

# C++

Overloading, Constructors,  
Assignment operator

# Overloading

- Before looking at the initialization of objects in C++ with constructors, we need to understand what function overloading is
- In C, two functions with different arguments cannot have the same name
- Example:  

```
int add2(int a1, int a2) { return a1+a2; }  
int add3(int a1, int a2, int a3) { return  
a1+a2+a3; }
```
- In C++, you can use the same name for different functions (as long as it is not ambiguous)

# Example

- In C++, we can have:

```
int add (int a1, int a2) { return  
a1+a2; }
```

```
int add (int a1, int a2, int a3)  
{ return a1 + a2; }
```

- But adding the following (to the two above)  
would be a compile error:

```
double add (int a1, int a2) {return  
(double)a1 + (double)a2; }
```

**Return types are ignored in overload resolution.**

# Overloading

- **Overloading** allows to use the same name for functions (or member functions) with different input arguments

- Example with member functions:

```
class A {  
public:  
    void set (int a) { _a = a; }  
    void set (double f) { _f = f; }  
    void set (int a, double f) { _a = a; _f = f; }  
private:  
    double _f;  
    int _a;  
};
```

- Operator (which are functions in C++) can also be overloaded (this will be discussed later in this class)

# Overload resolution

- When a function is called, the compiler must determine which version to invoke
- Done by comparing the types of the arguments with the types of the parameters of all function in scope
- Invoke the function that best match to the arguments

# Overload resolution

A series of criteria are used:

- Exact match
- Match using promotion (e.g. `bool` to `int`, `char` to `int`, ...)
- Match with conversions (e.g. `int` to `double`, `T*` to `void*`, `int` to `unsigned int`)
- Match with user-defined conversions (see conversion constructor)
- Match using ellipsis ...

If two matches are found at the same level, the call is ambiguous and rejected.

# Example 1

```
void print(double);
```

```
void print(long);
```

```
void f() {
```

```
    print(1L); // print(long)
```

```
    print(1.0); // print(double)
```

```
    print(1); // error: ambiguous
```

```
}
```

## Example 2

```
void f(int);  
void g() {  
    void f(double);  
    f(1); // call f(double)  
}
```

In this case, the best match within the scope is selected. Only `f(double)` is in the scope.



# Default arguments

- It is possible to assign a default value to arguments of a function:

```
void f(int a, int b = 1) { // ... }  
int main() {  
    f(2);  
    f(2, 5);  
}
```

- There may be more than one default argument. In that case, all arguments without default values must be declared to the left of the arguments with default values:

```
void f(int a, int b = 1, int c = 2) // ok  
{ // function body }  
void f(int a = 1, int b, int c = 2) // not ok  
{ // function body }
```

# Default argument

- An argument can have its default value specified only once.
- It is done when the function is first declared.
- Usually it is done in the header file (not in the .cpp file)
- Example:

```
// a.h  
extern int f(int a, int b=1);
```

```
// a.cpp  
int f(int a, int b) {  
    return a+b;  
}
```

```
// main.cpp  
#include "a.h"  
  
int main() {  
    int c = f(1);  
    int d = f(1,2);  
}
```

# Example with member functions

- The same applies for member functions

```
// A.h
class A {
public:
    void set(int a, int b = 1);
private:
    int num;
};
```

```
// A.cpp
#include "a.h"

void A::set(int a, int b) {
    num = a * b;
}
```

```
// main.cpp
#include "a.h"

int main () {
    A a;
    a.set(1);
    a.set(2, 2);
}
```

# Summary

- **Function overloading** corresponds to more than one function definition for the same function name
- When overloading a function name, the function definitions must have different number of parameters, or some parameters with different types
- Overloading allows for **static polymorphism** (later: static polymorphism with templates and dynamic polymorphism)
- A function can have default argument values. The default value should be specified once only.

# Constructors

- A **constructor** is a particular type of member function that is used to create (or initialize) an object
- It has the same name as the class
- A constructor has no return type and does not return anything in its body (its definition)
- A constructor is called when an object is created. It is called when creating local objects on the stack, when creating objects on the heap, etc
- Constructors can be overloaded.

# Example

```
class A {  
public:  
    A() { _a = 0; }  
  
    A(int a) { _a = a; }  
  
    A(int a, int mult) {  
        _a = a * mult;  
    }  
  
    void print () {  
        std::cout << _a << std::endl;  
    }  
  
private:  
    int _a;  
};
```

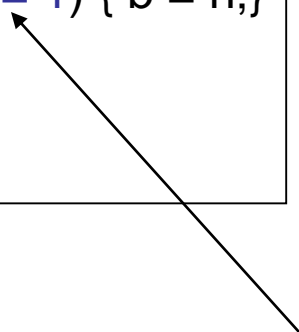
```
int main () {  
    A a;  
    A b(5);  
    A c(5, 2);  
  
    a.print(); // prints 0  
    b.print(); // prints 5  
    c.print(); // prints 10  
}
```

# Default constructor

- The default constructor is a constructor that can be called **without** an argument
- Examples:

```
class A {  
public:  
    A() {a = 0;}  
private:  
    int a;  
};
```

```
class B {  
public:  
    B(int n = 1) { b = n;}  
private:  
    int b;  
};
```



Note that a default constructor can have arguments if they have default values.

# Default constructor

- If there is no user-defined constructor, then a default constructor is *implicitly declared*
- Example:  

```
class A { public: int a; };
```

has no user-defined constructor, so one is implicitly declared
- An implicitly declared constructor is *implicitly defined* when used
- A *non-trivial* constructor requires the compiler to synthesize an implementation (e.g. the class contains non primitive types)



# Default constructor

- Be careful: if you provide a constructor, then the compiler will not implicitly define/synthesize one. This can lead to subtle issues.
- Example:

```
class A{  
public:  
    A(int a) { _a = a; }  
private:  
    int _a;  
};  
  
int main() {  
    A a; // compile-time error;  
}
```
- The code above will try to call the default constructor but there is no default constructor provided by the user and no default constructor automatically generated (because there is a user defined constructor) !

# Default constructor and initialization

- Initialization rules a bit subtle:
  - Statically allocated object: default initialization
  - For local variables and free-store (heap) objects: default initialization for members of class type, while members of built-in types are left uninitialized

- Example:

```
class A { public: int a; };  
A a3;  
int main(void) {  
    A a1;  
    std::cout << a1.a << std::endl; // warning a1.A::a uninitialized  
    A a2=A();  
    std::cout << a2.a << std::endl; // print 0  
    std::cout << a3.a << std::endl; // print 0  
    return 0;  
}
```

# Default constructor syntax

- What does the following mean:  
`A a ( ) ; ?`
- It declares a function `a` with no argument that returns an object of type `A`
- The funny thing is that:  
`A a (5) ;`  
would create an object of type `A` with the name `a` by calling the constructor:  
`A :: a (int a)`
- The proper way to call the default constructor is: `A a = A ( ) ;`

# Initialization of arrays

- The following code:  
    `A a[10];`  
or the following code:  
    `A* pa = new A[10];`  
creates an array of 10 object of type A by calling the default constructor.
- If your class does not have a default constructor, the code above will produce an error during compilation

# Initialization of arrays - 2

- An alternative initialization of array is to use the explicit initialization of arrays syntax:

```
class A {  
    public:  
        A(int i, int j);  
        // ...  
};
```

```
int main () {  
    A a[10] = {A(1,1), A(2,1), A(3,3), .....};  
}
```

# Conversion constructors (type conversion)

- A constructor with a single argument specifies a conversion from the type of its argument to the type of its class

- Example:

```
class A {  
    public:  
        A(int a) { _a = a;}  
    private:  
        int _a;  
};
```

Now we can do things as: `A a(1)` or even `A a = 1` (more on this later)

- If somewhere else in your code, you use an `int` when an `A` was expected, the constructor `A::A(int a)` will be silently called (creating a temporary object of type `A`). You may prevent that by using the keyword **explicit**:

```
//...  
explicit A(int a) { _a = a;}  
// ...
```

In that case, you will have to explicitly call `A::A(int)` no implicit conversion from an `int` to a `A` will occur.

# Type conversion - 2

- Note: it is possible to have a conversion constructor that has more than one argument when default arguments are used
- Example:

```
class A {  
public:  
    A(int a, int b = 1) { // ...}  
private:  
    int _a;  
};  
  
int main () {  
    A a(1); // will call A::A(int a, int b = 1)  
}
```

# Copy constructor

- A copy constructor is a constructor with only one argument of the same type:

```
class A {  
    public:  
        A(const A& another_a) {} // copy constructor  
};
```

- Copy constructors are used when:

- Initializing an object from another object:

```
A a1;  
A a2(a1); OR A a2 = a1; // BOTH call the copy constructor
```

- Passing objects to a function by value:

```
A a;  
void f(A a); // f is defined somewhere else  
f(a); // a copy of a is made by calling the copy constructor
```

- Returning an object from a function:

```
A func_return_a () {  
    A a;  
    a.set(5);  
    return a; // the copy constructor is called to make a copy of a  
}
```



# Default copy constructor

- If no copy constructor is defined by the user, the compiler will automatically generate one
- The default copy constructor provides memberwise bit copy. This can lead to disastrous effect if the class has data member which are pointers

```
class A {  
    public:  
        A() { a = 0; }  
        void set(int n) { a = n; }  
    private:  
        int a;  
};  
  
int main() {  
    A a1;  
    A a2(a1); // calls the default copy ctor  
    A a3 = a1; // calls the default copy ctor  
}
```

# Default copy constructor

```
class A {
public:
    A() { a = new int; *a = 0; }
    ~A() { delete a; }
    void set(int n) {*a = n;}
    void print() { std::cout << *a << std::endl; }
private:
    int* a;
};

int main () {
    A a1;
    A a2 = a1;
    a1.set(2);
    a1.print(); // prints 2
    a2.print(); // prints 2 ?????
}
```

It may appear strange that a2.print() will print 2.

But it is normal since the user is not providing any copy constructor therefore the default copy constructor generated by the compiler will be used.

This copy constructor does bitwise copy of the member data, therefore a1.a and a2.a will both point to the same memory location.

In addition this program is most likely going to produce a segmentation fault since we are trying to delete two times the memory pointed by a:

- 1) when the destructor of a2 is called
- 2) when the destructor of a1 is called

# Copy constructor

```
class A {  
public:  
    A() { a = new int; *a = 0; }  
    A(const A& another_a) {  
        a = new int;  
        *a = *another_a.a;  
    }  
    ~A() { delete a; }  
    void set(int n) {*a = n;}  
    void print() { std::cout << *a << std::endl;  
    }  
private:  
    int* a;  
};
```

```
int main () {  
    A a1;  
    A a2 = a1;  
    a1.set(2);  
    a1.print(); // prints 2  
    a2.print(); // prints 0  
    // no more memory problem  
}
```

# Member initialization

- Data members can be initialized in the body of the constructor or by using an **initialization list**
- For member objects it is actually preferable to initialize them in the initialization list
  - Be careful that the corresponding member (copy or maybe conversion) constructor should exist
  - List members in the initialization list in the order in which they are declared

```
class A {  
    public:  
        A();  
        A(const A& a);  
        // ...  
};  
  
class B {  
    public:  
        B() : _n1(0), _n2(0) { // rest of the ctor}  
        B(int i, int j, A a) : _n1(i), _n2(j), _a(a) {  
            // rest of the ctor  
        }  
    private:  
        int _n1, _n2;  
        A _a;  
}
```

# Member initialization

- The member initializer list starts with a colon ':' and the individual member initializers separated by commas ','
- Example:  
`B() : _n1(0), _n2(0)`
- The members' constructors are called before the body of the constructor is executed
- It is best to specify the initializers in the member declaration order (otherwise: compiler emits a warning)
- If a member constructor needs no arguments, it need not be mentioned

# Initialization of const and reference member

- **Reference members** can be initialized only in initialization list
- **Const members** can be initialized only in initialization list

```
class A {  
public:  
    A(const int& b) : _b(b), _c(1) {  
        // rest of the ctor  
    }  
private:  
    int& _b;  
    const int _c;  
};
```

```
class A {  
public:  
    A(const int& b) {  
        _c = 1; // compile-time error  
        _b = ?; // b or &b or ??  
    }  
    // rest is same as above  
};
```

# Overloading = (assignment operator)

- The assignment operator is used whenever you assign the “content” of one object to another object. Example:  

```
A a1;  
A a2(5);  
a1 = a2; // call the assignment operator
```
- The syntax for the assignment operator is:  

```
T& operator= (const T& t);
```
- If the user did not define an assignment operator in his class, the compiler will automatically generate one
- This default assignment operator will do a bitwise copy of the object data members
  - This leads to the same type of problems we saw with the default copy constructor (e.g. if some data member are pointers)

# Overloading =

```
class A { };

class B {
public:
    B() : p_(new A()) {}
    B(const B& b) : p_(new A(*b.p_)) {}
    ~B() { delete p_; }

private:
    A* p_;
};
```

```
int main () {
    B b1;
    B b2;
    b2 = b1; // calls assignment operator
    // now b2.p_ and b1.p_ points to the
    // same memory: modifying *b1.p_ will
    // also modify *b2.p_. Is it really what
    // we want ?
}
```

The code above is trying to delete several time the same memory which may result in a seg. fault.



# Fixing the previous problem

```
class A { }; // assume A has a copy ctor

class B {
public:
    B() : p_(new A()) {}
    B(const B& b) : p_(new A(*b.p_)) {}
    B& operator= (const B& b2) {
        delete p_;
        p_ = new A(*b2.p_);
        return *this;
    }
    ~B() { delete p_; }

private:
    A* p_;
};
```

```
int main () {
    B b1;
    B b2;
    b2 = b1; // calls our assignment operator
}
```

No more memory problem.

Now `*b1.p_ == *b2.p_` but `b1.p_` and `b2.p_` are pointing to different memory locations.

# Copy constructor and assignment operator

- Be careful that:

```
B b1;  
B b2 = b1; // calls the copy constructor  
           // this is the same B b2(b1)
```

while:

```
B b1;  
B b2;  
b2 = b1; // calls the assignment operator  
         // this is same as: b2.operator=(b1);
```

# Assignment operator and self assignment

- When overloading the assignment operator, you have to keep in mind that self assignment can happen
- Self assignment looks like:  
A a;  
a = a;
- Be careful when implementing the assignment operator especially if you have data member which are pointers and you delete, allocate memory for this pointers in your assignment op.

# Assignment operator and self assignment

- The following code handles self-assignment properly:

```
A& A::operator= (const A& a1) {  
    if (this == &a1) return *this;  
    // put your code here  
    return *this;  
}
```

# Move semantic (Rvalue references)

```
Vector operator+(const Vector& a, const Vector& b) {  
    Vector r(a.size());  
    for (int i=0; i<a.size(); ++i)  
        r[i] = a[i] + b[i];  
    return r;  
}  
  
void f(const Vector& x, const Vector& y, const Vector& z)  
{  
    Vector r;  
    // ...  
    r = x + y + z;  
    // ...  
}
```

# Move semantic

- Returning from `+` involves a copy of `r` out of the local variable.
- In `f()`, it makes copying a `Vector` at least twice.
- If a `Vector` is large, it is not very efficient.
- In `+` all we want is get the result out of the function.
- This intent can be expressed with move semantic (or Rvalue references).
- Move ctor and assignment "move" their arguments rather than "copy" them.

# Move semantic

```
class Vector {  
    // ...  
    // copy ctor  
    Vector(const Vector& o);  
    // copy assignment  
    Vector& operator=(const Vector& o);  
  
    // move ctor  
    Vector(Vector&& o);  
    // move assignment  
    Vector& operator=(Vector&& o);  
};
```

# Move semantic

```
// example of impl.  
Vector::Vector(Vector&& o)  
    : elem{o.elem}, sz{o.sz}  
{  
    o.elem = nullptr;  
    o.sz = 0;  
}
```



# Move semantic

- `&&` means rvalue reference.
- I.e. a reference to which we can bind an rvalue.
- rvalue is something that we can not assign to (e.g. an integer returned from a function call).
- Move operation: applied when an rvalue reference is used as an initializer or as the rhs of an assignment.

# Move semantic

```
#include <utility> // for move()

void f() {
    Vector x(100000);
    Vector y(100000);
    Vector z(100000);

    z = x; // copy
    y = std::move(x); // move (explicit)

    return z; // move
}
```

# Uniform initialization list

- To initialize objects, prefer using an initialization list everywhere (C++11):

```
// Assume we have a constructor  
// X(int, int) in the class X  
X x1 = X{1, 2};  
X x2 = {1, 2};  
X x3{1, 2};  
X* xp = new X{1, 2};
```