list_t (lab 7)

Monday, October 04, 2010
9:04 AM

- I jumped the gun with data_t — C++ version of what I did last time in it needed until lab 8, but "the big three" of C++ doesn't really apply to
- Back up a bit
- list_t: what we need for lab 7 (C++ version)
- for lab 7, use: (for data_t) list_t

class data_t — like a C struct
{
    public: — visible by all
    std::string name;
    int id;
};

int id;
};

- we'll go back to fuller version of data_t for lab 8.

- for lab 7, need list_t &, ctt I/O for data_t

- C++ I/O : using streams

#include <iostream> instead of
#include <stdio.h>

#include <string> for C++ strings
(instead of
char name[NAME_LEN];)

int main(int argc, char *argv[])
{

list_t list; // not a pointer
just a C++ object

data_t *data; // pointer to data_t object

free(data);

in C: data = malloc(sizeof(data_t))

call to memory "manager"
to allocate arbitrary memory

in C++: data = new data_t();

replaces malloc

you're asking object to create an instance of itself via its constructor

provide initialization arguments (optional)

in lab 7, we use default constructor

class data_t

delete data;

class data_t
{
  public:
  std::string name;
  int id;
};

replaces free

---

std::string name; //for I/O
int num; //for I/O

// instead of scanf, use cin stream
// see cppreference.com

works like scanf but also has state info
(e.g. are we at EOF)

// before, we had: while((count=scanf(...))==1)
{...}

while(!std::cin.eof()) {
  std::cin >> name >> num;

interpret this as:

std::cin.operator>>(std::string)

std::cin.operator>>(int)

same function name

different arguments

operator overloading

(std::cin.operator>>(std::string)).operator>>(int)

this function (operator >>) MUST return an istream object (what the cin is)

```
if (std::cin.good()) {
    data = new data_t;
    data->name = name;
    data->id = num;
    list.add((void *)data)
}
```

I would prefer to write: `data = new data_t(name, num);`

need a constructor for data_t

C++ does not like void *

C++ likes strong typechecking

```
}
:
```

rest is similar to ldb 2, i.e.,

```
list.reset()
data = (data_t *)list.get_data()
list.next_link();
list.not_end()
list.del()
```

— like a destructor

```
list.reset();
while (list.not_end()) {
    data = (data_t *) list.get_data();
    std::cout << data->name << " " << data->id << std::endl;
        ↳ like printf("%s %d\n", data->name, data->id);
    list.next_link();
}
```

```
    lst.next_item();
}
```

I'd rather type

```
std::cout << data;
```

```
class data_t
{   public:
// constructor (overloaded)
data_t(std::string iname="", int inum=-1)
{   name = iname;   id = inum; }
```

std::string.operator=(std::string)

OR

line continuator

```
data_t(std::string iname="", int inum=-1): \
    name(iname), \
    id(inum)
{ };
```

copy calling constructor

no-op constructor for data_t

// overloading ostream operator<<

accepts data_t, not *data_t

```
friend std::ostream& operator<<(const data_t& rhs)
```

return type

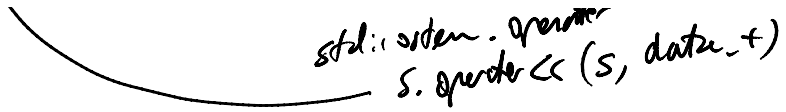
```
std::ostream& s,
{ s << rhs.name << "_" << rhs.id << std::endl;
  return s;
}
```

```
friend std::ostream& operator<<(std::ostream& s, data_t* rhs)
{   return (s << (*rhs)); }
```

interpret as

std::ostream.operator<<(s, data_t)

s.operator<<(s, data_t)




```
private:
    std::string name;
    int         id;
};
```