10. Data Types, Choosing, Conversion, Constants

CPSC 120: Introduction to Programming
Pratishtha Soni~ CSU Fullerton

Agenda

0. Sign-in sheet
1. Technical Q&A
2. Data Types
3. Strings
4. Choosing a Data Type
5. Type Conversion
6. Constants

1. Technical Q&A

Technical Q&A

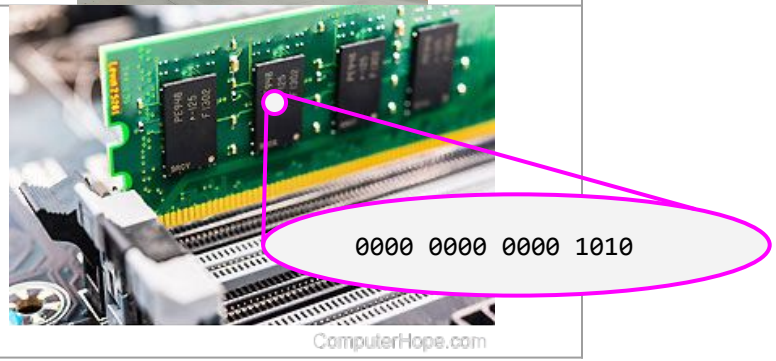
Let’s hear your noted questions about...

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

Reminder: write these questions in your notebook during lab

2. Data Types

Review: Objects and Variables

Kind of Object	Name	Picture
building	Engineering Building (E)	
piece of data stored in memory	variable `int score{ 10 };`	

Review: Variable Declaration and Initialization

statement:

data-type identifier **{** *expression* **};**

Examples:

```
int count{ 0 };

double temperature{ 98.6 };
```

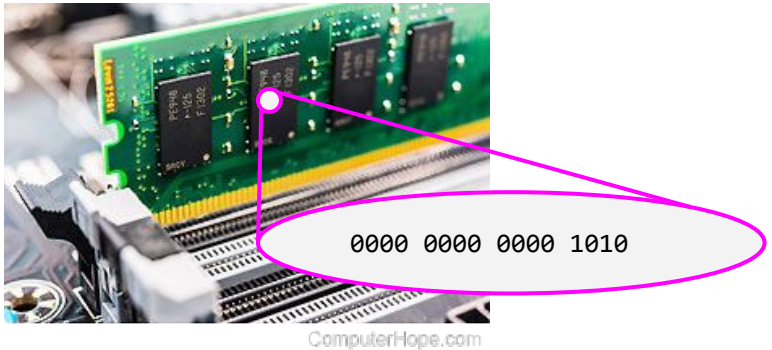
Semantics:

- Declare variable with name *identifier* and type *data-type*
- Initialize *identifier* to store the result of evaluating *expression*

Next: how to fill in *data-type*, *expression*, *identifier*

Review: Data Types

- **Data type**
  - “Type” for short
  - Format for storing an object in memory
  - Defined operations in source code
- Will explore many data types
- For today, just two...
- **`int`**: integer (whole number)
- **`double`**: double-precision floating-point number (decimal number)




```
int count{ 0 };

double temperature{ 98.6 };
```

Types We Are Using

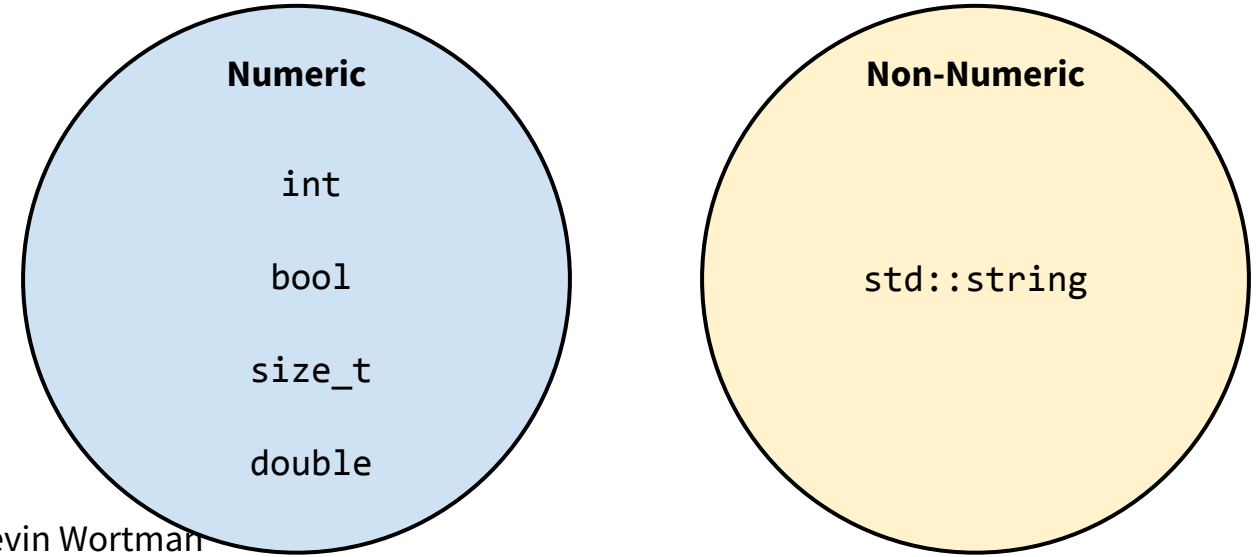
Type	Appropriate For
`int`	whole number
`double`	decimal number
`bool`	true/false
`size_t`	size of a container
`std::string`	text

Data Types and Type Categories

- Organize data types into **categories**
- Based on
  - how they work
  - how we use them in code

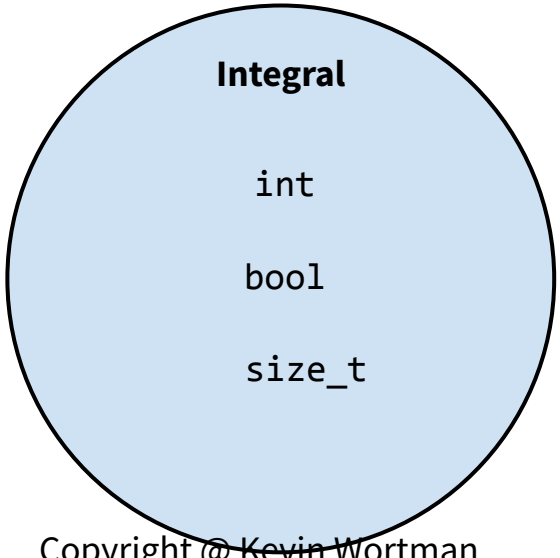
Numeric versus Non-Numeric

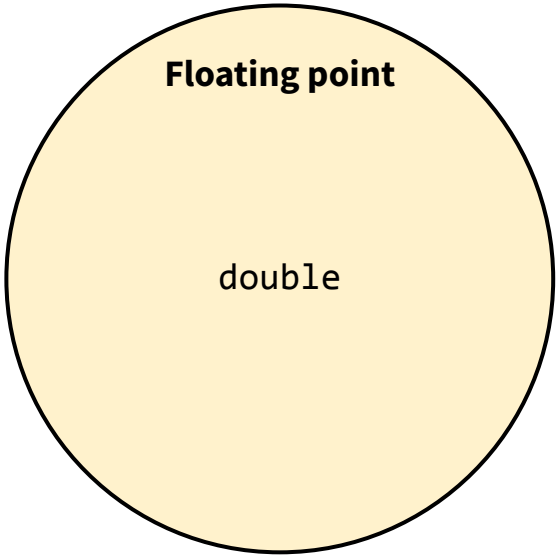
- **Numeric** type: stores a number
- **Non-numeric** type: anything else



Integral versus Floating Point

- **Integral** type category: numeric; whole numbers
- **Floating point** type category: numeric; decimal numbers



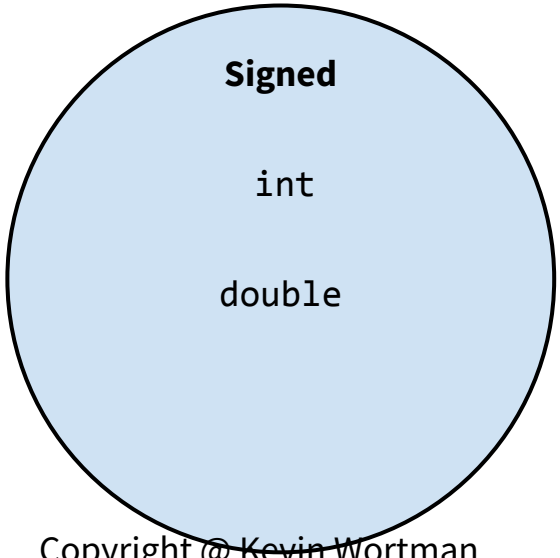


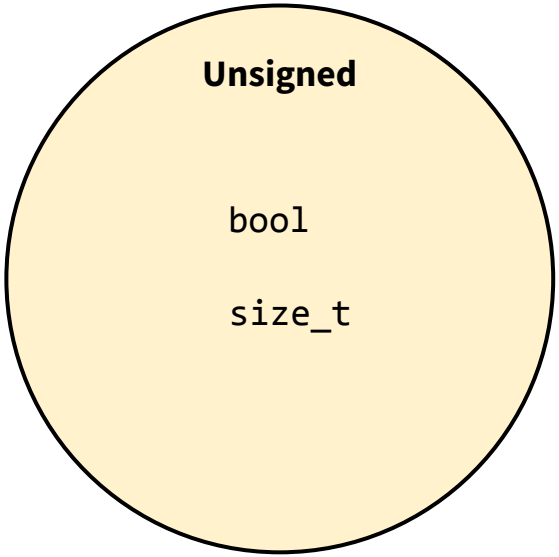
Neither

`std::string`

Signed versus Unsigned

- **Signed:** capable of representing negative number
- **Unsigned:** non-negative numbers only



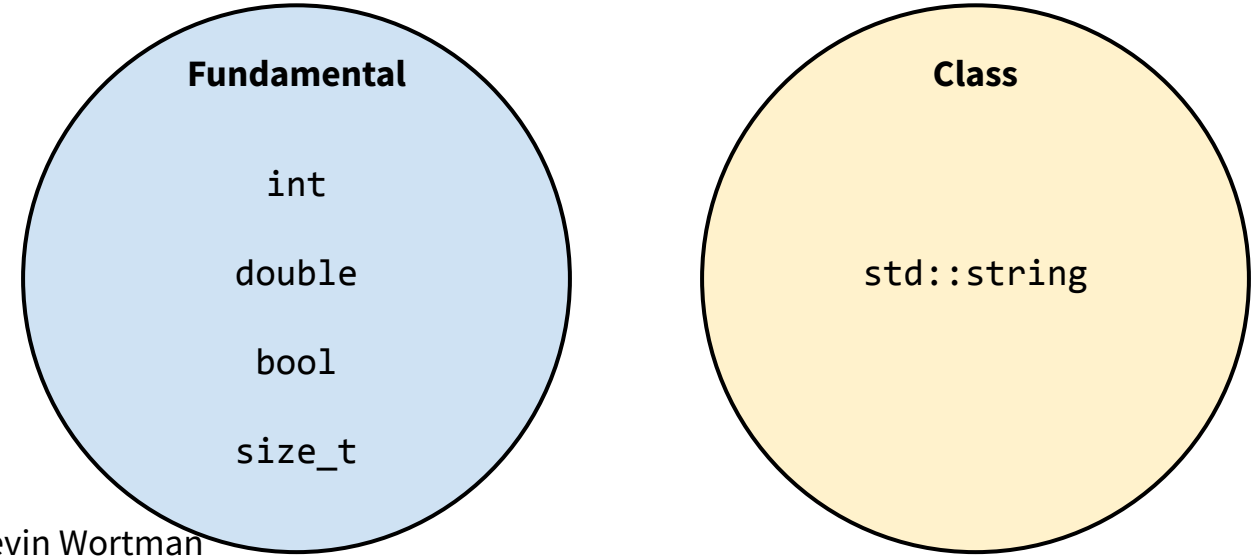


Neither

`std::string`

Fundamental versus Class Type

- **Fundamental** type: defined by CPU hardware
- **Class** type: defined by source code; has constructor and member functions



Floating Point Imprecision

- A floating point type (`double`) uses scientific notation

$$mantissa \times 10^{exponent}$$

$$4.732 \times 10^{4} = 47{,}320$$

- Limited number of digits in mantissa
- May be no effect from adding/subtracting a small number with a big one
- **Floating point imprecision:** when arithmetic on floating point types produces mathematically-incorrect values

Demo: Floating Point Imprecision

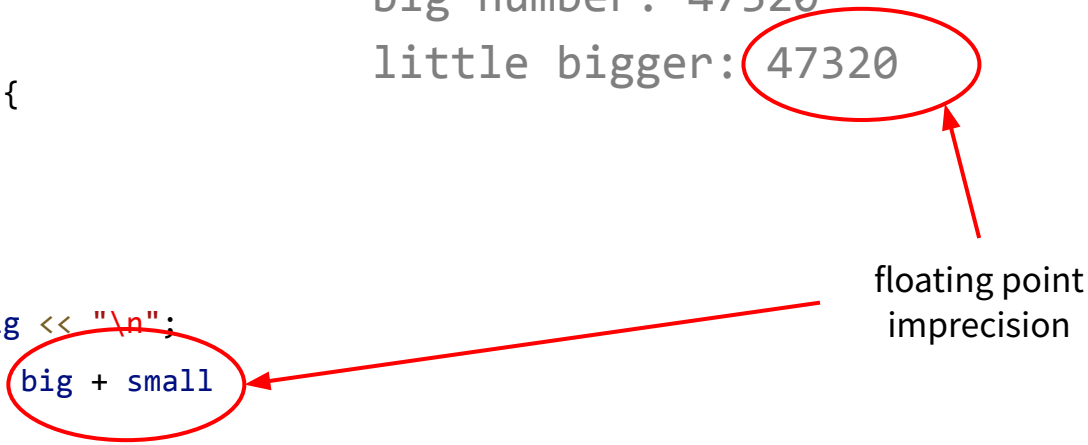
```
#include <iostream>

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

double big{47320};
double small{.001};
std::cout << "big number: " << big << "\n";
std::cout << "little bigger: " << big + small
          << "\n";

return 0;
}
```

```
$ ./a.out
big number: 47320
little bigger: 47320
```



size_t

- `size_t`: data type for the size of a container
- Numeric
- Integral
- Unsigned
- Important for `std::vector`
  - Coming soon
- Alias for `unsigned int`

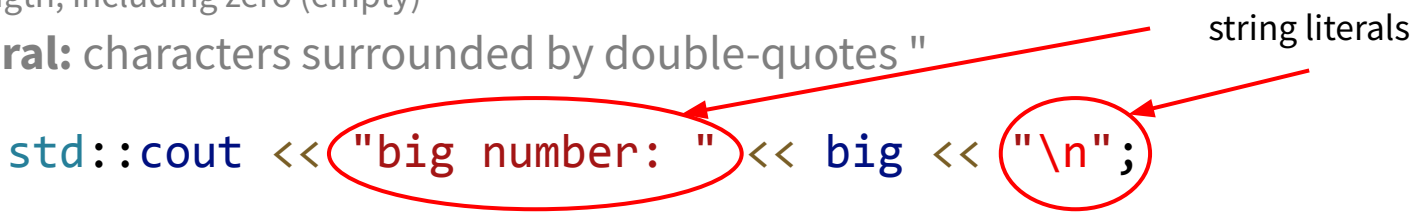
Type Alias

- Type **alias**: alternative name for a data type
- More descriptive name
- Single Point of Truth
  - Example: we could change `size_t` to be an alias for `unsigned long` by only changing one line of code

3. Strings

Characters and Strings

- **Character:** a symbol
  - Example: any keyboard key
- Character **encoding**: code for representing characters in a numeric type
  - ASCII
- **String:** a sequence of characters
  - Human-readable text
  - Any length, including zero (empty)
- **String literal:** characters surrounded by double-quotes "



std::string

- std::basic_string: data type for a string using ASCII encoding
- **std::string**: default string type
  - Alias for std::basic_string
- Class type

Class Types Initialize by Default

- Review: every primitive type declaration must also initialize

```
int length{0}; // OK
int width; // logic error
```

- However, class types initialize themselves
  - OK to omit initialization
  - std::string is a class type

```
std::string first{"Ada"}; // OK
std::string last; // OK, holds empty string ""
```

4. Choosing a Data Type

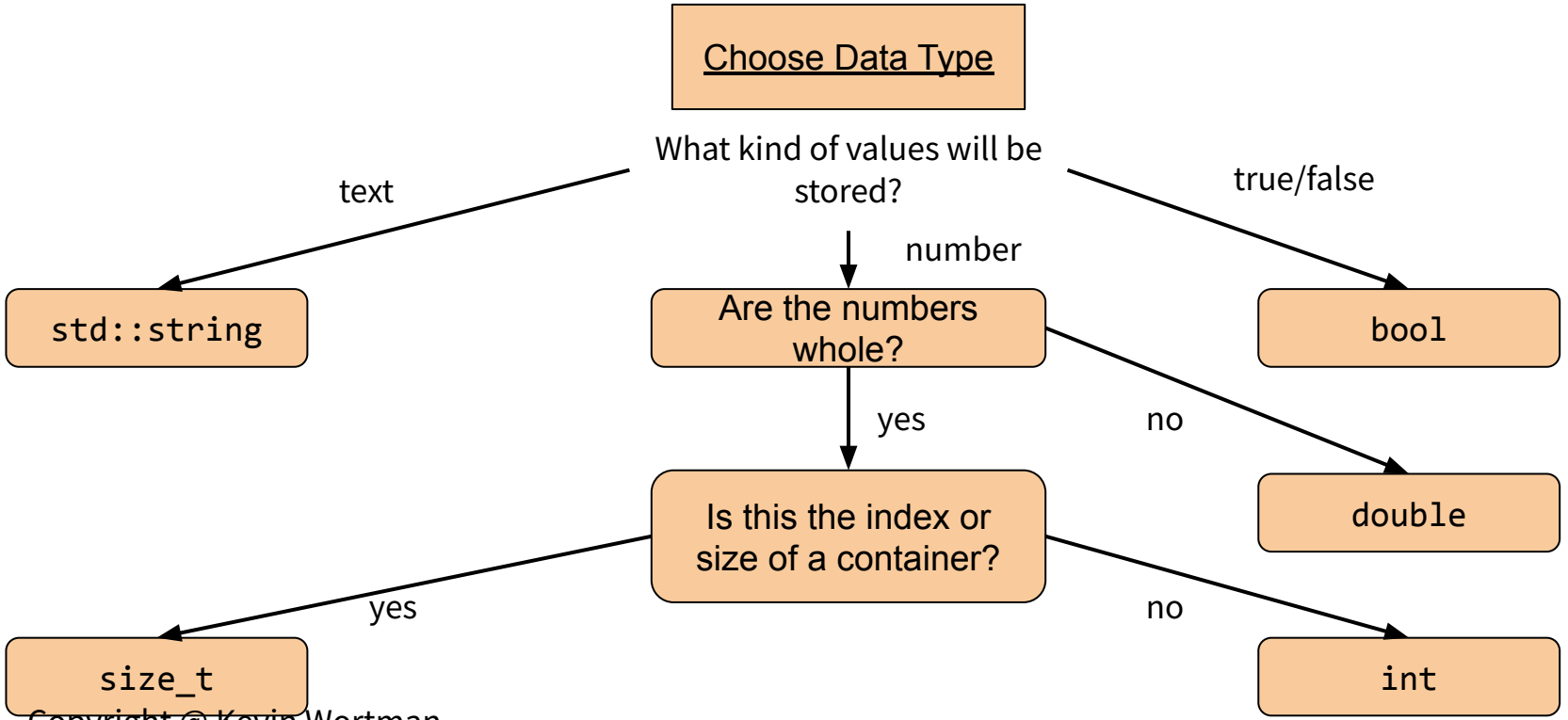
When to Choose a Data Type

- Need to choose a data type when...
  - Declaring a variable
  - (later) Designing a class
  - Exam questions 😊
- Problem solving
- Which data type is best fit?

Principles for Choosing a Data Type

- Data type should be **necessary** and **sufficient** for values stored in the variable
- **Sufficient:** every anticipated value could be represented
  - Text → std::string
  - Number → numeric
    - May be negative → signed
    - May be decimal → double
  - (data type actually works)
- **Necessary:** data type is not more than you need
  - True/false → bool (not it or std::string)
  - Number → numeric (not std::string)
  - Whole number → integral (not floating point)
  - (avoid waste)

Flowchart: Choose Data Type



5. Type Conversion

Implicit Semantics

- **Implicit** (adj): implied, rather than expressly stated
  - Ex.: when a barista calls your name, implicity you should pick it up
- Some semantics are implicit
  - Automatically happen even if you don’t write code for it
- Automates tedious programming tasks
  - Division of labor

Mixed Expressions

- **Mixed expression***:* involves values of different types

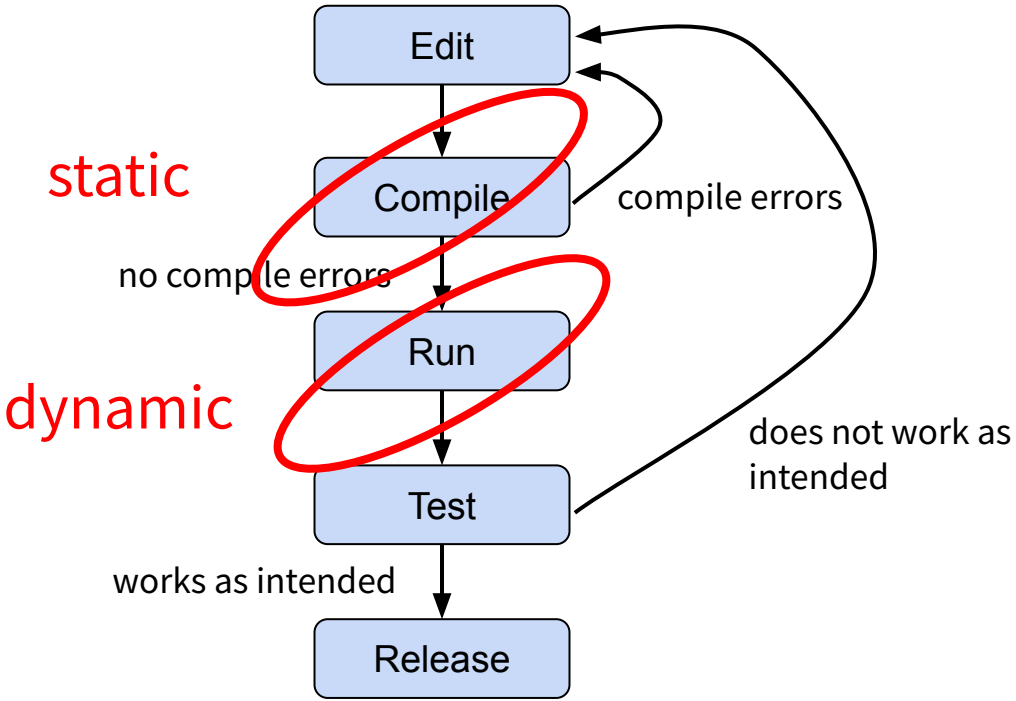
```
int x{3};
double y{2.5};
std::cout << x+y;
```

- **Implicit Type Promotion:** in a mixed expression, *the narrower type is implicitly converted to the wider type*
  - `double` is wider than `int`
  - `double`s are implicitly promoted to `int`s
  - `cout` above prints 5.5, not 5

Type Casting

- *Type cast:* expression to explicitly convert a value to a different data type
- Several alternatives
  - C-style cast
  - Functional cast
  - **static_cast** (preferred)

Static versus Dynamic



Static versus Dynamic

Static or Dynamic?	Happens When?	Aspects of Code Below
Static	**Compile-time** = while code is being compiled	• Checks: header, format, lint • Compile error checks ○ Ex: `0.0` is right type for `price`
Dynamic	**Runtime** = while program is running	• Printing `"Enter price: "` • Initializing `price` to 0.0 • User typing in new value

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

static_cast

- static_cast: function that converts a value to a different data type
- Built-in function
  - Doesn't need `#include`
- Creates and returns a new different object
- Compiler determines how to convert statically
  - (also a `dynamic_cast`)

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

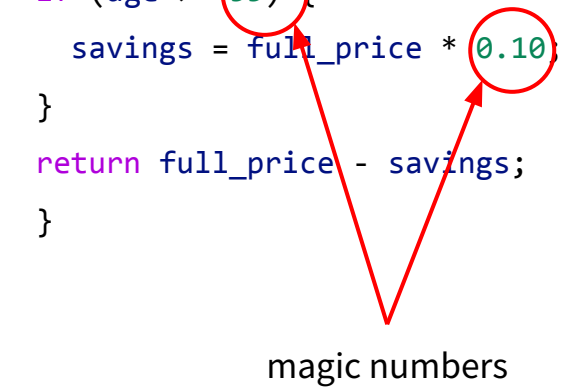
(not 11.5)

6. Constants

Understand the Problem: Magic Numbers

- **Magic number:** numeric literal that represents a business logic concept
- Unclean
  - What does 55 mean?
  - What does 0.10 mean?
- Labor-intensive to change
  - Policies are likely to change someday
  - Hard work to find and change all 55, 0.10 occurrences
  - (Division of labor)

```
double PriceAfterSeniorDiscount(
  double full_price, int age) {
  double savings{0.0};
  if (age >= 55) {
    savings = full_price * 0.10;
  }
  return full_price - savings;
  }
```



Review: Single Point Of Truth (SPOT)

- **Single Point Of Truth (SPOT):** an idea is represented in **only one place**
  - aka **Don't Repeat Yourself (DRY)**
- General principle
- In programming:
  - define a "magic number" **once** in a **constant variable**
  - define an algorithm **once** in a **function**
- **Ideal Division of Labor** principle:
  - **humans** should not copy-paste the same idea
  - tedious, error-prone
  - **computer** should do that by looking at the **SPOT**

Constant Variables

- Data type may be preceded by `const`
- const variables
  - must be initialized
  - cannot be re-assigned
  - so never change
- Best practice: declare a constant variable to represent each magic number
- Code becomes readable
- Easier to change later
- Example: chromium::cc::layers::Viewport::kPinchZoomSnapMarginDips

```
const int kMinimumAgeForSeniorDiscount{55};
const double kSeniorDiscountPercentage{10.0};

double PriceAfterSeniorDiscount(
  double full_price, int age) {
  double savings{0.0};
  if (age >= kMinimumAgeForSeniorDiscount) {
  savings = full_price *
            kSeniorDiscountPercentage / 100.0;
  }
  return full_price - savings;
}
```