SCHOOL OF SCIENCE
SCHOOL OF ENGINEERING
SCHOOL OF BUSINESS AND MANAGEMENT
HKUST INFO DAY
ONE UNIVERSITY, INFINITE POSSIBILITIES
SCHOOL OF HUMANITIES AND SOCIAL SCIENCE
INTERDISCIPLINARY PROGRAMS OFFICE

COMP 2012H Honors Object-Oriented Programming and Data Structures

Topic 17: rvalue Reference and Move Semantics

Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

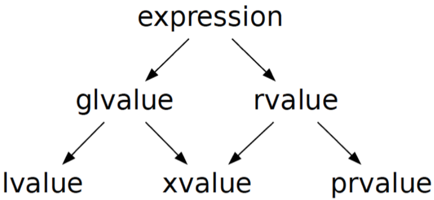
Review: lvalue & rvalue of a Variable

- A variable is a symbolic name assigned to some memory storage.
- The difference between a variable and a literal constant is that a variable is addressable. E.g., $\boxed{x = 100;}$ x is a variable and 100 is a literal constant; x has an address and 100 doesn't.
- A variable has dual roles, depending on where it appears.

```
x = x + 1;
```

  - lvalue: its location (read-write)
  - prvalue (pure rvalue) [C++11]: its value (read-only)

```
int x;          // OK
4 = 1;          // Error! Why?
(x + 10) = 6; // Error! Why?
```



Standard for Programming Language C++ Section 3.10

- An lvalue (so called, historically, because lvalues could appear on the left-hand side of an assignment expression) designates a function or an object.
  - Example: If E is an expression of pointer type, then *E is an lvalue expression referring to the object or function to which E points. As another example, the result of calling a function whose return type is an lvalue reference is an lvalue.
- An xvalue (an "eXpiring" value) also refers to an object, usually near the end of its lifetime (so that its resources may be moved, for example). An xvalue is the result of certain kinds of expressions involving rvalue references.
  - Example: The result of calling a function whose return type is an rvalue reference is an xvalue.
- A glvalue (generalized lvalue) is an lvalue or an xvalue.
- An rvalue (so called, historically, because rvalues could appear on the right-hand side of an assignment expressions) is an xvalue, a temporary object or subobject thereof, or a value that is not associated with an object.
- A prvalue ("pure" rvalue) is an rvalue that is not an xvalue.
  - Example: The result of calling a function whose return type is not a reference is a prvalue. The value of a literal such as 12, 7.3e5, or true is also a prvalue.

Part I

Temporary Objects and rvalue References

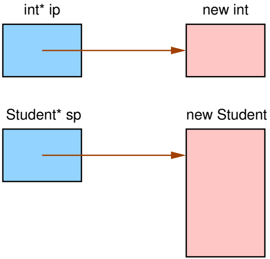
Unnamed Objects I: Dynamically Allocated Objects/Values

Syntax: Pointer Variable Definition

T* <variable> = <dynamic object>;

Examples of Pointers

```
int* ip = new int;
Word* wp = new Word;
Student* sp = new Student;
```



Dynamic objects allocated and returned by the new operator are unnamed. You need to use pointers to hold them.

- Dynamic objects are managed by the heap.
- If you lose all pointers to a dynamic object, you lose the object — resulting in a memory leak.

Unnamed Objects II: Temporary Objects/Values

Syntax: rvalue Reference Definition

T&& <variable> = <temporary object>;

Temporary objects/values are another kind of unnamed objects/values created automatically on the stack during

{TO1} const reference initialization

{TO2} argument passing (e.g., type conversion)

{TO3} function returned value (by copying)

{TO4} evaluation of expressions (e.g., result of sub-expressions)

- Temporary objects are managed by the stack.
- They are destructed automatically by the stack when they are no longer needed.
- An rvalue reference is an alias of a temporary object/value.

Before C++11: const T&

Syntax before C++11

const **T**& <variable> = <temporary object>;

- In the past, you may prolong the life of a temporary object by assigning it to a const reference.
- You can't modify a temporary object through its const reference because a temporary object is considered as an rvalue.
- Now C++11 allows you to create a rvalue reference to hold temporary objects so that you may explicitly manipulate them in some safe ways.
- Once created as an alias of a temporary object, an rvalue reference variable is just like a regular lvalue variable: it has both the roles of lvalue or prvalue of the temporary object, depending on where it is used.

Temporary Values 1, 3, 4 with Basic Types

```
 1  #include <iostream>     /* File: TO-int.cpp */
 2  using namespace std;
 3
 4  int square(int x) { return x*x; }
 5  void cbv(int x) { cout << "call-by-value: " << x << endl; }
 6  void cbr(int& x) { cout << "call-by-ref: " << x << endl; }
 7  void cbcr(const int& x) { cout << "call-by-const-ref: " << x << endl; }
 8
 9  int main()
10  {
11      int a = 3;
12      int& b = 4;                   // Error! Why?
13      const int& c = 5;             // TO1: const ref initialization
14      int d = square(3);            // TO3: function returned value
15      int e = a + c + d;            // TO4: result of sub-expression
16      cbv(a);                       // OK: int x = a
17      cbr(a);                       // OK: int& x = a
18      cbr(8);                       // Error: int& x = 8
19      cbcr(8); return 0;            // TO1: const int& x = 8
20  }
```

- lvalue reference only binds to another lvalue.
- const lvalue reference accepts an rvalue because a temporary value is created which can be referenced (lines #13, #19).

Class Word: word.h I

```
#include <iostream>     /* File: word.h */
#include <cstring>
using namespace std;

class Word
{
  private:
    int freq = 0;
    char* str = nullptr;

  public:
    Word() { cout << "default constructor" << endl; }

    Word(const char* s, int f = 1) : freq(f), str(new char [strlen(s)+1])
        { strcpy(str, s); cout << "conversion: "; print(); }

    Word(const Word& w) : freq(w.freq), str(new char [strlen(w.str)+1])
        { strcpy(str, w.str); cout << "copy: "; print(); }

    ~Word() { cout << "destructor: "; print(); delete [] str; }

    void print() const
        { cout << (str ? str : "null") << " ; " << freq << endl; }
```

Class Word: word.h II

```
    Word operator+(const Word& w) const
    {
        cout << "\n~~~ " << str << " + " << w.str << " ~~~\n";
        Word x;         // Which constructor?

        x.freq = freq + w.freq;
        x.str = new char [strlen(str) + strlen(w.str) + 1];
        strcpy(x.str, str);
        strcat(x.str, w.str);

        return x;       // How is x returned?
    }

    Word to_upper_case() const
    {
        Word x(*this);  // Which constructor?

        for (char* p = x.str; *p != '\0'; p++)
            *p += 'A' - 'a';

        return x;       // How is x returned?
    }
};
```

Temporary Objects with User-defined Types: TO-word.cpp

```
#include "word.h"       /* File: TO-word.cpp */

void print_word(const Word& x)
{
    cout << "<<<\n"; x.print(); cout << ">>>\n";
}

int main()
{
    const Word& w1 = "batman";    // TO1: const ref initialization
    w1.print();
    print_word("superman");       // TO2: argument passing

    Word w2 = w1.to_upper_case(); // TO3: function returned value
    w2.print();
    (w1 + " or " + w2).print();   // TO4: result of sub-expression

    cout << "\n*** It's all destructions now ***" << endl;
    return 0;
} /* g++ -std=c++11 -fno-elide-constructors TO-word.cpp */
```

TO-word.cpp Output

```
conversion: batman ; 1
batman ; 1
conversion: superman ; 1
<<<
superman ; 1
>>>
destructor: superman ; 1
copy: batman ; 1
copy: BATMAN ; 1
destructor: BATMAN ; 1
copy: BATMAN ; 1
destructor: BATMAN ; 1
BATMAN ; 1
conversion:  or  ; 1
```

```
~~~ batman +  or  ~~~
default constructor
copy: batman or  ; 2
destructor: batman or  ; 2

~~~ batman or  + BATMAN ~~~
default constructor
copy: batman or BATMAN ; 3
destructor: batman or BATMAN ; 3
batman or BATMAN ; 3
destructor: batman or BATMAN ; 3
destructor: batman or  ; 2
destructor:  or  ; 1

*** It's all destructions now ***
destructor: BATMAN ; 1
destructor: batman ; 1
```

Temporary Objects of User-defined Types: Remarks

- Temporary **Word** objects are created on lines #10, #12, #14, and #16.
- On lines #10 and #12, C-strings are converted to temporary **Word** objects which are then bound to the `const Word&`.
- `w1.to_upper_case()` returns a temporary **Word** object that is copied to `w2`.
- `(w1 + " or")` returns a temporary **Word** object which is added to `w2`.
- `(w1 + " or " + w2)` returns another temporary **Word** object which calls `print()`.
- The lifetime of a temporary **Word** object is at the end of the expression that creates it unless it is held by a rvalue reference or const reference.
- A temporary object that is held by a rvalue reference or const reference dies as its reference variable goes out of scope.

rvalue Reference && (C++11) for int

```
#include <iostream>     /* File: rvalue-ref-int.cpp */
using namespace std;

int square(int x) { return x*x; }

int main()
{
    /* rvalue reference with values of basic types */
    int a = 8;
    int&& b;            // Error: rvalue ref must be initialized
    int&& c = a;        // Error: rvalue ref can't bind to lvalue

    int&& d = 5; cout << d << endl;
    int&& e = square(5); cout << e << endl;

    d = e = 10;                             // d, e used as lvalues
    cout << d << '\t' << e << endl << endl; // d, e used as rvalues
    return 0;
}
```

rvalue Reference && (C++11) for string

```
#include <iostream>     /* File: rvalue-ref-string.cpp */
using namespace std;

string wrap(string s) { return "begin." + s + ".end";  }

int main()
{
    /* rvalue reference with user-defined objects */
    string s1 {"w"};
    string&& s2;        // Error: rvalue ref must be initialized
    string&& s3 = s1;   // Error: rvalue ref can't bind to lvalue

    string&& s4 = "x"; cout << s4 << endl;
    string&& s5 = wrap("x"); cout << s5 << endl;

    s4 = "z";           // s4 used as lvalue
    cout << s4 << endl; // s4 used as rvalue
    s5 = s1;            // s5 used as lvalue
    cout << s5 << endl; // s4 used as rvalue
    return 0;
}
```

rvalue Reference && to Hold Temporary Objects

- The term rvalue reference sounds contradictory as it seems to be a reference to an rvalue! In the past,
    - A reference (alias) can only be created for an lvalue which is mutable.
    - Temporary objects are treated as rvalues as they are not supposed to be changed. Why would you want to modify a temporary object which will disappear soon?
- An rvalue reference allows you to give a name to a temporary object, manipulate it, and modify it if it is safe to do so.
- rvalue references are mainly used for real "objects" to improve code efficiency in certain scenarios (e.g., move operations).
- Like its lvalue reference counterpart, an rvalue reference
    - must be initialized when it is created
    - once bound, cannot be re-bound to another temporary object
- An rvalue reference cannot be bound to an lvalue but only to a temporary object.

Temporary Word Objects and rvalue Reference

```
#include "word.h"       /* File: temp-word.cpp */
void print_word(const Word& w) { cout << "print const Word&: "; w.print(); }
void print_word(Word&& w) { cout << "print Word&&: "; w.print(); }

int main()
{
    /* Use const Word& to hold a temporary Word object */
    Word song("imagine"); cout << endl;
    const Word& w1 = song.to_upper_case(); cout << endl;
    song.print(); w1.print(); cout << endl;

    /* Use Word&& to hold a temporary Word object */
    Word movie("batman", 2); cout << endl;
    Word&& w2 = movie.to_upper_case(); cout << endl;
    movie.print(); w2.print(); cout << endl;

    print_word(song); print_word(movie);
    print_word(w1); print_word(w2); cout << endl;

    /* Directly pass a temporary Word object to a function */
    print_word("Beatles"); cout << endl;
    print_word(movie.to_upper_case()); cout << endl; return 0;
} /* g++ -std=c++11 -fno-elide-constructors temp-word.cpp */
```

Temporary Word Objects and rvalue Reference: Output

```
conversion: imagine ; 1

copy: imagine ; 1
copy: IMAGINE ; 1
destructor: IMAGINE ; 1

imagine ; 1
IMAGINE ; 1

conversion: batman ; 2

copy: batman ; 2
copy: BATMAN ; 2
destructor: BATMAN ; 2

batman ; 2
BATMAN ; 2
```

```
print const Word&: imagine ; 1
print const Word&: batman ; 2
print const Word&: IMAGINE ; 1
print const Word&: BATMAN ; 2

conversion: Beatles ; 1
print Word&&: Beatles ; 1
destructor: Beatles ; 1

copy: batman ; 2
copy: BATMAN ; 2
destructor: BATMAN ; 2
print Word&&: BATMAN ; 2
destructor: BATMAN ; 2

destructor: BATMAN ; 2
destructor: batman ; 2
destructor: IMAGINE ; 1
destructor: imagine ; 1
```

const lvalue Reference vs. rvalue Reference

Similarities:

- Both **const T&** and **T&&** can be bound to a temporary value/object.
- Both are references and must be initialized when they are created.

Differences:

- **const T&** can't be modified but **T&&** can be. In fact, once created, an **T&&** can be used like a regular variable.
- f(**const T&**) can take almost any arguments: (const) rvalue/lvalue, temporary value/object, and even rvalue reference!
- f(**T&&**) can take only temporary value/object.
- If you have both f(**const T&**) and f(**T&&**), and the input argument is a temporary value/object ⇒ **T&&**.

Part II

Move Semantics

The move Trick with rvalue References

- A temporary object is not supposed to be used after it is read.
- Trick: So we can cheat while reading it and steal its resources.
- However, there is a catch: since the temporary object will be destructed after it is used, it must be left in a state where its destructor can be safely called.
- Example: instead of implementing deep copy in a copy constructor, we now may have a move constructor which will simply move (sometimes swap) resources from its input argument *if* it is a temporary object of the same class.
  ⇒ more efficient as no memory allocation is needed.
- Similarly, the trick may be used to define a move assignment operator instead of a copy assignment operator.
- The normal copy constructors and copy assignment operators are still useful if the input argument must be preserved and cannot be modified on return.

Move Constructor and Move Assignment I

```
#include <iostream>     /* File: word-move.h */
#include <cstring>
using namespace std;

class Word
{
  private:
    int freq = 0; char* str = nullptr;
  public:
    Word() { cout << "default constructor" << endl; }
    Word(const char* s, int f = 1) : freq(f), str(new char [strlen(s)+1])
        { strcpy(str, s); cout << "conversion: "; print(); }
    Word(const Word& w) : freq(w.freq), str(new char [strlen(w.str)+1])
        { strcpy(str, w.str); cout << "copy: "; print(); }
    Word(Word&& w) : freq(w.freq), str(w.str)    // Move constructor
        { w.freq = 0; w.str = nullptr; cout << "move: "; print(); }
    ~Word() { cout << "destructor: "; print(); delete [] str; }

    Word to_upper_case() const
    {
        Word x(*this);
        for (char* p = x.str; *p != '\0'; p++) *p += 'A' - 'a';
        return (x);               // Return-by-value now done by move!
    }
```

Move Constructor and Move Assignment II

```
    void print() const
        { cout << (str ? str : "null") << " ; " << freq << endl; }

    Word& operator=(const Word& w) { // Copy assignment
        if (this != &w) {            // No assignment for the same Word
            delete [] str;
            str = new char [strlen(w.str)+1];
            freq = w.freq; strcpy(str, w.str);
            cout << "copy assignment: "; print();
        }
    return *this;
    }

    Word& operator=(Word&& w) {   // Move assignment
        if (this != &w) {         // No assignment for the same Word
            delete [] str;
            freq = w.freq; str = w.str;
            w.freq = 0; w.str = nullptr;
            cout << "move assignment: "; print();
        }
    return *this;
    }
};
```

Move Constructor and Move Assignment ..

```
#include "word-move.h"     /* File: "word-move.cpp" */

void print_word(const Word& w) { cout << "print const Word&: "; w.print(); }
void print_word(Word&& w) { cout << "print Word&&: "; w.print(); }

int main()
{
    cout << "*** Copy Semantics ***" << endl;
    Word book {"batman"};
    Word movie(book);
    Word song("imagine");
    movie = song;
    print_word(book); cout << endl;

    cout << "*** Move Semantics ***" << endl;
    Word novel {"outliers"}; cout << endl;
    Word novel2 = novel.to_upper_case();    // move constructions
    cout << endl; novel.print(); novel2.print(); cout << endl;

    Word band = "Beatles"; cout << endl;    // move construction
    band = "Eagles"; cout << endl;          // move assignment

    cout << "*** It's all destructions now ***" << endl;
    return 0;
} /* g++ -std=c++11 -fno-elide-constructors word-move.cpp */
```

Move Constructor and Move Assignment: Output

```
*** Copy Semantics ***
conversion: batman ; 1
copy: batman ; 1
conversion: imagine ; 1
copy assignment: imagine ; 1
print const Word&: batman ; 1

*** Move Semantics ***
conversion: outliers ; 1

copy: outliers ; 1
move: OUTLIERS ; 1
destructor: null ; 0
move: OUTLIERS ; 1
destructor: null ; 0
outliers ; 1
OUTLIERS ; 1

conversion: Beatles ; 1
move: Beatles ; 1
destructor: null ; 0

conversion: Eagles ; 1
move assignment: Eagles ; 1
destructor: null ; 0

*** It's all destructions now ***
destructor: Eagles ; 1
destructor: OUTLIERS ; 1
destructor: outliers ; 1
destructor: imagine ; 1
destructor: imagine ; 1
destructor: batman ; 1
```

std::move( ) — Casting Into rvalue Reference

Syntax: Casting into rvalue Reference

std::move(lvalue object) $\equiv$ rvalue reference of the object

- A standard C++ library function.
- The function std::move() actually does NOT move anything.
- It only does static casting.

std::move( ) Example: word-pair.h

```
#include "word-move.h"  /* File: word-pair.h */
class Word_Pair
{
  private:
    Word w1; Word w2;

  public:
    // Pass by const&, construct by copying
    Word_Pair(const Word& a, const Word& b) : w1(a), w2(b)
        { cout << "-- Copy inputs --\n"; a.print(); b.print(); }

    // Pass by &, construct by moving
    Word_Pair(Word& a, Word& b) : w1(std::move(a)), w2(std::move(b))
        { cout << "-- Move with inputs --\n"; a.print(); b.print(); }

    // Pass by rvalue reference &&, construct by moving
    Word_Pair(Word&& a, Word&& b) : w1(std::move(a)), w2(std::move(b))
        { cout << "-- Another move with inputs --\n"; a.print(); b.print(); }

    void print() const
    {
        cout << "word1 = "; w1.print();
        cout << "word2 = "; w2.print();
    }
};
```

std::move( ) Example: word-pair1.cpp

```cpp
#include "word-pair.h"     /* File: "word-pair1.cpp" */

int main()
{
    cout << "\n*** Print the book's info ***" << endl;
    Word author { "Stephen Hawking" };
    Word title { "Brief History of Time" };
    Word_Pair book { author, title };
    book.print();

    cout << "\n*** Print the book2's info ***" << endl;
    Word_Pair book2 { book }; // Really memberwise copy
    book2.print();

    cout << "\n*** Print the couple's info ***" << endl;
    Word husband { "Mr. C++" };
    Word wife { "Mrs. C++" };
    Word_Pair couple { std::move(husband), std::move(wife) };
    couple.print();

    cout << "\n*** It's all destructions now ***" << endl;
    return 0;
} /* g++ -std=c++11 word-pair1.cpp */ // What is the output?
```

std::move( ) Example: word-pair1.cpp Output

```
*** Print the book's info ***
conversion: Stephen Hawking ; 1
conversion: Brief History of Time ; 1
move: Stephen Hawking ; 1
move: Brief History of Time ; 1
-- Move with inputs --
null ; 0
null ; 0
word1 = Stephen Hawking ; 1
word2 = Brief History of Time ; 1

*** Print the book2's info ***
copy: Stephen Hawking ; 1
copy: Brief History of Time ; 1
word1 = Stephen Hawking ; 1
word2 = Brief History of Time ; 1
```

```
*** Print the couple's info ***
conversion: Mr. C++ ; 1
conversion: Mrs. C++ ; 1
move: Mr. C++ ; 1
move: Mrs. C++ ; 1
-- Another move with inputs --
null ; 0
null ; 0
word1 = Mr. C++ ; 1
word2 = Mrs. C++ ; 1

*** It's all destructions now ***
destructor: Mrs. C++ ; 1
destructor: Mr. C++ ; 1
destructor: null ; 0
destructor: null ; 0
destructor: Brief History of Time ; 1
destructor: Stephen Hawking ; 1
destructor: Brief History of Time ; 1
destructor: Stephen Hawking ; 1
destructor: null ; 0
destructor: null ; 0
```

word-pair1.cpp Output Explained

```
Word_Pair(const Word& a, const Word& b): w1(a), w2(b) ...

Word_Pair(Word& a, Word& b): w1(std::move(a)), w2(std::move(b)) ...
```

- word-pair1::line#8: the construction of **Word_Pair book** has 2 choices above, but the 2nd constructor has a higher precedence as the arguments match exactly.
- word-pair1::line#12: **Word_Pair book2** is created by the compiler-generated copy constructor of **Word_Pair**, which will do memberwise copy for each of w1 and w2.
- word-pair1::line#18: by converting the arguments **husband** and **wife** to their rvalue references, **Word_Pair couple** is created by the 3rd constructor in word-pair.h.
- Temporary objects are destructed at the end of the expression creating them unless they are held by rvalue/const references.
- Non-temporary objects are destructed in the reverse order of their constructions.

std::move( ) Example: word-pair2.cpp

```
#include "word-pair.h"     /* File: "word-pair2.cpp" */

int main()
{
    cout << "\n*** Print the synonym's info ***" << endl;
    Word_Pair synonym { Word("happy"), Word("delighted") };
    synonym.print();

    cout << "\n*** Print the const name's info ***" << endl;
    const Word first_name { "Albert" };
    const Word last_name { "Einstein" };
    Word_Pair name { first_name, last_name };
    name.print();

    cout << "\n*** It's all destructions now ***" << endl;
    return 0;
} /* g++ -std=c++11 word-pair2.cpp */ // What is the output?
```

std::move( ) Example: word-pair2.cpp Output

```
*** Print the synonym's info ***
conversion: happy ; 1
conversion: delighted ; 1
move: happy ; 1
move: delighted ; 1
-- Another move with inputs --
null ; 0
null ; 0
destructor: null ; 0
destructor: null ; 0
word1 = happy ; 1
word2 = delighted ; 1
```

```
**** Print the const name's info ***
conversion: Albert ; 1
conversion: Einstein ; 1
copy: Albert ; 1
copy: Einstein ; 1
-- Copy inputs --
Albert ; 1
Einstein ; 1
word1 = Albert ; 1
word2 = Einstein ; 1

*** It's all destructions now ***
destructor: Einstein ; 1
destructor: Albert ; 1
destructor: Einstein ; 1
destructor: Albert ; 1
destructor: delighted ; 1
destructor: happy ; 1
```

Summary: Compiler-generated Member Functions (Again)

Unless you define the following, they will be implicitly generated by the compiler for you (under some conditions):

1. default constructor
   (but only if you don't define other constructors)
2. default copy constructor
3. default (copy) assignment operator function
4. default move constructor (C++11)
5. default move assignment operator function (C++11)
6. default destructor

C++11 allows you to explicitly generate or not generate them:

- to generate: = default;
- not to generate: = delete;

That's all!

Any questions?