Binary operator Overloading

- Binary operators such as `==`, `!=`, `<=`, `>=` can be redefined as non-member functions for a class
  - This leaves the re-defined operator(s) untouched when applied to other datatypes
- Redefining operators is called *operator overloading*
- Notice that the above operators act on *two* instances of a class, and do not mutate either of the instances
  - So correctly do not 'belong' to either of those instances

Overloading With Non-member Operators

- When we overload non-mutating binary operators, it is appropriate that they *are defined at the class level, not at the member function level*
  - This is achieved by:
    - Including the profile of the re-defined operator in the class `.h` file after the closing `};` of the member section
    - Including the implementation of the overloaded operator in the class `.cpp` file, but without the `<class_name>::` prefix.

Example of Non-member Overloaded Operator

```
// This code follows the last }; of the
// class member profiles in account.h

// Non member functions for account class

// Precondition: None
// Postcondition:  Returns true if the
//    balance of acc1 is the same as
//    acc2, false otherwise
bool operator == (const account& acc1,
                  const account& acc2);

// This code is included in the file
// account.cpp after the member function
// definitions

bool operator == (const account& acc1,
                  const account& acc2) {
    return (acc1.balance() == acc2.balance());
}
```

Overloading with Member Operators

- Operators such as `+=`, `-=` can also be overloaded
  - Since the left side of expressions involving such operators are mutated by the operator, the operation 'belongs' to the left instance, so it is appropriate that the overloaded operator is included in the member functions for the instance
  - This is achieved by:
    - Including the profile of the re-defined operator in the class `.h` file together with the other member functions
    - Including the implementation of the overloaded operator in the class `.cpp` file, including the `<class_name>::` prefix

Example of Member Overloaded Operator

```
// This code is included with the
// class member functions in account.h

// Precondition:  acc1 and acc2 are
//                instances of account
// Postcondition:  the balance of acc1 is
//                 increased by the
//                 balance of acc2
void operator += (const account& acc2);

// This code is included with the member
// function definitions in account.cpp

void account::operator += (const account& acc2)
{
    acct_balance += acc2.balance();
}
```

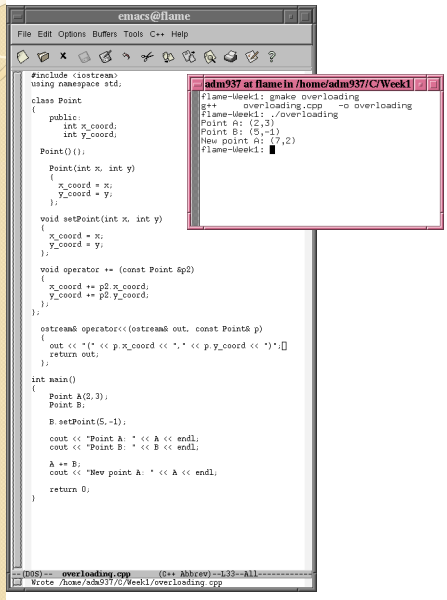
Comments

- Notice that, for *non-member* operators:
  - The profile is defined *after* the last `};` of the class member definitions
  - The signature of the function implementation *does not* include the class name
    - E.g. `bool operator ==`
- Notice that, for *member* operators:
  - The profile is defined with the class member functions
  - The signature of the function implementation includes the class name
    - E.g. `void account::operator +=`
- The parameters are defined using reference types, so no copying is required, increasing efficiency if the parameters are large objects
  - By adding the keyword `const` we ensure that the parameters cannot be mutated by the operator

Arithmetic Operators

- The arithmetic operators `+`, `-`, `*` and `/` can also be overloaded for a class
  - This can be used, for example, for objects that represent Cartesian points
  - The form of the overload definition is as previously shown
- The operators `<<` and `>>` can also be overloaded
  - For example to allow chaining of output involving complex data structures

Example



typedef

- Use of the `typedef` keyword provides for definition of a bag class that can be used for different type collections
  - This defines a user-defined *name* for the data type
    - So a single line of code change supports a type change
    - E.g. `typedef account value_type;` would be added at the top of the public section of the class definition, so that `value_type` *acts as an synonym for* `account` throughout the code
    - Changing the associated type in one place changes the code to implement for that type

typedef

```
main(){
    int balance;
    balance = 2;
}
```

Is the same as:

```
main(){
    typedef int Dollars;
    Dollars balance = 2;
}
```

typedef

```
typedef Account value_type;

value_type functionX();
value_type functionY();
value_type functionZ();
```

If you change the typedef to:

```
typedef Person value_type;
```

You change the return types of the functions as well => Extra flexibility.

Dynamic variables

- The statement `account* ptr;` creates a pointer variable that can store a pointer to an instance of class `account`
  - at this time no instance of `account` exists
- The statement `ptr = new account;` creates a new dynamic variable of class `account` and stores a pointer to it in `ptr`
  - This dynamically allocates heap space for the instance of `account`

Constructors and new

- The `new` operator works as expected with constructors
  - Let us assume the statements `account* ptrX;`
  - So `ptr1 = new account();` will use the default constructor while `ptr2 = new account(50);` will use the constructor that takes an argument
    - The instance can then be accessed using `ptr2 -> balance()`
- The `new` operator can be used to create an array of instances
  - So `ptr3 = new account[30];` allocates a dynamic array of 30 instances of `account`, each created using the default constructor
    - Items in the array are accessed as, e.g. `ptr3[5].deposit(40);`

Comments on the Heap

- The heap is not of infinite size
  - So a program that constantly allocates dynamic variables can run out of heap space
- When the heap runs out of space, the `new` operator fails
  - This is shown by a `bad_alloc` exception
    - In the absence of exception handling the program outputs an error message and halts
- Programmers should return memory to the heap when dynamic variables are no longer needed
  - This is achieved using the `delete` operator

The delete Operator

- The space allocated to a dynamic variable pointed to by `ptr1` is released using the statement
  `delete ptr1;`
- In the case that ptr3 points to a dynamic array, the space allocated to store the array is released using the statement
  `delete [] ptr3;`
  - There is no need to specify the size of the array

Destructors

- We have seen that the space allocated to an instance can be returned to free memory by the use of `delete`
  - But what if that instance stored pointers to dynamic variables as part of its member data?
- To ensure that memory is properly released, classes encapsulating dynamic variables must define a *destructor*
  - Like a constructor these have no return type
  - The destructor is always indicated by the symbol `~` followed by the class name
    - E.g. `~arr_bag();` would be the destructor for an implementation of `arr_bag` using a dynamic array
      - If the dynamic member array was declared using `value_type* data;` then the implementation of the destructor would define the single statement `delete [] data;`
- Destructors examples.

Container Classes

- Container classes define a data structure in which collections of data items can be stored (e.g. lists, queues, vectors, sets, etc.)
- At present we will learn how to construct these classes
  - Later we will see that many such classes are provided in the STL (Standard Template Library)
    - To this end we will use the same names and notations as in the STL
- We will emphasise creation of generic containers that can be used (with minimum change) to store many different data types

Linked Lists

- A *linked list* is a sequence of items connected by a link
  - If the items were thought of as balloons with the link to a balloon thought of as the cord tied to it, a linked list would be analogous to a chain of such balloons with each next cord emanating from the current balloon
- The links are implemented using pointers
- The items (balloons above) are implemented using objects called *nodes*
  - Each storing a content object (often called `data`) and a pointer to the next node (often called `next`)
  - Arrays vs. linked lists

Structure of a Node

- Let us use `node` as the name of the class
  - Each instance of node would have as member data an instance of the content object and a pointer to a node
    - i.e. the member data consists of an instance of `value_type` and a `node*`

```
class node
{
public:
    typedef <obj_type> value_type;
    …
private:
    value_type data;
    node* next;
};
```

Use of Linked Lists

- Linked lists are typically used as the internal data structure of a container class
  - E.g. to replace the array as the underlying data structure of the `bag` class
- The container class stores, as part of its member data, pointers to the `head` (and usually the `tail`) of the list
  - Any mutation of the list must ensure that these pointers are correctly maintained

The NULL Pointer

- When an instance of `node` is created, the `next` pointer has no target
  - The `NULL` pointer is defined in `<cstdlib>` and is used as the value of a pointer that does not point anywhere
    - It can be assigned in the usual way, e.g. `next = NULL;`
- For the container class, if there is no data (and therefore no node instances), then `head` and `tail` should be set to `NULL`
- The `next` member data of the tail node should be `NULL`
  - As appropriate for the last node

Constructor for node

- It should be possible to create a new instance of node using no, one, or two arguments
  - In the case of a single argument, this represents an instance of `value_type`
  - In the case of two arguments, an instance of `value_type` and a pointer to another `node` are provided

```
//Constructor
node(const value_type& initial_data =
        value_type(), node* initial_link = NULL)
{
    data = initial_data;
    next = initial_link;
}
```

- Note that the constructor for `value_type` is called if no argument is passed to the constructor
  - So the data item will be initialised as defined by that constructor

Mutator Member Functions

- Functions that allow a `node` instance to be changed are

```
void set_data(const value_type& new_data)
{
    data = new_data;
}
void set_link(node* new_link)
{
    next = new_link;
}
```

- To retrieve the current data

```
value_type data() const
{
    return data;
}
```

- To retrieve the pointer data

```
node* link()
{
    return next;
}
```

The NULL Pointer

- Programmers should take great care to *never* dereference a `NULL` pointer
  - This would be done using `*ptr` or `ptr->` when `ptr` has been set to `NULL`
- Dereferencing a `NULL` pointer does not cause a syntax error
  - Rather the program will attempt to interpret the `NULL` pointer as a valid address
    - Sometimes resulting in an address violation or core dump
    - Other times resulting in access to unintended parts of memory, causing difficult-to-trace errors
      - E.g. overwriting part of the OS resulting in errors *after the program has ceased execution*
- The rule is clear:

NEVER, NEVER, EVER DEREFERENCE A `NULL` POINTER