Initialization List

The paramaterless ctor (aka default ctor)

```
int main() {
   B b;
}
```

```
class A {
public:
 A() {
  std::cout <<
  "A - " <<
  "parameterless" <<
  " ctor\n";
 }
};
```

```
class B {
   A _a1,_a2;
public:
   B() {
     std::cout <<
     "B – parameterless " <<
     "ctor\n";
   }
};
```

```
// output
A – parameterless ctor
A – parameterless ctor
B – parameterless ctor
```

The paramaterless ctor (aka default ctor)

```
int main() {
    B b;
}
```

```
class A {
public:
  A(int a) {
    std::cout <<
    "A ctor with one
     parameter\n";
  }
};
```

```
class B {
    A _a1,_a2;
public:
    B() {
      std::cout <<
      "B – parameterless " <<
      "ctor\n";
    }
};
```

```
// compilation error
No parameterless ctor for _a1, _a2
```

The initialization list

```
int main() {
    B b(2,3);
}
```

```
class A {
public:
    A(int a) {
        std::cout <<
        "A (" << a << ") "
        << std::endl;
    }
};
```

```
class B {
    A _a1,_a2;
public:
    B(int i, int j)
    :_a1 (i), _a2(j)
    {
        std::cout
        << "B cons"
        << std::endl;
    }
};
```

```
// output
A (2)
A (3)
B cons
```

Initialization using pointers (1)

```
int main() {
    B b(2);
}
```

```
class A {
public:
    A(int a) {
        std::cout <<
        "A (" << a << ") "
        << std::endl;
    }
};
```

```
class B {
    A *_ap;
public:
    B(int i);
};

B::B(int i) {
    _ap = new A (i);
    cout << "B cons\n";
}
```

```
// output
A (2)
B cons
```

Initialization using pointers (2)

```
int main() {
    B b(2);
}
```

```
class A {
public:
    A(int a) {
        std::cout <<
        "A (" << a << ") "
        << std::endl;
    }
};
```

```
class B {
    A *_ap;
public:
    B(int i);
};

B::B(int i)
    : _ap (new A(i))
{
    cout << "B cons\n";
}
```

```
// output
A (2)
B cons
```

The initialization list

1. Initialization of object members.
2. Initialization of constants and reference variables.
3. In the future: Initialization of parent class.
4. It is faster and safer to use the initialization list than initialization in the constructor

More on initialization & C++11: init_demo.zip

Reference variables

References

- A *reference* is an alias –
  an alternative name to an existing object

References, example (not a useful one)

- A *reference* is an alias –
  an alternative name to an existing object

```
int i = 10;
int& j = i; // j is a int reference
            // initialized only
            // once !
j += 5; // changes both i and j
```

References, example (not a useful one)

- A *reference* is an alias –
  an alternative name to an existing object

```
int i = 10;
int& j = i; // j is a int reference
            // initialized only
            // once !
j += 5; // changes both i and j
int* k = new int();
j = k; // error k is a pointer
j = *k; // ok j and i equals to *k
```

The famous swap

```c
// C version
void swap
     (int *a, int *b)
{
   int t = *a;
   *a = *b;
   *b = t;
}
```

```c
// C++ version
void swap
     (int &a, int &b)
{
   int t = a;
   a = b;
   b = t;
}
```

- More intuitive syntax
- No pointer arithmetic mistakes
- Ref variables are actually const pointers (standard implementation)
- Must be initialized in their declaration (initialization list), like const variables

The famous swap: std::swap later in the course

Pointer vs non-const reference

References can be used as output parameters, similar to pointers.

Pros:

- It is hard to have reference to undefined value
- The syntax inside the function is clearer

Cons:

- You can’t see what you are doing at call site (but this shouldn’t be a problem if the function is named right and documented)

Pointer vs non-const reference

As a convention always order argument, in first out last.

Lvalue & Rvalue

```
int a=1;
a=5; // Lvalue = Rvalue, Ok
a=a; // Lvalue = Lvalue, Ok
5=a; // Rvalue = Lvalue Comp. error
5=5; // Rvalue = Rvalue Comp. error
```

Lvalues: variables, references …

Rvalues: numbers, temporaries …

Temporary: A result of expression that isn't stored – a+5 creates a temporary int with value 6.

R/L value and references

non-const Reference – only to a non const Lvalue.

const reference – to both Lvalue and Rvalue

```
int lv=1;
const int clv=2;
int& lvr1=lv;
int& lvr2=lv+1; //error!
int& lvr3=clv;  //error!
const int& cr1=clv;
const int& cr2=5+5;
```

R/L value and references

non-const Reference – only to a non const Lvalue.

const reference – to both Lvalue and Rvalue

```
int lv=1;
const int clv=2;
int& lvr1=lv;
int& lvr2=lv+1; //error!
int& lvr3=clv;  //error!
const int& cr1=clv;
const int& cr2=5+5; // This is useful for
                    // Functions arguments
```

Lvalue & Rvalue

```
int a=1;
a
a
5
5
```

L

R

t

Reference – only to Lvalue
Const Reference – to Lvalue & Rvalue

C++11:
Rvalue reference (&& used for move ctor and assignments)

Temporary: A result of expression that isn't stored – a+5 creates a temporary int with value 6.

A fancy way to pass arguments to function

```
// Pass by value
void foo (int a)
{
    ...
}
// Pass by pointer
void foo (int *pa)
{
    ...
}
```

```
// pass by const ref
void foo (const int &a)
{
    ...
}
```

- Avoid copying objects, without allowing changes in their value.

Return a reference to variable

```
class Buffer
{
    size_t _length;
    int *_buf;
public:
    Buffer (size_t l) :
    _length (l),
    _buf (new int [l])
    {
    }
    int& get(size_t i)
    {
        return _buf[i];
    }
};
```

```
int main ()
{
    Buffer buf(5);
    buf.get(0)= 3;
}
```

Return a ref. to a legal variable (e.g. not on the function stack). Will be more useful with operators overloading

Ref to NULL?

Possible to get, but undefined behavior.

If you do:

```
int* p= NULL; // nullptr c++-11
int& r= *p;
```

You do get reference to NULL, but the behavior of such program is **undefined**.

Reference

- Must always refer to “legitimate” objects, i.e. allocated existing object
- After initialization, cannot refer to a different object
- The “interface” of references is as if they were the object themselves
- Often implicitly implemented by the compiler as pointers or helps to “inline” the function (we’ll soon learn what is “inline”)

By Value

```
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
int main()
{
    int x=3, y=7;
    swap(x, y);
    // still x == 3, y == 7 !
}
```

“By value” arguments cannot be changed!

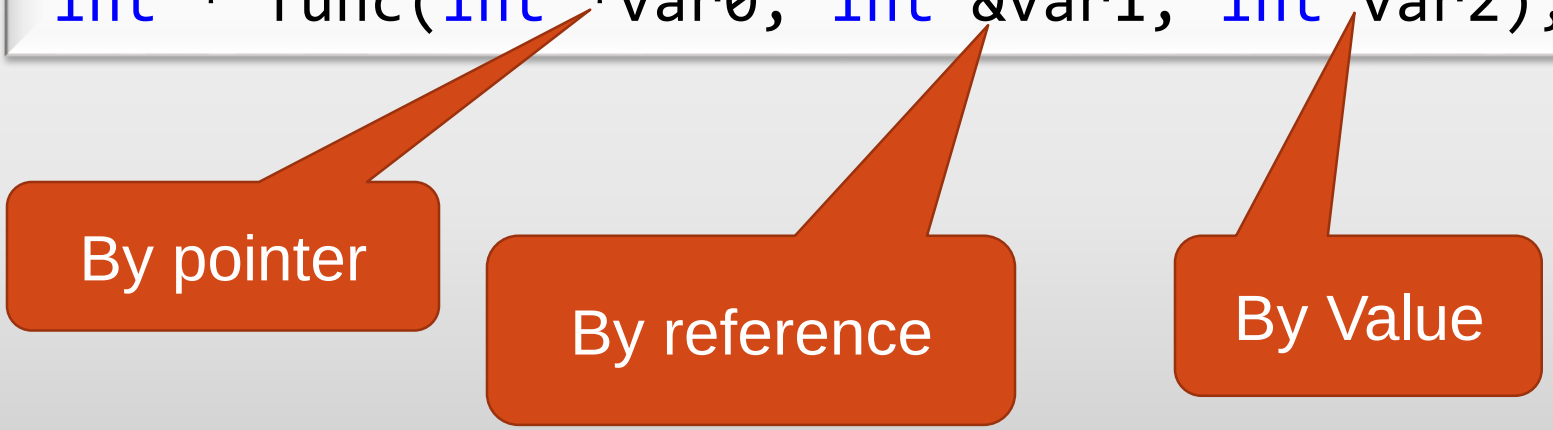
Pointers vs. Reference

```
void swap(int* a, int* b)
{
   int temp = *a;
   *a = *b;
   *b = temp;
}
...
int main()
{
   int x=3, y=7;
   swap(&x, &y);
   // x==7, y==3
}
```

```
void swap(int& a, int& b)
{
   int temp = a;
   a = b;
   b = temp;
}
...
int main()
{
   int x=3, y=7;
   swap(x, y);
   // x==7, y==3
}
```

Summary

```
int * func(int *var0, int &var1, int var2);
```



But it can be viewed as always passing “by value”!

The value can be pointer or reference!

References - why?

- **Efficiency – avoid copying arguments**
- Enables modifying variables outside a function
- *But that can be done with pointers too!*
- Everything that can be done with references, can be done with pointers
- But some “dangerous” features of pointers cannot be done (or harder to do) with references
- Easier to optimize by the compiler
- More convenient in many cases (see examples)
- **Widely used as parameters and return values**

Reference – more

- Like with pointers, don’t return a pointer or reference to a local variable
- You can return a pointer or a reference to a variable that will survive the function call, for example:
  - A heap variable (malloc, new, etc.)
  - A variable from a lower part of the stack
  - Globals, static variables and static members of a class

```
void add(Point& a, Point b)
{
      // a is reference, b is a copy
      a._x+= b._x;
      a._y+= b._y;
}

int main()
{
      Point p1(2,3), p2(4,5);
      add(p1,p2); // note: we don’t send pointers!
                  // p1 is now (6,8)
      ...
}
```

```
void add(Point& a, const Point& b)
{
    // a is reference,
    // b is a const ref
    a._x+= b._x;
    a._y+= b._y;
}

int main()
{
    Point p1(2,3), p2(4,5);
    add(p1,p2); // note: we dont send pointers!
            // p1 is now (6,8)
    ...
}
```

- b is Reference => is not copied
- b is Const => we can't change it
- Important for large objects!

```
Point& add(Point& a, const Point& b)
{
   // a is reference, b is a const ref
   a._x+=b._x;
   a._y+=b._y;
   return a;
}

int main()
{
   Point p1(2,3), p2(4,5), p3(0,1);
   add(add(p1,p2),p3);              // now p1 is (6,9)
   cout << add(p1,p2).getX();       // note the syntax
   ...
}
```

C++ const

Const variables – like in c

```
int * const p1 = &i; // a const
// pointer to an un-const variable
```

- `p1++; // c.error`
- `(*p1)++; // ok`

```
const int * p2 = &b; // an un-const
// pointer to a const variable
```

- `p2++; // ok`
- `(*p2)++; // c.error`

```
const int * const p3 = &b; // a const
// pointer to a const variable
```

Const objects & functions (1)

```
class A
{
public:
    void foo1() const;
    void foo2();
};
void A::foo1() const
{
}
void A::foo2()
{
}
```

```
int main()
{
    A a;
    const A ca;
    a.foo1();
    a.foo2();
    ca.foo1();
    ca.foo2(); // comp.
               // error
}
```

Const objects & functions (2)

```
class A
{
public:
    void foo() const;
    void foo();
};
void A::foo() const
{
    cout << "const foo\n";
}
void A::foo()
{
    cout << "foo\n";
}
```

```
int main()
{
    A a;
    const A ca;
    a.foo ();
    ca.foo();
}
```

```
// output
foo
const foo
```

Why?

Overload resolution, again:

```
A::foo(A* this)

A::foo(const A* this)
```

Return a const ref. to variable

```
class Buffer
{
    size_t _length;
    int *_buf;
public:
    Buffer (size_t l):
    _length (l),
    _buf (new int [l])
    {
    }
    const int& get(size_t i) const
    {
        return _buf[i];
    }
};
```

```
int main ()
{
    Buffer buf(5);
    buf.get(0)
            = 3; // illegal
    std::cout <<
    buf.get(0);
}
```

Why?

Const objects with pointers – like in c

```
class B
{
public:
    int _n;
};
class A
{
public:
    B* _p;
    A();
    void foo() const;
};
A::A() : _p(new B)
{
    _p->_n = 17;
}
void A::foo() const
{
    //_p++; // this will not
            // compile
    _p->_n++; // this will !
}
int main()
{
    const A a;
    std::cout <<
    a._p->_n << std::endl;
    a.foo();
    std::cout <<
    a._p->_n << std::endl;
}
```

```
// output
17
18
```

Const objects with references

```
class A
{
public:
    int & _i;
    A(int &i);
    void foo() const;
};
A::A(int &i) : _i(i)
{

}
void A::foo() const
{
    _i++;
}
```

```
int main()
{
    int i = 5;
    const A a (i);
    std::cout <<
    a._i << std::endl;
    a.foo();
    std::cout <<
    a._i << std::endl;
}
```

```
// output
5
6
```

Initialization of const and ref.

```
class A
{
    int& _a;
    const int _b;
public:
    A(int& a);
};
A::A(int& a)
{
    _a = a;
    _b = 5;
}
```

```
// compilation error
Const and ref vars must initialized in their
declaration (when they are created):
For fields of a class it's in the
initialization list
```

Initialization of const and ref

```
class A
{
    int& _a;
    const int _b;
public:
    A(int& a);
};
A::A(int& a)
{
    _a = a;
    _b = 5;
}
// compilation error
```

```
class A
{
    int& _a;
    const int _b;
public:
    A(int& a);
};
A::A(int& a)
 : _a(a), _b(5)
{
}
// compiles ok
```

mutable

- `mutable` means that a variable can be changed by a const function (even if the object is const)
- Can be applied only to non-static and non-const data members of a class

mutable: example #1

```
class X
{
public:
  ...
  X() : _fooAccessCount(0) {}

  bool foo() const
   {
      ++_fooAccessCount;
      ...
   }
   unsigned int fooAccessCount() { return _fooAccessCount; }

private:
   mutable unsigned int _fooAccessCount;
};
```

mutable: example #2

```
class Shape
{
public:
  ...
  void set...(...) { _areaNeedUpdate= true; ... }
  double area() const
   {
      if (_areaNeedUpdate) {
         ...
         _areaNeedUpdate= false;
      }
      return _area;
   }
private:
   mutable bool _areaNeedUpdate= true;
   mutable double _area;
};
```