CS122 Algorithms and Data Structures

MW 11:00 am - 12:15 pm, MSEC 101

Instructor: Xiao Qin

Lecture 1: Introduction and Review

Acknowledgement

- Lecture slides are modified from those made available by Dr. William Spears with the University of Wyoming

- **Office Hours:** MW 1:00 pm-2:00 pm
- **Email:** xqin@cs.nmt.edu
- **Class Web Page:**

  http://www.cs.nmt.edu/~xqin/courses/cs122
- **Prerequisite:** CS111 with a grad of C or better

What You Will Learn…

- Fundamental techniques in the design and analysis of data structures
- Algorithm design and analysis techniques
- Apply concepts introduced in classes in the development of rather large programs

- **Textbook**

Adam Drozdek, "Data Structures and Algorithms in C++," by Brooks/Cole Thomson Learning, 2nd Edition, 2001, ISBN: 0-534-37597-9.

- **Additional Optional Reading**

Mark Allen Weiss, "Data Structures and Algorithm Analysis in C++," Addison-Wesley, 2nd Edition, 1999.

- **Reference books**. Any one of the following books is recommended:
  - *The C++ programming language* by Bjarne Stroustrup (3rd or Special edition)
  - *Absolute C++* by Walter Savitch
  - *C++: How to program* by Deitel & Deitel (4th edition).

Exams and Grading

Unannounced Quizzes	10%	Four quizzes
Mid-term	15%	Oct. 13, 2004
Final Exam	25%	During finals week
Written homework	25%	
Programs	25%	

Homework

- Programming Language: C++
- Complier: *gcc*, the GNU project C++ compiler.
- There is a 20% deduction for late homework. The deduction becomes 50% if the homework is two days late. No credit is given after three days.

Cheating

- Don't do it.
  - Discussing concepts, issues, and high-level ideas with your classmates is allowed. Copying from them is NOT!
  - Do not recycle code from the Internet
  - All code for programs must be written from scratch

Programming Style

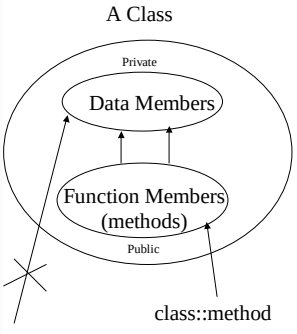
- Identify yourself
  - –Name, student ID, assignment number.
- Comment everything
  - –Logic of your program.
  - –Purpose of variables, data structures, functions, and classes.
- Naming style
  - –For example, use lowercase for variables and uppercase for constants.

Programming Style (cont.)

- Indentation
  - – 4 or 8 spaces of indentation
- Simplicity
  - – Make it easy for someone else to read
  - – Avoid multiple nesting of ifs.
  - – Functions must be reasonably short
- Be consistent
- Instructor Wiley's C++ Tips

http://wwwcsif.cs.ucdavis.edu/~wiley/standard.htm

Introduction to Classes



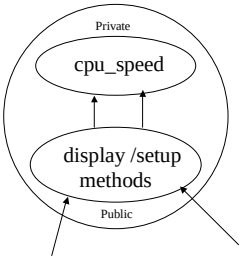
An "object" is an instantiation of a class.

Information Hiding:
Private members can only be accessed by methods in its class. Public members can be accessed by methods in any class.

Development issues:
What the program should do?
How it could be done?

Example: The "Laptop" Class

The laptop Class

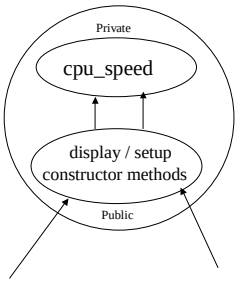


Laptop::setup Laptop::display

```
class Laptop
{
private:
        int cpu_speed;
public:
        int display ()
        { return cpu_speed; }
        void setup (int new_speed)
        { cpu_speed = x; }
};
```


Instantiating an Object

The Laptop Class



A constructor is a method that describes how an instance of a class is constructed.

Let's create two Laptops objects dell1 and dell2:

```
Laptop dell1; // cpu_speed = ?
Laptop dell2(2); // cpu_speed = 2
```


Constructors

The constructors must have the same name as the class.

```
class Laptop
{
private:
        int cpu_speed;
public:
        Laptop () // Zero-parameter constructor
        { cpu_speed = 0; }

        Laptop (int initial_speed) // One-parameter
        { cpu_speed = initial_speed; }

        int display ()
        { return cpu_speed; }

        void setup (int new_speed)
        { cpu_speed = new_speed; }
};
```


Default Parameters

Initial_speed becomes 0 if no parameter is provided.

```
class Laptop
{
private:
        int cpu_speed;
public:
        Laptop (int initial_speed = 0)
        { cpu_speed = initial_speed; }

        int display ()
        { return cpu_speed; }

        void setup (int new_speed)
        { cpu_speed = new_speed; }
};
```

Create two Laptop objects: Dell1 and Dell2.

```
#include <stdio.h>
class Laptop
{
private:
   int cpu_speed;
public:
                Laptop (int initial_speed = 0)
                { cpu_speed = initial_speed; }
                int display ()
                { return cpu_speed; }
                void setup (int new_speed)
                { cpu_speed = new_speed; }
};

int main()
{
                Laptop Dell1;
                Laptop Dell2(2);

                printf("Dell1 has a %d GHz cpu.\n", Dell1.display());

                Dell1.setup(3);
                printf("Dell1 has a %d  GHz cpu.\n",Dell1.display());
                printf("Dell2 has a %d  GHz cpu.\n",Dell2.display());
}
```

dell1.cpu_speed = 0
dell2.cpu_speed = 2
dell1.cpu_speed = 3

Pointers

- A pointer is an address in memory.
- The & operator (ampersand) changes a thing into a pointer (the address of the thing).
- The * operator changes a pointer into a thing.

key = 5 → Integer key = 5

x_ptr = &x → Integer key = 5

*x_ptr = 10 → Integer x = 10

```
#include <stdio.h>

int main()
{
                int key;
                int *key_ptr;
                x = 5;
                x_ptr = &x;
                *x_ptr = 10;
                printf("x =%d");
                return(0);
}
```


Pointers to Objects

Dell is a pointer to a Laptop. To allocate space, a "new" operator must be used.
To reclaim the space use the "delete" operator. Use the "→" operator to access the methods.

```
#include <stdio.h>
class Laptop
{
private:
   int cpu_speed;
public:
                Laptop (int initial_speed = 0)
                { cpu_speed = initial_speed; }
                int display ()
                { return cpu_speed; }
                void setup (int new_speed)
                { cpu_speed = new_speed; }
};

int main()
{
                Laptop *dell = NULL;
                dell = new Laptop(2);

                printf("Dell has a %d GHz CPU.\n", Dell.display());
                Dell.setup(3);
                printf("Dell has a %d GHz CPU.\n", Dell.display());
                delete dell;
                return(0);
}
```

cpu_speed = 2 GHz
cpu_speed = 3 GHz

Parameter Passing

Call by Value

The normal technique is call by value, where the actual argument is copied into the formal parameter. Changing the formal parameter does not change the actual argument.

formal parameter — actual argument

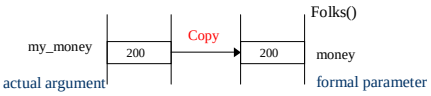
```
#include <stdio.h>
int folks (int money)
{
                money = 100;
                return money;
}

int main()
{
                int my_money = 200;
                printf("Folks say %d \n", folks(my_money));
                printf("I still need  %d \n", my_money);
                return 0;
}
```

my_money doesn't change value.
Need to convince your folks to give you more.

Parameter Passing

Call by Value

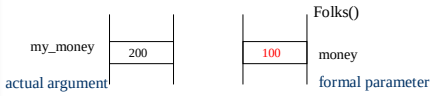
```
#include <stdio.h>
int folks (int money)
{
        money = 100;
        return money;
}

int main()
{
        int my_money = 200;
        printf("Folks say %d \n", folks(my_money));
        printf("I still need  %d \n", my_money);
        return 0;
}
```



Parameter Passing

Call by Value

```
#include <stdio.h>
int folks (int money)
{
        money = 100;
        return money;
}

int main()
{
        int my_money = 200;
        printf("Folks say %d \n", folks(my_money));
        printf("I still need  %d \n", my_money);
        return 0;
}
```



Parameter Passing (cont.)

Call by Reference

We are passing the address of the argument rather than a copy.

Any change to the parameter changes the argument.

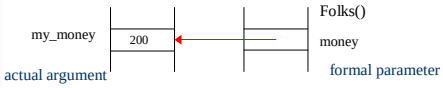
```
#include <stdio.h>

int folks (int &money)
{
        money = 100;
        return money;
}

int main()
{
        int my_money = 200;
        printf("Folks say %d \n", folks(my_money));
        printf("OK, I'll take %d \n", my_money);
        return 0;
}
```

my_money is now 100. Folks send you 100.

Parameter Passing (cont.)

Call by Reference

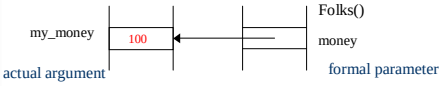
```
#include <stdio.h>

int folks (int &money)
{
        money = 100;
        return money;
}

int main()
{
        int my_money = 200;
        printf("Folks say %d \n", folks(my_money));
        printf("OK, I'll take %d \n", my_money);
        return 0;
}
```



Parameter Passing (cont.)

Call by Reference

```
#include <stdio.h>

int folks (int &money)
{
        money = 100;
        return money;
}

int main()
{
        int my_money = 200;
        printf("Folks say %d \n", folks(my_money));
        printf("OK, I'll take %d \n", my_money);
        return 0;
}
```



Parameter Passing (cont.)

Call by Constant Reference

We are passing the address of argument, not a copy.

We are not allowed to change the parameter. This particular example generates a compiler error.

```
#include <stdio.h>

int folks (const int &money)
{
        money = 100; // Compiler error!!
        return money;
}

int main()
{
        int my_money = 200;
        printf("Folks say %d \n", folks(my_money));
        printf("Nope, I need %d \n", my_money);
        return 0;
}
```

Folks aren't allowed to change the amount of money you need!

Return Passing

- The standard is return by value. This returns a copy (e.g., x = folks(y)).
- Return by reference passes back an address, not a copy. The returned address can be used on the left-hand-side of a standard assignment statement (e.g., folks(y) = x). Generally avoided.
- Return by constant reference also passes back an address. The returned address can not be used on the left-hand-side of a standard assignment statement.

Return by Constant Reference

- If an object is small, we can return it by value. However, if the object is large, we can avoid the copy by returning its constant reference.

Return by Constant Reference (cont.)

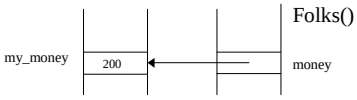
This example shows a return by constant reference. The function folks(y) now returns an address and avoids the overhead of a copy.

```
#include <stdio.h>
const int &folks (int &money)
{
        money = 100;
        return money;
}

int main()
{
        int my_money = 200;
        int *my_money_ptr;
        my_money_ptr = folks(my_money);
        printf("Folks says %d \n", *my_money_ptr);
        return 0;
}
```

Folks point out that they left 100 under your mattress. No need to send it.

Return by Constant Reference (cont.)

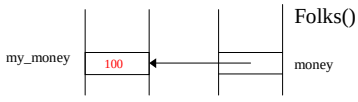
```
#include <stdio.h>
const int &folks (int &money)
{
        money = 100;
        return money;
}

int main()
{
        int my_money = 200;
        int *my_money_ptr;
        my_money_ptr = folks(my_money);
        printf("Folks says %d \n", *my_money_ptr);
}
```



actual argument formal parameter

Return by Constant Reference (cont.)

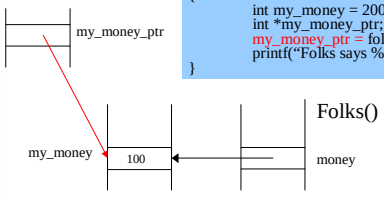
```
#include <stdio.h>
const int &folks (int &money)
{
        money = 100;
        return money;
}

int main()
{
        int my_money = 200;
        int *my_money_ptr;
        my_money_ptr = folks(my_money);
        printf("Folks says %d \n", *my_money_ptr);
}
```



actual argument formal parameter

Return by Constant Reference (cont.)

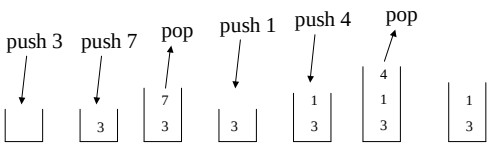
```
#include <stdio.h>
const int &folks (int &money)
{
        money = 100;
        return money;
}

int main()
{
        int my_money = 200;
        int *my_money_ptr;
        my_money_ptr = folks(my_money);
        printf("Folks says %d \n", *my_money_ptr);
}
```



Structs versus Classes

- Let us implement a stack, using both structs and classes.
- Classes implement information hiding better than structs.
- A stack holds items, and that the next item that can be removed is the last item that was added to the stack.

A series of operations executed on a stack



Stacks Using Structs

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

const int SIZE = 100;

struct stack {
        int count;
        int data[SIZE];
};


void init (struct stack &s) {
        s.count = 0;
}

void push (struct stack &s, const int item) {
        s.data[s.count] = item;
        s.count++;
}

int pop (struct stack &s) {
        s.count--;
        return(s.data[s.count]);
}
```

```
void main()
{
    struct stack s;
    init(s);

    push(s, 1);
    push(s, 2);
    push(s, 3);

    printf("Expect a 3 %d \n", pop(s));
    printf("Expect a 2 %d \n", pop(s));
    printf("Expect a 1 %d \n", pop(s));

    s.count = 10; /* This is a bug */
}
```

Note that the data and count fields are accessible to anyone.

Stacks Using Classes

```
#include <stdlib.h>
#include <iostream.h>

const int SIZE = 100;

class stack {
private:
        int count;
        int data[SIZE];
public:
        void init (void) {
                count = 0;
        }
        void push (const int item) {
                data[count] = item;
                count++;
        }
        int pop (void) {
                count--;
                return(data[count]);
        }
};
```

```
void main()
{
    stack s;
    s.init();

    s.push(1);
    s.push(2);
    s.push(3);

    printf("Expect a 3 %d \n", s.pop(s));
    printf("Expect a 2 %d \n", s.pop(s));
    printf("Expect a 1 %d \n", s.pop(s));
}
```

Nobody can directly access the data and count fields.

More Types of Constructors

- Copy Constructor: required to construct a new object, initialized to a copy of the same type of object (e.g., Laptop Dell2(Dell1)).
- Operator= Constructor: called when "=" is applied to two objects after they have both been previously constructed (e.g., Dell2 = Dell1).
- Destructor: called when the object goes out of scope or is subjected to a delete.

More on Constructors…

The destructor.

```
~Laptop ()
{
        printf("Calling the Laptop destructor\n");
}
```

The copy constructor, e.g., Laptop Dell2(Dell1);

```
Laptop (const Laptop &source_laptop)
{
        cpu_speed = source_laptop.cpu_speed;
}
```

The assignment constructor, e.g., Dell2 = Dell1;

```
const Laptop &operator= (const Laptop &source_laptop) {
        if (this != &source_laptop) // Standard alias test
                cpu_speed = source.cpu_speed;
        return *this;
}
```


The Laptop Class

Copy constructor →

Assignment constructor →

```
#include <stdio.h>

int main()
{
    Laptop Dell1;

    Dell1.setup(3);
    printf("cpu is %d\n", dell1.display());

    Laptop Dell2(5);
    printf("cpu is %d\n", dell2.display());

    Laptop Dell3(Dell2);
    printf("cpu is %d\n", dell3.display());


    Laptop Dell4;
    Dell4 = Dell1;
    printf("cpu is %d\n", dell4.display());

    return(0);
}
```


Templates

- A function or class template is not an actual function or class, but instead is a pattern for what could become a function or class.
- We have a template argument Obj that can be replaced by any type to generate a function or class

The “Computer” Template

- The “Computer” template is like the Laptop class, but works for any type Obj, provided that Obj has a zero-parameter constructor, a copy constructor, and an assignment constructor.
- Obj will be passed by constant reference, to avoid expensive copying.
- “Computer” is not a class, it is a template. Computer<int> and Computer<double> are actual classes.

Complete Computer Template

```
template <class Obj>
class Computer
{
private:
   Obj cpu_speed;

public:
   explicit Auto (const Obj &initial_value = Obj())
    { cpu_speed = initial_value; }

   ~computer () {
      printf(““The destructor\n”); }

   Computer (const Computer &rhs) { //copy constructor
      cpu_speed = rhs.cpu_speed_size; }

   // assignment constructor
   const Computer &operator= (const Computer &rhs) {
         if (this != &rhs) // Standard alias test
               cpu_speed = rhs.cpu_speed;
      return *this;}

   const Obj &display() const { return cpu_speed_size; }
   void setup (const Obj &x){ cpu_speed = x; }

};
```

```
#include <stdio.h>

void main()
{

    Computer<int> Dell1;
    printf(“Dell1 %d“, Dell1.display());

    Dell11.setup(3);
    printf(“Dell1 %d“, Dell1.display());


  Computer<double> HP1;
  printf(“HP1 %d“, HP1.display());

    HP1.setup(2.8);
    printf(“HP1 %d“, HP1.display());
}
```