

# OOP

Gavrilut Dragos  
Course 2

# Summary

- ▶ Pointers and References
- ▶ Method overloading
- ▶ NULL pointer
- ▶ “const” specifier
- ▶ “friend” specifier



# Pointers and ► References

# Pointers and References

## App-Pointer

```
void SetInt(int *i)
{
    (*i) = 5;
}
void main()
{
    int x;
    Set(&x);
}
```

## App-Reference

```
void SetInt(int &i)
{
    i = 5;
}
void main()
{
    int x;
    Set(x);
}
```

## App-Pointer (asm - SetInt)

```
SetInt:
    push    ebp
    mov     ebp,esp
    mov     eax,[ebp+8]
    mov     [eax],5
    mov     esp,ebp
    pop     ebp
    ret
```

## App-Reference (asm - SetInt)

```
SetInt:
    push    ebp
    mov     ebp,esp
    mov     eax,[ebp+8]
    mov     [eax],5
    mov     esp,ebp
    pop     ebp
    ret
```

# Pointers and References

- ▶ The resulted code is identical (both the “Pointer” and “Reference” program will link into the same assembler code).
- ▶ However, from the programmer point of view, using a reference fixes some possible problems (perhaps the most know one is that one does not need to use the “->” operator - instead the “.” operator can be used). Another important one is that a check for NULL pointers is no longer required.

## Pointer

```
struct Date
{
    int X;
}
void SetInt(Date *d)
{
    d->X = 5;
}
```

## Reference

```
struct Date
{
    int X;
}
void SetInt(Date &d)
{
    d.X = 5;
}
```

# Pointers and References

- ▶ References and pointers are created in the following manner:

## Pointer

```
int i = 10;  
int *p = &i;
```

## Reference

```
int i = 10;  
int &refI = i;
```

- ▶ The difference is that pointers can remain uninitialized.

## Pointer

```
int i = 10;  
int *p;
```

## Reference

```
int i = 10;  
int &refI;
```

Compile error -  
uninitialized reference

This forces the programmer to initialize a reference.  
It also guarantees that a reference points to a valid memory location.

# Pointers and References

- ▶ A pointer value can be changed (that is a pointer can point to different memory addresses). A reference can only point to a variable and once it is initialized the memory address where it points to can not be changed.

## Pointer

```
int i = 10;  
int j = 20;  
int *p = &i;  
p = &j;
```

- ▶ A pointer can have the value NULL. A reference can only point to a memory address that exists.

## Reference

```
int i = 10;  
int j = 20;  
int &refI = i;  
&refI = j;
```

Compiler error - trying to change a reference that was already initialized.

# Pointers and References

- ▶ Pointers accept certain arithmetic operations (+, -, ++, etc). This is not valid for references.

## Pointer

```
int i = 10;  
int j = 20;  
int *p = &i;  
p++;  
(*p) = 30;
```

## Reference

```
int i = 10;  
int j = 20;  
int &refI = i;  
refI++;  
(&refI)++;
```

Compile error

- ▶ In case of pointers, variable “i” and “j” are allocated consecutively on the stack. The operation “p++” moves the pointer p from the memory address of the variable “i” to the memory address of the variable “j”. At the end of the execution “j” will have the value 30.



# Pointers and References

- ▶ A pointer can be converted to another pointer (cast). In particular any pointer can be converted to a void pointer (void\*). A reference can not be converted to another reference.

## Pointer

```
int i = 10;  
char *p = (char *)&i;
```

## Reference

```
int i = 10;  
char &refI = i;
```

Compile error

- ▶ This thing guarantees that a reference points to a memory address where a certain type of variable resides.

# Pointers and References

- ▶ A pointer may point to another pointer and so on. This is not possible for references - a reference refers only a variable.

## Pointer

```
int i = 10;  
int *p = &i;  
int *p_to_p = &p;  
**p_to_p = 20;
```

## Reference

```
int i = 10;  
int &refI = i;  
int & &ref_to_refI = refI;
```

Compile error

- ▶ It is important in this example to differentiate between “& &” (two references separated with a space (‘ ’) character) and “&&” (two consecutive references).

# Pointers and References

- ▶ A pointer can be used in an array and be dynamically initialized. This is not possible for references.

## Pointer

```
int *p[100];
```

## Reference

```
int &ref[100];
```

Compile error

- ▶ However, a reference may point to a temporary (or constant) value.

## Pointer

```
int *p = &int(10);
```

## Reference

```
const int &refI = int(12);
```

This code will not compile if the “const” specifier is not used as it refers to a constant numerical value.

# Pointers and References

- ▶ A pointer can be used in an array and be dynamically initialized. This is not possible for references.

## Pointer

```
int *p[100];
```

## Reference

```
int &ref[100];
```

Compile error

- ▶ However, a reference may point to a temporary (or constant) value.

## Pointer

```
const int &refI = int(12);  
int *p = (int *)&refI;
```

## Reference

```
const int &refI = int(12);
```

It is however possible to create a pointer that points to a reference of a temporary (constant) value.



# ▶ Method overloading

# Method overloading

- ▶ Method overloading is a technique used in C++ where one can define 2 or multiple functions/methods with the same name (or operators).
- ▶ A function / method is uniquely identified by its signature:

return-type ***FunctionName (param1-type, param2-type, ... )***

A function/method signature is form out of:  
1) function name  
2) Parameters type (if parameters are present)

- ▶ Since parameters are part of the function signature, multiple functions/methods with the same name but different parameters are possible.
- ▶ However, this does not apply to return type (meaning that functions with the same name and parameters but different return type can not exist).

# Method overloading

## App.cpp

```
class Math
{
public:
    int  Add (int v1, int v2);
    int  Add (int v1, int v2, int v3);
    int  Add (int v1, int v2, int v3, int v4);
    float Add (float v1, float v2);
};

int Math::Add(int v1, int v2)
{
    return v1 + v2;
}

int Math::Add(int v1, int v2, int v3)
{
    return v1 + v2 + v3;
}

int Math::Add(int v1, int v2, int v3, int v4)
{
    return v1 + v2 + v3 + v4;
}

float Math::Add(float v1, float v2)
{
    return v1 + v2;
}
```

# Method overloading

- ▶ Method overloading is NOT possible if the methods have the same signature (same name, same parameters)
- ▶ In the next case, both methods are named **Add** and have two parameters of type **int** ). The return type (even if in this case is different) will not be considered , thus the two **Add** functions are consider duplicates !

## App.cpp

```
class Math
{
public:
    int Add(int v1, int v2);
    long Add(int v1, int v2);
};
int Math::Add(int v1, int v2)
{
    return v1 + v2;
}
long Math::Add(int v1, int v2)
{
    return v1 + v2;
}
```



# Method overloading

- ▶ Be careful when you are using parameters with default value. From the compiler point of view, using this feature does not mean that a function has fewer parameters !
- ▶ This code will NOT compile as *Add* has the same signature !

## App.cpp

```
class Math
{
public:
    int  Add(int v1, int v2);
    long Add(int v1, int v2 = 0);
};
int Math::Add(int v1, int v2)
{
    return v1 + v2;
}
long Math::Add(int v1, int v2)
{
    return v1 + v2;
}
```

# Method overloading

- ▶ Another special case are methods with variadic parameters (“...”). However, they are not recommended in case of method overloading as the interpretation can be misleading.

## App.cpp

```
class Math
{
public:
    int  Add(int v1, int v2);
    long Add(int v1, ...);
};

int Math::Add(int v1, int v2)
{
    return v1 + v2;
}

long Math::Add(int v1, ...)
{
    return v1;
}
```

# Method overloading

- ▶ When a function/method that was overloaded is called, the compiler determines which one of the existing definitions of that function/method it should use. This process is called **overload resolution**
- ▶ It is possible that the result of this process will be inconclusive (e.g. - the compiler can not decide the best fit for a specific name). In this case a compiler error will be raised, and the ambiguity will be explained.

# Method overloading

## Overload resolution steps:

1. Check if an exact match is possible (a method exists with the same name and the exact same parameters type)

Defined	<code>void Compute(int x, double y, char z)</code>
Called	<code>Compute(100, 1.5, 'A')</code>

# Method overloading

## Overload resolution steps:

2. Check if a numerical promotion is possible (convert a type into another one without losing precision and the value).
  - ✓ **bool**, **char**, **short**, **unsigned char** and **unsigned short** can be promoted to **int**
  - ✓ **float** can be promoted to **double**
  - ✓ Any enumeration (**enum**) without an explicit type can be converted to **int**

Defined	<code>void Compute(int x, double y, char z)</code>
Called	<code>Compute(true, 1.5f, 'A')</code>
Promotion	<code>Compute(1, 1.5, 'A')</code>

`true` is promoted to *int* value (1)  
`1.5f` (a float value) is promoted to double value 1.5

# Method overloading

## Overload resolution steps:

3. Check if a numerical conversion is possible (convert a type into another one with the possibility of losing the actual value / precision).

Defined	<code>void Compute(int x, double y, char z)</code>
Called	<code>Compute(3.5, 1.5, 'A')</code>
Conversion	<code>Compute(3, 1.5, 'A')</code>

3.5 (a double value) will be converted to *int* value 3 (losing precision)

It is possible that the conversion may apply to several overloaded methods. If this is a case, an ambiguity error will be thrown, and the program will not compile

# Method overloading

## Overload resolution steps:

### 4. Casts are attempted:

- ✓ Every non-const pointer can be casted to its const pointer form
- ✓ Every non-const pointer can be casted to *void \** or *const void \**
- ✓ Every const pointer can be casted to *const void \**
- ✓ *NULL* macro (define) can be converted to numerical value *0*

Defined	<code>void Compute(int x, const void* y, char z)</code>
Called	<code>Compute(NULL, "C++", 'A')</code>
Cast	<code>Compute(0, (const void*)"C++", 'A')</code>

NULL is converted to *int* value 0  
"C++" (a *const char \** pointer) is cast to *const void \**

# Method overloading

## **Overload resolution** steps:

5. Explicit casts (if any) are applied. We will discuss more on this topic when we will study inheritance and C++ operators.
6. If none of these attempts result in finding a match - a fallback method / function (if any) is used. A fallback method is a method that only has variadic parameters
7. If there isn't such a method , the compiler will produce an error.

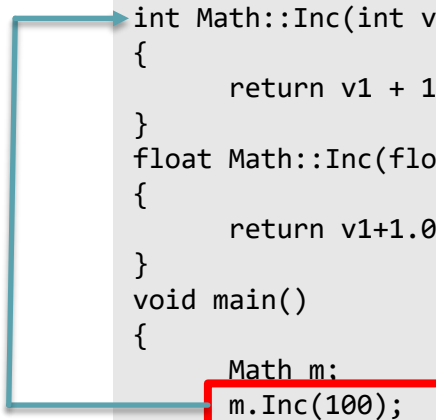


# Method overloading

- ▶ 100 is considered an “*int*” type value. Since there is a method by the name *Inc* that has a parameter of type “*int*”, the compiler will use that method.
- ▶ In this case we have an exact-match situation.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    float Inc(float v1);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
float Math::Inc(float v1)
{
    return v1+1.0f;
}
void main()
{
    Math m;
    m.Inc(100);
}
```

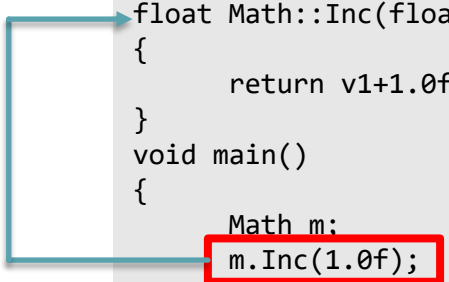
A diagram consisting of a teal line that starts from the left, points to the `Inc(int v1)` method definition, then extends horizontally to the right and points down to the `m.Inc(100);` call in the `main` function. The `m.Inc(100);` line is also enclosed in a red rectangular box.

# Method overloading

- ▶ 1.0f is a “*float*” value. Since there is a method by the name *Inc* that has a parameter of type “*float*”, the compiler will use that method.
- ▶ In this case we have an exact-match situation.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    float Inc(float v1);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
float Math::Inc(float v1)
{
    return v1+1.0f;
}
void main()
{
    Math m;
    m.Inc(1.0f);
}
```

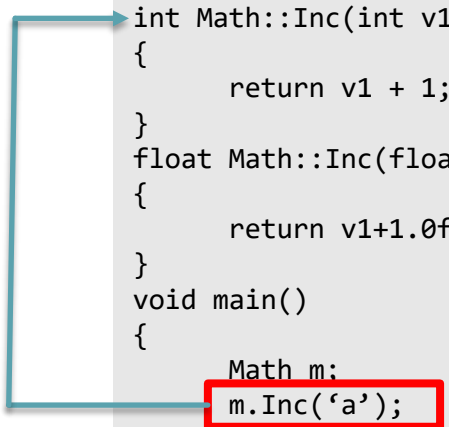
A diagram consisting of a teal line that starts from the left, points to the `m.Inc(1.0f);` line in the `main()` function, and then branches to point to the `float Math::Inc(float v1)` method definition, illustrating the compiler's selection of the correct overloaded method based on the argument type.

# Method overloading

- ▶ 'a' is a "**char**" type value. As there is no method with the name *Inc* that has one parameter of type "char", the compiler promotes the **char** value to **int** and uses the *Inc* method with one parameter of type "**int**".

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    float Inc(float v1);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
float Math::Inc(float v1)
{
    return v1+1.0f;
}
void main()
{
    Math m;
    m.Inc('a');
}
```

A diagram consisting of a teal line that starts from the left, points to the `m.Inc('a');` line in the `main` function, then branches upwards and to the right to point at the `int Math::Inc(int v1)` method definition. This illustrates how the compiler resolves the call to the `int` version of the `Inc` method because the `char` argument 'a' is promoted to `int`.

# Method overloading

- ▶ 'a' is a "**char**" type value. As there is no method with the name **Inc** that has one parameter of type "char", the compiler promotes the **char** value to **int** and tries again. Since there is no **Inc** method that has an int parameter (but there are two **Inc** methods, an ambiguity case will be declared, and the code will not compile. Even if, a **char** can fully be converted (without any value lost) into a **short**, promotion only works for **int** and **double** types.

## App.cpp

```
class Math
{
public:
    int Inc(short v1);
    float Inc(float v1);
};
int Math::Inc(short v1)
{
    return v1 + 1;
}
float Math::Inc(float v1)
{
    return v1+1.0f;
}
void main()
{
    Math m;
    m.Inc('a');
}
```

# Method overloading

- ▶ If during the promotion phase the compiler DOES NOT find any possible promotion, but there are at least **two** methods/functions with the same name as the one attempted to be promoted the compiler will throw an error (this will be considered to be an ambiguity case). Having at least two methods with the same name is an indicator that method overloading is desired and another overload for the specific call is required.
- ▶ However, if the promotion fails and there only **ONE** method with that name, a conversion is attempted (in this case it is considered that method overloading was not something desired by the programmer and the compiler attempts to match the parameters even if this means losing precision / value).

# Method overloading

- ▶ 1.0 is a double value. As there is not any *Inc* method that receives a *double* parameter, promotion is attempted. Unfortunately - can not be promoted (without losing value) to either *int* or *float*.
- ▶ Since there are two *Inc* function, this code will not compile, and an ambiguity case will be explained as an error.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    float Inc(float v1);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
float Math::Inc(float v1)
{
    return v1+1.0f;
}
void main()
{
    Math m;
    m.Inc(1.0);
}
```

# Method overloading

- ▶ In this case, 1.0 is a double value, and as there is no *Inc* function that has one parameter of type double, promotion is attempted. Unfortunately, **double** can not be converted to **char** without losing precision.
- ▶ However, as there is only **ONE** function Inc, the compiler will convert double to char (even if this means losing precision). This code will compile with warnings.

## App.cpp

```
class Math
{
public:
    int Inc(char v1) {
        return v1 + 1;
    }
};

void main()
{
    Math m;
    m.Inc(1.0);
}
```

**warning C4244: 'argument':  
conversion from 'double' to  
'char', possible loss of data**

# Method overloading

- ▶ In this case, 1.0 is a double value, and as there is no *Inc* function that has one parameter of type double, promotion is attempted. Unfortunately, *double* can not be converted to *char* without losing precision.
- ▶ However, as there is only *ONE* function Inc, the compiler will try to convert *double* to *char \**. As this is not possible (only conversions numerical conversions are possible) the compiler will produce an error and the code will not compile.

## App.cpp

```
class Math
{
public:
    int Inc(char* v1) { return 1; }
};
void main()
{
    Math m;
    m.Inc(1.0);
}
```

**error C2664: 'int Math::Inc(char \*)':  
cannot convert argument 1 from  
'double' to 'char \*'**



# Method overloading

- ▶ Pointer conversions are also impossible. “&d” is a “*double \**” that can not be converted to “*char \**”.
- ▶ This code will produce a compiler error.

## App.cpp

```
class Math
{
public:
    int Inc(char* v1);
};
int Math::Inc(char* v1)
{
    return 1;
}
void main()
{
    Math m;
    double d = 1.0;
    m.Inc(&d);
}
```

**error C2664: 'int Math::Inc(char \*)':  
cannot convert argument 1 from  
'double \*' to 'char \*'**

# Method overloading

- ▶ Pointer conversions are also impossible. “&d” is a “*double \**” that can not be converted to “*char \**”.
- ▶ However, using an **explicit cast** will solve this problem. In this case, the code will compile.

## App.cpp

```
class Math
{
public:
    int Inc(char* v1);
};
int Math::Inc(char* v1)
{
    return 1;
}
void main()
{
    Math m;
    double d = 1.0;
    m.Inc( (char *)&d );
}
```

# Method overloading

- ▶ However, any non-constant pointer can be converted to “**void \***”. The next example will compile.
- ▶ A constant pointer can not be converted to a non-constant pointer implicitly (without a cast). A non-constant pointer can always be converted to its constant equivalence. That is why, if you don't need to modify the value where the pointer points, it is best to use **const** pointers for method/function parameters.

## App.cpp

```
class Math
{
public:
    int Inc(void* v1);
};
int Math::Inc(void* v1)
{
    return 1;
}
void main()
{
    Math m;
    double d = 1.0;
    m.Inc(&d);
}
```

# Method overloading

## ► Methods with variadic parameters:

1. **Fallback methods** → methods with only one parameter that is variadic (with a signature in the form **<name> (...)**). These methods are the last to be used (only if there is no possible conversion, cast or promotion or if there is no ambiguity in terms of promotion/conversion. These functions are not allowed in C language.
2. **Regular methods** → methods that have at least one parameter that is not variadic and **ONE** variadic parameter (e.g. **<name>(int,...)** or **<name>(char,short,...)** ). These methods are used just like the regular methods and the same rules apply to them as well.

# Method overloading

- ▶ In case of methods with variadic parameters the compiler will use the best fit (in terms of exact parameters). This code compiles and the method *Inc(int)* is used.

## App.cpp

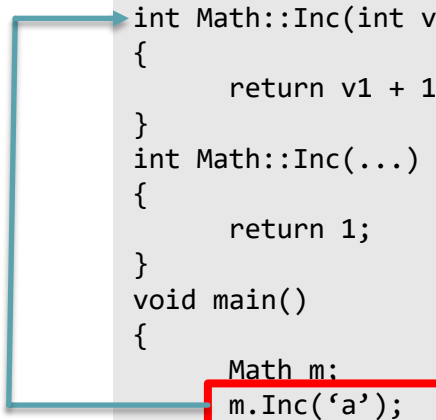
```
class Math
{
public:
    int Inc(int v1);
    int Inc(...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc(123);
}
```

# Method overloading

- ▶ In this case, as there is no *Inc* method that has a parameter of type *char*, the compiler promotes ``a`` (char value **97**) to *int* and uses the *Inc(int)* method.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc('a');
}
```

A diagram consisting of a teal line that starts from the left, points to the `Inc(int v1)` method definition, then extends horizontally to the right and points down to the `m.Inc('a');` call in the `main` function. A red rectangle highlights the `m.Inc('a');` line.

# Method overloading

- ▶ 1.0 is a double value. We can not apply numerical promotion to int to use *Inc(int)* method. However, we can convert the double to an int (with possible loss of value) and then use *Inc(int)* method. The code compiles. The fallback function is used only if no promotion/conversion is possible.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc(1.0);
}
```

# Method overloading

- ▶ This is an ambiguous case. There is no promotion possible. However, the double value 1.0 can be converted to both *int* (and use *Inc(int)* method, or *float* and use *Inc(float)* method). As there are two possibilities, this code is considered ambiguous and an error is thrown.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(float v1);
    int Inc(...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(float v1)
{
    return v1 + 1;
}
int Math::Inc(...)
{
    return 1;
}
void main() {
    Math m;
    m.Inc(1.0);
}
```

error C2668: 'Math::Inc': ambiguous call to overloaded function  
note: could be 'int Math::Inc(float)'  
note: or 'int Math::Inc(int)'  
note: while trying to match the argument list '(double)'



# Method overloading

- ▶ In this case the parameter used is a *const char \** (a pointer). There is no promotion and no conversion possible. Thus, the compiler must use the fallback method *Inc(...)*

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc("test");
}
```

# Method overloading

- ▶ A similar case → there is no method overloaded with 2 parameters, so the compiler uses the fallback method *Inc(...)*

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc(1.0, 2);
}
```

# Method overloading

- ▶ This is an ambiguous case. 123 is an *int* value and there are two methods that match exactly with the call *Inc(123) : Inc(int)* and *Inc(int,...)*. The code will NOT compile.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(int v1,...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(int v1,...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc(123);
}
```

warning C4326: return type of 'main' should be 'int' instead of 'void'  
error C2668: 'Math::Inc': ambiguous call to overloaded function  
note: could be 'int Math::Inc(int,...)'  
note: or 'int Math::Inc(int)'  
note: while trying to match the argument list '(int)'

# Method overloading

- ▶ This is an ambiguous case. *true* is a *bool* value and we don't have an exact method to match *Inc(bool)*. In this case numerical promotion is apply, *bool* is promoted to *int* and now we have two methods that match: *Inc(int)* and *Inc(int,...)*. The code will NOT compile.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(int v1,...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(int v1,...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc(true);
}
```

error C2668: 'Math::Inc': ambiguous call to overloaded function  
note: could be 'int Math::Inc(int,...)'  
note: or 'int Math::Inc(int)'  
note: while trying to match the argument list '(bool)'

# Method overloading

- ▶ This code will NOT compile. None of the methods *Inc(int)* and *Inc(int,...)* matches the *const char \** parameter and there is no fallback method.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(int v1,...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(int v1,...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc("test");
}
```

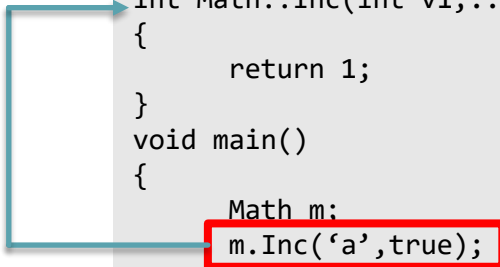
error C2664: 'int Math::Inc(int,...)': cannot convert argument 1 from 'const char [5]' to 'int'  
note: There is no context in which this conversion is possible

# Method overloading

- ▶ This code will compile. 'a' (*char*) is promoted to *int* and since there is only one method that accepts two parameters, the compiler will use it.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(int v1,...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(int v1,...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc('a',true);
}
```

A diagram consisting of a teal L-shaped line. The vertical segment is on the left, and the horizontal segment points to the call `m.Inc('a',true);` in the `main` function. This call is enclosed in a red rectangular box. The diagram illustrates that the compiler resolves the call to the two-parameter `Inc` method.

# Method overloading

- ▶ This code will compile. 'a' (*char*) is promoted to *int* and since there is only one method that accepts two parameters, the compiler will use it.
- ▶ Fallback methods (*Inc(...)*) are used only if no match is possible.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(int v1,...);
    int Inc(...);
};
int Math::Inc(int v1) {
    return v1 + 1;
}
int Math::Inc(int v1,...) {
    return 1;
}
int Math::Inc(...) {
    return 2;
}
void main()
{
    Math m;
    m.Inc('a',true);
}
```

# Method overloading

- ▶ This code will compile. However, in this case there is no match possible from *Inc(const char \*, bool)* (including promotions and conversions) to the existing methods *Inc(int)* and *Inc(int,...)*. However, as a fallback function is also available, the compiler will choose to use it.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(int v1,...);
    int Inc(...);
};
int Math::Inc(int v1) {
    return v1 + 1;
}
int Math::Inc(int v1,...) {
    return 1;
}
int Math::Inc(...) {
    return 2;
}
void main()
{
    Math m;
    m.Inc("test",true);
}
```



# Method overloading

- ▶ When dealing with overloaded methods with multiple parameters, promotion and conversion rules are evaluated for each parameter.
- ▶ **Overload resolution** will choose the combination of promotion/conversions that covers highest number of unique parameters.
- ▶ If there are at least two solutions that have at least one different parameter that each one of the solution can cover (match / promote or convert) an ambiguous case is considered and an error will be thrown.
- ▶ If nothing matches, a promotion is considered stronger than a conversions , and can be used to resolve overload resolution (in fact a promotion is sometimes considered equal as importance with an exact match).

# Method overloading

- ▶ This code will compile. There is an exact match (a function *Add* with two parameters, first of type *char* and the second of type *int*).

## App.cpp

```
class Math
{
public:
    void Add(char x, int y);
    void Add(int x, char y);
};
void Math::Add(char x, int y)
{
    printf("Add(char,int)");
}
void Math::Add(int x, char y)
{
    printf("Add(int,char)");
}
void main()
{
    Math m;
    m.Add('a',100);
}
```

# Method overloading

- ▶ This code compiles. The compiler promotes the second parameter from *bool* to *int* and uses *Add(char,int)*

## App.cpp

```
class Math
{
public:
    void Add(char x, int y);
    void Add(int x, char y);
};
void Math::Add(char x, int y)
{
    printf("Add(char,int)");
}
void Math::Add(int x, char y)
{
    printf("Add(int,char)");
}
void main()
{
    Math m;
    m.Add('a',true);
}
```

# Method overloading

- ▶ This is an ambiguous case as we have two possibilities:
  - a.  $100 = \text{int}$  , we convert  $200(\text{int})$  to *char* and use *Add(int,char)*
  - b.  $100(\text{int})$  is converted to *char*,  $200$  is considered an *int* and we use *Add(char,int)*

## App.cpp

```
class Math
{
public:
    void Add(char x, int y);
    void Add(int x, char y);
};
void Math::Add(char x, int y)
{
    printf("Add(char,int)");
}
void Math::Add(int x, char y)
{
    printf("Add(int,char)");
}
void main()
{
    Math m;
    m.Add(100,200);
}
```

error C2666: 'Math::Add': 2 overloads have similar conversions  
note: could be 'void Math::Add(int,char)'  
note: or 'void Math::Add(char,int)'  
note: while trying to match the argument list '(int, int)'

# Method overloading

- ▶ This code compiles. We also have two possibilities, but the first one is better:
  - a. 100 = *int*, 1.5 is converted to *char* and we use *Add(int,char)* [ one conversion + one exact match]
  - b. 100 (int) is converted to *char*, 1.5 to *int* and we use *Add(char,int)* [ two conversions ]

## App.cpp

```
class Math
{
public:
    void Add(char x, int y);
    void Add(int x, char y);
};
void Math::Add(char x, int y)
{
    printf("Add(char,int)");
}
void Math::Add(int x, char y)
{
    printf("Add(int,char)");
}
void main()
{
    Math m;
    m.Add(100,1.5);
}
```

# Method overloading

- ▶ This code compiles. We also have three possibilities:
  - a. 'a' = char (exact match), 1.5 (a double is converted to int) and 2.5 (a double is converted to int) → **1 x exact match, 2 x conversion**
  - b. 'a' ( char is promoted to int), 1.5 (a double is converted to char) and 2.5 (a double is converted to int) → **1 x promotion, 2 x conversion**
  - c. 'a' ( char is converted to float), 1.5 (a double is converted to bool) and 2.5 (a double is converted to int) → **3 x conversion**
- ▶ Solution a) has an exact match and since there is no other solutions that can match another parameter than the first one, this will be selected.

## App.cpp

```
class Math
{
public:
    int Add(char x, int y, int z) { return 1; }
    int Add(int x, char y, int z) { return 2; }
    int Add(float x, bool y, int z) { return 3; }
};

void main()
{
    Math m;
    int x = m.Add('a', 1.5, 2.5);
}
```

# Method overloading

- ▶ This code will NOT compile. We also have three possibilities:
  - a. 'a' = char (exact match), 1.5f (a float is converted to int) and 2.5 (a double is converted to int) → **1 x exact match, 2 x conversion**
  - b. 'a' (char is promoted to int), 1.5f (a float is converted to char) and 2.5 (a double is converted to int) → **1 x promotion, 2 x conversion**
  - c. 'a' = char (exact match), 1.5f (a float is converted to bool) and 2.5 (a double is converted to int) → **1 x exact match, 2 x conversion**
- ▶ Since both solutions a) and c) have an exact match for the 1<sup>st</sup> parameter, this will be considered an ambiguity case.

## App.cpp

```
class Math
{
public:
    int Add(char x, int y, int z) { return 1; }
    int Add(int x, char y, int z) { return 2; }
    int Add(char x, bool y, int z) { return 3; }
};

void main()
{
    Math m;
    int x = m.Add('a', 1.5f, 2.5);
}
```

error C2668: 'Math::Add': ambiguous call to overloaded function  
note: could be 'int Math::Add(char,bool,int)'  
note: or 'int Math::Add(char,int,int)'  
note: while trying to match the argument list '(char, float, double)'

# Method overloading

- ▶ This code will NOT compile. We also have three possibilities:
  - a. 'a' = char (exact match), 'a' (a char is promoted to int) and 2.5 (a double is converted to int) → **1 x exact match, 1 x conversion, 1 x promotion**
  - b. 'a' (char is promoted to int), 'a' = char (exact match) and 2.5 (a double is converted to int) → **1 x exact match, 1 x conversion, 1 x promotion**
  - c. 'a' = char (exact match), 'a' (a char is converted to bool) and 2.5 (a double is converted to int) → **1 x exact match, 2 x conversion**
- ▶ Solution a) and c) have an exact match for the 1<sup>st</sup> parameter (but not for the second one). Solution b) has an exact match for the 2<sup>nd</sup> parameter (but not for the first one). This is considered an ambiguity.

## App.cpp

```
class Math
{
public:
    int Add(char x, int y, int z) { return 1; }
    int Add(int x, char y, int z) { return 2; }
    int Add(char x, bool y, int z) { return 3; }
};

void main()
{
    Math m;
    int x = m.Add('a', 'a', 2.5);
}
```

**error C2666: 'Math::Add': 3 overloads have similar conversions**  
note: could be 'int Math::Add(char,bool,int)'  
note: or 'int Math::Add(int,char,int)'  
note: or 'int Math::Add(char,int,int)'  
note: while trying to match the argument list '(char, char, double)'



# Method overloading

- ▶ Let's consider the following code:

## App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, char z) { return 3; }
};
```

- ▶ and let's consider that we call Math::Add with the following parameters:

Math m; m.Add(1.5, true, 1.5)

#		X (double)	Y (bool)	Z (double)
1.	Add ( <i>char</i> x, <i>int</i> y, <i>int</i> z)	conversion	promotion	conversion
2.	Add ( <i>double</i> x, <i>int</i> y, <i>int</i> z)	exact match	promotion	conversion
3.	Add ( <i>char</i> x, <i>bool</i> y, <i>char</i> z)	conversion	exact match	conversion

- ▶ Solution 1 (has an exact match for the first parameter), Solution 2 has an exact match for the second parameter → **ambiguity case**

# Method overloading

- ▶ Let's consider the following code:

## App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, char z) { return 3; }
};
```

- ▶ and let's consider that we call Math::Add with the following parameters:

Math m; m.Add(false, true, 1.5)

#		X (bool)	Y (bool)	Z (double)
1.	Add ( <i>char</i> x, <i>int</i> y, <i>int</i> z)	conversion	promotion	conversion
2.	Add ( <i>double</i> x, <i>int</i> y, <i>int</i> z)	conversion	promotion	conversion
3.	Add ( <i>char</i> x, <i>bool</i> y, <i>char</i> z)	conversion	exact match	conversion

- ▶ One solution that has a match (solution 3). There is no promotion or exact match for “X” or “Z” in solution 1 or 2 → code will compile

# Method overloading

- ▶ Let's consider the following code:

## App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, char z) { return 3; }
};
```

- ▶ and let's consider that we call Math::Add with the following parameters:

Math m; m.Add(1.5, true, 1.5)

#		X (double)	Y (bool)	Z (double)
1.	Add ( <i>char</i> x, <i>int</i> y, <i>int</i> z)	conversion	promotion	conversion
2.	Add ( <i>double</i> x, <i>int</i> y, <i>int</i> z)	exact match	promotion	conversion
3.	Add ( <i>char</i> x, <i>bool</i> y, <i>char</i> z)	conversion	exact match	conversion

- ▶ Solution 1 (has an exact match for the first parameter), Solution 2 has an exact match for the second parameter → **ambiguity case**

# Method overloading

- ▶ Let's consider the following code:

## App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, char z) { return 3; }
};
```

- ▶ and let's consider that we call Math::Add with the following parameters:

Math m; m.Add('a', true, 1.5)

#		X (char)	Y (bool)	Z (double)
1.	Add ( <i>char</i> x, <i>int</i> y, <i>int</i> z)	exact match	promotion	conversion
2.	Add ( <i>double</i> x, <i>int</i> y, <i>int</i> z)	conversion	promotion	conversion
3.	Add ( <i>char</i> x, <i>bool</i> y, <i>char</i> z)	exact match	exact match	conversion

- ▶ Solution 3 has two exact matches (for X and Y), solution 1 has one match (just for X). As solution 3 covers solution 1, and solution 2 does not have a promotion or exact match for “Z”, code will compile and solution 3 will be selected

# Method overloading

- ▶ Let's consider the following code:

## App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, char z) { return 3; }
};
```

- ▶ and let's consider that we call Math::Add with the following parameters:

Math m; m.Add('a', true, 100)

#		X (char)	Y (bool)	Z (int)
1.	Add ( <i>char</i> x, <i>int</i> y, <i>int</i> z)	exact match	promotion	exact match
2.	Add ( <i>double</i> x, <i>int</i> y, <i>int</i> z)	conversion	promotion	exact match
3.	Add ( <i>char</i> x, <i>bool</i> y, <i>char</i> z)	exact match	exact match	conversion

- ▶ Solution 3 matches parameters 1 and 2, solution 1 matches parameters 1 and 3 (there is no clear solution) → **ambiguity case**

# Method overloading

- ▶ Let's consider the following code:

## App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, char z) { return 3; }
};
```

- ▶ and let's consider that we call Math::Add with the following parameters:

Math m; m.Add(1.5, true, 'a')

#		X (double)	Y (bool)	Z (char)
1.	Add ( <i>char</i> x, <i>int</i> y, <i>int</i> z)	conversion	promotion	promotion
2.	Add ( <i>double</i> x, <i>int</i> y, <i>int</i> z)	exact match	promotion	promotion
3.	Add ( <i>char</i> x, <i>bool</i> y, <i>char</i> z)	conversion	exact match	exact match

- ▶ Solution 3 matches parameters 2 parameters (Y and Z) but not parameter “X”. Since there is also a solution that could match parameter “X” (solution 2) this will be considered an **ambiguity case**.

# Method overloading

- ▶ Let's consider the following code:

## App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, char z) { return 3; }
};
```

- ▶ and let's consider that we call `Math::Add` with the following parameters:

`Math m; m.Add(100, 1.5, 1.5)`

#		X (int)	Y (double)	Z (double)
1.	Add ( <i>char</i> x, <i>int</i> y, <i>int</i> z)	conversion	conversion	conversion
2.	Add ( <i>double</i> x, <i>int</i> y, <i>int</i> z)	conversion	conversion	conversion
3.	Add ( <i>char</i> x, <i>bool</i> y, <i>char</i> z)	conversion	conversion	conversion

- ▶ No cases with exact match, all solutions have 3 conversions → **ambiguity case**

# Method overloading

- ▶ Let's consider the following code:

## App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, char z) { return 3; }
};
```

- ▶ and let's consider that we call Math::Add with the following parameters:

Math m; m.Add(1.0f, 1.5, 1.5)

#		X (float)	Y (double)	Z (double)
1.	Add ( <i>char</i> x, <i>int</i> y, <i>int</i> z)	conversion	conversion	conversion
2.	Add ( <i>double</i> x, <i>int</i> y, <i>int</i> z)	<b>promotion</b>	conversion	conversion
3.	Add ( <i>char</i> x, <i>bool</i> y, <i>char</i> z)	conversion	conversion	conversion

- ▶ No cases with exact match, however there is one case that has an accepted promotion while the rest only have conversions. No promotion/exact match for “Y” and “Z” for Solution 1 and 3. Solution 2 is selected and code compiles.



# Method overloading

- ▶ Let's change the previous definitions a little bit:

## App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, double z) { return 3; }
};
```

- ▶ and let's consider that we call Math::Add with the following parameters:

Math m; m.Add(1.0f, 1.5, 1.0f)

#		X (float)	Y (double)	Z (float)
1.	Add ( <i>char</i> x, <i>int</i> y, <i>int</i> z)	conversion	conversion	conversion
2.	Add ( <i>double</i> x, <i>int</i> y, <i>int</i> z)	<b>promotion</b>	conversion	conversion
3.	Add ( <i>char</i> x, <i>bool</i> y, <i>double</i> z)	conversion	conversion	<b>promotion</b>

- ▶ Solution 2 is valid for “X” (due to promotion), Solution 3 is valid for “Z” (due to promotion). → **ambiguity case**

# Method overloading

- ▶ Let's change the previous definitions a little bit:

## App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, double z) { return 3; }
};
```

- ▶ and let's consider that we call Math::Add with the following parameters:

Math m; m.Add(1.0f, 1.5, 1.5)

#		X (float)	Y (double)	Z (double)
1.	Add ( <i>char</i> x, <i>int</i> y, <i>int</i> z)	conversion	conversion	conversion
2.	Add ( <i>double</i> x, <i>int</i> y, <i>int</i> z)	<b>promotion</b>	conversion	conversion
3.	Add ( <i>char</i> x, <i>bool</i> y, <i>double</i> z)	conversion	conversion	<b>exact match</b>

- ▶ This is a case where promotion and exact match are seen as equals. Solution 2 is valid for “X” (due to promotion), Solution 3 is valid for “Z” (due to exact match). → **ambiguity case**

# Method overloading

- ▶ Let's change the previous definitions a little bit:

## App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, double z) { return 3; }
};
```

- ▶ and let's consider that we call `Math::Add` with the following parameters:

`Math m; m.Add(1.0f, 1.5, 100)`

#		X (float)	Y (double)	Z (int)
1.	Add ( <i>char</i> x, <i>int</i> y, <i>int</i> z)	conversion	conversion	exact match
2.	Add ( <i>double</i> x, <i>int</i> y, <i>int</i> z)	<b>promotion</b>	conversion	exact match
3.	Add ( <i>char</i> x, <i>bool</i> y, <i>double</i> z)	conversion	conversion	conversion

- ▶ Solution 2 covers “X” (due to promotion) and “Z” (due to exact match). There is no other solution better, or one that can cover “Y” → Solution 2 is selected and the code compiles.

# Method overloading

- ▶ When dealing with the **const** keyword there are also some differences in terms of method overloading and overload resolution
- ▶ For numerical types (types that are transmitted to a method by value) **const** is ignored from the method / function signature
- ▶ For pointers and references, **const** is used in the method / function signature.

# Method overloading

- ▶ This case will NOT compile - but not due to an ambiguity problem, but rather to the fact that both *Inc(int)* and *Inc(const int)* are considered to have the same signature: *Inc(int)*

## App.cpp

```
class Math
{
public:
    int Inc(int x) { return x + 2; }
    int Inc(const int x) { return x + 1; }
};

void main()
{
    Math m;
    int x = 10;
    m.Inc(x);
}
```

error C2535: 'int Math::Inc(int)': member function already defined or declared  
note: see declaration of 'Math::Inc'

# Method overloading

- ▶ In this case , the two *Inc* methods are considered to have a different signature and therefor are used in the overload resolution. As “&d” is an *int \** than the best match ( meaning *Inc(int \*)* will be chosen). The code compiles.

## App.cpp

```
class Math
{
public:
    int Inc(int * x)
    {
        return *x + 2;
    }

    int Inc(const int * x)
    {
        return *x + 1;
    }
};

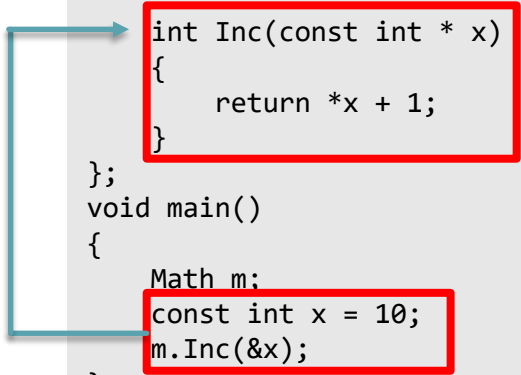
void main()
{
    Math m;
    int x = 10;
    m.Inc(&x);
}
```

# Method overloading

- ▶ Similarly, if we change “x” local variable from main function to be a constant, the second function *Inc(const int \*)* will be chosen as a perfect match.

## App.cpp

```
class Math
{
public:
    int Inc(int * x)
    {
        return *x + 2;
    }
    int Inc(const int * x)
    {
        return *x + 1;
    }
};
void main()
{
    Math m;
    const int x = 10;
    m.Inc(&x);
}
```



# Method overloading

- The same logic applies for references as well.

## App.cpp

```
class Math
{
public:
    int Inc(int & x)
    {
        return x + 2;
    }
    int Inc(const int & x)
    {
        return x + 1;
    }
};
void main()
{
    Math m;
    const int x = 10;
    m.Inc(x);
}
```

## App.cpp

```
class Math
{
public:
    int Inc(int & x)
    {
        return x + 2;
    }
    int Inc(const int & x)
    {
        return x + 1;
    }
};
void main()
{
    Math m;
    int x = 10;
    m.Inc(x);
}
```

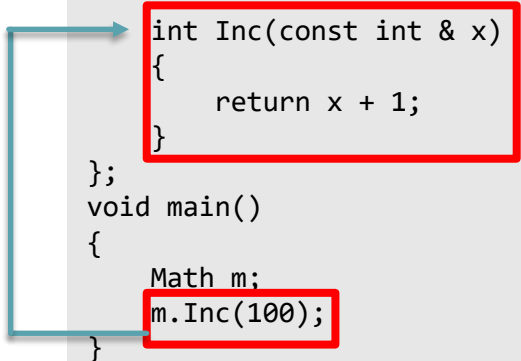


# Method overloading

- ▶ The same logic applies for references as well.
- ▶ In particular, when dealing with constant numerical values they will always be translated into a const reference.

## App.cpp

```
class Math
{
public:
    int Inc(int & x)
    {
        return x + 2;
    }
    int Inc(const int & x)
    {
        return x + 1;
    }
};
void main()
{
    Math m;
    m.Inc(100);
}
```



# Method overloading

- ▶ In this case the code will not compile as a constant (const) value CAN NOT be converted to a non-constant value.

## App.cpp

```
class Math
{
public:
    int Inc(int & x)
    {
        return x + 2;
    }
};

void main()
{
    Math m;
    m.Inc(100);
}
```

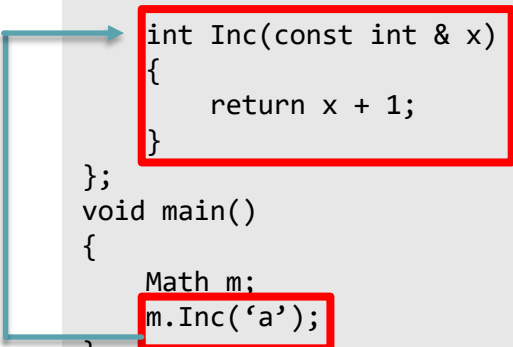
**error C2664: 'int Math::Inc(int &)': cannot convert argument 1 from 'int' to 'int &'**

# Method overloading

- ▶ The rest of the promotion / conversion rules apply.
- ▶ In this example, 'a' is of type **char**. As there is no **Inc** method that receives a char parameter, 'a' will be promoted to an **int** and then to a **const int &**. This code will compile. 'a' is a constant → and not a variable. As such it will be linked to a const reference (as you can not change its value).

## App.cpp

```
class Math
{
public:
    int Inc(int & x)
    {
        return x + 2;
    }
    int Inc(const int & x)
    {
        return x + 1;
    }
};
void main()
{
    Math m;
    m.Inc('a');
}
```

A blue arrow originates from the 'a' in the call m.Inc('a') within the main() function and points to the signature of the second Inc method, int Inc(const int & x), which is highlighted with a red rectangular box.



► NULL pointer

# NULL pointer

- ▶ Let's consider the following code:

## App.cpp

```
void Print(int value)
{
    printf("Number: %d\n", value);
}
void Print(const char* text)
{
    printf("Text: %s\n", text);
}

void main()
{
    Print(10);
    Print("C++ test");

    Print(NULL);
}
```

- ▶ The code compiles correctly. What is the output of this code ?

# NULL pointer

- ▶ Let's consider the following code:

**App.cpp**

```
void Print(int value)
{
    printf("Number: %d\n", value);
}
void Print(const char* text)
{
    printf("Text: %s\n", text);
}

void main()
{
    Print(10);
    Print("C++ test");
    Print(NULL);
}
```

**Output**  
Number: 10  
Text: C++ test  
Number: 0

- ▶ Why the last call of Print function is considered to be a number ?

# NULL pointer

- ▶ Let's consider the following code:

**App.cpp**

```
void Print(int value)
{
    printf("Number: %d\n", value);
}
void Print(const char* text)
{
    printf("Text: ");
}
void main()
{
    Print(10);
    Print("C++ t
    Print(NULL);
}
```

**Preprocessor directives:**

```
#ifndef NULL
#ifdef __cplusplus
#define NULL 0
#else
#define NULL ((void *)0)
#endif
#endif
```

- ▶ Why the last call of Print function is considered to be a number ?

# NULL pointer

- ▶ So - NULL is defined as a number. While during promotion, value 0 can be translated into a NULL pointer, there are often cases (similar to previous one) where the intended parameter is a pointer ( a NULL pointer ) and not a number.
- ▶ The solution was to create a new constant (keyword) that refers only to null pointers. This constant is called **nullptr**

## App.cpp

```
void Print(int value) { ... }  
void Print(const char* text) { ... }  
  
void main()  
{  
    Print(nullptr);  
}
```

- ▶ In the previous example, the compiler will now call “Print(const char\*)” function.



# NULL pointer

- ▶ The following assignments are valid for NULL constant and all variable will be set to 0, false or a null pointer.

## App.cpp

```
void main()
{
    int x = NULL;
    char y = NULL;
    float f = NULL;
    bool b = NULL;
    const char* p = NULL;
    int * i = NULL;
}
```

- ▶ The following assignments are invalid (code will NOT compile):

## App.cpp

```
void main()
{
    int x = nullptr;
    char y = nullptr;
    float f = nullptr;
}
```

# NULL pointer

- ▶ The following assignments are valid and the code will compile.

## App.cpp

```
void main()
{
    bool b = nullptr;
    const char* p = nullptr;
    int * i = nullptr;
}
```

- ▶ Keep in mind that **nullptr** can still be used as a bool value (equal to **false**). However, even if this cast is possible, **nullptr** will always chose a pointer to a bool. The following example works and does not yield any ambiguity:

## App.cpp

```
void Print(bool value) { ... }
void Print(const char* text) { ... }

void main()
{
    Print( nullptr );
}
```

The compiler will choose to call "Print (const char\* )" function

# NULL pointer

- ▶ However, the following example will produce an ambiguity and the code will not compile:

## App.cpp

```
void Print(bool value) { ... }  
void Print(const char* text) { ... }  
void Print(int* value) { ... }  
  
void main()  
{  
    Print( nullptr );  
}
```

- ▶ The compiler will yield an error that states that it does not know what to choose for the call of “Print (nullptr)” and that it has two possible variants to choose from.



▶ “const” specifier

# Classes (methods) - the "const" specifier

- ▶ Whenever a method is declared within a class, a special keyword can also be used to specify a certain behavior for that method: **"const"**
- ▶ The following code compiles without problem. At the end of execution member x from object "d" will be 1;

## App.cpp

```
class Date
{
    private:
        int x;
    public:
        int& GetX();
};
int& Date::GetX()
{
    x = 0;
    return x;
}
void main()
{
    Date d;
    d.GetX()++;
}
```

# Classes (methods) - the "const" specifier

- ▶ This code will not compile because GetX function () returns a constant reference to a number. This means that the operator "++" from "d.GetX () ++" has to modify a number that is considered constant.

## App.cpp

```
class Date
{
    private:
        int x;
    public:
        const int& GetX();
};
const int& Date::GetX()
{
    x = 0;
    return x;
}
void main()
{
    Date d;
    d.GetX()++;
}
```

# Classes (methods) - the "const" specifier

- ▶ The code compiles. Method GetX () returns a reference to a constant integer whose value is 0. In the main function, “x” maintains a copy of the value returned by GetX() function (a copy that can be modified).
- ▶ This is the recommended solution if we want to give **read-only access** to a member variable ( in particular if it is NOT a basic type ).

## App.cpp

```
class Date
{
    private:
        int x;
    public:
        const int& GetX();
};
const int& Date::GetX()
{
    x = 0;
    return x;
}
void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

# Classes (methods) - the "const" specifier

- ▶ When dealing with pointers or references, “const” specifier can be used in the following ways:

## App.cpp

```
void main()
{
    int x;
    const int * ptr;
    ptr = &x;
    *ptr = 1;
}
```

This code will not compile as *ptr* points to a constant int that CAN NOT BE modified.

- ▶ In the previous example - the const specifier is part of the value. This means that we can modify the pointer (NOT the value) without any issue.

## App.cpp

```
void main()
{
    int x;
    const int * ptr;
    ptr = &x;
    ptr += 1;
}
```

This code will run as we DO NOT modify the actual value, we just modify the pointer.



# Classes (methods) - the "const" specifier

- ▶ When dealing with pointers or references, “const” specifier can be used in the following ways:

## App.cpp

```
void main()
{
    int x;
    int * const ptr;
    ptr = &x;
}
```

This code will not compile as *ptr* is a constant pointer that points towards a non-constant value.

- ▶ In the previous example - the const specifier refers to the pointer and NOT the value it points to.

## App.cpp

```
void main()
{
    int x;
    int * const ptr = &x;
    *ptr = 1;
}
```

This code will run as it initialize the constant pointer from the beginning.

This code will also run. “ptr” pointer points towards a non-const value that can be modified.

# Classes (methods) - the "const" specifier

- ▶ When dealing with pointers or references, “const” specifier can be used in the following ways:

## App.cpp

```
void main()
{
    int x;
    int * const ptr;
    ptr = &x;
}
```

This code will not compile as *ptr* is a constant pointer that points towards a non-constant value.

- ▶ In the previous example - the const specifier refers to the pointer and NOT the value it points to.

## App.cpp

```
void main()
{
    int x;
    int * const ptr = &x;
    ptr += 1;
}
```

This code will run as it initializes the constant pointer from the beginning.

This code will NOT run as we try to modify a constant pointer.

# Classes (methods) - the "const" specifier

- ▶ When dealing with pointers or references, “const” specifier can be used in the following ways:

## App.cpp

```
void main()
{
    int x;
    const int * const ptr = &x;
    *ptr = 1;
    ptr += 1;
}
```

In this case both the pointer and the value it points to are constant. The code will not compile - one can not modify the pointer or the value.

# Classes (methods) - the "const" specifier

“const” specifier respects the **Clockwise/Spiral Rule** for C language.

C/C++ expression	Explanation	Change value	Change Pointer
<code>int * ptr;</code>	Non-const pointer to a non-const value	YES	YES
<code>const int * ptr;</code>	Non-const pointer to a const value	NO	YES
<code>int const * ptr;</code>	Non-const pointer to a const value	NO	YES
<code>int * const ptr;</code>	Const pointer to a non-const value	YES	NO
<code>const int * const ptr;</code>	Const pointer to a const value	NO	NO

In particular, a syntax like “*int \* const ptr*” is equivalent to a reference (*int &*) and “*const int \* const ptr*” to “*const int &*”

# Classes (methods) - the "const" specifier

“const” specifier respects the **Clockwise/Spiral Rule** for C language.

C/C++ expression	Explanation
<code>int ** ptr;</code>	Non-const pointer to a non-const pointer to a non-const value
<code>const int ** ptr;</code>	Non-const pointer to a non-const pointer to a const value
<code>int ** const ptr;</code>	Const pointer to a non-const pointer to a non-const value
<code>int * const * const ptr;</code>	Const pointer to a const-pointer to a non-const value
<code>const int * const * const ptr;</code>	Const pointer to a const-pointer to a const value

# Classes (methods) - the "const" specifier

- ▶ This code will not compile. The usage of “const” keyword at the end of the method declaration specifies that within that method data members of that class **can not** be modified. In the next example, “x” is a data member from class Date and assigning value 0 to it contradicts the “const” keyword from method definition.

## App.cpp

```
class Date
{
    private:
        int x;
    public:
        const int& GetX() const;
};
const int& Date::GetX() const
{
    x = 0;
    return x;
}
void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

error C3490: 'x' cannot be modified because it is being accessed through a const object

# Classes (methods) - the "const" specifier

- ▶ Let's assume that we have the following code:

## App.cpp

```
class Date
{
private:
    int x;
    int y,z,t;
public:
    const int& GetX() const;
};
const int& Date::GetX() const
{
    x = 0;
    return x;
}
void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

- ▶ And we want to make sure that access to data members “y”, “z” and “t” are read only, but for data member “x” we have read/write access.
- ▶ If we use a “const” function (as define in this example) → “x” will be read-only as well.

# Classes (methods) - the "const" specifier

- ▶ Let's assume that we have the following code:

## App.cpp

```
class Date
{
private:
    mutable int x;
    int y,z,t;
public:
    const int& GetX() const;
};
const int& Date::GetX() const
{
    x = 0;
    return x;
}
void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

- ▶ Starting with C++11 there is a new specifier called “*mutable*” that allows write access to a data member even if “const” specifier is used.
- ▶ This code will compile.



# Classes (methods) - the "const" specifier

- ▶ Let's assume that we have the following code:

## App.cpp

```
class Date
{
private:
    const mutable int * x;
    int y,z,t;
public:
    const int& GetX() const;
};
const int& Date::GetX() const
{
    x = &y;
    return *x;
}
void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

- ▶ “**const**” can be used with “**mutable**”. In the previous example mutable refers to the value of the pointer and does not interfere with the **const** qualifier. This translates that you can modify the pointer (through the **mutable** qualifier) but you can not modify the value (due to the const qualifier at the end of the GetX() method).

# Classes (methods) - the "const" specifier

- ▶ Let's assume that we have the following code:

## App.cpp

```
class Date
{
private:
    const mutable int * const x;
    int y,z,t;
public:
    const int& GetX() const;
};
const int& Date::GetX() const
{
    x = &y;
    return *x;
}
void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

- ▶ This code will not compile as “x” being a const pointer (not a pointer to a const value) can not be mutable at the same time (it will imply that it can be changed). At the same time, “x=&y” can not run as “x” is a const pointer.

# Classes (methods) - the "const" specifier

Usually *mutable* specifier is used when:

- ▶ A class is run in a multi-threaded environment and you need a variable that can be used between multiple threads
- ▶ Lambda expressions
- ▶ As a way to control what data members can be modified within a class from a const method.

# Classes (methods) - the "const" specifier

- The code compiles correctly because "x" is no longer a member of an instance but a global static member (it does not belong to the object).

## App.cpp

```
class Date
{
    private:
        static int x;
    public:
        const int& GetX() const;
};
int Date::x = 100;
const int& Date::GetX() const
{
    x = 0;
    return x;
}
void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

# Classes (methods) - the "const" specifier

- ▶ The code does not compile because the “const” modifier from the end of GetX declaration can not be used for static functions as it needs an instance to apply to (access to **this** pointer that is impossible if the method is declared as **const**)

## App.cpp

```
class Date
{
    private:
        static int x;
    public:
        static const int& GetX() const;
};
int Date::x = 100;
static const int& Date::GetX() const
{
    x = 0;
    return x;
}
void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

# Classes (methods) - the "const" specifier

- ▶ This code compiles. “const” specifier refers to the current object/instance alone. It does not apply to another instance of a different type (it will only apply to the instance represented by “this”).

## App.cpp

```
class Date
{
private:
    int x;
public:
    void ModifyX(Date * d) const
    {
        d->x = 0;
    }
};

void main()
{
    Date d1,d2;
    d1.ModifyX(&d2);
}
```

# Classes (methods) - the "const" specifier

- ▶ This code will NOT compile as a “const” method refers to the current instance (“this” pointer).

## App.cpp

```
class Date
{
private:
    int x;
public:
    void ModifyX(Date * d) const
    {
        this->x = 0;
    }
};

void main()
{
    Date d1,d2;
    d1.ModifyX(&d2);
}
```

# Classes (methods) - the "const" specifier

- ▶ “const” is part of object type
- ▶ A class method/function can not modify a parameters if it is defined as “const”

## Without const specifier

```
class Date
{
private:
    int x;
public:
    void Inc();
};
void Date::Inc()
{
    x++;
}
void Increment(Date &d)
{
    d.Inc();
}
void main()
{
    Date d;
    Increment(d);
}
```

## With const specifier

```
class Date
{
private:
    int x;
public:
    void Inc();
};
void Date::Inc()
{
    x++;
}
void Increment(const Date &d)
{
    d.Inc();
}
void main()
{
    Date d;
    Increment(d);
}
```

Compile error, d is const



# Classes (data members) - the "const" specifier

- ▶ “const” can be used for data members as well. The following code will not compile as the const value is not initialized.

## App.cpp

```
class Data
{
    const int x;
public:
    int GetX() { return x; }
};
void main()
{
    Data d;
}
```

- ▶ To instantiate such a code , a value has to be added in to the const data member in the class definition (more on this topic in the course related to constructors).

## App.cpp

```
class Data
{
    const int x = 10;
public:
    int GetX() { return x; }
};
```

The background of the slide is a solid dark blue. On the left side, there is a decorative graphic consisting of several overlapping, semi-transparent triangles and polygons in various shades of blue, ranging from a very dark navy to a light sky blue. These shapes create a dynamic, layered effect.

▶ “friend” specifier

# “friend” specifier

- ▶ For a class a “friend” function is a function that can access methods and data members that with private modifier define within that class.
- ▶ A “friend” function does not belong to the class (in this case to the Date class). From this point of view access specifier is irrelevant (it doesn't matter if the “friend” function is written in the private or the public section)

## App.cpp

```
class Date
{
    int x;
public:
    Date(int value) : x(value) {}
    void friend PrintDate(Date &d);
};

void PrintDate(Date &d)
{
    printf("X = %d\n", d.x);
}

void main()
{
    Date d1(1);
    PrintDate(d1);
}
```

# “friend” specifier

## App.cpp

```
class Date
{
    int x;
public:
    Date(int value) : x(value) {}
    friend class Printer;
};
class Printer
{
public:
    void PrintDecimal(Date &d);
    void PrintHexazecimal(Date &d);
};
void Printer::PrintDecimal(Date &d)
{
    printf("x = %d\n", d.x);
}
void Printer::PrintHexazecimal(Date &d)
{
    printf("x = %x\n", d.x);
}
void main()
{
    Date d1(123);
    Printer p;
    p.PrintDecimal(d1);
    p.PrintHexazecimal(d1);
}
```

- ▶ “friend” specifier can be applied to an entire class
- ▶ In this case , all methods from the “friend” class can access the members from the original class (e.g. all methods from class Printer can access the private data from class Data).

# “friend” specifier

## App.cpp

```
class Data;
class Modifier
{
public:
    void SetX(Data & d, int value);
};
class Data
{
    int x;
    int& GetXRef() { return x; }
public:
    int GetX() { return x; }
    friend void Modifier::SetX(Data &, int);
};
void Modifier::SetX(Data & d, int value)
{
    d.GetXRef() = value;
}

void main()
{
    Data d;
    Modifier m;
    m.SetX(d, 10);
    printf("%d\n", d.GetX());
}
```

- ▶ A method from a class can also be declared as friend for a class.
- ▶ The declaration must include the exact method signature and the return type.
- ▶ In this case, method ***SetX(Data& , int)*** from class **Modifier** can access private data from class **Data**.

**Q & A**