Initialization

A variable is a named "chunk" of memory which contains a value, while a value is **a set of bits, interpreted according to its type**. Initialization provides a value **when a variable is created**.

Here are three ways to initialize a variable:

```
int a{42};    // uniform initialization
int b(35.5);  // direct initialization
int c = 4;    // legacy initialization
```

- Starting with C++11, **uniform**, **universal** or **list initialization** is the preferred way to initialize most variables. This form of initialization is value-preserving, like initialization in Java and C#. Attempts to use an initializer that would lose information (called a **narrowing conversion**), are rejected.
- **Direct initialization** uses parentheses, not braces, surrounding the initializer. Direct initialization permits **narrowing conversions**, where the initializer is implicitly **truncated** if it is too large. In the example above, the initializer `35.5` is truncated to the `int` value `35`. Direct initialization allows you to supply multiple initializers which is appropriate for many class types.
- **Legacy**) initialization is inherited from C. Like direct initialization, both widening and narrowing conversions are allowed.

What happens when variables are **not** initialized? In Java, C# and Python, **they can't be used**. (This is called the **definite assignment** rule. In C++, they **may be used**, according to these rules.

- **Primitive local** variables, which are not initialized, are **undefined**. Using such a variable is **undefined behavior** but it is not a syntax error, as in Java/C#. (Primitive variables are the built-in types like `int`, `double`, `char` and `bool`.)
- **Library variables** (such as `string`) are **automatically** initialized by implicitly calling their constructors (unlike Java).
- **Global** primitive variables are automatically initialized to `0`.