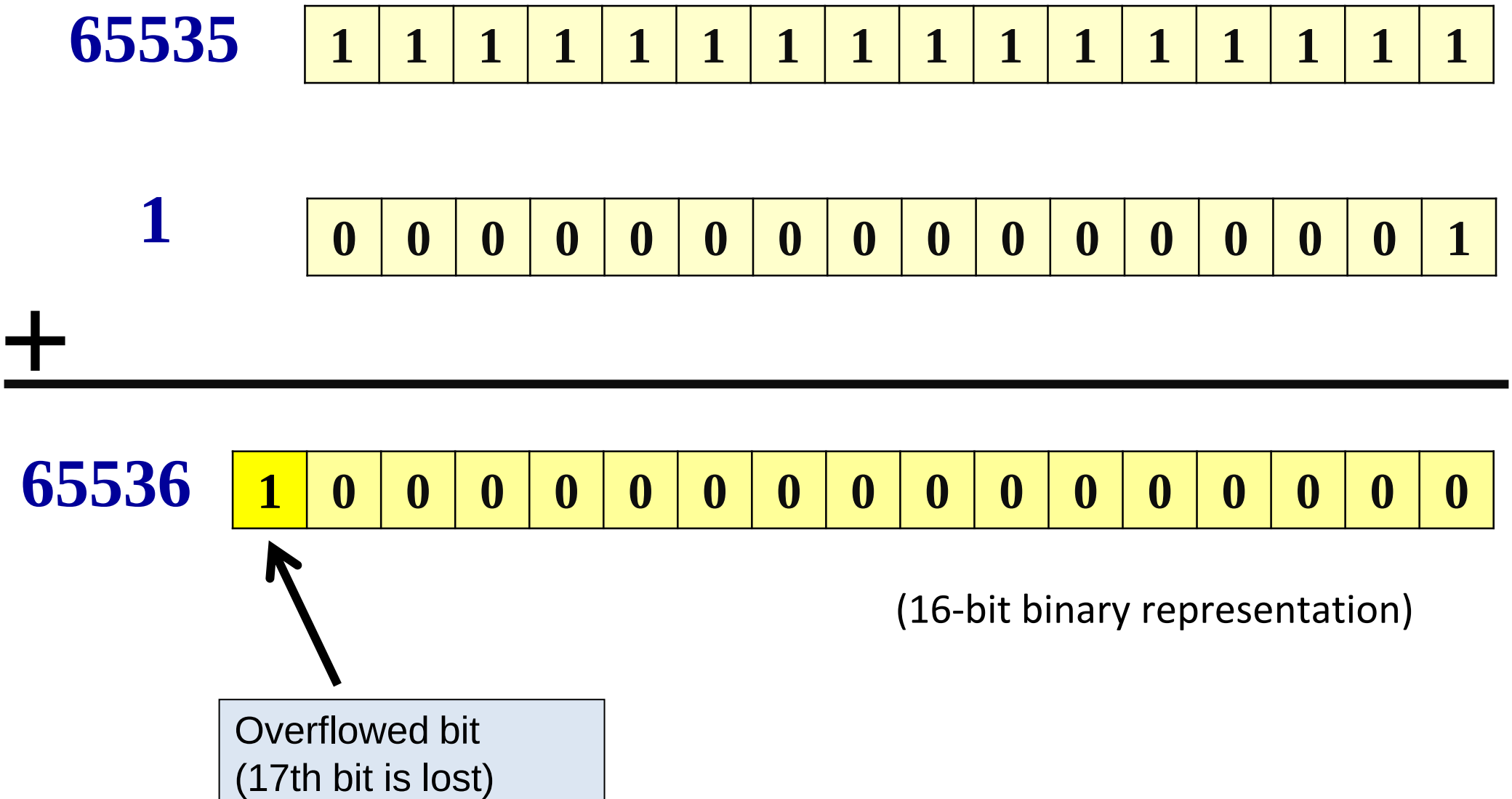
- When an arithmetic value is not within the range of a variable, overflow occurs.
- Example: Suppose the price variable is defined as **unsigned short integer**, which means its length is 2 bytes (16 bits).
- The range of allowed values are between 0 and 65535.
- Due to range overflow, the final value of price will be zero, not 65536.

```c
#include <stdio.h>

int main()
{
   unsigned short int price;
   price = 65535;
   price = price + 1;
   printf("Result = %d \n", price);
}
```

```
 65535
+    1
------
     0
```

Overflow


65535
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
+
65536
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
(16-bit binary representation)
Overflowed bit
(17th bit is lost)

Fractional Values

- fractional parts of values are discarded in integers

```
int price = 5.77;
```

- `price` becomes `5`

Inaccurate Value Problem

- inaccurate representation of floating point values

```
double price = 5.77;
int dollars = price;
int cents = (price - dollars) * 100;
```

- `dollars` becomes `5`
- `(5.77 - 5) * 100` gives `76.99999999999996`
- `cents` becomes `76`

Inaccurate Value Problem

- real numbers are represented using floating point
  - inherently inaccurate:

    0.1 + 0.2 = 0.299999999999
  - affects all programming languages

Incompatible Value

- type of expression not compatible with type of variable

```
int weight = "65";
double height = "1.74";
```

Integer Division

- if both operands are integers, division result is integer

- quotient:

  `x / y`

- remainder:

  `x % y`

Integer Division Example

- convert price in cents to dollars and cents

```
int price = 577;
int dollars = price / 100;
int cents = price % 100;
```

Unintended Integer Division

- convert price in cents to dollars

```
int price = 577;
double price = price / 100;
```

  - result is `5.0`, not `5.77`

- make an operand floating point:

```
double price = price / 100.0;
```

Type Conversion

- converting value from one type to another:

  type casting

- some conversions do not cause information loss
  - convert integer to floating point
- some conversions cause information loss
  - convert floating point to integer

Implicit Conversion

- explicit conversion: by programmer

- implicit conversion: by compiler
  - assign an integer value to floating point variable
  - add a floating point value and an integer value

Explicit Conversion

- syntax for type conversion:

```
(TYPE) EXPRESSION
```

Unintended Integer Division

- make `price` floating point:

```
int price = 577;
double dollars = (double) price / 100;
```

Incorrect Fix

- conversion applies to first expression

```
double price = (double) (price / 100);   // 5.0
```