Types, Values & Simple I/O

EECS 230
Spring 2016
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.
#include "eecs230.h"

int main()
{
    cout << "Please enter your name: ";
    string first_name;
    cin >> first_name;
    cout << "Hello, " << first_name << "\n";
}
Header files

#include ”eecs230.h”

Includes our course header file, which provides an interface to libraries, into your program
string first_name;
cin >> first_name;

- We *declare* 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")
```cpp
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 include, since every program will have that from now on.
Syntax of `cin`

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

means the same thing as

```
cin >> a;
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`

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

means the same thing as

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

**IS THIS MAGIC?** No, because

- `cin >> a` returns a reference to `cin`
- `cin >> a` returns a reference to `cin`
Syntax of cin

\[
\text{cin} \gg \text{a} \gg \text{b};
\]

means the same thing as

\[
\text{cin} \gg \text{a};
\text{cin} \gg \text{b};
\]

IS THIS MAGIC? No, because

- \text{cin} \gg \text{a} returns a reference to cin
- \text{cin} \gg \text{a} \gg \text{b} means (\text{cin} \gg \text{a}) \gg \text{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`

\[ \text{cin} \gg a \gg b; \]

means the same thing as

\[ \text{cin} \gg a; \quad \text{cin} \gg b; \]

**IS THIS MAGIC?** No, because

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


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

\relax string s | \text{T:int } x \text{ or } \text{T:double } x

The type of a variable determines what operations are valid and what they mean for that type.
Integers and numbers

<table>
<thead>
<tr>
<th>\texttt{string} \texttt{s}</th>
<th>\textit{T:int} \texttt{x} or \textit{T:double} \texttt{x}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{cin} \texttt{&gt;&gt;} \texttt{s} reads a word</td>
<td>\texttt{cin} \texttt{&gt;&gt;} \texttt{x} reads a number</td>
</tr>
</tbody>
</table>
Integers and numbers

<table>
<thead>
<tr>
<th>\texttt{\textbackslash relax string} \texttt{s}</th>
<th>\texttt{\textbackslash T:int} \texttt{x} or \texttt{\textbackslash T:double} \texttt{x}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{cin} \texttt{\gg} \texttt{s} reads a word</td>
<td>\texttt{cin} \texttt{\gg} \texttt{x} reads a number</td>
</tr>
<tr>
<td>\texttt{cout} \texttt{\ll} \texttt{s} writes</td>
<td>\texttt{cout} \texttt{\ll} \texttt{x} writes</td>
</tr>
</tbody>
</table>
Integers and numbers

<table>
<thead>
<tr>
<th>\texttt{\textbackslash relaxed} \texttt{string} \texttt{s}</th>
<th>\texttt{\textbackslash T:} \texttt{int} \texttt{x} \texttt{or} \texttt{\textbackslash T:} \texttt{double} \texttt{x}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{cin} \texttt{&gt;&gt;} \texttt{s} \texttt{reads a word}</td>
<td>\texttt{cin} \texttt{&gt;&gt;} \texttt{x} \texttt{reads a number}</td>
</tr>
<tr>
<td>\texttt{cout} \texttt{&lt;&lt;} \texttt{s} \texttt{writes}</td>
<td>\texttt{cout} \texttt{&lt;&lt;} \texttt{x} \texttt{writes}</td>
</tr>
<tr>
<td>\texttt{s1} + \texttt{s2} \texttt{concatenates}</td>
<td>\texttt{x1} + \texttt{x2} \texttt{adds}</td>
</tr>
</tbody>
</table>
## Integers and numbers

<table>
<thead>
<tr>
<th>\texttt{\textbackslash relax \texttt{\texttt{string}} s}</th>
<th>\texttt{\textbackslash T: int x} or \texttt{\textbackslash T: double x}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{cin \gg s} reads a word</td>
<td>\texttt{cin \gg x} reads a number</td>
</tr>
<tr>
<td>\texttt{cout \ll s} writes</td>
<td>\texttt{cout \ll x} writes</td>
</tr>
<tr>
<td>\texttt{s1 + s2} concatenates</td>
<td>\texttt{x1 + x2} adds</td>
</tr>
<tr>
<td>\texttt{++s} is an error</td>
<td>\texttt{++x} increments in place</td>
</tr>
</tbody>
</table>

The type of a variable determines what operations are valid and what they mean for that type.
## Integers and numbers

<table>
<thead>
<tr>
<th><code>\texttt{std::string}</code> $s$</th>
<th><code>\texttt{\textbf{T:int} \textbf{x} or \textbf{T:double} \textbf{x}}</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\texttt{cin &gt;&gt; s}</code> reads a word</td>
<td><code>\texttt{cin &gt;&gt; x}</code> reads a number</td>
</tr>
<tr>
<td><code>\texttt{cout &lt;&lt; s}</code> writes</td>
<td><code>\texttt{cout &lt;&lt; x}</code> writes</td>
</tr>
<tr>
<td><code>\texttt{s1 + s2}</code> concatenates</td>
<td><code>\texttt{x1 + x2}</code> adds</td>
</tr>
<tr>
<td><code>\texttt{++s}</code> is an error</td>
<td><code>\texttt{++x}</code> increments in place</td>
</tr>
</tbody>
</table>

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
- jflsiejlsf_
- else
- time$to$market
- 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)
- **time$to$market** (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

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
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

```cpp
int main()
{
    cout << "Please enter a floating-point number: ";
    double f;
    cin >> f;
    cout << "f == " << f << endl;
    cout << "f + 1 == " << f + 1 << endl;
    cout << "\n2f == " << 2 * f << endl;
    cout << "\n3f == " << 3 * f << endl;
    cout << "\nf² == " << f * f << endl;
    cout << "\n√f == " << sqrt(f) << 'n';
}
```
A simple computation

```cpp
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

<table>
<thead>
<tr>
<th>type</th>
<th>bits *</th>
<th>literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td></td>
<td>* stored as 8 bits</td>
</tr>
<tr>
<td>char</td>
<td>8</td>
<td>'a', 'b', '4', '/'</td>
</tr>
<tr>
<td>int</td>
<td>32 or 64</td>
<td>0, 1, 765, -6, 0xCAFE</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>0L, 1L, 10000000000L</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>0.0, 1.2, -0.765, -6e15</td>
</tr>
<tr>
<td>string</td>
<td>varies</td>
<td>&quot;Hello, world!&quot;</td>
</tr>
</tbody>
</table>

* on current architectures
**Types and literals**

<table>
<thead>
<tr>
<th>type</th>
<th>bits *</th>
<th>literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>1 †</td>
<td>true, false</td>
</tr>
</tbody>
</table>

* on current architectures
† stored as 8 bits
## Types and literals

<table>
<thead>
<tr>
<th>type</th>
<th>bits *</th>
<th>literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>1 †</td>
<td>true, false</td>
</tr>
<tr>
<td>char</td>
<td>8</td>
<td>'a', 'B', '4', '/'</td>
</tr>
</tbody>
</table>

* on current architectures
† stored as 8 bits
## Types and literals

<table>
<thead>
<tr>
<th>type</th>
<th>bits *</th>
<th>literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>1 †</td>
<td>true, false</td>
</tr>
<tr>
<td>char</td>
<td>8</td>
<td>'a', 'B', '4', '/'</td>
</tr>
<tr>
<td>int</td>
<td>32 or 64</td>
<td>0, 1, 765, −6, 0xCAFE</td>
</tr>
</tbody>
</table>

* on current architectures
† stored as 8 bits
## Types and literals

<table>
<thead>
<tr>
<th>type</th>
<th>bits *</th>
<th>literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>1 †</td>
<td>true, false</td>
</tr>
<tr>
<td>char</td>
<td>8</td>
<td>'a', 'B', '4', '/'</td>
</tr>
<tr>
<td>int</td>
<td>32 or 64</td>
<td>0, 1, 765, −6, 0xCAFE</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>0L, 1L, 1000000000000L</td>
</tr>
</tbody>
</table>

* on current architectures
† stored as 8 bits
Types and literals

<table>
<thead>
<tr>
<th>type</th>
<th>bits *</th>
<th>literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>1 †</td>
<td>true, false</td>
</tr>
<tr>
<td>char</td>
<td>8</td>
<td>'a', 'B', '4', '/'</td>
</tr>
<tr>
<td>int</td>
<td>32 or 64</td>
<td>0, 1, 765, −6, 0xCAFÉ</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>0L, 1L, 1000000000000L</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>0.0, 1.2, −0.765, −6e15</td>
</tr>
</tbody>
</table>

* on current architectures
† stored as 8 bits
## Types and literals

<table>
<thead>
<tr>
<th>type</th>
<th>bits *</th>
<th>literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>1†</td>
<td>true, false</td>
</tr>
<tr>
<td>char</td>
<td>8</td>
<td>'a', 'B', '4', '/'</td>
</tr>
<tr>
<td>int</td>
<td>32 or 64</td>
<td>0, 1, 765, −6, 0xCAFE</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>0L, 1L, 100000000000L</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>0.0, 1.2, −0.765, −6e15</td>
</tr>
<tr>
<td>string</td>
<td>varies</td>
<td>&quot;Hello, world!&quot;†</td>
</tr>
</tbody>
</table>

* on current architectures
† stored as 8 bits
‡ actually has type `const char[]`, but converts automatically to `string`
C++ provides built-in types:

- `<bool>`
- `(unsigned)` `<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++
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 *declaration* names an object
Objects

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

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

\T:int a;

\T:int b = 9;
\T:auto c = 'z'; // c is a char
\T:double x = 6.7;
\T:string s = "hello!";
\T:string t;
Declaration and initialization

\text{T:int a;}

\text{a:} \begin{array}{c}
\text{-2340024}
\end{array}

\text{T:int b = 9;}

\text{b:} \begin{array}{c}
\text{9}
\end{array}

\text{T:auto c = 'z';}

\text{c:} \begin{array}{c}
\text{'z'}
\end{array}

\text{T:double x = 6.7;}

\text{x:} \begin{array}{c}
\text{6.7}
\end{array}

\text{T:string s = "hello!";}

\text{s:} \begin{array}{c}
\text{"hello!"}
\end{array}

\text{T:string t;}

\text{t:} \begin{array}{c}
\text{"
\end{array}

\text{18}
Declaration and initialization

\T:int a;
\T:int b = 9;
Declaration and initialization

\T:int a;
\T:int b = 9;
\T:auto c = 'z';  // c is a char

a: -2340024
b: 9
c: ‘z’
Declaration and initialization

\T:int a;
\T:int b = 9;
\T:auto c = 'z';  \textit{\texttt{\textbackslash \texttt{c \textit{is a char}}}}
\T:double x = 6.7;
Declaration and initialization

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

\T: | a: | -2340024 | b: | 9 |
\T: | c: | ‘z’ | x: | 6.7 |
\T: | s: | 6 | “hello!” |
Declaration and initialization

\texttt{int a;}
a: \texttt{-2340024}

\texttt{int b = 9;}
b: \texttt{9}

\texttt{auto c = 'z'; \quad \text{// c is a char}}
c: \texttt{‘z’}

\texttt{double x = 6.7;}
x: \texttt{6.7}

\texttt{string s = "hello!";}
s: \texttt{6 "hello!"}

\texttt{string t;}
t: \texttt{0 ""}
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.

\[ \text{int } a = 7; \]
Assignment and increment

The value of a variable may change.

\begin{itemize}
\item \texttt{int a = 7;}
\item \texttt{a = 9;}
\end{itemize}
Assignment and increment

The value of a variable may change.

\begin{itemize}
\item \texttt{int a = 7;}
\item \texttt{a = 9;}
\end{itemize}
Assignment and increment

The value of a variable may change.

\[
\text{T:int } a = 7; \\
\text{a = 9;} \\
\text{a = a + a;}
\]
Assignment and increment

The value of a variable may change.

\[
\text{T:int } a = 7; \quad a = 7 \\
a = 9; \quad a = 9 \\
a = a + a; \quad a = 18
\]
Assignment and increment

The value of a variable may change.

\texttt{int }a\texttt{ = }7; \quad a: \quad 7

\texttt{a = 9;} \quad a: \quad 9

\texttt{a = a + a;} \quad a: \quad 18

\texttt{a += 2;}
Assignment and increment

The value of a variable may change.

\[ \text{T: int } a = 7; \]
\[ a = 9; \]
\[ a = a + a; \]
\[ a += 2; \]
Assignment and increment

The value of a variable may change.

\[
\begin{align*}
\text{T:} & \text{int } a = 7; \quad 7 \\
& a = 9; \quad 9 \\
& a = a + a; \quad 18 \\
& a += 2; \quad 20 \\
& ++a; 
\end{align*}
\]
Assignment and increment

The value of a variable may change.

\texttt{int a = 7; 7}
\texttt{a = 9; 9}
\texttt{a = a + a; 18}
\texttt{a += 2; 20}
\texttt{++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)

```cpp
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)

```cpp
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:
    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)

```c++
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:
    cout << " x: " << x << " c: " << c << " d: " << d << endl;
}
```

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

```cpp
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 `<int>`s is implemented by an integer add operation
  - `=` on `<int>`s 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 `<int>`s is implemented by an integer add operation
  - `=` on `<int>`s is implemented by a simple copy operation
  - C++ provides direct access to most of facilities provided by modern hardware
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