Types, Values, Variables & Assignment

EECS 211

Winter 2018

Road map

- Strings and string I/O
- Integers and integer I/O
- Types and objects *
- Type safety

* Not as in object orientation—we'll get to that much later.

Input and output

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    cout << "Please enter your name: ";

    string first_name;
    cin >> first_name;

    cout << "Hello, " << first_name << '\n';
}
```

Using libraries

```
#include <iostream>
#include <string>
```

Includes the I/O stream library header, which lets us refer to cin and cout to do I/O, and the string library header, which lets us use strings.

Using libraries

```
#include <iostream>
#include <string>
```

Includes the I/O stream library header, which lets us refer to cin and cout to do I/O, and the string library header, which lets us use strings.

```
using namespace std;
```

Tells C++ to let us refer to things in the **st**andar**d** library without prefixing them with std::. Otherwise we'd have to write std::cin.

Main function

```
int main()
{

    ⋮

}
```

Wraps the *main function* of every program.

Input and type

```
string first_name;
cin >> first_name;
```

- We *define* a variable first_name to have type string
  - This means that first_name can hold textual data
  - The type of the variable determines what we can do with it
- Here, cin>>first_name; reads characters until it sees whitespace (“a word”)

Reading multiple words

```
int main()
{
    cout << "Please enter your first and second names:\n";

    string first;
    string second;
    cin >> first >> second;
    string name = first + ' ' + second;

    cout << "Hello, " << name << '\n';
}
```

Fine print: left out the includes and using, since every program will have those from now on

Syntax of cin

```
cin >> a >> b;
```

means the same thing as

```
cin >> a;
cin >> b;
```

Syntax of cin

```
cin >> a >> b;
```

means the same thing as

```
cin >> a;
cin >> b;
```

IS THIS MAGIC?

Syntax of cin

cin $>>$ a $>>$ b;

means the same thing as

cin $>>$ a;
cin $>>$ b;

IS THIS MAGIC? No, because

- cin $>>$ a returns a reference to cin

Syntax of cin

```
cin >> a >> b;
```

means the same thing as

```
cin >> a;
cin >> b;
```

IS THIS MAGIC? No, because

- cin >> a returns a reference to cin
- cin >> a >> b means (cin >> a) >> b

Syntax of cin

```
cin >> a >> b;
```

means the same thing as

```
cin >> a;
cin >> b;
```

IS THIS MAGIC? No, because

- cin >> a returns a reference to cin
- cin >> a >> b means (cin >> a) >> b
- *i.e.,* operator>> is *left associative*

Syntax of cin

```
cin >> a >> b;
```

means the same thing as

```
cin >> a;
cin >> b;
```

IS THIS MAGIC? No, because

- cin >> a returns a reference to cin
- cin >> a >> b means (cin >> a) >> b
- *i.e.,* operator>> is *left associative*
- (same deal for cout and operator<<)

Reading integers

```
int main()
{
    cout << "Please enter your first name and age:\n";
    string first_name;
    int age;
    cin >> first_name >> age;
    cout << "Hello, " << first_name << ", age "
         << age << '\n';
}
```

Integers and numbers

string s	int x or double x

Integers and numbers

string s	int x or double x
cin >> s reads a word	cin >> x reads a number

Integers and numbers

string s	int x or double x
cin >> s reads a word	cin >> x reads a number
cout << s writes	cout << x writes

Integers and numbers

string s	int x or double x
cin >> s reads a word	cin >> x reads a number
cout << s writes	cout << x writes
s1 + s2 concatenates	x1 + x2 adds

Integers and numbers

string s	int x or double x
cin >> s reads a word	cin >> x reads a number
cout << s writes	cout << x writes
s1 + s2 concatenates	x1 + x2 adds
++s is an error	++x increments in place

Integers and numbers

string s	int x or double x
cin >> s reads a word	cin >> x reads a number
cout << s writes	cout << x writes
s1 + s2 concatenates	x1 + x2 adds
++s is an error	++x increments in place

The type of a variable determines

- what operations are valid
- and what they mean for that type

Names, a/k/a identifiers

A legal name in C++

- starts with a letter,

Names, a/k/a identifiers

A legal name in C++

- starts with a letter,
- contains only letters, digits, and underscores, and

Names, a/k/a identifiers

A legal name in C++

- starts with a letter,
- contains only letters, digits, and underscores, and
- isn't a language keyword (*e.g.,* if).

Names, a/k/a identifiers

A legal name in C++

- starts with a letter,
- contains only letters, digits, and underscores, and
- isn't a language keyword (*e.g.*, if).

Which of these names are illegal? Why?

- purple line
- number_of_bees
- jflsiejslf_
- else
- timetomarket
- Fourier_transform
- 12x
- y2

Names, a/k/a identifiers

A legal name in C++

- starts with a letter,
- contains only letters, digits, and underscores, and
- isn't a language keyword (*e.g.*, if).

Which of these names are illegal? Why?

- purple line (space not allowed)
- number_of_bees
- jflsiejslf_
- else (keyword)
- timetomarket (bad punctuation)
- Fourier_transform
- 12x (starts with a digit)
- y2

Also, don’t start a name with an underscore

The compiler might allow it, but technically such names are reserved for the system

Choose meaningful names

- Abbreviations and acronyms can be confusing: myw, bamf, TLA

Choose meaningful names

- Abbreviations and acronyms can be confusing: myw, bamf, TLA
- Very short names are meaningful only when there's a convention:
  - x is a local variable
  - n is an int
  - i is a loop index

Choose meaningful names

- Abbreviations and acronyms can be confusing: myw, bamf, TLA
- Very short names are meaningful only when there's a convention:
  - x is a local variable
  - n is an int
  - i is a loop index
- The length of a name should be proportional to its scope

Choose meaningful names

- Abbreviations and acronyms can be confusing: myw, bamf, TLA
- Very short names are meaningful only when there's a convention:
  - x is a local variable
  - n is an int
  - i is a loop index
- The length of a name should be proportional to its scope
- Don't use overly long names

Choose meaningful names

- Abbreviations and acronyms can be confusing: myw, bamf, TLA
- Very short names are meaningful only when there's a convention:
  - x is a local variable
  - n is an int
  - i is a loop index
- The length of a name should be proportional to its scope
- Don't use overly long names
  - Good:
    - partial_sum
    - element_count

Choose meaningful names

- Abbreviations and acronyms can be confusing: myw, bamf, TLA
- Very short names are meaningful only when there's a convention:
  - x is a local variable
  - n is an int
  - i is a loop index
- The length of a name should be proportional to its scope
- Don't use overly long names
  - Good:
    - partial_sum
    - element_count
  - Bad:
    - the_number_of_elements
    - remaining_free_slots_in_the_symbol_table

Simple arithmetic

```
#include <cmath> // For sqrt
⋮
int main()
{
    cout << "Please enter a floating-point number: ";

    double f;
    cin >> f;

    cout <<   "f      == " << f
         << "\nf + 1 == " << f + 1
         << "\n2f    == " << 2 * f
         << "\n3f    == " << 3 * f
         << "\nf²    == " << f * f
         << "\n√f    == " << sqrt(f) << '\n';
}
```

A simple computation

```
#include <cmath>
#include <iostream>

using namespace std;

int main()
{
    double r;

    cout << "Please enter the radius: ";
    cin >> r;

    double c = 2 * M_PI * r;
    cout << "Circumference is " << c << '\n';
}
```

Types and literals

type	bits *	literals

* on current architectures

Types and literals

type	bits *	literals
bool	1 †	`true`, `false`

* on current architectures
† stored as 8 bits

Types and literals

type	bits *	literals
bool	1 †	`true`, `false`
char	8	`'a'`, `'B'`, `'4'`, `'/'`

* on current architectures

† stored as 8 bits

Types and literals

type	bits *	literals
bool	1 †	`true`, `false`
char	8	`'a'`, `'B'`, `'4'`, `'/'`
int	32 or 64	`0`, `1`, `765`, `-6`, `0xCAFE`

* on current architectures
† stored as 8 bits

Types and literals

type	bits *	literals
bool	1 †	`true`, `false`
char	8	`'a'`, `'B'`, `'4'`, `'/'`
int	32 or 64	`0`, `1`, `765`, `-6`, `0xCAFE`
long	64	`0L`, `1L`, `1000000000L`

* on current architectures
† stored as 8 bits

Types and literals

type	bits *	literals
bool	1 †	`true`, `false`
char	8	`'a'`, `'B'`, `'4'`, `'/'`
int	32 or 64	`0`, `1`, `765`, `-6`, `0xCAFE`
long	64	`0L`, `1L`, `10000000000L`
double	64	`0.0`, `1.2`, `-0.765`, `-6e15`

* on current architectures

† stored as 8 bits

Types and literals

type	bits *	literals
bool	1 †	`true`, `false`
char	8	`'a'`, `'B'`, `'4'`, `'/'`
int	32 or 64	`0`, `1`, `765`, `-6`, `0xCAFE`
long	64	`0L`, `1L`, `1000000000L`
double	64	`0.0`, `1.2`, `-0.765`, `-6e15`
string	varies	`"Hello, world!"` ‡

* on current architectures

† stored as 8 bits

‡ actually has type const char[], but converts automatically to string

Types

- C++ provides built-in types:
  - bool
  - (unsigned or signed) char
  - (unsigned) short
  - (unsigned) int
  - (unsigned) long
  - float
  - double

Types

- C++ provides built-in types:
  - bool
  - (unsigned or signed) char
  - (unsigned) short
  - (unsigned) int
  - (unsigned) long
  - float
  - double
- C++ programmers can define new types
  - called “user-defined types”
  - you’ll learn to define your own soon

Types

- C++ provides built-in types:
  - bool
  - (unsigned or signed) char
  - (unsigned) short
  - (unsigned) int
  - (unsigned) long
  - float
  - double
- C++ programmers can define new types
  - called “user-defined types”
  - you’ll learn to define your own soon
- The C++ standard library (STL) provides types
  - *e.g.,* string, vector, complex
  - technically these are user-defined, but they come with C++

Objects

- An *object* is some memory that can hold a value (of some particular type)

Objects

- An *object* is some memory that can hold a value (of some particular type)
- A *variable* is a named object

Objects

- An *object* is some memory that can hold a value (of some particular type)
- A *variable* is a named object
- A *definition* names and creates an object

Objects

- An *object* is some memory that can hold a value (of some particular type)
- A *variable* is a named object
- A *definition* names and creates an object
- An *initialization* fills in the initial value of a variable

Definition and initialization

```
int a;
```

Definition and initialization

```
int a;
```

Definition and initialization

```
int a;
```

a: -2340024

Definition and initialization

```
int a;
int b = 9;
```

a: -2340024

b: 9

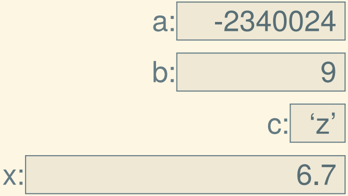
Definition and initialization

```
int a;
int b = 9;
auto c = 'z';    // c is a char
```

a:	-2340024
b:	9
c:	‘z’

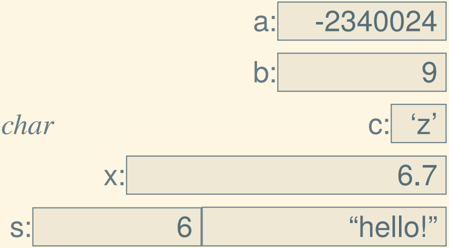
Definition and initialization

```
int a;
int b = 9;
auto c = 'z';   // c is a char
double x = 6.7;
```



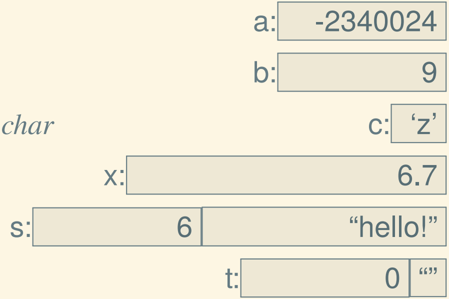
Definition and initialization

```
int a;
int b = 9;
auto c = 'z';   // c is a char
double x = 6.7;
string s = "hello!";
```



Definition and initialization

```
int a;
int b = 9;
auto c = 'z';    // c is a char
double x = 6.7;
string s = "hello!";
string t;
```



Language rule: Type safety

Definition: In a *type safe* language, objects are used only according to their types

Language rule: Type safety

Definition: In a *type safe* language, objects are used only according to their types

- Only operations defined for an object will be applied to it
- A variable will be used only after it has been initialized
- Every operation defined for a variable leaves the variable with a valid value

Language rule: Type safety

Definition: In a *type safe* language, objects are used only according to their types

- Only operations defined for an object will be applied to it
- A variable will be used only after it has been initialized
- Every operation defined for a variable leaves the variable with a valid value

Ideal: Static type safety

- A program that violates type safety will not compile
- The compiler reports every violation

Language rule: Type safety

Definition: In a *type safe* language, objects are used only according to their types

- Only operations defined for an object will be applied to it
- A variable will be used only after it has been initialized
- Every operation defined for a variable leaves the variable with a valid value

Ideal: Static type safety

- A program that violates type safety will not compile
- The compiler reports every violation

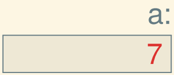
Ideal: Dynamic type safety

- An operation that violates type safety will not be run
- The program or run-time system catches every potential violation

Assignment and increment

The value of a variable may change.

```
int a = 7;
```



Assignment and increment

The value of a variable may change.

	a:
int $a = 7$;	7
$a = 9$;	

Assignment and increment

The value of a variable may change.

	a:
int a = 7;	7
a = 9;	9

Assignment and increment

The value of a variable may change.

	a:
int $a = 7;$	7
$a = 9;$	9
$a = a + a;$	

Assignment and increment

The value of a variable may change.

	a:
int $a = 7;$	7
$a = 9;$	9
$a = a + a;$	18

Assignment and increment

The value of a variable may change.

	a:
int a = 7;	7
a = 9;	9
a = a + a;	18
a += 2;	

Assignment and increment

The value of a variable may change.

	a:
int a = 7;	7
a = 9;	9
a = a + a;	18
a += 2;	20

Assignment and increment

The value of a variable may change.

	a:
int a = 7;	7
a = 9;	9
a = a + a;	18
a += 2;	20
++a;	

Assignment and increment

The value of a variable may change.

	a:
int a = 7;	7
a = 9;	9
a = a + a;	18
a += 2;	20
++a;	21

A type safety violation: implicit narrowing

Beware! C++ does not prevent you from putting a large value into a small variable (though a compiler may warn)

```
int main()
{
    int a = 20000;
    char c = a;
    int b = c;

    if (a != b)        // != means "not equal"
        cout << "oops!: " << a << " != " << b << '\n';
    else
        cout << "Wow! We have large characters\n";
}
```

Try it to see what value b gets on your machine

A type-safety violation: uninitialized variables

Beware! C++ does not prevent you from trying to use a variable before you have initialized it (though a compiler typically warns)

```
int main()
{
    int x;          // x gets a "random" initial value
    char c;         // c gets a "random" initial value
    double d;       // d gets a "random" initial value

    // not every bit pattern is a valid floating-point value, and on some
    // implementations copying an invalid float/double is an error:
    double dd = d; // potential error: some implementations

    // prints garbage (if you're lucky):
    cout << " x: " << x << " c: " << c << " d: " << d << '\n';
}
```

A type-safety violation: uninitialized variables

Beware! C++ does not prevent you from trying to use a variable before you have initialized it (though a compiler typically warns)

```
int main()
{
    int x;          // x gets a "random" initial value
    char c;         // c gets a "random" initial value
    double d;       // d gets a "random" initial value

    // not every bit pattern is a valid floating-point value, and on some
    // implementations copying an invalid float/double is an error:
    double dd = d; // potential error: some implementations

    // prints garbage (if you're lucky):
    cout << " x: " << x << " c: " << c << " d: " << d << '\n';
}
```

Always initialize your variables. Watch out: The debugger may initialize variables that don't get initialized when running

A technical detail

In memory, everything is just bits; type is what gives meaning to the bits:

- (bits/binary) 01100001 is the int 97 *and also* char 'a'
- (bits/binary) 01000001 is the int 65 *and also* char 'A'
- (bits/binary) 00110000 is the int 48 *and also* char '0'

```
char c = 'a';
cout << c;   // print the value of character c, which is 'a'
int i = c;
cout << i;   // print the integer value of the character c, which is 97
```

A word on efficiency

For now, don't worry about "efficiency"

- Concentrate on correctness and simplicity of code

A word on efficiency

For now, don't worry about "efficiency"

- Concentrate on correctness and simplicity of code

C++ is derived from C, low-level programming language

- C++'s built-in types map directly to computer main memory
  - a char is stored in a byte
  - an int is stored in a word
  - a double fits in a floating-point register
- C++'s built-in ops. map directly to machine instructions
  - $+$ on ints is implemented by an integer add operation
  - $=$ on ints is implemented by a simple copy operation
  - C++ provides direct access to most of facilities provided by modern hardware

A word on efficiency

For now, don't worry about "efficiency"

- Concentrate on correctness and simplicity of code

C++ is derived from C, low-level programming language

- C++'s built-in types map directly to computer main memory
  - a char is stored in a byte
  - an int is stored in a word
  - a double fits in a floating-point register
- C++'s built-in ops. map directly to machine instructions
  - $+$ on ints is implemented by an integer add operation
  - $=$ on ints is implemented by a simple copy operation
  - C++ provides direct access to most of facilities provided by modern hardware

A bit of philosophy

- One of the ways that programming resembles other kinds of engineering is that it involves tradeoffs.
- You must have ideals, but they often conflict, so you must decide what really matters for a given program.
  - Type safety
  - Run-time performance
  - Ability to run on a given platform
  - Ability to run on multiple platforms with same results
  - Compatibility with other code and systems
  - Ease of construction
  - Ease of maintenance
- Don't skimp on correctness or testing
- By default, aim for type safety and portability