Names and Types
/ Example 1.1
#include <iostream>
#include <string>

using std::cout;
using std::endl;
using std::string;

int main() {
    string name = "Josh";
    cout << "Hello " << name << '!' << endl;
}

Chaining cout expressions

- Remember `<<` is a *binary* operator that returns the stream
  \[
  \text{cout} \ll \text{“Hello “} \ll \text{name} \ll \text{‘!’} \ll \text{endl};
  \]
  - *Do* \text{cout} \ll \text{“Hello “}
    - Print string, expression return \text{cout stream}
  - The do next pair \text{cout} \ll \text{name}
    - \text{name} is a string variable, print the string
  - Then do \text{cout} \ll \text{‘!’}
    - Single quotes, a single character
  - Finally \text{cout} \ll \text{endl}
Overloaded `<<` operator

- These three calls are to *three different functions* because of types
  - Print a constant string
  - Print a string
  - Print a character
The `endl` indicates you want the output to end the line and have the next output begin at the front of the next line.

- Does other things too (a flush) which we’ll discuss later.
3 ways to deal with std

- Three ways, one is **disallowed**
  - Merge all of std with the global namespace, using namespace std;
  - Indicate every time for each value the namespace it comes from
  - Declare up front only those particular elements you want to merge
Merging

- **using namespace std;**

- This essentially merges all the declarations in std into the global namespace.
  - **No std:: required anywhere**
  - **Possible points off your assignments if you do this!**
Full merge is bad

- This is the easy way, but it is fraught with problems:
  - What just got merged (you don’t know)?
  - When you indicated a variable, what namespace did it come from?
  - Affects everyone who includes your file
Mark every variable with std

- If you mark each one, you can differentiate what namespace it belongs to

```cpp
std::cout << "Hi mom" << std::endl;
```

- Allows for the same names from different namespaces
- The most general way to go
- Can get to be a pain
Merge only what you need

- You can get away with this

```cpp
#include<iostream>
using std::cout;
using std::endl;
cout << "Hi Mom" << endl;
```
What's wrong with the following code:

```cpp
#include<iostream>
using std::cout;
int main() {
    cout << "Nothing is wrong" << endl;
}
```

- It is doing a full merge.
- The `endl` isn't in scope.
- Nothing is wrong
- I don't know
// Example 1.2
#include <iostream>
#include <string>

using std::cin;       // console input stream
using std::cout;      // console output stream
using std::endl;      // end of line marker
using std::string;    // STL string package

int main() {
    cout << "What's your name:";
    string name;
    cin >> name;
    cout << "What's your age:";
    int age;
    cin >> age;
    cout << "Hello " << name << ", you are " << age << '!' << endl;
    // return 0;
}
Declaration

- Before you use a variable, you must declare it
  - At least say what type it will hold
    - Cannot change that variable’s type for the duration of the program
  - Could include an initial value
    - If you don’t, the class gets to decide the initial value
- Different than Python
Extraction Operator

- For `cin` (input stream) we have the extraction operator (`>>`)
  - Pulls a **typed value** from the console input up to
    - white space
    - end of line
    - error
Typed value and cin

- When you run the extraction operator, cin is **overloaded** to deal with the type of variable the value is going into:
  - If it is an int, only read digits
  - If it is a float, read digits, ‘.’, ‘E’
  - If it is a string, reads anything
- If it hits a problem (read a float into an int) it reads what it can and then errors out
Other things in this version

- We included the string header. We could do STL string operations, but we just declared a string.

- Return value commented out (not required).
Things to note

- `cout` expression doesn’t have an `endl`
  - We can `cin` from the same line

- We have two declares
  - Integer age and string name
  - Didn’t give initis, takes defaults
    - 0 for int, “” for string
    - Questionable for int, compiler dependent

- Two different ops for `>>` (type dependent)
Spacing

- Shouldn’t use more than 80 columns on a line for readability
- Below is acceptable (note indentation)

    ```cpp
    cout << "Hello " << name << "", 
    << 65 - age << " years to retirement"
    << endl;
    ```
Why should you initialize a variable before you use `cin` to fill it?

- `cin` only works if the variable is uninitialized.
- In case the `cin` fails, you can check if the variable is unchanged.
- There is no good reason to do so.
- I don't know
Fundamental Types!

```cpp
#include <iostream>
using std::cout;
using std::endl;
#include <limits>
using std::numeric_limits;

int main() {
    cout << "Size of bool:" << sizeof(bool) << endl << endl;
    cout << "Size of char:" << sizeof(char) << endl << endl;
    cout << "Size of short:" << sizeof(short) << endl;
    cout << "Smallest short:" << numeric_limits<short>::min() << endl;
    cout << "Largest short:" << numeric_limits<short>::max() << endl << endl;
    cout << "Size of int:" << sizeof(int) << endl;
    cout << "Smallest int:" << numeric_limits<int>::min() << endl;
    cout << "Largest int:" << numeric_limits<int>::max() << endl << endl;
    cout << "Size of long:" << sizeof(long) << endl;
    cout << "Smallest long:" << numeric_limits<long>::min() << endl;
    cout << "Largest long:" << numeric_limits<long>::max() << endl << endl;
    cout << "Size of long long:" << sizeof(long long) << endl;
    cout << "Size of float:" << sizeof(float) << endl;
    cout << "Smallest float:" << numeric_limits<float>::lowest() << endl;
    cout << "Largest float:" << numeric_limits<float>::max() << endl;
    cout << "Digits in matissa, float:" << numeric_limits<float>::digits << endl << endl;
    cout << "Size of double:" << sizeof(double) << endl;
    cout << "Smallest double:" << numeric_limits<double>::min() << endl;
    cout << "Largest double:" << numeric_limits<double>::max() << endl;
    cout << "Digits in matissa, double:" << numeric_limits<double>::digits << endl << endl;
    cout << "Size of long double:" << sizeof(long double) << endl;
    cout << "Smallest long double:" << numeric_limits<long double>::min() << endl;
    cout << "Largest long double:" << numeric_limits<long double>::max() << endl;
    cout << "Digits in matissa, long double:" << numeric_limits<long double>::digits << endl;
}
```
Lots of types and modifiers

- We have to get the types right in C++
  - Compiled language needs to select the correct overloaded operator at compile time
  - Provide aids to the programmer to control how information is moved about
Compiler is a program

- Three things
  - A compiler is another program. It translated code to something else (usually an assembly language)
    - It can make mistakes or have quirks
  - When you get down to blaming the compiler for your program’s errors you should probably call it a day
    - Likely it is you, not the compiler
  - Want to know more? Take CSE 450!
Details of type can depend

- The C++ standard does not fully detail the required size of a type
  - It sets minimums and maximums
  - The compiler programmers are free to exceed those if they choose
  - You should run code on your compiler to see
### g++ 7.2.0, Ubuntu 32-bit

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>1 byte</td>
<td>Boolean (0, empty/false, everything_elses / true)</td>
</tr>
<tr>
<td>char</td>
<td>1 byte</td>
<td>Hold a character</td>
</tr>
<tr>
<td>short (short int)</td>
<td>2 bytes</td>
<td>±32,768</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes (2^{32})</td>
<td>Basic integer, ~±2x10^9</td>
</tr>
<tr>
<td>long (long int, long long)</td>
<td>8 bytes (2^{64})</td>
<td>64 bit integers, ~±9x10^{18}</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>24 bits in significand</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>53 bits in significand</td>
</tr>
<tr>
<td>long double</td>
<td>16 bytes</td>
<td>64 bits in significand</td>
</tr>
</tbody>
</table>
To take advantage of the C++17 standard, you have to tell the compiler you are using code that is pursuant to that standard

- Visual Studio: set the profile
- In CLI, \texttt{g++ -std=c++17}
- Example 2.1 requires it
How do you run g++ in C++17 mode?

- `g++ -std=c++2017` ...
- `g++ -std=c++11` ...
- `g++ -std=c++17` ...
- `g++ c++1017` ...
Suggestions

- When in doubt:
  - Use `int` for an integer (if there is any chance of exceeding an `int`, specify an exact size like `int64_t` from `<cstdint>`)
  - Use `double` for a float
- Especially true for doubles as floating-point numbers introduce all kinds of round-off errors
- The more precision the better
// Example 2.2
#include <iostream>
using std::boolalpha;
using std::cout;
using std::endl;
using std::fixed;
#include <iomanip>
using std::setprecision;

int main() {
    // 4 different initializers. Part of the declaration! Type on left.
    short my_short;
    int my_int = 23;
    bool my_bool(1); // c++11
    double my_double = {3.1415926535897932}; // c++11

    // cout << boolalpha;        // set out stream state, print bools as strings
    cout << fixed << setprecision(6);
    cout << "Bool:" << my_bool << ", Int:" << my_int << ",int:" << my_short
         << ", Double:" << my_double << endl;

    cout << "my_short:" << my_short << endl;
    my_short = -1;
}
Initialize Variables

- C++, because of its legacy support and feature creep has many ways to do things
- One of them is initialization of a variable
  - Some subtleties here
  - Let’s look at basics
Variants

- No init (compiler dependent)
- Assign init
- Parenthesis init(11)
- Curly init {11}

- There are some subtleties here that are worth noting (lots more later)
Three types

▪ These are initializations because they are *declaring* a variable

▪ Direction initialization (both equivalent)
  ▪ `long my_long(my_int);`
  ▪ `int my_int = 123;`

▪ Initializer list (depends on type)
  ▪ `long another_long{1}`
C++ and efficiency

- Unlike in Python, in C++ we worry about efficiency
  - One of the main reasons to use C++
  - Can cause complications (but that is kind of the point)
- For efficiency’s sake, we want to avoid copies (because they are expensive)
What does = mean?

- Remember the context problem for C++
- The = (equal) sign means different things in different contexts
  ```cpp
  int my_int = 23; // initialization
  my_int = 123; // different op, assign
  ```
Which of the following statements are DECLARATIONS?

- `int x;`
- `int x = 3;`
- `int x(3);`
Which of the following statements are ASSIGNMENTS?

- `int x;`
- `int x = 3;`
- `int x(3);`
- `x = 3;`
Which of the following statements are INITIALIZATIONS?

• `int x;`
• `int x = 3;`
• `int x(3);`
• `x = 3;`
Expressions
// Example 2.3
#include <iostream>
using std::cout;
using std::endl;
using std::fixed;

int main() {
    int my_int = 9, an_int = 4;

    double my_double = 3.7, a_double = 5.8;

    cout << endl;
    cout << "int my_int: " << my_int << endl;
    cout << "int an_int: " << an_int << endl << endl;

    cout << "double my_double: " << my_double << endl;
    cout << "double a_double: " << a_double << endl << endl;

    cout << "*** Integer computations ***" << endl << endl;
    cout << "my_int + an_int: " << (my_int + an_int) << endl;
    cout << "my_int - an_int: " << (my_int - an_int) << endl;
    cout << "my_int * an_int: " << (my_int * an_int) << endl;
    cout << "my_int / an_int: (integer division!) " << (my_int / an_int) << endl;
    cout << "my_int % an_int: " << (my_int % an_int) << endl << endl;

    cout << "*** Compound computations ***" << endl << endl;
    cout << "my_int + an_int / 5:   " << (my_int + an_int / 5) << endl;
    cout << "(my_int + an_int) / 5: " << ((my_int + an_int) / 5) << endl;
    cout << "my_int / an_int * 5:   " << (my_int / an_int * 5) << endl;
    cout << "my_int / (an_int * 5): " << (my_int / (an_int * 5)) << endl;

    cout << "*** Real computations ***" << endl << endl;
    cout << "my_double + a_double: " << (my_double + a_double) << endl;
    cout << "my_double - a_double: " << (my_double - a_double) << endl;
    cout << "my_double * a_double: " << (my_double * a_double) << endl;
    cout << "my_double / a_double: " << (my_double / a_double) << endl;

    cout << "*** Mixed-type computations ***" << endl << endl;
    cout << fixed;
    cout << "my_int - my_double: " << (my_int - my_double) << endl;
    cout << "my_int / my_double: " << (my_int / my_double) << endl;

    cout << "*** Type conversions ***" << endl;
    cout << "static_cast<double>( my_int ): " << static_cast<double>(my_int) << endl;
    cout << "static_cast<int>( my_double ): " << static_cast<int>(my_double) << endl;
}

// hex and octal
int temp_int;
temp_int = 010; // leading 0, it's octal
    cout << "temp_int = 010 yields:" << temp_int << endl;
    temp_int = 0x10; // Leading '0x', it's hex
    cout << "temp_int = 0x10 yields:" << temp_int << endl;
Math operators

- Integers (all return integers)
  - Addition and subtraction: +, -
  - Multiplication: *
  - Division
    - / of two integers (returns an integer)
    - remainder: %

- Floating Point (all return floats)
  - Add, subtract, multiply, divide: +, -, *, /
Octal and Hex

- Pay attention to this

```cpp
int temp_int;
temp_int = 010; // leading 0, octal
cout << temp_int; // prints 8
temp_int = 0x10; // 0x means hex
cout << temp_int; // prints 16
```
Type Conversion

- Converts one type to another
  - e.g. convert an integer to a floating point
  - Often called a cast

- There are a number of cast operators

- Right now we’ll talk about `static_cast`
  - Requires the “cast to” type in `< >`
  - `static_cast<int>(1.789) -> 1`
  - No rounding!
Automatic Cast

- When does C++ do an auto cast:
  - The binary operator (overloaded) you requested does not exist (the combination of types doesn’t exist)
  - There is a conversion operator of one of those types that works for an op
    - C++ tries to apply conversions that maintain information
  - In mixed math, int / long are auto cast to float / double
**Integer Math**

```cpp
int int2 = 2, int3 = 3;
double float3 = 3;
cout << int2 / int3; // ??
cout << int3 / int2; // ??
cout << int2 / float3; // ??
cout << int2 % int3; // ??
cout << int3 % int2; // ??
```
If no precedence, left to right in pairs

- $1 + 2 + 3 + 4$
  - $(1 + 2) + 3 + 4$
    - Addition returns a result, 3
  - $(3 + 3) + 4$
    - Addition returns a result, 6
- $6 + 4$
  - Returns 10
Side Effects

// Example 2.4
#include <iostream>
using std::cout;
using std::endl;

int main() {
    int my_int, an_int;

    cout << "** assignment returns a value!" << endl;
    cout << "my_int = 15 returns:" << (my_int = 15) << endl;

    cout << endl << "*** Chained assignment expressions ***" << endl;
    an_int = my_int = 5;
    cout << "my_int: " << my_int << endl;
    cout << "an_int: " << an_int << endl << endl;

    cout << "*** Compound assignment expressions ***" << endl;
    my_int = 15;
    my_int += 2;
    cout << "Statements: my_int = 15 followed by my_int += 2;" << endl;
    cout << "my_int: " << my_int << endl << endl;

    cout << "*** Increment expressions ***" << endl;
    my_int = 20;
    cout << "my_int starts at 20" << endl;
    cout << "Value of ++my_int: " << ++my_int << endl;
    cout << "Value of my_int:   " << my_int << endl;
    cout << "Value of my_int++: " << my_int++ << endl;
    cout << "Value of my_int:   " << my_int << endl;

    cout << "*** Decrement expressions ***" << endl;
    my_int = 30;
    cout << "my_int starts at 30" << endl;
    cout << "Value of --my_int: " << --my_int << endl;
    cout << "Value of my_int:   " << my_int << endl;
    cout << "Value of my_int--: " << my_int-- << endl;
    cout << "Value of my_int:   " << my_int << endl;
}
Assignment Expressions

- Format: \textit{lvalue} = \textit{rvalue}
- \textit{rvalue} (right-hand-side of =) represents a value
- \textit{lvalue} (lhs of =) represents a memory location
- We are \textbf{copying} the value to the location
- Return value is a rvalue
Follow precedence rules

Example $x = 2 + 3 \times 5$
- Evaluate the expression $(2 + (3 \times 5))$: 17
- Change the value of $x$ to be 17
- Return the value 17

Example ($y$ has the value 2): $y = y + 3$
- Evaluate expression $(y + 3)$: 5
- Change the value of $y$ to be 5
- Return the value 5
Chaining

- is right associative
- Example: $x = y = 5$

Behavior
- Right associative $x = (y = 5)$
- Expression $y = 5$ returns value 5
- $x = 5$
Side-effect vs. return

- A function / operator can do **two things**
  - Perform some operation (write to output, change a variable’s value)
    - This is the **side-effect**
  - **Return value** after the operation
    - Return can be assigned, etc.
Seen this in `<<` operator

- `cout << whatever`
  - Side-effect, dump `whatever` to the `cout` stream
  - Return the stream (in this case `cout`)
- Allows for chaining
- `cout << 1 << 2`, pairs left to right
  - `cout << 1` -> returns `cout`
  - `cout << 2`
Shortcut: Increment

- Order (pre or post) matters. Side-effect the same, return value different

- Example: \( x = ++y; \)
  - Pre-increment, return **changed value**
    - \( y = y + 1; \)
    - \( x = y; \)

- Example: \( x = y++; \)
  - Post-increment, return original value
    - \( x = y; \)
    - \( x = y; \)
    - \( y = y + 1; \)
Other shortcuts

- Decrement: --
  - Example $y = x--$

- Compound assignment:
  - $y += x$ equivalent to $y = y + x$

- Others
  - $-=, *=, /=, %= $
What is the value of x?

```cpp
int x = 4;

int y = x++;

++x;
```

- 4
- 5
- 6
- I don't know
What is the value of `y`?

```c
int x = 4;

int y = x++;

++x;
```

- 4
- 5
- 6
- I don't know
Boolean and IO Operations
// Example 2.6
#include <iostream>
using std::boolalpha;
using std::cout;
using std::endl;

int main() {
    bool bool_true = true, bool_false = false;
    int first = 0, second = 0;

    cout << "bool_true:" << bool_true << ", bool_false:" << bool_false << endl;
    cout << "Turning on boolalpha" << boolalpha << endl;
    cout << "bool_true:" << bool_true << ", bool_false:" << bool_false << endl;

    cout << "std  &&, bool_true && bool_false:" << (bool_true && bool_false) << endl;
    cout << "alt and,  bool_true and bool_false:" << (bool_true and bool_false) << endl;

    cout << "std  ||, bool_true || bool_false:" << (bool_true || bool_false) << endl;
    cout << "alt or,  bool_true or bool_false:" << (bool_true or bool_false) << endl;

    cout << "Short circuiting" << endl;
    cout << "first:" << first << ", second:" << second << endl;
    (first = 100) || (second = 200);
    cout << "after  (first = 100) || (second = 200)" << endl;
    cout << "first:" << first << ", second:" << second << endl;
    first = 0;
    second = 0;
    cout << "first:" << first << ", second:" << second << endl;
    (first = 0) && (second = 200);
    cout << "first:" << first << ", second:" << second << endl;
}
Boolean Expressions

- Value: True or false
  - Remnant of C:
    - Integer value of 0 is equivalent to false
    - Nonzero integer value is equivalent to true
    - Both true and false are C++ terms
      - true == 1, false == 0
- Example expression: age < 40
  - Format: expression op expression
  - Result: 0, 1
Logical Operators

- Logical Operators
  - And: `&&`
  - Or: `||` (two vertical bar chars)
  - Not: `!`

- `(0 <= my_int) && (my_int <= 3)`
- `(0 <= my_int) || (my_int <= 3)`
- `!my_int`
### Truth Tables

| p     | q     | !p    | P && q | P || q |
|-------|-------|-------|--------|--------|
| True  | True  | False | True   | True   |
| True  | False | False | False  | True   |
| False | True  | True  | False  | True   |
| False | False | True  | False  | False  |
Alternative logical ops

- Turns out C++ does support `and`, `or`, and `not` as in Python
  - Your book doesn’t mention it
  - You are probably not in the C++ club if you do that
// Example 2.7
#include <iostream>
using std::boolalpha;
using std::cout;
using std::endl;

int main() {
    bool const bool_T = true, bool_F = false;

    int const my_int = 3, an_int = 8;

    cout << endl;
    cout << "bool bool_T: " << bool_T << endl;
    cout << "bool bool_F: " << bool_F << endl << endl;

    cout << "int my_int: " << my_int << endl;
    cout << "int an_int: " << an_int << endl << endl;

    cout << "my_int == an_int: " << (my_int == an_int) << endl;
    cout << "my_int != an_int: " << (my_int != an_int) << endl;

}
Relational Operators

- Less than: <
- Greater than: >
- Equal to: == (not the same as =)
- Not equal to: !=
- Less than or equal to: <=
- Greater than or equal to: >=
Examples

- If the value of integer `my_int` is 5, the value of the expression `my_int < 7` is true (1).
- If the value of char `my_char` is ‘A’, then the value of the expression `my_char == 'Q'` is false (0).
Be careful of floating point equality comparison, especially with zero

- e.g. `my_double == 0`
- Float arithmetic is approximate
- Use `!=` if you can
- If not, use a tolerance
  - Value +/- the tolerance
Want: \(0 \leq \text{my\_int} \leq 3\) (not like Python!)

Consider \text{my\_int} with value of 5

Left-associative: \((0 \leq \text{my\_int}) \leq 3\)

\((0 \leq \text{my\_int})\) is true, which has value 1

Therefore: \(1 \leq 3\)

Value of expression is true!

Solution: \((0 \leq \text{my\_int}) \land (\text{my\_int} \leq 3)\)
Three Things

- Assignments return a value!

- For each type
  - false: 0 / empty value
  - true: everything else

- Short circuiting
  - When it is “obvious” what a logical result will be, that result is returned and the compiler *ignores* the rest of the logical expression
What is the output?
&& short circuits on false

int first = 0, second = 0
(first = 0) && (second = 200);
cout << "First:" << first << ", Second:" << second << endl;

■ What is the output?
Your default should be to not use short-circuiting! It is confusing (especially to beginners) and often unintentional.
Intro to cout formatting

// Example 2.8
#include <iostream>
using std::boolalpha;
using std::cout;
using std::endl;
using std::fixed;
using std::left;
using std::right;
using std::scientific;

#include <iomanip>
using std::setfill;
using std::setprecision;
using std::setw;

int main() {
  double pi = 3.1415926535897932;
  bool bool_true = true;

  // float formatting
  // float format
Besides sending output (via `<<`) to `cout` or input (via `>>`) to `cin`, you can also set state in the stream:

- You set the stream to have a particular characteristic
- State persists in the stream until you reset it
  - Mostly
iostream, for output

- **fixed**: fixed points for floats
- **scientific**: use scientific notation
- **setprecision(prec)**: set the decimal points (with rounding) for floats
  (#include<iomanip>)
- **boolalpha/noboolalpha**: Show true or false for Booleans (0 or 1 otherwise)
More iostream, for output

- **left, right**: Align output to the left or right (left or right justified)
- **showpoint, noshowpoint**: Always use a decimal point on output vs only have a decimal point when there is a fractional part
\texttt{iomanip}, \textit{for output}

- \texttt{setw(space\_cnt)}
  - Min width the output occupies
  - Does \textbf{not} set state, must be set for \textit{every} field output
  - Wider if output is wider

- \texttt{setfill(char)}
  - In a wider field, fill with char
  - Space is default
- We haven’t seen if statements yet, but here is one anyway

  int x = 5;
  if (x = 1)
      dosomething;

- That compiles fine, is always true, and probably not what you wanted (==)