505 22240 / ESOE 2012 Data Structures: Lecture 5
Binary Search and Inheritance

§ The Stack and Heap

◎Two separate pools of memory

・The heap stores all dynamic objects, including arrays and all corresponding class variables.

・The stack stores all local variables, including parameters.

・When a method is called, C++ creates a stack frame (aka. activation record) that stores the

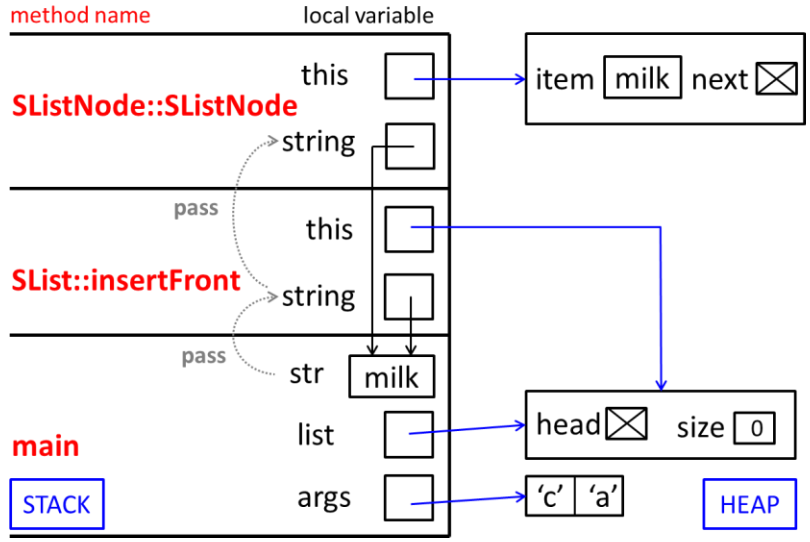
(1) parameters to be processed by the called method,

(2) local variables in the calling method &

(3) return statement and expression in the calling method.

・e.g.

```
int main(char* args) {
    string str = "milk";
    SList* list = new SList;
    list->insertFront(str);
    //…
}
```



・When method finishes, its stack frame is erased.

◎Parameter passing by value and pointer

・e.g.

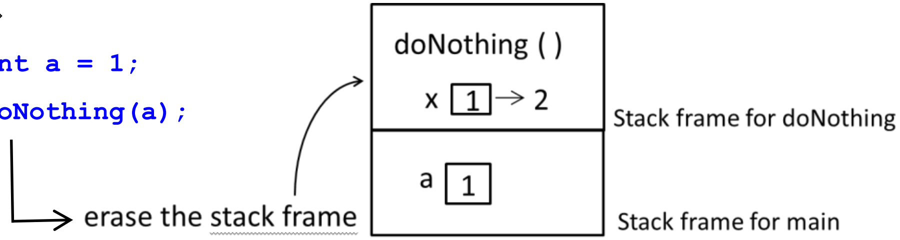
```
class IntBox {
public:
    int i;

    void doNothing(int x) {
        x = 2;
    }
    void set3(IntBox* ib) {
        ib->i = 3;
    }
```

```
    void badSet4(IntBox* ib) {
        ib = new IntBox;
        ib->i = 4;
    }
};
```

⇨

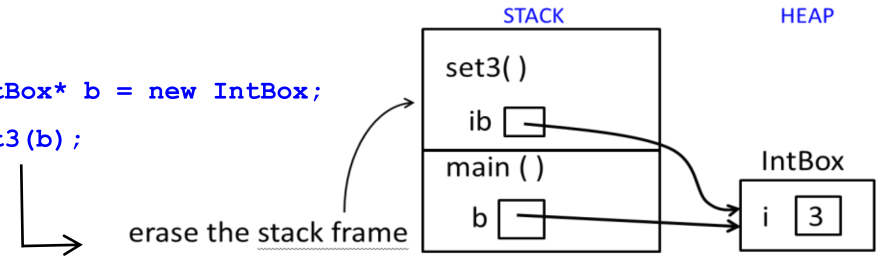
```
int a = 1;
doNothing(a);
```



· When a parameter is a pointer (reference), the reference is copied, but not the thing points to.

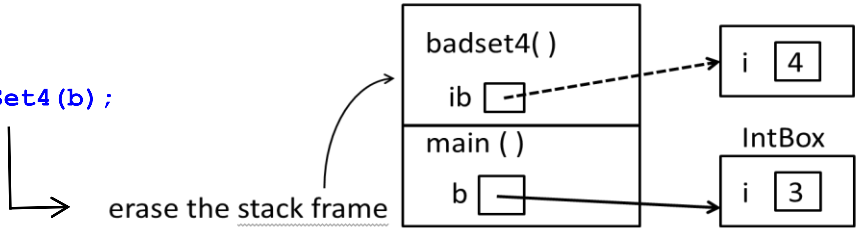
⇨

```
IntBox* b = new IntBox;
set3(b);
```



⇨

```
badSet4(b);
```

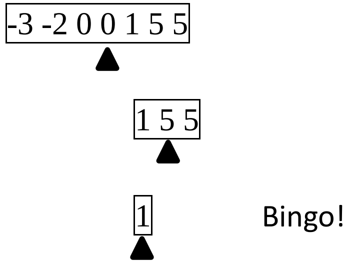


§ Binary Search

· Search a sorted array → for value "findMe".

· If we find "findMe", return its array index; otherwise, return FAILURE.

· e.g. search "1" (looking for the middle value and check.)



◎Recursion base cases

① findMe = middle element: return its index.

② Subarray of length zero: return FAILURE.

· e.g.

```
const int FAILURE = -1;
int bsearch(int *i, int left, int right, int findMe) {
    if (left > right) {
        return FAILURE;
    }
    int mid = (left + right) / 2;
    if (findMe == i[mid]) {
        return mid;
    } else if (findMe < i[mid]) {
        return bsearch(i, left, mid-1, findMe);
    } else {
        return bsearch(i, mid+1, right, findMe);
```

```
    }
}
```

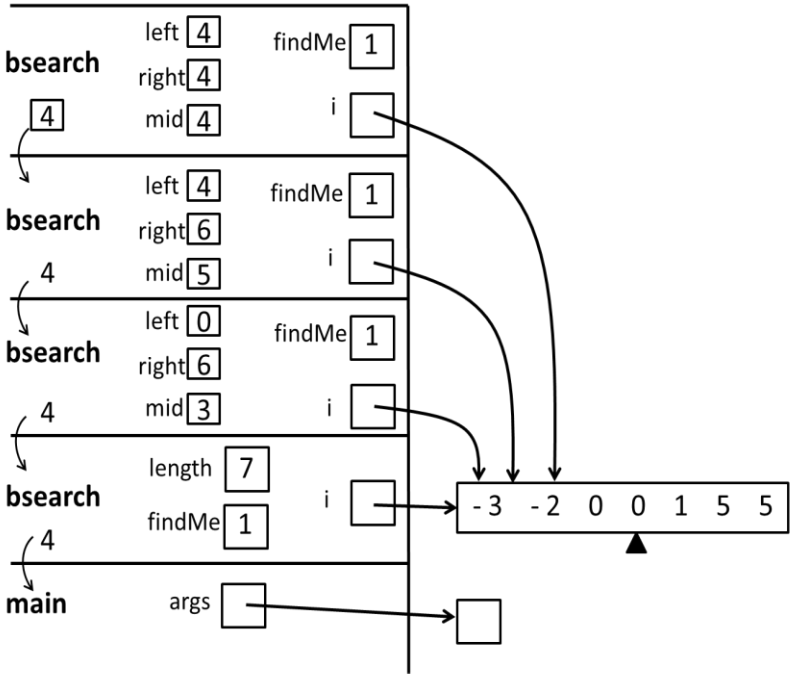
◎How fast does the binary search perform?

n … n/2 … n/4 … n/8 …… 1

⇨ Takes $\log_2 n$ recursive bsearch calls.

◎Stack & Heap Analysis

```
int bsearch(int *i, int findMe) {
    int length = sizeof(i) / sizeof(int);
    return bsearch(i, 0, length-1, findMe);
}
```



§ Inheritance

・Inheritance is a compile-time mechanism in C++ that allows you to extend a class (called the <u>base class</u>, <u>parent class</u>, or <u>superclass</u>) with another class (called the <u>derived class</u>, <u>child class</u>, or <u>subclass</u>).

・e.g.

```
class TailList : public SList {
    /* head and size inherited from SList. */
private:
    SListNode* tail;

public:
    void insertEnd(const string& str) {
        /* Your solution here. */
    }

    /* Methods in SList are inherited. */

    void insertFront(const string& item) {
        SList::insertFront(item);
        if (size == 1) {
            tail = head;
        }
    }
};
```

・TailList is a <u>subclass</u> of SList.

・SList is the <u>superclass</u> of TailList.

◎A Subclass can modify a superclass in 3 ways:

(1) It can declare new fields.

(2) It can declare new methods.

(3) It can override old methods with new implementations.

◎Inheritance & Constructors

・When a derived class is constructed, you need to take care that the appropriate constructor is called for its base class.

・The constructor for a base class needs to be called in the initializer list of the derived class.

・Bottom-up class hierarchies in C++: base class first, then its members, then the derived class.

・e.g. Default constructor:

```
TailList( ) {
    SList( );            // sets size = 0, head = NULL.
    tail = NULL;
}
```

・With parameters:

```
TaiList(int x): SList(x) {
    tail = NULL;      └──> Sets size = x.
}
```

◎The "protected" keyword

・Make change to the superclass SList:

```
class SList {
protected:
    SListNode* head;
    int size;
    //…
};
```

・"protected" is a level of protection somewhere between "public" and "private".

・A "protected" field / method is visible to declaring class and all its <u>subclasses</u>.

・"private" fields aren't visible to subclasses.

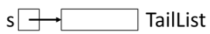
◎Static and Dynamic Binding

Every TailList IS an SList

```
SList* s = new TailList;         // Fine!
TaiList* t = new SList;        // COMPILE-TIME ERROR
```

・Static type: the type of a variable, e.g., **SList** in the first statement.

・Dynamic type: the class of the object the variable points to, e.g., **TailList** on the right-hand side of the first statement, as shown below.



★Static binding:

```
SList* a = new SList;
a->insertEnd(str);          // calls SList::insertEnd( )
s->insertEnd(str);          // calls SList::insertEnd( )
```

⇨ C++'s default action is to consider the function of an object's <u>declared type</u>, not its actual type.

★Dynamic binding:

・Call the function of the corresponding object, with the keyword "virtual" added to the function's declaration.

```
class SList {
    virtual void insertEnd(const string& str) { … }
    //…
};
class TailList {
    virtual void insertEnd(const string& str) { … }
    //…
};
s->insertEnd(str);      // calls TailList::insertEnd( )
a->insertEnd(str);      // calls SList::insertEnd( )
```

・If a base class defines any virtual functions, it should define a virtual destructor, even if it is empty, e.g., `virtual ~SList() { }` in the `Slist` class.

◎Subtleties of Inheritance

① Suppose we write a new method in TailList called eatList( ). We can't call eatList( ) on SList.

```
TailList* t = new TailList;
t->eatList( );                // O.K.
SList* s = new TailList;      // O.K.
s->eatList( );                // COMPILE-TIME ERROR
```

Why?

⇨ Not every SList has an "eatList( )" method.

∴C++ can't use dynamic binding on s.

② Dynamic Cast:

```
dynamic_cast<desired_type>(expression)
```

The dynamic_cast operation converts "expression" to an object of type "desired_type".

```
TailList* sp = dynamic_cast<TailList*>(s);
// cast s to TailList*
sp->eatList( );             // Groovy!
```

・If an illegal pointer cast is attempted, the result is a null pointer.

③ Static Cast:

```
SList* s = new SList;
TailList* t = new TailList;
s = t;          // O.K.
t = s;          // COMPLITE-TIME ERROR
t = static_cast<TailList*>(s);          // COMOPILE O.K.
// RUN-TIME DANGER!
```

・static_cast conversion uses NO runtime check.

・You need to be careful to ensure that objects are cast to the correct data types.

§ Abstract Classes

・A class whose sole purpose is to be extended.

・An abstract class must contain at least ONE pure virtual function → a function that is set equal to zero in the class declaration (lacks an implementation).

```
class List {
protected:
    int size;

public:
    int length( ) {return size;}
    virtual void insertFront(const string& item) = 0;
```

↳ pure virtual function

```
};
List* myList;                  // O.K.
myList = new List;           // COMPILE-TIME ERROR
```

・Abstract classes don't allow you to create objects directly.

・You need to inherit abstract classes to do so:

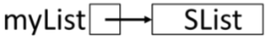
```
class SList : public List {
protected:
    // inherits "size"
    SListNode* head;
public:
    // inherits "lengh( )"
    virtual void insertFront(const string& item) { … }
};
```

◎A non-abstract class may never

・Contain a pure virtual function method.

・Inherit one without providing an implementation.

```
List* myList = new SList;        // O.K.
myList->insertFront(str);
```



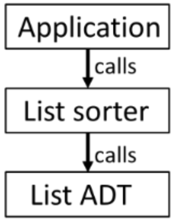
◎One list sorter can sort every kind of List.

```
void listSort(List l) { … }
```

★Subclasses of list: SList, DList, TailList, …

・TimedList: records time spent during operations.

・TransactionList: logs all changes on a disk.



The application, not the list sorter, chooses what kind of list is used.

§ Field Shadowing

・Just as methods can be overridden in subclasses, fields can be "shadowed" in subclasses.

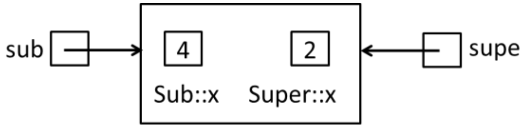
・e.g.

```
class Super {
public:
    int x;
    virtual int f( ) {return 2;}
    Super( ) {x = 2;}              // constructor
};

class Sub : public Super {
public:
    int x;            // shadows Super::x
    virtual int f( ) {return 4;}
    // overrides Super::f( )
    Sub( ) {Super( );    x = 4;}      // constructor
    void g( ) {
        int i;
```

```
        i = this->x;        // 4
        i = Super::x;       // 2
    }
};
```

```
Sub* sub = new Sub;
Super* supe = sub;
```




```
int i;
i = supe->x;                          // 2
i = sub->x;                           // 4
i = static_cast<Super*>(sub)->x;      // 2
i = static_cast<Sub*>(supe)->x;       // 4

i = supe->f();                        // 4
i = sub->f();                         // 4
i = static_cast<Super*>(sub)->f();    // 4
i = static_cast<Sub*>(supe)->f();     // 4
```

· The last four statements yield the same results. Since both variables pointing to a Sub, the method Sub::f() always overrides Super::f().

· Field shadowing is a nuisance.

· Avoid having fields in subclasses whose names are the same as fields in their superclasses.