# Week 2

# Chapter 3: Pointers and Reference Variables

Outline:

- Pointers
- Pointer Arithmetic
- Memory Allocation
- Memory Leak
- Member Access Operators . and ->
- Reference Variables

# Pointer Variables

- Pointer variables are variables that hold memory addresses of variables.
- In conjunction with pointer variables are two unary operators & and .
- The unary operator &, address of, gives the address of its operand.
- The unary operator ., dereferencing, gives the contents pointed at by its operand.

# Illustrating Pointer Variables

```
$ cat ptrvar.cpp
#include <iostream>
using namespace std;
int main() {
    int *p;
    int x = 12345;
    //  int z;      // uncomment it will change y's location
    int y;
    p = &x; cout << &p << '\t' << p << '\t' << *p << endl;
    p = &y; cout << &p << '\t' << p << '\t' << *p << endl;
    y = 67890; cout << &p << '\t' << p << '\t' << *p << endl;
}
```

```
$ g++ ptrvar.cpp
$ a.out
0xffbff89c 0xffbff898 12345
0xffbff89c 0xffbff894 -4196076
0xffbff89c 0xffbff894 67890
```

# The meaning of int* x, y;

- The declaration int a,b; is the same as int a; int b;
- Consider the declaration

  int* x, y;
- It is the same as

  int *x; int y;
- It is clearer to place * just before the variable as

  int *x,y;

  so as to remind us that int is applicable to *x and y.

# Pointer Arithmetic

- An integer n can be added or subtracted from a pointer variable p pointing at variables of type *theType*.
- The result of p + n is

  p ← p + n × sizeof(*theType*)

# Pointer Arithmetic Example

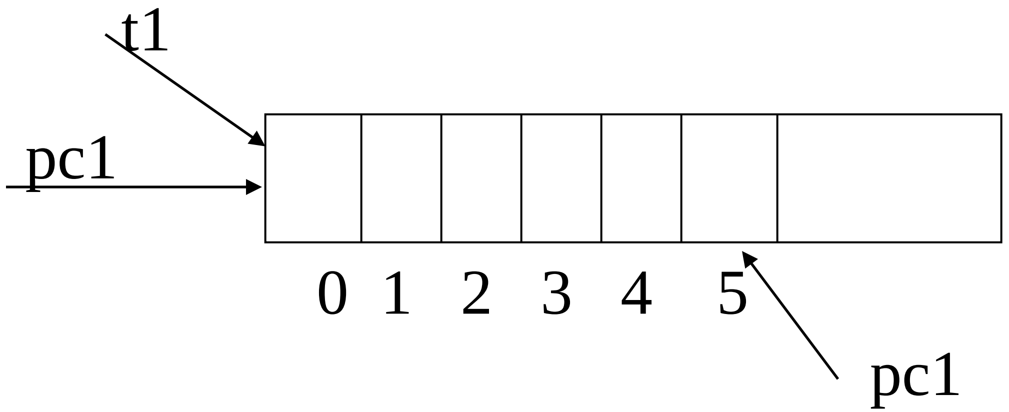
```
$ cat parith.cpp
#include <iostream>
using namespace std;
int main() {
      int *p, sum = 0;
      int a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
      for( p = &a[0]; p <= &a[9]; p++ ) sum += *p;
      // or  for( p = a; p <= a + 9; p++ ) sum += *p;
      cout << sum << endl;
}
```

```
$ g++ parith.cpp
$ a.out
45
```

# Arrays and pointers – (1)

array name is a constant pointer that points to the zeroth array element

```
char t1 [10];
char *pc1 = t1;          //ok
char t2 [10];
t1 = t2;                 // error
pc1 = t1 + 5             // pc1 points to t1 [5]
*t1  = *(t1 + 5)         // assigns value of t1[5] to t1[0]
```



# Arrays and pointers – (2)

```
#include <iostream>
using namespace std;
int main() {
  char t1 [] = "How are you?";
  *t1  = *(t1 + 5);      // same as  t1[0] = t1[5]
  cout << "The character at t1[0] is " << t1[0] << endl; // r
  cout << "The modified string is: " << t1 << endl;
                              // row are you?
  return 0;
}
```

```
$ g++ array1.cpp
$ a.out
The character at t1[0] is r
The modified string is: row are you?
```

# Arrays and pointers – (3)

- If the index is out of range when we access an array,
  - we will get garbage if we read from the entry, or
  - we will encounter a runtime error some time later if we write to the entry

Example:

```
char temp = 'a';
int i = 7;
char *c = &temp;
c++;                // Instead of using *c++
                    // c points to somewhere unintended
```

# Array Access: index or pointer?

```
#include <iostream>
using namespace std;
int main() {
  int a[] = {1,2,3,4,5,6,7,8,9};
  int i;
  for (i = 0; i < 9; ++i) {
    cout << a[i];            // access through index []
  }
  cout << endl;
  int *p;
  for (p = a; p != a+9; ++p) { // a+9 : beyond the array
    cout << *p;              // access through pointer
  }
  cout << endl;
  return 0;
}
```

```
$ g++ array2.cpp
$ a.out
123456789
123456789
```

# Memory Allocation

- A variable may reside in the data segment, on the stack, or on the heap.
- Variables in the data segment are global variables.
- Variables in the stack are automatic variables. They are local to the function when it is active. They are deallocated when the function returns.
- Variables on the heap are allocated by the new operator. They persist until deallocated by the delete operator.

# Memory Allocation Example (1)

```
$ cat memory.cpp
#include <iostream>
using namespace std;
int g[] = { 1, 2, 3 };          // a global int array
void show(int *v, int n) {
    for(int i=0; i<n; ++i) cout << ' ' << v[i];
    cout << endl;
}
```

# Memory Allocation Example (2)

```
int * h() { // allocating a 3-int array on the heap
    int *p = new int[3];
    p[0] = 4; p[1] = 5; p[2] = 6;
    return p;
}
int * s() { // allocating a 3-int array on the stack
    int q[] = { 7, 8, 9 };
    return q;
}
void c() {// allocating a 3-int array on the stack
    int n[] = { -9, -8, -7 };
}
```

# Memory Allocation Example (3)

```
int main() {
    int *pg, *ph, *ps;
    pg = g; show(pg,3);
    ph = h(); show(ph,3);
    ps = s(); show(ps,3);
    c();
    show(pg,3);
    show(ph,3);
    show(ps,3);
}
```

```
$ g++ memory.cpp
$ a.out
1 2 3
4 5 6
7 8 9
1 2 3
4 5 6
-9 -8 -7
```

# Memory Allocation Example (4)

- Note that the memory areas pointed at by the pointer variables pg and ph are intact in between function calls.
- But the memory area pointed at by the pointer variable ps is overwritten by the local variables of functions that have been called.

# Memory Leak

- Memory leak refers to memory areas allocated by the new operator that are not deallocated when they are no longer referenced.
- Such unreferenced memory is lost and is unavailable for further use.

# Memory Leak Example (1)

```
$ cat leak.cpp
#include <iostream>
using namespace std;
int main() {
    int *p;
    for(int i = 0; 1; i++) {
        p = new int[100000000];
        cout << i << endl;
    }
}
```

```
$ g++ leak.cpp
$ a.out
0
1
2
3
4
5
6
7
8
9
terminate called after
    throwing an instance of
    'std::bad_alloc'
```

# Memory Leak Example (2)

```
$ cat leak2.cpp
#include <iostream>
using namespace std;
int main() {
    int *p;
    for(int i = 0; 1; i++) {
        p = new int[100000000];
        delete p;
        p = 0;
        cout << i << endl;
    }
}
```

```
$ g++ leak2.cpp
$ a.out
..... # many lines are not shown
22911
22912
22913
22914
22915
22916
22917
22918
22919
^C # interrupt it!
```

# Stale Pointers

- Stale pointers are pointers that point at memory areas that have been deallocated.
- Using a stale pointer leads to unexpected or invalid contents.

# Stale Pointers Example (1)

```
$ cat stale.cpp
#include <iostream>
using namespace std;
int main() {
     string *s = new string( “banana" );
     cout << s <<’ ’<< *s << endl;
     delete s;
     s = new string( “papaya" );
     cout << s <<’ ’<< *s << endl;
     delete s;
     new string( “durian" );
     cout << s <<’ ’<< *s << endl;
}
```

# Stale Pointers Example (2)

- The contents of the memory area pointed at by s is not the expected contents.

$ g++ stale.cpp

$ a.out

0x21668 banana

0x21668 papaya

0x21668 durian

- Note that the value of s is not changed by delete.

# Reference Variables

- A reference variable is an alias of a variable. That is, the same memory area has more than one name.
- In particular, a reference formal parameter allows parameter passing by reference.
- A reference variable which is not a reference formal parameter must be initialized at declaration time.

# Reference Variables Example (1)

```
$ cat refvar.cpp
#include <iostream>
using namespace std;
void f( int &x ) { x++; }
void g( int x ) { x++; }
void h( int *x ) { (*x)++; }
```

# Reference Variables Example (2)

```
int main() {
    int x = 0, y = 0;
    int &z = x;   // z, x are regarded as the same var
    f(x); cout << z << endl;
    g(x); cout << z << endl;
    h(&x); cout << z << endl;
    z = y;        // z (hence x) is reset with y's value
    f(z); cout << x << endl;
    g(z); cout << x << endl;
    h(&z); cout << x << endl;
    cout << y << endl;        // y not affected
}
```

```
$ g++ refvar.cpp
$ a.out
1
1
2
1
1
2
0
```

# What are Pointer Variables For

- To achieve call-by-reference
- An array can be iterated by a pointer
- Use pointers to refer to a large memory area to avoid copying the data
- Pointers are needed to implement linked data structures
- Pointers are useful for inheritance and polymorphism

# Chapter 4: Classes

Outline:

- C++ class syntax
- Accessors versus Mutators
- Keyword explicit
- Friend
- Return by Constant-Reference
- Separation of Interface and Implementation
- Static Members

# C++ Classes

- C++ supports classes.
- C++ classes declaration syntax differs from that of java classes slightly:
  - C++ classes declarations must be terminated by semi-colons.
  - C++ classes are declared without accessibility specifiers (public, private, protected).
  - Accessibility of the members (data or operations) of a C++ class can be specified collectively.
  - A C++ class operation is called a member function

# Class Example (1)

```
$ cat intcell.cpp
#include <iostream>
using namespace std;
class IntCell {
public:
    IntCell(int initialValue = 1) {          // Constructor
        storeValue = initialValue;
        cout << "construct IntCell(" << initialValue << ")\n";
    }
    int getValue() { return storeValue; }
    void setValue(int val) { storeValue = val; }
private:
    int storeValue;
};
```

# Class Example (2)

```
int main() {
    IntCell m1;            // construct object
    IntCell m2 = 2;        // construct object
    IntCell m3(3);         // construct object
    cout << m1.getValue() << ' '        // access member function
         << m2.getValue() << ' '
         << m3.getValue() << endl;
    m1 = m2;               // Copy an object
    m2.setValue(20);
    cout << m1.getValue() << ' '
         << m2.getValue() << ' '
         << m3.getValue() << endl;
}
```

```
$ g++ intcell.cpp
$ a.out
construct IntCell(1)
construct IntCell(2)
construct IntCell(3)
1 2 3
2 20 3
```

# Member Access Operators . and ->

- If c1 is an object of class C, i.e.,

  C c1;

  then to access a field y in c1 or a member function f() through c1, use the member access operator .:

  c1.y;

  c1.f();
- Let x be a pointer to c1, an object of the class C. That is,

  C *x = &c1;
- To access the field y or function f() through x, the following expressions are equivalent:

  (*x).y        or        x->y

  (*x).f()        or        x->f()

# Accessors versus Mutators

- A member function that does not change any member data (except those qualified as mutable) is an accessor.
- A member function that might change some member data is a mutator.
- When the state of an object should not be changed, only accessors of that object can be called.
- The signature of a member function includes its accessor/mutator status.

# Accessors versus Mutators Example (1)

```
$ cat const.cpp
#include <vector>
using namespace std;
class C {
      public: int get() {return v;}
      private: int v;
};
int countZero(const vector<C> & cs) { // cs immutable
      int count=0;
      for(int i=0; i<cs.size(); i++) if( cs[i].get() == 0 ) count++;
      return count;
}
```

# Accessors versus Mutators Example (2)

$ g++ -c const.cpp

const.cpp: In function ‘int countZero(const std::vector<C, std::allocator<C> >&)’:

const.cpp:13: error: passing ‘const C’ as ‘this’ argument of ‘int C::get()’ discards qualifiers

# Accessors versus Mutators Example (3)

```
$ cat constfix.cpp
#include <vector>
using namespace std;
class C {
     public: int get() const                    // It is an accessor
               {return v;}
     private: int v;
};
int countZero(const vector<C> & cs) {   // cs immutable
     int count=0;
     for(int i=0; i<cs.size(); i++) if( cs[i].get() == 0 ) count++;
     return count;
}
```

```
$ g++ -c constfix.cpp
```

# Keyword explicit

- If a class C can be instantiated using a constructor with one actual parameter, then the defintion

  C x(p);

  is equivalent to

  C x = p;

- To forbid this equivalence (i.e., to stop the automatic type conversion from p of certain type to an object x of class C) in the statement C x = p;, the relevant constructor should be marked as explicit, i.e., an object of C has to be created explicitly using the statement C x(p);

# Implicit Conversion Example (1)

```
$ cat implicit.cpp
#include <iostream>
using namespace std;
class C {
public:
    C(int a=0, int b=1) { cout << "non-copy constructor\n"; }
    C(const C & c) { cout << "copy constructor\n"; }
    void operator = (const C & c) {
        cout << "copy assignment\n";
    }
};
```

# Implicit Conversion Example (2)

```
int main() {
    C a(2), b = 3;
    C c(b), d = a;
    a = 4;
    a = C(5);
    a = b;
}
```

```
$ g++ implicit.cpp
$ a.out
non-copy constructor
non-copy constructor
copy constructor
copy constructor
non-copy constructor
copy assignment
non-copy constructor
copy assignment
copy assignment
```

# Keyword explicit Example (1)

```
$ cat implicit1.cpp
#include <iostream>
using namespace std;
class A {
      public:
      A(int a) {
      cout << "A(" << a << ")\n"; }
};
class B {
      public:
      B(int b1, int b2=2) {
            cout << "B(" << b1 << ',' << b2 << ")\n";
      }
};
```

# Keyword explicit Example (2)

```
int main() {
    A p(1), q = 2;
    B r(3), s = 4, t(5,6);
}
```

```
$ g++ implicit1.cpp
$ a.out
A(1)
A(2)
B(3,2)
B(4,2)
B(5,6)
```

# Keyword explicit Example (3)

```
$ cat explicit.cpp
#include <iostream>
using namespace std;
class A {
    public:
    explicit A(int a) {
        cout << "A(" << a << ")\n";
    }
};
class B {
    public:
    explicit B(int b1, int b2=2) {
        cout << "B(" << b1 << "," << b2 << ")\n";
    }
};
```

# Keyword explicit Example (4)

```
int main() {
      A p(1),
      q = 2;
      B r(3),
      s = 4,
      t(5,6);
}
```

```
$ g++ explicit.cpp
explicit.cpp: In function ‘int main()’:
explicit.cpp:20: error: conversion from ‘int’ to non-scalar type ‘A’ requested
explicit.cpp:22: error: conversion from ‘int’ to non-scalar type ‘B’ requested
```

# Initializer Lists

- Data members can be initialized with an initializer list, instead of assignments, in a constructor.
- Initialization with an initializer list is more efficient. Briefly, with assignment a data member has to be constructed then copied; but with initalizer it is constructed directly.

# Initializer Lists Example (1)

```
$ cat init.cpp
#include <iostream>
using namespace std;
class A {
public: A(int i=1, float f=2) : count(i), value(f) {} // All init done in IL
     int count; float value;
};
class B {
public: B(int i=3, float f=4) { count = i; value = f; } // No IL
     int count; float value;
};
class C {
public: C(int i=5, float f=6) : count(i) { value = f; } // Some init in IL
     int count; float value;
};
```

# Initializer Lists Example (2)

```
int main() {
    A a1, a2(-1), a3(-2,-3);
    B b1, b2(-4), b3(-5,-6);
    C c1, c2(-7), c3(-8,-9);
    cout << a1.count << '\t' << a1.value << endl;
    cout << a2.count << '\t' << a2.value << endl;
    cout << a3.count << '\t' << a3.value << endl;
    cout << b1.count << '\t' << b1.value << endl;
    cout << b2.count << '\t' << b2.value << endl;
    cout << b3.count << '\t' << b3.value << endl;
    cout << c1.count << '\t' << c1.value << endl;
    cout << c2.count << '\t' << c2.value << endl;
    cout << c3.count << '\t' << c3.value << endl;
}
```

```
$ a.out
1 2
-1 2
-2 -3
3 4
-4 4
-5 -6
5 6
-7 6
-8 -9
```

# The Big Three

- C++ classes come with three default functions: the assignment operator=, the copy constructor, and the destructor. They are called the big three
- The assignment operator is called to copy the rhs object of the operator= to the lhs object which has already been constructed.
- The copy constructor is called to initialize an object which is being constructed from another object. The default behavior is to allocate memory and copy data member values.
- The destructor is called when delete is applied to an object or when the object is out of scope
- Usually the big three defaults provided by C++ are acceptable.
- But if the data members involve pointers the three default functions may fail to work properly

# Bad Big Three Defaults Example (1)

```
$ cat big3bad.cpp
#include <iostream>
using namespace std;
class V {
private:
    int *v;
public:
    V(int p=0) { v = new int(p); }
    int get() const { return *v; }
    void set(int n) { *v = n; }
};
```

# Bad Big Three Defaults Example (2)

```
int main() {
    V a=1, b(2);
    V c=a, d(b);
    cout <<a.get()<<b.get()<<c.get()<<d.get()<<endl;
    a = d;
    cout <<a.get()<<b.get()<<c.get()<<d.get()<<endl;
    b.set(3);
    cout <<a.get()<<b.get()<<c.get()<<d.get()<<endl;
}
```

```
$ g++ big3bad.cpp
$ a.out
1212
2212
3313
```

# Correct Big Three Overrides Example (1)

```
$ cat big3good.cpp
#include <iostream>
using namespace std;
class V {
private:
    int *v;
public:
    V(int p=0) { v = new int(p); }
    int get() const { return *v; }
    void set(int n) { *v = n; }
```

# Correct Big Three Overrides Example (2)

```
// Override the big three:
V(const V & p) { v = new int( *p.v ); }        // Why not p->v?
const V & operator = (const V & rhs) {
    if(this != &rhs) { // make sure lhs and rhs are diffferent
        *v = *rhs.v;
        return *this;
    }
}
~V() { delete v; }
};
```

# Correct Big Three Overrides Example (3)

```
int main() {
    V a=1, b(2);
    V c=a, d(b);
    cout <<a.get()<<b.get()<<c.get()<<d.get()<<endl;
    a = d;
    cout <<a.get()<<b.get()<<c.get()<<d.get()<<endl;
    b.set(3);
    cout <<a.get()<<b.get()<<c.get()<<d.get()<<endl;
}
```

```
$ g++ big3good.cpp
$ a.out
1212
2212
2312
```

# Order of Object Destruction(1)

```
$ cat big3.cpp
#include <iostream>
using namespace std;
class C {
    public:
    C(int v) {
        cout << "int constructor\t" << this << endl;;
        value = v;
    }
    C(const C & rhs) {
        cout << "copy constructor\t" << this << endl;
        value = rhs.value;
    }
```

# Order of Object Destruction(2)

```
    const C & operator=(const C & rhs) {
        cout << "operator=\t" << this << '\t' << &rhs << endl;
        if(this != &rhs)            // standard alias test
        value = rhs.value;
        return *this;
    }
    ~C() {
        cout << "destructor\t" << this << endl;
    }
    private:
    int value;
};
```

# Order of Object Destruction(3)

```
const C & f(const C para) {
    cout << "inside f()\n";
    return para;
}
main() {
    C x1 = 0;
    C x2(1);
    C y1 = x1;
    C y2(x2);
    x2 = y1;
    y2 = x1;
    f(y2);
}
```

```
$ g++ big3.cpp
$ a.out
int constructor 0xffbff898
    int constructor 0xffbff890
        copy constructor 0xffbff888
            copy constructor 0xffbff880
assignment operator 0xffbff890 0xffbff888
assignment operator 0xffbff880 0xffbff898
                copy constructor 0xffbff878
                inside f()
                destructor 0xffbff878
            destructor 0xffbff880
        destructor 0xffbff888
    destructor 0xffbff890
destructor 0xffbff898
```

# Friends

- A class may grant access to its private members to other classes or other functions by declaring them as friends.

# Friends Example (1)

```
$ cat friend.cpp
class A {
    public: int bare;
    private: int hide;
};
class B {
    void f(A x) { x.bare = 0; }
    void g(A x) { x.hide = 0; }
};
```

```
$ g++ -c friend.cpp
friend.cpp: In member function ‘void B::g(A)’:
friend.cpp:5: error: ‘int A::hide’ is private
friend.cpp:10: error: within this context
```

# Friends Example (2)

```
$ cat friend1.cpp
class A {
    friend class B;         // the class B is friend
    public: int bare;
    private: int hide;
};
class B {
    void f(A x) { x.bare = 0; }
    void g(A x) { x.hide = 0; }
};
```

```
$ g++ -c friend1.cpp
```

# Friends Example (3)

```
$ cat friend2.cpp
class A {
      friend void B(); // the function void B() is friend
      public: int bare;
      private: int hide;
};
void B() {                       // A global function
      A x;
      x.hide = 0;
};
```

```
$ g++ -c friend2.cpp
```

# Return by Constant-Reference Revisited

- The default return mechanism is return by value that involves copying value back to the caller.
- Copying can be avoided by using return by constant-reference.
- When returning by constant-reference, two things must be ensured:
  - The returned object must be alive after the return.
  - The caller should use a const reference variable to receive the returned object.

# Separation of Interface and Implementation

- The interface of a class refers to the declaration of all its member data and functions but without the definition of some or all the member functions.
- The implementation of a class refers to the definitions of the member functions that are not defined in the interface.
- With this separation, functions using a class do not have to be recompiled when some member functions of the class change.
- Member functions defined in the interface are automatically made in-line, saving the cost of routine calls.

# Separation of Interface and Implementation Example (1)

```
$ cat complex.h
class Complex {
    public:
        Complex(int,int);
        int size2();
    private:
        int re, im;
};
```

# Separation of Interface and Implementation Example (2)

```
$ cat complex.cpp
#include "complex.h"
Complex::Complex(int a, int b) {
    re = a;
    im = b;
}
int Complex::size2() {
    return re*re + im*im;
}
```

# Separation of Interface and Implementation Example (3)

```
$ cat testComplex.cpp
#include <iostream>
using namespace std;
#include "complex.h"
int main() {
    Complex a(1,2);
    cout << a.size2() << endl;
}
```

```
$ g++ -c complex.cpp
$ g++ -c testComplex.cpp
$ g++ complex.o testComplex.o
$ a.out
5
```

# Static Members

- A class may have static member data and static member function.
- Static members exists independently of all instances of the class.
- A static data member must be allocated and initialized beyond the class declaration.

# Static Members Example (1)

```
$ cat count.cpp
#include <iostream>
using namespace std;
class C {
    private:
        static int count;
    public:
        C() { count++; }
        static int show() { return count; }
};
```

# Static Members Example (2)

```
// allocation and initialization of static count
int C::count = 0;          // regard static data as global data
int main() {
    C x[1000], y[200], z[80];
    cout << C::show() << endl;
}
```

```
$ g++ count.cpp
$ a.out
1280
```