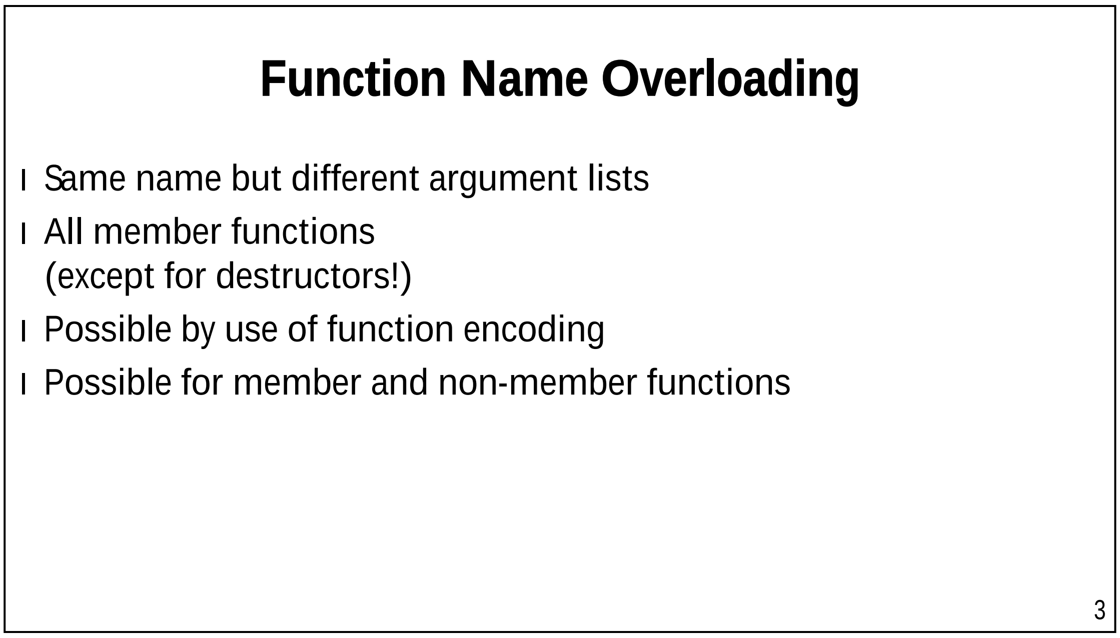
Improving Your Classes

Overview

- Function Name Overloading
- Call by Reference vs. Call by Value
- The 'const' specifier
- The Copy Constructor
- Inline Functions

2



Function Name Overloading

- Same name but different argument lists
- All member functions
  (except for destructors!)
- Possible by use of function encoding
- Possible for member and non-member functions

3

Function Name Overloading

C++, unlike some other languages as C and FORTRAN, allows the programmer to define functions whose names are the same but have different input arguments. Lets take the function print() which must be able to print an integer or a double value. In C we would have to declare two functions with a unique name.

```
void print_int(int value);
void print_double(double value);
```

Without overloading

In C++ we can define two functions with the same name, print(), but with different arguments. We can overload the function by its name and arguments.

We can call the function print() with an int or with a double as argument, the system executes the correct function.

```
void print(int value);          // prints integer value
void print(double value);       // Prints double value
void print(int value, int times);    // Prints value certain times
```

With overloading

Function overloading only applies to the combination of function names and argument types. You cannot overload on the return types.

In the following example two print functions are declared with the same name and arguments but different return types.

```
void print(int);
int print(int);
```

Not possible

Function overloading concentrates solely on the function name in combination with the argument types. Overloaded functions however do not need to have the same return type just as long as their combination of name and argument types is different

Return type does not matter

```
void print(int value);
float print(double value);
```

The above example is possible because the combination of the function name and argument types is unique.

The function overload mechanism can be used when creating class interfaces. Take the constructor of a class. Using this information we can create several constructors. Different constructors mean different ways to create an object.

Overloading Constructors

```
class Date
{
public:
      Date();                         // Default constructor
      Date(int day, int month, int year);
      // Constructor with day, month and year
      Date(int day, int month);       // Current system year is
                                      // selected
};

class Point
{
public:
      Point();                        // Default constructor
      Point(double nx, double ny);    // With x and y value
};
```

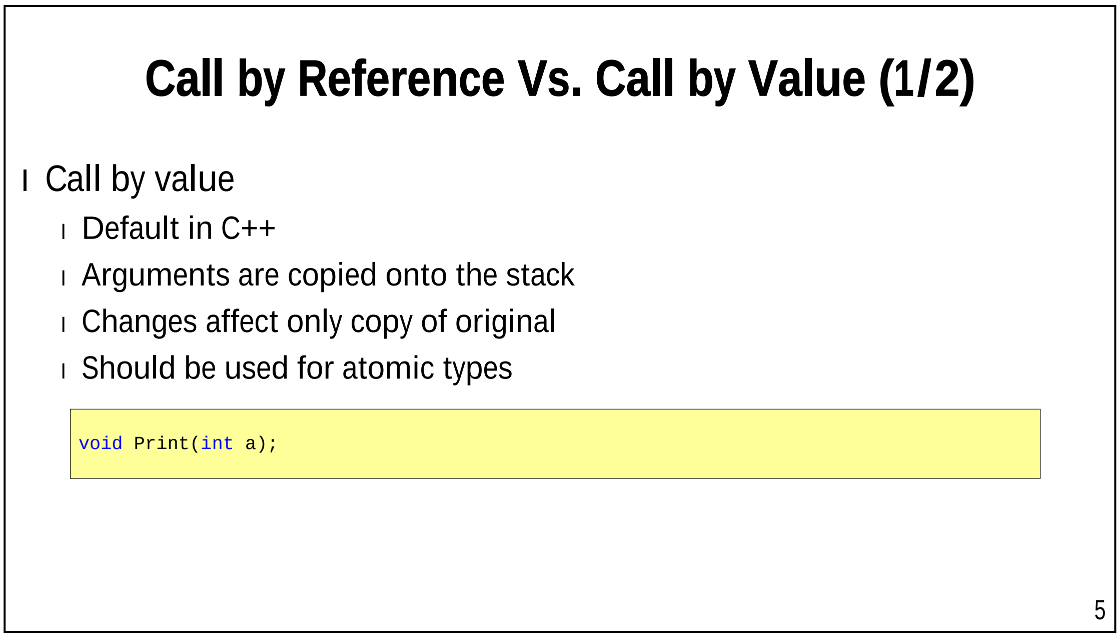
Function overloading is used instead of the classic 'set' and 'get' functions of a class. Take for example the Date class. This class has member function for setting and getting the day, month and year. We could easily use function overloading to get rid of all the get and set prefixes, as below:

New way for Get and Set

```
class Date
{
public:
      int day();                            // Return the day
      void day(int n_day);                  // Set the day
};

class Point
{
public:
      double xvalue();                      // Return the x value
      void xvalue(double newx);             // Set the x value
};
```

Example Function Overloading

```
class Point
{
public:
        ...
  double Distance(Point otherPoint); // Between two points
  double Distance();                 // To origin
};

void Display(Point p);
void Display(Circle c);
```



Call by Reference Vs. Call by Value (1/2)

- Call by value
  - Default in C++
  - Arguments are copied onto the stack
  - Changes affect only copy of original
  - Should be used for atomic types

```
void Print(int a);
```

5

CALL BY REFERENCE VS. CALL BY VALUE

When calling functions we can pass arguments to the functions. These arguments can be passed by value or by reference. When passing arguments by value a copy of the argument is created on the stack. The function receives a copy of the original argument to work with; any changes to this argument inside the function are discarded when the function ends.

This is both an advantage and disadvantage. When the function changes one of its arguments the original value of the argument is still intact.

Call By Value

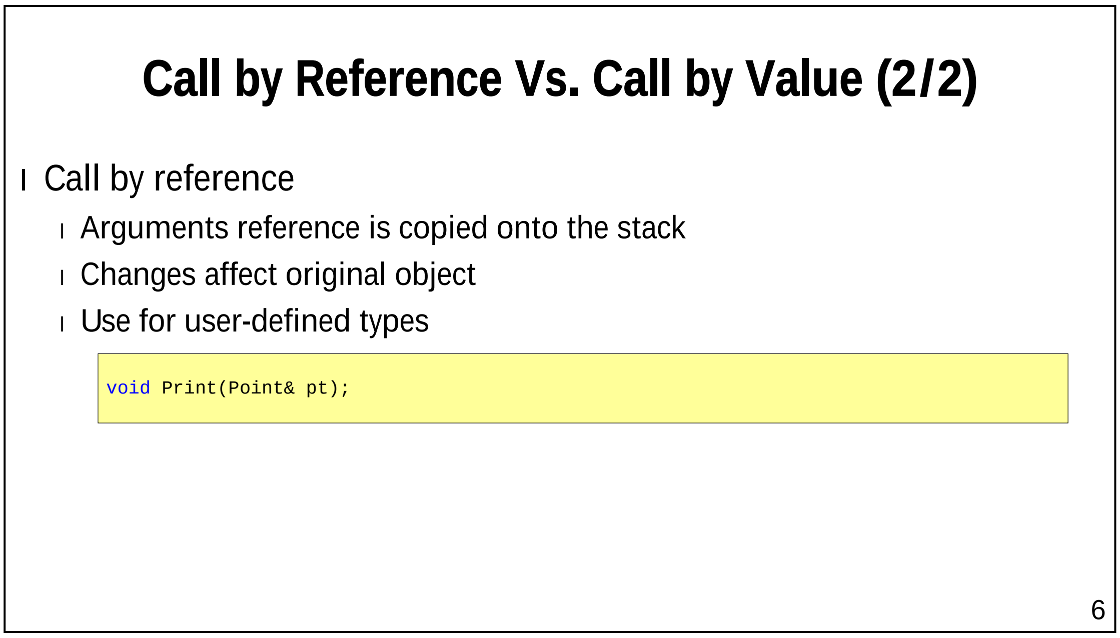
```
// This functions has a copy of the original argument
void change(int value)
{
      value = 10;
}

void main()
{
      int var = 20;
      change(var);      // var still has the value 20
}
```

In some cases you want the function to be able to change the argument in a way that the original value also changes. For example take the function swap(), this function should change the two arguments it receives.

Incorrect Swap

```
void swap(int val1, int val2)
{
      int tmp = val1;
      val1 = val2;
      val2 = tmp;
}
```

When calling this function the original values are not exchanged, because the function has two copies of the original values. One way to solve this is using call by reference.



Call by Reference Vs. Call by Value (2/2)

- Call by reference
  - Arguments reference is copied onto the stack
  - Changes affect original object
  - Use for user-defined types

```
void Print(Point& pt);
```

6

CALL BY REFERENCE

When using call by reference the original arguments are passed to the function. You specify by reference as follows:

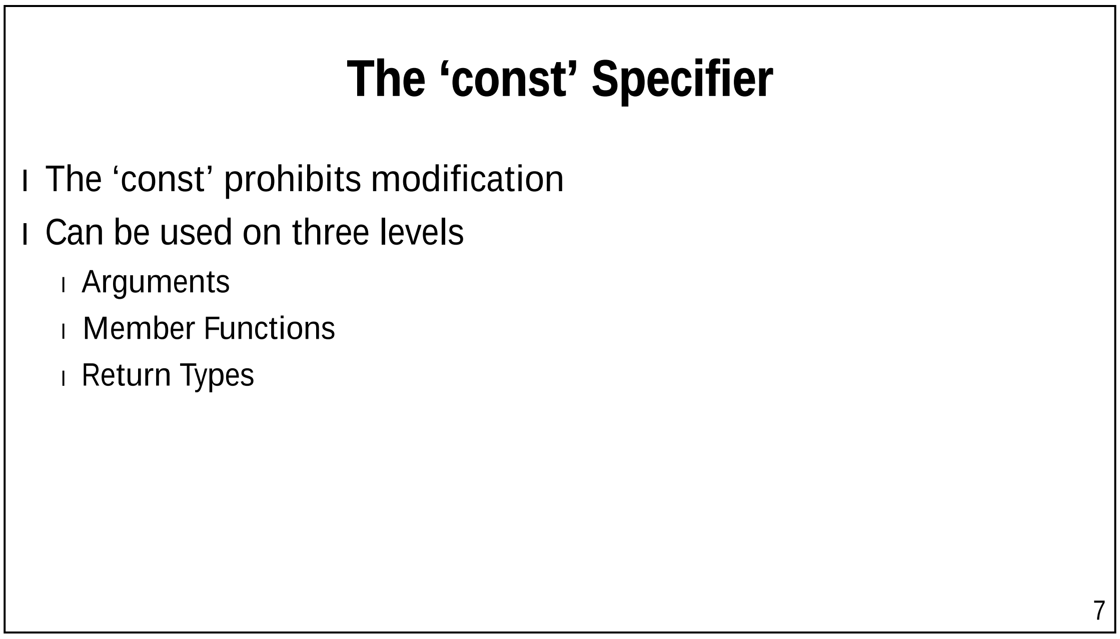
```
void swap(int& val1, int& val2);
```

Correct Swap

The function swap receives two references to ints. Reference arguments can be looked at as normal values. This is the opposite of C. In C you have to use pointers and de-referencing techniques when using call by reference. In C++ you only have to add a '&' after the type of the argument.

```
return_type function(type & var_name);
```

Use call by reference when passing objects to functions. If you do not use references for arguments the arguments are copied. If an argument would be a database with 30.000 drawings stored in it this would take up a lot of stack. That is why a lot of style guides force you to use reference for object types.



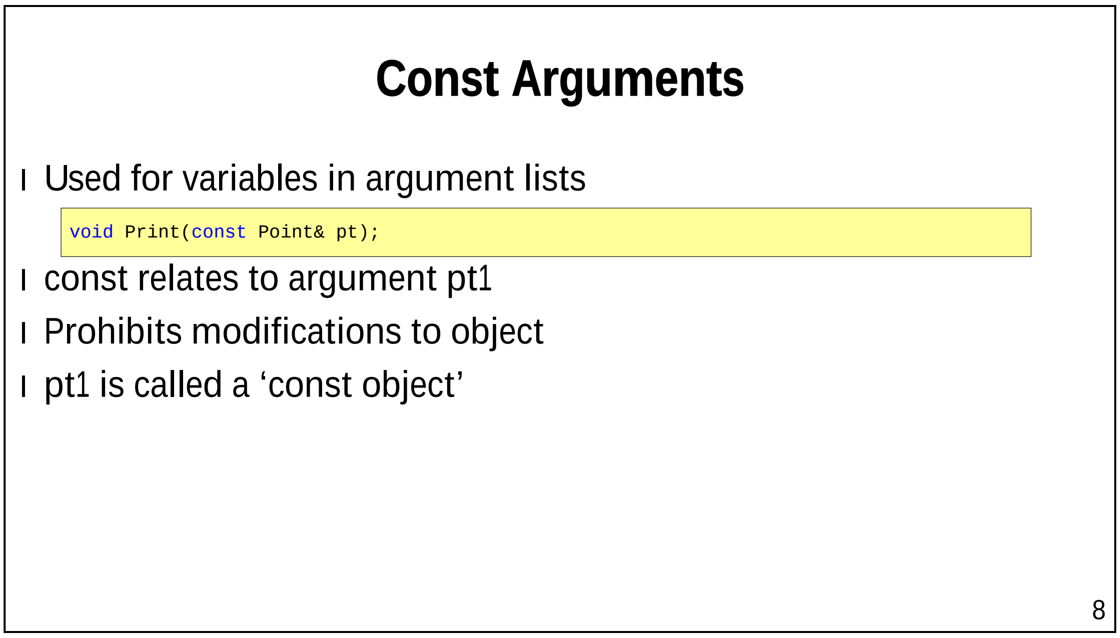
The ‘const’ Specifier

- The ‘const’ prohibits modification
- Can be used on three levels
  - Arguments
  - Member Functions
  - Return Types

7

THE ‘CONST’ SPECIFIER

The const specifier has a lot of uses in C++. In C you could use it for const variables and const arguments. Using the const you could create a variable that would get an initial value that could never be changed it was a const variable. The same applied to arguments. Specifying a const argument meant that the function could not use the variable to store temporary values or change the variable. This is often not used because the function already has a copy by default that has no relation with the original variable or value passed. If the function would change the copy it would only affect its own code. In C++ these usages are still applicable but one more has been added.



Const Arguments

- Used for variables in argument lists

```
void Print(const Point& pt);
```

- const relates to argument pt1
- Prohibits modifications to object
- pt1 is called a ‘const object’

8

Const Arguments

The call by reference option is used to not create a copy of the arguments passed to the function. The possibility to change the argument can be seen as a disadvantage too. Suppose the caller of the function wants to be sure that his object will not change within the body of the function.

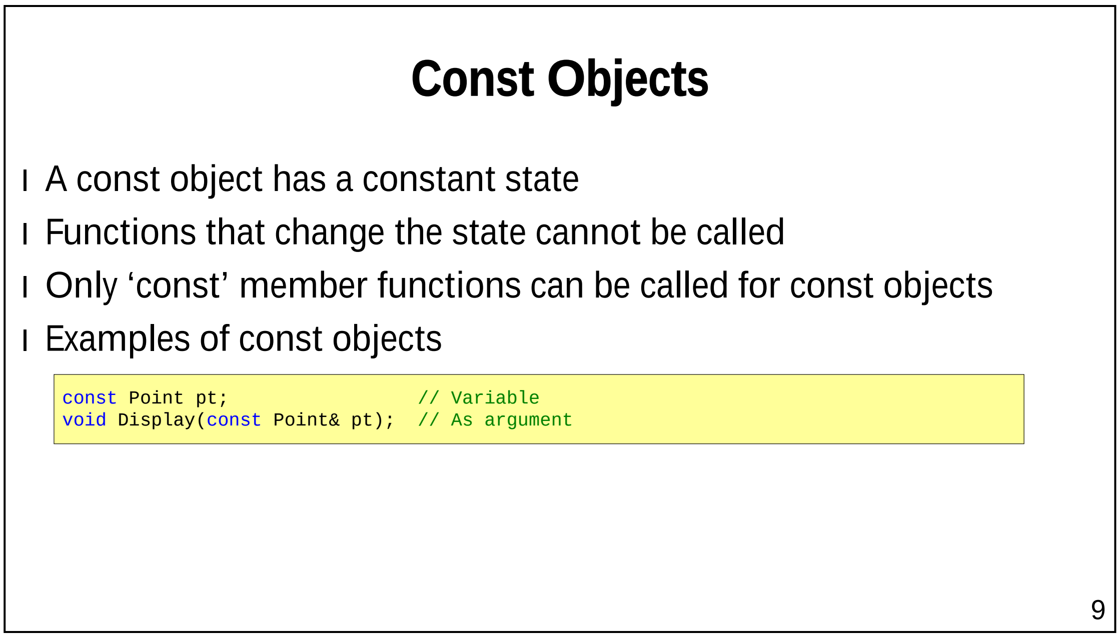
You can make sure that the argument is not changed by specifying that the argument is a 'const' argument. This is accomplished by placing the 'const' keyword in front of the type.

Const References

```
void print(const Point& pt);
void print(const Date& dt);
```

The print function cannot change the argument in the print function. You can call these functions with a const object and a non-const object. When you call the functions with a non-const objects the objects get more secure inside the function.

Using the reference and 'const' for built in types is unnecessary and we strongly advise that you use call by value.



Const Objects

- A const object has a constant state
- Functions that change the state cannot be called
- Only ‘const’ member functions can be called for const objects
- Examples of const objects

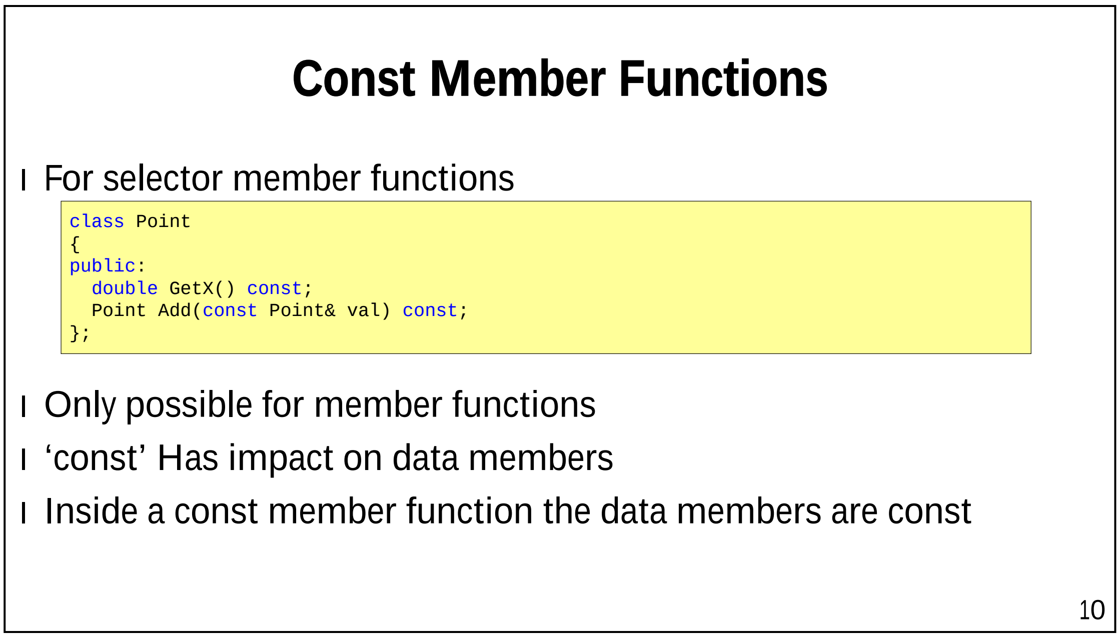
```
const Point pt;                 // Variable
void Display(const Point& pt);  // As argument
```

9

Const Objects

Creating const objects is much the same as creating const variables in C. A const object cannot change after it has been declared. Const objects are often used for global values, which only get a value from program start up.

These const objects cannot change so all member functions that change the object state cannot be called. The compiler checks to see if the functions you call on the object do not change it, it does so by checking if it is a so called ‘const’ member function. Const member functions specify that they do not change any state of the object. By default all member functions are non-const.



Const Member Functions

- For selector member functions

```
class Point
{
public:
  double GetX() const;
  Point Add(const Point& val) const;
};
```

- Only possible for member functions
- 'const' Has impact on data members
- Inside a const member function the data members are const

10

Const Member Functions

Making an object const makessure that only so called 'const' member functions can be called for the const object.

Const Objects

```
const Date dt;    // Const date object
const Point pt;   // Const point object
```

Const member functions are functions that do not change the state of the object. Typical examples of const member functions are accessing or selector functions.

Const Member Functions

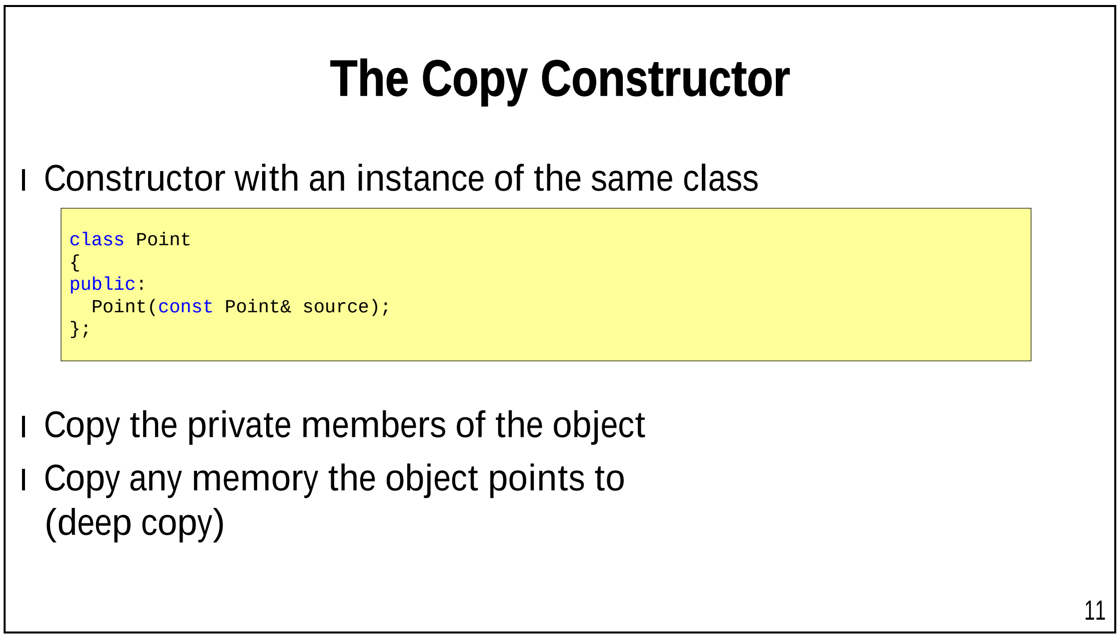
```
class Date
{public:
      int ret_day() const;          // Constant member function
      int ret_month() const;        // Constant member function
};

class Point
{public:
      double ret_x() const;
      Point add(const Point& pt) const;
};
```

Making a function const that changes the state of the object will produce a compile error.

Making a member function const implies that the state of the current object in the body of the function is const, the compiler checks. **This is the reason why normal functions cannot be const.**
So there are three possibilities for the const keyword:

- const arguments; const relates to argument
- const member function; const relates to state of object
- const return types; the return type cannot be changed by the caller

Use const arguments when passing objects by reference. Use const member functions when the functions do not change the state of the object (selector functions).



The Copy Constructor

- Constructor with an instance of the same class

```
class Point
{
public:
  Point(const Point& source);
};
```

- Copy the private members of the object
- Copy any memory the object points to (deep copy)

11

THE COPY CONSTRUCTOR

The constructors for a class can be overloaded to make it possible to create an instance of a class in different ways.

If we want to create an object by passing the information of an object of the same type we have to create a constructor that has an object of the same type as argument.

Copy Constructor

```
class Date
{
public:
	Date(const Date& d);
};

class Point
{
public:
	Point(const Point& pt);
};
```

This constructor is called the copy constructor. Notice the const and reference type of the argument (we are passing a user-defined type). The const is not mandatory but the reference is. This constructor is the function that is used to make a copy. When you call a function with a call-by-value object the object needs to be copied. The copy constructor is called. Suppose you would not specify a reference argument for this constructor. If the system wants to make a copy of the call-by-value object it will call the copy constructor that wants a copy (because of the call by value) it is recursive.

CALL TO COPY CONSTRUCTOR

The copy constructor is be called by creating an object and passing an object of the same type.

Calling Copy Constructor

```
Date d1;
Date d2(d1);      // Copy constructor is called
```

The copy constructor is also called by the system when a copy of an object is needed. Suppose you pass an object by value, the system needs to make a copy. In this case the copy constructor is called. When a function returns an object a copy is created again the copy constructor is called.

The copy constructor can be called in three different ways.

1) by passing an object as argument when creating an instance
```
      Point p1;
      Point p2(p1);         // Explicit call to copy constructor
```

2) by passing an object as call by value
```
      void print(Point p); // No call by reference
      Point p1;
      print(p1);           // Copy of p1 provided to print function
```

3) function returning an object
```
      Point function()
      {
            Point p1;
            return p1;     // Copy of p1 is created and returned
      }
```

RESPONSIBILITY COPY CONSTRUCTOR

The responsibility of the copy constructor is to create an instance from an instance of the same class. Creating a copy of an instance of the same type is performed by copying the data members of the original object. The argument of the constructor is the source and the current instance, with its data members, is the destination NOT VICE VERSA.

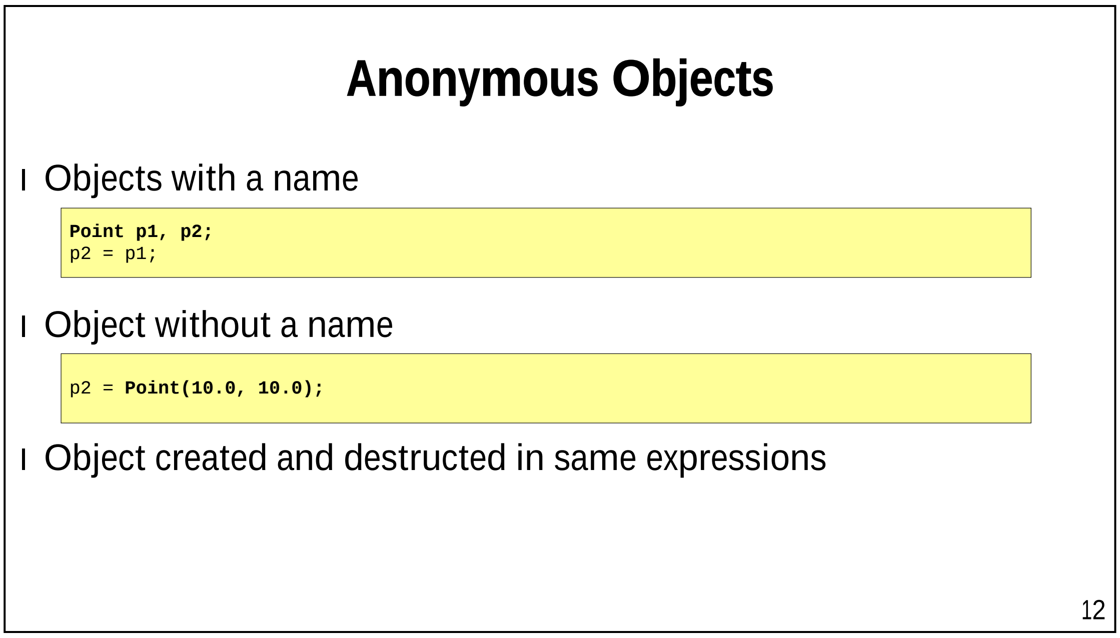
In later chapters we shall see that a class can have pointers in its state. Memory blocks need to be copied and not the pointer itself. This topic will be elaborated when pointers in classes are discussed.

When we do not deliver a copy constructor the system will use make a bitwise copy of the object when a copy is created. This is not what we want. We do not want to rely on something the system provides.

Copy Constructor

```
Date::Date(const Date& d)
{
      m_day = d.m_day;
      m_month = d.m_month;
      m_year = d.m_year;
}
```

This material is intended for students of the "C++ Programming for Financial Engineering" certificate



Anonymous Objects

- Objects with a name

```
Point p1, p2;
p2 = p1;
```

- Object without a name

```
p2 = Point(10.0, 10.0);
```

- Object created and destructed in same expressions

12

ANONYMOUS OBJECTS

An anonymous object is an instance of a class which has no variable name bound to it. Such objects are used when we wish to create an instance of a class without being burdened with having to give it a name. For example the function;

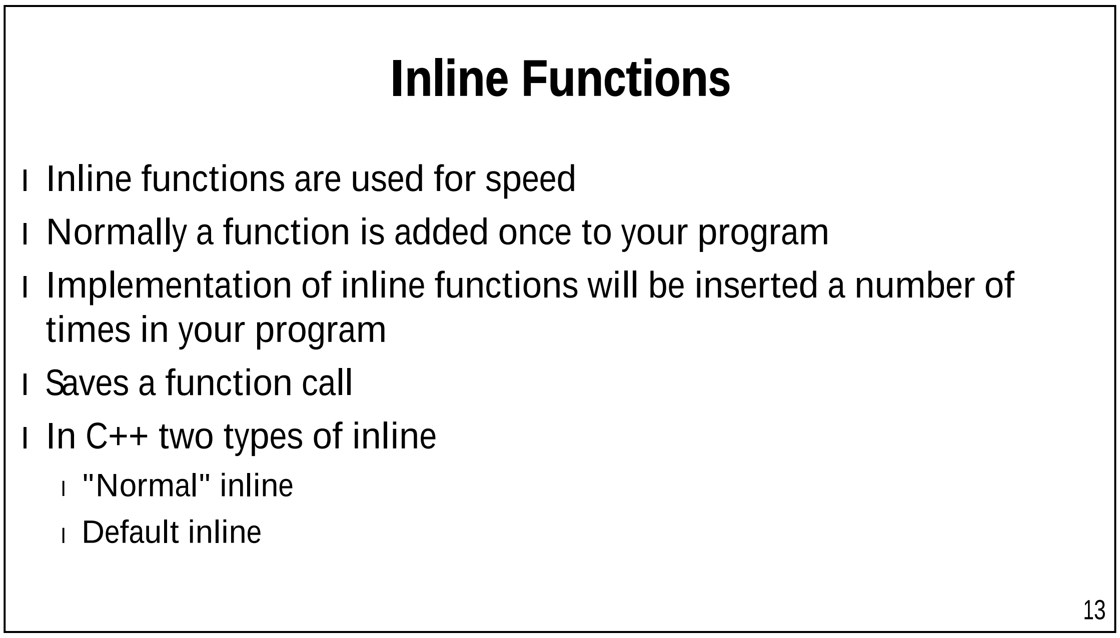
Without anonymous object

```
Point Point::operator+(const Point& pt)
{
        Point res;

        res.set_x(x + pt.x);
        res.set_y(y + pt.y);

        return res;
}
```

Can be optimised so that its body uses an anonymous object:

With anonymous object

```
Point Point::operator+(const Point& pt)
{
        return Point(x + pt.x, y + pt.y);
}
```



Inline Functions

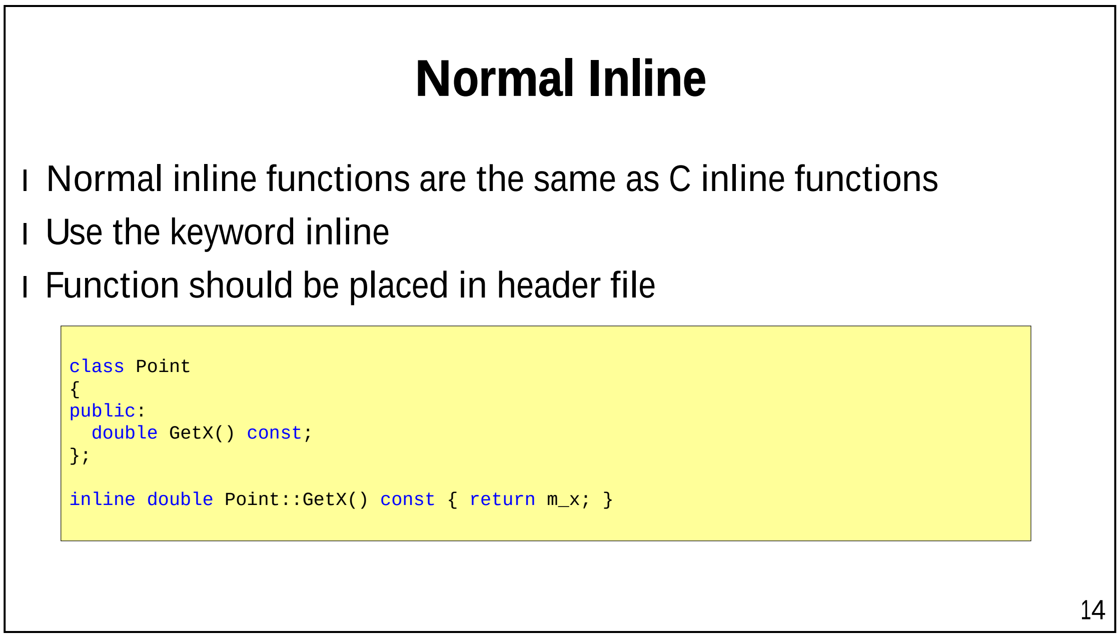
- Inline functions are used for speed
- Normally a function is added once to your program
- Implementation of inline functions will be inserted a number of times in your program
- Saves a function call
- In C++ two types of inline
  - "Normal" inline
  - Default inline

13

INLINE FUNCTIONS

Inline functions are generally something you should try to prevent from using. Inline functions are functions that are inserted into your code whenever the function is called. The general functions are placed once in your program memory at a certain memory address and when the function is called the code at that specific address is executed. The code of an inline function is copied to the place where the function is called. So for example if we have a general function which is called 10 times in the program the function resides only at one specific memory address. An equivalent inline function would be placed ten times in the program code. This is less efficient for memory but more efficient for speed.

Inline function is not a new technique of C++ but something that is part of C since the beginning. The C variant is called the "normal inline". C++ has added one that is called "default inline" which is used for member functions in classes.



Normal Inline

- Normal inline functions are the same as C inline functions
- Use the keyword inline
- Function should be placed in header file

```
class Point
{
public:
  double GetX() const;
};

inline double Point::GetX() const { return m_x; }
```

14

NORMAL INLINE

Normal functions have only one piece of actual code in an application. Every time that a function is called the system jumps to the address of that function and executes it. With inline functions you save a jump to a function address. The disadvantage is that your program will grow the more you use these functions. The body of the function is copied every time you try to execute the function.

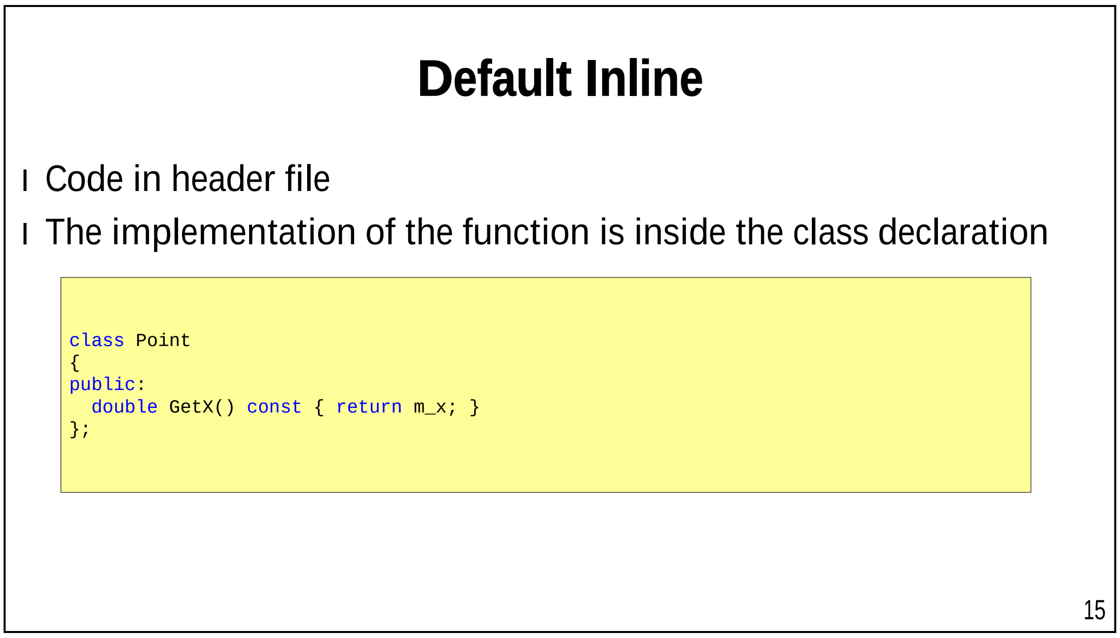
Inline functions are specified with the 'inline' keyword.

Inline Functions

```
inline double sqr(double x)
{
       return (x*x);
}
```

Member functions can be created inline as well:

```
class Simple
{
private:
       int val;
public:
       Simple();
       ~Simple();

       int GetValue();
};

inline int Simple::GetValue()
{
       return val;
}
```



Default Inline

- Code in header file
- The implementation of the function is inside the class declaration

```
class Point
{
public:
  double GetX() const { return m_x; }
};
```

15

Default Inline

Besides the normal inline functions C++ offers default inline functions. These functions are placed in the header file of the class.

Default Inline Functions

```
class Point
{
public:
        double xvalue() {return x;}
};
```

These functions are used for short functions, for example the accessing functions as shown in the example.

Try to avoid using these functions. If you use these functions the principle of separating the class in a header and source file is violated. And when in the future the implementation of the accessing function changes you have to change the header file that we normally use for the documentation of the class. The documentation must therefore be changed too.