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.

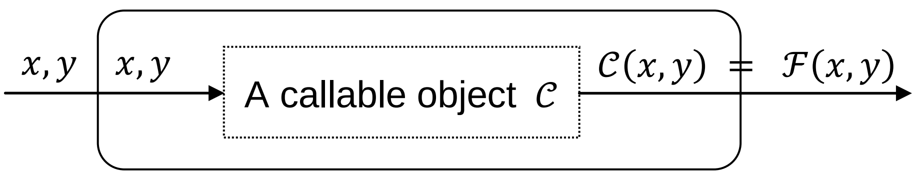
- Example

  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.




**`bfn`** object $\mathcal{F}$

```
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

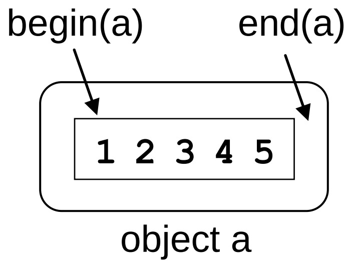
- List-initialization is the initialization of an object from a braced initializer list.
- Narrowing conversions are not allowed at the top level in list-initializations.
- Example

```
// 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 $\{1, \ldots, n\}$

```
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.

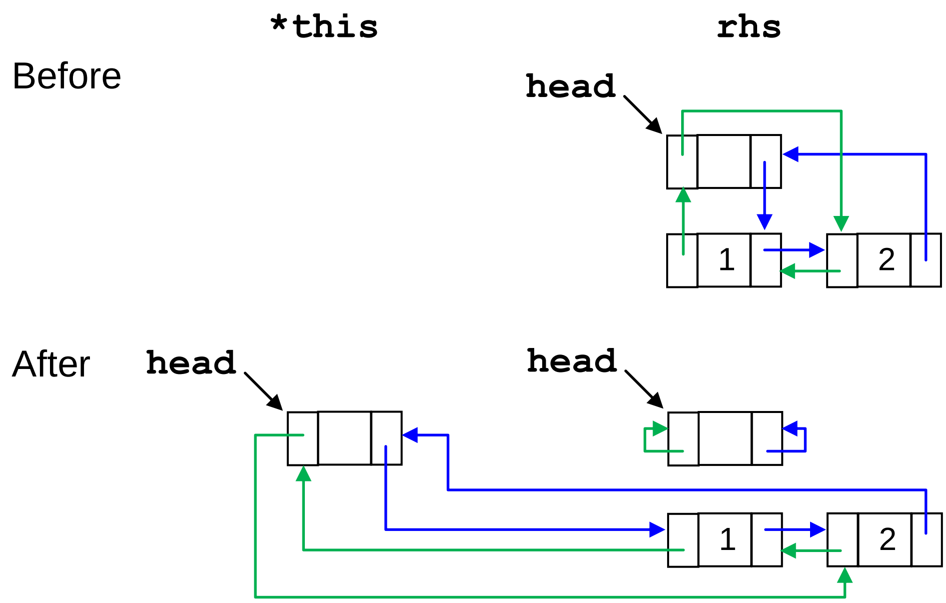
  1 The value of **`*this`** should be the same as the original value of **`rhs`**.
  2 The stolen object **`rhs`** should be left in a state where it can be correctly manipulated thereafter.
  N.B. The object **`rhs`** may not be a temporary.

- 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?

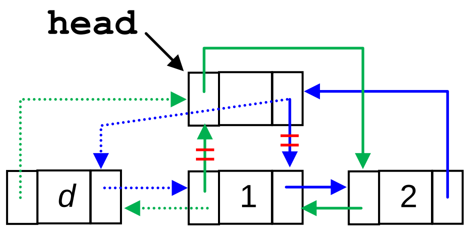
Method A – Steal all but the header node



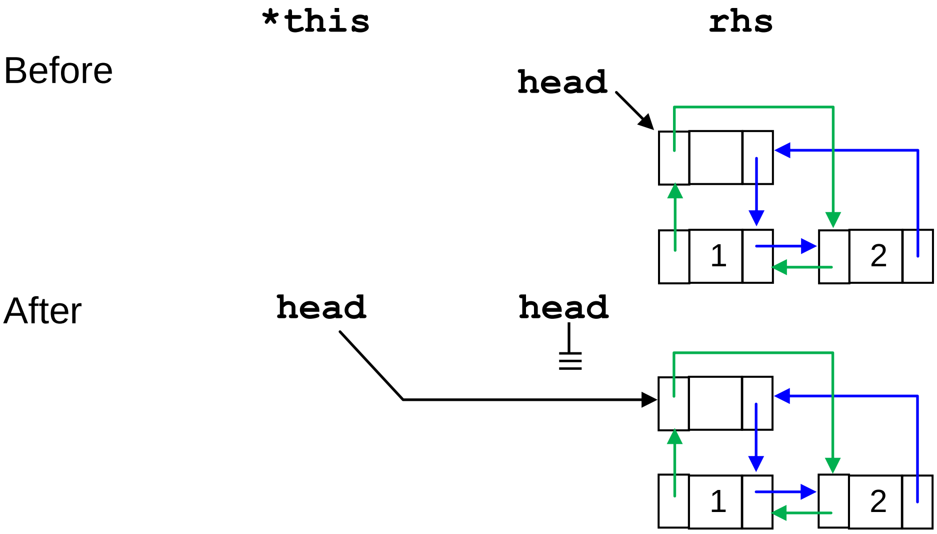
- Example (Cont'd)

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

# 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

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: | $T \rightarrow$ int | identity or integral promotion |
| Case 2: | $T \rightarrow$ double | identity or floating promotion |
| Case 3: | $T \rightarrow$ int $\wedge$ $T \rightarrow$ double | both are conversions |

Comment
Overload resolution has a higher priority than availability and accessibility, i.e.
Overload resolution $\rightarrow$ 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

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
```