Outline

1. Introduction: C++

2. Containers

3. Classes
Outline

1. Introduction: C++

2. Containers

3. Classes
Variable types

Besides the types we have learned from C, we have a new type: bool (Boolean). Example:

```c
bool test=false;
```

Recall:
- short, int, long
- float, double, long double
- char
Object: region of memory that has a type.

double salary, wage;
int month, day, year;
std::string address;
Sales_item curr_book;

Note that the last type is defined by a library (std) and the last variable is a structure.
Defining Constants

- Sometimes we want to define variables that cannot be modified during the execution time. We declare them as `const`:
  ```c
  const int numsize=600;
  ```
- We have to initialize any constant variable when it is defined.
- Constant variables are local to the file in which it is defined.
- To make the constant variable accessible throughout the program we use `extern` declaration.
The usual C++ library of input/output tasks is iostream. It let us to write/read to/from a stream. (screen or files).

The most common objects are:
- cout: ostream object that writes to the standard output.
- cin: istream object that reads the standard output.
- >>: read input from the istream object.
- <<: write output to an ostream object.
Example

```cpp
#include <iostream>
using namespace std;

int main ()
{
    int i;
    cout << "Please enter an integer value: ";
    cin >> i;
    cout << "The value you entered is " << i;
    cout << " and its double is " << i*2 << "\n";
    return 0;
}
```
Alternative name of an object. Example:

```c
int ival=124;
int &refVal=ival;
```

All the operations on a reference are operations on the underlying object. Example:

```c
refVal=+2;
int ii=refVal;
```

adds 2 to ival and assigns the new value of ival to ii.

The references can refer to a `const` object.

The references are important as functions arguments.
unsigned long x = 4;

void func1(unsigned long& val) {
    val = 5;
}
func1(x);

//or:

void func2(unsigned long* val) {
    *val = 5;
}
func2(&x);

Which one is better: Pass by reference vs. Pass by value?
What is the difference between these two codes:

- `int ival=1024, ival2=2048;`  
  `int *pi=&ival, *pi2=&ival2;`  
  `pi=pi2;`

- `int &ri=ival, &ri2=ival2;`  
  `ri=ri2;`
Recall that the C/C++ functions receive their arguments by value. Also it is possible to use pointers as arguments.

We pass a `const` type variables (the function will make a copy that can’t be changed within the procedure)

Also we can use references:

```c
void swap(int &v1, int &v2){
    int tmp=v2;
    v2=v1;
    v1=tmp;
}
```

In this case we call the function like `swap(x,y)`.

Important: it is safer to use references than pointers.
For large-sized objects it is expensive to copy an argument of a function. Then we can use a reference to a constant variable to solve this problem.

```cpp
bool isShorter(const string &s1, const string &s2){
    return s1.size() < s2.size();
}
```
STL: Standard Template Library. C++ library of container classes, algorithms and iterators.

Container: object containing objects. (C++ template)

Most important containers:
- Vector
- String
- Bitset

In this course we are interested in quantitative applications, then Vector will be our first container.
**Vector**

- Collection of objects of a single type, each object has an associated integer index.
- You have to include an appropriate header:
  ```cpp
  #include <vector>
  using namespace std;
  ```
- The Vector class is a class Template (we’ll see the definition later). It can store different object types: strings, ints, doubles, classes.
Definitions:

- `vector<T> name;` // vector that holds objects of type T
- `vector<T> v2(v1);` // v2 is a copy of v1
- `vector<T> v3(n,i);` // v3 has n elements with value i
- `vector<T> v4(n);` // v4 has n values with an initialized value.

Examples:

- `vector<int> ivec1;`
- `vector<int> ivec2(10,-1);`
- `vector<string> svec(10,"HI");`

Remark: Usually is more efficient to define an empty vector and add elements to it.
Operations on vectors

- `v.empty()`: returns true if v is empty.
- `v.size()`: returns the number of elements in v.
- `v.push_back(t)`: adds element with value t to end of v.
- `v[n]`: returns element at position n in v.
- `v1=v2`: replaces elements in v1 by a copy of elements in v2.
- `v1==v2`: returns true if v1 and v2 are equal.
- `!=, <, <=, >, >=`: normal meanings (componentwise).
Remarks:

- All vectors start at 0.
- An element must exist in order to subscript it. Elements are not added when we assign through a subscript.
Iterators

- We can use subscripts to examine a vector, but we can use an iterator defined by the library.
- Advantage: all the containers have iterators, but not all the containers support indexing.
- Syntax (example):
  ```cpp
  vector<int>::iterator iter;
  ```
- Each container class defines its own iterator type that can be used to access elements in the container.
Iterators

- Each container has a `begin` operator indicating an iterator to the first element. Example:
  
  ```cpp
  vector<int>::iterator iter=ivec.begin();
  ```
- The containers also have an `end` operator pointing to the place next to the last element of the container.
- An iterator uses the operator `*` to access the element to which it refers.
  ```cpp
  *iter=2000;
  iter++;
  *iter=2001;
  ```
Example

```cpp
for(vector<int>::iterator iter=ivec.begin();
    iter!=ivec.end();iter++)
    *iter=0;
```

- If we don’t want to change the values of a vector, we can use a `const_iterator` (it is defined for any container):

  ```cpp
  for(vector<int>::const_iterator iter=ivec.begin();
      iter!=ivec.end();iter++)
      cout<< *iter << endl;
  ```

- The usual arithmetic on pointers apply to iterators.
Exercise (Bond Pricing)

Assume that you want to compute the price of a bond ($B_0$) with Face Value=1 and coupon payments at discrete times $t_1, \ldots, t_N$. Then this price becomes:

$$B_0 = \sum_{i=1}^{N} C_{t_i} e^{-rt_i}$$

if a continuously compounded interest rate $r$ is assumed. Solve this problem using vector containers.
Yield to maturity

Given the market price $B_0$, we want to find the interest rate $y$ (Yield to Maturity) such that:

$$B_0 = \sum_{i=1}^{N} C_t e^{-yt_i}$$

Compute this rate using Bisection method. Assume that your cash flow produces a unique YTM.
Compute the duration $D$ and Macaulay $D_M$ as follows:

\[
D = \frac{1}{B_0} \sum_{i=1}^{N} t_i C_t e^{-rt_i}
\]

\[
D_M = \frac{1}{B_0} \sum_{i=1}^{N} t_i C_i e^{-yt_i}
\]

and also the convexity $Cv$:

\[
Cv = \frac{1}{B_0} \sum_{i=1}^{N} t_i^2 C_i e^{-yt_i}
\]
Outline

1. Introduction: C++
2. Containers
3. Classes
We can distinguish two components on each class:

- **Implementation**: private members that define the data and any function that is not intended for general use.
- **Interface**: all the functions defined on the public part of the class.

Syntax:

```cpp
class name_class{
public:
    //data members and member functions

private:
    //data members and member functions
};
```
We cannot initialize the members of a class as part of their definition. The initialization process is done using Constructors.

The public section of the class defines members that can be accessed by any part of the program.

The private section contains members that can’t be manipulated by the final user.
We will create a C++ class containing information about European options. (Duffy 2004)

- The class will contain the following member data:
  - Stock price
  - Volatility
  - Strike price
  - Time to expiry
  - Risk-free interest rate
  - Option market price.

- And it will contain some of the functions we have computed before:
  - BS Price
  - Greeks
  - Implied volatility
class EuropeanOption{
private:
  double r; // Interest rate
  double sigma; // volatility
  double K; // strike price
  double T; // Maturity
  double S; // Stock price
  double C0; // option m. price
public:
  ...
};
Class member functions

- The member functions have the usual components: return type, name, parameters and function body.
- The function prototype must be defined within the class. The function body can be defined outside.
- A member function which is defined inside the class is implicitly treated as inline function. (function which is expanded “in line” during compilation) (see http://en.wikipedia.org/wiki/Inline_function).

(STAT 598W)
Lecture 8
31 / 47
Class member functions

Example:

```cpp
bool same_optionprice(const EuropeanOption &comp) const
    {return C0==comp.C0; }
```

Note that the compiler re-writes a call from the previous function as:

```cpp
EuropeanOption eur1();
    \later we'll define the proper constructor

eur1.same_optionprice(comp); \real code
EuropeanOption::same_optionprice(&eur1, comp); \pseudo-code
```

The hidden parameter &eur1 is called this. The parameter this is a pointer to the class.
Class member functions

- The `const` keyword (after the function definition) modifies the type of the `this` pointer.
- Then the class itself can’t be changed when its member functions are used.
- The pointer `this` can be used to refer its members, but it is unnecessary. Example `this->c0 = c0`.
- The member functions can be defined outside the class, but you have to indicate that they belong to the class.
Example:

double EuropeanOption::Price() const
{
    if(optType=="C"){
        return CallPrice();
    }
    else
        return PutPrice();
}

Recall :: is the scope operator.
The constructor function has (or have) the same name as its class, and it cannot return a type.

Default constructor:

```cpp
type<int> v1; // default constructor: empty vector
string s;  // default constructor: empty string
```

Then we want to initialize an “empty” European option class.
Default constructor:

```c++
EuropeanOption::EuropeanOption()
{
    r=0;
    sigma=0;
    ...
}
```

and it is declared in the class as follows (without the const keyword):

```c++
public:
EuropeanOption();
```
Also you can use this alternative notation (inside the class definition). This definition is preferred:

```cpp
EuropeanOption(): r(0), sigma(0), ... {}
```

If you don’t define any constructor, the compiler will do it for you using default values for each class member.

The class declaration should be saved in a .h file. The member functions should be saved in a separate .cpp file.
Function Overloading

- Two functions that appear in the same scope are overloaded if they have the same name but they have different arguments.
- It’s not the same to redeclare a function than to overload a function.
- Example:
  ```
  class termStructure()
  {
  private:
      vector<double> rates;
      vector<double> time;
  
  public:
      termStructure();
      double meanrate() const;
      double meanrate(const termStructure &);
  }
  ```
Function Overloading

- We can call the overloaded function as usual:

```cpp
termStructure t1,t2;
double v1=t1.meanrate();
double v2=t1.meanrate(t1);
```

- Note that the only difference between both functions is the argument list.

Remark: Since we will be using our class more than once, we don’t want multiple definitions on the same file. We can use:

```cpp
#ifndef EUROPEANOPT_H
#define EUROPEANOPT_H

//Definition of the class

#endif
```
More about Constructors

- The constructors can be overloaded. The only restriction is that every argument list is unique. Example:
  
  EuropeanOption();
  EuropeanOption(const vector<double>& );

- The constructors are executed automatically when a class is defined.
  
  EuropeanOption s1;
  EuropeanOption s1(v1);

- If a class has only public members it can be initialized like any C structure:
  
  Data val1={1000, 2000, "My_name"};
Copy Constructor

- Constructor that takes an object of the same class as parameter.
- C++ has two types of initialization:
  - Direct initialization: it uses "()". It directly invokes the proper constructor. Example:
    ```cpp
    EuropeanOption option1;
    EuropeanOption option2(20, 50, 0.01, 0.3, 1.23);
    ```
  - Copy initialization: it uses "=". It calls the proper constructor and it passes the temporarily object to the copy-constructor. Example (it only applies to single-argument classes):
    ```cpp
    dim v1 = 20;
    EuropeanOption option1 = v1;
    // here we are using the constructor:
    // EuropeanOption(const double &);
    ```
The general syntax of a copy-constructor is:

```
Class_name(const class_name &);
```

If you don’t define the copy-constructor, the compiler will define one for you using the copy-constructors for each member’s class.
We need to define an operation that releases the memory allocated on each class we have constructed.

Suppose we have this situation:

EuropeanOption *p1= new Europeanoption;
*p1(preopt);

EuropeanOption p2(*p1);
delete p1;

the last step calls the destructor of our class.
The destructors are important when you allocate memory during the class operation.

The compiler defines a default destructor.

General syntax:
```
~Name_class() { ...code... }
```

Usually the destructor: (1) is defined as a virtual function and (2) the function is empty.
Construct the class `EuropeanOption` with the following components:

- **Data members.** \( (S, K, r, T, \sigma \text{ and type of Option}) \).
- **2 constructors:** Default and user-defined.
- **Copy constructor.**
- **Destructor**
- **Functions that computes the Option price, Delta and Vega.**
Example 2

Objective: Construct a class of term structures (interest rates) with basic operations.

- Assume we have a term structure composed by continuously compounded rates: $r_1, \ldots, r_n$ and times: $t_1, \ldots, t_n$.
- For any intermediate value we will use linear interpolation between the closest adjacent nodes.
- Define the discount factor and forward interest rate as:

\[
    d_t = e^{-rt}

    f_{t_1, t_2} = r_{t_2} \frac{t_2}{t_2 - t_1} - r_{t_1} \frac{t_1}{t_2 - t_1}
\]

Include the usual constructors (2 at least), copy-constructor and destructor.
Example 3

Objective: Construct a class of Bonds with a corresponding term structure.

- Assume we have a market Bond price \( B_0 \), a term structure \( K \), coupons \( C_1, \ldots, C_m \) and payment times \( t_1, \ldots, t_m \).
- Compute the YTM solving with the bisection method:

\[
B_0 = \sum_{i=1}^{m} C_{t_i} e^{-y t_i}
\]

- Compute the theoretical price \( \tilde{B}_0 \), the duration \( D \), duration with YTM \( D_y \), convexity \( C_x \) and convexity with YTM \( C_{xy} \). Use linear interpolation in your calculations.

Include the usual constructors (2 at least), copy-constructor and destructor.