Reference variables

- Version 1: Dr. Ofir Pele
- Version 2: Dr. Miri Ben-Nissan
- Version 3: Dr. Erel Segal-Halevi

Reference – two definitions

- (a) A pointer that is used like an object.
- (b) Alias - alternative name to existing object.

References, example

- (a) A pointer that is used like an object.
- (b) Alias - alternative name to 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 *(folder 1)*

- (a) A pointer that is used like an object.
- (b) Alias - alternative name to 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
```

Pointer vs. Reference

	Pointer	**Reference**
Initialization	Optional	Mandatory
Dynamic	Yes	No
Arithmetic	Yes	No
Always defined	No	Yes
Notation	(*p), p->x	r, r.x
Containers	Yes	No

The famous swap

```
// Wrong version
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 !
}
```

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

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

Lvalue & Rvalue

Lvalue = **Left Value –** can appear at left side of =.

= **Located Value –** has a fixed memory location.

Examples: variables, references …

Rvalue = not Left Value. Numbers, temporaries …

```
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
a+5=7;  // Temporary = Rvalue - Comp. error
f(5)=7; // RIDDLE: Is this legal?
```

Lvalue & Rvalue

Lvalue = **Left Value –** can appear at left side of =.

= **Located Value –** has a fixed memory location.

Examples: variables, references …

Rvalue = not Left Value. Numbers, temporaries …

```
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
a+5=7;  // Temporary = Rvalue - Comp. error
f(5)=7; // .. it depends: we will see soon.
```

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
```

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 (folder 2)

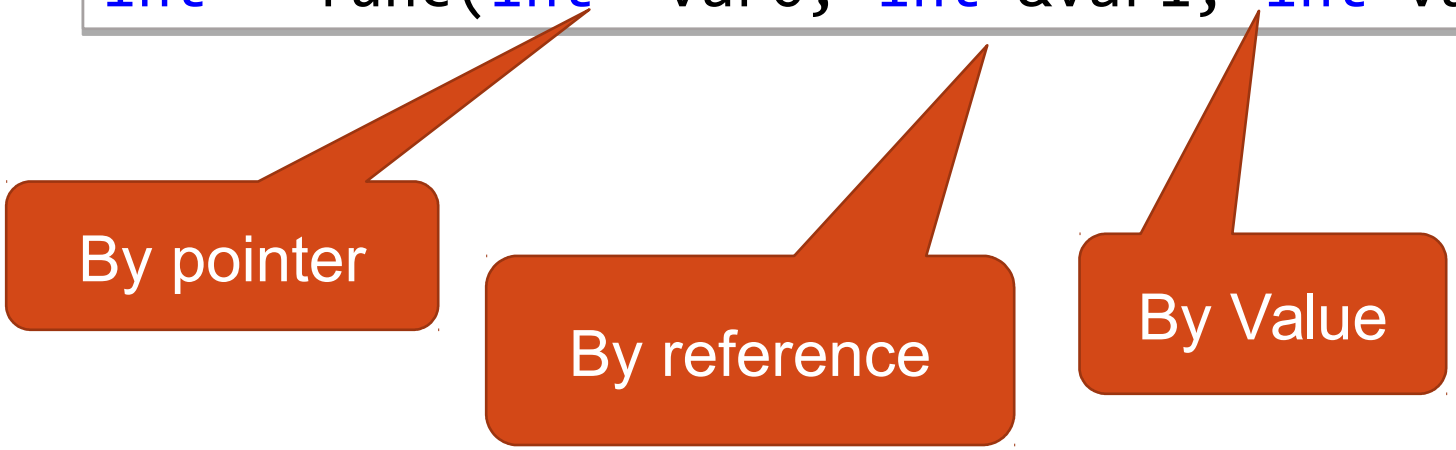
```
class Buffer
{
    size_t _length;
    int *_buf;
public:
    Buffer (size_t l) :
    _length (l),
    _buf (new int [l])
    {
    }
    int& get(size_t i)
    {
        return _buf[i];
    }
};
```

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

Return a ref. to a legal variable (e.g. not on the function stack).

Summary

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



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**

Return a reference from a function *(folder 2)*

- Don’t return a reference to a local variable.
- You can return a pointer or a reference to a variable that will survive the function call, e.g:
  - A heap variable (allocated with new).
  - A variable from a lower part of the stack.
  - Globals.
  - Class members.
    - Useful for call-chaining.

```
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!

Parameter passing

By value	By reference	By const reference
void f (Point x) {…}	void f (Point& x) {…}	void f (const Point& x) {…}
x is copied	x is not copied	x is not copied
Compiler lets f modify x, but changes have no effect outside	f can modify x	compiler does not let f modify x

```
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;
}
```

```
// output
17
18
```

```
void A::foo() const
{
    //_p++;//won't
compile     _p->_n++;
// this will !
}


int main()
{
    const A a;
    cout <<
        a._p->_n <<
endl;
    a.foo();
    cout <<
       a._p->_n <<
endl;
}
```

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;
};
```

Copy Constructor *(folder 3)*

- Called whenever an object of type T is copied.
- Copy Constructor to class T gets as argument **const T&** *(Why?)*
- You should consider for each class whether it needs **deep copy** or **shallow copy**.