2 Parameter Passing Efficiency

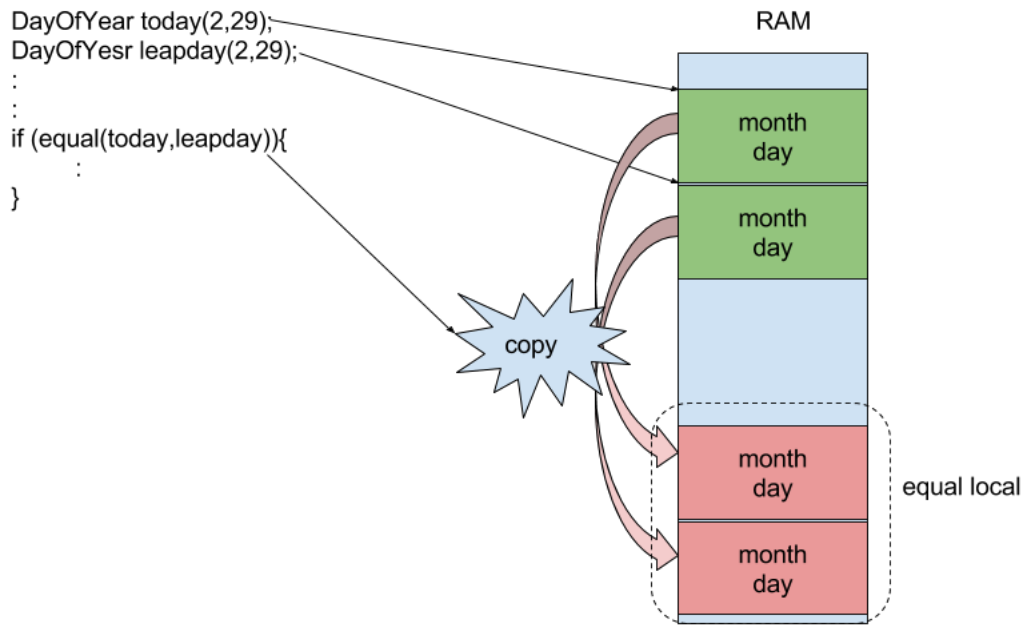
How data move in the code below?

```
// friend
bool equal(DayOfYear date1, DayOfYear date2)
{
    return (date1.month == date2.month &&
            date1.day == date2.day );
}

// main function
int main(int argc, const char * argv[]) {
    DayOfYear today(2,29);
    DayOfYear leapday(2, 29);

    if (equal(today,leapday)){
        std::cout << "Today is leap day\n";
    }
    today.output();

    return 0;
}
```



- A call-by-value parameter less efficient than a call-by-reference parameter
  - The parameter is a local variable initialized to the value of the argument
    - This results in two copies of the argument
- A call-by-reference parameter is more efficient
  - The parameter is a placeholder replaced by the argument
    - There is only one copy of the argument
- We used call-by-reference in order to return more than one variables from functions.
  - We can use a container (class) that contains multiple variables and return it with a return statement.

2.1 Class Parameters

- It can be much more efficient to use call-by-reference parameters when the parameter is of a class type
- When using a call-by-reference parameter
  - If the function **does not change** the value of the parameter, **mark** the parameter so **the compiler knows it should not be changed**
- **const Parameter Modifier** to mark a call-by-reference parameter so it cannot be changed:
  - Use the modifier **const** before the parameter type
  - The parameter becomes a **constant parameter**
  - **const** used in the function declaration **and** definition

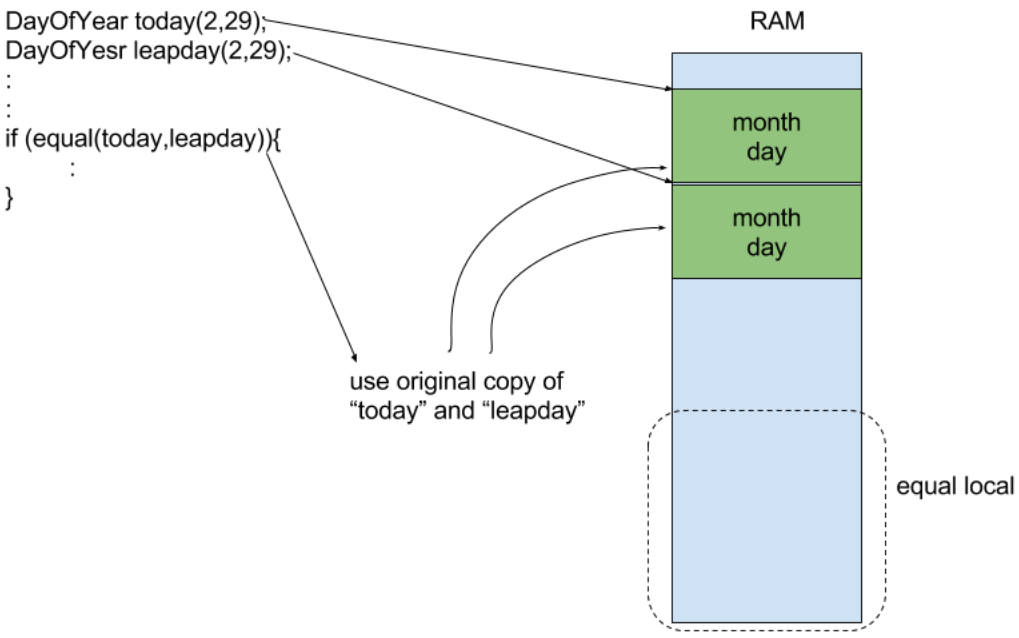
2.1.1 const Parameter Example

- A function declaration with constant parameters

```
friend bool equal(const DayOfYear& date1, const DayOfYear& date2);
```

- A function definition with constant parameters

```
// friend
bool equal(const DayOfYear& date1, const DayOfYear& date2)
{
    return (date1.month == date2.month &&
            date1.day == date2.day );
}
```



compare **sample32.cpp** and **sample33.cpp**

- When a function has a constant parameter, the compiler will make certain the parameter cannot be changed by the function

2.1.2 const And Accessor Functions

- Will the compiler accept an accessor function call from the constant parameter?

```
// friend
bool equal(const DayOfYear& date1, const DayOfYear& date2)
{
    date1.output( );
    date2.output( );
    return (date1.month == date2.month &&
            date1.day == date2.day );
}
```

- The compiler will **not** accept this code
  - There is **no guarantee** that **output** will not change the value of the parameter

2.2 const Modifies Functions

- If a constant parameter makes a member function call...
  - The member function called must be marked so the compiler knows it will not change the parameter
  - **const** is used to mark functions that will not change the value of an object
  - **const** is used in the function declaration and the function definition

2.3 Function Declarations With const

- To declare a function that will not change the value of any member variables:
  - Use **const** after the parameter list and just before the semicolon

```
class DayOfYear
{
public:
    :
        :
    void output( ) const;
        :
```

    - "**output()**" is Read-Only function.

- To define a function that will not change the value of any member variables:
  - Use const in the same location as the function declaration

```
void DayOfYear::output() const
{
    std::cout << "month = " << month
    <<  ",  day = " << day
    << std::endl;
}
```

- Now that output is declared and defined using the **const** modifier, the compiler will accept this code

```
// friend
bool equal(const DayOfYear& date1, const DayOfYear& date2)
{
    date1.output( );
    date2.output( );
    return (date1.month == date2.month &&
            date1.day == date2.day );
}
```

2.4 use of const

- Using **const** to modify parameters of class types improves program efficiency
  - **const** is typed in front of the parameter's type
- Member functions called by constant parameters must also use **const** to let the compiler know they do not change the value of the parameter
  - **const** is typed following the parameter list in the declaration and definition
- Once a parameter is modified by using **const** to make it a constant parameter
  - Any member functions that are called by the parameter must also be modified using **const** to tell the compiler they will not change the parameter
  - It is a good idea to modify, with **const**, every member function that does not change a member variable