Object Interactive

Weng Kai
http://fm.zju.edu.cn

C’tor and D’tor

Point::init()

```
class Point {
public:
    void init(int x,int y);
    void print() const;
    void move(int dx,int dy);

private:
    int x;
    int y;
} ;

Point a;
a.init(1,2);
a.move(2,2);
a.print();
```

Guaranteed initialization with the constructor

- If a class has a constructor, the compiler automatically calls that constructor at the point an object is created, before client programmers can get their hands on the object.
- The name of the constructor is the same as the name of the class.

How a constructor does?

```
class X {
  int i;
public:
  X();
};
```

How a constructor does?

```
class X {
  int i;
public:
  X();
};
```

constructor

How a constructor does?

```
class X {
  int i;
public:
  X();
};
```

constructor

```
void f() {
  X a;
  // ...
}
```

How a constructor does?

```
class X {
  int i;
public:
  X();
};
```

constructor

```
void f() {
  X a;
  // ...
}
```

a.X();

Constructors with arguments

- The constructor can have arguments to allow you to specify how an object is created, give it initialization values, and so on.

```
Tree(int i) {…}
Tree t(12);
```

- Constructor1.cpp

The default constructor

- A *default constructor* is one that can be called with no arguments.

The default constructor

- A *default constructor* is one that can be called with no arguments.

```
struct Y {
    float f;
    int i;
    Y(int a);
};
```

The default constructor

- A *default constructor* is one that can be called with no arguments.

```
struct Y {
    float f;
    int i;
    Y(int a);
};
```

```
Y y1[] = { Y(1), Y(2), Y(3) };
```

The default constructor

- A *default constructor* is one that can be called with no arguments.

```
struct Y {
    float f;
    int i;
    Y(int a);
};
```

```
Y y1[] = { Y(1), Y(2), Y(3) };
```

```
Y y2[2] = { Y(1) };
```

The default constructor

- A *default constructor* is one that can be called with no arguments.

```
struct Y {
    float f;
    int i;
    Y(int a);
};
```

```
Y y1[] = { Y(1), Y(2), Y(3) };
```

```
Y y2[2] = { Y(1) };
```

```
Y y3[7];
```

The default constructor

- A *default constructor* is one that can be called with no arguments.

```
struct Y {
    float f;
    int i;
    Y(int a);
};
```

```
Y y1[] = { Y(1), Y(2), Y(3) };
```

```
Y y2[2] = { Y(1) };
Y y3[7];
Y y4;
```

“auto” default constructor

“auto” default constructor

- If you have a constructor, the compiler ensures that construction *always* happens.

“auto” default constructor

- If you have a constructor, the compiler ensures that construction *always* happens.
- *If* (and only if) there are no constructors for a class (**struct** or **class**), the compiler will automatically create one for you.
    - Example:AutoDefaultConstructor.cpp

The destructor

- In C++, cleanup is as important as initialization and is therefore guaranteed with the destructor.
- The destructor is named after the name of the class with a leading tilde (~). The destructor never has any arguments.

The destructor

- In C++, cleanup is as important as initialization and is therefore guaranteed with the destructor.
- The destructor is named after the name of the class with a leading tilde (~). The destructor never has any arguments.

```
class Y {
public:
  ~Y();
};
```

When is a destructor called?

- The destructor is called automatically by the compiler when the object goes out of scope.
- The only evidence for a destructor call is the closing brace of the scope that surrounds the object.

Storage allocation

Storage allocation

- The compiler allocates all the storage for a scope at the opening brace of that scope.

Storage allocation

- The compiler allocates all the storage for a scope at the opening brace of that scope.
- The constructor call doesn’t happen until the sequence point where the object is defined.
  - Examlpe: Nojump.cpp

Aggregate initialization

Aggregate initialization

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

Aggregate initialization

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

Aggregate initialization

- `int a[5] = { 1, 2, 3, 4, 5 };`
- `int b[6] = {5};`
- `int c[] = { 1, 2, 3, 4 };`
  - `sizeof c / sizeof *c`
- `struct X { int i; float f; char c; };`
  - `X x1 = { 1, 2.2, 'c' };`

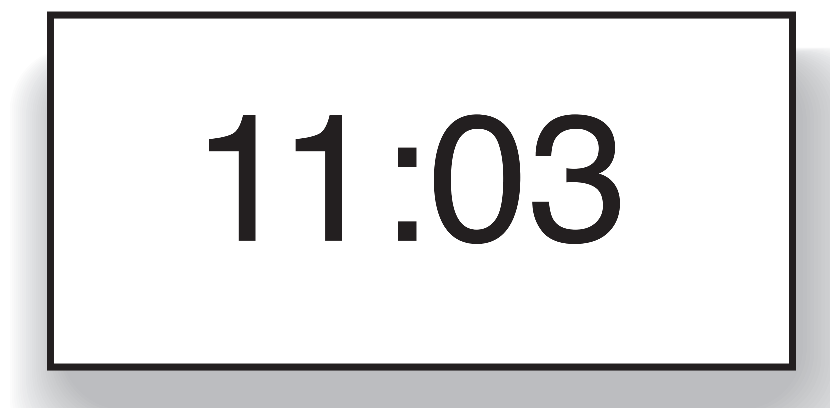
Aggregate initialization

- `int a[5] = { 1, 2, 3, 4, 5 };`
- `int b[6] = {5};`
- `int c[] = { 1, 2, 3, 4 };`
  - `sizeof c / sizeof *c`
- `struct X { int i; float f; char c; };`
  - `X x1 = { 1, 2.2, 'c' };`
- `X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} };`
- `struct Y { float f; int i; Y(int a); };`

Aggregate initialization

- `int a[5] = { 1, 2, 3, 4, 5 };`
- `int b[6] = {5};`
- `int c[] = { 1, 2, 3, 4 };`
  - `sizeof c / sizeof *c`
- `struct X { int i; float f; char c; };`
  - `X x1 = { 1, 2.2, 'c' };`
- `X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} };`
- `struct Y { float f; int i; Y(int a); };`
- `Y y1[] = { Y(1), Y(2), Y(3) };`

Clock display



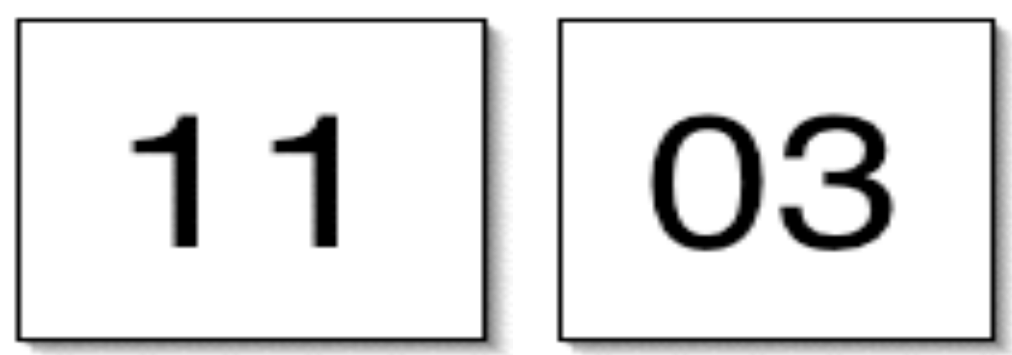
Abstract

- Abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem.
- Modularization is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.

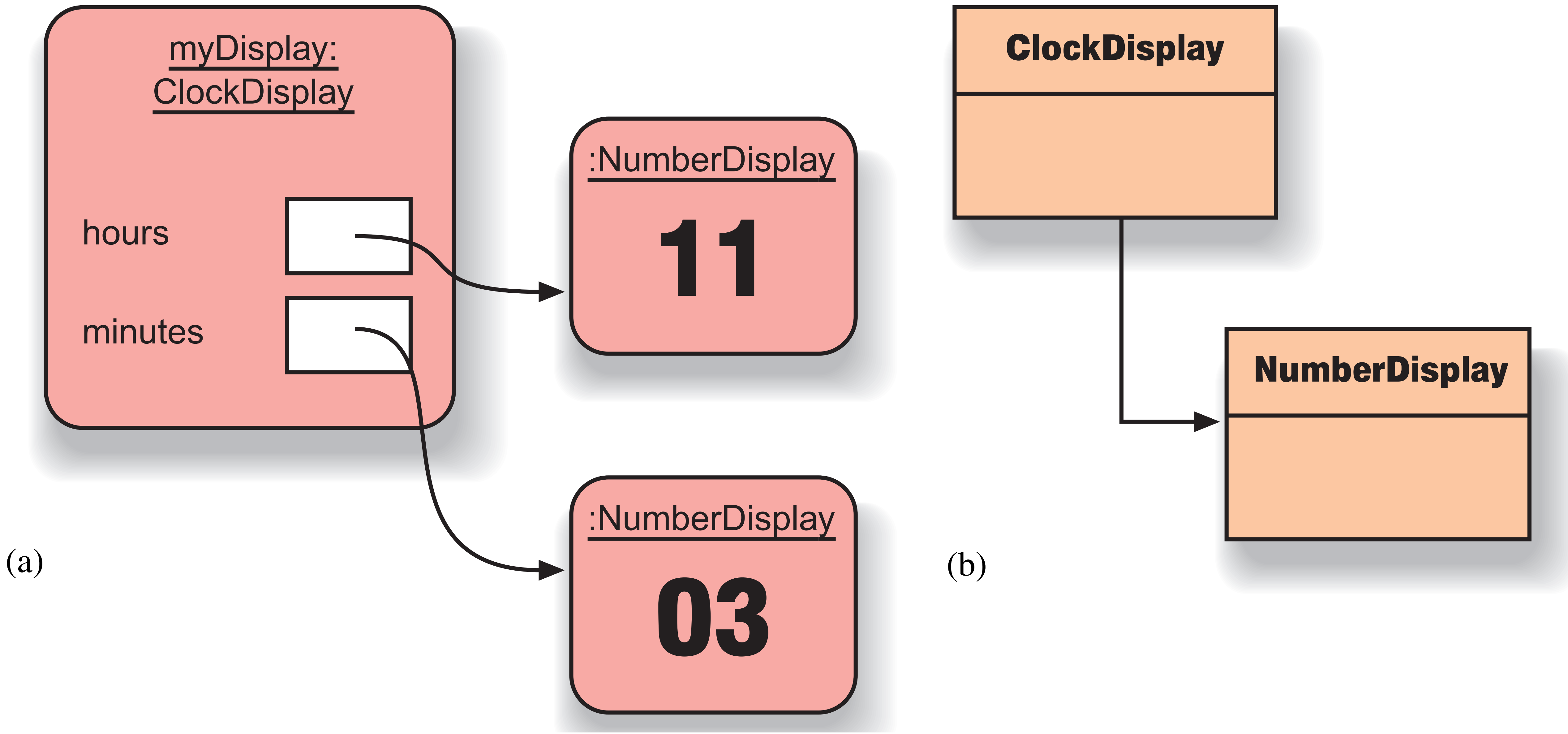
Modularizing the clock display

One four-digit display?

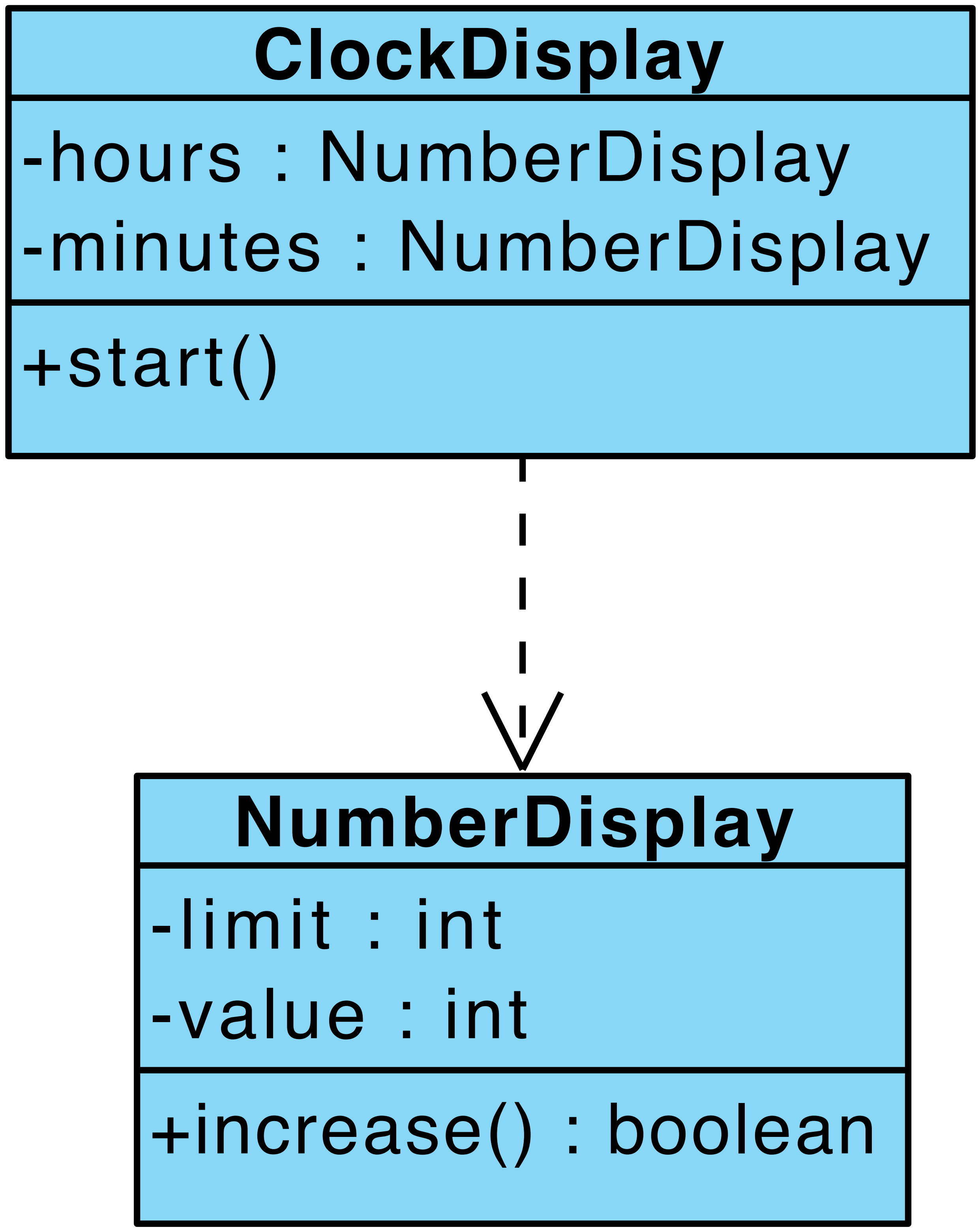
Or two two-digit displays?



Objects & Classes



Class Diagram



Implementation - ClockDisplay

```
class ClockDisplay {
    NumberDisplay hours;
    NumberDisplay minutes;

    Constructor and
    methods omitted.
}
```

Implementation - NumberDisplay

```
class NumberDisplay {
    int limit;
    int value;

    Constructor and
    methods omitted.
}
```

local variable

- Local variables are defined inside a method, have a scope limited to the method to which they belong.

local variable

```
int TicketMachine::refundBalance() {
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```

local variable

```
int TicketMachine::refundBalance() {
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```

A local variable of the same name as a field will prevent the field being accessed from within a method.

Fields,parameters,local variables

- All three kinds of variable are able to store a value that is appropriate to their defined type.
- Fields are defined outside constructors and methods
- Fields are used to store data that persists throughout the life of an object. As such, they maintain the current state of an object. They have a lifetime that lasts as long as their object lasts.
- Fields have class scope: their accessibility extends throughout the whole class, and so they can be used within any of the constructors or methods of the class in which they are defined.

- As long as they are defined as private, fields cannot be accessed from anywhere outside their defining class.
- Formal parameters and local variables persist only for the period that a constructor or method executes. Their lifetime is only as long as a single call, so their values are lost between calls. As such, they act as temporary rather than permanent storage locations.
- Formal parameters are defined in the header of a constructor or method. They receive their values from outside, being initialized by the actual parameter values that form part of the constructor or method call.

- Formal parameters have a scope that is limited to their defining constructor or method.
- Local variables are defined inside the body of a constructor or method. They can be initialized and used only within the body of their defining constructor or method.
- Local variables must be initialized before they are used in an expression – they are not given a default value.
- Local variables have a scope that is limited to the block in which they are defined. They are not accessible from anywhere outside that block.

Initialization

Member Init

- Directly initialize a member
  - benefit: for all ctors
- Only C++11 works

Initializer list

```
class Point {
private:
  const float x, y;
  Point(float xa = 0.0, float ya = 0.0)
     : y(ya), x(xa) {}
};
```

- Can initialize any type of data
  - pseudo-constructor calls for built-ins
  - No need to perform assignment within body of ctor
- Order of initialization is order of ***declaration***
  - Not the order in the list!
  - Destroyed in the reverse order.

Initialization vs. assignment

Initialization vs. assignment

```
Student::Student(string s):name(s) {}
```

- initialization
- before constructor

```
Student::Student(string s) {name=s;}
```

- assignment

Initialization vs. assignment

```
Student::Student(string s):name(s) {}
```

- initialization
- before constructor

```
Student::Student(string s) {name=s;}
```

- assignment
- inside constructor
- string must have a default constructor

Function overloading

- Same functions with different arguments list.

```
void print(char * str, int width); // #1
void print(double d, int width); // #2
void print(long l, int width); // #3
void print(int i, int width); // #4
void print(char *str); // #5

print("Pancakes", 15);
print("Syrup");
print(1999.0, 10);
print(1999, 12);
print(1999L, 15);
```

Example: leftover.cpp

Overload and auto-cast

```
void f(short i);
void f(double d);

f('a');
f(2);
f(2L);
f(3.2);
```

Example: overload.cpp

Default arguments

- A default argument is a value given in the declaration that the compiler automatically inserts if you don't provide a value in the function call.

```
Stash(int size, int initQuantity = 0);
```

- To define a function with an argument list, defaults must be added from right to left.

```
int harpo(int n, int m = 4, int j = 5);
int chico(int n, int m = 6, int j);//illeagle
int groucho(int k = 1, int m = 2, int n = 3);

beeps = harpo(2);
beeps = harpo(1,8);
beeps = harpo(8,7,6);
```

Example: left.cpp

const object

Constant objects

Constant objects

- What if an object is const?

```
const Currency the_raise(42, 38);
```

- What members can access the internals?

Constant objects

- What if an object is const?

```
const Currency the_raise(42, 38);
```

- What members can access the internals?

- How can the object be protected from change?

Constant objects

- What if an object is const?

```
const Currency the_raise(42, 38);
```

- What members can access the internals?
- How can the object be protected from change?
- Solution: declare member functions const
  - Programmer declares member functions to be safe

Const member functions

- Cannot modify their objects

```
int Date::set_day(int d){
   //...error check d here...
   day = d;    // ok, non-const so can modify
}

int Date::get_day() const {
    day++;        //ERROR modifies data member
   set_day(12); // ERROR calls non-const member
   return day;  // ok
}
```

Const member function usage

- Repeat the const keyword in the definition as well as the declaration

```
int get_day () const;

int get_day() const { return day };
```

- Function members that do not modify data should be declared const

- const member functions are safe for const objects

Const objects

Const objects

- Const and non-const objects

```
// non-const object
Date  when(1,1,2001);      // not a const
int day = when.get_day(); // OK
when.set_day(13);          // OK

// const object
const Date birthday(12,25,1994);  // const
int day = birthday.get_day();     // OK
birthday.set_day(14);            // ERROR
```

Constant in class

```
class A {

    const int i;

};
```

- has to be initialized in initializer list of the constructor

Compile-time constants *in classes*

Compile-time constants *in classes*

```
class HasArray {

    const int size;
    int array[size]; // ERROR!

    ...

};
```

Compile-time constants *in classes*

```
class HasArray {

    const int size;
    int array[size]; // ERROR!

    ...

};
```

- Make the const value static:
  - static const int size = 100;
  - static indicates only one per class (not one per object)

Compile-time constants *in classes*

```
class HasArray {
    const int size;
    int array[size]; // ERROR!
    ...
};
```

- Make the const value static:
  - static const int size = 100;
  - static indicates only one per class (not one per object)
- Or use "anonymous enum" hack

```
class HasArray {
    enum { size = 100 };
    int array[size]; // OK!
    ...
};
```

type of function parameters and return value

way in

- void f(Student i);
  - a new object is to be created in f
- void f(Student *p);
  - better with const if no intend to modify the object
- void f(Student& i);
  - better with const if no intend to modify the object

way out

- Student f();
  - a new object is to be created at returning
- Student* f();
  - what should it points to?
- Student& f();
  - what should it refers to?

hard decision

```
char *foo()
{
    char *p;
    p = new char[10];
    strcpy(p, "something");
    return p;
}
void bar()
{
    char *p = foo();
    printf("%s", p);
    delete p;
}
```

hard decision

```
char *foo()
{
   char *p;
   p = new char[10];
   strcpy(p, "something");
   return p;
}
void bar()
{
   char *p = foo();
   printf("%s", p);
   delete p;
}
```

define a pair functions of alloc and free

hard decision

```
char *foo()
{
    char *p;
    p = new char[10];
    strcpy(p, "something");
    return p;
}
void bar()
{
    char *p = foo();
    printf("%s", p);
    delete p;
}
```

define a pair functions of alloc and free

let user take resp., pass pointers in & out

tips

- Pass in an object if you want to store it
- Pass in a const pointer or reference if you want to get the values
- Pass in a pointer or reference if you want to do something to it
- Pass out an object if you create it in the function
- Pass out pointer or reference of the passed in only
- Never new something and return the pointer

Container

A personal notebook

- It allows notes to be stored.
- It has no limit on the number of notes it can store.
- It will show individual notes.
- It will tell us how many notes it is currently storing.

Collection

- Collection objects are objects that can store an arbitrary number of other objects.

What is STL

- STL = Standard Template Library
- Part of the ISO Standard C++ Library
- Data Structures and algorithms for C++.

Why should I use STL?

- Reduce development time.
  - Data-structures already written and debugged.
- Code readability
  - Fit more meaningful stuff on one page.
- Robustness
  - STL data structures grow automatically.
- Portable code.
- Maintainable code
- Easy

C++ Standard Library

- Library includes:
  - A Pair class (pairs of anything, int/int, int/char, etc)
  - Containers
    - **Vector (expandable array)**
    - **Deque (expandable array, expands at both ends)**
    - **List (double-linked)**
    - **Sets and Maps**
  - Basic Algorithms (sort, search, etc)
- All identifiers in library are in std namespace

using namespace std;

The three parts of STL

- Containers
- Algorithms
- Iterators

The ‘Top 3’ data structures

- map
  - –Any key type, any value type.
  - –Sorted.
- vector
  - –Like c array, but auto-extending.
- list
  - –doubly-linked list

All Sequential Containers

- vector: variable array
- deque: dual-end queue
- list: double-linked-list
- forward_list: as it
- array: as “array”
- string: char. array

Example using the vector class

- Use “namespace std” so that you can refer to vectors in C++ library
- Just declare a vector of ints (no need to worry about size)
- Add elements
- Have a pre-defined iterator for vector class, can use it to print out the items in vector

```
#include <iostream>
#include <vector>
using namespace std;

int main( ) {
    vector<int> x;
    for (int a=0; a<1000; a++)
        x.push_back(a);
    vector<int>::iterator p;
    for (p=x.begin();
            p<x.end(); p++)
        cout << *p << " ";
    return 0;
}
```

generic classes

```
vector<string> notes;
```

- Have to specify two types: the type of the collection itself (here: vector) and the type of the elements that we plan to store in the collection (here: string)

vector

- It is able to increase its internal capacity as required: as more items are added, it simply makes enough room for them.
- It keeps its own private count of how many items it is currently storing. Its size method returns the number of objects currently stored in it.
- It maintains the order of items you insert into it. You can later retrieve them in the same order.

Class Exercises

- The code for the vector example exists at vector.cpp. Modify this code so it puts 5000 items in the vector, and then prints out every fifth element
  - Element 0, element 5, element 10, etc.

Basic Vector Operations

- Constructors
  - **vector<Elem> c;**
  - **vector<Elem> c1(c2);**
- Simple Methods
  - **V.size( ) // num items**
  - **V.empty( ) // empty?**
  - **==, !=, <, >, <=, >=**
  - **V.swap(v2) // swap**
- Iterators
  - **I.begin( ) // first position**
  - **I.end( ) // last position**
- Element access
  - **V.at(index)**
  - **V[index]**
  - **V.front( ) // first item**
  - **V.back( ) // last item**
- Add/Remove/Find
  - **V.push_back(e)**
  - **V.pop_back( )**
  - **v.insert(pos, e)**
  - **V.erase(pos)**
  - **V.clear( )**
  - **V.find(first, last, item)**

Class Exercises

- Take a look at the code in vector2.cpp . Predict the output of this program.
- Run the program to check your output.

List Class

- Same basic concepts as vector
  - Constructors
  - Ability to compare lists (==, !=, <, <=, >, >=)
  - Ability to access front and back of list

    x.front(), x.back()
  - Ability to assign items to a list, remove items

    x.push_back(item), x.push_front(item)

    x.pop_back(), x.pop_front()

    x.remove(item)

Sample List Application

- Declare a list of strings
- Add elements
  - Some to the back
  - Some to the front
- Iterate through the list
  - Note the termination condition for our iterator

    p != s.end( )
  - Cannot use **p < s.end( )** as with vectors, as the list elements may not be stored in order

```
#include <iostream>
using namespace std;
#include <list>
#include <string>
int main( ) {
    list<string> s;
    s.push_back("hello");
    s.push_back("world");
    s.push_front("tide");
    s.push_front("crimson");
    s.push_front("alabama");
    list<string>::iterator p;
    for (p=s.begin(); p!=s.end(); p++)
            cout << *p << " ";
    cout << endl;
}
```

Maintaining an ordered list

- Declare a list
- Read in five strings, add them in order
- Print out the ordered list

```
#include <iostream>
    using namespace std;
#include <list>
#include <string>
int main( ) {
    list<string> s;  string t;
    list<string>::iterator p;
    for (int a=0; a<5; a++) {
        cout << "enter a string : ";
        cin >> t;
        p = s.begin();
        while (p != s.end() && *p < t)
            p++;
        s.insert(p, t);
    }
    for (p=s.begin(); p!=s.end(); p++)
        cout << *p << " ";
    cout << endl;   }
```

Maps

- Maps are collections that contain pairs of values.
- Pairs consist of a <u>key</u> and a <u>value</u>.
- Lookup works by supplying a key, and retrieving a value.
- An example: a telephone book.

Using maps

- A map with strings as keys and values

:HashMap

"Charles Nguyen"	"(531) 9392 4587"
"Lisa Jones"	"(402) 4536 4674"
"William H. Smith"	"(998) 5488 0123"

Example Program

```
#include <map>
#include <string>
map<string,float> price;
price["snapple"] = 0.75;
price["coke"] = 0.50;
string item;
double total=0;
while ( cin >> item )
        total += price[item];
```

Example Program

```
#include <map>
#include <string>
map<string,float> price;
price["snapple"] = 0.75;
price["coke"] = 0.50;
string item;
double total=0;
while ( cin >> item )
        total += price[item];
```

Example Program

```
#include <map>
#include <string>
map<string,float> price;
price["snapple"] = 0.75;
price["coke"] = 0.50;
string item;
double total=0;
while ( cin >> item )
        total += price[item];
```

Example Program

```
#include <map>
#include <string>
map<string,float> price;
price[“snapple”] = 0.75;
price[“coke”] = 0.50;
string item;
double total=0;
while ( cin >> item )
        total += price[item];
```

Simple Example of Map

```
map<long,int> root;
root[4] = 2;
root[1000000] = 1000;
long l;
cin >> l;
if (root.count(l)) cout<<root[l]
else cout<<"Not perfect square";
```

Two ways to use Vector

- Preallocate

```
vector<int> v(100);
v[80]=1;  // okay
v[200]=1;  // bad
```

- Grow tail

```
vector<int> v2;
int i;
while (cin >> i)
  v.push_back(i);
```

Example of List

```
list<int> L;
for(int i=1; i<=5; ++i)
  L.push_back(i);
//delete second item.
L.erase( ++L.begin() );
copy( L.begin(). L.end(),
 ostream_iterator<int>(cout,
“,")); // Prints: 1,3,  4,5
```

Iterator

Iterators

- Declaring

```
list<int>::iterator  li;
```

- Front of container

```
list<int> L;
li = L.begin();
```

- Past the end

```
li = L.end();
```

Iterators

- Can increment

```
list<int>::iterator li;
list<int> L;
li=L.begin();
++li;  // Second thing;
```

- Can be dereferenced

```
*li = 10;
```

Algorithms

- Take iterators as arguments

```
list<int> L;
vector<int> V;
// put list in vector
copy(     L.begin(),
        L.end(),
        V.begin()   );
```

List Example Again

```
list<int> L;
for(int i=1; i<=5; ++i)
  L.push_back(i);
//delete second item.
L.erase( ++L.begin() );
copy( L.begin(). L.end(),
 ostream_iterator<int>(cout, “,"));
// Prints: 1,2,3,5
```

Typdefs

- Annoying to type long names
  - map<Name, list<PhoneNum> > phonebook;
  - map<Name, list<PhoneNum> >::iterator finger;
- Simplify with typedef
  - typedef PB map<Name,list<PhoneNum> >;
  - PB phonebook;
  - PB::iterator finger;
- Easy to change implementation.

Using your own classes in STL Containers

- Might need:
  - Assignment Operator, `operator=()`
  - Default Constructor
- For sorted types, like `map<>`
  - Need less-than operator: `operator<()`
    - Some types have this by default:
      - int, char, string
    - Some do not:
      - char *

Example of User-Defined Type

```
struct point
    {
    float x;
    float y;
    }
vector<point> points;
point p; p.x=1; p.y=1;
points.push_back(1);
```

Example of User-Defined Type

- Sorted container needs sort function.

```
struct full_name {
    char * first;
    char * last;
    bool operator<(full_name & a)
      {return strcmp(first, a.first) < 0;}
    }
map<full_name,int> phonebook;
```

Performance

- Personal experience 1:
  - STL implementation was 40% slower than hand-optimized version.
    - STL: used deque
    - Hand Coded: Used “circular buffer” array;
  - Spent several days debugging the hand-coded version.
  - In my case, not worth it.
  - Still have prototype: way to debug fast version.

Performance

- Personal experience 2
- Application with STL list ~5% slower than custom list.
- Custom list “intrusive”
  - struct foo {
  -   int a;
  -   foo * next;
  - };
- Can only put foo in one list at a time ☹

Pitfalls

- Accessing an invalid vector<> element.

```
vector<int> v;
v[100]=1;  // Whoops!
```

Solutions:

- use push_back()
- Preallocate with constructor.
- Reallocate with reserve()
- Check capacity()

Pitfalls

- Inadvertently inserting into map<>.

  if (foo[”bob”]==1)

  //silently created entry “bob”

  Use count() to check for a key without creating a new entry.

  if (foo.count(“bob”))

Pitfalls

- Not using empty() on list<> .
  - Slow

    ```
    if ( my_list.count() == 0 ) { ... }
    ```
  - Fast

    ```
    if ( my_list.empty() ) {...}
    ```

Pitfalls

- Using invalid iterator

```
list<int> L;
list<int>::iterator li;
li = L.begin();
L.erase(li);
++li;              // WRONG
```

- Use return value of erase to advance

```
li = L.erase(li);  // RIGHT
```

Common Compiler Errors

- vector<vector<int>> vv;

  missing space

  lexer thinks it is a right-shift.

- any error message with pair<...>

  map<a,b> implemented with pair<a,b>

Other data structures

- set, multiset, multimap
- queue, priority_queue
- stack , deque
- slist, bitset, valarray