# CSI 32
# Chapter 3: Objects, Types and Values

# We will talk about

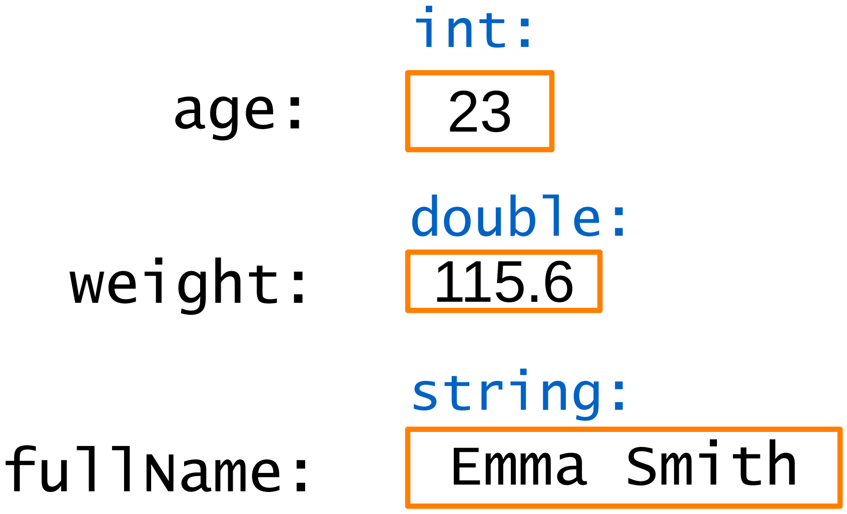
- Input
- Output
- Types: int, float, string
- Objects
- Operations and operators

# Types and objects

```
int age = 23;

double weight = 115.6

string fullName = “Emma Smith”
```

# Types and objects

```
int age = 23;

double weight = 115.6

string fullName = “Emma Smith”
```



int:

age: 23

double:

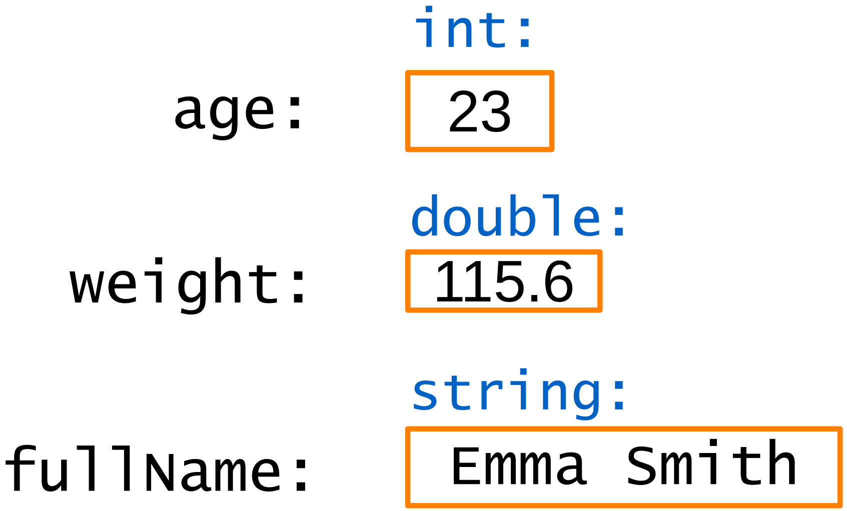
weight: 115.6

string:

fullName: Emma Smith

# Types and objects

```
int age = 23;
double weight = 115.6
string fullName = “Emma Smith”
```



int:

age: 23

double:

weight: 115.6

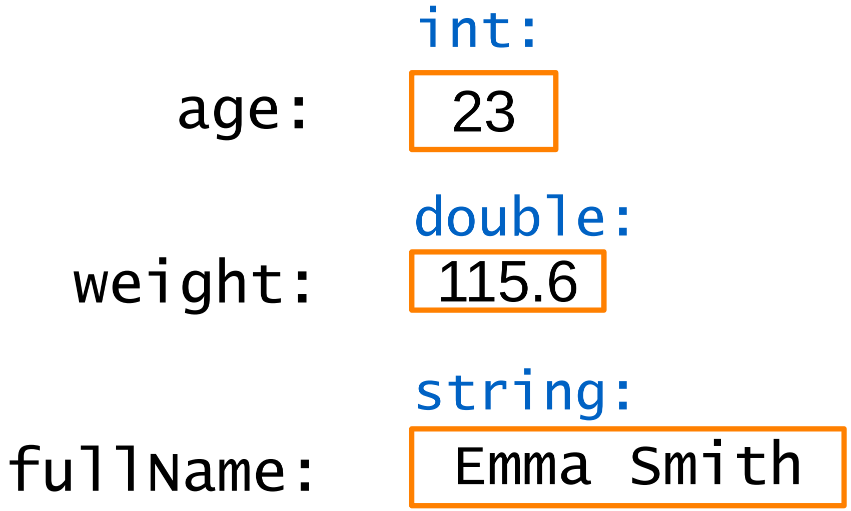
string:

fullName: Emma Smith

- An *object* is a region of memory with a *type* that specifies what kind of information can be placed into it
- OOP (*Object Oriented Programming*): *object* is an *instance* of a *class*
-

# Types and objects

```
int age = 23;

double weight = 115.6

string fullName = “Emma Smith”
```



int:

age: 23

double:

weight: 115.6

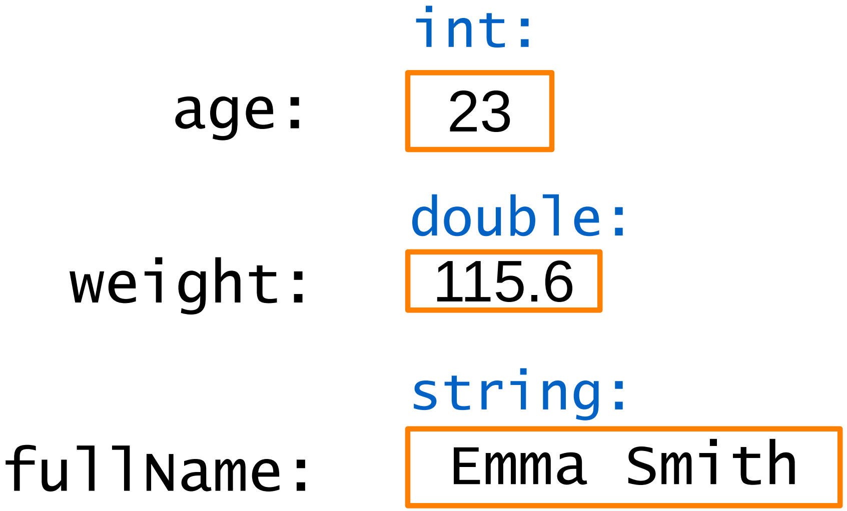
string:

fullName: Emma Smith

- An *object* is a region of memory with a *type* that specifies what kind of information can be placed into it
- A named object is called *variable*

# Types and objects

```
int age = 23;

double weight = 115.6

string fullName = “Emma Smith”
```



int:

age: 23

double:

weight: 115.6

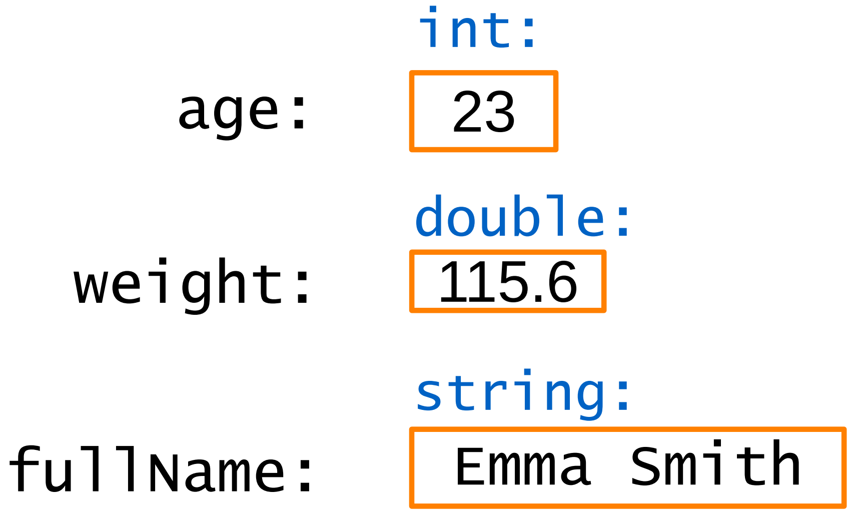
string:

fullName: Emma Smith

- An *object* is a region of memory with a *type* that specifies what kind of information can be placed into it
- A named object is called *variable*
- *Type* defines a set of possible values and a set of operations (for an object)

# Types and objects

```
int age = 23;

double weight = 115.6

string fullName = “Emma Smith”
```



int:

age: 23

double:

weight: 115.6

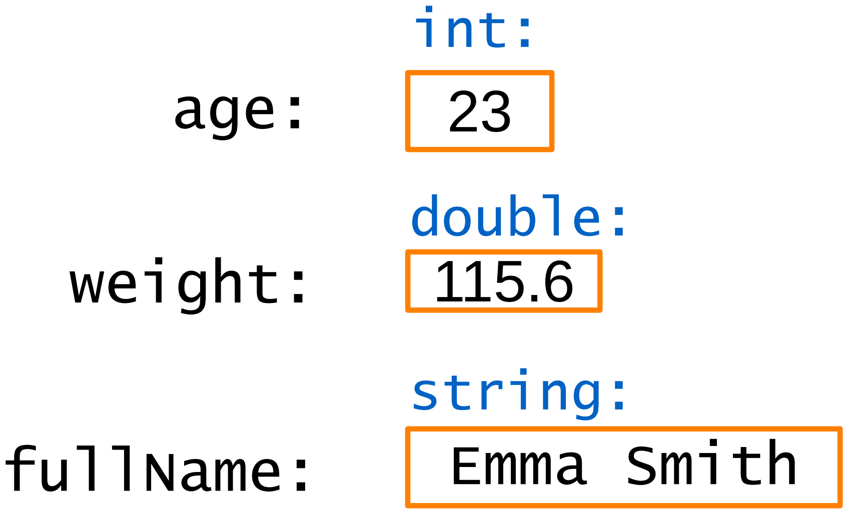
string:

fullName: Emma Smith

- The data items we put into *variables* are called *values*, or
- *value* is a set of bits in memory interpreted according to a type

# Types and objects

```
int age = 23;
double weight = 115.6
string fullName = “Emma Smith”
```



- int type, on a typical computer, uses 4 bytes (4*8 = 32 bits)
- double uses 8 bytes
- string takes up different amounts of space

# Some other built-in types

| built-in types | brief description | # of bytes (usually) |
|---|---|---|
| `char` x | x is a character | 1 |
| `bool` x | x is a boolean (`true` or `false`) | 1 |
| `float` x | x is a floating point number (short double) | 4 |
| `long int` x<br>`long` x | long integer | 8 |
| unsigned int x<br>unsigned x | non-negative integers from [0, $2^{32}$-1] | 4 |

# Declaration, definition, initialization, assignment

```
int age; // a definition

int myFunc(int, double, char);
// a declaration

age = 23; // an initialization
age = 30; // an assignment

double weight = 115.6
// definition and initialization
// definition can provide an
// initial value
```

- A *declaration* is a statement that gives a name to an object
- The statement that introduces a new name into a program and sets aside memory for a variable is called a *definition*
- *Initialization* gives a variable its initial value
- Assignment is giving a variable a new value

# Input and output with strings

```
#include <iostream>

using namespace std;

int main()
{
    string firstName, lastName;
    cout << "Enter your first name: ";
    cin >> firstName;
    cout << "Enter your last name: ";
    cin >> lastName;

    cout << "Hello, ";
    cout << firstName << " " << lastName << "! ";
    cout << "How are you?\n";
    return 0;
}
```

see InputOutputWithStrings.cpp

# In-class work – part 1

Grab the file In-classWork1.cpp from our website: https://natna.info/CSI32/notes.html and follow the instructions given there.

# In-class work – part 2

Grab the file In-classWork2.cpp from our website: https://natna.info/CSI32/notes.html and follow the instructions given there.

# Integers and strings

- Strings
  - cin >> reads a word
  - cout << writes
  - + concatenates
  - += s adds the string s at end
  - ++ is an error
  - - is an error
  - … see pages 66-67

- Integers and floating-point numbers
  - cin >> reads a number
  - cout << writes
  - + adds
  - += n increments by the int n
  - ++ increments by 1
  - - subtracts
  - … see pages 66-67

The type of a variable determines which operations are valid and what their meanings are for that type

(it's called "overloading" or "operator overloading")

# Type safety

- Every object is given a type when it is defined
- A program (part of the program) is type-safe when objects are used according to the rules for their type:
  - A variable is used only after it is initialized
  - Only operations defined for the variable's declared type are applied
  - Every operation defined for a variable leaves the variable with a valid value
- A C++ compiler cannot guarantee complete type safety

# Type safety: safe and unsafe conversions

- Safe conversions
  - `bool` to `char`
  - `bool` to `int`
  - `bool` to `double`
  - `char` to `int`
  - `char` to `double`
  - `int` to `double`

For some computers, for a really large int we can suffer a loss of precision when converting to double

- Unsafe conversions
  - When converting to a value of another type that doesn’t equal to the original value

grab at typeSafeyExamples.cpp at our website https://natna.info/CSI32/notes.html

follow the to-do instructions

# Type safety: safe conversions

- Every object is given a type when it is defined.
- A program (part of the program) is type-safe when objects are used according to the rules for their type:
  - A variable is used only after it is initialized
  - Only operations defined for the variable's declared type are applied
  - Every operation defined for a variable leaves the variable with a valid value
- A C++ compiler cannot guarantee complete type safety

# C++11 Hint

- C++ 11 introduced an initialization notation that outlaws narrowing conversions

```
double x{2.7}; // OK
int y{x};         // error: double → int might narrow

int a{1000};   // OK
char b{a};       // error: int → char might narrow

char b1{1000}; // error, assuming 8-bit chars (28 = 256), hence narrowing
char b2{48};    // OK
```

# C++14 Hint

- You can use the type of an *initializer* as the type of a variable
  - // “auto” means “the type of the initializer”
  - `auto x = 1;` // 1 is an int, so x is an int
  - `auto y = 'c';` // 'c' is a char, so y is a char
  - `auto d = 1.2;` // 1.2 is a double, so d is a double
  - `auto s = "Howdy";` // "Howdy" is a string literal of type const char[]<br>// so don’t do that until you know what it means!
  - `auto sq = sqrt(2);` // sq is the right type for the result of sqrt(2)<br>// and you don’t have to remember what that is
  - `auto duh;` // error: no initializer for auto

# Resources used for these slides

- slides provided by B. Stroustrup at https://www.stroustrup.com/PPP2slides.html

- Class textbook