C++

Friday, October 01, 2010
9:02 AM

C++:

- entirely object-oriented
- that's the mindset: everything is an object

e.g.,

C:

printf("%d", x[i]); ≡ fprintf(stdout, "%d", x[i]);

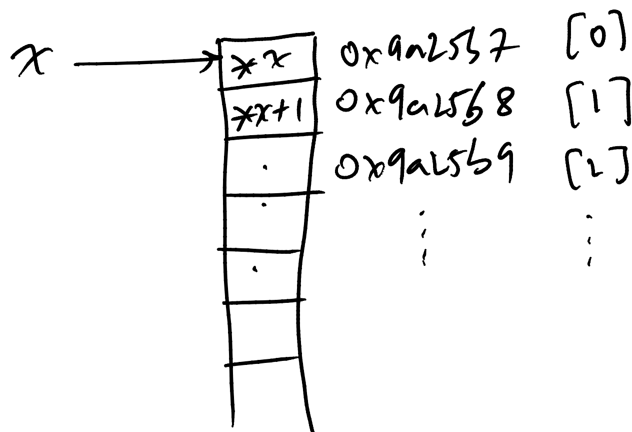
- prints an integer, the ith element of array x
- x[i] in C means "address of x + offset" (memory)

x[i] ≡ *x + i

*x: de-reference x (x is a pointer)

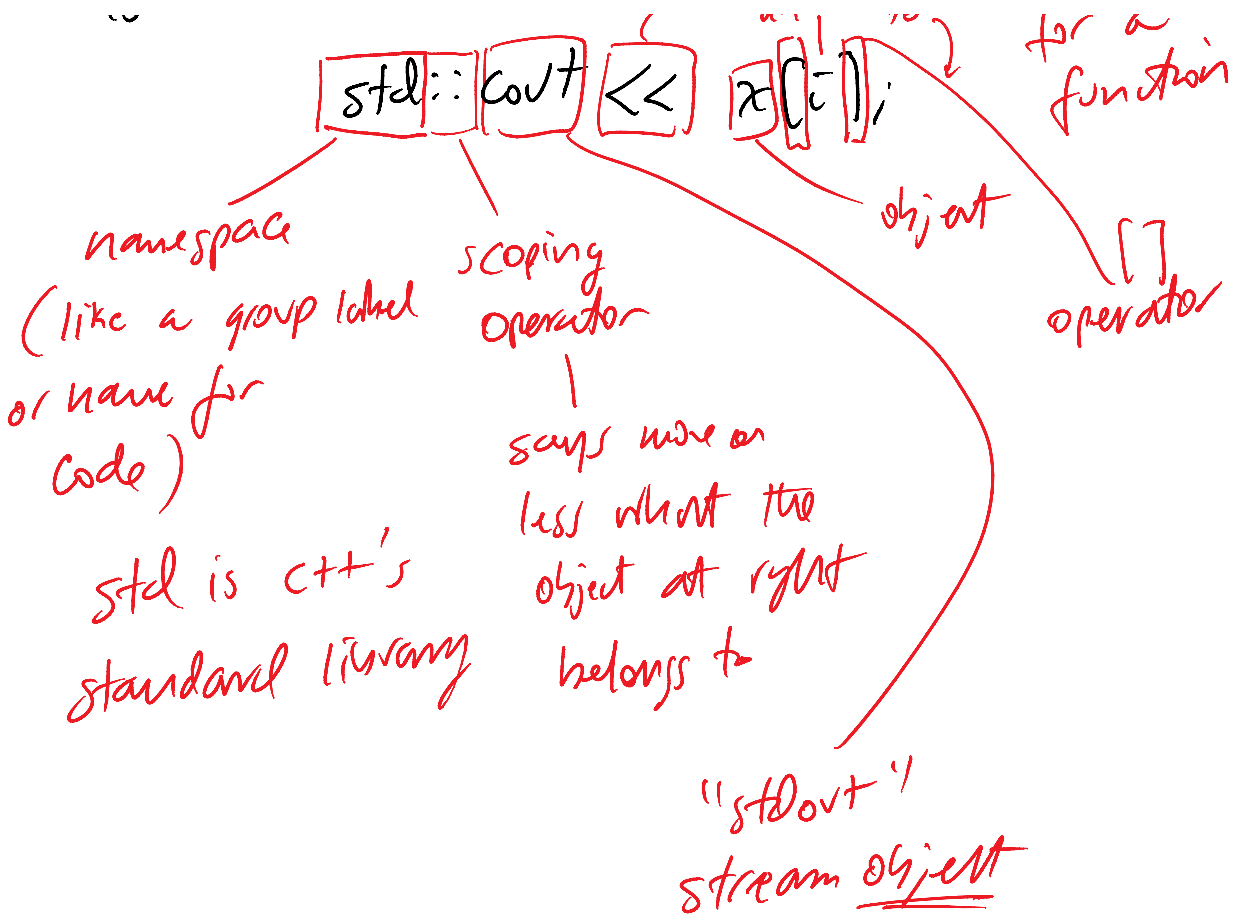
+ i: offset into memory

pointer arithmetic (can do stuff like *x++)



- in C++, fprintf(stdout, "%d", x[i]);

is:



- what's really happening:

std::cout. ostream<<(x[i])

cout's output function name

↑

a class (like a struct)

```
class cout
{
  __ ostream<<(int);
}
```

operator << ≡ a function

- what about [ ]? Also an operator that belongs to object x — x is probably an array or list (in general it's a container)

-

```
class list_t
{
    int operator[](int i)
        { return data[i]; }
}
```

by doing this, you are overriding available operator

- in C++

```
int x, y, z;

z = x + y;

z.operator=(x.operator+(y));
```

mytype x, y, z;

z = x + y; } you have to define

operator =

operator+

for mytype

Friday, October 01, 2010
9:24 AM

- Building your own classes:
  (object)

- For every object, you must provide:
  "the big three"

① copy constructor
  ↳ allocates & initializes
  (like most of our _init()
  (functions) — malloc
  — memset)

constructor given a copy of an object

② destructor
  - de-allocates, or frees memory

③ assignment operator
  operator=
  what does it mean to say

$x = y$ — if these are "heavy" objects (lots of data), may need a <u>deep</u> copy

Friday, October 01, 2010
9:30 AM

- example: before (lab 2),
  we had (in main.c)

```
typedef struct data_type
{
string ——— char name[NAME_LEN];
int ——— int id;
} data_t;
```

(5 lines of code — you had provide your own malloc calls, free calls, etc)

Now, in C++:

- replace

```
#include <stdio.h>
#include <string.h>
```

(crossed out)

```
#include <iostream>
#include <string>
```

— notice no .h

```
class data_t
{
    public: // things available to
               users of this class
               (classes other than data_t)

    // constructors (overloaded)
```

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

const (we promise not alter contents)

reference

```
// copy constructor
data_t (const data_t& rhs)
  { id = rhs.id; name = rhs.name; }
```

doing two things:

can call the constructor as

data_t();

data_t("bla", 10);

called with a data_t as argument

data_t a("bob", 10);
data_t b(a);

copy constructor

```
// destructor (here, default ok)
// ~data_t();
```

because I didn't allocate memory (didn't new)

C++ version of malloc

```
// assignment operator
```

gets called a = b; assignment op

```
const data_t&   operator=(const data_t& rhs)
{
   if ( this != &rhs) {
      name = rhs.name;
      id = rhs.id;
   }
}
```

this: pointer to current (this) object

same as this→name = rhs.name

rhs.name;

```
    }
    }      same as this->name = rhs.name
                      ≡ (*this).name = rhs.name;

friend std::ostream& operator<<(std::ostream& s, const data_t& rhs)
    { s << rhs.name << " " << rhs.id << std::endl;
      return s;
    }
    private: // stuff only this class can access
    std::string   name;
    int           id;
};  // ends class definition
```

- want to do the following:

```
data_t    *data;
          pointer to data_t
   ⋮
std::cout << data;  // print out
   ⋮                  pointer to data
     prints out
     0x9a2056

or
std::cout << *data;
```

std::cout.operator<<(data_t);

C++ barfs because
it doesn't know
how to print data_t

you have to override
(overload operator<<())