Intro to Classes

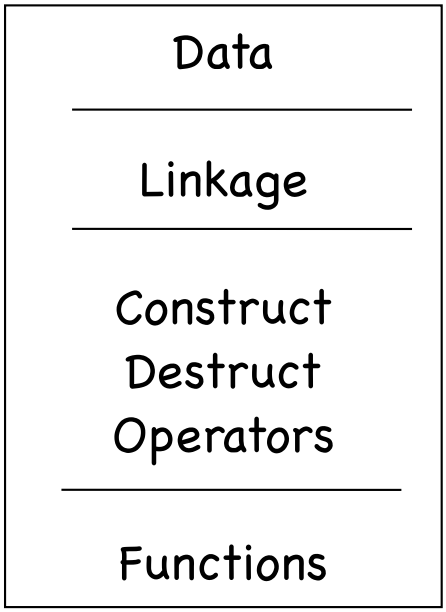
Classes / Objects

- Same idea as with `struct`
  - create a new type of combined variable
  - each instance (variable) has its own data
  - each variable can have linkage / structural information
    - think of it as dependance to other instances
  - everything works like with regular variables : allocation, assignments, function calls, pointers etc.
- Can have member functions
- special member functions: Constructor, Destructor, Operators

Objects

MEMBERS

- data variables
- links variable (optional)
- class operations
- Functions (operate on data)



Rectangles

```
class rectangle{
   double width;
   double length;
   rectangle();
   void SetWidth(double w);
   void GetWidth();
   void SetLength(double l);
   void GetLength();
   double ComputeArea();
}
```

```
rectagle r;
r.SetWidth(19.1);
r.SetLength(12.04);
double area;
area=r.ComputeArea();

rectangle *r2 = new rectangle;
r2->SetWidth(10.6);
(*r2).SetLength(17);
area= r2->ComputeArea();
*r2=r1; ?????
delete r2;
```

Access to members

- public: can be accessed directly from code outside the member functions
  - `rectangle x;`
  - `x.length =1;` //length is a public member
- private: only member functions can access these
  - think of these variables as "internal data"
  - cannot write `x.length` outside member functions, if "length" is private
  - member function can use `x.length` in their definition

Array of objects

- rectangle A[10];
- A[0].setwidth(15); A[0].setlength(20);
- double area = A[0].ComputeArea();

Constructor

- rectangle() //same name as the class
- A special function that is called automatically when class variable is declared
  - Initialization of the declared variable
  - Dynamically allocate members (that need to be allocated)
- No return type
- Possible to have more than one constructor for the class

Destructor

- ~rectangle() // "~" + classname
- A special function called automatically when objects are deleted
  - `rectangle *x = new rectangle;`
  - `....`
  - `delete x;` *// calls the destructor*
- Usually necessary when the object contains members dynamically allocated, for which we need to free the memory
- No return type

Class declaration style

- Often, the class declaration, functions are written in a different file "rectangle.h"
- Any program that uses the class would have to include the class definition
  - #include "rectangle.h"

Object Oriented Programming

- OOP = method of writing software centered on objects
  - opposite to procedural programming of functional programing
- Objects created from abstracted data types (like class definitions)
- Objects are variables for all practical purposes
  - function calls and returns
  - comparison, assignments
- Objects have methods that operates on them
- Facilitates big projects (like a game), simplifies modularity
- Reusability, hierarchy

`string` class

string definitions

- #include <string>
  - `string myname; myname="Virgil";`
  - `string yourname; cin >> yourname;`
  - `if (myname<yourname) {cout<< "myname is smaller than yours\n";}`
  - `string hisname ("William");`
  - `if(hisname>yourname) {cout<<"hisname is bigger than yours\n";}`

string operators

- >> extracts characters from stream and inserts them into the string
- << inserts the string into a stream
- = assignment
- +, += concatenation
- [] reference to character at index (like an array)

string member Functions

- `myname.length()`
- `myname.append()`
- `myname.assign()`
- `myname.at()`
- `myname.begin()`
- `myname.c_str()`
- `myname.clear()`
- `myname.compare()`
- `myname.copy()`
- `myname.empty()`
- `myname.end()`

string member Functions

- `myname.erase()`
- `myname.find()`
- `myname.insert()`
- `myname.replace()`
- `myname.resize()`
- `myname.size()`
- `myname.substr()`
- `myname.swap()`

More on C++ Classes

Object Members

- a member might be a struct or a (different)class object
- example: myqueue class has a member of type listobject

```
class myqueue{
  listobject* builder;
  listobject* consumer;
....
}
```

Static members

- regular members are "instance" members: each class object has its own
- static member is only one variable for all objects.
- static function can access only static members
  - but it can be called before defining any objects of the class
  - can be called as an instance function, or as a class function
  - `class::function()`
  - `object.function()`
- static member/functions exist before any object is declared

Friends of Class

- outside the class functions, private members are unaccessible
- bend the rule: define a function "friend" of the class
  - then it can access private members
  - without being a method/member function

```
class myqueue{
private:
  listobject* builder;
  listobject* consumer;
  ....
public:
void enqueue(void*);
....
friend int OTHERCLASS::MyFriendFunction (myqueue q);
}
```

Assignments of class objects

```
class rectangle{
  int length; int width;
  void* address;
}
rectangle r1;
r1.width=10;
r1.length=20;
rectangle r2;
r2=r1;  //r2 has now the same length, width
```

- r2 has also the same address (pointer)
  - a modification of *(r2.address) value implies *(r1.address) value is also modified, and vice-versa
  - when we want different pointers, but we want to copy the values at these address, we build a COPY CONSTRUCTOR

Copy constructors

- a user-defined constructor that performs "deep copy"
- `class object1;` *//write something on it*
- `class object2=object1;` *//also copies the pointers members*
- copy constructor: reallocates the pointers, copies the values
- requires parameter to be a reference

Operator Overloading

- redefine operators to work in a particular way for your class
- example: for rectangle class we might want the comparison operator ">" to compare the two areas
  - `if (r1>r2) {...}`
- `class rectangle{`
  - `int length; int width;`
  - `void* address;`
  - `bool operator>(rectangle);`
- `}`
- `bool rectangle::operator>(rectangle r2){`
  - `if(length*width>r2.length*r2.width) return true;`
  - `else return false;`
- `}`

Operator Overloading

- overload += to mean append for `myqueue class`
  - `q1.+=q2 should append queue q2 to queue q1`

```
void myqueue::operator+=(myqueue q2){
  //copy elements from second queue
  listobject* e = q2.consumer;
  while (e!=NULL){
    enqueue(e->address);
    e=e->next;
  }
}
```

Operator Overloading: overload +

- want to write q3=q1+q2;

```
myqueue myqueue::operator+(myqueue q2){
  //create a new queue - this is the result
  myqueue ret;
  //copy elements from first queue
  listobject *e = consumer;
  while (e!=NULL){....
    ret.enqueue(e->address); e=e->next;
  }
  //copy elements from second queue
  e = q2.consumer;
  while (e!=NULL){
    ret.enqueue(e->address);e=e->next;
  }
  return ret;
}
```

Operator Overloading: overload [ ]

- want q[i] to return the i-th element in the queue (0 is the consumer element)
  - this is like array index functionality

```
void* myqueue::operator[](int n){
  cout<<"\n---operator [] called\n";
  int i=0;
  listobject* e=consumer;
  while(e!=NULL){
    if (i==n) {return e->address;}
    i++; //counting
    e=e->next;
  }
  return NULL;
}
```

Operator overload: <<

```
class{...
  friend ostream &operator << (ostream &strm, myqueue);
}
```

```
ostream &operator << (ostream &strm, myqueue q){
  listobject* e=q.consumer;
  while(e!=NULL){
    strm<<e->address<<" ";
    e=e->next;
  }
  return strm;
}
```

`this` pointer

- inside a member function (method), this is a pointer to the "current" instance that is calling the method
- useful for returning a pointer to the current instance
  - otherwise we wouldn't know the address of the instance/object that just called the method