ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Classes
Constructors and other tools

Constructors : Intro

- Constructors defined like any member function, except:
  1. Must have same name as class
  2. Cannot return a value - not even void!

No-arg constructor

2-arg constructor

```
//Class definition
class DayOfYear
 {
public:
  DayOfYear();
  DayOfYear(int, int);
  void input();
private:
  int month;
  int day;
};
```

The 2nd constructor is *overloaded* to take 2 arguments

```
//Class implementation
DayOfYear::DayOfYear()
{
month = 1;
day = 1;
}

DayOfYear::DayOfYear(int theMonth,
               int theDay)
{
month = theMonth;
day = theDay;
}

void DayofYear::input()
{

...
}
```

When are constructors executed

- As soon as an object has been created
- It may be created statically
  - **`DayOfYear birthday;`**
    `// Constructor with no arguments called`
  - **`DayOfYear birthday(3, 31);`**
    `// Constructor with 2 arguments called`
- Or created dynamically
  - we will come to this later
- Or, an explicit call to constructor can be made after object already exists to 're-initialise' (much less usual)
  - Such a call returns "anonymous object" which can then be assigned

```
DayOfYear holiday(7, 4);

holiday = DayOfYear(5, 5);
```

1. Two argument constructor called implicitly

2. Explicit constructor call returns new "anonymous object"

3. Assigned back to current object

Constructor implementation

```
DayOfYear::DayOfYear(int mnthval, int dayval)
{
   month = mnthval;
   day = dayval;
}
```

But be careful, we could not initialise *array* member variables so simply

- Alternative implementation uses member-wise initialisation

```
DayOfYear::DayOfYear(int monthValue, int dayValue)
                     : month(monthValue), day(dayValue)
{ }
```

- Second line called "Initialization Section”
- Body of method can be left empty
- This is the preferable implementation version

- Remember additional purposes of constructors
  - Not just to initialize data, may also validate the data!
    - Ensure only appropriate data is assigned to private member variables
    - Powerful OOP principle

Default Constructor

- Defined as: constructor with no arguments
- Does the compiler generate such a constructor automatically?
  - Yes & No
  - If no constructors AT ALL are supplied by programmer → Yes
    - We don't need to code it, but visual C++ implicitly supplies an empty constructor which does nothing
  - If any other constructors (with arguments) is supplied → No
    - Now Visual C++ assumes programmer is taking care of constructors, and if a default isn't defined, its not wanted
- If no default constructor exists:
  - Cannot declare: `MyClass myObject;`

    With no initializers
- So rule is:

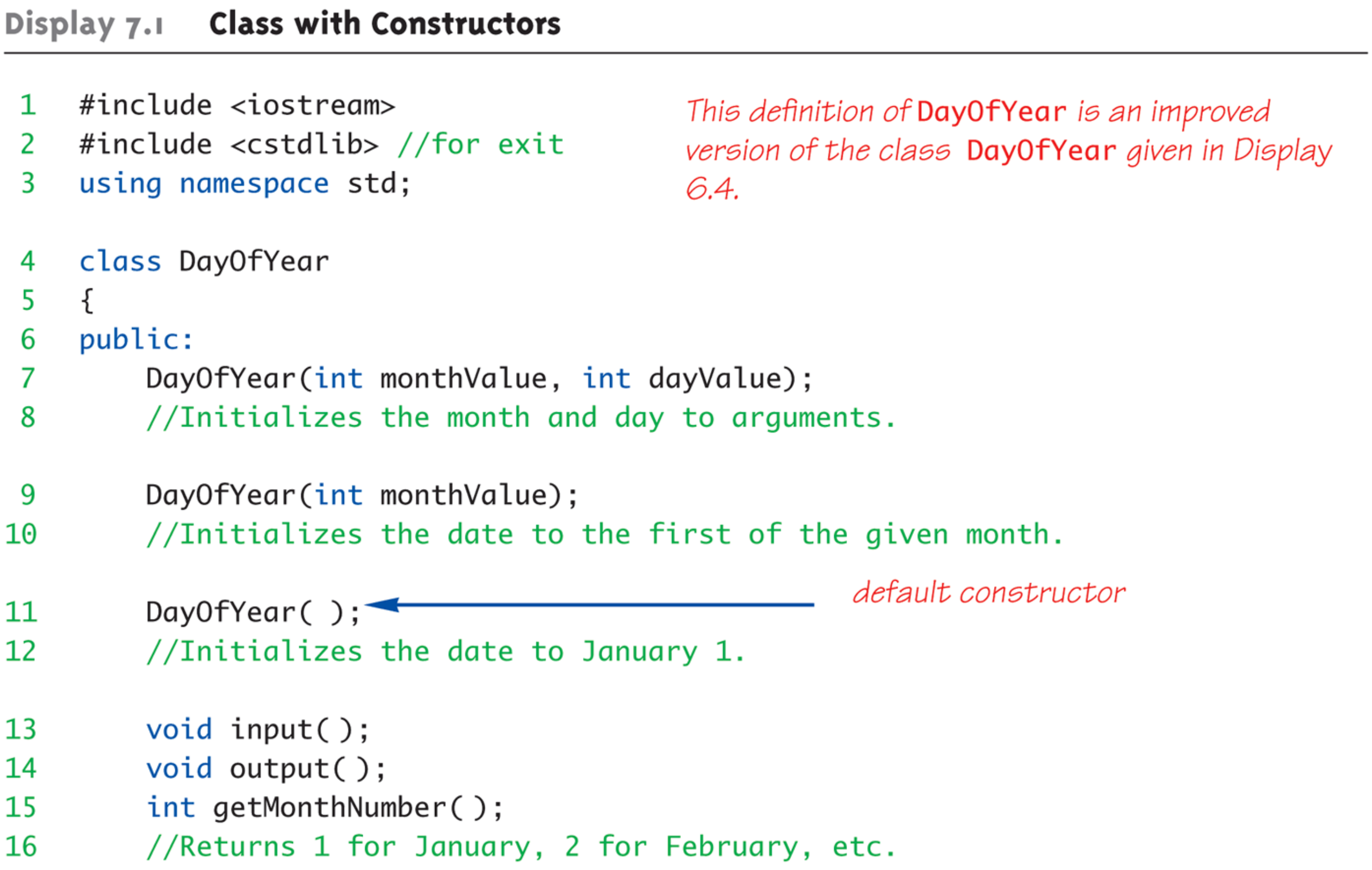
  if supplying any constructors, make sure to supply a default one unless we really don't want to allow objects with no user initialisation

Constructors - more

- Constructors with no arguments can be confusing
- Standard functions with no arguments:
  - Called with syntax: `callMyFunction();` Including empty parentheses
- But *object declarations* expecting to use a default constructor do not use parantheses:
  - `DayOfYear date1;` ✓ Like this
  - `DayOfYear date();` × This is wrong
    - Why is this wrong?
      - Compiler sees a function declaration/prototype!
      - Yes! Look closely!

`date()` is a stand-alone function which returns a `DayOfYear` object

Class with Constructors Example:
Display 7.1 Class with Constructors (1 of 3)



Display 7.1 **Class with Constructors**

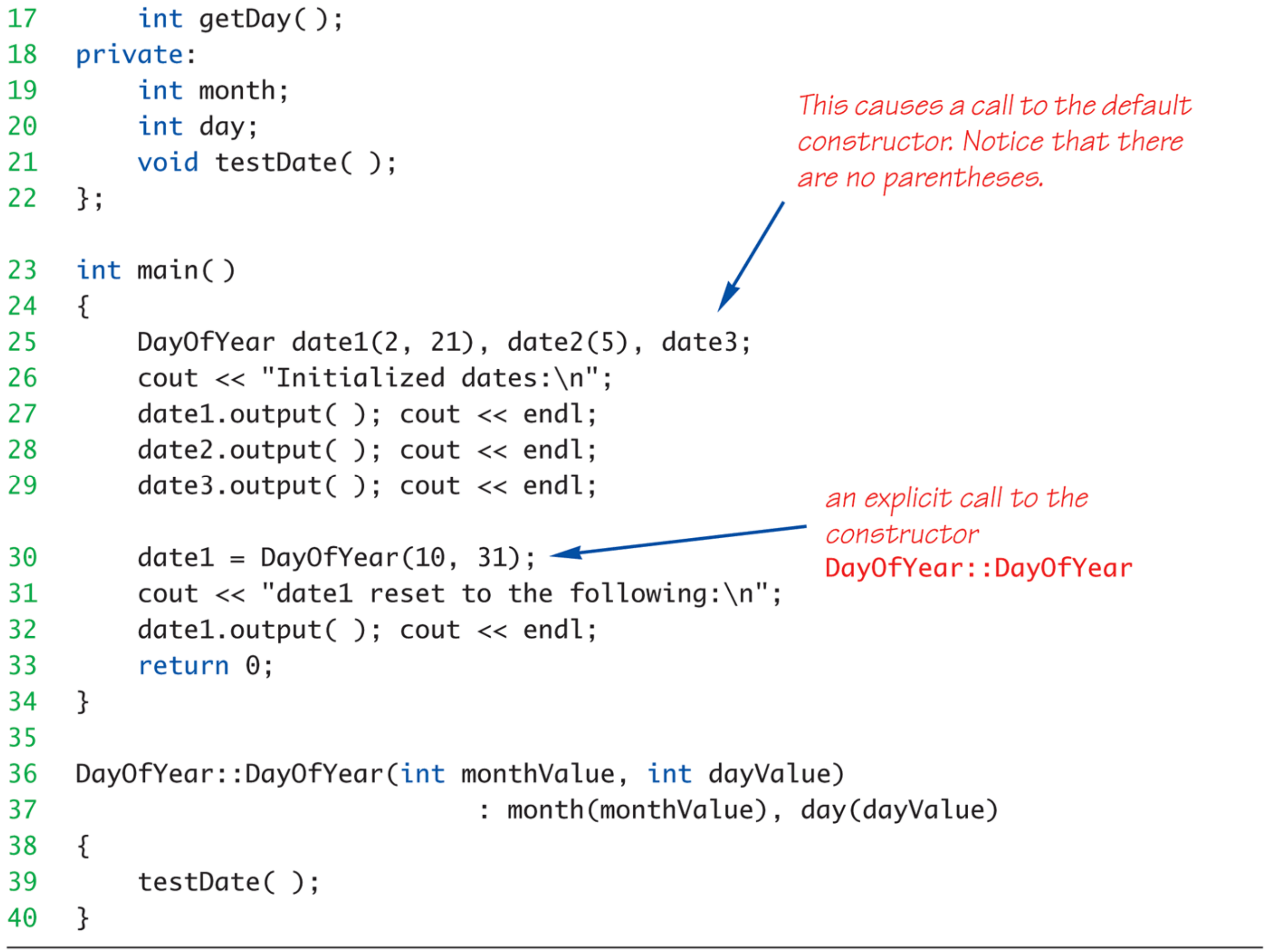
```
#include <iostream>
#include <cstdlib> //for exit
using namespace std;

class DayOfYear
{
public:
    DayOfYear(int monthValue, int dayValue);
    //Initializes the month and day to arguments.

    DayOfYear(int monthValue);
    //Initializes the date to the first of the given month.

    DayOfYear( );
    //Initializes the date to January 1.

    void input( );
    void output( );
    int getMonthNumber( );
    //Returns 1 for January, 2 for February, etc.
```

This definition of **DayOfYear** *is an improved version of the class* **DayOfYear** *given in Display 6.4.*

default constructor

Class with Constructors Example:

Display 7.1 Class with Constructors (2 of 3)

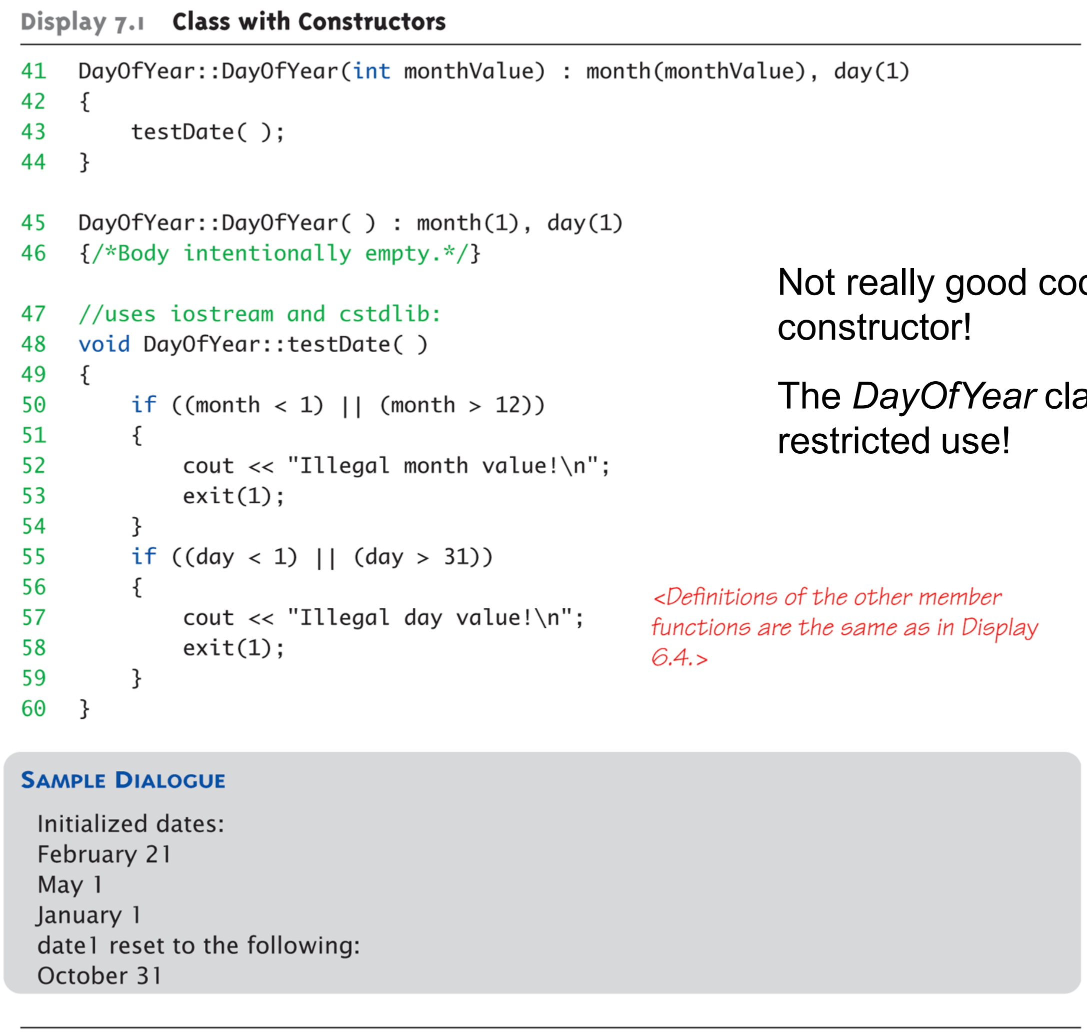


```
    int getDay( );
private:
    int month;
    int day;
    void testDate( );
};

int main( )
{
    DayOfYear date1(2, 21), date2(5), date3;
    cout << "Initialized dates:\n";
    date1.output( ); cout << endl;
    date2.output( ); cout << endl;
    date3.output( ); cout << endl;

    date1 = DayOfYear(10, 31);
    cout << "date1 reset to the following:\n";
    date1.output( ); cout << endl;
    return 0;
}

DayOfYear::DayOfYear(int monthValue, int dayValue)
                          : month(monthValue), day(dayValue)
{
    testDate( );
}
```

This causes a call to the default constructor. Notice that there are no parentheses.

an explicit call to the constructor `DayOfYear::DayOfYear`

Class with Constructors Example: **Display 7.1** Class with Constructors (3 of 3)



Display 7.1 **Class with Constructors**

```
DayOfYear::DayOfYear(int monthValue) : month(monthValue), day(1)
{
    testDate( );
}

DayOfYear::DayOfYear( ) : month(1), day(1)
{/*Body intentionally empty.*/}

//uses iostream and cstdlib:
void DayOfYear::testDate( )
{
    if ((month < 1) || (month > 12))
    {
        cout << "Illegal month value!\n";
        exit(1);
    }
    if ((day < 1) || (day > 31))
    {
        cout << "Illegal day value!\n";
        exit(1);
    }
}
```

<Definitions of the other member functions are the same as in Display 6.4.>

SAMPLE DIALOGUE

```
Initialized dates:
February 21
May 1
January 1
date1 reset to the following:
October 31
```

Not really good code to call in a constructor!

The *DayOfYear* class is now of restricted use!

Destructors

- A destructor is a special function like a constructor
  - Constructors are called by the system as soon as an object has been set up
    - It is the first code executed after the memory for the object data has been allocated
  - Destructors are called just before the object disappears
    - It is the last code executed while there is still memory allocated to hold the object data

```
class WithCounter
{
public:
  WithCounter();
  ~WithCounter();
  static int getCounter();
private:
…
  static int counter;
};
```

destructor

```
WithCounter::WithCounter()
{
  counter++;
}


WithCounter::~WithCounter()
{
  counter--;
}
```

- The destructor is the place to 'tidy up' anything that was done in the constructor (*e.g close opened files*)
- A destructor (which does nothing) is automatically provided if we don't provide one

static members

- A static data member is a data member of the class which has the same value in every object of the class
  - The value can be obtained by a get-method called on any object
  - The value could also be obtained by a get-method called at ‘class level’
- A static data member is *not* a constant value
  - It can be changed in one object, but the new value is applied to every object
  - And it can be changed by a set-method called ‘at class level’
- One common use is to keep a count of how many instances of the class are declared in the program
  - For example whenever we enter a function that declares some objects of the class, the count goes up
  - And when we return from the function the count should go down again

```
class WithCounter
{
public:
…
private:
…
  static int counter;
};
```

? Where must we put the code to increment the count?

In the constructor

? How can we be sure it starts at zero? Where will we initialise it?

Outside any function in the implementation code for the class

```
int WithCounter::counter = 0;
```

At class scope

? What about decrementing the count when an object is no longer present?

Code a destructor

Static member functions

- A function that can be called ‘at class level’
  - Means that nothing in the function depends on knowing data specific to a particular object
  - So these are functions that access only static data

```
Class Counter
{
public:
…
  static int getCount();
private:
…
  static int count;
};
```

```
int Counter::getCount()
{
  return count;
}
```

- A static function can be called on an object in the normal way

```
Counter counter;
…
int num = counter.getCount ();
```

- Or can be called ‘at class level’

```
int num = Counter::getCount ();
```

This is usually considered better coding practise!

Const member functions

If a member function is declared **`const,`** it contracts not to change anything that is *specific to a particular object*

- ie it will not change the values held in any *non-static* data member of the class
- For example ‘get’ member functions should be declared const – they don’t change the data member, just return its value
- *if you declare an object as a const object, then only const member functions can be called on it (unless they are static functions)*
  - *because the compiler cannot be sure other functions will not change data members of the object*.

```
class Counter
{
public:
   Counter();
   ~Counter();
   static int getCount ();
   double getmember2() const;
 private:
…
   static int count;
   double member2;
```

Don’t try to declare a static function as const. It cant change any data member specific to the object anyway.

Notice that the keyword comes after the function declaration

Creating and using classes : header file for class definition

- Class Independence
  - Separate class definition/specification (The “interface”) from class implementation
  - Place in two files

- Class definition
  - Separate from code which use the class
  - Placed in a header (.h) file

- Programs that use the class will "include" the header file
  - `#include "myclass.h“`
  - This places the declaration of the class in the file

- Quotes indicate *you* wrote header
    - Find it in *your* working directory (folder)
  - Recall library includes, e.g., `#include <iostream>`
    - < > indicate predefined library header file
    - Find it in C++ library directory (folder)

Class implementation

- Program Parts
  - Kept in separate files
  - Compiled separately
  - Linked together before program runs

- Place class implementation in .cpp file
  - Typically give interface file and implementation file same name
    - `myclass.h` and `myclass.cpp`
  - All class's member functions defined here
  - Implementation file must #include class's header file
  - If implementation changes, only `myclass.cpp` need be changed
    - Header file and user program stay the same
  - Compile with other files in your application

- Commercially, could build a library of classes
  - Re-used by many different programs
  - Linked in when required, just like the pre-defined libraries
  - We would only have to supply header files for clients of the class to include.

Danger: Multiple Compiles of Header Files

- Header files
  - Potentially could be included multiple times
    - e.g., one header file includes another, and then both are included in your file
  - This would give a compiler error (re-definition of the class)

- Use preprocessor directive
  - Use the `#ifndef` pre-processor directive
  - Tell compiler to include header only once in any one file.

- header file structure:

```
#ifndef FNAME_H_
 #define FNAME_H_
  …
  … //Contents of header file
  …
 #endif
```

This technique avoids multiple inclusions of the header file

- Make FNAME the name of the header file to ensure it is unique, only defined in this one header file

Assignment vs. Initialisation?

Initialization calls the constructor, assignment calls the = operator.

Initialization is creating an instance(of type) with certain value (no pervious value).

```
int i = 0;
```

Assignment is to give value to an already created instance (of type) (pervious value is destroyed).

```
i = 0;
```

Classes always call a constructor on declaration, even if not explicitly coded. A lack of a default constructor will cause compiler error if none is called.

Summary

- Constructors and destructors
  - The constructor contains code which must be the first thing executed after an object has been created
  - Destructors will be the last code executed before the memory for an object is relinquished
  - Preferably use member-wise initialisation of data members
- Static data members and methods
  - Static data members have class scope. They are initialised at the class level
  - Static methods can (and should) be called at class level. They can only change *static* data members.
- Const objects and methods
  - Notice we have NOT talked about const data members
  - Const methods will not change any data specific to an object (*they may change static data though*)
- Separate files for class definition and implementation