Variables and Types

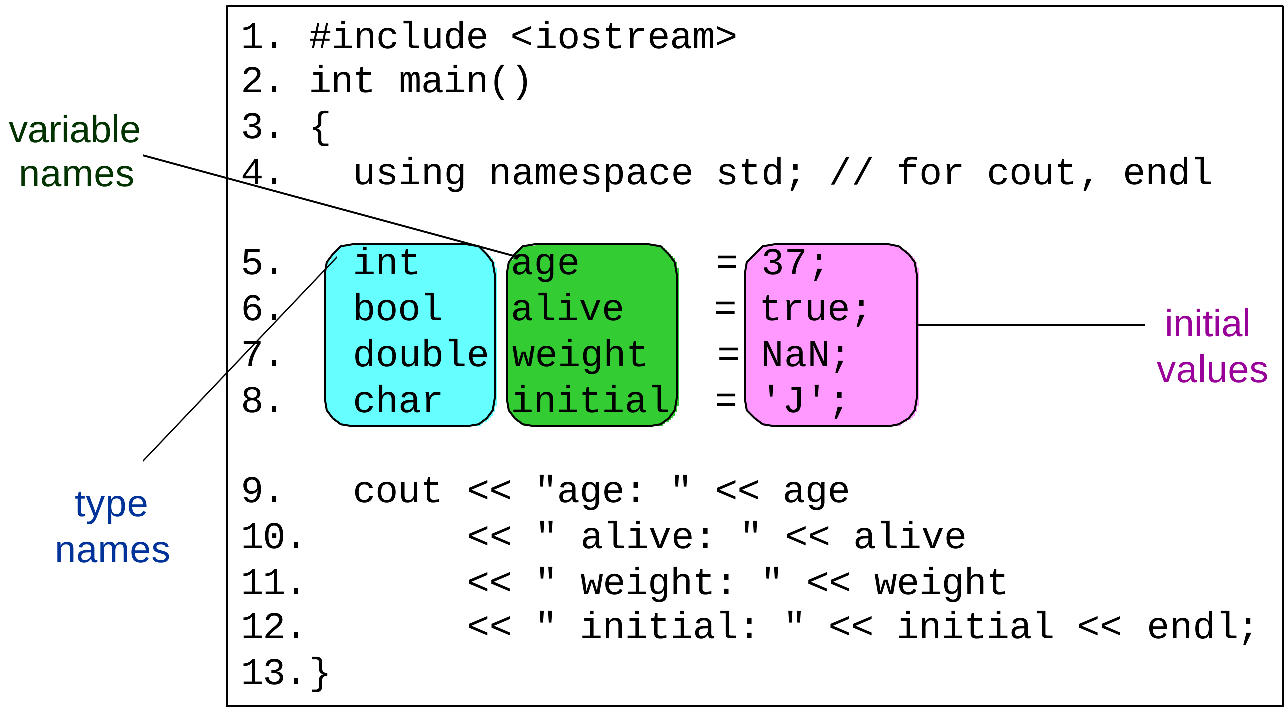
Basic Types

- Here are the most commonly used basic types:
  - integers: `int`
  - real numbers: `double`
  - characters: `char`
  - booleans: `bool`

1. Basic types are those types that are built into the language, as opposed to ones like string and complex, which are defined in libraries.
2. There are a number of different types that can be used to represent integers, reals and characters, but these are the most common.

Every Variable has a Type

- type, name, and optionally initial value:



```cpp
#include <iostream>
int main()
{
  using namespace std; // for cout, endl

  int    age     = 37;
  bool   alive   = true;
  double weight  = NaN;
  char   initial = 'J';

  cout << "age: " << age
       << " alive: " << alive
       << " weight: " << weight
       << " initial: " << initial << endl;
}
```

1. age, alive, weight, and initial are variables that are local to the block they are declared in, in this case, local to the function main. That is, at the end of the main function, they will disappear.
2. Don't worry about anything outside the colored boxes; it is provided to make the program complete as shown; we will return to output operators and namespaces in more detail later.
3. Within the main function, you can refer to these variables, as we do here to print them out.
4. Variables do not have to be initialized; if they are not they will have undefined values until you assign to them. (This is only true for the built-in types, like int, bool, double, char).
5. NaN is a special value meaning "Not a Number". It is part of the IEEE floating-point number specification.
6. There are a few other special numbers like this, e.g., Inf.
7. If you declare a variable without specifying an initial value, the value is undefined until you assign to the variable (this is really only true for built-in types; we will see constructors in XXX).

Variable Names

- Names of variables must start with an alphabetic letter
- They can contain alphabetic letters, digits, and underscores
- A common convention is to start local variables with a lower-case letter

1. Variable names can start with an underscore, but this is unadvisable, as certain names starting with an underscore are reserved for the compiler's use.
2. Names with two consecutive underscores anywhere in the name are reserved for the compiler.
3. It is less important which convention you use than that you be consistent.

Variable Names: examples

Legal names

```
f00
sTrAnGe
somewhatLessStrange
find_first_not_in
Vector
vector
_foo
```

Illegal names

```
2BeOrNot2Be
no-lisp-style-vars
not__good
no_F*&%_way
auto
export
no$dec$stuff$either
```

1. f00 is ill-advised because it is difficult to distinguish zeros from capital Os.
2. sTrAnGe capitalization will confuse the readers of your code.
3. mixed caps, or cuddly caps, are common: each word is capitalized (exccept the first, for variables), and underscores are not used.
4. Another convention (used in the standard library) is to separate words with underscores.
5. Vector is a poor name for a variable, as it is capitalized. Initial capitals are usually reserved for types.
6. vector is an even worse name for a variable, as vector is a name in the standard library.
7. Eventually, you should have a passing familiarity with the names in the standard library, to avoid using those names for your variables.
8. _foo is a legal name for a variable, but leading underscores are not generally recommended.
9. Names cannot star with a digit, they cannot contain hyphens, they cannot contain double underscores, they cannot contain non-alphanumeric characters other than underscore.
10. auto and export are keywords.

Literals

- Each type has literals:
  - bool: `true, false`
  - int: `0, -999, 12345, +100`
  - char: `'a', 'x', '?', '\''`
  - double: `32.4, 3e6, 3e-5, 2.5e9`

1. true and false are keywords; you cannot (at least should not) define them yourself.
2. Older compilers do not have the bool type built-in, but this is a detail you can ignore.
3. Integers can be positive or negative; the + sign for positive integers is optional.
4. The range of integers that can be stored varies from compiler/platform to compiler/platform. Most compilers for non-embedded systems have integers that are at least 32 bits, which is enough for most purposes.
5. You can find the largest and smallest integers (or any other integral type, like char, short, unsigned, etc), using the numeric_limits facility, e.g., numeric_limits<int>::max() and numeric_limits<int>::min().
6. Character literals are single-quoted.
7. You can use normal or scientific notation; with the latter, the e can be lower or upper case.

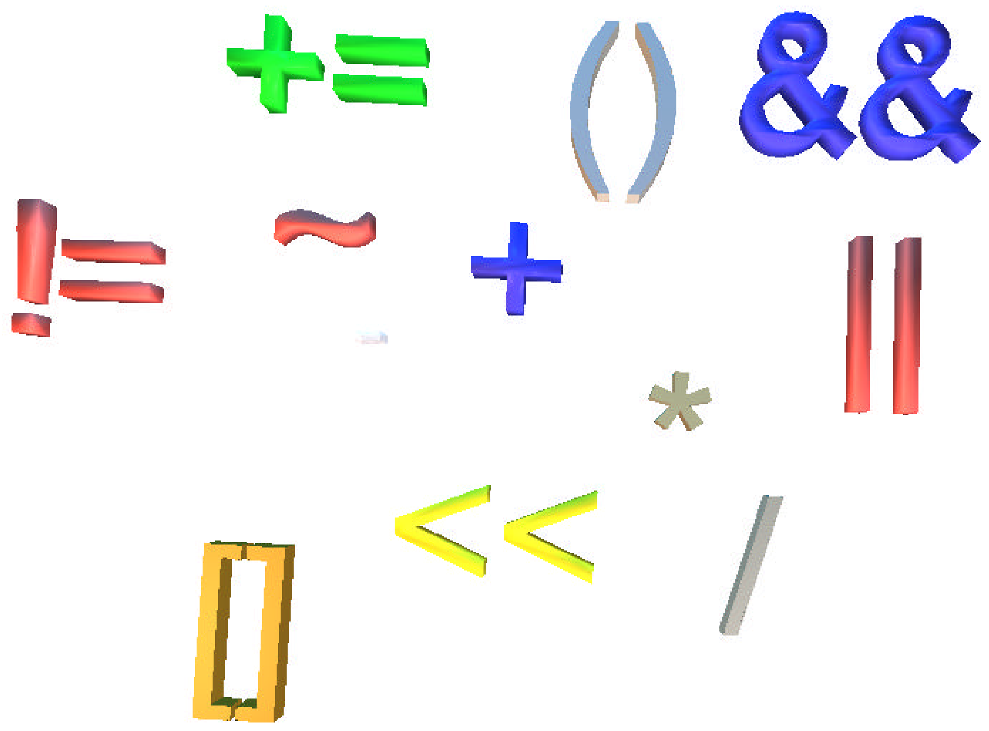
Expressions

- There are various operators that allow you to use the objects; many will be familiar from mathematics

```
double d = 2 * 3 + 4.5;
double dd = 1.0 – 2.3;
int one = 1;
int two = 2;
int three = one + two;
```

1. There are many many operators in C++, and we will talk about various ones at various times throughout the course.
2. Operators are special symbols that apply to the objects around them.
3. There are prefix operators, like negation: -d is the object d with the operation negation applied to it.
4. There are postfix operators like increment: d++ means increment d by one
5. d– means decrement d by one
6. You will see that most operators have corresponding assignment variants:
7. you can say a + b, which makes a new object that is the sum of the two, or you can say a+=b, which means
"add b to a".

Operators



1. The usual arithmetic operations are available: +, -, *, / for addition, subtraction, multiplication and division.
2. Division with integers drops any remainder: 7 / 2 will become 3.
3. If you want to make sure the operations are done with real numbers (not integers), make sure that the first operand is a double.
4. For most of the arithmetic operations, there is a corresponding assignment-operation:
5. a += b means a = a + b
6. a -= b means a = a – b
7. a*=b means a = a * b
8. a/=b means a = a / b

Operator Precedence

- Just like in mathematics, some operators hug their arguments more tightly than others

```
1. a + b * c

   a + (b*c)

2. cout << a+b

   cout << (a+b)

3. done = tired || injured && ! wizard

   done = ( tired || (injured && (! wizard) ) )
```

1. You can find lists giving the precedence of the various operators in any good C++ book, such as Stroustrup's 3rd edition.
2. The precedence of operators in C++ is the same as it is in C.
3. You can always add parentheses to clarify precedence. If there is any doubt in your mind as to the precedence in the code you write, don't look it up, add parentheses. Use parentheses if there is any chance of confusion to yourself (later) or someone else reading the code.
4. Assignment has very low precedence, so you never have to parenthesize the expression on the left of an assignment. (To be accurate, the comma operator has lower precedence than assignment, but it is rarely used on the right-hand side of an assignment). That is, "a = b , c + 2" means "(a = b), c+2", which assigns b to a, and has the final value c+2. This is a minor detail, don't worry about it if it is confusing you.
5. Operators can be redefined for user-defined types, but their precedence does not change.
6. Here I am assuming that a, b and c are integers or doubles, and done, tired, injured and wizard are booleans.

Operator Associativity

- Associativity specifies which direction the operators are evaluated in:

$$\underbrace{\underbrace{\underbrace{\underbrace{a + b} + c} + d} + e}$$

left-associative

right-associative

$$\underbrace{a = \underbrace{b = \underbrace{c = \underbrace{d = e}}}}$$

1. Most operators work left-to-right.
2. The first expression is equivalent to ((((a+b) + c) + d) + e)
3. The assignment operators (=, +=, -=, etc) work from right to left.
4. The second expression is equivalent to a = (b = (c = (d = e)))
5. That is, e is assigned to d, then the result of this assignment (which is the new value of d and is the same as the value of e), is assigned to c, and so on, making all of the variables equal to e.

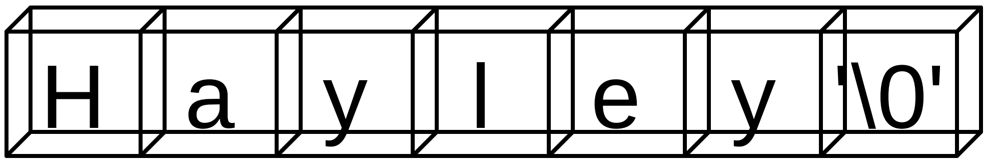
Implicit Conversions

- There are various places where C++ will *convert* one type to another for you
  - `char` to `int`
  - `int` to `long`
  - `float` to `double`
  - `int` to `float`
  - numeric to `bool`
  - `char*` to `string`
- These conversions are usually natural

<Picture of caterpillar to butterfly>

1. "Natural", of course, is a subjective thing, but most of the arithmetic conversions will seem natural to people as they behave like in mathematics.
2. Characters have their ascii value; if you assign the character '3' to an int, you will get 51, not 3. You don't really need to know the ascii table for most purposes, but it is convenient to know that '0' is 48, 'A' is 64, and 'a' is 96, and that the digits, the lower case letters and the uppercase letters are consecutive. This makes it easy to do arithmetic with characters, and to iterate through the letters or digits.
3. When people design libraries like the string library, they must be very careful to make sure that they only allow implicit conversions when it is not likely to cause confusion. For this reason, there is an implicit conversion from a character array to a string, but not vice versa. To convert from a string s to a character array (or pointer), you say s.c_str().
4. Any non-zero numeric value can be converted implicitly to a bool, and becomes true; 0 can be converted implicitly to false.
5. Technically, char* to string is slightly different; it is what is called a user-defined conversion (specified by a piece of C++ code) as opposed to a predefined conversion, which is built in to the compiler.
6. You can say, for example,

Strings



1. This is a short section, not intended to cover C++ strings fully, but to give you enough of any idea of how they work to be getting on with.

Strings and character arrays

- In C, there is no string type
- Character arrays are used instead
- Null-terminated
- In C++ you can still use character arrays
- But the standard `string` class has many advantages
- `string` is part of the standard library, not a built-in type

1. Manipulating text is painful in C because there is no built-in string type, as there is in many other languages. Instead, one uses null-terminated character arrays, and library functions like strcmp, strcat, strlen, etc. These require the user to manage memory, and are much lower-level than one would want.
2. The string class is part of the ANSI standard C++ library. For the most part, you needn't be concerned with what is in the standard library and what is part of the language --- C++ is designed to allow the creation of new types that behave almost exactly like the built-in ones. complex and string are two examples of such types.
3. A null-terminated string is a sequence of characters in memory whose end is specified by a character with the ascii value 0 (called a null character, don't confuse this with the space character). This has the advantage that the length of the string doesn't have to be stored explicitly (saving space), but also some disadvantages: you have to traverse the entire string to find out how long it is, and you cannot have null characters embedded in the string.

Example using strings

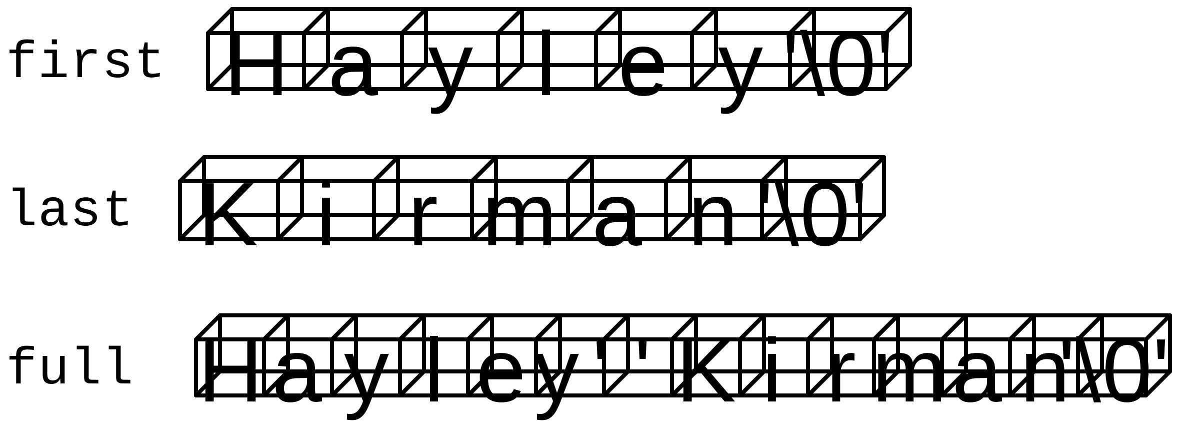
- This program reads lines from standard input and prints them out, numbered:

```
#include <iostream>
#include <string>
int main()
{
  using namespace std;
  for (int i = 1; ; ++i)
  {
    string s;
    getline (cin, s); // read a line
    if (!cin)         // end of file
      break;          // exit loop
    cout << i << ": " << s << endl;
  }
}
```

1. Don't worry if there are parts of this that you don't understand --- we will return to things like for loops and namespaces later in the course.
2. <iostream> is needed for getline, cin, cout and endl.
3. <string> is needed for the string class.
4. Everything in the standard library (including getline, cin, cout, endl, and string) is in the namespace std.
5. In general, if you can, avoid importing the entire namespace at the top level (it's ok inside a function), especially in headers.
6. The stream library was designed long before the string class was, so istream knows nothing about strings. The string library therefore provides a global function getline that takes an istream and a string, and reads one line from the stream, putting it in the string.

String concatenation

```
string first ("Hayley");
string last ("Kirman");
string full = first + " " + last;
```



1. The string first contains "Hayley"; the string last contains "Kirman".
2. Strings can be concatenated with +.
3. You can use (double-quoted) string literals, but the first item must be a string; we'll see why when we talk about operator overloading.
4. The dotted box indicates that this is a code fragment, not a complete program or file.

Creating Strings

- You can make an empty string
- You can make a string from a string literal
- You can make a copy of a string

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

int main()
{
  using namespace std;
  string empty;
  string a ("any text you like");
  string b (a);
  cout << b << endl;
}
```

1. To create a variable of type string, just specify the type and the name you want to give the variable. We will cover what names are legal and what are not in the next chapter.

Output

- You can write strings to streams like any other object
- You may need to get a character pointer corresponding to the string; use `c_str()`

```
string a ("Fred Flintstone");
cout << a << endl;
string filename ("foo.txt");
FILE *f = fopen (filename.c_str());
```

1. Many software components have interfaces that expect null-terminated character arrays.
2. Remember that what you are using here is really std::string, std::cout, std::endl.
3. We haven't seen the syntax for member functions yet --- if the last line here confuses you, don't worry about it.