Теми:

- Методи и параметри. Даннови членове и пропъртита.
- Модификатори на достъп в клас.
- Accessor-методи. Mutator-метод.
- Методи на клас. Видове и модификатори. Припокриване.

Functions

- declare function prototypes
- define function bodies
- call functions
- deal with local and global variable scope
- define and use overloaded functions

Global functions

- A function declared outside any class declaration
- Example:

```
 void Show(XY xy) // global function,working with
                   // class methods
     {
       printf("x=%f, y =%f\n", xy.GetX(), xy.GetY());
     }
```

- The global function can have name, matching the name of a class member function if differs in parameter list;

Function parameters and arguments

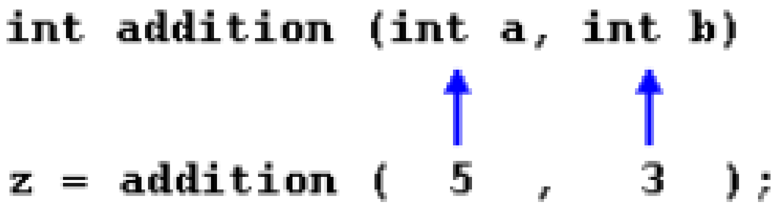
- A **function parameter** (sometimes called a **formal parameter**) is a variable declared in the prototype or declaration of a function:

```
void test(int x); //prototype-x is a parameter
void test(int x) //declaration-x is a parameter
{
}
```

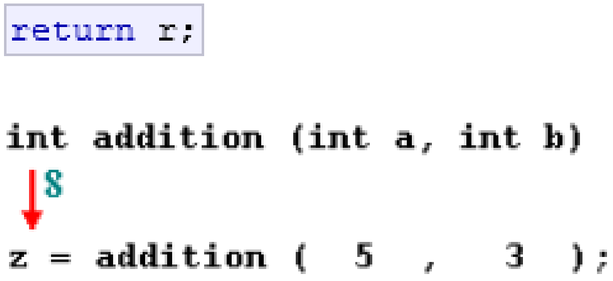
- An **argument** (sometimes called an **actual parameter**) is the value that is passed to the function by the caller:

`test(6);` // 6 is the argument passed to parameter x

`test(y+1);`// the value of y+1 is the argument passed to parameter x

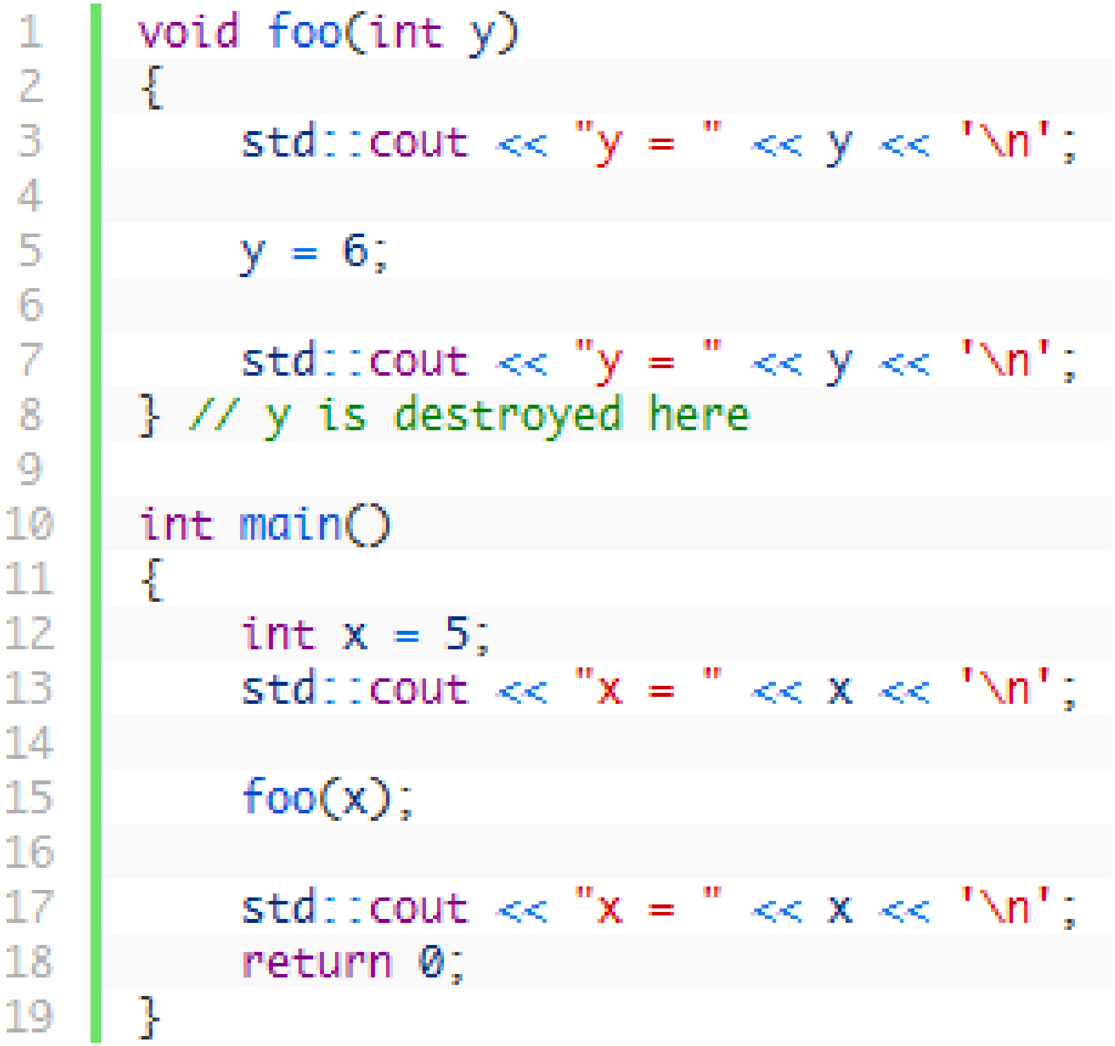


```
int addition (int a, int b)

z = addition (  5  ,   3  );
```



```
return r;

int addition (int a, int b)

z = addition (  5  ,   3  );
```

Passing arguments by value

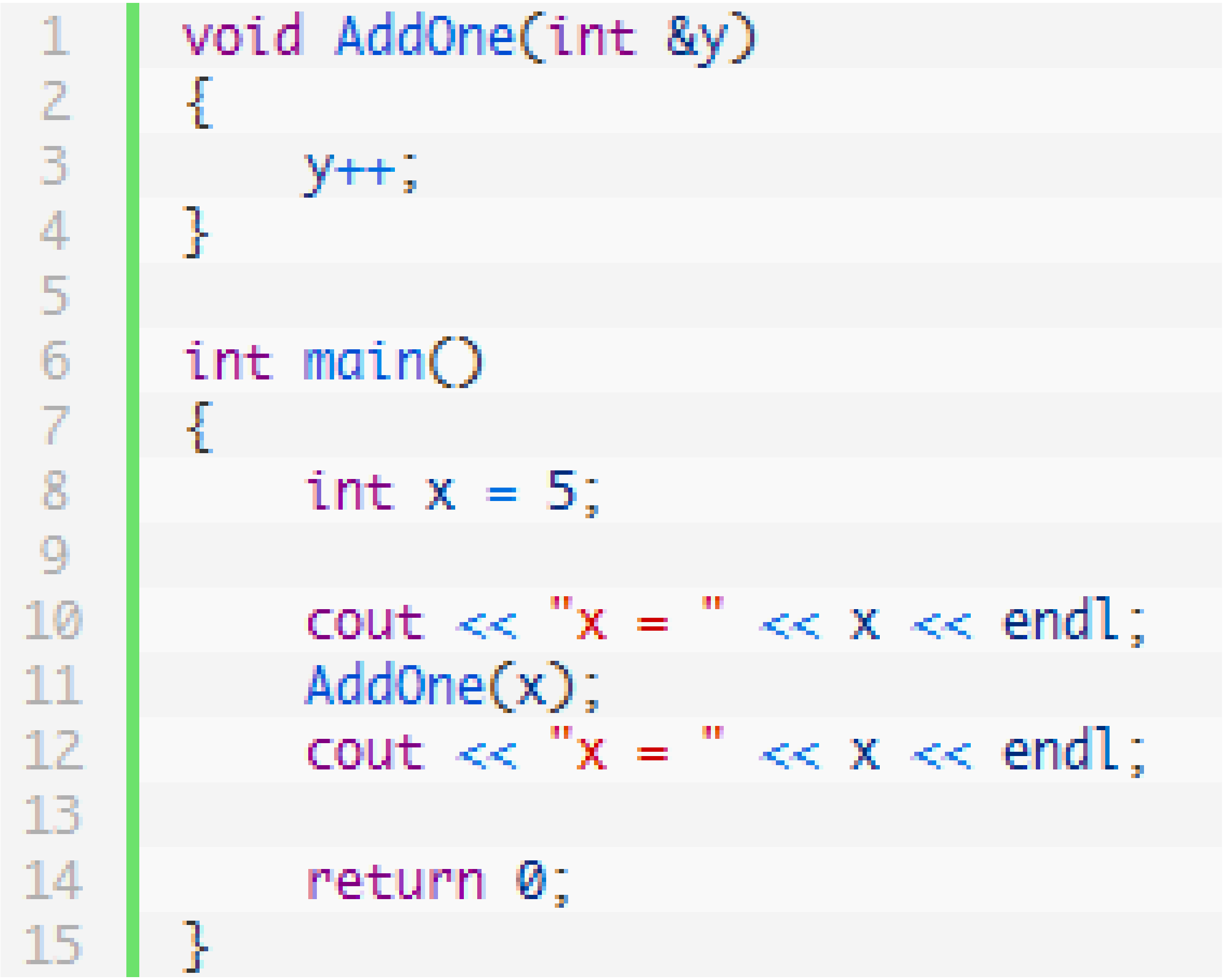


```cpp
void foo(int y)
{
    std::cout << "y = " << y << '\n';

    y = 6;

    std::cout << "y = " << y << '\n';
} // y is destroyed here

int main()
{
    int x = 5;
    std::cout << "x = " << x << '\n';

    foo(x);

    std::cout << "x = " << x << '\n';
    return 0;
}
```

Advantages § Disadvantages

- Advantages of passing by value:
  - Arguments passed by value can be variables (e.g. x), literals (e.g. 6), expressions (e.g. x+1), structs & classes, and enumerators.
  - Arguments are never changed by the function being called, which prevents side effects.
- Disadvantages of passing by value:
  - Copying structs and classes can incur a significant performance penalty, especially if the function is called many times.
- When to use pass by value:
  - When passing fundamental data type and enumerators.
- When not to use pass by value:
  - When passing arrays, structs, or classes.

Passing arguments by reference



```
void AddOne(int &y)
{
    y++;
}

int main()
{
    int x = 5;

    cout << "x = " << x << endl;
    AddOne(x);
    cout << "x = " << x << endl;

    return 0;
}
```

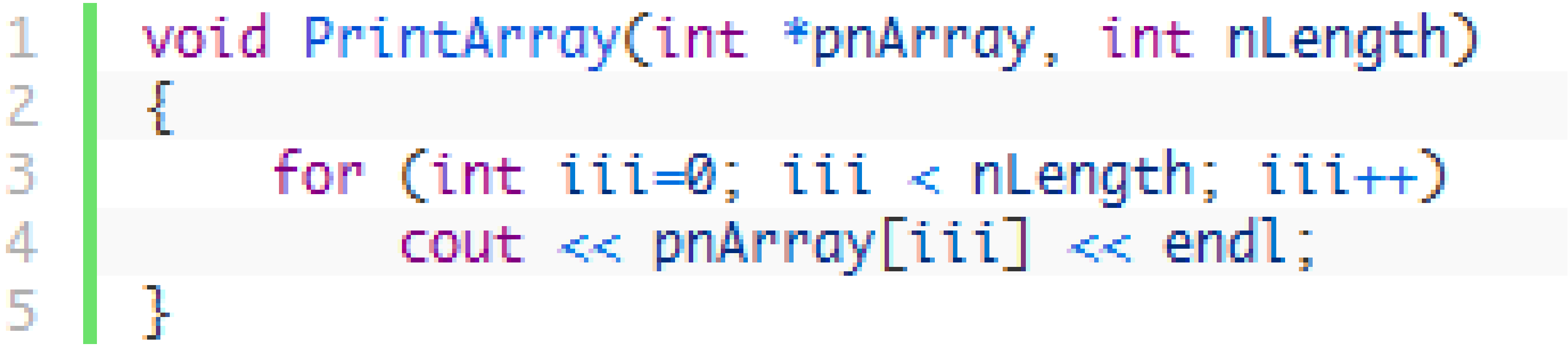
Advantages of passing by address

- It allows us to have the function change the value of the argument, which is sometimes useful.
- Because a copy of the argument is not made, it is fast, even when used with large structs or classes.
- We can pass by const reference to avoid unintentional changes.
- We can return multiple values from a function.

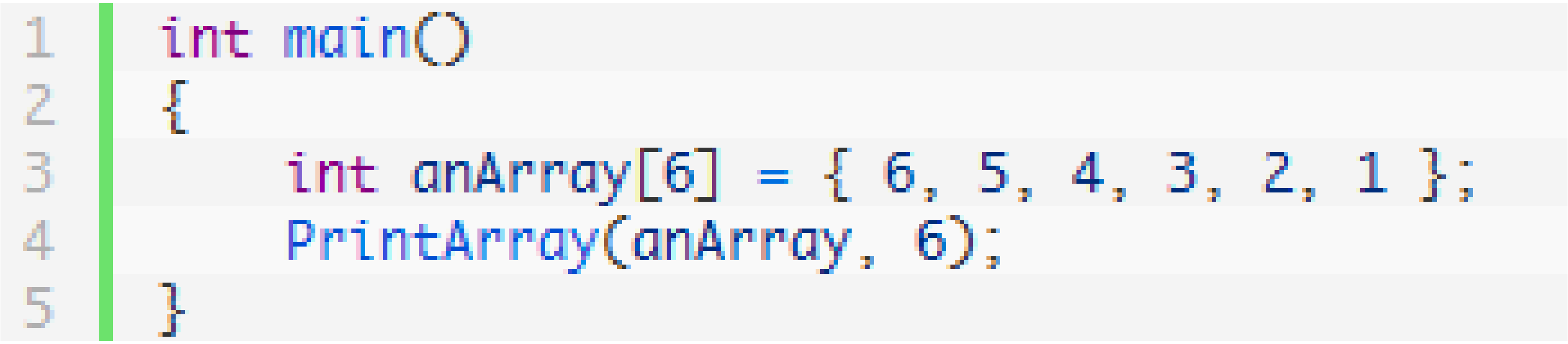
Disadvantages of passing by reference

- Because a non-const reference can not be made to a literal or an expression, reference arguments must be normal variables.
- It can be hard to tell whether a parameter passed by reference is meant to be input, output, or both.
- It’s impossible to tell from the function call that the argument may change.
- Because references are typically implemented by C++ using pointers, and dereferencing a pointer is slower than accessing it directly, accessing values passed by reference is slower than accessing values passed by value.

Passing arguments by address



```cpp
void PrintArray(int *pnArray, int nLength)
{
    for (int iii=0; iii < nLength; iii++)
        cout << pnArray[iii] << endl;
}
```



```cpp
int main()
{
    int anArray[6] = { 6, 5, 4, 3, 2, 1 };
    PrintArray(anArray, 6);
}
```

Advantages of passing by address

- It allows us to have the function change the value of the argument, which is sometimes useful
- Because a copy of the argument is not made, it is fast, even when used with large structs or classes.
- We can return multiple values from a function.

Disadvantages of passing by address

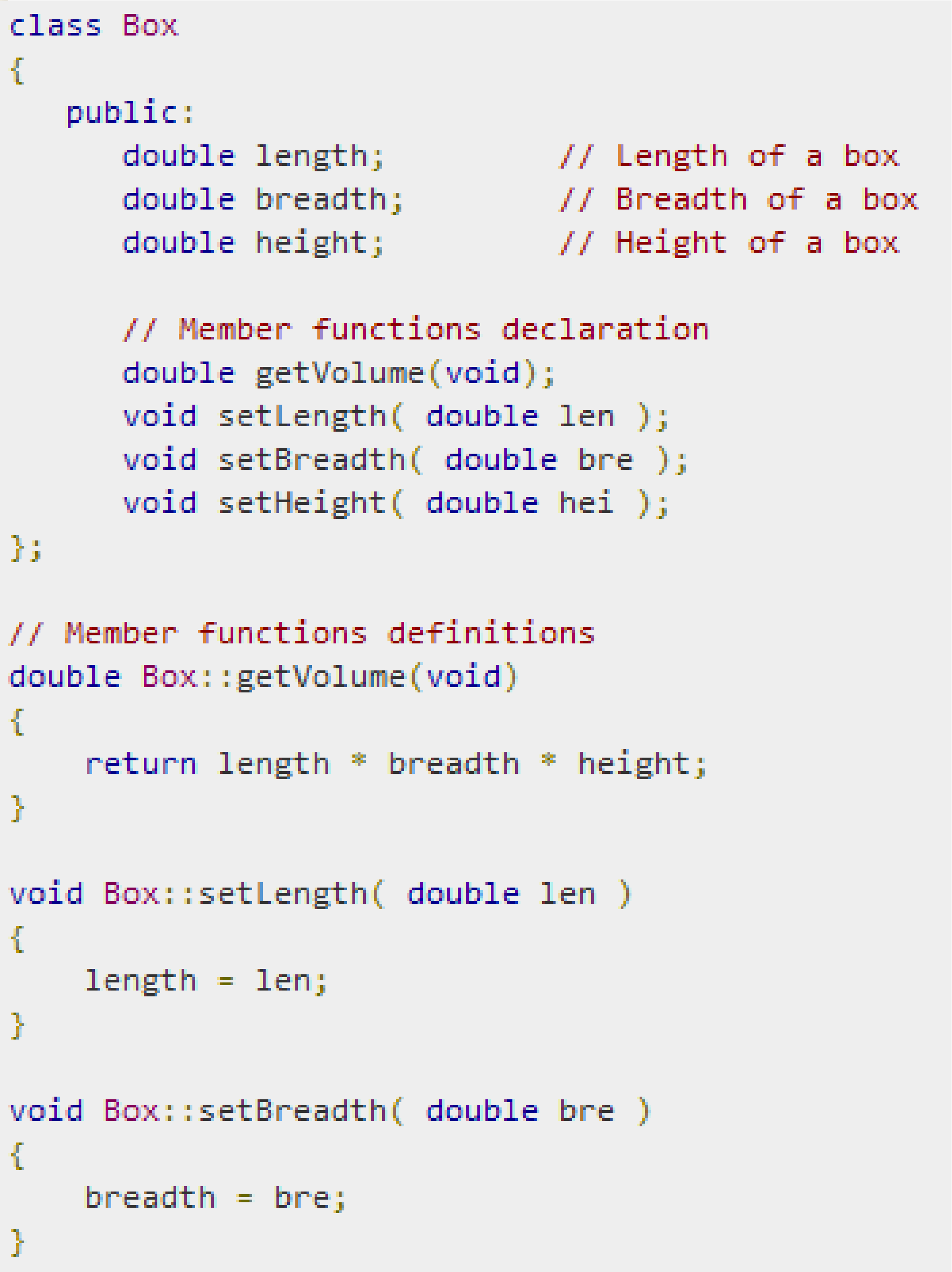
- Because literals and expressions do not have addresses, pointer arguments must be normal variables.
- All values must be checked to see whether they are null. Trying to dereference a null value will result in a crash. It is easy to forget to do this.
- Because dereferencing a pointer is slower than accessing a value directly, accessing arguments passed by address is slower than accessing arguments passed by value.

Member functions

Constructor and destructor are member functions that are always present.
You can add your special-purpose class member functions – declaring them in class declaration
And defining them in class definition:

```
class XY {
public :
        double x,y;
        XY();
        XY(double xarg, double yarg);
        double Getx() const {return x;}
        double GetY() const { return y;}
};
```

Пример



```
class Box
{
   public:
      double length;         // Length of a box
      double breadth;        // Breadth of a box
      double height;         // Height of a box

      // Member functions declaration
      double getVolume(void);
      void setLength( double len );
      void setBreadth( double bre );
      void setHeight( double hei );
};

// Member functions definitions
double Box::getVolume(void)
{
    return length * breadth * height;
}

void Box::setLength( double len )
{
    length = len;
}

void Box::setBreadth( double bre )
{
    breadth = bre;
}
```

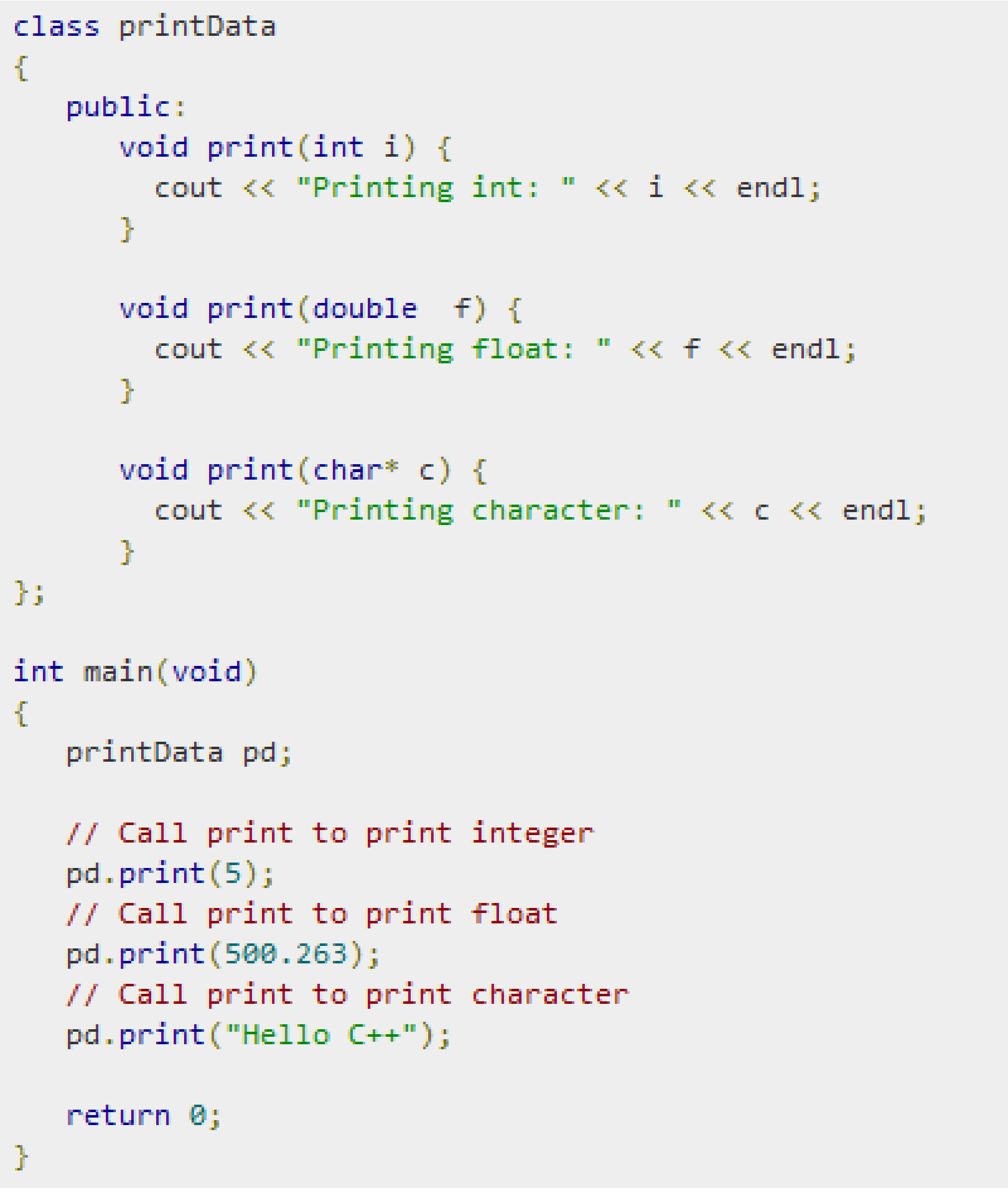
Function overloading

- more than 1 function with same name and different parameter list

- example:

```
double average(double nmber1, double number2);
double average( int array[], int arraysize);
```

- if difference is return type only – you receive a compiler error



```
class printData
{
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }

      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }

      void print(char* c) {
        cout << "Printing character: " << c << endl;
      }
};

int main(void)
{
   printData pd;

   // Call print to print integer
   pd.print(5);
   // Call print to print float
   pd.print(500.263);
   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

Local variables

- Fields are one sort of variable.
  - They store values through the life of an object.
  - They are accessible throughout the class.
- Methods can include shorter-lived variables.
  - They exist only as long as the method is being executed.
  - They are only accessible from within the method.

Local variables

```
public int refundBalance()
{
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```

Information hiding

- Data belonging to one object is hidden from other objects.
- Know what an object can do, not how it does it.
- Information hiding increases the level of *independence*.

Programming tips

public and private elements

```
class XY {
        private:
        double x,y;
        public:
                XY();
                XY(double a, double b);
                double GetX() const;
                double GetY() const;
};
```

- class members are private by default
- object declaration:

```
        XY bottom(4.0, 10.0);
```

- we can access functions from outside

Method calls (2)

Syntax :

object . *methodName* (*parameter-list*)

```
class Date
{
   private:
    int m_nMonth;
    int m_nDay;
    int m_nYear;

   public:
    Date(int nMonth, int nDay, int nYear)
    {
        m_nMonth = nMonth;
        m_nDay = nDay;
        m_nYear = nYear;
    }

    int GetMonth() const { return m_nMonth; }
    int GetDay() const { return m_nDay; }
    int GetYear() const { return m_nYear; }
};
```

Пример (2/2)

```
void PrintDate(const Date &cDate)
{
// although cDate is const, we can call const member functions
    std::cout << cDate.GetMonth() << "/" <<
        cDate.GetDay() << "/" <<
        cDate.GetYear() << std::endl;
}

int main()
{
    const Date cDate(10, 16, 2020);
    PrintDate(cDate);

    return 0;
}
```

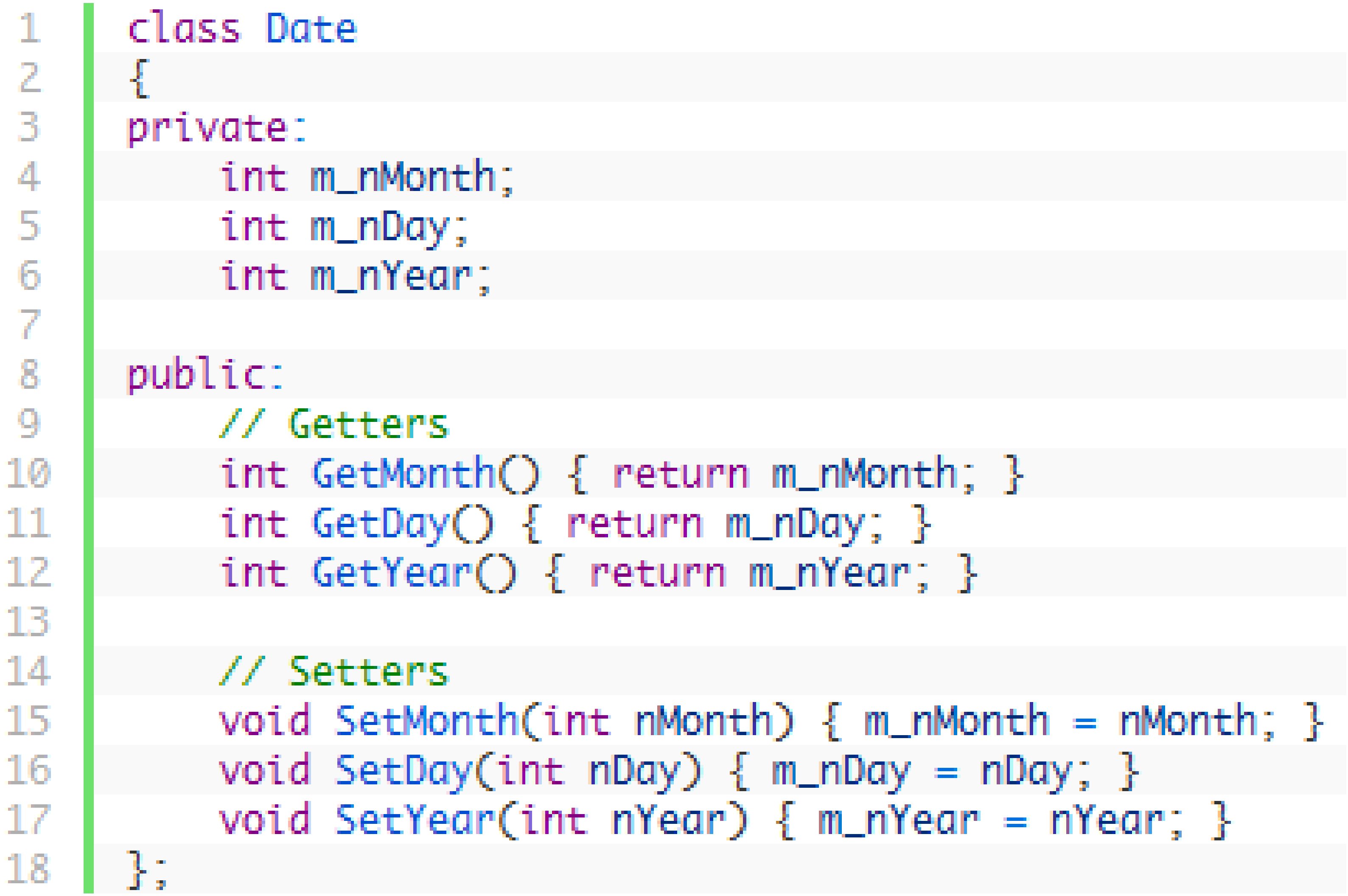
Accessor methods

- Methods implement the behavior of objects.
- Accessors provide information about an object.
- Methods have a structure consisting of a header and a body.
- The header defines the method's *signature*.
- The body encloses the method's statements.

Mutator methods

- Have a similar method structure: header and body.
- Used to *mutate* (i.e., change) an object’s state.
- Achieved through changing the value of one or more fields.
  - Typically contain assignment statements.
  - Typically receive parameters.

Пример



```
class Date
{
private:
    int m_nMonth;
    int m_nDay;
    int m_nYear;

public:
    // Getters
    int GetMonth() { return m_nMonth; }
    int GetDay() { return m_nDay; }
    int GetYear() { return m_nYear; }

    // Setters
    void SetMonth(int nMonth) { m_nMonth = nMonth; }
    void SetDay(int nDay) { m_nDay = nDay; }
    void SetYear(int nYear) { m_nYear = nYear; }
};
```

Свойства на класа в C++/CLI

Основни характеристики

- Името на свойството извиква функция;
- Свойството има get() и set() функция;
  - **read-only property** – дефиниция само на get() функция;
  - **write-only property** - дефиниция само на set() функция.

Видове свойства

- **Скаларни свойства**

  Клас String - свойството Length е скаларно и е read-only защото е дефинирана само функцията get():

  ```
  str->Length
  ```

- **Индексни свойства**

  Класът String ви дава достъп до отделен символ от низа, което е индексно свойство:

  ```
  str[2]
  ```

Дефиниране на скаларни свойства

```
ref class Weight
{
     private:
           int lbs;
     public:
           property int pounds
           {
             int get() { return lbs; }
             void set(int value) {lbs = value;}
           }

};
```

Примери

- **read-only property**

```
property double meters
{
            double get();
}
```

- **write-only property**

```
property double meters
{
            void set(int x);
}
```

Тривиални скаларни свойства

```
value class Point
{
  public:
       property int x;
       propecrty int y;
  };
```

Използване на свойства

```
Weight^ wt = gcnew Weight;
wt->pounds = 162;

Console::WriteLine(L”Weight is {0} lbs.”,
                          wt->pounds);
```

Когато е **ref class** , винаги се достъпва свойството с оператора ->