C++ Programming

Structures, Classes, Linked Lists

M1 Math

Michail Lampis
michail.lampis@dauphine.fr

Structures

- Structures are a general way to store more complicated objects in C/C++
- General syntax:

```
struct generalName {
    declare member1;
    declare member2;...
};
struct generalName obj1, obj2;
```

An example

- C does not have a type for complex numbers
- Recall: a complex number is z = a + ib

```
struct complex {
    double a;
    double b;
};
struct complex z1,z2;
```

Struct = a new type

- The idea here is that when we define a structure we define a new type of object
- Hence, we can define variables that have this type
- At declaration, enough memory is allocated to store all member properties of the structure object

Accessing inside members

- Data stored inside a struct can be accessed using the . operator

```
struct Complex z1;
z1.a = 7.2;
z1.b = 5;
...
```

The = operator

- The = operator is automatically defined for all structures

```
struct Complex z1,z2;
z1.a = 7.2;
z1.b = 5.3;
z2 = z1; //This is OK, will copy value-by-value
```

The = operator

- **Be careful!**
  - The meaning of the = operator when applied to structures/objects is different in C++ and Java
  - Recall: in Java = copies a **reference** when dealing with objects
  - In C++, = copies field-by-field
  - This may or may not be what you want...

Hidden =s

- Consider the following program

```
struct Complex add(struct Complex z1, struct Complex a2)
{
    struct Complex res;
    res.a = z1.a + z2.a;
    res.b = z1.b + z2.b;
    return res;
}
```

Hidden =s

- The previous program is correct
- However, it performs a number of = operations
  - For each parameter, a new copy is allocated and data copied field-by-field
  - For the return value, a new copy is allocated and res is copied to it
  - Note: This is desirable, since res is deleted when the function terminates!

Performance

- The fact that so many copies are generally performed means that performance degrades when passing/returning structs
- Generally: we **NEVER** pass/return a struct
  - We prefer to pass/return a pointer/reference to a struct
  - This is better because only a pointer (~8bytes) must be copied, independent of the complexity of the struct

Passing pointers

- Consider the following program

```
struct Complex* add(struct Complex* z1, struct Complex* a2)
{
    struct Complex res;
    res.a = (*z1).a + (*z2).a;
    res.b = (*z1).b + (*z2).b;
    return &res;
}
```

Wait a minute

- The previous program contains a serious mistake...
- What is the problem with the previous program?

Pointers to struct

- The mistake was the res will be deleted, so its address should not be returned...
- But the syntax is correct:
  - (*z1) is an expression that dereferences the pointer z1
  - Its type is struct Complex
  - Therefore, we can apply . to it.

Fixed (?)

- Consider the following program

```
struct Complex add(struct Complex z1, struct Complex a2)
{
    struct Complex *res = new struct Complex;
    (*res).a = (*z1).a + (*z2).a;
    (*res).b = (*z1).b + (*z2).b;
    return res;
}
```

- This is correct, but someone has to eventually delete the new struct

Pointers to structures

- Because pointer are used **heavily** when dealing with structures, we avoid the (*p).field notation
- Instead we can use the - > operator

```
struct Complex *z1 = new struct Complex;

(*z1).a = 2.2;

z1->a = 2.2; //These two are equivalent!
```

Declaring a structure

- The declaration struct Complex { … }; describes the general form of the objects of type Complex, that is, their **class**.
- C allows variables to be declared in the same line:

  struct Complex {double a; double b;} z1,z2;
- C also allows anonymous structs!

  struct {double a; double b;} z3,z4;
- Careful! z1,z3 don't have the same type (z1=z3 fails)

Inside a structure

- We can declare as many fields as we like, using standard conventions for variable names.
- Members can be arrays (of predetermined size)
  - The = operator will also copy array fields!
- Members can be structures

```
struct C2 { struct Complex z; } zz;
zz.z.a = 5.2;
```

- Members can be pointers

Structure inside a structure

- Consider the following declaration

```
struct myStruct {
    int a;
    struct myStruct s;
} obj1;
```

- How much memory does such an object need?
  - Infinite!!!

Structure inside a structure

- Thankfully, the previous example does not compile
  - MyStruct has not been defined yet!
- But this does:

```
struct myStruct {
    int a;
    struct myStruct *s;
} obj1;
```

- This is strange, but is used **A LOT!**

A linked list of structures

- Consider the following declaration

```
struct Node {
    int data;
    struct Node * next;
};
```

- Each object contains a number and a pointer to the next object
- We can build a list of ints!
- Its size is unlimited!

A simple linked list

```
struct Node *cur,*prev;
cur = prev = NULL;
for(int i=0; i<20; i++){
    cur = new struct Node;
    cur - > data = i;
    cur - > next = prev;
    prev = cur;
}
```

A simple linked list

- At the end of the previous program
  - What is cur pointing to?
  - What is cur->data?
  - What is cur->next?
  - What is cur->next->next->next->data?
- NULL (or 0) is a special value that signifies that a pointer points to NOTHING
  - NULL ->data is a run-time error (seg fault)

Exercise

- Write a simple loop that prints all the numbers stored in a linked list
- Given: a pointer to the first element

Solution

```
void print_list(struct Node *head)
{
    while(head){
        cout << head->data;
        head = head->next;
    }
}
```

Classes

- A class is a more sophisticated version of a struct
- Allows us to define “objects”
  - Initially, C++ was called “C with objects”
- Defining a class uses similar syntax and has a similar logic to defining a struct

Example

```
class Complex {
    double a;
    double b;
};
class Complex z1,z2;
```

What's the difference?

- Let's start with some easy differences:
  - In C++ you don't have to use the class keyword when defining instances of an object (i.e. variables). You only need it to define the general class;
  - Complex z1; //this would be correct
  - In fact, this is also true for struct, in C++. However, it is not true in C.
  - Using the struct/class keyword is allowed in C++

Access restrictions

- A second easy difference is that classes protect inside data from being accessed.

```
class Complex {
    double a;
    double b;
};
Complex z1,z2;
z1.a = 5.2; //Error!
```

Public and Private

- The fields of a class are divided into public and private
- By default, all fields are private
- We can specify the public part using the public: label

```
class Complex {
    public:
    double a;
    double b;
};
Complex z1,z2;
z1.a = 5.2; //OK
```

Methods

- The most important difference is that classes may also contain methods, that is, functions which are applied on the object.
- By default the methods of an object can access all fields of the class, including private fields

Method example

```
class Complex {
    double a,b;
    public:
    double abs();
};
Complex z1; …
cout << z1.abs( ); //OK
```

Methods

- You can think of methods as fields of a class, which happen to be functions
- Methods can be accessed as any other field
  - With the . operator
  - With the - > operator (if using a pointer)
- Can a structure contain a function field?
  - In C?
  - In C++?

Struct vs Class

- Consider the following C code

```
struct Complex {
        double a;
        double b;
        double (*abs)();
    } z1;
…
z1.abs( );
```

Struct vs Class

- The code of the previous page is valid!
- Each item of type (struct) Complex contains a function (abs) represented by a pointer
- Before calling it we need to set this function pointer to something
- But if we do, it works in the usual C++/Java way
- So why invent a new language, if this worked in C?

Class = Data + Methods

- Answer: in the previous example all instances of the Complex structure contain a function (pointer) abs
- But this function could be different for two different instances!
- In a class we define the functions as part of the general pattern of the class
- Think of it as setting abs to the same function each time an instance is declared

Defining methods

- Inline

```
class Complex {
    double a,b;
    public:
    double abs() { return a*a+b*b; };
};
```

- This is usually only done for very simple functions

Defining methods

```
class Complex {
    double a,b;
    public:
    double abs(); //only declaration
};
…
double Complex::abs( )
{ return a*a + b*b; }
```

Objects in a Program

- Notice that we can completely separate the definition of an object from the implementation of the functions.
- This is intentional!
- Typically C++ programs are written as follows:
  - A header file (.h) contains the **definition** of the relevant class
  - An implementation file (.cpp) contains the implementation
  - Any program that needs to use a class needs to include only the header file
  - The .cpp files can be compiled separately

A tiny project

- Header file: complex.h

```
//This file defines the Complex class
#ifndef COMPLEX_H
#define COMPLEX_H
class Complex {
        double a;
        double b;
        public:
        void set(double,double);
        double abs();
};
#endif
```

A tiny project

- Implementation file

```
#include "complex.h"
void Complex::set(double newa,double newb)
{
        a = newa;
        b = newb;
}
double Complex::abs()
{
        return a*a+b*b;
}
```

A tiny project

- A program that uses the Complex class

```
#include <iostream>
#include "complex.h"
using namespace std;
int main()
{
    Complex z1;
    double a,b;
    cout << "Enter a,b" << endl;
    cin >> a >> b;
    z1.set(a,b);
    cout << z1.abs() << endl;
}
```

More about methods

- As we mentioned methods are functions associated with an object
- When a method is called, it is **applied** on a specific object instance

  z1.abs( );
- A method is allowed to refer to the private fields of a class
  - This means → we are referring to the private fields of **this particular instance**.

More about methods

- A methods can also refer to the (whole of the) instance on which it was called
- For this we can use the **this** keyword

```
void Complex::set(double newa,double newb)
{
        this - > a = newa;
        this - > b = newb;
}
```

The this keyword

- The keyword this returns a **pointer** to the current object
- If there is no ambiguity, writing this->a and a are equivalent
  - But there may be ambiguity, since we are allowed to reuse a as the name of a parameter or local variable
- Useful: when we want to return a ref to the current object, or call other methods on it.