Basics of C++ (cont.)

Inheritance and abstraction
Documenting your code
Operator overloading
Typedef
Destructors

Read chapters 11, 12 and 13 of the textbook!

These slides will cover some of the contents, but the textbook is much more detailed.

Inheritance

- **Inheritance** is used when a new class adds new functionalities to another more general class. The syntax for inheritance is given by:

```
# include...
class Base {
   public:
    // member functions/variables
}

class Derived:public Base {
  public:
  // specialized member functions
}
```

- It increases code re-use, and reduces code duplication. Those two improve maintainability and consistency of the software.

DEMO

Abstraction

- **Abstraction** is used when you need to create several implementations for functions, depending on the type of specialization that you want:

```
# include…
class AbstractClassName{
   public:
    // at least one virtual member function
}

class Derived:public AbstractClassName{
  public:
  // specialized member functions
}
```

- Reduces the length of the code. Avoids the use of a parameter to determine the type of object, and switch-case statements to decide which method to call.

DEMO

Documenting your code

- What is expected from you in assignments?
  - Inline comments whenever necessary in the .cpp files

```
#include "account.h"
// Constructors
account::account()
{
    acct_balance = 0;
}
account::account(int initial_balance)
{
    if (initial_balance >= 0) // checks if initial value is OK
    {acct_balance = initial_balance;}
    else {cout << "Initialization value is negative!";}

}
void account::deposit(int amount)
{
    if (amount >= 0) // checks if deposit value is OK
    {acct_balance += amount;}
    else {cout << "Deposit value is negative!";}
}
```

Documenting your code

- What is expected from you in assignments?
  - Pre and post conditions

```
class account
{
    public:
     // Constructors
     account();
     account(int initial_balance);

    // Members that mutate data
     void deposit(int amount);
     // Pre conditions: Receives a positive integer value that will be added to the
     //                 variable acct_balance
     // Post conditions: If the integer is positive, the variable acct_balance will be
     //                  changed, otherwise an error message will be displayed

     // Members that query data but do not change it
     int balance() const;
     // Pre conditions: None
     // Post conditions: Returns the value of acct_balance
...
```

Operator overloading

- C++ allows operators such as == and + to be redefined, or overloaded.
- That is particularly important for data structures, as we need them to work with any type of object, i.e. both primitive types and user-defined ones.
- Suppose that my data structure uses the operator << (`cout`) to display its contents. If I am storing a primitive type, say `integer`, it will work, as C++ knows how to print an integer.
- But if I am using `account`, it will not work, as C++ does not know how to print an `account` object. We get a compilation error.
- If we overload the operator << for `account`, the code works just fine.

Operator overloading

- There are two types of operators: member and non-member. Let's start with the non-member ones, e.g. binary operators
- Binary operators such as ==, !=, <=, >= can be redefined as non-member functions for a class
- Notice that the above operators act on two instances of a class, and do not mutate either of the instances
- So they do not 'belong' to either of those instances
- In that case, they must defined at the class level, but not at the member function level

Operator overloading

- This is achieved by:
- Including the profile of the re-defined operator in the class `.h` file after the closing brackets " `};` " of the member section
- And including the implementation of the overloaded operator in the class `.cpp` file, but without the `<class_name>::` prefix

Operator overloading

```
// 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());
}
```

Operator overloading

- The second type of operator is the member ones. Those are operators that mutate the left-hand side variable, e.g. +=, and -=
- Since the contents of a variable of that class are changed, the operation 'belongs' to the class, so it is appropriate that the overloaded operator is included in the member functions section
- 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, with the `<class_name>::` prefix

Operator overloading

```
// 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();
}
```

DEMO

Operator overloading

- Notice that 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 in its implementation
- The arithmetic operators +, -, * and / can also be overloaded for a class
- The operators << and >> can also be overloaded, as shown in the last demo, for example to allow chaining of output involving complex data structures

Making your code more generic - typedef

- Suppose you have created a data structure, and it works for a given type, say `int`. Now, if you want to make it work for `string`, you will need to change every occurrence of `int` for `string` in your .h and .cpp files. This procedure is a potential source of error, especially if the two types are not inherently incompatible, e.g. `int` vs. `double`, or `float` vs. `double`.
- The `typedef` keyword allows a single line of code change to support a type change throughout the entire code.
- E.g. `typedef int 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 `int` throughout the code

Making your code more generic - typedef

```
main()
{

  int balance;
  balance = 2;
}
```

- Is the same as:

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

```
typedef int value_type;

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

- If you change the typedef to:

```
typedef double value_type;
```

- You change the return types of the functions as well.
- This eliminates the possibility of leaving unchanged types behind, which could cause errors.

Destructors

- Since the heap is not garbage collected, it is necessary for the user to monitor its state, and make sure that when dynamic variables go out of scope, their content is correctly deleted and all memory space used by them is released.
- Otherwise, a program that constantly allocates dynamic variables can run out of heap space
- This is achieved using the `delete` operator
- The space allocated to a dynamic variable pointed to by `ptr1` is released using the statement `delete ptr1;`
- Moreover, if a variable `ptr2` 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. C++ will find the end 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

DEMO

Introduction to Data Structures

Container classes
Linked lists
`NULL` pointer

Read chapter 16 of the textbook!

These slides will cover some of the contents, but the textbook is much more detailed.

Container classes

- Container classes define a data structure in which collections of data items can be stored (e.g. lists, queues, vectors, sets, etc.)
- In this course we will learn how to build these classes from scratch
- Later we will see that many such classes are provided in the C++ STL (Standard Template Library)
- To this end we will use the same names and notations as in the STL
- We will emphasise the 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 (i.e. the balloons above) are implemented using objects called `nodes`
- Each node stores a content object (often called data) and a pointer to the next node (often called next)
- Arrays vs. linked lists

What is a node?

- Let us use `node` as the name of the class that implements the nodes of the linked list
- As said before, each instance of `node` would have as member data an instance of the content object and a pointer to a node
- That is, the member data consists of an instance of `value_type`, so the data structure stores a generic type, and a pointer to a node `node*`

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

What about a linked list?

- Linked lists are typically used as the internal data structure of a container class, e.g. queue, stack, etc.
- The container class stores, as part of its member data, pointers to the head and 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 the `<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 stored, then head and tail should be set to `NULL`
- The member data `next` of the tail node should be `NULL`

The 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

Accessor and mutator member functions for node

- Functions that query the node instance:

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

node* get_next() const
{return next;}
```

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

```
void set_data(const value_type& new_data)
{data = new_data;}

void set_next(node* next_ptr)
{next = next_ptr;}
```

More about the NULL pointer

- You should take great care to never dereference a `NULL` pointer
- This happens when 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
- The rule is clear:

NEVER, NEVER, EVER DEREFERENCE A NULL POINTER

See you next week!