(NOTE: Examples used are quite silly and hypothetical `->` done so intentionally to give very simple illustrations.)

- What is *initializer list* (or *initialization list*) and why we should use it.
  - Suppose we have an existing **`Int`** data type:

```
class Int
{
public:
   Int();
   Int(int i);
   Int& operator=(const Int& rhs);
private:
   int data;
};
```

and we want to develop a new **`Date`** data type that uses the **`Int`** data type:

```
class Date
{
public:
   Date();
   Date(const Date& src);
   Date(const Int& d, const Int& m, const Int& y);
private:
   Int day, month, year;
};
```

  - We can implement the **`Date`** constructors as follows:

```
Date::Date()
{
   day = 1;
   month = 1;
   year = 1900;
}

Date::Date(const Date& src)
{
   day = src.day;
   month = src.month;
   year = src.year;
}

Date::Date(const Int& d, const Int& m, const Int& y)
{
   day = d;
   month = m;
   year = y;
}
```

  - But it is more efficient to use the *initializer list syntax* when implementing the constructors as shown below:

```
Date::Date() : day(1), month(1), year(1900) { }

Date::Date(const Date& src) : day(src.day), month(src.month), year(src.year) { }

Date::Date(Int d, Int m, Int y) : day(d), month(m), year(y) { }
```

This is because the initializer list is executed at the very beginning of the construction of an object, *before the body of the constructor is entered*, and values (of data members) in the object (being constructed) are initialized in the process.

The first implementation (which assigns values to **`day`**, **`month`** and **`year`** instead of initializing them) is less

efficient because initialization (of **day**, **month** and **year**) is still performed *implicitly*:

- In **Date()** and **Date(const Date&)**, constructor **Int()** is called 3 times to initialize **day**, **month** and **year**.
- In **Date(const Int&, const Int&, const Int&)**, constructor **Int(int)** is called 3 times to initialize **day**, **month** and **year**.

In each case, values initialized in **day**, **month** and **year** are promptly replaced when the assignment statements in the body are executed. Thus, values are (unnecessarily) stored *twice* into **day**, **month** and **year**:

- Once through *implicit initialization*.
- Once through *assignments in the constructor body*.

- To illustrate, if we compile and run the following code:

```
#include <iostream>

class Int
{
public:
   Int();
   Int(int i);
   Int& operator=(const Int& rhs);
private:
   int data;
};

Int::Int()
{
   std::cout << "in Int::Int()..." << std::endl;
   data = 0;
}

Int::Int(int i)
{
   std::cout << "in Int::Int(int)..." << std::endl;
   data = i;
}

Int& Int::operator=(const Int& rhs)
{
   std::cout << "in Int::operator=(const Int&)..." << std::endl;
   if (this != &rhs)
      data = rhs.data;
   return *this;
}

class Date
{
public:
   Date();
   Date(const Date& src);
   Date(const Int& d, const Int& m, const Int& y);
private:
   Int day;
   Int month;
   Int year;
};

Date::Date()
{
   std::cout << "in Date::Date()..." << std::endl;
   day = 1;
   month = 1;
   year = 1900;
}
```

```
Date::Date(const Date& src)
{
   std::cout << "in Date::Date(const Date&)..." << std::endl;
   day = src.day;
   month = src.month;
   year = src.year;
}

Date:: Date(const Int& d, const Int& m, const Int& y)
{
   std::cout << "in Date::Date(const Int&, const Int&, const Int&)..." << std::endl;
   day = d;
   month = m;
   year = y;
}

int main()
{
   Date defDate, dDayDate(6,6,1944), dDayDate2 = dDayDate;
   return 0;
}
```

we will get 27 messages (each corresponding to a function call):

```
in Int::Int()...
in Int::Int()...
in Int::Int()...
in Date::Date()...
in Int::Int(int)...
in Int::operator=(const Int&)...
in Int::Int(int)...
in Int::operator=(const Int&)...
in Int::Int(int)...
in Int::operator=(const Int&)...
in Int::Int(int)...
in Int::Int(int)...
in Int::Int(int)...
in Int::Int()...
in Int::Int()...
in Int::Int()...
in Date::Date(const Int&, const Int&, const Int&)...
in Int::operator=(const Int&)...
in Int::operator=(const Int&)...
in Int::operator=(const Int&)...
in Int::Int()...
in Int::Int()...
in Int::Int()...
in Date::Date(const Date&)...
in Int::operator=(const Int&)...
in Int::operator=(const Int&)...
in Int::operator=(const Int&)...
```

- But if we re-write the implementation for **`Date`**'s constructors as follows:

  (NOTE: **`Int`**'s constructors should be similarly re-written although that won't change the ensuing outcome.)

```
Date::Date() : day(1), month(1), year(1900)
{ std::cout << "in Date::Date()..." << std::endl; }

Date::Date(const Date& src) : day(src.day), month(src.month), year(src.year)
{ std::cout << "in Date::Date(const Date&)..." << std::endl; }

Date:: Date(const Int& d, const Int& m, const Int& y) : day(d), month(m), year(y)
{ std::cout << "in Date::Date(const Int&, const Int&, const Int&)..." << std::endl; }
```

  we will get only 9 messages:

```
in Int::Int(int)...
in Int::Int(int)...
in Int::Int(int)...
in Date::Date()...
in Int::Int(int)...
in Int::Int(int)...
in Int::Int(int)...
in Date::Date(const Int&, const Int&, const Int&)...
in Date::Date(const Date&)...
```

- A situation where the use of initialization lists in constructors is *mandatory* (*i.e.*, a must).
  - If we have *constants* or *references* in our class, they can only be initialized using an initialization list.
  - For example, this code fragment will cause an error:

```
class RareType
{
public:
   RareType(double& r);
private:
   const int MAX_SIZE;
   double& ref;
};

RareType::RareType(double& r)
{
   MAX_SIZE = 100;
   ref = r;
}
```

  but this won't:

```
class RareType
{
public:
   RareType(double& r);
private:
   const int MAX_SIZE;
   double& ref;
};

RareType::RareType(double& r) : MAX_SIZE(100), ref(r) { }
```

- In summary:
  - When implementing constructors, use the initializer list syntax *wherever possible*.
    - (Unless there are other special considerations that dictate otherwise.)
- Caveat:
  - The initializer list syntax can only be used in *constructors*.
  - When using the initializer list syntax in the constructors of a class with multiple fields (data members), the names of the fields being initialized should appear *in the order in which they are declared in the class*. (The GNU C++ compiler will typically give warning messages if the field names appear out of order.)
