W.P.Hwang
CS-CCS-NCTU

C++11 supplementary

Lambda expression

- A lambda expression denotes an anonymous function.
- Basic syntax

  [*capture***] (***parameters***) ->** *return-type* **{** *body* **}**

  Example

  ```
  [](int x) -> int { return x*x; }
  ```

  In case the trailing return type is omitted, the type of **x*x** is the return type.
- A lambda expression creates a function object of a unique class type – called the *closure type* – that supports **operator()**.
- Example

  ```
  int main()
  {
     cout << [](int x){ return x*x; }(3);
     cout << [](int x){ return x*x; }.operator()(3);
  }
  ```
- The name of the closure type of each lambda expression is uniquely generated by the compiler.
  E.g. the two lambda expressions in preceding example are of distinct type.
- To give a lambda expression a name, the name of its closure type must be known. To this end, we may resort to **auto**, **decltype**, template argument deduction, etc.

- Example

```
int main()
{
   auto f = [](int x){ return x*x; }
   decltype(f) g = f;
   cout << f(3) << g(3);
   int a[7]={1,2,3,4,5,6,7}; // C++ as a better C, p63
   cout << accumulate(a,a+7,0,
               [](int x,int y){ return x+y; });
}
```

Use template argument deduction to deduce its type.

- A lambda expression with free variables is meaningless.
For example, what is the meaning of this lambda expression?

```
[](int x){ return x+y; }
```

y is a free variable

- Free variables must be captured by value (copy) or reference.
- Example

may be omitted

```
int main()
{
   int x=2,y=3;
   auto f = [x,y]() { return x+y; };      // value
   auto g = [&x,&y]() { return x+y; };  // reference
   x=4; y=5;
   cout << f() << g();      // 59
}
```

Comments

```
[=]{ return x+y; };      // default capture by value
[&]{ return x+y; };      // default capture by reference
```

// both capture **x** by value and **y** by reference

```
[=,&y]{ return x+y; }
[&,x]{ return x+y; }
```

- Example

```
#include <algorithm>        // for  for_each
int main()
{
   int a[7]={1,2,3,4,5,6,7};
   int sum=0;
   for_each(a,a+7,[&sum](int x)->void{ sum+=x; });
   cout << sum;
}                                  // may be omitted
```

Note that the call to `for_each` essentially executes the loop:

```
for (int* it=a;it!=a+7;++it)
   [&sum](int x){ sum+=x; }(*it);
```

- Example (May be skipped on first reading)

```
int main()
{
   int x=2,y=3;
   auto f = [x,&y]{ return x+y; };
   x=4; y=5;
   cout << f();
}
```

is compiled to something like

```
int main()
{
   int x=2,y=3;
   class I_have_no_name {
   public:
      I_have_no_name(int a,int& b) : x(a),y(b) {}
      int operator()() const { return x+y; }
   private:
      int x,&y;
   };
   auto f = I_have_no_name(x,y);
   x=4; y=5;
   cout << f();
}
```

Polymorphic function wrapper

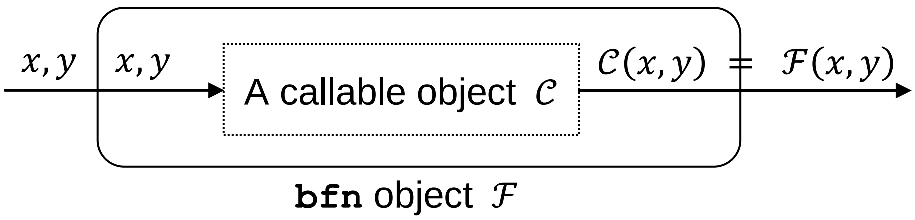
- The **`function`** class template provides polymorphic wrappers that encapsulate arbitrary callable objects.
- Example

  The type
  `std::function<int(int,int)>`
  encapsulates all callable objects that have the call signature **`int(int,int)`**.

```
#include <functional>
int add(int x,int y) { return x+y; }
int main()
{
   typedef function<int(int,int)> bfn;
   bfn f = [](int x,int y){ return x+y; };
   bfn g[2] = {plus<int>(),add};
   cout << f(2,3) << g[0](2,3) << g[1](2,3);
}
```

Comment

A **`bfn`** object holds a callable object and supports a call operation that forwards to that object.




```
int bfn::operator()(int x,int y) const
{
   return 𝒞(x,y);           // ℱ forwards x and y to 𝒞
}
```

Notice that , for $\mathcal{C} =$ **`plus<int>()`**, $\mathcal{F}$ forwards **x** and **y** to $\mathcal{C}$ by reference. (This is OK.)
For the other two cases, $\mathcal{F}$ forwards **x** and **y** to $\mathcal{C}$ by value.

- Example – Function composition (C++ as a better C, p65)

  // Version A

```
function<int(int)> c(int f(int),int g(int))
{
   return [f,g](int x){ return f(g(x)); };
}
int f(int x) { return x+x; }
int g(int x) { return x*x; }
int main()
{
  cout << c(f,g)(3) << endl;
}
```

  // Version B – File comp.cpp

```
#include <iostream>
#include <functional>
using namespace std;
typedef function<int(int)> ufn;
ufn c(ufn f,ufn g)
{
   return [f,g](int x){ return f(g(x)); };
}
int main()
{
   cout << c([](int x){ return x+x; },
                  [](int x){ return x*x; })(3);
   cout << endl;
}
```

  Note: Use GNU C++ compiler to compile the file comp.cpp.

  bsd2> g++47 -std=c++11 -rpath=/usr/local/lib/gcc47 comp.cpp
  bsd2> ./a.out
  18 for GLIBCXX_3.4.14

auto specifier

- **`auto`** is no longer is storage class specifier, e.g.

```
void p()
{
   auto int x;          // error in C++11
   static int y;
}
```

- **`auto`** is now a type specifier, signifying that
  1. the type of a variable being declared shall be deduced from its initializer using template argument deduction, or
  2. a function declarator shall include a *trailing-return-type*.

- Example

```
void p()
{
   auto x=3;              // x has type int
   const auto* y=&x;      // y has type const int*
   static auto z=x;       // z has type int
}
```

Trailing return type

- Trailing-return-types are convenient when the return type of a function is complex.
- Example (C++ as a better C, p65)

```
auto msg() -> void { cout << "hello\n"; }
auto mkmsg() -> void (*)() { return msg; }
auto main() -> int {  mkmsg()(); (*mkmsg())(); }
```

List-initialization

- List-initialization is the initialization of an object from a braced initializer list.
- Narrowing conversions are not allowed at the top level in list-initializations.
- Example

```
// variable initialization
int a[2]={1,2};             // ok, as usual
int b[2]={1,2.0};           // error in C++11, narrowing
int c[2]={1,(int)2.0};      // ok, not a top-level narrowing
int d[2]{1,2};              // new in C++11
int e[2]{};                 // default to 0,0

struct X { int x,y; };
X a={1,2};
X d{1,2};
X f({1,2});
// Only class type can parenthesize a braced initializer list

int a={2};
int d{2};
// Q: Which is ill-formed?
// int x={2.0},y{2.0},z=2.0,w(2.0);

// assignment
d={3};

// new expression
int* a=new int{2};
int* b=new int[3]{1,2,3};
X* c=new X[3]{{1,2},{3,4},{5,6}};
int n=2;
int* d=new int[n]{1,2,3};
// Warning, unable to verify the length of initializer list
int* e=new (operator new(sizeof(int))) int{2};
```

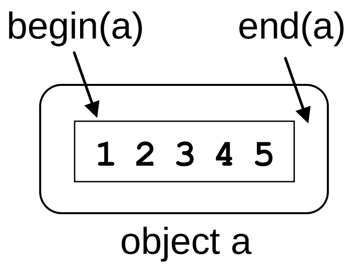
- Example (Cont'd)

// return statement

```
#include <utility>
pair<int,int>  f() { return {1,2}; }
```

// function argument

```
#include <initializer_list>
int sum(initializer_list<int> a)
{
   int s=0;
   for (const int* it=begin(a);it!=end(a);++it)
      s+=*it;
   return s;
}
int main()
{
   cout << sum({1,2,3,4,5});
}
```



Comment

An object of type **initializer_list<T>** provides access to an array of objects of type **const T**.

Range-based `for` statement

- Syntax

for (*for-range-declaration* **:** *expression* **)** *statement*
for (*for-range-declaration* **:** *braced-init-list* **)** *statement*

- Example

The preceding **for** loop may be written as:

```
for (int i : a) s+=i;
```

```
int array[5] = {1,2,3,4,5};
for (int& i : array) i++;
for (int i : {1,2,3,4,5}) cout << i;
for (char c : "Snoopy") cout << c;
```

Resource stealing and move semantics

Lvalue reference and rvalue reference

- Syntax and semantics

  cv `T&` lvalue reference
  cv `T&&` rvalue reference

  Except where explicitly noted, they are semantically equivalent (e.g. they must be initialized, the binding can't be altered, etc) and commonly referred to as references.

- Example

```
int x=2;
int& y=x;            // int& y=2;      ×
const int& z=2;
int&& z=2;           // int&& z=x;     ×
```

Reference binding and overload resolution rules

- The table below summarizes the binding and resolution rules.

Expression / Reference type	`const T` rvalue	`T` rvalue	`const T` lvalue	`T` lvalue	Priority
`const T&`	O	O	O	O	low
`T&`				O	
`const T&&`	O	O			
`T&&`		O			high

- Comments

  1 Lvalues prefer lvalue references, whereas rvalues prefer rvalue references.
  2 `const T&&`
  This type is hardly used.

Name rule

- Named rvalue references are treated as lvalues.
  Unnamed rvalue references are treated as rvalues.

- Example

```
string::string(const char*);      // ctor
string::string(string&&);         // move ctor
string::string(const string&);    // copy ctor

void q(string y) {}

void p(string&& x)          // x is bound to an rvalue
{
    q(x);                   // x is an lvalue; call copy ctor
    // x is visible here
}

p(string("Pluto"));
q(string("Pluto"));         // call move ctor, may be elided
```

Comment

Copy elision: the copy/move ctor may be elided by RVO.
Copy elision is more efficient than resource stealing.

Comment

If **x** were treated as an rvalue inside the function **p**, it would be moved to **y** by move constructor and the rest of the function would see a modified **x**, violating the guarantee that resource stealing does not visibly modify anything.

- Example

```
const int& x=2;
x++;            // error
int&& y=2;
y++;            // y is an lvalue
```

Moving from lvalues

- **std::move** is a function that turns its argument into an rvalue without doing anything else.

- Here is a simplified version of **move** that works only for lvalues:

```
template<typename T>
T&& move(T& a)
{
    return static_cast<T&&>(a);
}
```

- Example (Cont'd)

  To steal the resource bound to **x**, we have to write

```
void p(string&& x)
{
    q(move(x));      // move(x) is an rvalue; call move ctor
}
```

  Put another way, **move(x)** is an unnamed rvalue reference, hence it is an rvalue.

- Example

```
string s("Snoopy");
string t(s);             // call copy ctor
string u(move(s));       // call move ctor
```

  Even if **s** isn't a temporary, we may steal its resource.

- Example

```
int s(2);
int t(s);                // copy
int u(move(s));          // copy, too

s=2;                     // lvalue, ok
move(s)=2;               // rvalue, no
```

- The simplified version of **`move`** can't be called on an rvalue, e.g.
  `move(2)`
  is illegal.
  However, although redundant, **`std::move`** actually works fine when called on an rvalue.

  ```
  int t(2);                  // copy
  int u(std::move(2));       // copy, too
  ```

  Here is the complete definition:

  ```
  template<typename T>
  typename remove_reference<T>::type&&
  std::move(T&& a)
  {
     typedef typename remove_reference<T>::type X;
     return static_cast<X&&>(a);
  }
  ```

  For it to work, there are special template argument deduction rule for **`T&&`** and reference collapsing rules.

- Example

  ```
  template<typename T>
  void std::swap(T& x,T& y)
  {
     T z=std::move(x);
     x=std::move(y);
     y=std::move(z);
  }
  ```

  Comments

  1 If **`T`** is a non-class type, **`swap<T>`** has no harm.
  2 If **`T`** is a class type with callable move constructor and move assignment operator, say,
  ```
  T::T(T&&);
  T& T::operator=(T&&);
  ```
  they will be invoked; otherwise, **`T`** shall have callable copy constructor and copy assignment operator, say,
  ```
  T::T(const T&);              // must be const
  T& T::operator=(const T&);   // must be const
  ```

- Example (C++ as a better C, p24)

```
template<typename T>
int partition(T* a,int l,int h)
{
   ~~T x=a[h];~~ T x=std::move(a[h]);
   int i=l-1;
   for (int j=l;j<h;j++)
      if (a[j]<x) {
         i++; ~~T z=a[i]; a[i]=a[j]; a[j]=z;~~
         std::swap(a[i],a[j]);
      }
   ~~a[h]=a[i+1];~~   a[h]=std::move(a[i+1]);
   ~~a[i+1]=x;~~      a[i+1]=std::move(x);
   return i+1;
}
```

- Example (Combination generation)

```
// Version A – Call by constant lvalue reference
// Cf. C++ as a better C, pp.72~73
int c(int n,int k,const stack<int>& s)
{
   if (k==0||n==k) {
      for (int i=1;i<=k;i++) cout << i << ' ';
      stack<int> t(s);
      while (!t.empty()) {
         cout << t.top() << ' '; t.pop();
      }
      cout << endl;
      return 1;
   } else {
      const_cast<stack<int>&>(s).push(n);
      int r=c(n-1,k-1,s);
      const_cast<stack<int>&>(s).pop();
      return r+c(n-1,k,s);
   }
}
```

`std::move(s)`

unnecessary, but harmless

- Example (Cont'd)

```
int c(int m,int n)
{
   return c(m,n,stack<int>());
}

// Version B – Call by rvalue reference
int c(int n,int k,stack<int>&& s)
{
   if (k==0||n==k) {
      // same as version A
   } else {
      s.push(n);
      int r=c(n-1,k-1,std::move(s));     // a must
      s.pop();
      return r+c(n-1,k,std::move(s));   // a must
   }
}

// Version C – Call by value (copy + move)
// Generate elements in stack s + any k-permutation of {1, ..., n}
int c(int n,int k,stack<int> s)
{
   if (k==0||n==k) {
      // similar to version A, except that no stack copy
      // is needed – just display stack s
   } else {
      s.push(n);
      int r=c(n-1,k-1,s);        // copy; can't be moved
      s.pop();
      return r+c(n-1,k,s);       // copy → move
   }                             // for efficiency
}
```

`std::move(s)`

The stack **s** is no longer needed, so move it!

Copy/move constructor

- Moving (or move-assigning) object **rhs** to object ***this** should satisfy the following properties.

  1 The value of ***this** should be the same as the original value of **rhs**.

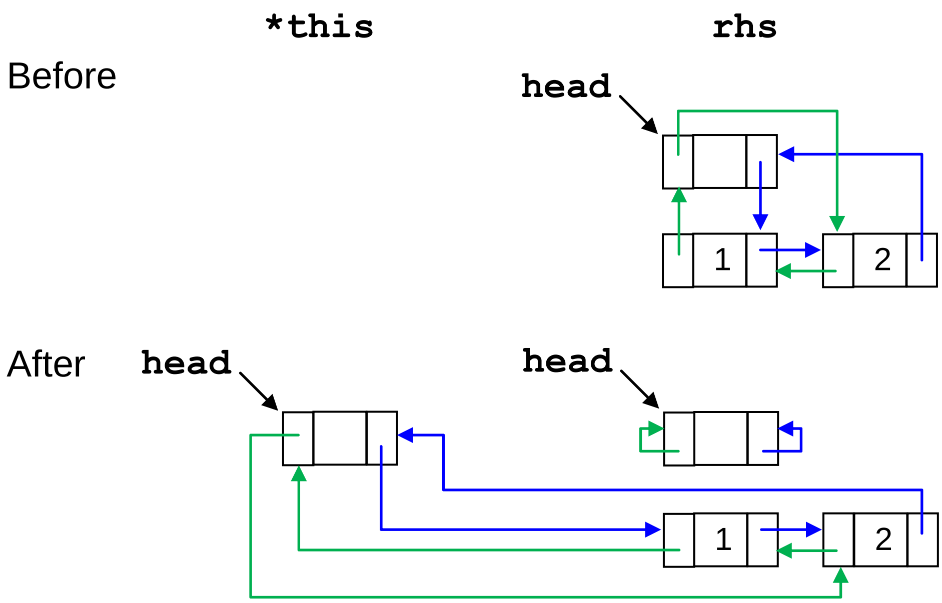
  2 The stolen object **rhs** should be left in a state where it can be correctly manipulated thereafter.
  N.B. The object **rhs** may not be a temporary.

- Example

```
template<typename T>
class deque {
public:
    // other members omitted
    deque(deque&& rhs);          // move ctor
    void push_front(const T&);
private:
    node *head;         // point to a doubly linked list with
};                      // a header node
```

How to define the move ctor?

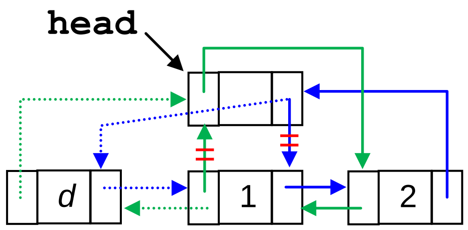
Method A – Steal all but the header node



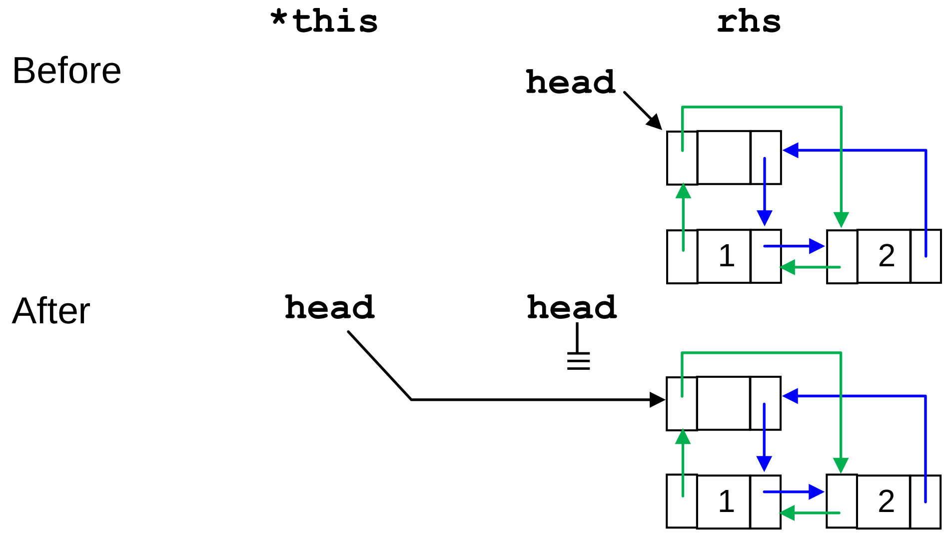
- Example (Cont'd)

Since every object has a header node, all deque operations, e.g. `push_front()`, may be defined with this in mind:

```
template<typename T>
void deque<T>::push_front(const T& d)
{
   head->succ=new node(d,head,head->succ);
   head->succ->succ->pred=head->succ;
}
```



Method B – Steal the header node, too



Since not every object has a header node, all deque operations have to detect it.

```
template<typename T>
void deque<T>::push_front(const T& d)
{
   if (head==nullpr)  // allocate a header node
   // same as above
}
```

Tradeoff: temporary one-node saving vs permanent detection

- If a class doesn't declare a copy ctor, one is implicitly declared as defaulted iff
  - there is no user-declared move ctor
  - there is no user-declared move assignment operator

  Such an implicit declaration is deprecated if
  - there is a user-declared copy assignment operator
  - there is a user-declared dtor

- If a class doesn't declare a move ctor, one is implicitly declared as defaulted iff
  - there is no user-declared copy ctor
  - there is no user-declared copy assignment operator
  - there is no user-declared move assignment operator
  - there is no user-declared dtor

- The implicitly-defined copy/move ctor performs a memberwise copy/move of its members.

- Example

```
template<typename T,typename Container=deque<T> >
class stack {
public:
   // use Container's puch_back(), pop_back(), etc to
   // define stack's push(), pop(), etc.
private:
   Container c;
};
```

The implicitly-defined copy/move ctor below needn't be explicitly defined, since the class doesn't dynamically allocate memory.

```
template<typename T,typename Container>
stack<T,Container>::stack(const stack& rhs)
: c(rhs.c) {}

template<typename T,typename Container>
stack<T,Container>::stack(stack&& rhs)
: c(std::move(rhs.c)) {}
```

N.B. VC++ fails to implicitly generate the move ctor.

Copy/move assignment operator

- If a class doesn't declare a copy assignment operator, one is implicitly declared as defaulted iff
  - there is no user-declared move ctor
  - there is no user-declared move assignment operator

  Such an implicit declaration is deprecated if
  - there is a user-declared copy ctor
  - there is a user-declared dtor

- If a class doesn't declare a move assignment operator, one is implicitly declared as defaulted iff
  - there is no user-declared copy ctor
  - there is no user-declared move ctor
  - there is no user-declared copy assignment operator
  - there is no user-declared dtor

- The implicitly-defined copy/move assignment operator performs a memberwise copy/move assignment of its members.

- Example (Cont'd)

  // implicitly-defined copy ctor
```
template<typename T,typename Container>
stack<T,Container>&
stack<T,Container>::operator=(const stack& rhs)
{
   if (this!=&rhs) c=rhs.c;
   return *this;
}
```

  // implicitly-defined move ctor
```
template<typename T,typename Container>
stack<T,Container>&
stack<T,Container>::operator=(stack&& rhs)
{
   if (this!=&rhs) c=std::move(rhs.c);
   return *this;
}
```

Explicit copy/move ctor

- Example

```
struct o_O {
   o_O() { cout << 1; }
   o_O(const o_O&) { cout << 2; }
   o_O(o_O&&){ cout << 3; }
   ~o_O() { cout << 4; }
};

o_O f(o_O s) { return s; }
int main() { o_O oO; f(oO); }
int main() { f(o_O()); }
```

copy ctor	move ctor	`f(oO)`	`f(o_O())`
-	-	`123444`	`1344`
explicit deleted	-	Illegal call	`1344`
-	explicit deleted	`122444`	`1244`
explicit deleted	explicit deleted	Illegal function	

Initializer-list constructor

- List-initialization for class **T**

  T x={ $a_1, a_2, \cdots, a_n$ **};** // copy initialization
  T x({ $a_1, a_2, \cdots, a_n$ **});** // direct initialization
  T x{ $a_1, a_2, \cdots, a_n$ **};** // direct initialization

  Basically, they are semantically equivalent unless explicit ctors are involved.

  1 If **{}** is empty, call **T::T()**
  2 If **T** has an initializer-list ctor, call **T::T({** $a_1, a_2, \cdots, a_n$ **})**; otherwise, call **T::T(** $a_1, a_2, \cdots, a_n$ **)**

- Example

```
struct X {
   X(int y=0) : x(y) {}
   int x;
};
X a(2);          // call X(2)
X b={2};         // call X(2), but error with explicit ctor
X b({2});        // call X(2)
X b{2};          // call X(2)
X c{2,3,4};      // error, no callable ctor
```

- Example

```
#include <initializer_list>
#include <numeric>
struct X {
   X(int y=0) : x(y) {}                // 1
   X(initializer_list<int> a)          // 2
   : x(accumulate(begin(a),end(a),0,
              [](int x,int y){return x+y;})) {}
   int x;
};
X a(2);          // call X(2)
X b{2};          // call X({2})
X c{2,3,4};      // call X({2,3,4})
```

- Example

```
string::string(initializer_list<char> a)
:  _size(a.size()),
   _capacity(_cap(_size)),
   _data(new char[_capacity+1])
{
   char* p=_data;
   const char* it=begin(a);
   while (it!=end(a)) *p++=*it++;
   *p='\0';
}
```

Deleted definitions

- Example (Class and ADT, p65)

```
class X {
public:
   void p() { X a,b; a=b; }         // compile error
private:                            // redundant
   X& operator=(const X&) = delete;
};
void q() { X a,b; a=b; }            // compile error
```

Comments and move-assignable

1 **X** isn't copy-assignable (by members and non-members).
2 Obviously, the accessibility of a deleted member function is immaterial.
3 A function must be deleted on its first declaration.

```
struct Y { Y(); };
Y::Y() = delete;           // error, not first declaration
int main() {}
```

N.B. VC++ doesn't yet support **delete**. GNU C++ fails to detect this error.
Name this program **Y.cpp** and check it on clang++.
bsd2> clang++ -std=c++11 Y.cpp

- A deleted function also participates in overload resolution.

- Example

```
void p(int) {}               // 1
void p(double) = delete;     // 2, enforce int invocation

p(2);           // case 1: select 1
p('2');         // case 1
p(2.3);         // case 2: select 2, but error
p(2.3f);        // case 2
p(2u);          // case 3: ambiguous
```

Case 1: $T \to$ int — identity or integral promotion
Case 2: $T \to$ double — identity or floating promotion
Case 3: $T \to$ int $\wedge$ $T \to$ double — both are conversions

Comment
Overload resolution has a higher priority than availability and accessibility, i.e.
Overload resolution $\to$ Deleted? Accessible?

- Example (Cont'd)

```
class X {
public:
   void p(int) {}          // 1
private:
   void p(double) {}       // 2
};
X().p(2);       // case 1: select 1
X().p('2');     // case 1
X().p(2.3);     // case 2: select 2, but error
X().p(2.3f);    // case2
X().p(2u);      // case 3: ambiguous
```

Explicitly-defaulted functions

- Only special member functions can be explicitly-defaulted.
  N.B. They are originally implicitly declared as defaulted.
- Example

```
class X {
public:
   void p() { X a,b; a=b; }       // ok
private:
   X& operator=(const X&) = default;
};
void q() { X a,b; a=b; }          // compile error
```

Comments **X** isn't move-assignable by members and non-members

1 **X** is copy-assignable by members only.
2 An explicitly-defaulted special member function is implicitly defined if it is odr-used.
3 The accessibility of an explicitly-defaulted special member function is material.
4 An explicitly-defaulted special member function needn't be declared so on its first declaration.

```
struct Y { Y(); };
inline Y::Y() = default;      // ok
```

Range-based `for` statement: Implementation

- The following function templates are available when any of `<iterator>`, `<deque>`, `<string>`, `<vector>`, `<list>`, etc. is included.

```
template<typename T,size_t N>
T* begin(T (&array)[N]) { return array; }

template<typename T,size_t N>
T* end(T (&array)[N]) { return array+N; }

template<typename Container>
auto begin(Container& c) -> decltype(c.begin())
{ return c.begin(); }
                                    Container::iterator
template<typename Container>
auto end(Container& c) -> decltype(c.end())
{ return c.end(); }
```

  Each has a `const` version that returns a `const_iterator`.

- Motivation

```
int a[3]={1,2,3};
for (int* it=a;it!=a+3;++it) cout << *it;

⇒ for (auto it=begin(a);it!=end(a);++it)
      cout << *it; ⇢ int x=*it; cout << x;

⇒ for (int x : a) cout << x;

vector<int> v(3,5);
for (vector<int>::iterator it=v.begin();
                       it!=v.end();++it) ++*it;

⇒ for (auto it=begin(v);it!=end(v);++it)
      ++*it; ⇢ int& y=*it; ++y;

⇒ for (int& y : v) ++y;
```

- Motivation (Cont'd)

```
string s("Snoopy"),t("Pluto");

for (char& u : s+t) cout << ++u;
⇒ string&& c = s+t;      ⇢ auto && c = s+t;
   for (auto it=begin(c);it!=end(c);++it) {
      char& u=*it;
      cout << ++u;          can't be s+t
   }

for (char& u : s=t) cout << ++u;
⇒ string& c = (s=t);     ⇢ auto && c = s=t;
   // same for loop as above
```

- **for** (*for-range-declaration* **:** *expression* **)** *statement*
  for (*for-range-declaration* **:** *braced-init-list* **)** *statement*
  ≡
  auto && *c* **=** (*expression*) or *braced-init-list* **;**
  for (auto *it* **= begin(***c***)** , *finish* **= end(***c***)** ; *it* **!=** *finish* **; ++***it***) {**
     for-range-declaration = ******it* **;**
     statement
  }

  Comments

  1. If *expression* is an lvalue, *c* is an lvalue reference.
     If *expression* is an rvalue, *c* is an rvalue reference.
  2. **end(***c***)** is evaluated only once – this is useful when *c* is an array (∵ **array+N** is evaluated only once).

- Example   an object of the type **initializer_list<int>**

```
for (int x : {1,2,3}) cout << x;

int a[2][3]={1,2,3,4,5,6};

for (int (&x)[3] : a)      // & is necessary here
   for (int &y : x)        // & is optional here
      cout << y;
```

Move semantics: string addition

- Motivation

  ```
  string s,t,u,v;
  ((s+t)+u)+v      // rvalue + lvalue
  s+(t+(u+v))      // lvalue + rvalue
  (s+t)+(u+v)      // rvalue + rvalue
  ```

- String addition: lvalue + lvalue

  ```
  string operator+(const string& lhs,const string& rhs)
  {
     string s(lhs); s+=rhs; return s;
  }
  ```

  Comment – Compare the following three definitions

  ```
  string s(lhs); s+=rhs; return s;     // copy + elision

  string s(lhs); return s+=rhs;        // copy + copy

  return string(lhs)+=rhs;             // copy + copy
  ```

  How does copy elision work on the return statement?
  return *name*; // *name* isn't a function parameter

  Caller
  cout << s+t; ⇒ **operator+(s,t,***a temporary location***)**
  string u=s+t; ⇒ **operator+(s,t,***location of* **u)**

  Callee – *Let x be the caller-provided location*

  ```
  string operator+(lhs,rhs, x)
  {
     ~~string s(lhs); s+=rhs; return s;~~
     string x(lhs); x+=rhs;
  }
  ```

  Q: Why can't the *name* be a function parameter?

- String addition: rvalue + lvalue

```
string operator+(string&& lhs,const string& rhs)
{
   return std::move(lhs+=rhs);
}
```

Comment – Compare the following three definitions

```
return std::move(lhs+=rhs);            // move

lhs+=rhs; return std::move(lhs);       // move

string s(std::move(lhs));              // move + elision
s+=rhs; return s;
```

For the last definition, don't write

```
return std::move(s);  // exra move; elision doesn't apply
```

- String addition: lvalue + rvalue

```
string operator+(const string& lhs,string&& rhs)
{
   return std::move(rhs.insert(0,lhs));
}
```

- String addition: rvalue + rvalue

```
string operator+(string&& lhs,string&& rhs)
{
   return std::move(lhs+=rhs);
}
```

or, equivalently,

```
return std::move(rhs.insert(0,lhs));
```

Canonical assignment

- A "canonical implementation" of copy assignment

```
string& string::operator=(string rhs)
{
   swap(*this,rhs);
   return *this;
}
```

where **swap** is a **string** specialized algorithm:

```
void string::swap(string& rhs)        // member
{
   std::swap(_size,rhs._size);        // generic swap
   std::swap(_capacity,rhs._capacity);
   std::swap(_data,rhs._data);
}

void swap(string& lhs,string& rhs)   // specialized
{
   lhs.swap(rhs);
}
```

- Comments

1 This is a copy assignment operator, since its parameter is passed by value.

2 Pro
Easy to define
Needn’t define a move assignment operator
(In fact, if you do so, an assignment would be ambiguous, because of the coexistence of call-by-value and call-by-reference.)

3 Con
Inefficiency (See the comparisons below)

- Canonical copy assignment vs C++11-style copy assignment

  ```
  string a("snoopy"),b("pluto");
  a=b;
  ```

  Canonical copy assignment
  1 copy **b** // copy **b** to **rhs**
  2 swap // swap **a** and **rhs** (= **b**)
  3 delete a // delete **rhs** (= original **a**)

  C++11-style copy assignment
  1 self-assignment test
  2 delete **a**
  3 copy **b** // copy **b** to **a**

  Since swapping two strings takes 9 operations, the swap is more expensive than the test.

- Canonical copy assignment vs C++11-style move assignment

  ```
  a=move(b);
  ```

  Canonical copy assignment
  1 move **b** // move **b** to **rhs**
  2 swap // swap **a** and **rhs** (= **b**)
  3 delete **a** // delete **rhs** (= original **a**)

  C++11-style move assignment
  1 self-assignment test
  2 delete **a**
  3 move **b** // move **b** to **a**

  Again, the swap is more expensive than the test.

- More example …

  ```
  vector<string> a(9,"snoopy"),b(8,"pluto");
  a=b;
  ```

  The canonical copy assignment for **vector** has to manage heap storage (∵ passing vector **b** by value). But, the C++11-style copy assignment needn't (∵ passing vector **b** by reference and vector **a** has enough capacity.)

Iterators

Iterator traits

- An iterator has five traits, namely, iterator_category, value_type, difference_type, pointer, and reference.

- An iterator class has to define its traits as typedef names.

  Example

```
template<typename T>
class list<T>::iterator {
   typedef bidirectional_iterator_tag
                               iterator_category;
   typedef T                   value_type;
   typedef ptrdiff_t           difference_type;
   typedef T*                  pointer;
   typedef T&                  reference;
};
```

- The traits of a pointer type are defined in a specialization of `iterator_traits`,

```
template<typename T>
struct iterator_traits<T*> {
   typedef random_access_iterator_tag
                               iterator_category;
   typedef T                   value_type;
   typedef ptrdiff_t           difference_type;
   typedef T*                  pointer;
   typedef T&                  reference;
};
```

- **`iterator_traits`** helps finding out the traits of an iterator class or a pointer type.

```
template<typename Iterator>
struct iterator_traits {
    typedef typename Iterator::iterator_category
            iterator_category;
    typedef typename Iterator::value_type
            value_type;
    typedef typename Iterator::difference_type
            difference_type;
    typedef typename Iterator::pointer
            pointer;
    typedef typename Iterator::reference
            reference;
};
```

Standard iterator tags

- In STL, there are 5 category tag classes defined as

```
struct input_iterator_tag {};

struct output_iterator_tag {};

struct forward_iterator_tag
: public input_iterator_tag {}; // public inheritance

struct bidirectional_iterator_tag
: public forward_iterator_tag {};

struct random_access_iterator_tag
: public bidirectional_iterator_tag {};
```

where the public inheritances means that

a random access iterator is a bidirectional iterator, which in turn is a forward iterator, which in turn is an input iterator.

- These category tag classes are used as compile time tags for algorithm selection.

- Example

distance() is well defined for input iterators, but can be implemented more efficiently for random access iterators.

```
template<typename InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first,InputIterator last)
{
   return distance(first,last,typename
   iterator_traits<InputIterator>::iterator_category());
}

template<typename InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first,InputIterator last,
                                   input_iterator_tag)
{
   typename
   iterator_traits<InputIterator>::difference_type d=0;
   for (InputIterator it=first;it!=last;++it)
      d++;
}

template<typename RandomAccessIterator>
typename iterator_traits<InputIterator>::difference_type
distance(RandomAccessIterator first,
         RandomAccessIterator last,
         random_access_iterator_tag)
{
   return last-first;
}
```

Comment

If **InputIterator** isn't a random access iterator, **last** shall be reachable from **first** using **opeartor++**.

- Example

```
template<class InputIterator,class Distance>
void advance(InputIterator& it,Distance n)
{
   advance(it,n,typename
    iterator_traits<InputIterator>::iterator_category());
}
```

must ≥ 0

```
template<typename InputIterator>
void advance(InputIterator& it,Distance n,
                              input_iterator_tag)
{  while (n>0) { ++it; --n; } }

template<typename BidirectionalIterator>
void advance(BidirectionalIterator& it,Distance n,
                        bidirectional_iterator_tag)
{
   if (n>0) while (n>0) { ++it; --n; }
   else while (n<0) { --it; ++n; }
}

template<typename RandomAccessIterator>
void advance(RandomAccessIterator& it,Distance n,
                        random_access_iterator_tag)
{  it+=n; }
```

- Example

```
forward_list<string> xs{"Snoopy","Pluto","Garfield"};
cout << distance(xs.begin(),xs.end());   // 3
cout << distance(xs.end(),xs.begin());   // no
forward_list<string>::iterator it=xs.begin();
advance(it,2); cout <<*it;
advance(it,-2);                          // no

vector<string> zs(xs.begin(),xs.end());
cout << distance(zs.begin(),zs.end());   // 3
cout << distance(zs.end(),zs.begin());   // -3
```

Move iterator

- **move_iterator** is an iterator adaptor with the same behavior as the underlying iterator except that it turns the value of **operator*()** from lvalue to rvalue reference.

- Here is an abridged version of **move_iterator**

```
template<typename Iterator>
class move_iterator {
public:
// itearor traits
    typedef typename iterator_traits<Iterator>::value_type
            value_type;
    typedef value_type&& reference;
// members
    move_iterator(Iterator it) : current(it) {}
    reference operator*() const      // ≠ Iterator
    { return std::move(*current); }
    move_iterator& operator++()      // = Iterator
    { ++current; return *this; }
private:
    Iterator current;
};
```

- A non-member help function

```
move_iterator<Iterator>
make_move_iterator(const Iterator& i) {
    return move_iterator<Iterator>(i);
};
```

- Example

```
list<string> xs(3,"Snoopy");
list<string> ys(make_move_iterator(xs.begin()),
                make_move_iterator(xs.end()));
cout << xs.size() << ys.size();    // 33
for (const string& s : xs) cout << s;
for (const string& s : ys) cout << s;
```