

Application classes

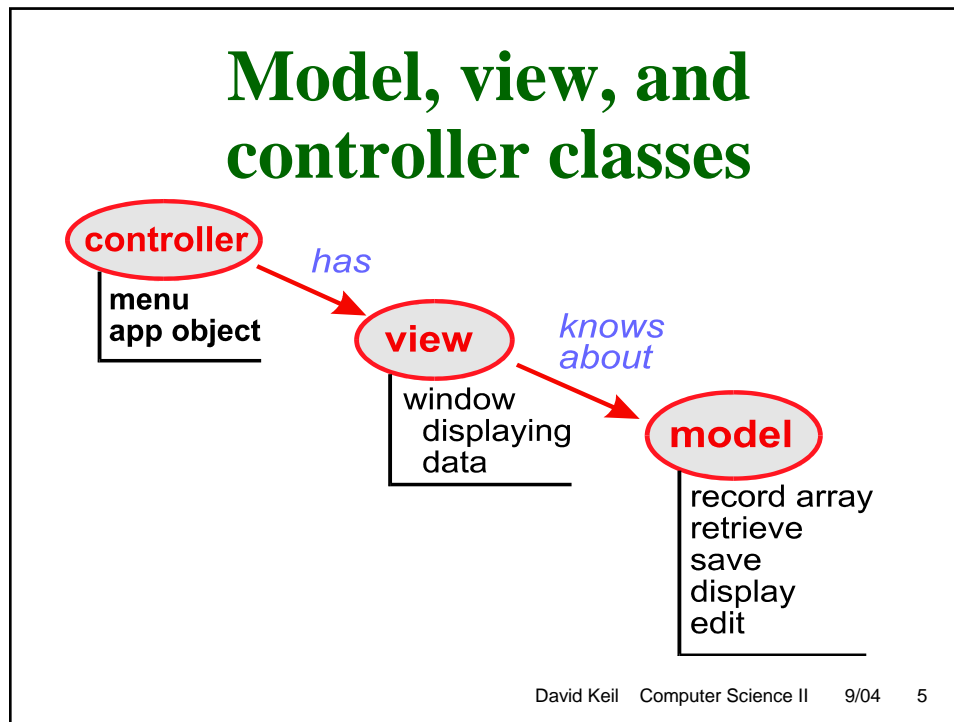
- Used in Windows and Java programming
- A user-interface library defines a general-purpose application class
- Application programmer defines a class that *inherits* from library class, extends its features
- Application programmer may focus on special purpose of application rather than on user-interface details

David Keil Computer Science II 9/04 3

Model-view-controller architecture

Kind of class	Example
<i>Model</i>	Array of database records Spreadsheet cells in linked-list grid
<i>View</i>	Window Button
<i>Controller</i>	Menu Instance of application class

David Keil Computer Science II 9/04 4



Subtopic: Separate compilation

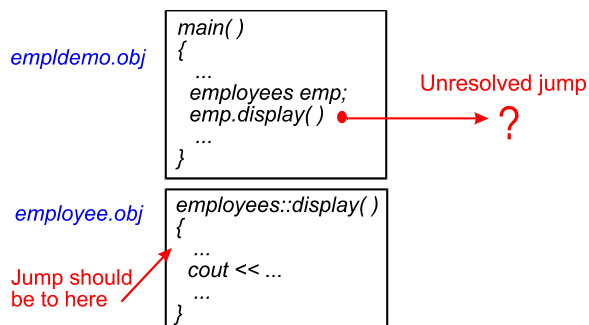
- *Project*: a set of source files ready to link together after compilation
- Exactly one source file has definition of *main*
- Other source files (e.g., libraries) may define functions called from *main*, etc.
- Programmer selects files to comprise the project
- *Build* command automatically recompiles any source files modified since last compile and links them into an *.exe* file

.obj files (linkable object code)

- Contain machine code
- Result of compilation
- May or may not define *main*
- Function call addresses may not be resolved
- Not executable
- Raw material to the linker

David Keil Computer Science II 9/04 7

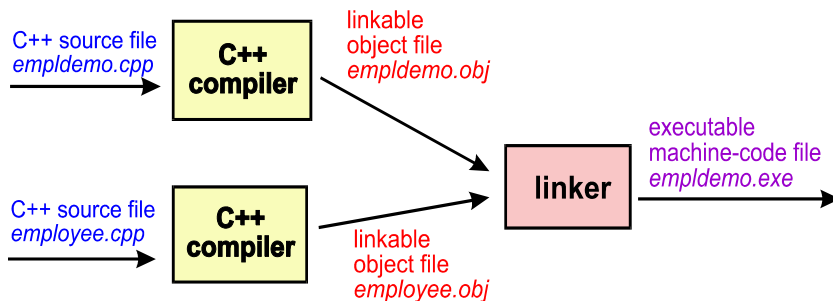
Unresolved jumps



- The compiler may leave the destination addresses of function-call jumps *unresolved*
- This is because a function may be undefined in the source file containing the call

David Keil Computer Science II 9/04 8

The linker resolves function calls



- The linker takes the compiled source files from a project, in *.obj* form, and determines destination addresses of unresolved function-call jumps

David Keil Computer Science II 9/04 9

Why use separate header files?

- *Problem:* Multiple source files may use a type how to make it available to all?
- *Solution:* Include header file containing type declaration in all source files that use that type
- Header files (*.h*) contain constant and type declarations
- Compilable source files (*.c, .cpp*) contain function definitions

David Keil Computer Science II 9/04 10

Advantages of separate compilation

- Makes class libraries available for reuse
- Permits updates to libraries to be automatically incorporated in applications by doing a new *Build*
- Speeds up creation of *.exe* files by avoiding unnecessary recompiles of library code
- Permits developers to distribute libraries as *.obj* files while protecting source

David Keil Computer Science II 9/04 11

Testing and correctness

- Testing can only prove the existence of faults, not their absence
- The cost of correcting a fault late in development process is great
- Regression testing finds errors introduced by maintenance process
- Integration testing determines whether separately developed modules work together
- Software developers sometimes use formal methods of program verification
- Engineering applies teamwork, standards, science, and math to problem solving

David Keil Computer Science II 9/04 12

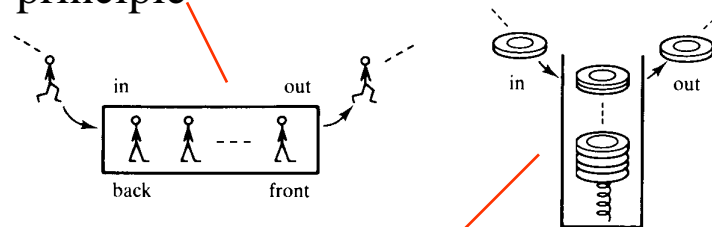
Subtopic: Choosing appropriate data structures

- What is the specialized task?
- Is the collection ordered?
- Are search and insertion, deletion times critical?
- Guideline: Use libraries, such as the Standard Template Library (STL)
- Two collections with restricted access: *Stack, Queue*

David Keil Computer Science II 9/04 13

Stacks and queues

- Specialized collections with restricted access
- A queue works on first-in, first-out principle



- A stack is last-in, first-out (LIFO)

David Keil Computer Science II 9/04 14

Queue operations

- Initialize
- See if empty
- See if full*
- Enqueue
- Dequeue

Stack operations

- Initialize
- See if empty
- See if full*
- Push
- Pop

* *A stack or queue may be full in array implementation; is virtually unbounded in linked-list implementation*

David Keil Computer Science II 9/04 15

Applications for queues

- Maintain chronological order of print jobs or transactions while waiting for one to complete
- Simulate retail or bank operation
- In general, store and update a list of items that must be kept in chronological order by arrival time

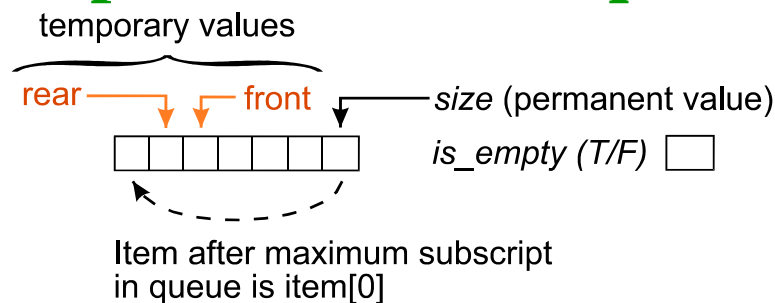
David Keil Computer Science II 9/04 16

Three queue implementations

- Inefficient, array-based:
Enqueue(x): Append x to array
Dequeue: Retrieve first element;
 shift all others one to the left
- More efficient, array-based: **circular array**
- More efficient, list-based:
linked list with tail pointer
Enqueue(x): Append x to list using *tail* ptr
Dequeue: Retrieve and delete first node

David Keil Computer Science II 9/04 17

Circular-array implementation of queue

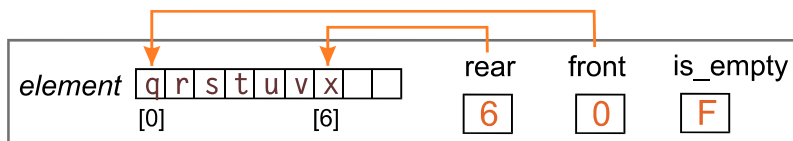


- Either *front* or *rear* may wrap back to 0

David Keil Computer Science II 9/04 18

A queue structure type

```
#define MAX_Q_SZ 9
struct queues
{
    int element[MAX_Q_SZ];
    int front, rear;
    bool is_empty;
};
```



- *Enqueue* operation increments *rear*
- *Dequeue* increments *front*

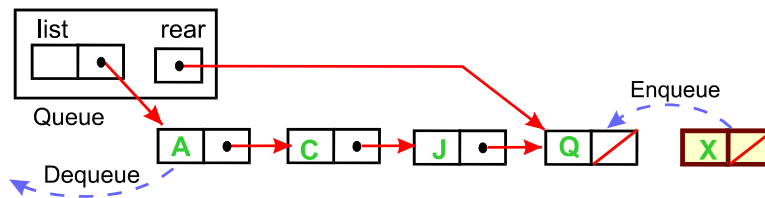
David Keil Computer Science II 9/04 19

Implementing a queue with linked list

- Dequeue:
 1. $c \leftarrow \text{data}(\text{next}(\text{header}))$
 2. delete first node
 3. return c
- Enqueuing new node may require traversal to rear ($O(n)$) to append to list
- Linked list with extra pointer to rear makes *enqueue* instantaneous

David Keil Computer Science II 9/04 20

Operations on queues based on lists



David Keil Computer Science II 9/04 21

Queue of *char*

```

struct queues
{
    char_list L;
    nodes* rear;
};

struct nodes
{
    nodes data;
    nodes* next;
};

struct char_list
{
    nodes header;
};

bool q_empty(queue q)
{
    return(q.L.header.next == NULL);
};

char q_dequeue(queue& q)
{
    if (q.L.header.next == NULL) exit();
    char ret_val = q.L.header.next->data;
    q.L.delete_node(&q.L.header);
    return ret_val;
};

void q_enqueue(queues& q, char c)
{
    append(q.L, c);
    // or: list_insert(q.L, rear, c);
};

```

David Keil Computer Science II 9/04 22

Stacks

- Restricted-access collection
- Insert on top (*push*)
- Retrieve/delete from top (*pop*)
- Last-in, first-out
- Implementations:
 - array
 - linked lists
- Backtracking

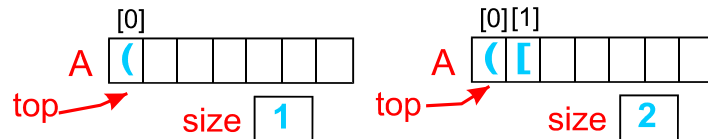
David Keil Computer Science II 9/04 23

Why activation records use a stack structure

- Program must manage local data of all currently executing functions
- Local variable and parameter space must be freed when a function terminates
- Result: a function call triggers *push*, termination triggers *pop*

David Keil Computer Science II 9/04 24

One array implementation of stack



- *push(c)*
requires shifting *all* array elements to right; $A[0] \leftarrow c$
- *pop()*
requires shifting all elements to left, returning $A[0]$
- **Problem:** describe a more efficient array implementation

David Keil Computer Science II 9/04 25

Application for stack: checking delimiter balance

```

balance1.dat:           balance2.dat:
{                         {
  a =                     a =
    (b[1] + c[2])         (b[1] + c(2])
    * d;                   * d;
}                         }

balance3.dat:
{
  a = ]
}

```

- To determine: are all braces, parentheses, brackets correctly balanced?

David Keil Computer Science II 9/04 26

Check delimiter balance

```
For each item x of input stream
  If it is a left delimiter
    push item on stack
  If it is a right delimiter
    If stack is empty
      show error ("Unmatched right")
     $y \leftarrow \text{pop}(\text{stack})$ 
    If x is not mate of y
      show error ("Unmatched left")
  If stack not empty
    show error ("Unmatched left")
```

David Keil Computer Science II 9/04 27

Linked-list implementation of stack

- *Insertion* at head of list (push at top of stack) is $O(1)$ steps
- *Deletion* of first node (pop) is $O(1)$ steps
- Size of stack is limited only by storage available

David Keil Computer Science II 9/04 28

A linked-list-based stack of *char*

```

struct lists
{
    nodes header;
};

struct stacks
{
    lists list;
};

struct nodes
{
    char data;
    nodes *next;
};

void stack_init(stacks* s);
void stack_push(stacks* s, char c);
char stack_pop(stacks* s);
boolean stack_is_empty(stacks s);

```

[balance.c]

David Keil Computer Science II 9/04 29

Operations on *char* stack

```

char stack_pop(stacks* stack)
{
    lists* p_list = &(stack->list);
    char ch;
    if (! stack_is_empty(*stack))
    {
        ch = p_list->header.next->data;
        list_delete(&(p_list->header));
        return ch;
    }
    else
        return '\0';
}

void stack_push(stacks* p_stack, char ch)
{
    list_prepend(&(p_stack->list), ch);
};

```

[balance.c]

David Keil Computer Science II 9/04 30

See what's at front of list

Delete front node from list: