Pointers & Dynamic Arrays

Allocating memory at run-time
Review of pointers

• Pointer: variable that holds the address of a memory location as its value

• Pointers are used to indirectly reference the contents of the memory location
Review of pointers

• Each byte of memory is associated with a physical (numeric) address
• When a variable is declared, a set of bytes is allocated -- address of the first byte is address of the variable
• A pointer can reference this address
**Pointers**

**Pointer declaration & use**

```c
int n, *ip;  // int variable & int
             // pointer declared
n = 5;       // initializes int variable
ip = &n;     // assigns address of n
to ip -- ip is now a
            // reference to n
```
**Pointer operations continued**

*ip = 7;  // dereferences ip -- assigns new value to memory ip points to (so \( n = 7 \))

*ip++;  // adds 1 to \( n \)

ip++;  // moves pointer to next location -- caution!!! -- in this instance, pointer now references unknown location
Assignment operator & pointers

char c, d, *e, *f;
e = &c;
f = &d;
...
e = f;
...
*f = *e;

• Can assign address of ordinary variable to pointer of same type
• Can assign value of one pointer to another of same type
• Can assign content referenced by one pointer to content referenced by another
Summary of pointer operations

• Declaration: type *id1,*id2;

• Assignment:
  – address:  id = &variable;
  – pointer:  id2 = id1; // must be same type

• Dereferencing:
  – in assignment:  *id2 = value;
  – in input/output:  cin >> *id1;  cout << *id1;

• Pointer arithmetic:  id2++;
Dynamic Variables

- Dynamic variable is neither declared nor initialized, and has no identifier -- only reference is through pointer
- Created at runtime (ordinary variables created at compile time)
- Operator new used to allocate memory for dynamic variable
Dynamic variable example

double *dp; // declares pointer

dp = new double; // allocates dynamic // memory

*dp = 1.01; // assigns value to // unnamed dynamic // variable via pointer
Dynamic Arrays

• Can use new to allocate entire array
• Pointer references first array index
• Array version especially useful because size of array can be left unspecified until runtime
  – can ask user for # of input elements
  – dynamic behavior = determined when program is running
Dynamic Array Example

int size, *nums; // size will hold array size;
    // nums will be used to
    // reference array elements

cout << "How many values to input? ";
cin >> size;

nums = new int [size]; // creates array
Dynamic arrays continued

- Dynamic array elements can be referenced just like static array elements -- use pointer name & index: `nums[0]`, `nums[1]`, etc.

- Once memory is allocated with `new`, it can’t be re-used until it is released with the `delete` operator:

```cpp
    delete dp;        // releases dynamic variable
    delete []nums;    // releases dynamic array
```
Pointers & arrays as parameters

• Pointers as value parameters
  – simulates pass by reference in C
  – can change value at memory location referenced by pointer, but can’t change pointer

• Array parameters
  – array name acts as pointer to first element
  – size need not be specified
  – changes to parameter affect actual argument
Syntax for pass by value

• Pointers: type fname (type *pointer);
  – example: int f1 (int *ip);

• Arrays: type fname (type array[], type size);
  – example: int f2 (int nums[], size_t extent);
  – although it’s not required that we pass the size, it’s still handy to have it!
Pointers & arrays as const parameters

• Pointer example: int cubit (const int *ip);
  – function can examine item referenced by ip
  – can’t change its value

• Array example:
  – double avg (const double data[], size_t size);
  – can examine & use values in data array, but can’t change them
Pointers as reference parameters

• Use when:
  – need to change value of pointer (address pointer is referencing)
  – need this change to affect the actual parameter

• Example: function that assigns dynamic array to a pointer -- prototype below:
  ```c
  void allocate_doubles (double*& p, size_t& n);
  ```
Allocate_doubles function

```c
void allocate_doubles (double*& p, size_t& n)
{
    cout << "Enter the number of spaces to allocate: ";
    cin >> n;
    p = new double[n];
}

function use:
    double *dnums;
    size_t array_size;
    allocate_doubles (dnums, array_size);
```
Using a dynamic array to create a container class -- Bag ADT II

Original (static) bag:
```cpp
class Bag {
    ...
    private:
        Item data [CAPACITY];
        size_t used;
    }
};
```

Dynamic bag:
```cpp
class Bag {
    ...
    private:
        Item* data;
        size_t used;
        size_t capacity;
    }
};
```
Invariant for dynamic Bag

• Member variable used stores a count of the Items in the Bag
• Member variable data is a pointer to a dynamic array in which Items are stored
• Member variable capacity stores the current maximum size of the Bag
Member functions in dynamic Bag

• Modifiers allocate dynamic memory as needed
  – constructor: sets Bag’s initial capacity
  – insert, += : allocate new, larger dynamic array if increased capacity is needed to perform operation
  – resize: new member function -- enlarges Bag -- used by insert & +=
Member functions in dynamic Bag

• Value semantics: functions that determine how values are copied from one object to another -- can’t use automatic defaults with dynamic memory
  – assignment operator
  – copy constructor

• Both must be defined as Bag member functions
Value semantics -- why not automatic?

Suppose you have the following declaration:

```c
Bag x, y;
```

and the following statement:

```c
x = y;
```

Automatic assignment will copy values from y’s member variables to x, including:

```c
x.data = y.data;
```

This doesn’t make a copy of the dynamic array -- instead, it just means there are now 2 pointers to the same dynamic array.
New member function: destructor

• Member function that returns dynamically allocated memory when no longer needed
• Always named ~name (ex. ~Bag())
• Has no parameters or return value
• Automatically activated when object becomes inaccessible
• Almost never called explicitly
Destructor uses

- Local variable in a function -- if it is a dynamic object, the object’s destructor is called when the function returns
- Dynamic object passed by value to a function -- same as with local variable
- A dynamic variable of a dynamic object type is deleted
Dynamic bag: class declaration & definitions

class Bag
{
  enum {DEF_CAP = 30};
  // sets default capacity for bag -- can always be resized
  typedef char Item;
  Bag(size_t initCap = DEF_CAP);
  // constructor uses default argument so that user program
  // can set different initial size if desired
  Bag (const Bag& source); // copy constructor
  ~Bag( ); // destructor
  ...
}
void resize ( size_t new_cap);
// precondition: new_cap is a positive, non-zero integer
// postcondition: Bag capacity is changed to new_cap; if
// new_cap is less than used, Bag capacity = used
void insert (const Item& entry);
// postcondition: Item is inserted in the Bag; if doing so exceeds
// Bag capacity, Bag is resized
void remove (const Item& target);
// postcondition: if target value is in the Bag, an instance of the
// value is removed; if target value is not there, Bag is unchanged
…
void operator += (const Bag& addend);
// postcondition: contents of addend Bag have been added
// to current Bag; if process causes current Bag capacity to be
// exceeded, Bag is resized
void operator = (const Bag& source);
// postcondition: contents of current Bag are now identical to
// contents of source Bag
size_t size( ) const {return used};
size_t occurrences (const Item& target) const;
// const functions are unchanged from static Bag
...
friend Bag operator + (const Bag& b1, const Bag & b2);
// unchanged from static Bag
private:
    Item *data;  // pointer to partially -filled dynamic array
    size_t used; // how much of array currently in use
    size_t capacity; // current Bag capacity
};
Bag member functions

// default constructor
Bag::Bag(size_t init_cap)
{
    data = new Item[init_cap];
    capacity = init_cap;
    used = 0;
}

Bag member functions

// copy constructor
Bag::Bag (const Bag& source)
{
    data = new Item[source.capacity];
capacity = source.capacity;
used = source.used;
    for (int n=0; n<used; n++)
        data[n] = source.data[n];
}

pointers
Bag member functions

// destructor
Bag::~Bag()
{
    delete [] data;
}

Bag member functions

// resize
void Bag::resize(size_t new_cap)
{
    Item *larger;         // temp pointer to new
                           // larger array
    if (new_cap == capacity)
        return;
    return;

    // Bag is already the requested size

    ...
Resize function continued

... if (new_cap < used)
    new_cap = used;
// can’t shrink Bag smaller than what’s
// already in use
larger = new Item[new_cap];
// allocate sufficient memory for new array
...

pointers
Resize function continued

...  

for (int n=0; n < used; n++)
    larger[n] = data[n];
data = new_cap;
capacity = new_cap;
}
Insert function

```c++
void Bag::insert(const Item& entry) {
    if (used == capacity)
        resize (used + 1);
    data[used] = entry;
    used++;
}
```
Remove function

```cpp
void Bag::remove(const Item& target) {
    int n; // will be index of target value
    for (n=0; (n<used) && (data[n] != target); n++);  
        // search array until target found or no more to search
    if (n == used)
        return;  // quit if target not found
    used--; 
    data[n] = data[used];
}
```
Function operator +=

```cpp
void Bag::operator += (const Bag& addend) {
    size_t addend_size = addend.used;
    if (used+addend_size > capacity)
        resize(used+addend_size);
    for (int n=0; n<addend_size; n++)
        data[n] = addend.data[n];
    used++;
}
}```
Assignment operator

```cpp
void Bag::operator = (const Bag& source)
{
    Item *new_data;
    if (capacity != source.capacity)
    {
        new_data = new Item[source.capacity];
        delete [] data;
        data = new_data;
        capacity = source.capacity;
    }
    used = source.used;
    for (int n=0; n < used; n++)
        data[n] = source.data[n];
}
```
Prescription for a dynamic class

• Some of the member variables are pointers
• Member functions allocate & release dynamic memory as needed
• Automatic value semantics must be overridden
• Class must have a destructor
Copy constructor uses with a dynamic class

- When one object is initialized as a copy of another:
  
  Bag y(x);
  Bag y=x;

- When an object is returned from a function

- When a value parameter is an object