More Classes

Week 3

Copyright © 2017 Pearson Education, Ltd.

Operator Overloading

Copyright © 2017 Pearson Education, Ltd.

Operator overloading

- Operators such as =, +, and others can be redefined when used with objects of a class
- Overloading refers to multiple meanings of the same name or symbol.
  - Name overloading : overloaded function.
  - Symbol overloading : overloaded operator.
- An operator is a symbol that tells the compiler to perform specific mathematical, logical manipulations, or some other special operation.
  - arithmetic operator: + , −, ∗, /
  - logical operator: && and ||
  - pointer operator: & and ∗
  - memory management operator: new, delete[ ]
- A binary operator is an operator that takes two operands
- A unary operator is one that takes one operands
- Operator overloading refers to the multiple definitions of an operator.
- Arithmetic operator such as + and / are already overloaded in C/C++ for different built-in types (2 / 3) result is 0 while (2 / 3.0) result is 0.666667

Copyright © 2017 Pearson Education, Ltd.

Operator overloading

- The name of the function for the overloaded operator is operator followed by the operator symbol, *e.g.*,
- **operator+** to overload the + operator, and
- **operator=** to overload the = operator
- Prototype for the overloaded operator goes in the declaration of the class that is overloading it
- Overloaded operator function definition goes with other member functions

Operators that can be overloaded

+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operators that cannot be overloaded

.	.*	::	?:	sizeof

Copyright © 2017 Pearson Education, Ltd.

Restrictions on Operator overloading

- Overloading restrictions
  - Precedence of an operator cannot be changed
  - Associativity of an operator cannot be changed
  - Arity (number of operands) cannot be changed
  - Unary operators remain unary, and binary operators remain binary
  - Operators &, *, + and - each have unary and binary versions
  - Unary and binary versions can be overloaded separately
- No new operators can be created (Use only existing operators)
- No overloading operators for built-in types
  - Cannot change how two integers are added
  - Produces a syntax error

Copyright © 2017 Pearson Education, Ltd.

Member vs Non-member

- Member vs non-member
  - Operator functions can be member or non-member functions
  - When overloading ( ), [ ], -> or any of the assignment operators, must use a member function
- Operator functions as member functions
  - Leftmost operand must be an object (or reference to an object) of the class
  - If left operand of a different type, operator function must be a non-member function
- Operator functions as non-member functions
  - Must be friends if needs to access private or protected members
  - Enable the operator to be commutative

Copyright © 2017 Pearson Education, Ltd.

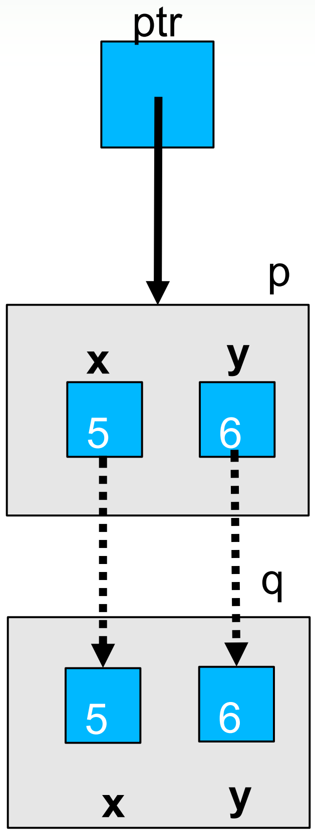
The = and & operators

- Operator = and operator & are overloaded implicitly for every class, so they can be used for each class objects
- operator = performs member wise copy of the data members.
- operator & returns the address of the object in memory.

```
class Point {
    int x;
    int y;

public:
    Point();
    Point (int x, int y);
    void setX(int x);
    void setY(int y);
    int getX()const;
    int getY()const;
};
```

```
Point::Point():x(0),y(0){ }

Point::Point(int x, int y){
    this->x = x;
    this->y = y;
}
void Point::setX(int x){ this->x = x; }
void Point::setY(int y){ this->y = y; }
int Point::getX()const{ return x;      }
int Point::getY()const{ return y;      }
```

```
int main() {
    Point p(5,6);
    Point q;
    q = p;
    Pointer *ptr;
    ptr = &p;
}
```



Copyright © 2017 Pearson Education, Ltd.

Overloading the = operator [1]

- ❑ Operator can be invoked as a member function:

  obj.operator=(obj2);

- ❑ It can also be used in more conventional manner:

  obj = obj2;

- ❑ Operator is called via object on left side
- ❑ The **this** pointer always points to the object that is being used to call the member function.

```
class Triangle {
private:
   Point *corners;

 public:
   Triangle();
   Triangle(const Point p[]);
   Triangle(const Triangle&);
   void setPoints(const Point p[]);
   void draw();
   Triangle& operator=(const Triangle&);
   ~Triangle();
};
```

```
Triangle::Triangle() {
    corners = new Point[3];
}

Triangle::Triangle(const Point a[]) {
    corners = new Point[3];
    for (int i=0;i<3;i++) {
      corners[i].setX(a[i].getX());
      corners[i].setY(a[i].getY());
    }
}

Triangle::Triangle(const Triangle& t) {
    corners = new Point[3];
    for (int i=0;i<3;i++){
      corners[i].setX(t.corners[i].getX());
      corners[i].setY(t.corners[i].getY());
    }
}

Triangle& Triangle::operator=(const
Triangle& t) {
    for (int i=0;i<3;i++) {
      corners[i].setX(t.corners[i].getX());
      corners[i].setY(t.corners[i].getY());
    }
    return *this;
}
```

```
Triangle::~Triangle() {
    delete[ ] corners;
}

void Triangle::setPoints(const
Point p[ ]) {
for (int i=0;i<3;i++)  {
  corners[i].setX(p[i].getX());
  corners[i].setY(p[i].getY());
 }
}
void Triangle::draw() {
 cout << "Triangle:";
 for (int i=0;i<3;i++) {
     cout << "("
            << corners[i].getX()
            <<","
            << corners[i].getY()
            << ")";
  }
  cout << endl;
}
```

Triangle t1 = t2;

Will invoke the copy constructor not the operator=

Copyright © 2017 Pearson Education, Ltd.

Overloading the = operator [2]

```
int main(){
    Point a[] = {Point(5,7), Point(1,2), Point(3,4)};
    Triangle t;
    t.setPoints(a);
    t.draw();
    Triangle t2,t3;
    t3 = t2 = t;
    t.draw();
    t2.draw();
    t3.draw();
    t3.operator=(t2.operator=(t));      // t3=t2=t;
    return 0;
}
```

chaining

```
void Triangle::construct(){
      corners = new Point[3];
}

Triangle& Triangle::operator=(const Triangle& t){
   construct();
   for (int i=0;i<3;i++){
     corners[i].setX(t.corners[i].getX());
     corners[i].setY(t.corners[i].getY());
   }
   return *this;
}
```

```
Triangle:(5,7)(1,2)(3,4)
Triangle:(5,7)(1,2)(3,4)
Triangle:(5,7)(1,2)(3,4)
Triangle:(5,7)(1,2)(3,4)
```

❑ The this pointer always points to the object that is being used to call the member function.

Copyright © 2017 Pearson Education, Ltd.

Overloading the + operator as a member function [1]

```
class Point {
    int x;
    int y;

public:
    Point();
    Point (int x, int y);
    void setX(int x);
    void setY(int y);
    int getX()const;
    int getY()const;
   void display()
    Point operator+(const Point &p);
};
```

```
Point Point::operator+ (const Point &p)
{
     return Point(x+p.x,y+p.y);
}
```

```
Point::Point():x(0),y(0){ }
Point::Point(int x, int y){
    this->x = x;
    this->y = y;
}
void Point::setX(int x){
    this->x = x;
}
void Point::setY(int y){
    this->y = y;
}
int Point::getX()const{
    return x;
}
int Point::getY()const{
    return y;
}
void Point::display(){
    cout << "(" << x  << ",”
    << y << ")"<<endl;
}
```

```
int main( ) {
    Point p1(10,20);
    Point p2(1,1);

    Point p3;
    p3 = p1 + p2;

    p1.display();
    p2.display();
    p3.display();

    return 0;
}
```

```
(10,20)
(1,1)
(11,21)
```

Point p3;
P3 = p1 + p2;

↔

Point p3;
P3 = p1.operator+(p2);

Copyright © 2017 Pearson Education, Ltd.

Overloading the + operator as a member function [2]

```
Point Point::operator+ (const Point &p)
{
   Point t;
   t.x = x + p.x;
   t.y = y + p.y;
   return t;
}
```

```
Point Point::operator+ (const Point &p)
{
   return Point(x+p.x,y+p.y);
}
```

Copyright © 2017 Pearson Education, Ltd.

Overloading the + operator as a member function [3]

```
class Point {
    int x;
    int y;

public:
    Point();
    Point (int x, int y);
    void setX(int x);
    void setY(int y);
    int  getX()const;
    int  getY()const;
    void display();
    Point& operator+(int x);
};
```

```
Point Point::operator+ (int a)
{
    x = x + a;
    y = y + a;
    return *this;
}
```

```
int main( )
{
        Point p1(10,20);
        Point p2(1,1);

        Point p3(3,5);
        p3 = p3 + 6;

        p1.display();
        p2.display();
        p3.display();

        return 0;
}
```

```
(10,20)
(1,1)
(9,11)
```

Copyright © 2017 Pearson Education, Ltd.

Overloading the + operator as a friend function

```
class Point {
    int x;
    int y;

public:
    Point();
    Point (int x, int y);
    void setX(int x);
    void setY(int y);
    int getX()const;
    int getY()const;
    friend Point  operator+ (const Point &p, const Point &q);
};
```

```
int main( )
{
    Point p1(10,20);
    Point p2(1,1);

    Point p3;
    p3  = p1 + p2;

    p1.display();
    p2.display();
    p3.display();

    return 0;
}
```

```
Point operator+ (const Point &p, const Point &q)
{
    Point z;
    z.x =p.x + q.x;
    z.y =p.y + q.y;
    return Point(z.x, z.y);
}
```

```
(10,20)
(1,1)
(11,21)
```

Copyright © 2017 Pearson Education, Ltd.

Overloading the << and >> operators

- Overloaded stream operators >>, << must return reference to istream, ostream objects and take istream, ostream objects as parameters

```
class Point {
 int x;
 int y;
public:
 Point();
 Point (int x, int y);
 void setX(int x);
 void setY(int y);
 int getX()const;
 int getY()const;
 friend ostream& operator<<(ostream &out, const Point& p);
 friend istream& operator>>(istream &in, Point& p);
};
```

```
Point::Point():x(0),y(0){ }


Point::Point(int x, int y){
    this->x = x;
    this->y = y;
}
void Point::setX(int x){ this->x = x; }
void Point::setY(int y){ this->y = y; }
int Point::getX()const{ return x;      }
int Point::getY()const{ return y;      }
```

```
Enter x and y coord: 1 3
(10,20)(2,2)(1,3)
```

```
ostream& operator<<(ostream &out, const Point& p)
{
    out << "(" << p.x << "," << p.y << ")";
    return out;
}

istream& operator>>(istream &in, Point& p){
    cout << "Enter x and y coord: ";
    in >> p.x >> p.y;
    return in;
}
```

```
int main(){
    Point p(10,20);
    Point q(2,2);
    Point t;
    cin >> t;
    cout << p ;
    cout << q ;
    cout << t;
    return 0;
}
```

Copyright © 2017 Pearson Education, Ltd.

More operator Overloading

```
class Complex{
public:
    Complex();
    Complex( double );
    Complex( double, double );
    void print() const;
    Complex operator+( const Complex& ) const;
    Complex operator-( const Complex& ) const;
    Complex operator*( const Complex& ) const;
    Complex operator/( const Complex& ) const;
    bool operator==( const Complex& ) const;
    bool operator!=( const Complex& ) const;
private:
    double real;
    double imag;
};
```

```
Complex::Complex() { real = imag = 0.0; }
Complex::Complex( double re ) {
    real = re;       imag = 0.0;
}
Complex::Complex( double re, double im ) {
    real = re;      imag = im;
}
void Complex::print() const {
    cout << real << " + " << imag << "i\n";
}
```

Copyright © 2017 Pearson Education, Ltd.

```
bool Complex::operator==( const Complex& u ) const
{
     return  (real == u.real && imag == u.imag) ;
}

bool Complex::operator!=( const Complex& u ) const
{
    return  !(real == u.real && imag == u.imag) ;
}
```

```
Complex Complex::operator-( const Complex& u )
const {
     return Complex ( real - u.real, imag - u.imag );
}
Complex Complex::operator*( const Complex& u )
const {
    return Complex v( real * u.real - imag * u.imag,
            imag * u.real + real * u.imag );
}
Complex Complex::operator/( const Complex& u )
const {
    double abs_sq = real * u.real + imag * u.imag;
    return Complex ( ( real * u.real + imag * u.imag )
/abs_sq, ( imag * u.real - real * u.imag ) / abs_sq );
}
Complex Complex::operator+( const Complex& u )
const {
     return Complex ( real + u.real, imag + u.imag );
}
```

More operator overloading

```
int main() {
  Complex c1( 8.8, 0 );
  Complex c2( 3.1, -4.3 );
  Complex c3 = c1 + c2;
  Complex c4 = c2 - c1;
  Complex c5 = c4 / c1;
  Complex c6 = c4 * c1;
  c1.print();
  c2.print();
  c3.print();
  c4.print();
  c5.print();
  c6.print();

  if ( c3 == c4 ) cout << "equal";
  else  cout << "not equal";
}
```

```
8.8 + 0i
3.1 + -4.3i
11.9 + -4.3i
-5.7 + -4.3i
1 + 0.754386i
-50.16 + -37.84i
not equal
```

Copyright © 2017 Pearson Education, Ltd.

Overloading the [ ] operator

- ❑ Can create classes that behave like arrays, provide bounds-checking on subscripts
- ❑ Must consider constructor, destructor
- ❑ Overloaded [] returns a reference to object, not an object itself

```
class Array
{
private:
    int *ptr;
    int size;
public:
    Array(int *, int);
    int &operator [ ] (int);
    void print() const;
};
```

Copyright © 2017 Pearson Education, Ltd

```
int &Array::operator[ ] (int index)
{
    if (index >= size) {
        cout << "out of bound";
        exit(0);
    }
    return ptr[index];
}

Array::Array(int *p = NULL, int s = 0)
{
    size = s;
    ptr = NULL;
    if (s != 0) {
        ptr = new int[s];
        for (int i = 0; i < s; i++)
            ptr[i] = p[i];
    }
}

void Array::print() const
{
    for(int i = 0; i < size; i++)
        cout<<ptr[i]<<" ";
    cout<<endl;
}
```

```
int main()
{
    int a [ ] = {1, 2, 4, 5};
    Array arr1(a, 4);
    arr1[2] = 6;
    arr1.print();
    arr1[8] = 6;
    return 0;
}
```

```
1 2 6 5
out of bound
```

Overloading the ++ and -- operators

- Pre/post incrementing/decrementing operators
- Allowed to be overloaded
- Distinguishing between pre and post operators
  - prefix versions are overloaded the same as other prefix unary operators

    d1.operator++();

  - convention adopted that when compiler sees post incrementing expression, it will generate the member-function call

    d1.operator++(**0**);

- **0** is a dummy value to make the argument list of operator++ distinguishable from the argument list for ++operator

```
class Point {
    int x;
    int y;
public:
    Point();
    Point (int x, int y);
    Point (const Point &);
    void setX(int x);
    void setY(int y);
    int getX()const;
    int getY()const;
    Point operator++();
    Point operator++(int);
    void display();
};
```

```
//prefix++      ++x
Point Point::operator++() {
    x++;
    y++;
    return *this;
}

//postfix++     x++
Point Point::operator++(int) {
    Point p(*this);
    operator++();
    return p;
}
```

```
int main() {
    Point p(10,20);
    p++;
    p.display();
    ++p;
    p.display();

    Point p2 = p++;
    p2.display();
    p.display();
```

```
Point p3 = ++p;
    p3.display();
    p.display();

    return 0;
}
```

```
(11,21)
(12,22)
(12,22)
(13,23)
(14,24)
(14,24)
```

Copyright © 2017 Pearson Education, Ltd.

Overloading the ! operator

- ❑ If we use a class member function to overload a binary operator, the member function has only one parameter.
- ❑ Similarly, if we use a class member function to overload a unary operator, the member function has no parameters.

```
class Point {
    int x;
    int y;
public:
    Point();
    Point (int x, int y);
    void setX(int x);
    void setY(int y);
    int getX()const;
    int getY()const;
    Point operator ! ( );
    ~Point();
};
```

```
Point Point::operator! () {
    Point tmp( -x, -y );
    return tmp;
}
```

```
int main() {
    Point a[] = {Point (5,7), Point(1,2), Point(3,4)};
    for (int i=0;i<3;i++)
        a[i] = !a[i];

    for (int i=0;i<3;i++)
        cout << a[i].getX() <<"," << a[i].getY() << endl;

    return 0;
}
```

```
-5,-7
-1,-2
-3,-4
```

Copyright © 2017 Pearson Education, Ltd.

Operator Functions

- Using class member functions, the overloaded operator is invoked as a member function on an object.

```
a = b + c;
a = b.operator+( c );
```

- Using stand-alone functions, the overloaded operator is invoked as a function that treats the two operands equally.

```
a = operator+( b , c );
```

- An operator intended to accept a basic type as its first operand can only be overloaded as stand alone function.

Copyright © 2017 Pearson Education, Ltd.

rvalue References and Move operations

Copyright © 2017 Pearson Education, Ltd.

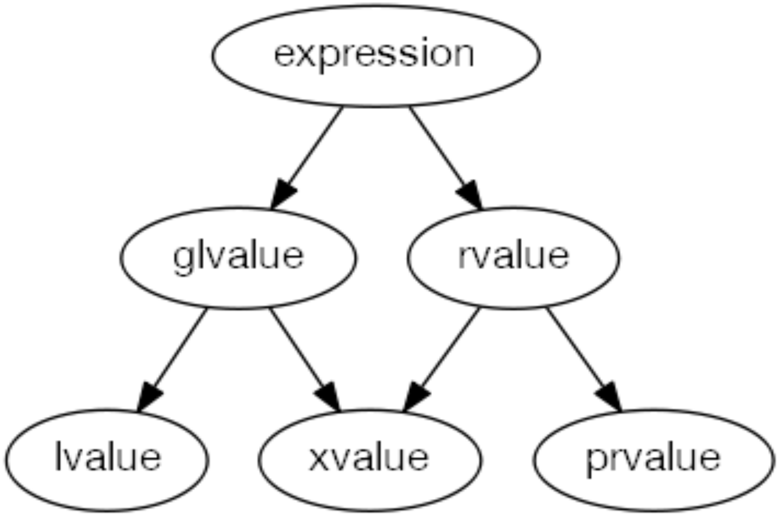
History

- In C++98 an expression is either an lvalue or rvalue
  - **lvalue**: occupies some identifiable location in memory **(has name, has an address, can return an address using the & operator, and can be changed)**
  - **rvalue**: not an **lvalue**
- Expressions have two properties
  - **Has an identity:** it is possible to determine whether the expression refers to the same entity as another expression, such as by comparing addresses of the objects or the functions they identify.
  - Can be moved from: Move constructor, move assignment operator, or another function overload that implements move semantics can bind to the expression

Copyright © 2017 Pearson Education, Ltd.

History

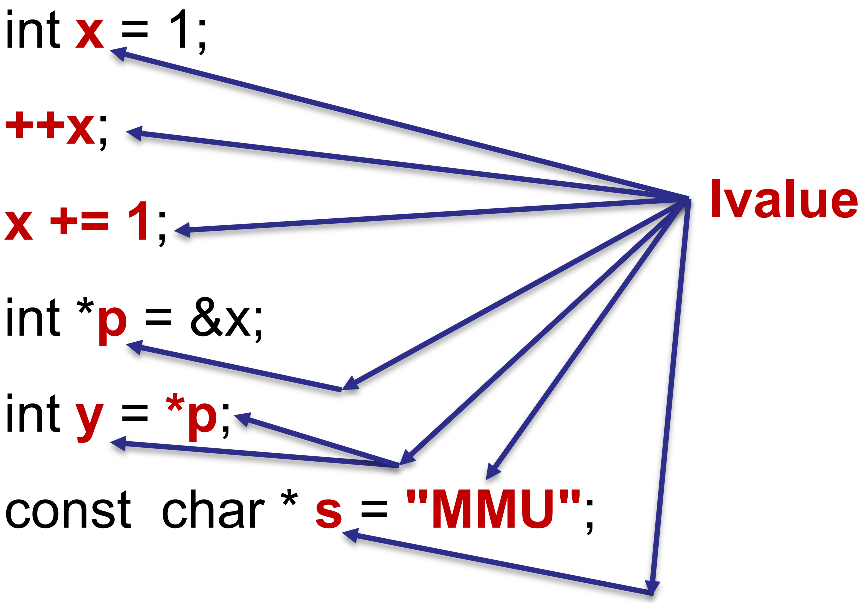
- In C++11 new value categories were added:
  - glvalue (generalized lvalue) : has identity
  - rvalue (right value) : can be moved from
  - lvalue (left value) : Has an identity and cannot be moved from
  - xvalue (expiring value) : Has an identity and can be moved from
  - prvalue (pure rvalue) : does not have identity and can be moved from



Copyright © 2017 Pearson Education, Ltd.

lvalue

- The name of a variable, a function, or a data member, regardless of type



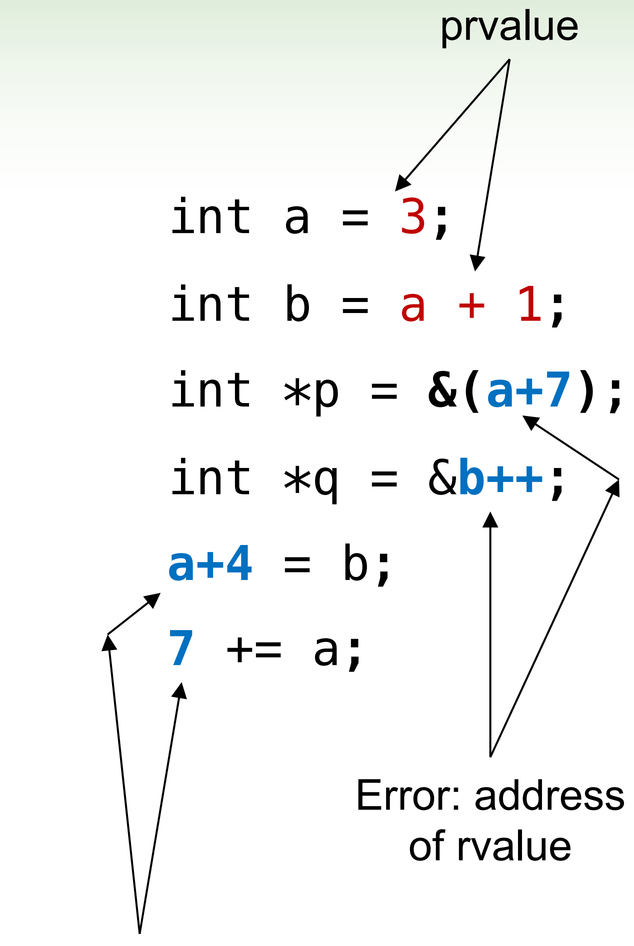
```
int x = 1;
++x;
x += 1;
int *p = &x;
int y = *p;
const  char * s = "MMU";
```

lvalue

Copyright © 2017 Pearson Education, Ltd.

prvalue

- □ prvalue is an rvalue that is **not an xvalue**
- □ A literal except string literal such as 10, true, nullptr
- □ A function call or an overloaded operator expression, whose return type is non-reference, such as str.substr(2,4), s1+s2.
- □ x++ and x- - built-in post-increment and post-decrement expressions
- □ x+y, x%y, x<<y and all built-in arithmetic expressions
- □ X && y, x||y, !x logical expressions
- □ X < y, x==y, x>=y comparison operators
- □ &x address expression



prvalue

```
int a = 3;
int b = a + 1;
int *p = &(a+7);
int *q = &b++;
a+4 = b;
7 += a;
```

Error: address of rvalue

rvalue cannot appear on the left side of an expression

Copyright © 2017 Pearson Education, Ltd.

xvalue

- ❑ A function call or an overloaded operator expression, whose return type is rvalue reference to object, such as **std::move(x)**
- ❑ **X[i]** where X is an array rvalue
- ❑ **X.m**, the member of object expression, where X is an rvalue and m is a non-static data member of non-reference type; *
- ❑ A cast expression to rvalue reference to object type, such as **ststic_cast<int&&>(x).**

Copyright © 2017 Pearson Education, Ltd.

lvalue

What is wrong with the following code ?

```
int foo( ) {
   return 2;
}

int main( ) {
    foo( ) = 2;
    return 0;
}
```

test.c: In function 'main': test.c:8:5: error: **lvalue** required as left operand of assignment

What is wrong with the following code ?

```
int& foo() {
    return 2;
}

int main( ) {
    int a = foo();
    return 0;
}
```

testcpp.cpp: In function 'int& foo()':
testcpp.cpp:5:12: error: invalid initialization of non-const reference of type 'int&' from an **rvalue** of type 'int'

Copyright © 2017 Pearson Education, Ltd.

lvalue and rvalue

❑ Can this code compile?

```
int globalvar = 20;

int& foo() {
    return globalvar;
}

int main() {
    foo() = 10;
    return 0;
}
```

Returns lvalue reference

A reference to the global variable

Lvalue:

○ Has name

○ Has address

○ Can return the address using &

Copyright © 2017 Pearson Education, Ltd.

examples

```
int x = 1;          // x is an lvalue

int y = 2;          // y is an lvalue

int z = x + y;      // + requires rvalues,
                    // x and y are converted to rvalues
                    // and an rvalue is returned
```

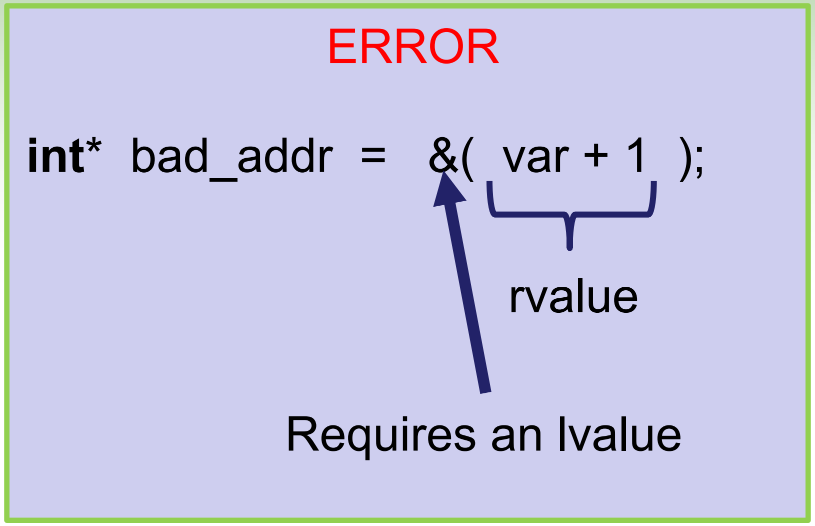
```
int arr [ ] = {1, 2, 3, 4};

int* p = &arr[0];

*(p + 1) = 10;
```

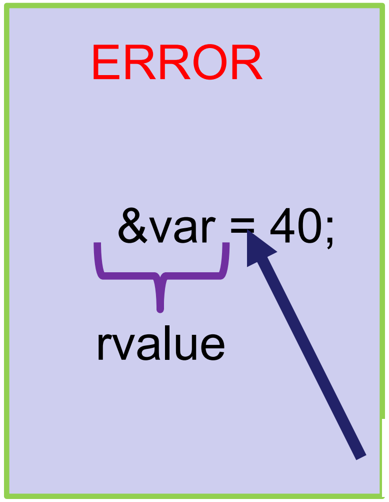
$$\overbrace{*\,(\underbrace{p+1}_{\text{rvalue}})}^{\text{lvalue}}$$

Copyright © 2017 Pearson Education, Ltd.

examples

```
int var = 10;
int* bad_addr = &(var + 1);
int* addr = &var;
&var = 40;
```





Requires an lvalue to the left of the assignment

Copyright © 2017 Pearson Education, Ltd.

Move assignment and Move constructor

Copyright © 2017 Pearson Education, Ltd.

Move Assignment and Move Constructor

- **Copy assignments** and **copy constructors** are used when objects contain **dynamic memory**.
- Deallocating memory in the target object, then allocating memory for the copy, then destroying the temporary object, is resource-intensive.
- Move assignment and move constructors, which use **rvalue** references, are much more efficient.
- Move assignment (overloaded = operator) and move constructor use an rvalue reference for the parameter
- The dynamic memory locations can be “**taken**” from the parameter and assigned to the members in the object invoking the assignment.
- Set dynamic fields in the parameter to **nullptr** before the function ends and the parameter’s destructor executes.
- Though introduced in C++ 11, move operations have already been used by the compiler:
  - when a non-void function returns a value
  - when the right side of an assignment statement is an rvalue
  - on object initialization from a temporary object

Copyright © 2017 Pearson Education, Ltd.

Default Class Operations

- Managing the details of a class implementation is tedious and potentially error-prone.
- The C++ compiler generates automatically five methods:
  - Default constructor,
  - Copy constructor,
  - Copy assignment operator,
  - Move constructor, and
  - Destructor.

Rule of five

If you provide your own implementation of any of these functions, you should provide your own implementation for all of them.

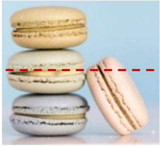
Copyright © 2017 Pearson Education, Ltd.

Move Constructor and Move Assignment

```
class myArray {
    string name;
    int size;
    int *data;

public:
    myArray();
    myArray(string name, int size);
    myArray(string name,int arr[], int size);
    myArray(const myArray& ot);
    myArray& operator=(const myArray& ot);
    myArray(myArray && ot);
    myArray& operator=(myArray&& ot);
    ~myArray();
    void speak();
    void init(string,int);
};
```

```
myArray::~myArray()
{
    cout << name << "-> destructor\n";
    delete[] data;
}
```

```
void myArray::init(string name, int size){
    this->name = name;
    this->size = size;
    data = new int[this->size]{};
}

myArray::myArray() {
    init("MMU",5);
    cout << name << " -> def. const\n";
}

myArray::myArray(string name, int size) {
    init(name,size);
    cout << name << " -> Param1. const\n";
}

myArray::myArray(string name,int arr[], int size)
{
    cout << name << " -> Param2. const\n";
    init(name,size);
    for (int i=0;i<size;i++)
        data[i]=arr[i];
}
```



Copyright © 2017 Pearson Education, Ltd.

Move Constructor and Move Assignment

```
myArray::myArray(const myArray& ot)
{
   cout << name << " -> Copy const .. using "
         << ot.name << endl;
   name = ot.name;
   size = ot.size;
   init(name,size);
   for (int i=0;i<size;i++)
      data[i]=ot.data[i];
}
myArray& myArray::operator=(const myArray& ot)
{
   cout << name <<" Operator= "
         << ot.name << endl;
   init(ot.name,ot.size);
   for (int i=0;i<size;i++)
     data[i]=ot.data[i];
   return *this;
}
```

```
void myArray::speak ( ) {
   cout <<"->";
   for (int i=0;i<size;i++) cout << data[i] << "_" << endl;
}
```

```
myArray::myArray(myArray&& ot)
{
   cout << name << " -> Move const ..using "
         << ot.name << endl;
   name = ot.name;
   size = ot.size;
   data = ot.data;
   ot.data = nullptr;
   size = 0;
  }
myArray& myArray::operator=(myArray&& ot)
{
   cout << name
         << " -> Move operator= using “
         <<ot.name << endl;
   if (this != &ot)
   {
      size = ot.size;
      data = ot.data;
      ot.data = nullptr;
   }
   return *this;
}
```

Copyright © 2017 Pearson Education, Ltd.

Move Constructor and Move Assignment

```cpp
int main(){
    int arr[]={1,3,4,5,7};
    cout << "-----1-------\n";
    myArray a;
    a.speak();
    cout << "-----2-------\n";
    myArray b("b",arr,5);
    b.speak();
    cout << "----2.5------\n";
    myArray c("c",5);
    cout << "-----3-------\n";
    myArray d(b);
    cout << "-----4------\n";
    myArray e(move(myArray("tempo1",5)));
    e.speak();
    cout << "-----5------\n";
    myArray f;
    cout << "-----6------\n";
    f = b;
    cout << "-----7------\n";
    myArray g;
    cout << "-----8------\n";
    g = myArray("tempo2",20);
    cout << "-----9------\n";

  return 0;
}
```

```
-----1------
MMU -> def. const
->0_0_0_0_0_
-----2------
b -> Param2. const
->1_3_4_5_7_
-----2.5----
c -> Param1. const
-----3------
b -> Copy const .. using b
-----4------
tempo1 -> Param1. const
 -> Move const ..using tempo1
tempo1-> destructor
->
-----5------
MMU -> def. const
-----6------
MMU Operator= b
-----7------
MMU -> def. const
-----8------
tempo2 -> Param1. const
MMU -> Move operator= using
tempo2
tempo2-> destructor
-----9------
MMU-> destructor
b-> destructor
tempo1-> destructor
b-> destructor
c-> destructor
b-> destructor
MMU-> destructor
```

Copyright © 2017 Pearson Education, Ltd.

Member Initialization Lists

Copyright © 2017 Pearson Education, Ltd.

Member Initialization List

- Used in constructors for classes involved in aggregation.
- Allows constructor for enclosing class to pass arguments to the constructor of the enclosed class
- Member Initialization lists can be used to simplify the coding of constructors. The compiler may also generate more efficient code.
- You should keep the entries in the initialization list in the same order as they are declared in the class

```
class StudentInfo
{
           ...
};

class Student
{
 private:
   StudentInfo   personalData;

 public:
   Student(string fname, lname): personalData(fname, lname){};
};
```

Copyright © 2017 Pearson Education, Ltd.

Member Initialization List

```
class myArray {
    string name;
    int size;
    int *data;

public:
    myArray();
    myArray(string name, int size);
    myArray(string name,int arr[], int size);
    :
    :
    ~myArray();
    void speak();
};
```

```
myArray::myArray()
: name("X"),size(5), data(new int[5])
{ }
myArray::myArray(string name, int size)
: name(name),size(size), data(new int[5])
{ }

myArray::myArray(string name,int arr[], int size)
: name(name),size(size), data(new int[5])
{
    for (int i=0;i<size;i++)
        data[i]=arr[i];
}
```

Copyright © 2017 Pearson Education, Ltd.