

Lecture 1

C++ in Financial Mathematics

Presentation of the course

Which programming languages are being asked for in major financial centres?

C++ is considered to be "the hottest" programming language to know in the financial sector.

If you want a job (investment bank, hedge fund etc), you have to know C++!

Banks (and not only) are using C++...

"C++ is used in situations where speed is everything," says the head of one bank.

Presentation of the course

Why C++ is so fast? Because it is a compiled language.

- Compiled languages are translated to the target machine's native language by a program called a compiler. This results in very fast code.
- Interpreted languages (R, Matlab, Python) are read by a program called an interpreter and are executed by that program. Interpreted languages are usually much slower than a compiled program.

"We've seen more requirements not just from the large banks, but also from high frequency trading...", says the managing director of a financial centre.

Much of the code in banking is in C++, and **there is a perception that if you can do C++ you can do anything!**

Presentation of the course

Aim and course design

The aim of this course is to give an introduction to C++ programming and object oriented design with a special focus on applications in computational finance. The course is a combination of lectures and practical programming sessions.

Presentation of the course

This lecture is primarily designed for people who have never programmed before. It is mainly structured into two parts:

- The first part consists in a solid introduction to C++ programming, including object-oriented programming and generic programming. The theoretical notions will be accompanied by very intuitive examples!
- The second part will focus only on financial examples! We'll show how the features of C++ (learnt during the first part) can be used to solve real financial problems. We'll study different models and create programs which will answer the following important financial problem: How do you compute the price of complex financial products?

Presentation of the course

You'll mainly learn how to design programs in financial mathematics (for example, how to price call/put options in the Black-Scholes model either using the analytical formulas, either using Monte-Carlo methods, how to price a portfolio of derivatives etc.).

Presentation of the course

At the end of the lecture:

- You'll have a deep knowledge of C++
- You'll be able to create high quality programs in C++ in financial mathematics. What does high quality mean?
 - Easy updating, maintenance
 - Testing
 - Facility to reuse the code and to extend it rapidly
 - Flexibility
 - Ease of understanding and readability (even for large programs)
 - Scalability (software which continues to work with exponentially increasing data volume)

Presentation of the course

Suggestions

- Try to understand the lectures and try to do the exercises which are proposed during the practicals by yourself! It is impossible to learn a language without attempting to speak it!
- If there is something you don't understand, don't panic! You can contact me whenever you need a help!

Some References

- J. Armstrong, C++ in Financial Mathematics
- D. Brown, G. Satir, C++ The Core Language: A Foundation for C Programmers, Cambridge
- M. Capinscki, T. Zastawniak, Numerical Methods in Finance with C++, Cambridge. (*Finite Difference Method*)
- Daniel Duffy, Introduction to C++ for Financial Engineers: An Object-Oriented Approach, Wiley Finance.
- M. Joschi, C++ Design Patterns and Derivatives Pricing, Cambridge
- B. Stroustrup (Creator of C++), Programming: Principles and Practice Using C++. Pearson Education, 2014.
- B. Eckel, Thinking in C++, Vol.I and II, Prentice Hall.

←
Easy
to
Read

One useful way to view C++ is as a collection of languages:

- the C language (the basic language, and pointers),
- an object-oriented programming language,
- A template programming language (generic programming).

Plan of the lecture

- Your first program in C++
- Basic Data types, Variables and Constants
- Operators

Your first program in C++

The first program is a program called "Hello world", which simply prints "Hello world" to your computer screen. Although it is very simple, it contains all the fundamental components that C++ programs have.

```
# include<iostream> //1.  
using namespace std; //2.  
int main() //3.  
{  
cout<<"Hello world!"; //4.  
return 0; //5.  
}
```

- ① "Header file" containing information about library which handles inputs from keyboard & output to the screen using the key words cin & cout.
- ② The latest version of C++ divides names (e.g. cin & cout) into sub collections of names called namespaces.

Namespace A	Namespace B
x	x

To avoid confusion of which x to choose we do:

A::x or B::x

:: is called the resolution operator.

Using namespace std avoids writing std::cout etc.

- ③ Every C++ program must have int main() function & your program goes inside the main() fn body between { }.
- ④ cout is like print function. << is insertion operator.
cin >> is input function. The object cin is used to read from keyboard.
- ⑤ In main() function, return 0 means program executed successfully & return non-zero means program didn't execute successfully & there were some errors.
The return 0 & return 1 has another meaning of false & true respectively if used in user defined functions.

Your first program in C++

```
# include<iostream>
```

This statement is called an include directive. It tells the compiler and the linker that the program will need to be linked to a library of routines that handle input from the keyboard and output to the screen (specifically the `cin` and `cout` statements that appear later). The header file "iostream" contains basic information about this library. You will learn much more about libraries later in this course.

Your first program in C++

```
using namespace std;
```

This statement is called a using directive. The latest versions of the C++ standard divide names (e.g. cin and cout) into subcollections of names called namespaces. This particular using directive says the program will be using names that have a meaning defined for them in the std namespace (in this case the iostream header defines meanings for cout and cin in the std namespace).

Your first program in C++

```
int main();
```

The function named *main()* is a special function in all C++ programs; it is the function called when the program is run. The execution of all C++ programs begins with the *main* function!

Lines 3 and 6: { and }

The open brace (*{*) at line 3 indicates the beginning of *main*'s function definition, and the closing brace (*}*) at line 6 indicate its end. Everything between these braces is the function's body that defines what happens when *main* is called.

The "return 0;" statement in *main* indicates the program worked as expected without any error during its execution.

Your first program in C++

Line 4: cout<<"Hello World!";

This line is a C++ statement. Statements are executed in the same order that they appear within a function's body.

This statement has three parts:

- cout, which identifies the standard character output device (usually, this is the computer screen). If you don't write the using directive, you have to write std::cout.
- the insertion operator << which indicates that what follows is inserted into cout.
- Finally, a sentence within quotes ("Hello world!") is the content inserted into the standard output.

In C++, the separation between statements is specified with an ending semicolon (;).

Basic data types and variables

When you use a variable in C++, you must specify the type of data that will be stored in that variable. Once you have chosen the type of data to be stored in a given variable you can't change it. The jargon phrase is that it is a *statically typed language*.

Why is it important to specify the type of data?

Because the data is stored in the computer data as strings of 1s and 0s. The integers are encoded as binary numbers, characters are represented as numbers written in binary etc.

Basic data types and variables

Memory terminology

Because data is stored on computers using 1s and 0s, it is natural to store integers using binary.

- A single binary digit is called *a bit* (truncation of binary digit).
- 8 binary digits are called *a byte*.
- $1024 (= 2^{10})$ bytes make a kilobyte.
- $1048576 (= 2^{20})$ bytes make a megabyte.
- Numbers representing memory locations are often written in hexadecimal. The base is 16. The 16 characters are 0, 1, 2, 3..., 9, A, B, ..., F.

↓ ... ↓
10 ... 15

Basic data types and variables

Variables

- We can assign symbolic names, known as **variables**, for storing information in the memory. Variables can be used to store floating-point numbers, characters and even pointers to other locations in memory.
- It is called variable because you can change the value stored.
- The variables must be declared before using them.

Declaration:

```
Type_of_data Name_Variable;
```

Basic data types and variables

Build-in Data Types: Types and keywords

- **Boolean** ↪ **bool**
- **Character** ↪ **char**
- **Integer** ↪ **int**
- **Floating point** ↪ **float**
- **Double floating point** ↪ **double**
- The **float**, **double** are used to store real numbers. Note that C++ considers the numbers 1.0 and 1 to have different types and so to be different.
- **Bool** is a variable which can take the value either false or true.

Basic Data Types and variables

- A **char** is a data type which is intended to store a character.

There is an important technical point concerning data of type "char". In memory a **char** is stored as a number between 0 and 255 and, consequently, takes up exactly one byte. Hence the data of type "char" is simply a subset of the data type "int". The ASCII code associates an integer value for each symbol in the character set. We can even do arithmetic with characters. For example, the following expression is evaluated as true on any computer using the ASCII character set:

```
'9' - '0' == 57 - 48 == 9
```

The ASCII code for the character '9' is decimal 57 and the ASCII code for the character '0' is decimal 48.

Basic data types and variables

However, declaring a variable to be of type "char" rather than type "int" makes an important difference as regards the type of input the program expects, and the format of the output it produces.

```
int main()
{
    int number;
    char character;
    cout<<"Type in a character: \n";
    cin>>character;
    number=(int)character;      //cast character to integer.
    cout<<"The character is "<<character;
    cout<<"is represented as the number ";
    cout<<number<<" in the computer. \n";
    return 0;
}
```

*↓
Notice the
parenthesis*

Basic data types and variables

C++ gives you a number of choices for storing integer data depending on the potential range of values your variable might take. We give below other data-type specifiers that are available, which all mean an integer of one form or another:

- signed → +ve & -ve values.
- unsigned → +ve values only.
- short
- long
- long long

eg: int signed x
int long y

For the floating-point types: long double.

- So for IR numbers we have :
- float
 - double
 - long double

Basic data types and variables

You can store smaller numbers in shorts than you can in long longs.

You can be sure that you'll be able to store a value between -2^{31} and $2^{31} - 1$ in an **int** variable and a value between -2^{63} and $2^{63} - 1$ in a long long variable.

By specifying that a variable is **unsigned**, you are saying that it is nonnegative. Specifying the sign of a number takes up one bit of memory.

Basic data types and variables

In order to get the size of various data types, use the **sizeof()** operator, whose returned type is **size_t**.

```
int main() { cout << "Size of char : " <<
    sizeof(char) << endl;
    cout << "Size of int : " << sizeof(int)
<< endl;
    cout << "Size of short int : " <<
    sizeof(short int) << endl;
    cout << "Size of long int : " <<
    sizeof(long int) << endl;
    cout << "Size of float : " << sizeof(float)
<< endl;
    cout << "Size of double : " <<
    sizeof(double) << endl;
    return 0; }
```

Basic data types and variables

An example

```
int main()
{cout<<"Please enter your age (followed by
    "<<" 'enter'):\n";
int your_age;
cin>>your_age;
// note how several values can be output by a
// single statement\\
// a statement that introduces a variable is
// called a declaration\\
// a variable holds a value of a specified
// type\\
```

Basic data types and variables

Comments

- We read into a variable
 - Here, your_age
- A variable has a type
 - Here, int
- The type of a variable determine what operations we can do on it

Basic data types and variables

Casting

- Sometimes it is important to guarantee that a value is stored as a real number, even if it is in fact a whole number. A common example is where an arithmetic expression involves division. When applied to two values of type **int**, the division operator "/" signifies integer division, so that (for example) $7/2$ evaluates to 3. In this case, if we want an answer of 3.5, we can simply add a decimal point and zero to one or both numbers - " $7.0/2$ ", " $7/2.0$ " and " $7.0/2.0$ " all give the desired result.
- If both the numerator and the divisor are variables, this trick is not possible. Instead, we have to use a type **cast** (= **convert between one data type and another**). For example, we can convert "7" to a value of type *double* using the expression "`static_cast<double>(7)`".

Basic data types and variables

Casting

Hence in the expression

```
answer=static_cast<double>(numerator) / denominator
```

the "/" will always be interpreted as real-number division, even when both "numerator" and "denominator" have integer values.

Other type names can also be used for type casting. For example, "static_cast<int>(14.35)" has an integer value of 14.

Note that we can also write $(int)(14.35)$ - C style casting!

Basic data types and variables

Casting: Another example *(Casting down incorrectly)*

Note that a cast operation might be performed even if we don't precise it explicitly. Sometimes it leads to a wrong result.

```
int a=3;    //declaration & assignment of variable in one go.  
int b=5;  
double c=a/b;  
cout<<c;
```

The compiler does perform some automatic casting, but in the wrong place. The result that we obtain is: 0.

Basic data types and variables

Casting: Another example *(Casting down incorrectly)*

Why do we obtain this wrong result? Because the code is equivalent to the following one:

```
int a=3;  
int b=5;  
int divisionResult=3/5;  
double c= static_cast<double>(divisionResult);  
cout<<c;
```

Again the result is 0.

Basic data types and variables

Casting: Another example *(Casting down correctly)*

How to get the expected result?

```
int a=3;  
int b=5;  
double c= (static_cast<double>(a)) /b;  
cout<<c;
```

You can cast either denominator or numerator. Both will work.

Example : Write a program which prints out the integer value associated with a character.

```
#include <iostream>
using namespace std;
int main()
{
    int x;
    char y;
    cin >> y;
    x = (int) y; // Casting from character to integer.
    cout << "the associated code to character" << y
        << "is" << x << endl;
    return 0;
}
```

Example : Write a program which prints "x=3" on screen.

```
#include <iostream>
using namespace std;
int main()
{
    int x; // declaration of variable.
    x = 3; // initialization of variable.
    cout << "x=" << x;
    return 0;
}
```

3

Basic data types and variables

Casting

- Casting a **double** to an **int** potentially loses information (the decimal part).
- Casting an **int** to a **double** isn't it a problem.
- Casting an **int** to a **float** is risky, since the float data type uses binary scientific notation with only a handful of significant figures. So an int cannot be represented precisely using a float.
- Casting a **bool** to an **int** converts true to 1 and false to 0.
- Casting an **int** to a **bool** is possible, but will normally result in a loss of information.
- Casting a **char** to an **int** converts a character to the number used to represent that character on your system.

Basic data types and variables

Some tips on formatting real number output When program output contains values of type **float**, **double**, **long double**, we may wish to restrict the precision with which these values are displayed on the screen, or specify whether the value should be displayed in fixed or floating point form.

```
int main()
{
    float number;
    cout<<"Type in a real number. \n";
    cin>>number;
    cout.setf(ios::fixed);
    cout.precision(2);
    cout<< "The square root of "<< number<<
    "is approximately";
    cout<<sqrt(number)<< "\.n";
    return 0;
}
```

Basic data types and variables

Some tips on formatting real number output

Output

Type in a real number.

200

The square root of 200.00 is approximately 14.14.

whereas replacing line 5 with "cout.setf(ios::scientific)" produces the output:

Type in a real number.

200

The square root of 2.00e + 02 is approximately 1.41e + 01.

In scientific notation all numbers are written in the form $p * 10^n$, where p is called the significand or mantissa. In the above example, $p = 1.41$ and $n = 1$.

Basic data types and variables

You can assign an alternative name to a data type, using **typedef**.

```
typedef int Integer; //Change int to Integer.
```

We can now declare variables to be of type "Integer":

```
typedef int Integer;  
Integer i,j,k;
```

It is often used in order to simplify the declaration of compound types such as the struct type (which we'll see later).

Basic data types and variables

Later in the course we will study the topic of data types in much more details. We will see how the programmer may define his or her own data type. This facility provides a powerful programming tool when complex structures of data need to be represented and manipulated by a C++ program.

Basic data types and variables

- We have seen that **variables have to be declared before they can be used in a program**, using statements such as:

```
float number;
```

- Between this statement and the first statement which assigns "number" an explicit value, the value contained in the variable "number" is arbitrary. **In C++ is possible to initialise variables with a particular value at the same time as declaring them.** We can thus write:

```
double PI= 3.1415926535; //declaration & initialisation.
```

Constants

- We can specify that a variable's value cannot be altered during the execution of a program using the reserved word **const**.

```
const double PI= 3.1415926535;
const char tab = '\t';
const int zip = 12440;
```

int

- For Constants, we must declare & initialise at the same time.

Enumerations

Constants of type "int" may also be declared with an enumeration statement.

```
enum{MON, TUES, WED, THURS, FRI, SAT, SUN};
```

can be seen as

```
const int MON=0;  
const int TUES=1;  
const int WED=2;  
const int THURS=3;  
const int FRI=4;  
const int SAT=5;  
const int SUN=6;
```

Enumerations

By default, members of an "enum" list are given the values 0, 1, 2 but when "enum" members are explicitly initialised, uninitialised members of the list have values that are one more than the previous value on the list:

```
enum{MON=1, TUES, WED, THURS, FRI, SAT=-1,  
SUN};
```

In this case, the value of "FRI" is 5, and the value of "SUN" is 0.

Operators

- Assignment operator
- Arithmetic operators
- Compound operators
- Relational operators
- Logic operators

Operators

We can operate on variables and constants using **operators**.

- **Assignment operator (=)**

- The assignment operator assigns a value to a variable.

```
x=5;
```

The assignment operator assigns to a variable *x* the value contained in variable *y*.

```
x=y;
```

The initial value of *x* at the moment this statement is executed is lost and replaced by the value of *y*.

Operators

- If we assign the value of y to x and y changes at a later moment, it will not affect the new value taken by x .
- The following assignment

```
y=2+(x=5);
```

is equivalent to:

```
x=5;  
y=2+x;
```

Operators

- **Arithmetic operators (+,-,*,/, %)** The arithmetical operations in C++ are:

- Operator `+`: addition
- Operator `-`: subtraction
- Operator `*`: multiplication
- Operator `/`: division
- Operator `%`: modulo

The last operator gives the remainder of a division of two values.

Operators

- **Compound assignment ($+=$, $-=$, $*=$, $/=$, $%=$)** Compound assignment operators modify the current value of a variable. They are equivalent to assigning the result of an operation to the first operand:
 - The expression $y+ = x;$ is equivalent to $y = y + x;$
 - The expression $y- = 5;$ is equivalent to ~~$x = x - 5;$~~ $y = y - 5$
 - The expression $x/ = y;$ is equivalent to $x = x / y;$
 - The expression $x* = y;$ is equivalent to $x = x * y;$

Operators

- **Increment and decrement (++,--)**

The increase operator (++) and the decrease operator (--) increase or reduce by one the value stored in a variable. The following three statements are equivalent:

```
++x;  
x+=1; // Compound Assignment Operator.  
x=x+1;
```

The following three statements are also equivalent:

```
x++;  
x+=1;  
x=x+1;
```

Operators

- **Increment and decrement (++,--)**

These operators can be used both as a prefix ($++x$) and as a postfix ($x++$)!

When an increment or decrement is used as part of an expression, there is an important difference between prefix and postfix forms. If you are using prefix form then increment or decrement will be done before rest of the expression, and if you are using postfix form, then increment or decrement will be done after the complete expression is evaluated.

Operators

Increment operator ++ as prefix/postfix

```
x=3;  
y=++x;  
// x contains 4, y contains 4
```

```
x=3;  
y=x++;  
// x contains 4, y contains 3
```

In *Example 1*, the value assigned to *y* is the value of *x* after being increased. In *Example 2*, *y* has (after assignment) the value that *x* had before being increased.

Relational/logical Operators and boolean expressions

- Intuitively, we think of expressions such as " $2 < 7$ ", " $1.2 != 3.7$ " and " $6 \geq 9$ " as evaluating to "**true**" or "**false**" ($!$ = means not equal to).
- Such expressions can be written using **the relational and comparison operators** (" $==$ ", " $!=$ ", " $>$ ", " \geq ", $<$, " \leq ") and combined using the **logical operators** " $\&\&$ " ("and"), " $\|$ " ("or") and " $!$ " ("not").

Relational/logical Operators and boolean expressions

Logical operators

- *Operator "&&":* Called Logical AND operator. If both operands are non-zero, then condition becomes true.
If A holds 1 and B holds 0, then

(A&&B) is **false**.

- *Operator "||":* Called Logical OR operator. If any of the two operands is non-zero, then condition becomes true.
If A holds 1 and B holds 0, then

(A | | B) is **true**.

Relational/logical Operators and boolean expressions

Logical operators

- *Operator "!"*: Called Logical NOT operator. Use to reverse the logical state of its operand. If a condition is true, then logical operator NOT will make it false.

If A holds 1 and B holds 0, then

`! (A || B)` is **false**.

Relational/logical Operators and boolean expressions

Examples

Expression	True or False
$(6 \leq 6) \&\& (5 < 3)$	False
$(6 \leq 6) (5 < 3)$	True
$(5! = 6)$	True
$(5 < 3) \&\& (6 \leq 6) (5! = 6)$	True
$(5 < 3) \&\& ((6 \leq 6) (5! = 6))$	False
$!((5 < 3) \&\& ((6 \leq 6) (5! = 6)))$	True

The order of precedence in logical operators are:

!, &&, || in this order. We can always change default precedence using parentheses () .

Relational/logical Operators and boolean expressions

- The forth of these expressions is true because the operator `&&` has a higher precedence than the operator `||` as it has a behavior on booleans close to `*` and `+`. If in doubt, use `()` parentheses.
- Compound Boolean expressions are typically used as the condition in "if statements" and "for loops".

Example

```
if ((total_test_score >= 50) &&
    (total_test_score < 65))
cout<<"You have just scraped through the
test.\n";
```

Relational/logical Operators and boolean expressions

- **Conditional ternary operator (?)** The conditional operator evaluates an expression, returning one value if that expression evaluates to true, and a different one if the expression evaluates as false. Its syntax is:

```
condition ? result1 : result2
```

If condition is true, the entire expression evaluates to result1, and otherwise to result2.

Relational/logical Operators and boolean expressions

Example: Computation of the maximum between two integer numbers

```
int main()
{
    int a=5;
    int b=3;
    int max=( (a>b) ?a:b );
    std::cout<<"The maximum between "<<a<<"  

and "<<b<<" is "<<max;
    return 0;
}
```

Bitwise operators

The bitwise operators work on individual bits of binary representations of the data.

- `<<` shifts all the bits in a number to the left by a given amount. So $x << 3$ shifts all the bits 3 steps to the left.
- `>>` shifts all the bits in a number to the right by a given amount. So $x >> 3$ shifts all the bits 3 steps to the right.

Bitwise operators

The bitwise operators work on individual bits of binary representations of the data.

- $\&$ is the bitwise AND operator. The n-th bit of $a \& b$ is equal to 1 if the corresponding bit of a is 1 AND the corresponding bit of b is 1.
- $|$ is the bitwise OR operator. The n-th bit of $a | b$ is equal to 1 if the corresponding bit of a is 1 OR the corresponding bit of b is 1.

Other operators: NOT operator (\sim - the bit inversion operator), XOR operator.

Bitwise operators

An example

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

int main()
{
    int a=7; //a=111;
    int b=5; // b=101;
    std::cout<<(a<<b)<<std::endl; // prints 224
    std::cout<<(a>>b)<<std::endl; // prints 0
    std::cout<<(a&b)<<std::endl; // prints 5 = 101
    std::cout<<(a|b)<<std::endl; // prints 7 = 111
    return 0;
}
```

bin^{ary}
↓
11100000
" "

↑
11100000
" "

↑
101
" "

↑
111
" "

Bitwise operators

Remark: The bitwise operators will be of no interest to us in this course, we have given them in order to have a complete image about the existing operators in C++ (this does not mean that the bitwise operators are not useful, they can very useful if you are working with computer graphics!)

Summing up

- General structure of a C++ program
- Basic types of data, casting
- Variables, constants: need to be declared before they are used

```
int x;  
x=5;
```

A const can't be changed!

```
const double y=5;  
y=7; // ERROR
```

- Boolean expressions and operators (assignment, arithmetic operators, compound operators, logic operators (AND, OR, NOT)), conditional ternary operator, bitwise operators.

• Variable Example:

```
#include <iostream>
using namespace std;
int main()
{
    int x;
    int y = 10;
    x = 3;
    x = 7;
    x = y;
    cout << x;
    return 0;
}
```

← This program prints
10 as this was the last
value assigned to x.

• Constants Example:

```
#include <iostream>
using namespace std;
int main()
{
    const int c = 5; // Must declare & initialise constant at the same time.
    c = 10; // ← Can't reassign a constant (error).
    return 0;
}
```

• Conversion from One Data Type to another (Cont'd)

Eg 1:

```
#include <iostream>
using namespace std;
int main()
{
    char C;
    int number;
    C = '7'; // Assign character 7 to variable C.
    number = 7;
    number = (int)C; // Converting character to integer & assigning it.
    cout << number;
    return 0;
}
```

Eg 2:

```
#include <iostream>
using namespace std;
int main()
{
    cout << 1/2; // prints 0 & we lose the decimal part.
    cout << 1.0/2; // prints 0.5. At least one of numerator or
                    // denominator has to be 1R number.

    int x = 9;
    int y = 2;
    cout << x/y; // prints 4. Division between integers produces integer.

    cout << static_cast<double>(x)/y; // Prints 4.5. Solves problem
                                        // by casting numerator as 1R.

    return 0;
}
```

The C-version of casting is much simpler, you just write:

(double) (x)

• Operators

(i) Assignment Operator

$x = 8;$

$y = 1 + (x = 9);$

$\Leftrightarrow \begin{cases} x = 9 \\ y = 1 + x \end{cases} \rightarrow \begin{cases} x = 9 \\ y = 10 \end{cases}$

e.g: $\left. \begin{array}{l} \text{int } x = 10; \\ \text{int } y = 20; \\ x = y; \\ y = 30; \end{array} \right\}$ What is the final value of x & y ?
 $\Rightarrow x = 20$
 $y = 30$

(ii) Arithmetic Operators

$+, -, *, /, \%$.

(iii) Compound Assignment

$$y += x \Leftrightarrow y = y + x$$

$$y -= 10 \Leftrightarrow y = y - 10$$

$$x /= y \Leftrightarrow x = x / y$$

$$x *= y \Leftrightarrow x = x * y$$

(iv) Increment/Decrement ($++$ & $--$) Operators

$++x;$ $\Leftrightarrow x += 1;$ $\Leftrightarrow x = x + 1;$	$x ++;$ $\Leftrightarrow x += 1;$ $\Leftrightarrow x = x + 1;$
---	--

Writing operator before or after operand only matters in expressions as it will affect results.

$x = 3;$ $y = ++x;$	$x = 3;$ $y = x ++;$
------------------------	-------------------------

\downarrow

$x = 4$ $y = 4$	$y = 3$ $x = 4$
--------------------	--------------------

$x = 3;$ }
 $y = ++x; \}$ is equivalent to $x = 3;$ }
 $x = x + 1;$ }
 $y = x$

$x = 3;$ }
 $y = x++;$ } is equivalent to $x = 3;$ }
 $y = x;$ }
 $x = x + 1;$ }

(V) Logical Operators

$&&$ → And ②

$||$ → OR ③

$!$ → Not ①



order of precedence

use () to change default
precedence.

Lecture 2

C++ in Financial Mathematics

Plan

- Flow of controls
- Functions

Flow of control

- If/switch statement
- Loops : for, while, do-while.
- Jump statements

Flow of control

If statement

(if - else if - - else if - else)

The syntax is the following:

```
if (expression1) {  
    statements1  
} else if (expression2) {  
    statements2  
} else if (expression3) {  
    statements3  
} else {  
    statements4  
}
```

Flow of control

If statement

Example

```
int x=7;
if (x==1) {
    std::cout<<"  x is 1 "<<std::endl;
}
else if (x==2) {
    std::cout<<"  x is 2 "<<std::endl;
}
else if (x==3) {
    std::cout<<"  x is 3 "<<std::endl;
}
else {
    std::cout<<"  x is not 1,2,3 "<<std::endl;
}
```

Flow of control

If statement

The *expression* is allowed to be any basic data type. The value 0 is interpreted as **false** and other values are interpreted as true. For example, the following code prints out that the test passed.

```
if (-1.743) {  
    cout<<"Test passed \n";  
}
```

Flow of control

If statement

A problem occurs when you use `=` in tests by accident instead of `==`.

```
int i=1;  
int j=3;  
if (i=j) {  
    cout<<"i is equal to j \n"; //runs this line.  
}
```

In the if statement, the code assigns a value of 3 to the variable *i*, and then, observing that 3 is non-zero, the above code prints out the false claim that *i* is equal to *j*.

Flow of control

Switch statement

Sometimes, it is possible to propose a more complex alternative to an if statement: the **switch** statement.

```
switch (expression) {  
    case constant1:  
        statementA1;  
        statementA2;  
        ...  
        break;  
    case constant2:  
        statement B1;  
        statement B2;  
        ...  
        break;  
        ...
```

Flow of control

Switch statement

```
default:  
statementZ1;  
statementZ2;  
...  
}
```

- The switch evaluates expression and, if expression is equal to constant1, then the statements beneath case constant 1: are executed until a break is encountered.
- If expression is not equal to constant1, then it is compared to constant2. If these are equal, then the statements beneath case constant 2: are executed until a break is encountered.

Flow of control

Switch statement

- If not, then the same process repeats for each of the constants, in turn. If none of the constants match, then the statements beneath default: are executed.
- Due to the peculiar behavior of switch-cases, curly braces are not necessary for cases where there is more than one statement (but they are necessary to enclose the entire switch-case).
- switch-cases generally have if-else equivalents but can often be a cleaner way of expressing the same behavior.

Flow of control

```
int main() {
    int x=7;
    switch(x) {
        case 1: cout<<"x is 1 \n";
        break;
        case 2: cout<<"x is 2 \n";
        break;
        case 3: cout<<"x is 3 \n";
        break;
        default:
            cout<<"x is not 1,2,3";
    }
    return 0;
}
```

Flow of control

The program will print

```
x is not 1, 2, or 3.
```

If we replace

```
int x=7;
```

with

```
int x=2;
```

then the program will print "x is 2".

Flow of control

Programming languages (in particular C++) provide various control structures that allow for the execution of a statement or group of statements multiple times. These control structures are called **loops**.

Flow of control

While loop

A **while** loop statement repeatedly executes a target statement as long as a given condition is true.

The syntax of a while loop in C++ is:

```
while (condition) {  
    statement(s);  
}
```

Statement may be a single statement or a block of statements. The loop iterates while the condition is true. When the condition becomes false, the control is passed to the line following the loop.

Flow of control

```
int main()
{
    int a=10;
    while(a<20)
    {
        cout<<"Value of a:"<<a<<endl;
        a++;
    }
    return 0;
}
```

Values printed on the screen: 10, 11, ..., 19.

Flow of control

```
void loopForever()
{
    while (true) {
        cout<<"Still looping\n" ;
    }
}
```

This program will loop forever. If you run the program, you can stop it by typing **CTRL+C** (on Windows for e.g.).

Flow of control

2. The do-while loop

```
do {statement} while (condition);
```

It behaves like a while-loop, except that *condition* is evaluated after the execution of *statement* instead of before, guaranteeing at least one execution of *statement*, even if *condition* is never fulfilled.

Flow of control

```
int main()
{
    int n=5;
    do { cout<<n<<" ";
          n--;
    } while (n!=0);
    return 0;
}
```

Values printed on the screen: 5, 4, 3, 2, 1.

Flow of control

3. The **for** loop

```
for (init; condition; increment) {  
    statement(s); }
```

Like that while-loop, this loop repeats statement while condition is true. In addition, the for loop provides an *initialization* expression (executed before the loop begins the first time) and an *increase* expression (after each iteration).

Flow of control

```
int main() {  
    for (int i=0; i<20; i++) { cout<<"Value of  
        i:"<<i<<endl; }  
    return 0;  
} // Prints 0,1,...,19.
```

Remark: C++ programmers start counting at 0. This is a matter of convention. All the standard data structures in C++ are labeled so that they start with 0. So it is strongly recommended that when programming in C++ to start counting from 0 and to use $<$ signs rather than \leq to compensate.

Flow of control

Jump statements

Break

break leaves a loop, even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end.

```
int main() {
    for (int n=10; n>0; n--)
        cout<<n<<", ";
    if (n==3)
    {
        cout<<"countdown aborted!";
        break;
    }
}
```

// Prints 10,9,...,3, countdown aborted.

Flow of control

Continue

The *continue* statement causes the program to skip the rest of the loop in the current iteration and go to the next statement. (*iteration*) Thus *continue* means "continue looping" whereas *break* means "break out of the loop".

```
int main() {
    for (int n=10; n>0; n--)
    {
        if (n==5) continue;
        cout<<n<<", ";
    }
    cout<< "End program";
}
// Prints 10, 9..., 6, 4, 3, 2, 1. End program.
```

Functions

Functions

- Build in functions
- User-defined functions: void and value returning
- Overloading
- Scope of the variables
- Static variables

Functions

- A function in C++ is a piece of code that you can call to perform some task
- They allow complicated programs to be divided into manageable pieces
- Some advantages of functions:
 - A programmer can focus on just that part of the program and construct it, debug it, and perfect it
 - Different people can work on different functions simultaneously
 - Can be re-used (even in different programs)
 - Enhance program readability

Functions

- Functions
 - Called modules
 - Like miniature programs
 - Can be put together to form a larger program

Functions

Predefined (Build-in) Functions

- In algebra, a function is defined as a rule or correspondence between values, called the function's arguments, and the unique value of the function associated with the arguments
 - If $f(x) = 2x + 5$, then $f(1) = 7$, $f(2) = 9$, and $f(3) = 11$
 - 1,2, and 3 are arguments
 - 7,9, and 11 are the corresponding values

Functions

Predefined (Build-in) Functions

- Some of the predefined mathematical functions are:

`sqrt(x)`
`pow(x,y)`
`floor(x)`

- Predefined functions are organized into separate libraries
- I/O (input/output) functions are in **iostream** header
- Math functions are in **cmath** header (you have to use
`# include<cmath>`).

Functions

Predefined Functions

- `pow(x,y)` calculates x^y
 - $\text{pow}(2.,3)=8.0$
 - Returns a value of type **double**
 - x and y are the parameters (or arguments)
 - The function has two parameters
- `sqrt(x)` calculate the nonnegative square root of x, for $x \geq 0.0$
 - $\text{sqrt}(2.25)$ is 1.5
 - Type **double**

Functions

Predefined Functions

- The floor function `floor(x)` calculates the largest whole number not greater than x
 - `floor(48.79)` is 48
 - Type **double**
 - Has only one parameter

Functions

User-Defined Functions

- **Value-returning functions:** have a return type
 - Return a value of a specific data type using the **return** statement
- **Void functions:** do not have a return type
 - Do **not** use a **return** statement to return a value (you can only use the keyword **return** (without a value!) in order to return the execution to the main program - we'll see an example).

Functions

User-Defined Functions - Value returning functions

Let us now consider the following mathematical function:

compoundInterest: $\mathbb{R} \times \mathbb{R} \times \mathbb{Z} \rightarrow \mathbb{R}$

given by

$$\text{compoundInterest}(P, i, n) = P \left(1 + \frac{i}{100}\right)^n - P.$$

Here P is the principal, i is the annual percentage rate, and n is the number of years.

Functions

User-Defined Functions

In C++ , one would write the **compoundInterest** function as follows:

```
double compoundInterest(double P, double i,
    int n)
{
    double interest = P*pow(1+0.01*i, n)-P;
    return interest;
}
```

Functions

Value returning Functions

The first line contains the following items (from left to right)

- Data type of the value returned: called the type
- Name of the function
- Number of parameters
- Data type of each parameter of the function

Functions

Value returning Functions : General Syntax

```
ReturnType functionName (ParameterType1  
    parameterName1, ParameterType2  
    parameterName2, .... ) {  
... statements....}
```

Functions

Value returning Functions - example of program

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

double compoundInterest(double P, double i,
    int n) {
    double interest=P*pow(1+0.01*i, n)-P;
    return interest;
}

int main(){
    int principal;
    double interestRate;
    int numberOfYears;
```

Functions

```
cout<<"How much are you investing?\n"
cin>>principal;
cout<<"What's the annual interest rate?\n"
cin>>interestRate;
cout<<"How long for (years)?\n ";
cin>>numberOfYears;
double interest=
compoundInterest(principal, interestRate,
numberOfYears);
cout<<"You will earn";
cout<<interest;
cout<<"\n";
return 0;
}
```

Functions

Remarks

- The definitions of the functions are written sequentially in the file. We first define compoundInterest completely, then we define the main completely.
- The variable names used when we call the function can be completely different from those used in the definition of the function. When we choose the names of parameters and variables in a function, those names have no meaning outside of the function.
- There is a **return** statement at the end of each function. When a function has computed the desired value, it sends it back to the caller using the **return** keyword.

Functions

Remarks

- The declaration and definition of the functions:
 - Notice that we defined the function **compoundInterest** before the **main** function. We did this because in **main** the **compoundInterest** function is called.
 - **Alternative:** C++ allows to **declare** a function and then to **define** it.

Functions

Value returning Functions - example of program

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

double compoundInterest(double P, double i,
    int n); // declaration

int main() {
    int principal;
    double interestRate;
    int numberOfYears;
    cout<<"How much are you investing?\n"
    cin>>principal;
    cout<<"What's the annual interest rate?\n"
```

Functions

```
cin>>interestRate;
cout<<"How long for (years)?\n ";
cin>>numberOfYears;
double interest=
compoundInterest(principal, interestRate,
numberOfYears);
cout<<"You will earn";
cout<<interest;
cout<<"\n";
return 0;
}
```

Functions

```
double compoundInterest(double P, double i,
    int n) // definition
{
    double interest=P*pow(1+0.01*i, n)-P;
    return interest;
}
```

Functions

Functions that do not return a value

Very often you want a function to perform a task and don't actually want to compute a value. To do this, you use the special keyword **void** to describe the return type.

```
void printHello() {
    cout<<"Hello\n";
}
```

Functions

Special case of using return in a function that do not return a value

In functions that do not return a value, you can use a *return* statement in the middle of a loop. This stops all looping and returns the execution to the point where the function was called.

```
void countdown() {  
    int i=10;  
    while (true) {  
        if (i==0) {  
            return; //Stops loop & exits func.  
        }  
        cout<<i<<"\n";  
        i--;  
    }  
}
```

Functions

Recursion

Function can call other functions. For example our **compoundInterest** function calls the function **pow**. An interesting feature is that functions can call themselves. This programming technique is called *recursion*.

Consider the following recursively defined sequence (which gives $n!$):

$$x_n = nx_{n-1}, \quad n \geq 1$$

with $x_0 = 1$.



recursion stopping condition.

Functions

Recursion

How to implement this recursive function in C++ ?

```
int factorial (int n)
{
    if (n==0) { return 1; } //recursion Stopping Condition.
    return n*factorial(n-1);
}
```

Functions

Overloading C++ allows *overloading* the function name, i.e. it allows more than one function to have the same name, provided all functions are either distinguishable by the typing or the number of their parameters.

Functions

Overloading - Example 1

```
int average(int first_number, int
second_number, int third_number);

int average(int first_number, int
second_number);
int main()
{
    int number_A = 5, number_B = 3, number_C
= 10;

    cout << "The integer average of " <<
number_A << " and ";
    cout << number_B << " is ";
    cout << average(number_A, number_B) <<
".\n\n";
```

Functions

Overloading - Example 1

```
cout << "The integer average of " <<
number_A << ", ";
cout << number_B << " and " << number_C <<
" is ";
cout << average(number_A, number_B,
number_C) << ".\n";

return 0;
}
```

Functions

Overloading - Example 1

```
int average(int first_number, int
            second_number, int third_number)
{
    return ((first_number + second_number +
            third_number) / 3);
}

int average(int first_number, int
            second_number)
{
    return ((first_number + second_number) /
            2);
}
```

Functions

Overloading - Example 2

```
int max(int a, int b)
{
    if (a>b) return a;
    return b;
}

double max(double a, double b)
{
    if (a>b) return a;
    return b;
}
```

Functions

The C++ compiler can work out which function we are calling. The code `max(1, 2)` would call the first version, whereas the code `max(1.0, 2.0)` would call the second version. This is desirable because we are avoiding unnecessary conversions from int variables to double variables.

The identity of a function is determined by both its name and the types of its parameters, this combination being called signature of the function. Two functions are the same if they have the same signature.

If when you call a function there isn't a version with just the right signature available, C++ will perform automatic casting if necessary. For example, if you type `max(1, 2.0)`, it will call the version of the code which treats all parameters as doubles.

Functions

Global and local variables

Consider the following code:

```
const double PI = 3.141592653589793;

double computeArea (int r) {
    double answer= 0.5*PI*r*r;
    return answer;
}

double computeCircumference (int r) {
    double answer= 2.0*PI*r;
    return answer;
}
```

Functions

Global and local variables

The *scope* of a variable refers to the parts of code where that variable can be used. We have two kinds of variables:

- *Global variables*: declared outside any function (Example: the constant PI)
- *Local variables*: the variable *answer*

The names for local variables within a function have no relationship with the names you use in another function. For example, we reused the variable name "answer" in two different functions to refer to different quantities.

The scope of local variables are only within the function.

Functions

Static variables (with local scope)

A local variable is initialized at each function call and each function call generates a copy of the variable. If a local variable is declared **static**, a unique object representing this variable will be created for all function calls. The static variable is initialized at the first execution of its definition.

We give now an example of a function which contains a loop. Inside the loop, we define two variables: n (which is of static type) and x (which is a normal variable). Compare the results!

Functions

Example

```
void f(int a)
{
    while (a--)
    {
        static int n=0; //initialised only once.
        int x=0;          //initialised at each iteration.
        cout<<"n=="<<n++<<" , x=="<<x++<<'\\n' ;
    }
}

int main()
{ f(3); return 0; }
```

0 Non-zero
 $\xrightarrow{\text{bool}}$ False $\xrightarrow{\text{bool}}$ True

Results:

```
n==0, x==0; n==1, x==0; n==2, x==0.
```

Summing up

- Operators
- Flow of control
 - Conditional flow of control: if, switch
 - Loops: for, while, do-while

Summing up

- Functions

- Predefined (built-in) functions - examples: pow, sqrt, floor (in order to use them, you have to include **cmath**)
- User defined functions:
 - Value returning function

```
double max (double a, double b);
```

- Void function

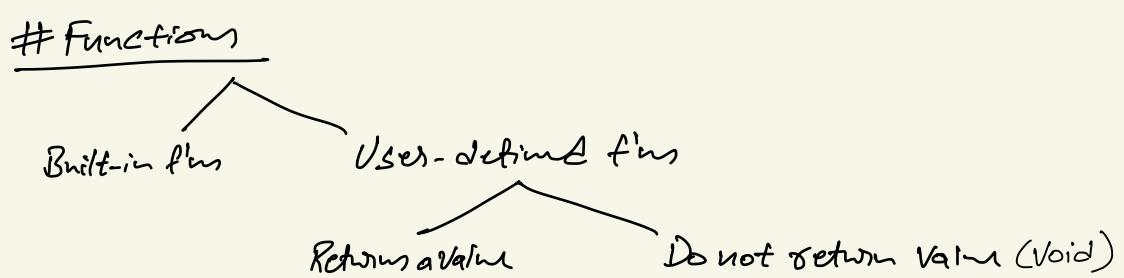
```
void print (double a);
```

Summing up

- Functions
 - Signature of a function: name+number of parameters+type of parameters.
 - Overloading functions - you can write functions with the same name, but different signatures!

```
double max (double a, double b);  
int max (int a, int b);
```

- Scope of the variable: global and local
- Static variables (with local scope)



- Value Returning functions

- We can declare & define a function at the same time before using it in the main() program or declare it first & use it in main() program & later define it.
- The variable defined inside a function is a local variable & its scope is only within the function.

- Recursion

- Programming technique where a function can call itself.

eg: factorial function.

```

int factorial (int n)
{
    if (n == 0) { return 1; } // recursion stopping condition.
    return n * factorial (n-1);
}
  
```

- Overloading

Multiple functions with same name but the difference is either in no. of parameters or in their type.

eg:

```

double average (int A, int B, int C)
{
    double average (int A, int B)
  
```

Same fn name but different no. of parameters.

{ int max (int a, int b)
double max (double a, double b)

Same file name & no. of parameters but different type of parameters.

- Overloading means the functions have same name but the Signatures are different.
- Depending on no. of parameters & type, the suitable function will be applied when called. For eg:

```
int main()
{
    double x = 2.2;
    double y = 3.1;
    int a = 3;
    int b = 2;
    cout << max(x,y) << endl; ← double max will be applied
    cout << max(a,b) << endl; ← int max will be applied
    return 0;
}
```

Static Variables (With local scope)

The word static is a key word used in different context with different meaning. Here, we use static variable in a loop. When a variable is declared static, it will only be initialised on 1st execution of loop & then after will keep its new value as shown in example with void() function. The non-static variable x is initialised x=0 at every iteration.

Pointers

A Pointer Variable is a variable which stores a memory address. This address can be a location of the following in the memory:

- Variable.
- Pointer.
- Function.

Pointers are a tool for directly manipulating computer memory.

• Declaring Pointer Variable

Pointers must be declared before they can be used like a normal variable. A pointer is associated with a type such as an int or double.

Syntax:

```
data-type * ptr;
```

example:

```
int * P1;           // declaring Pointer Variable P1.  
double * d;          
int * P1,* P2, j;  // first 2 are Pointer Variable & j is  
                   // Normal Variable.
```

• Initialising Pointers Via the Address Of Operator (&)

Eg:

```
int * pNumber;      // declaring Pointer pNumber.  
int number;         // declaring variable number.  
pNumber = &number;  // pNumber takes address of number.
```

Note: The type of variable & Pointer must match]

• Dereferencing Operator (*)

The indirection operator (or dereferencing operator)

operates on a pointer and returns the value stored at the address kept in the pointer variable.

Example:

```
int *pNumber; // pNumber is pointer to an int.  
int number = 20;  
pNumber = &number; // Takes address of number.  
*pNumber = 88; // changing value of number through the pointer  
which points to it. This is same as  
number = 88.
```

→ The indirection operator (*) can be used in both RHS ($t = *pNumber$) and the LHS ($*pNumber = 99$) of an assignment statement.

→ Note, that the symbol (*) has different meaning in a declaration statement & in an expression.

{ → Used in a declaration (int *P), it denotes that P is a pointer variable.
→ When it is used in an expression (*P = 99) or cout << *P, then it refers to the value pointed by P.

Lecture 3

C++ in Financial Mathematics

Plan

- Pointers to variables
- References
- Pointers to variables /references
- Passing arguments to functions using pointers/references
- Returning references/pointers
- Pointers to pointers
- Pointers to functions
- Static arrays/dynamic arrays
- C-style strings

Pointers, References, Arrays, Strings

Also
used in C.

Specific
to C++.

Pointers

What is a pointer?

A *pointer variable* is a variable which stores a memory address.
This address can be a location of one of the following in memory:

- Variable
- Pointer
- Function

Pointers are a tool for directly manipulating computer memory.

Pointers

Declaring pointer variable

Pointers must be declared before they can be used, like a normal variable. A pointer is associated with a type (such as an *int* or *double*).

SYNTAX

```
data_type * ptr;
```

This means: we declare a pointer variable called **ptr** as a pointer of *data_type*.

Pointers

```
int * p_1; // Declare a pointer variable  
           called p_1 pointing to an int (or int  
           pointer).  
double *d; // Declare a double pointer  
int * p_1, *p_2, j; // p_1 and p_2 are int  
                      pointers, j is an int.
```

Pointers

Initializing Pointers via the Address-of Operator (&)

When we declare a pointer, its content is not initialized. It can be initialized by assigning it a valid address. This could be done using the *address-of operator*(&).

The *address-of operator*(&) operates on a variable, and returns the address of the variable.

```
int * pNumber; // Declare a pointer variable  
                called pNumber pointing to an int.
```

```
int number; // Declare an int variable and  
            associate it a value  
pNumber=&number; // Assign the address of the  
                variable number to pointer pNumber
```

Pointers

Dereferencing Operator (*)

The *indirection operator* (or *dereferencing operator*) (*) operates on a pointer and returns the value stored at the address kept in the pointer variable.

```
int * pNumber; // Declare a pointer variable  
               called pNumber pointing to an int.  
int number=20; // Declare an int variable and  
                  associate it a value  
pNumber=&number; // Assign the address of the  
                 variable number to pointer pNumber  
cout<<*pNumber<<endl; // Print the value  
                      "pointed to" by the pointer, which is the  
                      int 88.20.  
*pNumber=99; // Assign a new value to where  
                  the pointer points to, NOT to the pointer.
```

Pointers

Remarks

- We have modified the value of the variable *number*, through the pointer *pNumber*.
- The *indirection operator* (*) can be used in both the RHS - right hand side - ($t=*\text{pNumber}$) and the LHS - left hand side - ($*\text{pNumber}=99$) of an assignment statement.
- Note that the symbol (*) has different meaning in a declaration statement and in an expression.
 - Used in a declaration (int *p), it denotes that p is a pointer variable.
 - When it is used in an expression ($*\text{p}=99$, $\text{cout} \ll *\text{p}$), it refers to the value pointed by p.

Pointers

Pointer has a Type!

A pointer is associated with a type (the one of the value it points to), which is specified during declaration. **A pointer can hold an address of the declared type; it can't hold an address of a different type.**

```
int i=88;
double d=55;
int * iPtr=&i;
double *dPtr=&d;
iPtr=&d; //ERROR iptr is integer pointer while d is double number.
dPtr=&i; //ERROR
iPtr=i; //ERROR i is integer variable while iPtr is a pointer to an integer.
        so, types are different.
int j=99;
iPtr=&j; // Change the address stored by
iPtr
```

Dynamic Memory Allocation

One can allocate memory at run time for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

When you do not need dynamically allocated memory anymore, you have to use **delete** operator, which de-allocates memory previously allocated by new operator.

Dynamic Memory Allocation

The example we have seen:

Static Memory Allocation (Memory
↳ allocated at compiled time)

```
int number=5;  
int *p=& number; // Assign a "valid" address  
into a pointer
```

New and Delete Operators

(Memory Allocated at Run time.)

```
// Dynamic allocation  
int * p2; // Not initialized  
p2=new int; // Dynamically allocate an int and  
assign its address to a pointer (Allocates some space in  
* p2=99; → memory for variable of type int & P2 stores the address)  
Assign 99 to location indicated by Pointer P2.  
  
delete p2; // Remove the dynamically allocated  
storage
```

Dynamic Memory Allocation

New and Delete Operators

Note that we can write:

```
int * p1=new int(88);  
...
```

At the address indicated by $p1 \rightarrow 88$. *(Address indicated by pointer P1
is assigned value of 88)*

In the case of **dynamic allocation: the programmer handles the memory allocation** and deallocation via **new** and **delete** operators.

Pointers and const

Pointers and const

Pointers and const

Pointing to const variables

Consider the program:

```
int x=7;  
int *ptr=&x;  
*ptr=6; //change value to 6 (Changing Value of const Variable)
```

What happens if x is const?

(EG: Non-Const Pointer to Const Variable - x)

```
const int x=7; // x is constant  
int *ptr=&x; // compiler error: cannot convert  
    const int* to int*  
*ptr=6; //change value to 6
```

↑ Error. Can't set a Non-Const Pointer to Const Variable.

Pointers and const

Pointing to const variables

The above code doesn't compile - we can't set a non-const pointer to a const variable.

Explanation

A const variable is one whose value cannot be changed. If we could set a non-const pointer to a const value, then we would be able to dereference the non-const pointer and change the value. This would violate the intention of const.

Pointers and const

(i) Non-const Pointers to Const Variable

Pointer to const variables

A *pointer to a const value* is a (non-const) pointer that points to a const value. To declare a pointer to a const value, use the *const* keyword before the data type:

```
const int x=7; // x is constant
const int *ptr=&x; // OK, ptr is pointing to a
                  "const int"
*ptr=6; //not OK, we cannot change a const
         value but we can do ptr = &y;
```

Note, the pointer is Non-const but it points to const value. So, we could change value the pointer stores to address of another const int variable. Pointer just points to a const int variable.

Pointers and const

Pointer to const variables

Consider now the following example.

```
int x=7; // x is not constant
const int *ptr=&x; // it is OK
```

A pointer to a const variable can point to a non-const variable (such as variable *x*).

Explanation

A pointer to a constant variable treats the variable as constant when it is accessed through the pointer, regardless of whether the variable was initially defined as const or not.

Pointers and const

Pointer to const variables

Consequently, the following example is OK:

```
int x=7; //  
const int *ptr=&x; // ptr points to a "const  
int"  
x=6;
```

but the following is **not OK**:

```
int x=7; // x is not constant  
const int *ptr=&x; // ptr points to a const int  
*ptr=6; // ptr treats its value as const, so  
// changing the value through ptr is not legal
```

Pointers and const

Pointer to const variables

Remark: The pointer points to a const value, but it is not const itself! In this case, the pointer can be redirected to point at other values:

```
int x_1=7; //  
const int *ptr=&x_1; // ptr points to a const  
int  
int x_2=6;  
ptr=&x_2; // OK, ptr points now at some other  
const int
```

Pointers and const

(ii) Const Pointers to Non-Const Variable

Const pointers

A **const pointer** is a pointer whose value can not be changed after initialization. To declare a const pointer, use the **const** keyword between the asterisk and the pointer name:

```
int x=5;  
int *const ptr=&x;
```

Like a normal const variable, a const pointer must be initialized to a value upon declaration. In other words, a const pointer will always point to the same address. In our example, ptr will always point to the address of x, until ptr goes out of scope and is destroyed.

Pointers and const

Const pointers

```
int x_1=5;  
int x_2=6;  
  
int * const ptr=&x_1; //OK, the const pointer  
                     is initialized to the address of x_1  
ptr=&x_2; // not OK, once initialized a const  
          pointer can not be changed
```

Pointers and const

Const pointers

Because the *value* being pointed to is still non-const, it is possible to change the value pointed to via dereferencing the const pointer:

```
int x_1=5;
int * const ptr=&x_1; //ptr will always point
                      to value
*ptr=6; // OK, since ptr points to a non-const
         int, same as saying x_1 = 6.
```

↳ Can't change the address the pointer points but can
change the value it holds as the variable is Non-const.

Pointers and const

(iii) Const Pointers to Const Variable

Const pointers to a const value

It is possible to declare a const pointer to a const value by using the *const* keyword both before the type and before the variable name:

```
int x=5;  
const int * const ptr=&x;
```

A const pointer to a const value can not be set to point to another address, nor can the value it is pointing to be changed through the pointer.

eg: Can't do both:
{ ptr = &z;
 *ptr = 7;

References

References

References

What is a reference?

A **reference** variable is an alias = another name for an already existing variable.

Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

References

Recall that we have denoted the *address-of* operator by &. C++ assigns an additional meaning to the operator & in the declaration of references variables.

In conclusion:

- When it is used in an expression, & denotes the address-of operator and is used to return the address of a variable
- When & is used in a declaration, it is part of the type identifier and is used to declare a *reference variable*.

SYNTAX

```
type & newName= existingName;
```

References

```
int main() { int number=88;
int & refNumber=number; // Declare a reference
(alias) to the variable number
    int *ptr=&number;
cout<<number<<endl; // Print value of variable
    number (88)
cout<<refNumber<<endl; // Print value of
    reference (88)
refNumber=99;
cout<<refNumber<<endl;
cout<<number<<endl; // Value of number also
    changed. }
```

References

number and **refnumber** both refer to the same location. Unlike the refnumber reference, the ptr pointer requires a storage space to store the address of number variable to which it points. To get the value of number variable, either the ptr pointer or refNumber reference can be used!

Reference doesn't take any Space in memory Whereas
Pointer takes Space in memory.

References

Another example

```
void g()
{
    int ii=0; // Variable ii initialised to 0.
    int& r=ii; // r is reference to ii.
    r++;
    int* pp=&r; // Pointer pp points to r. Exactly same as &ii.
}
```

- r increments by 1 the value of ii
 - pp points to the object referenced by the reference r (note that we can write $\&r$).
- $ii \& r$ are same.

References versus Pointers

Pointers and references are equivalent, but there are some differences:

(Only Const Pointers is initialized on declaration)

1. You need to **initialize the reference at the declaration**. Once a reference is established to a variable, **you cannot change the reference to reference another variable**. This is not the case for pointers.

```
int & iRef; // Error: 'iRef' declared as  
reference but not initialized.
```

The correct code is:

(Reference behaves like Const Pointer)

```
int x;  
int & iRef=x;
```

2. A valid reference must refer to an object; a pointer does not need. A pointer, even a const pointer, can have a null value. A null pointer doesn't point to anything.

To retain:

- We have seen pointers which hold the address of a variable.

```
int x;  
int * p; // declaration  
p=&x; // p stores the address of x
```

- References represent an alias (another name) for a variable.

```
int x;  
int & p=x; // declaration: p refers to x  
p++; // it is the same as x++
```

Remark: doing `p++` for a reference changes the value of `x`.
Doing `p++` in the case of a pointer doesn't affect the value of `x`.

Passing arguments to function C++ Returning references/pointers

How can we pass arguments to functions C++?

There are three ways of passing the arguments to functions:

- By value;
- By Reference - with pointer arguments
- By Reference - with reference arguments

How can we pass arguments to functions C++?

I. By value

When the argument is passed into functions *by value*, a clone copy of the argument is made and passed into the function. Changes to the clone copy inside the function have **NO EFFECT** on the original argument in the caller.

How can we pass arguments to functions C++?

I. By value

```
void swap(int a, int b) {  
    int temp;  
    temp=a;  
    a=b;  
    b=temp; }  
  
int main() {  
    int x=5;  
    int y=10;  
    swap(x,y);  
    cout<<x<<" "    return 0; }
```

swap	Main Program
Swap(a,b) a is copy of x & b is copy of y	x, y

Everything we do in Swap function are with copies of x & y and not with x & y themselves. So, we are interchanging the values of copies but x & y will remain same. Hence,

The values of x & y are not exchanged.
So, passing parameters by Value is not useful in this case.

The program prints out: x=5; y=10!

← x & y remains unchanged.

How can we pass arguments to functions C++?

II. By pointers

C++ allows to pass a pointer to a function. To do this, you have only to declare the function parameter as a pointer type.

We give an example where we pass two int pointers to a function which interchange their values: **the result reflects back in the calling function:**

```
void swap(int* a, int* b) {  
    int temp;  
    temp=*a;  
    *a=*b;  
    *b=temp; }
```

How can we pass arguments to functions C++?

II. By pointers

```
int main() {  
    int x=5;  
    int y=10;  
    swap (&x, &y);  
    cout<<x<< " " <<y;  
    return 0; }
```

Swap (int *a, int *b)	Main Program
a = &x; b = &y;	int x=5; int y=10;

All the changes we are doing in function swap will be done at the address of x & y so, on x & y themselves and not on the copies of x & y.

The program prints out: x=10; y=5.

The changes are "operated" at the addresses of x and y \Rightarrow x and y **are modified!**

How can we pass arguments to functions C++?

III. By reference

```
void swap(int &a, int &b) {  
    int temp;  
    temp=a;  
    a=b;  
    b=temp; }  
  
int main() {  
    int x=5;  
    int y=10;  
    swap(x, y);  
    cout<<x<<" "<<y;  
    return 0; }
```

Swap(int &a, int &b)	Main Program
a is alias for x. b is alias for y.	int x=5; int y=10;

$x \rightarrow \square \leftarrow a$ $y \rightarrow \square \leftarrow b$
a&b refer to same memory
location as x&y respectively.
Hence, Swap function interchanges
x&y.

It's similar to pointers but
much simpler syntactically
so, better to use by reference.

The program prints out: x=10; y=5.

The changes are "operated" at the addresses of x and y \Rightarrow x and y are modified!

How can we pass arguments to functions C++?

Remark: There is very little practical difference between passing data using a pointer and passing data using a reference. In the C language, you have to use pass by pointer because the concept of reference does not exist. In the C++ language, using pass by reference is the preferred approach, as it is simpler than pointers syntactically.

Can we return references or pointers?

You cannot return a reference to a local variable! (Same for Pointers)

```
int & squarePtr (int number) { // Reference returning function.  
    int Result=number*number; // Result is local Variable.  
    return Result; }
```

I have a warning message: "Reference to stack memory associated with local variable 'Result' returned". Take warning messages as errors!

& squarePtr is reference returning function & Result is what we want to return from function & this is a local variable. This is problematic since Result is destroyed soon as function is over & the function returns Reference to return.

Can we return references or pointers?

Exercise

```
int& f(int & a)    //Function returning reference with reference parameter.  
{    a=a+5;  
    return a; }  
int main(){  
    int a=5;  
    for (int i=0;i<2;i++)  
    { f(a)++; }  
    cout<<f(a);  
    return 0; }
```

Which is the value printed out by the program? Answer:22.

- Function f returns a reference; in this case it is OK because it does not return a reference to a local variable.
- We can write $f(a)++$.

Can we return references or pointers?

You cannot return a pointer to a local variable!

```
int * squarePtr (int number) { // Pointer returning Function
    int Result=number*number;
    return & Result; } // return address of Result as Function Return Type
                        // is pointer.
```

I have a warning message: "Address of stack memory associated with local variable 'Result' returned".

Pointer to pointers

Pointers to pointers

Pointer to pointers

- A pointer to a pointer is a form of multiple indirection or a chain of pointers.
- The first pointer contains the address of the second pointer, which points to the location that contains the actual value.

Declaration:

```
int **p;
```

When a value is indirectly pointed to by a pointer to a pointer, in order to access the value the asterisk operator should be applied twice.

Pointer to pointers

```
int main()
{ int var;
int *ptr;
int **pptr;
var=3000;
// take the address of var
ptr=&var;
// take the address of ptr using the address
// of operator &
pptr=&ptr;
```

Pointer to pointers

```
// take the value using pptr
cout<< "Value of var:"<< var << endl;
cout<< "Value available at *ptr:"<< * ptr<<
    endl;
cout<<"Value available at
    **pptr:"<<**pptr<<endl;
return 0;
}
```

Results:

```
Value of var: 3000
Value available at *ptr: 3000
Value available at **ptr: 3000
```

Function pointers

Function pointers

Function pointers

We have seen that a pointer can hold the address of a variable or of a pointer! Function pointers are similar, except that instead of pointing to variables, they point to functions!

Consider the function

```
int g()
{
    return 2;
}
```

- Identifier *g* is the function's name.
- The function returns an integer and has no parameters.

Function pointers

How to create a pointer to a function?

```
int (*h) (); // Pointer Function.
```

h is a pointer to a function that has no parameters and returns an integer. *h* can point to any function that matches this declaration!

In order to make a const function pointer, the const goes after the asterisk!

```
int (*const h) (); // Const Function Pointer.
```

Function pointers

How to assign a function to a pointer function?

```
int f()
{ return 2; }

int g()
{return 4; }

int main()
{
    int (*h) ()=f; // h points to f
h=g; // h now points to g
return 0; }
```

h is a pointer to a function that has no parameters and returns an integer. *h* can point to *f* and *g*!

Function pointers

Be careful: the type of the function pointer (parameters and return type) must match the one of the function.

```
// function prototypes
int f();
double g();
int h(int x);
// function pointers
int (*fPtr1) ()=f; // OK
int (*fPtr2) ()=g; // NOT OK - return types
                     don't match
double (*fPtr3) ()=g; //OK
fPtr1=h; // NOT OK - fPtr1 has no parameters,
          h has parameters
int (*fPtr4) (int)=h // OK
```

Function pointers

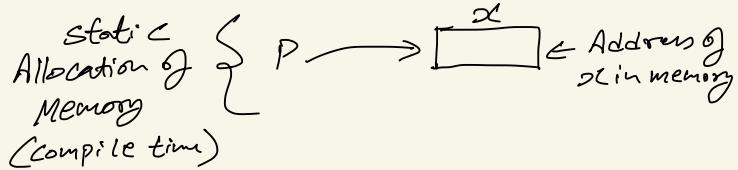
How to call a function using a function pointer?

```
int f(int x) {return x; }
int main()
{ int (*fPtr) (int)=f;
  fPtr(5); // call function f through thr
            pointer fPtr;
  return 0;
}
```

The function name is a pointer to the function, so you don't need to dereference using *. For the same reason, you don't have to use the symbol & in order to get the address of the function, as in the case of variables!

Pointers

```
int x=7;
int *P = &x;
```



Dynamic Allocation of Memory (Run time)

$\left\{ \begin{array}{l} \text{int } *P = \text{new int}; // \text{Pointer P takes new address which has been} \\ \quad \text{allocated.} \\ *P = 7; // \text{Address allocated by P is assigned value 7.} \end{array} \right.$

→ is same as: `int *P = new int(8);`

When P is no longer needed use delete to free up memory.

`delete P;`

● Pointers & Constants

(i) Non-Const Pointers to Non-Const Variable/Value.
Easy to do. Like the example above.

(ii) Non-Const Pointers to Const Variable/Values.

`const int x=7;` } Error, as type different.
`int *P = &x;`

We can't change the value of x through the pointer which points to it as x is const.
To solve this issue we do:

`const int x=10;` } OK, Now adding const keyword before type resolves problem.
`const int *P = &x;`

`int x=10;`

`const int *P = &x;` } OK.

`*P=7` } Not OK

`P = &y` } OK.

Note, the Pointer is Non-Const but it points to Const Variable/Value.
So, we could change the variable Pointer points to but not the value.
This is a Non-Const pointer that points to a Const int Variable/Value.

(iii) Const Pointer to Non-Const Variable/Value

Const Pointer must be initialised upon declaration.

int x; int y = 10;

int * const P = &x; // Const Pointer declaration & initialisation.

*P = 9; // OK.

*P = 7; // OK.

P = &y; // NOT OK.

We cannot change the Address Pointer points to but can change value it holds as variable/value is Non-Const.

(iv) Const Pointer to Const Variable/Value

In this case we can't change both address & value.

const int x = 10;

const int * const P = &x; // Const Pointer to Const Value.

int y = 9;

const int z = 11;

P = &y; // NOT OK.

*P = 20; // NOT OK.

References

Another name to existing variable. Once declared can't be changed.

int x = 10;

int & alias = x; // Another name for x is alias.

alias = 20; // This is same as x = 20.

int *P;

P = &x; // This is same as P = &alias.



Note, & operator used in declaration means

Reference while used as part of expression means address of memory location.

① Arrays

② Eg: C style strings. (Not preferred approach in C++)

③ Classes.

Arrays: Dynamic Vs Static Memory Allocation

Static allocation is less flexible than Dynamic Allocation.
The allocated memory is in compiled time & is fixed during the execution of program.
Dynamic Allocation is done in run time & you must remember to delete.

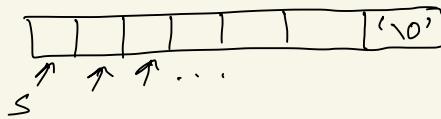
- Note, the name of an array = Pointer to first element.

MyArray[0] \rightarrow * MyArray
MyArray[1] \rightarrow * (MyArray + 1)

Eg: Write a function mystrlen() to count the no. of characters in a string without the null-terminated character. (Ex 2, Prob 8).

```
int mystrlen (char * s) // fun returning int with parameter string  
{ int length=0;  
  while ((*s) != '\0') // stopping condition of loop is to  
  { length++; // continue until the last character of  
    s++; // the string which is the null-terminating  
    character ('\0').  
  }  
  return length;  
}
```

This code replicates the function strlen() from C which we wrote ourselves.



Q: Write two functions reverseString which takes a char* string & reverse it. One should use strlen() & the other without. (Ex2, Prob.9).

(i) Version using the function strlen().

Void reverseString (char *l) // reverseString function takes parameter a pointer to char.

```
{ int n = strlen(l);  
    Char temp; // local variable temp which has type char.  
    for (int i = 0; i < n/2; i++)  
    {  
        temp = l[i];  
        l[i] = l[n-1-i];  
        l[n-1-i] = temp;  
    }  
}
```

In the for loop, if $i=0$

$temp = l[0] \rightarrow temp = 1^{st} \text{ element of } l.$

$l[0] = l[n-1-0] \rightarrow l[0] = \text{last element of } l.$

$l[n-1-0] = temp \rightarrow \text{last element} = 1^{st} \text{ element}$

Note, we go up to $n/2$, i.e. first half of array.

(ii) Version 2 → For this version which does not use the function strlen() to calculate the number of elements in the string, we need to pass additional parameter of size i.e. int n. Other than that, the rest of the code is almost same.

Object Oriented Programming

- ① Encapsulation.
- ② Inheritance.
- ③ Polymorphism.
- ④ Generic programming (Templates).

① Encapsulation (using classes)

To create user defined data type can be achieved using classes.

There are 3 steps in creating class:

(i) Class declaration
declaring member variables & functions.

(ii) Definitions of member functions

(iii) User Point
Users should not know all the details about implementation.
Some information from class will be private & public, This
is the idea of encapsulation. Private parts of class cannot
be accessible to users.

Q:

(i) Class declaration

Class Day

{ Public:

 int year;
 int month; } Member Variables
 int day;

 Void output(); ← member function which can only act on member variables

} ;

(ii) Definitions of member functions

Void Day::output() // output() function belongs to Class Day.

{ cout << day << " " << month << " " << years;

}

day, month, year corresponds to the day, month, year of
the object which will call this function.

(iii) User Point

int main()

{ Day today, birthday; // Declaring two objects today & birthday
that belong to class Day or are of type Day.

today.day = 13; // day is member variable corresponding to today.

today.month = 03;

today.year = 2021;

today.output(); // Prints 13 03 2021.

birthday.day = 18;

birthday.month = 02;

birthday.year = 1990;

birthday.output();

return 0;

}

We can access all these member variables & functions as they were declared public. By default, all member variables & functions in C++ class is set private so we must use the keyword public if we want it to set it to public.

Arrays

Arrays

Arrays

(Static Allocated Array) \rightarrow size fixed at compile time.

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

For example, five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.

Like any variable, an array must be declared before it is used.

```
type name [number_elements];
```

Arrays

```
// Create an uninitialized of length 5
int myArray[5];
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n"; }
```

- Create an array of 5 integers, without initialising it.
- Run through the entries and print them out.
- The entries start at 0.
- We use [] to access entries.
- There is no size function.

Since, the array is uninitialized, this code prints out random numbers sitting at the statically allocated memory location of myArray.

Arrays

```
// Create an initialised array
int myArray[] = {1, 1, 2, 3, 5};
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n";
}
```

- We can initialise an array by specifying the values.
- Simply place the values in a comma separated list between curly brackets.
- Notice that we no longer have to specify the length of the array when we create it.

Arrays

```
// Create an initialised array
int myArray[] = {1, 1, 2, 3, 5};
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n";
}
```

- We can initialise an array by specifying the values.
- Simply place the values in a comma separated list between curly brackets.
- Notice that we no longer have to specify the length of the array when we create it.

Arrays

```
// Create an initialised array to 0
int myArray[5] = {0};
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n";
}
```

- We specify the size of the array.
- We assign it the value {0}.
- This gives an array of the desired length full of zeros.

$$\text{myarray} = [0, 0, 0, 0, 0]$$

Arrays

```
// Create a general initialised array
int myArray[5] = {1,2,3};
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n";
}
```

- This prints out the values 1, 2, 3, 0, 0.
- The length of the array is specified.
- Some of the values are specified; the rest is padded with zero.

Arrays

Eg: Summing Elements of Array.

Passing arrays to functions

```
int sumArray( int toSum[], int length ) {  
    int sum = 0; // Local variable sum.  
    for (int i=0; i<length; i++) {  
        sum+=toSum[i];  
    }  
    return sum;  
}
```

array
↓

- One problem with arrays is that because we don't have a function which gives automatically their **size**, we must pass their length to a function. So, the functions receive as parameters the array and its length.

Arrays

How to call sumArray function in the main program?

```
int main()
{ int n=5;
int a[5]={1,2,3,4,5};
cout<<sumArray(a,n);
}
```

- The function call is **sumArray(a,n)**.

Arrays

Don't return arrays!

- Do NOT return arrays from functions.
- The caller receives a pointer to *where the array used to be*.
The computer may have reused that memory for almost anything.
- If you attempt to return an array, the behaviour is undefined.

Arrays

```
Function that returns pointer.  
int* thisFunctionReturnsAnArray(int &length) {  
    /* This produces a compiler warning */  
    length=5;  
    int array[5] = {1,2,3,4,5}; // local variable array.  
    return array; // The name of array is a pointer to the first element  
    of the sequence of elements.  
}  
  
NOT OK  
void testDontReturnArrays() {  
    int length=0;  
    int* b =  
        thisFunctionReturnsAnArray(length); // Warning message  
    for (int i=0;i<length;i++)  
        cout << b[i] << " ";  
    cout << "\n"; }  
}
```

(If we can't return pointer to local variable, this rule is the same as saying we can't return arrays from function)

// Warning message (don't return array from function)

Arrays

Don't return arrays!

```
int main()
{ testDontReturnArrays();
return 0;
}
```

I have a warning message: "Address of stack memory associated with local variable array returned".

Execute the program ↫ strange values! *(undefined behavior)*

Arrays

You can't vary the length of an array!

- You cannot change the length of an array.
- You cannot insert a new item or add some at the end.
- In fact the size is fixed AT COMPILE TIME!

Arrays

Multi-dimensional arrays

```
// Create an initialised 3x5 array
int myArray[][][5] = {{1, 2, 3, 4, 5},
                      {2, 0, 0, 0, 0},
                      {3, 0, 0, 0, 0}};
for (int i=0; i<3; i++) {
    for (int j=0; j<5; j++) {
        cout<<"Entry ("<<i<<, "<<j<<") =" ;
        cout<<myArray[i][j];
        cout<<"\n";
    }
}
```

*i = no. of row
j = no. of column*

Multi-Dim Array rule:

- RULE: You have to write explicitly the last dimension!

The no. of rows can be left empty like: myArray [] [5]
No. of rows ↗ *← No. of columns*

Arrays

Remark: In addition to accessing array elements using subscripts, array elements can also be accessed using pointers. Because an array name returns the starting address of the array (the address of the first element of the array), an array name can also be used as a pointer to the array.

```
int array[5]={0,1,2,3,4}
```

How to access the elements?

- Array-indexing:

```
array[0] // first element  
array[1] // second element  
array[2] // third element  
...
```

Arrays

Accessing Array elements using Pointers.

- Pointer notation:

```
*array; // first element  
*(array+1) // second element  
*(array+2) // third element  
...
```

The name of the array gives a pointer to the first element of the array.

Arrays

(Dynamically Allocated Array using Pointers)

Arrays and pointers

Because arrays are not flexible enough, we can work with pointers! This allows to work with sequences of data of varying lengths.

```
int n = 5;
int* myArray = new int[n]; // Declare myArray to be a
                           // pointer to a memory location
                           // which is allocated by us using
                           // new. int[n] means allocated
                           // memory for integer type
                           // array of size n.
for (int i=0; i<n; i++) {
    cout << "Entry " << i << "=";
    cout << myArray[i];
    cout << "\n";
}
delete [] myArray;
```

→ Release memory for sequence of elements not for only one element.

Arrays

Arrays and pointers

- `int * myArray` contains the memory address where the array starts.
- We use the `new ...[]` operator to allocate a chunk of memory. We are creating a sequence of `int` data types data in memory, but you can use other types of data instead.
- You can choose the size at runtime.
- The memory created will **NOT** be automatically deleted when the function exits.

Arrays and pointers

- You must use **delete []** operator to manually delete everything you create with the **new[]** operator. As we'll see, this is good and bad.
 - With arrays we couldn't return arrays because the memory was deleted automatically, but we don't have to remember to call **delete[]**.
 - With memory created using **new[]** we have to remember to delete the memory by hand, but you can safely return the data.

Arrays

Arrays and pointers

(Another way to pass an array as parameter to function using Pointer)

```
int sumUsingPointer( int* toSum, int length ) {  
    int sum = 0;  
    for (int i=0; i<length; i++) {  
        sum+=toSum[i];  
    }  
    return sum;  
}
```

Points to first element of array.

- We specify the type of the parameter as `int *`.
- The code here is identical to that with arrays except that we declare the type using `*` rather than `[]`.
- Note that you have to pass the number of elements as well as the pointer.

Arrays

→ This memory is located in heap memory as it is dynamically allocated.

Arrays and pointers

Returning arrays dynamically allocated

- We have seen that you should never return arrays from functions! If you do it, the code will behave unpredictably. It probably will print some junk if you run it.
- You are allowed to return a pointer created with `new []`, but then you'll have to make sure the caller knows whether or not they will be expected to call `delete[]` at some point.
- By convention in C and C++, if a function returns a pointer, the caller is **NOT** expected to call `delete[]`.

So, we can return an array which has been dynamically allocated with new operator, but it's not a good practice.

Arrays

```
Function that returns pointer to ind.
```

```
int* thisFunctionReturnsAPointer(int& n) {
    int* ret = new int[n]; //dynamically allocate array of size n.
    for (int i=0;i<n;i++) ret[i]=i; //Initialise array.
    return ret; //ret represents pointer to first element of array.
}

void usingReturnPointerFunction() {
    int n=5;
    int* b= thisFunctionReturnsAPointer(n); //OK.
    for (int i=0;i<n;i++) cout<<b[i];
    // free the memory
    delete[] b; //Violates convention of C/C++.
}
```

This violates the convention on NOT deleting the return value of a function, so it is considered to be confusing code.

Arrays

Looping with pointers

```
int sumUsingForAndPlusPlus( int* begin, int n)
{
    int sum = 0; // local Variable sum.
    int* end = begin + n; // Points to element after last element of array.
    for (int* ptr=begin; ptr!=end; ptr++) {
        sum += *ptr;
    }
    return sum; // increase sum by value stored at address ptr.
}
```

← Points to 1st element of array.

size of array

While ptr is different of end, so I am not at the end of array keep looping.



- You can use `++` to move a pointer on to the next item.
- You can use `==` to compare pointers.
- This code is equivalent to the last one, we just use `++` instead of arithmetic.

Arrays

Looping with pointers

The previous code is equivalent to:

```
int sumUsingForAndPlusPlus( int* toSum, int n)
{
    int sum = 0;
    for (int i=0; i<n; i++)
{sum+=toSum[i]; }
    return sum;
}
```

The above code is identical to the previous one, excepting that we declare the type using * rather than [].

C Style Strings

C Style Strings

C Style Strings

↑
Array of chars

The C-style character string originated within the C language and continues to be supported within C++. The string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello".

C Style Strings

```
char greeting[6]={'H', 'e', 'l', 'l', 'o',  
'\0'};
```

You can also write the above statement as follows:

```
char greeting[]="Hello";
```

C Style Strings

```
#include <iostream>
using namespace std;
int main();
{
char greeting[6]={'H','e','l','l','o', '\0'};
cout<<"Greeting message: ";
cout<<greeting<<endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

Greeting message: Hello

C Style Strings

C++ supports a wide range of functions that manipulate null-terminated strings:

- **strcpy(s1,s2)**
Copies string *s2* into string *s1*
- **strcat(s1,s2);** Concatenates string *s2* onto the end of string *s1*.
- **strlen(s1);** Return the length of *s1*.
- **strcmp(s1,s2);** Returns 0 if *s1* and *s2* are the same; less than 0 if *s1* < *s2*; greater than 0 if *s1* > *s2*.
- **strchr(s1,ch);** Returns a pointer to the first occurrence of character *ch* in the string *s1*.
- **strstr(s1,s2);**
Returns a pointer to the first occurrence of string *s2* in string *s1*.

C Style Strings

```
int main() {
    char str1[10] = "Hello";
    char str2[10] = "World";
    char str3[10];
    int len;
    // copy str1 into str3
    strcpy(str3, str1);
    cout << "strcpy(str3, str1) :" << str3 << endl;
    // concatenate str1 and str2
    strcat(str1, str2);
    cout << "strcat(str1, str2) :" << str1 << endl;
    // total length of str1 after concatenation
    len = strlen(str1);
    cout << "strlen(str1) :" << len << endl;
    return 0;
}
```

C Style Strings

When the code is compiled and executed, it produces the following result:

```
strcpy(str3, str1): Hello
strcat(str1, str2): HelloWorld
strlen(str1): 10
```

C Style Strings

How to use pointers in order to write your own `strlen` function?

```
int computeLengthOfString(const char* s )
{ int lenght=0;
  while ((*s) !=0) {
    s++;
    lenght++;
  }
  return lenght;
}
```

Summing up

- Pointers/references
- Pointers to pointers
- Pointers to functions
- Static/dynamic arrays
- C-style strings

Lecture 4

C++ in Financial Mathematics

Object-oriented programming Classes

Classes

A class can be visualized as a three-compartment box, as follows:

- **Classname** (or identifier): identifies the class
- **Data members or Variables** (or *attributes, states, fields*): contains the *static attributes* of the class.
- **Member functions** (or *methods, behaviors, operations*): contains the *dynamic operations* of the class.

Class Members: The data members and member functions are collectively called class members.

Classes

Terminology

We already know the following.

- Every *variable* in C++ has a type (for example, **double**).

We are learning a programming style called "object-oriented programming" in which we should use a slightly different vocabulary for the same ideas.

- Every *object* in C++ has a class (for example, **Myclass**).

Classes

The term *object* in object-oriented programming refers to a data type which consists of a bundle of data together with helpful functions to work with that data.

double, int, char* are not object-oriented types because they consist purely of data with no special helper functions.

The type of data and functions supported by an object depend only on its class. All instances of the same class have the same functions and the same types of data.

Classes

An intuitive example

Table : Class Student

Classname (Identifier)	Student
Data Member	
(Attributes)	name ; grade
Member Functions	
(Operations)	getName() ; printGrade()

Classes

An intuitive example

Table : Instances class Student

Classname	Paul: Student
Data Member	name="Paul Lee"; grade=3.5
Member Functions	getName(); printGrade()

Classname	Peter: Student
Data Member	name="Peter Tan"; grade=3.9
Member Functions	getName(); printGrade()

Classes

A first example

Class declaration:

```
class Day
{ public:
int year;
int month;
int day;
void output(); // Member function.
};
```

{} Member Variables

Member function can only act on member variables.

Classes

- The name of our class is **Day**.
- Our class **Day** contains 3 values called *year*, *month*, *day*. These represent the *members variables*. The class also contains a member function *void output()*.
- Users of our class are allowed to use the values of *year*, *month* and *day*, as well as the fonction *output()* in their own code, so these are marked **public**.

Classes

A first example

Using the class

```
int main()
{
    Day today, birthday;
    cout<<"Enter today's date: \n";
    cin>>today.year;
    cin>>today.month;
    cin>>today.day;
    cout<<"Enter your birthday:\n ";
    cin>>birthday.year;
```

Classes

A first example

```
cin>>birthday.month;
cin>>birthday.day;
today.output(); //Print day, month & year corresponding to today.
cout<<"Your birthday is";
birthday.output();
if(today.year==birthday.year&&
    today.month==birthday.month &&
    today.day==birthday.day)
cout<<"Happy birthday!\n";
else cout<<"Happy Unbirthday! \n";
return 0;
}
```

Classes

A first example

Member function definition

```
void Day::output () // f'm output() is member f'n of class Day.  
{  
cout<<"year= "<<year<<"month= "<<month<< " , day  
= "<<day<<endl ; }
```

Classes

A first example

Analysis of the code in the main function

- We first create two instances of the class Day: *today* and *birthday*.

```
Day today, birthday;
```

- We then initialize the objects *today* and *birthday*:

```
cin>>birthday.day; . . .
```

- When you want to access the data inside a class you use a dot . followed by the name of the member variable.

```
birthday.day
```

In our case, you can access the variables because they have been marked as **public**. Remove the word **public** and see what compiler error you get.

Classes

A first example

Methods

Calling member functions:

- The member function output is called with the object today as follows: `today.output()`
- The member function output is called with the object birthday as follows: `birthday.output()`

Classes

A first example

Defining member functions:

- When a member function is defined, the definition must include the class name because there may be two or more classes that have member functions with the same name.

void DayofYear::output()

- A member function is defined in the same way as any other function **except** that the *Class_name* and the scope resolution operator `::` are given in the function heading.

Classes

Defining member functions: SYNTAX

```
Returned Type  Class name:: Function  
      Name (Parameter List)  
{  Function Body Statements}
```

Classes

public and private members

- **Data hiding** is one important feature of Object Oriented Programming which allows preventing the direct access to the internal representation of a class type. The access restriction to the class members is specified by the labels **public** and **private**. The keywords public and private are called **access specifiers**.
A class can have multiple public, or private labeled sections.

By default, in C++ all member functions & variables are private.

Classes

public and private members

- (i) The **public** members are accessible from anywhere outside the class but within the program. You can set and get the value of public variables without any member function.
- (ii) A **private** member variable or function cannot be accessed, or even viewed from outside the class **Only the class** can access private members (we'll see later the friend functions). By default all the members of a class would be private.

Classes

GENERAL SYNTAX

```
class CLASS_NAME {  
public:  
DATA AND FUNCTION DECLARATIONS  
private:  
DATA AND FUNCTION DECLARATIONS  
};
```

Encapsulation

The idea of hiding data using **private** keyword is often called **encapsulation**. This concept refers to two things:

- Combining a number of items - variables and functions - into a single package, such as an object of some class;
- Preventing direct messing with the internal data of an object.

Encapsulation is the first feature of Object Oriented Programming!

Encapsulation

An intuitive example from the "real" life: The design of a car

- In a car all the lighting controls are put on the dashboard, they are separated from the controls for the windows and the seats. This grouping of functionality in a car's controls corresponds to the grouping of functionality into different classes in software design.
- You can control the car through standard functions (turn left, turn right) but the internal workings of a car are hidden from the user completely ⇒ **Software design principle:** keep most of the details private and make a small number of functions public (only those which are necessary to the user of the class).

Encapsulation

An intuitive example from the "real" life: The design of a car

- Cars are much easier to use because of the use of encapsulation. You don't need to be a trained mechanic in order to drive a car! Similarly, the concept of encapsulation in object oriented programming makes programs easy to use.

Encapsulation

Remark

It is considered good programming style to make *all* member variables private on your class. Doing this allows you to guarantee that your object always remains in a consistent state. In addition it allows you to change your mind in the future about the implementation details (i.e. how data is stored), without users of your class being affected in anyway.

Classes

An example of class with private members

```
class Day
{ public:
    void output();
    void set (int new_year, int new_month,
              int new_date);
    void input();
    int get_year();
    int get_month();
    int get_day();
private: void check_date(); // This function is private.
int year;
int month;
int day; };
```

Alternatively,
We can start writing member variables
without keyword Private & once you start
to write those who are Public you add the
key word Public. So, there are two ways to
do it.

All member variables are set Private. This is a
good way to do it & set part of member function
public so that user can use your class.

Classes (Member Function Definitions)

An example of class with private members

This input() fn is used to initialise member variables associated to given object.
and the fn checkdate() is used for validation of input data.

```
void Day::input() { cout<<"Enter the year:";  
    cin>>year;  
cout<<"Enter the month:"; cin>>month;  
cout<<"Enter the day";  
cin>>day;  
check_date(); // Private fn used in the implementation of Public fns &  
// not used by user directly.  
};  
  
void Day::checkdate()  
{ if ((year<1) || (year>2021) || (month<1) ||  
    (month>12) || (day<1) || (day>31))  
{ cout<<"Illegal date. Aborting program.\n";  
    exit(1); }  
}
```

Classes (Member Function Definitions)

```
int Day::get_year()  
{ return year; }
```

Since, members variables are set Private, we must use Accessors Functions like set & get to access them.

```
int Day::get_month()  
{ return month; }
```

```
int Day::get_day()  
{ return day; }
```

```
void Day::output() { ... }
```

```
void Day::set(int new_year, int new_month,  
             int new_date) { year=new_year;  
month=new_month;  
day=new_date; check_date(); }
```

An example of class with private members

- today.year=1997; //NO.
- today.month=12; //NO.
- today.day=25; // NO.
- cout<<today.month; // NO.
- cout<<today.day; // NO.
- if (today.month==1) //NO.

You can access the member variables only by using the member function set.

- today.set(12,1, 1) // OK

The function **set** is called **accessor function**.

Classes

A **constructor** is a member function of a class that has the same name as the class. A constructor is called automatically when an object of the class is declared. Constructors are used to initialize objects.

Rules

- A constructor must have the same name as the class.
- A constructor definition cannot return a value. No return type, no void.

- A class might have several types of constructors, so the constructor function can be overloaded.

Classes

The class BankAccount

```
class BankAccount
{ public:
    BankAccount (int dollars, int cents, double
                 rate); // Constructor with parameters.
    BankAccount (); // Default constructor which initialises member variables with
                    some default values.
    double get_balance();
    double get_rate();           } accessors function.
    void output(); // print
private:
    double balance;
    double interest_rate;      } member variables set private.
};
```

Classes

```
int main()
{
    BankAccount account1(999, 99, 5.5), account2;
    account1.output();
    account2.output();
    return 0;
}
```

Class name →

↓
Using constructor automatically
using constructor with parameters
to declare object account1.

↓
Default constructor will
be called here to declare
object account2.

Classes

(Definition of Constructor with Parameters)

Constructor 1

```
BankAccount::BankAccount(int dollars, int
    cents, double rate)
{ if((dollars<0) || (cents<0) || (rate<0))
{ cout<<"Illegal values for money or interest
    rate. \n";
exit(1);
balance=dollars+0.01*cents;
interest_rate=rate; }
```

exit(1) terminates the program (Exit with Failure) & don't execute rest of code after exit(1) line.

Classes

(Definition of Default Constructor)

Constructor 2: Default constructor

```
BankAccount::BankAccount () : balance(0);  
    interest_rate(0.0)  
{ };
```

Note that the last constructor definition is equivalent to

```
BankAccount::BankAccount ()  
{ balance=0; interest_rate=0.0; }
```

Two ways to write the same thing.

Classes

- We have **2** constructors. In other words, the constructor is *overloaded*.
- The first one is called **constructor with parameters** and the last constructor is called **default constructor**.
- A **default constructor** is simply a constructor that doesn't take parameters. If a default constructor is not defined in a class, the compiler itself defines one (*with random values*).
- Often times we want instances of our class to have specific values that we provide. In this case, we use **the constructor with parameters**.

Classes

- You can think of a constructor as a function that is automatically called before anyone is allowed to see the object. Technically speaking it isn't actually a function because it can **only** be called when the object is being initialised and because it doesn't have a return value.
- As we have seen in the example, inside the definition of the constructor you should set all **double**, **int** etc. fields to sensible default values. More generally, you should ensure that the object is in a consistent state before anyone sees it and you should perform whatever processing is required to achieve this.

Classes

(Using Constructors in Main Program)

Using constructors

Class_Name Object_Name(Arguments_for_Constructor);

eg:

BankAccount account1(999,99,5.5);

Classes

There is a third type of constructors called Copy Constructor.

Copy constructors

We have seen:

- The default constructor
- The Parametrized constructor

The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is **used** to:

- Initialize one object from another of the same type
- Copy an object to pass it as an argument to a function
- Copy an object to return it from a function.

Classes

Object name

Parameter's state will not be changed. Hence, const.
Parameter is used only to create new object & Parameter will not be changed.

Parameter is the reference to an object of same class.

Copy constructors syntax

Name_of_class Object_name (const Name_of_class & object) {...}

reference operator

Here, **object** is a reference to an object that is used to initialize another object.

If a copy constructor is not defined in a class, the compiler itself defines one.

Classes

Called Automatically when object of class goes out of scope.

Destructor

A **Destructor** is a special member of a class that is executed whenever an object of its class goes out of scope. Destructors are very useful for releasing resources in the case of dynamic allocation memory in the constructor (we'll see an example later!).

Destructor Syntax:

Parses empty parameter



```
~Name_of_class() { ... } ; \\
```

Example (in the case where there is no dynamic allocation)

```
~BankAccount () {} ;
```

↑
empty for static allocation. For dynamic allocation
we write code to release resources.

Classes

This pointer

Every object in C++ has access to its own address through a pointer called **this** pointer. It can be used inside a member function in order to refer to the invoking object.

```
BankAccount::BankAccount(int dollars, int  
    cents, double rate)  
{ if((dollars<0) || (cents<0) || (rate<0))  
{ cout<<"Illegal values for money or interest  
    rate. \n";  
exit(1); }  
*this.balance = dollars+0.01*cents;  
*this.interest_rate = rate; }
```

*this.balance is same as balance.

This pointer

```
(*this).balance = dollars + 0.01 * cents;
```

```
(*this).interest_rate = rate;
```

has the same meaning as

```
this → balance = dollars + 0.01 * cents;
```

```
this → interest_rate = rate;
```

This is a pointer to an object of the class BankAccount.
IMPORTANT RULE: To access the member variables through
a pointer, use the operator → !

Classes

This pointer

this always points to the object being operated on. More precisely, "this" is a const pointer. You can change the value of the underlying object it points to, but you can not make it point to something else!

↓
(for eg. in previous example, this has
the type BankAccount* const).

Classes

Some examples when you need the pointer this

- (i) If you have a constructor (or member function) that has a parameter with the same name as a member variable, you should use "this" (if not, ambiguity!)

```
class YourClass{  
private: int data; // ← Note, Member Variable data has  
// same name as Parameter data.  
  
public: YourFunction(int data) {  
this->data=data; // initialising Private member data with the  
// Parameter data. data=data without this would  
}  
};
```

- (ii) It will be used for the overloading of operators (see next lecture!).

Classes

We can use static member variable to keep track of number of different instances/object that has been created from a class. i.e. how many object of a class has been created.

Static members

- While most variables declared inside a class occur on an instance-by-instance basis (which is to say that for each instance of a class, the variable can have a different value), a static member variable has the same value in any instance of the class. For instance, if you wanted to number the instances of a class, you could use a static member variable to keep track of the last number used.



More precisely, static member variables and static functions are associated with the class, not with an instance.

Classes

Static members

- Since the static member variables do not belong to a single instance of the class, you have to refer to the static members through the use of the class name.

```
class_name::x;
```

- You can also have static member functions of a class. Static member functions are functions that do not require an instance of the class, and are called the same way you access static member variables. Static member functions can only operate on static members, as they do not belong to specific instances of a class.

```
class_name::static_function();
```

Classes

Static members

- **Static functions** can be **used** to modify static member variables to keep track of their values : you might use a static member function if you chose to use a counter to give each instance of a class a unique id.

```
class user
{ private:
    int id;
    static int next_id; // Static variable next_id.
public:
// constructor
user();
static int next_user_id() // Static member function.
{ next_id++;
    return next_id; }
};
```

Classes

Static members

```
int user::next_id = 0; //initialising static member variable.  
  
// constructor  
user::user()  
{  
    id = user::next_id++; // or  
    id=user::next_user_id();  
}  
};
```

The line

```
user a_user; //automatically runs constructor.
```

would set id to the next id number not assigned to any other user.

Summing up

- Arrays (static and dynamic allocation)
- Classes
 - The notions of object-instance
 - Member values and member functions
 - The concept of encapsulation
 - Constructor/destructor
 - This pointer
 - Classes with static members
- Working with different files

Classes

→ Member Variables and Functions : Private & Public

→ Special Functions: Constructors

Automatically gets called when an object is created.

eg: (Day d1) → Constructor provided by C++ compiler if not defined,
& works fine as we had no dynamic allocations to do.

3 types of Constructors:

- Default constructor → initialise member variable with some values.
- Constructor with Parameter → To initialise member variable with parameters.
- Copy Constructor → Create new object using already existing one.

• Destructors

eg: ~Day () { }

Destructor doesn't receive any parameter. In case of dynamic allocation, we write code in ~D to release the memory which has been allocated for the object.

[Note: The name of Constructors and Destructors must coincide with the name of class]

→ Static Members : Are associated with the class & not with particular instance or object of that class.
Main Application of static member variables is to count number of instances of that class that has been created.

→ Operator Overloading

int a;
int b;
a + b

MyString a;

MyString b;

a + b (?)



operator+ (const MyString, const MyString)

1st Example: Class Complex

In order to use the class complex in C++ do,
 include <complex>
 as class complex already exist in C++.

lets write our own class complex:

(I) Declaration of class:

Class Complex

{ Private:

 double x, y;

Public:

 Complex(); // default constructor.

 Complex(double x, double y); // constructor with parameters.

 Complex(const Complex & c); // copy constructor. Must include reference as if you don't it will

we could write several other constructors as well. For example, we could define another constructor with parameters which is Complex & it receives only one parameter. This means one of the private member variable should initialise with value of parameter and the other one with default value.

pass by value & in function it will do copy of parameter. But in order to do copy of an object copy constructor is called. Result in an loop if reference not used.

This means one of the private member variable should initialise with value of parameter and the other one with default value.

~Complex(); // trivial destructor.

Since, member variables are private, In order to ensure all modifications to member variables are done in a safe way & the object preserves its consistency, we introduce accessor functions get and set.

double getx() const;

double gety() const;

void set(double x, double y);

} accessor functions

// Member Operators of class.

Complex & operator=(const Complex & c);

Complex & operator+=(const Complex & c);

}; // finished declaration of class.

+ =, - =, /= etc has to be passed as member functions of the class.

// Non-Member Operators of the class
Complex operator + (const Complex & c1, const Complex & c2);
So, we saw binary operator which is member of a class & binary operator which is Non-member of a class.

In previous page, a function followed by keyword const, for eg: double getx() const; means the function does not change the values of the private member variables. This will be essential when we call functions with objects which are constants. For objects of type const, we can only call member functions which are const.

II Definition of the member functions

double Complex::getx() const
{ return x; } // returns Private Member Variable x.

double Complex::gety() const
{ return y; }

[Rule: For all functions which do not modify the member variables, use keyword const after function.]

lets define operators now.

// member operator of a class.

Complex & Complex::operator=(const Complex & c)
{ x = c.x; y = c.y;
return *this;
}

This (=) operator is a member function of a class. It's return type is a reference to a Complex, then it follows the name of the class, follows resolution operator, the keyword operator, = and between the brackets, I have an object which is a reference to a const Complex because the parameter will not be the value of c that will be changed but the value of the invoking object. $x = c.x;$ \leftarrow so on L.H.S is private member variable x of the invoking object & it receives the value x of the object which is passed as parameter. $return *this$ means we are returning a reference to the invoking object (return content which can be found at the address given by this).

"return * this" → the invoking object.

Alternatively, the code for operator (=) can be also written as:

Complex & Complex:: operator=(const Complex & c)

{ this → x = c.x;

this → y = c.y;

return * this;

3

(III) User Side (Part) (main program)

Complex z1; // default constructor is called.

Complex z2(2., 3.); // constructor with parameter is called.

Complex z3; // Again default constructor called.

z3 = z2;

↳ This line is same as

z3.operator=(z2);

↑
invoking object

Remark: Recall in defining the member operator of a class (=), we return a reference.

Complex & Complex:: operator=(const Complex & c)

{ x = c.x; y = c.y;

return * this;

3

We return a reference to Complex so that we can do things like below:

$z1 = z2 = z3 = z4; // \text{Can do multiple assignments.}$

We start with $z3 = z4$, this returns reference to $z3$ then the value of $z3$ will be passed as parameter to operator for $z2$ and so on.

Hence, it's essential to return a reference in order to do some assignments in chain.

Definition of Non-member functions

lets see how we overload the operator (+). The + operator was Non-member function which returns Complex. We don't write the class Resolution operator as it's a Non-member function of the class. So, We just write

Complex operator + (const Complex & C1, const Complex & C2)

{ Complex sum;

sum.setx(C1.getx() + C2.getx());

sum.sety(C1.gety() + C2.gety());

return sum;

}

Being Non-member of a class, we can't have direct access to private members. So, we need to use accessor functions.

User Part (Non-member function +)

Complex z1; // default constructor called.

Complex z2; // " " — " "

Complex z3(2.0, 3.0); // constructor with parameter called.

$z1 = z2 + z3;$

↳ same as

operator + (z2, z3);

Complex z4 = z2; // Declaring & initialising new object z4 using copy constructor which receives z2 as a parameter.

↳ same as

Complex z4(z2);

So, If I declare an object followed by equal another object, it is the copy constructor which is called. But If I declare object first then do something like $z1 = z2$, then it is the assignment operator which is called.

Lecture 5

C++ in Financial Mathematics

Overloaded operators

Operator Overloading in C++

This means the operators like +,- can be redefined to operate on new data types that you are creating. For eg. + operator can be redefined to concatenate strings.

- In C++ the overloading principle applies not only to functions, but to operators too. The operators can be extended to work not just with built-in types but also classes.
- A programmer can provide his own operator to a class by overloading the build-in operator to perform some specific computation when the operator is used on objects of that class.
- Overloaded operators are functions with special names **the keyword operator** followed by the symbol for the operator being defined. Like any other function, **an overloaded operator has a return type and a parameter list.**

Operator Overloading in C++

Example 1.

```
int a=2;  
int b=3;  
cout<<a+b<<endl;
```

The compiler comes with a built-in version of the operator (+) for integer operands - this function adds integers x and y together and returns an integer result. The expression $a + b$ could be translated to a function call which would take the following form

operator+(a, b) // + is a function which has the name operator.

Operator Overloading in C++

Example 2.

```
double c=2.0;  
double d=3.0;  
  
cout<<c+d<<endl;
```

The compiler also comes with a built-in version of the operator (+) for double operands. The expression $c + d$ becomes fonction call operator+(c,d), and function overloading is used to determine that the compiler should be calling the double version of this function instead of the integer version.

Operator Overloading in C++

Example 3.

Add two objects of class **string** (we'll see this class more in detail later).

```
Mystring string1="Hello, ";
Mystring string2="world!";
std::cout<<string1+string2<<std::endl;
```

The intuitive expected result is that the string “Hello, World!” would be printed on the screen. However, because Mystring is a user-defined class, the compiler does not have a built-in version of the plus operator that it can use for Mystring operands. In this case the operand will give an error. **Conclusion:** it is needed an overloaded function to tell the compiler how the + operator should work with two operands of type Mystring.

Operator Overloading in C++

Rules for Operator Overloading:

- Almost any existing operator in C++ can be overloaded. The **exceptions** are: conditional (?:), sizeof, scope (::), member selector (.), and member pointer selector (.*).
- You can only overload the operator that exist. You can not create new operators or rename existing operators.
- At least one of the operators must be an user-defined type.
- Is not possible to change the number of operands an operator could support. (*e.g + can only receive 2 operands*)
- All operators keep their default precedence and associativity.

When overloading operators, it's best to keep the function of the operators as close to the original intent of the operators as possible.

Operator Overloading in C++

There are two classifications of operators.

(i) A first classification of operators

- Unary operators: they operate on a single operand and the examples of unary operators are the following:
 - The increment (++) and decrement (--) operators.
 - The unary minus (-) operator.
 - The logical not (!) operator.
- Binary operators have two operands, as for example the addition operator +, the subtraction operator -, the division operator (/) etc.

Operator Overloading in C++

(ii) A second classification of operators (Member & Non-Member of a class)

- Member operators of a class

- Unary operators *Syntax*

operand is calling the function so no parameters & it is the invoking object. So, Member of class.

```
Class_type X{...public: // Here, we declare public member variables of a class.  
Class_type operator++() {...}  
}
```

↑ Unary doesn't require parameters as operand is the invoking object.

- Binary operators *Syntax*

Receives only one parameter, the other is the invoking object so, Member of class.

```
Class_type X{...public:  
Class_type operator+(const Class_type&  
          c) {...}  
}
```

↑ Receives only one parameter & other is the invoking object.

There are operators which can be only declared as member operators. Example: =, []...

Operator Overloading in C++

A second classification of operators

- Non-member operators of a class

- Unary operators *Syntax*

Non-member
so receives
our parameters
which is the
operands.

```
Class_type X{...}
```

receives parameter which is an
operand.
↓

```
Class_type operator++(Class_type& c) {...}
```

- Binary operators *Syntax*

Non-member
so, receives
2 parameters,
which are the
operands.

```
Class_type X{...}
```

receives 2 parameters which are
the operands.
↓

```
Class_type operator+(const Class_type& c,  
                      const Class_type& d) {...}
```

Since the unary operators only operate on the object they are applied to, unary operator overloads are generally implemented as member functions!

Difference between Member & Non-member Operator Class

- Member

Unary → Operand is calling the function & it is the invoking object
So, no parameter.

Binary → Operand is calling the function so for e.g.: for assignment (=)
operator, one of the operand is the one who invokes the
function & the other operand is passed as parameter, so only
one parameter.

- Non-Member

Binary → For e.g.: + operator is declared as Non-member as it
requires 2 parameters.

So, the main distinction between Member & Non-member is in
terms of passing parameters & invoking object. One clear distinction
is the no. of parameters that the operator function receives.

Operator Overloading in C++

Rules concerning operator overloading

- If you are overloading a unary operator, do so as member function.
- If you are overloading assignment (`=`), subscript `[]`, function call `(())` or member selection `(->)`, do so as member function.
- If you are overloading a binary operator that modifies its left operand (e.g. operator `+ =`) do so as a member function.
- If you are overloading a binary operator that does not modify its left operand (e.g. operator `+`), do so as a normal function or friend function.

↓
Non-Member operators
of a class.

Study case: Complex class

Complex class

- Making a class for complex numbers is a good educational example
- C++ already has a class **complex** in its standard template library (STL) - use that one for professional work

```
#include <complex>
complex<double> z(5.3,2.1), y(0.3);
cout<<z*y+3;
```

- However, writing your own class for complex numbers is a very good exercise for novice C++ programmers!

Complex class

How would we like to use the Complex Class?

```
void main()
{
    Complex a(0,1);
    Complex b(2), c(3,-1);
    Complex q=b;
}
cout<<"q="<<q<<", a="<<a<<", b="<<b<<endl;
q=a*c+b/a;
cout<<"Re (q) ="<<q.Re () <<",
    Im (q) ="<<q.Im () <<endl;
}
```

Complex class

Basic contents of class Complex

- **Private** data members: real and imaginary part
- Some **public** member functions:
 - Constructors (in order to construct complex numbers)

```
Complex a(0,1); //imaginary unit  
Complex b(2), c(3,-1);  
Complex q=b;
```

- Other functions (not the complete list, just examples):

```
cout<<c.Get_Re();  
cout<<c.abs();
```

Complex class

Basic contents of class Complex

- Some **operators** declared in the public part:
 - In order to write out complex numbers

```
cout<<"q="<<q<<" , a="<<a<<" , b="<<b<<endl;
```

- In order to perform arithmetic operations:

```
q=a*c+b/a; ;
```

Complex class

Writing our own Complex class.

```
class Complex
{
private:
    double re,im; //real and imaginary part
public:
    Complex(); // default constructor
    Complex(double re, double im); // Complex
    a(4,3); Constructor with parameters
    Complex (const Complex &c); // Complex
    q(a); Copy Constructor
    ~Complex () {} // trivial destructor.
    double Get_Re() const;
    double Get_Im() const;
```

Complex class

```
void Set_Re(double);
void Set_Im(double);

double abs () const; //double m=a.abs ();
// modulus
/*member operator*/
Complex& operator= (const Complex& c); //
a=b; //overloading assignment operator(=) declared as member
      function of the class (binary operator)
};
```

Complex class

```
/*non-member operator, defined outside the
class*/
Complex operator+ (const Complex& a, const
Complex& b);
Complex operator- (const Complex& a, const
Complex& b);
Complex operator/ (const Complex& a, const
Complex& b);
Complex operator* (const Complex& a, const
Complex& b);
```

Complex class

The simplest functions

- Extract the real and imaginary part (recall: these are private, i.e. invisible for users of the class; here we get a copy of them for reading)

```
double Complex::Get_Re() const {return re; }
double Complex:: Get_Im() const {return
    im; }
```

- Computing the modulus:

```
double Complex::abs() const {return
    sqrt(re*re+im*im); }
```

Complex class

Inline functions

In the case of inline functions, the compiler replaces the function call statement with the function code itself (process called expansion) and then compiles the entire code.

- There are two ways to do this:
 - (1) Define the member-function inside the class definition.
 - (2) Define the member-function outside the class definition and use the explicit keyword **inline**:

```
inline double Complex::Get_Re() const
{ return re; }
```

When are inline functions useful? Inline functions are best for small functions that are called often!

Complex class

The **const** concept

A **const member function** is a member function that guarantees it will not modify the object.

As we have seen, to make a member function **const**, we simply append the **const** keyword to the function prototype, after the parameter list, but before the function body.

```
double Complex::Get_Re() const { return re; }
```

Complex class

The **const** concept

Any const member function that attempts to change a member variable or call a non-const member function will cause a compiler error to occur.

```
void Complex::Set_Re() const {re=0;} //  
compile error, const functions can't change  
member variables.
```

Rule: Make any member function that does not modify the state of the class object const.

Remark: Note that **constructors cannot be marked as const.**

Complex class

The **const** concept

- Recall that **const** variables cannot be changed:

```
const double p=3;  
p=4; // ILLEGAL!! compiler error
```

- const** arguments (in functions)

```
void myfunc (const Complex& c)  
{c.re=0.2; /* ILLEGAL!! compiler error */}
```

Complex class

The **const** concept

- **const** Complex arguments can only call **const** functions:

```
double myabs (const Complex& c)
{return c.abs();} // ok, because c.abs() is
a const function.
```

- Without **const** in

```
double Complex::abs () {return
    sqrt (x*x+y*y);}
```

the compiler would not allow the `c.abs` call in `myabs`

```
double myabs (const Complex& c)
{return c.abs();}
```

because `Complex::abs` is not a **const** member function

Complex class

Question: how to create a complex number which is the sum of two complex numbers?

```
Complex c1 (1,2);
```

```
Complex c2(2,3);
```

```
Complex sum=c1+c2;
```

Answer: Overload the operator "+"

Complex class

Overloading the "+" operator

- To overload the `+` operator, first notice that the `+` operator will need to take two parameters, both of them of `Complex` type. To be more precise, these parameters must be **const** references to `Complex`.
- The operator `+` will return a `Complex` containing the result of the addition.
- To overload the `+` operator, we write **a function** that performs the necessary computation with the given parameters and return types. The only particular thing about this function is that it must have the name **operator+**.

Complex class

- The meaning of + for Complex objects is defined in the following function

```
Complex operator + (const Complex& c1,  
                    const Complex& c2 )
```

- The compiler translates

```
c=a+b;
```

into

```
c= operator+ (a, b);
```

Complex class

There are several ways to define the operator +.

First possibility:

```
Complex operator+ (const Complex& a, const
    Complex& b)
{ Complex temp;
    temp.Set_Re(a.Get_Re() + b.Get_Re());
    temp.Set_Im(a.Get_Im() + b.Get_Im());
    return temp; }
```

Complex class

Second possibility

```
Complex operator+ (const Complex& a, const  
Complex& b)  
{ return Complex (a.Get_Re() +b.Get_Re(),  
a.Get_Im() +b.Get_Im()); }
```

Complex class

Third possibility

```
Complex operator+ (const Complex& a, const  
                    Complex& b)  
{ Complex temp;  
    temp=a;  
    temp+=b;  
    return a;  
}
```

Here we use the following idea: we can first overload the assignment operator (=) and the operator += as member operators. Using these operators, one can overload the non-member operator +.

Complex class

The assignment operator

- Writing

```
a=b;
```

implies a call

```
a.operator= (b)
```

- this is the definition of assignment

Complex class

The assignment operator

- We implement operator= as a part of the class:

```
Complex& Complex::operator= (const Complex&
    c)
{
    re=c.re;
    im=c.im;
    return *this;
}
```

- If you forget to implement operator=, C++ will make one (this can be dangerous)

Complex class

The multiplication operator

- First attempt

```
Complex operator* (const Complex& a, const
    Complex& b)
{
    Complex h; // Complex()
    h.re=a.re*b.re-a.im*b.im;
    h.im=a.im*b.re+a.re*b.im;
}
```

Complex class

The multiplication operator

- Alternative (avoiding the *h* variable)

```
Complex operator* (const Complex& a, const
    Complex& b)
{
    return Complex(a.re*b.re-a.im*b.im,
        a.im*b.re+a.re+b.im)
}
```

Complex class

Remark

- The member operators `+ =`, `- =` can be implemented in the same way as `=`
- The non-member operators `-`, `/` can be implemented in the same way as `+` and `*`.

Complex class

Constructors

- Recall that constructors are **special** functions that have the same name as the class
- The declaration statement

```
class q;
```

calls the member function Complex()

- A possible implementation is

```
Complex:: Complex {re=im=0.0; }
```

In this case, declaring a complex number means making the number (0, 0).

Complex class

Constructors with arguments

- The declaration statement

```
class q(-3,1.4);
```

calls the member function Complex(double, double)

- A possible implementation is

```
Complex:: Complex (double re_, double im_)
{re=re_; im=im_; }
```

Complex class

Constructors with arguments

- A second possible implementation is

```
Complex:: Complex (double re, double im)  
{this->re=re; this->im=im; }
```

Note that in this case we use the pointer **this**, since we have parameters with the same name as the private members.

Complex class

Copy constructor/Assignment operator

- The statements

```
Complex q=b;  
Complex q(b);
```

makes a new object *q*, which becomes a copy of *b*. In this case, the **copy constructor** is called.

- Note the difference with respect to:

```
Complex b;  
Complex q;  
q=b;
```

where first the **default constructors** are called and then the **assignment operator** is used.

Complex class

Copy constructor

- First implementation :

```
Complex::Complex (const Complex& c)
{re=c.re; im=c.im; }
```

- Implementation in terms of assignment:

```
Complex::Complex (const Complex& c)
{*this=c; }
```

- Recall that **this** is a pointer to "this object", ***this** is the present object, so ***this=c** means setting the present object equal to *c*, i.e. this → operator=(*c*)

Complex class

Copy constructor

- The copy constructor defines the way in which the copy is done. This also includes the argument. That's why the following statement

```
Complex::Complex (const Complex c):  
    re(c.re), im(c.im) {}
```

represents an ERROR. In this case, this call would imply an infinite recurrence.

RULE: The correct declaration of the copy constructor is

```
Complex (const Complex& c);
```

Dont' forget the & symbol!

Complex class

Overloading the output operator

- Output format of a complex number: (re,im), i.e. (1.4, -1)
- Desired user syntax:

```
cout << c;
```

- The effect of « for a Complex object is defined in

```
ostream& operator<< (ostream& o, Const
    Complex& c)
{o << "(" << c.Re() << ", " << c.Im() << ")";
    return o; }
```

Complex class

Some comments on the overloaded operator <<

- The **operator** << is defined as a non-member function.
- The **operator** << always takes an ostream in its first input. This is because we always have a stream on the left of << (ostream is a class and **cout** is an object of "type" ostream).
- The second parameter is, in this case, a Complex. This is because this is the type of data we wish to print out.
- The function **operator** << returns a reference to the **ostream**. This will in practice always be the same **ostream** that we pass in as the parameter out.

Complex class

Why returning by reference?

- Recall that return by reference is acceptable so long as **you don't return a reference to a local variable**. Return by reference is more efficient than return by value, since it avoids copying (recall that when a function returns by value, the copy constructor is called).
- One effect of returning a reference is that whoever receives the reference can use that reference to modify whatever it points to. See an example on the following slide.

Complex class

Why returning by reference?

Consider the code:

```
cout<<"To be"<<"or not to be";
```

This code is equivalent to the following:

```
(cout<<"To be")<<"or not to be";
```

This shows why the fact the the operator `<<` returns a stream by reference is useful. **We can apply the `<<` operator again!**

Working with different files

Working with different files

When writing programs, we try to split the program into independent pieces or modules. In general, we create three files:

- *Header file* describing the class members (data and functions). The header file has the extension .h
- *The implementation of the class* goes into the .cpp file (member function definition)
- File containing the program that uses your class (which has the extension .cpp).

Remark: In the case when we don't have classes, only functions: Function declarations must be done in the header file and the definitions go into the .cpp file.

Working with different files

File: **Complex.h**

(declaring Member Variables & functions)

```
# pragma once
class Complex
{
private:
    double re;      // member variables.
    double im;
public:
    Complex();      // default constructor
    Complex(double x, double y); // constructor with parameters.
    Complex(const Complex& c); // copy constructor
    ~Complex(){};     // destructor
    double Get_Re(); // accessor fn.
    Complex& operator=(const Complex&); // and all
        the functions and operators // assignment operator overloading.
};
```

Working with different files

File: **Complex.cpp**

(Definition of Member functions)

```
# include "Complex.h"
Complex::Complex(): re(0.0), im(0.0){}; // default constructor
Complex::Complex(double x, double y) {re=x;
    im=y; }
double Complex::Get_Re()
{
    return re;
}

// and the other definitions
```

Working with different files

File: **main.cpp**

(Main file : User Part)

```
#include <iostream>
# include "Complex.h"

using namespace std;

int main()
{
    Complex z1; // default constructor
    cout<<z1.Get_Re()<<endl;
    return 0;
}
```

Working with different files

Some rules:

- **Pragma once**

Every header file has to start with **pragma once**. The reason you should start every file with pragma once is that it stops the same file being include twice.

- Don't include definitions of functions in the header file, **except** for the **inline functions!** → *include def'n of f'n in header file when the fn is declared.*
- Don't use **using namespace std** in a header file.
- Another rule you should follow is to never have circular dependencies through include. For example, two header files should not include each other.
- Each .cpp file has to include the header file.

Summing up

- Classes
 - Constructor/destructor
 - This pointer
 - Classes with static members
- Overloading operators
- Study case: Complex class
- Working with different files

Lecture 6

C++ in Financial Mathematics

Classes with Non-Trivial Destructor

We will learn to create our own vector class "YourVector", although C++ has an in-built vector class `#include <vector>`. This will allow us to create Matrix class which will be very similar as a matrix can be seen as 2-Dimensional vector & C++ does not have a built-in Matrix class.

A Matrix Class

- Constructors.
- Destructors (Non-trivial).
- Rule of three.
- Returning References.
- Overloading using const.

$$3 \times 2 \text{ Matrix} \rightarrow \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}$$

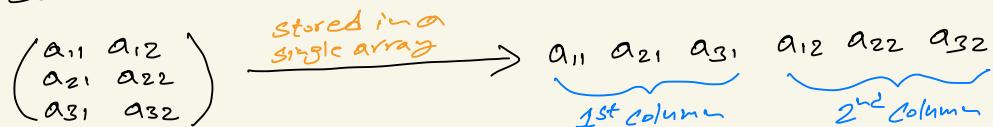
An inefficient way to store a matrix as double pointer.

`double **P`

`P[i]` → Pointer to i^{th} column.

This is not good as we have double indirection & takes more time than if you store all elements in one vector. You store all elements of 1st column, then 2nd column & so on (Very slow).

The efficient way to store the elements of a Matrix is to store all of them in one array.



Data Members (All Private)

- int rows (no. of rows)
- int cols (no. of columns)
- double * data (a pointer to the first cell/element)
- double * endPointer (a pointer after the last cell/element)

- The pointer data will point to a single chunk of memory of length $\text{nrows} \times \text{ncols}$ (i.e. the no. of elements)
- The cell (i,j) is stored at the location
 $\underline{\text{data} + (j * \text{nrows}) + i}$ $(j-1)$ instead of j

This indicates pointer to the element which can be found on this position ($\text{data} + \text{something}$).

- Matrix.h

pragma once

Class Matrix

{ private:

int nrows;

int ncols;

double * data;

double * endPointers;

public:

→ int nRows() const
 { return nrows; }

// accessor function to return no. of rows.

→ int nCols() const
 { return ncols; }

// accessor fn to return no. of columns.

→ These two functions are inline functions when we write the definition of function in the declaration of class in header file rather than CPP file. The inline functions are ones which are called very often & have small code so, it's much efficient to declare them inline.

double get(int i, int j) const Another accessor fn get. This is a function that returns $(i,j)^{\text{th}}$ element of matrix found on location $(i,j)^{\text{th}}$ in Matrix.

{ return data[Offset(i,j)]; }

offset(i,j) → returns the index $\text{data} + (j * \text{nrows}) + i$.
 $(j-1)$?

Void set(int i, int j, double value) // Accessor fn that sets value.
 { data[Offset(i,j)] = value; }

```

int offset (int i, int j) const
{
    assert (i >= 0 && i < nrows && j < ncols && j >= 0);
    return j * nrows + i; // offset function that returns index of (i,j)
                           // element.
}

```

// Constructors

```

Matrix::Matrix (int nrows, int ncols) : nrows(nrows), ncols(ncols)
{
    int size = nrows * ncols;
    data = new double [size]; // allocate memory dynamically using new.
    endPoint = data + size;
    memset (data, 0, sizeof (double)* size); // initialise all members
}

```

of array with value zero using function
memset() which is optimised low
level f'm that is faster than
looping.

// Destructor

```

~Matrix () { delete [] data; }

```

Since, the destructor is Non-trivial,
By Rule 3, we have to write:

- { - Copy constructor.
- Assignment operator.
- Destructor.

// Copy Constructor.

```

Matrix::Matrix (const Matrix & m)

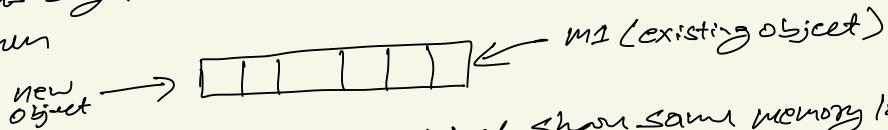
```

```

{
    nrows = m.nrows;
    ncols = m.ncols;
    int size = nrows * ncols;
    data = new double [size]; // allocate memory for new object I'm creating.
    endPoint = data + size;
    for (int i=0; i < nrows; i++)
        for (int j=0; j < ncols; j++)
            data [offset(i,j)] = m.data [offset(i,j)];
}

```

The Default Copy Constructor given by Compiler (in case if you don't write your own copy constructor) will not create an space in memory for new object which is created. The new object will share the same memory location as the object that is passed as parameter. lets say I have existing object M1 & I create a new object then



Both M1 & new object share same memory location.

The problem in this case is when each object goes out of scope then the destructor is called. So, we try to release the memory twice (the same memory location to be released twice) which leads to error. This is a problem when you have a Non-trivial destructor with using Default Copy Constructors. The same problem exist with the assignment operator. For this reason, we have one of three.

• Overloading Operators

2 types of subscript operators:-

- (i) Returns a value that can be modified (can be used on the L.H.S. of an assignment operator).
- (ii) Returns a value that can't be modified (can only be on R.H.S. of an assignment operator).

(i) **double & operator (int i, int j)** // When you want to modify then need to return a reference.
{return data[offset(i,j)];}
3

so, doing this allows us to do:

Matrix m(3,6); // matrix declared with 3 rows & 6 columns using constructor with parameter.

m(1,2) = 3 // same as doing M.operator (1,2) = 3.
we can change the return value because I returned a reference. Notice, the return value is on the L.H.S of the assignment operator.

(ii) `const double operator()(int i, int j) const`
`{ return data[Offset(i,j)]; }`

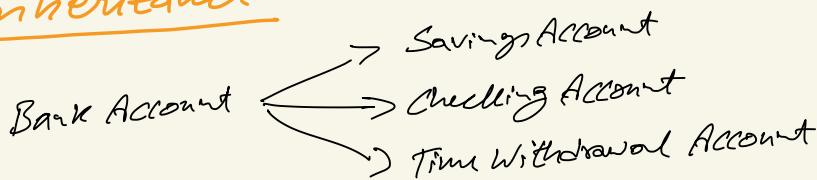
// This operator can only be used with
 const objects whose value I can't
 modify.

So, doing this can only allow us to do:

`const Matrix M2(3,8);`

`cout << M2(1,2);` // This is same as calling the second
 version operator `M2.operator()(1,2)`

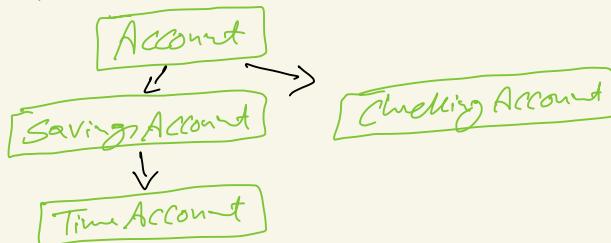
Inheritance



We want to write programs which have good design & flexible.
 i.e. if we want to add new functionality, we should be able to do so
 with minor changes to existing code. The main idea is we don't
 want to repeat code.

Questions:

Identify the member variables/functions which are common
 to all classes & those which are specific to a given class.
 Once this is done we construct hierarchy of classes, where
 the common member variables/functions are in the base class
 (Parent class) and the specialised ones are placed in derived
 class (Child class).



Class C1 is a Class C2 → (Inheritance)

Composition (Class C1 has a class C2)

Example: \circ Class Circle which uses Class Point to define some member variables. Class point which is defined is used to define a point in space with 2 coordinates. Then Class Circle has two member variables, Point C representing the center & a radius.

Class Point

{...}

Class Circle

{ Private:

 Point C; //center of circle.

 double radius;

 public:

 ...

}

This is very different to inheritance. Here, we use class point in the implementation of class circle but I don't inherit circle from the point as circle is not a point.
Don't confuse inheritance with composition as they are two different relationships between classes.

- Inheritance is extremely useful in order to build on a notion of Polymorphism which is very powerful when we do applications to Finance.

What have we learnt?

Classes

Study case: Complex class

- Default Constructor:

```
Complex();
```

- Constructor with parameters

```
Complex(double re, double im);
```

- Copy constructor

```
Complex(const Complex &c);
```

- Destructor

```
~Complex() {};
```

What have we learnt?

- **Pointer `this`:** pointer to the current instance of the class.

```
Complex::Complex(double re, double im)
{
    this->re=re;
    this->im=im;
}
```

or

```
Complex::Complex(double re, double im)
{
    (*this).re=re;
    (*this).im=im;
}
```

What have we learnt?

- **Static members:** Static member variables and static functions are associated with the class, not with a particular instance. For example, if you want to number the instances of a class, you could use a static member variable to keep track of this number.

What have we learnt?

- The **const** concept

A **const member function** is a member function that guarantees it will not modify the object.

As we have seen, to make a member function **const**, we simply append the **const** keyword to the function prototype, after the parameter list, but before the function body.

```
double Complex::Get_Re() const { return re; }
```

Const objects can only call **const** functions!

What have we learnt?

- **Overloading operators**

A programmer can provide his own operator to a class by overloading the build-in operator to perform some specific computation when the operator is used on objects of that class.

We have seen several examples. It is suggested to:

- Declare the operators `=`, `+=`, `*`, `=`, `/` and the unary operators as member functions of the class.
- Declare the operators `+`, `*`, `/`, `==` as non-member functions of the class.
- Declare the output operator `<<` as a non-member function of the class.

What have we learnt?

- **Working with several files**

- A header file (e.g. Complex.h) which starts with pragma once ;
- 2 source files (e.g. Complex.cpp and main.cpp) which starts with include"Complex.h";
- Don't put using namespace std in the header file;
- Don't define non-inline functions in the header file.

Plan

- Classes with Pointer Data Member
- Study case: YourVector class
- Some useful C++ classes

Classes with Pointer Data Member

Classes with Pointer Data members

Classes with Pointer Data Member

Every class that has a pointer data member should include the following member functions:

- a destructor
- a copy constructor
- operator= (assignment)

Classes with Pointer Data Member

The rule of three: Whenever you write a destructor (other than an empty destructor) you must:

- overload the assignment operator (=)
- write a copy constructor

In fact, if you write any one of these three things:

- a non-trivial destructor;
- a copy constructor;
- an assignment operator =;

then **you should write all three.**

Classes with Pointer Data Member

Consider the following example:

```
class Table{  
    int* Name;  
    int size;  
  
public:  
    Table ();  
    Table (int size);  
    ~Table();  
}
```

Classes with Pointer Data Member

Constructor by default

```
Table::Table ()  
{ this->size=0;  
Name=NULL; } ;
```

Constructor with parameters

```
Table::Table (int size)  
{ this->size=size;  
Name=new int [size]; }  
}
```

Classes with Pointer Data Member

Non-trivial Destructor Function

```
Table::~Table () { delete [ ] Name; }
```

The purpose of the destructor is to free any dynamically allocated storage.

Note that an object's destructor is called when that object is about to "go away"; i.e. when:

- A class object (a value parameter or a local variable) goes out of scope
- A pointer to a class object is deleted (the dynamically allocated storage pointed to by the pointer is freed by the programmer using the **delete** operator).

Classes with Pointer Data Member

Example

```
void f(Table T) {
    Table *p= new Table;
    while(...) {
        Table T1;
        ...
    }
    delete p;
}
```

In this example, the scope of value parameter T is the whole function; T goes out of scope at the end of the function. So when function f ends, T 's destructor function is called.

Classes with Pointer Data Member

The scope of variable *T1* is the body of the while loop.

- **T1's constructor** function is called at the beginning of every iteration of the loop
- **The destructor function** is called at the end of every iteration of the loop.

Variable *p* is a pointer to a *Table*. When a *Table* is allocated using **new**, that object's constructor is called. When the storage is freed, the object's destructor function is called (and then the memory for the Table itself is freed).

Classes with Pointer Data Member

The copy constructor

In the case of a class a **non-trivial destructor**, if you don't write a copy constructor, you may have problems!

Explanation

If you don't write a copy constructor, the compiler will provide one that just copies the value of each data member. If some data member is a pointer this causes *aliasing* (both the original pointer and the copy point to the same location), and may lead to trouble.

Classes with Pointer Data Member

Example

```
void h()
{
    Table t1;
    Table t2=t1;
}
```

The problem

- The **constructor by default** of Table is called once, for **t1**.
- **The object t2 is created by copy**, using the copy constructor provided by the compiler.
- The destructor is called two times, for t1 and t2. As t1.Name and t2.Name point to the same memory location, we try to free twice the same storage! ⇒ **Problem!**

Classes with Pointer Data Member

Conclusion: You have to define your own copy constructor in order to avoid the previous problem!

How should we define the copy constructor?

The copy constructor should perform the following tasks:

- Initialize the *size* field to have the same values as the one in *T.size* (where *T* is the copy constructor's Table parameter)
- Allocate a new array of int of size *T.size* ; we set *Name* to point to this new array
- Copy the values in the array pointed to by *T.Name* to the new array

Classes with Pointer Data Member

Overloading the operator =

As we have seen, in C++ you can assign from one class object to another (of the same type). For example,

```
Table T1, T2;  
T1=T2; // this assignment is OK
```

By default, class assignment is field-by-field assignment. This assignment is equivalent to:

```
T1.size=T2.size;  
T1.Name=T2.Name;
```

Classes with Pointer Data Member

Overloading the operator =

Problem: the same as in the case of a "default" copy constructor: the destructors of T_1 and T_2 are called; we try to free twice the same memory location.

Solution: Write in the class your own assignment operator!

Classes with Pointer Data Member

Some differences between the operator = and the copy constructor

- The object being assigned to has already been initialized.
- It is possible for a programmer to assign from a variable into *itself*, for example $T1 = T1$. The operator $=$ must check for this and do nothing.
- The operator $=$ function returns a reference to an object of type *Table* (this is important in order to write $(T1 = T2) = T3$). The copy constructor has no return value (or reference).

Case Study: YourVector Class (class with
non-trivial destructor)

Case Study: Array Class

```
class YourVector{    // declaration of Class.  
private:  
    int size; // the size  
    int* Name; // pointer to the first element  
    of pointer-based array
```

Case Study: YourVector Class

public:

```
int GetNaml(int i) const; // return element.
```

YourVector(); //default constructor

YourVector(**int** size); // constructor with parameter

YourVector(**const** YourVector&); //copy constructor;

~YourVector(); // destructor

int getsize() **const**; //return size;

YourVector& **operator=** (**const** YourVector &); // assignment operator which returns reference to vector.

// subscript operator for non-const objects return modifiable lvalue

int& **operator** [] (**int**);

// subscript operator for const objects returns rvalue; Values cannot be modified.

const **int** **operator** [] (**int**) **const**;

GetNaml() always here

Case Study: YourVector Class

```
// Non-member operators
// equality operator
    bool operator==(const YourVector& left,
const YourVector& right); // Checks if two vectors are equal.

// inequality operator; returns opposite of ==
    bool operator!=(const YourVector& left,
const YourVector& right);

// output operator
    std::ostream& operator<< (std::ostream&
out, const YourVector& v);

};
```

Case Study: YourVector Class

It is essential for copy constructors to use references in parameters otherwise parameter is passed by value, so in the function I would work with copy of parameter leading to an loop.

1. Constructor by copy Function.

```
YourVector::YourVector(const YourVector& VectorToCopy) : size(VectorToCopy.size) {  
    Name = new int [size]; // Allocates Dynamically some memory to store values.  
    for (int i=0; i<size; i++)  
        Name[i] = VectorToCopy.Name[i]; // Initialise ith element  
    }  
    of vector with ith element of the object which is passed as parameter.
```

Point to
1st element

Name of class followed by resolution operator name of function. In brackets I have reference to object of type YourVector which is const. **:size (vectorToCopy.size)** → To initialise the size of the new object that I am creating with the size of object which is passed as parameter.

Case Study: YourVector Class

In last page, if we didn't write our own copy constructor & the compiler provided the default one, it would not work correctly in that case as it will create new object & both of them would have their Name pointing to same memory location. So, both object would share same memory location. This would be problem when object go out of scope & memory would be released with destructor, the same memory location would be released twice which doesn't make sense.

2. Non-trivial Destructor

```
YourVector::~YourVector()
{ delete [] Name; // release pointer-based
  array space
}
```

Name of class Resolution operator Tilde function name

Case Study: YourVector Class

(About Assignment Operator)

Before presenting the overloaded operators, we explain two concepts from programming language called *lvalues* and *rvalues*.

- *Lvalue* is a value that resides in memory and is addressable (you can take its address). It stands for "left value" and it is a value that can be on the left-hand side of an assignment.
- *Rvalue* is a value that' not lvalue. It stands for "right value" and it is a value that can only be on the right-hand side of an assignment (the rvalue is intended to be non-modifiable).

Case Study: YourVector Class

For example, in the statement

```
x=5;
```

x is an lvalue and 5 is an rvalue.

In the statement

```
x=y+z;
```

x, y, z are lvalues , but $y + z$ is an rvalue (result of operator+).

Case Study: YourVector Class

3. Overloading the assignment operator

(Function returns reference to yourVector) (using class)

```
YourVector& YourVector::operator=(const
    YourVector& right)
{ if (&right!=this) //avoid self-assignment
    { // for arrays of different sizes,
      deallocate original left-side array, then
      allocate new left-side array
        if (size!=size.n)
        { delete [] Name; //release space;
          size=right.size;
          Name=new int [size];
        }
    }
}
```

`if (&right != this)` → Means if the address of the parameter right is different to this (Points to invoking object). What I am asking is does the invoking object coincide with the parameter right. If they are same we do nothing otherwise we have to define our assignment operator so we go to next if block.

`{block}`

`if (size != right.size) →` If sizes are not equal, first we release the space for the invoking object by delete keyword. Then, reinitialise size with size of parameter right. Finally, allocate memory for size elements with new keyword.

Case Study: YourVector Class

3. Overloading the assignment operator

We do this for both case if size same or different.

```
for (int i=0; i<size; i++)
{
    Name[i]=right.Name[i]; // copy
    array into object(invoking object)
}
return *this; // returning invoking object.
```

We are copying element by element from the vector right to the vector which I want to assign the value i.e. the invoking object.

Case Study: YourVector Class

Notice that **operator=** returns a reference to `*this`.
Which is the benefit?

```
YourVector t1(3); // Create vector object t1 of size 3 i.e. 3 elements, by  
YourVector t2(3); calling constructor with parameters.  
YourVector t3(3);
```

Assume that we initialize the values of `t1.Name` with 1, the elements of `t2.Name` with 2 and the elements of `t3.Name` with 3.
If we write

```
(t1=t2)=t3, // Benefit is to do several assignment in a chain.
```

then `t1.Name` holds the values of `t3.Name`.

Case Study: YourVector Class

Remark: We have respected the rule of three: we wrote a copy constructor, the destructor and the assignment operator.

Rule of three: You have to write the destructor, the copy constructor and the assignment operator. The copy constructor and the assignment operator provided by default by the compiler are **not OK!**

Case Study: YourVector Class

4. Operator == which determines if two vectors are equal

```
bool operator== (const YourVector& left, const
                  YourVector& right)
{
    if (left.getsize()!=right.getsize())
        return false;

    for (int i=0; i<size; i++)
        if (left.getName(i)!=right.getName(i))
            return false;
    return true;
}
```

Case Study: YourVector Class

5. Inequality operator; returns opposite of ==

```
bool operator!=(const YourVector& left,  
const YourVector& right)  
{ return !(left==right); };
```

Case Study: YourVector Class

We now overload the subscript operator (as **member of the class**) in two ways.

Case Study: YourVector Class

(Case when [] operator returns value that can be modified)

6. Overloading subscript operator 1

```
int & YourVector:: operator [](int subscript) {  
    // check for subscript out-of-range error  
    if (subscript<0 || subscript >=size)  
    {    std::cerr<<"\nError: Subscript  
" << subscript << "out of range" << std::endl;  
        exit(1); // terminate program;  
    subscript out of range  
    } // end if  
    return Name[subscript]; // reference  
    return; // i.e. return subscript element of Name reference. returning reference  
    helps us to use this operator to change the value through return value.  
};
```

Return referent to just followed by class name, resolution operator, keyword operator, [] & in brackets the parameter Subscript which is the element we want to use.

Case Study: YourVector Class

6. Overloading subscript operator 1

Remark:

The reference creates a modifiable lvalue. More precisely, one can write

```
YoursVector t1(3);  
t1[2]=5; //initialise 2nd element of vector t1 with 5.
```

t1[2] means we are calling [] function (previous page) where the parameter subscript = 2.

Case Study: YourVector Class

(Case when [] operator returns value that cannot be modified)

7. Overloading subscript operator 2

```
const int YourVector:: operator [](int subscript)
{
    // check for subscript out-of-range error
    if (subscript<0 || subscript >=n)
        { std::cerr<<"\nError: Subscript
" <<subscript <<"out of range"<<std::endl;
         exit(1); // terminate program;
    subscript out of range
    } // end if

    return Name[subscript];
};
```

The rest of the code is exactly same as the previous function except for the 1st line.

Case Study: YourVector Class

7. Overloading subscript operator 2

Remark:

- Using this subscript operator you can not write:

```
YourVector t1(3);  
t1[2]=5;
```

You obtain the following error: "Expression is not assignable".

- This subscript operator is used with const objects (*const YourVector t*).

Case Study: YourVector Class

8. Overloading the output operator <<

```
std::ostream & operator<< (std::ostream &
    output, const YourVector& v)
{
    for (int i=0; i<v.getsize();i++)
        std::cout<< v[i];
    return output; // enables cout<<x<<y
}
```

Case Study: YourVector Class

How to use YourVectorClass?

Case Study: YourVector Class

```
YourVector integers1();  
YourVector integers2(10);  
YourVector integers3=integers2;
```

- We instantiate two objects of class YourVector: integers1 with 7 elements, integers2 with 10 elements.

Case Study: YourVector Class

```
for (int i=0; i<=integers1.getsize(); i++)
integers1[i]=i;
for (int i=0; i<=integers2.getsize(); i++)
integers2[i]=2*i;
```

- We use the member function `getsize()` in order to obtain the size of the vector.
- In order to initialize the values of the vector `integers1` and `integers2`, it is used the first overloaded subscript operator.

Case Study: YourVector Class

```
if ( integers1 != integers2 )  
    cout << "integers1 and integers2 are  
not equal" << endl;
```

- We test the overloaded inequality operator by evaluating the condition.

Case Study: YourVector Class

```
Vector integers3( integers1 );
```

- This line instantiates a third Vector called integers3 and initializes it with a copy of YourVector integers1. This invokes the YourVector copy constructor to copy the elements of integers1 into integers3. The copy constructor can also be invoked by writing

```
YourVector integers3=integers1;
```

- The equal sign in the preceding statement is **not** the assignment operator. When an equal sign appears in the declaration of an object, it invokes a constructor for that object!

Case Study: YourVector Class

Some final remarks on the copy constructor

- We declare a copy constructor that initializes YourVector by making a copy of an existing YourVector object. As we have said, *such copying must be done carefully to avoid the pitfall of leaving both Vector objects pointing to same dynamically allocated memory. This is exactly the problem that occurs with default memberwise copying, if the compiler is allowed to define a default copy constructor for this class.*

Case Study: YourVector Class

Some final remarks on the copy constructor

- Note that a copy constructor must receive its argument by reference, not by value. Otherwise, the copy constructor call results in infinite recursion (a fatal logic error) because receiving an object by value requires the copy constructor to make a copy of the argument object. Recall that any time a copy of an object is required, the class's copy constructor is called. If the copy constructor received its argument by value, the copy constructor would call itself recursively to make a copy of its argument!

Using some useful C++ classes

- C++ Standard Library provides some classes, which can be used including the corresponding header files (we have already included <iostream> in our programs in order to use *cin* and *cout*).

Using some useful C++ classes

- include <vector> to work with vectors.

In your programs, you should use this class instead of creating your own vector class. We have learnt to write YourVector class in order to learn how to manipulate classes with non-trivial destructors!

- include <string> to use strings.
- include <sstream> to use strings efficiently.
- include <fstream> to work with files.

Using some useful C++ classes

- Matrices? Sorry, you have to write your own! The design of the class is very similar to *YourVector* class (you have to deal with the same non-trivial destructor and the rule of three!)

Using some useful C++ classes

Class Vector from the C++ Standard Library

```
// create a vector
vector<double> myVector;

// add three elements to the end
myVector.push_back( 12.0 );
myVector.push_back( 13.0 );
myVector.push_back( 14.0 );

// read the first, second and third elements
cout << myVector[0] << "\n";
cout << myVector[1] << "\n";
cout << myVector[2] << "\n";
```

Using some useful C++ classes

Class Vector from the C++ Standard Library

```
// change the values of a vector
myVector[0] = 0.1;
myVector[1] = 0.2;
myVector[2] = 0.3;

// loop through a vector
int n = myVector.size();
for (int i=0; i<n; i++) {
    cout << myVector[i] <<"\n";
}
```

Using some useful C++ classes

Class Vector from the C++ Standard Library

```
// Create a vector of length 10
// consisting entirely of 3.0's
vector<double> ten3s(10, 3.0 );

// Create a vector which is a copy of another
vector<double> copy( ten3s );

// replace it with myVector
copy = myVector;
```

Using some useful C++ classes

Class Vector from the C++ Standard Library

```
// The function size gives the number of  
elements  
cout<<ten3s.size();
```

Using some useful C++ classes

Class Vector from the C++ Standard Library

1. Some useful functions of the class Vector

```
push_back // add element at the end  
size() // return size  
resize() // change size  
operator [] // access element
```

Using some useful C++ classes

Class Vector from the C++ Standard Library

Use of resize:

```
myVector.resize(5); // resize the vector
// at this point, the vector, the vector
contains
// 0.1, 0.2, 0.3, 0,0

myVector.resize(2); // resize the vector
// at this point, the vector, the vector
contains
// 0.1, 0.2
```

Using some useful C++ classes

Class Vector from the C++ Standard Library

2. Array/vector

Just **like** arrays, vectors use contiguous storage locations for their elements. Vector combines the advantages of both static and dynamic arrays: the size can change dynamically as in the case of dynamic arrays (internally, vectors use a dynamically allocated array to store their elements) and the used memory is automatically deleted as in the case of static arrays.

Using some useful C++ classes

Passing big objects around When you write a function that takes a vector parameter you should write it like this:

```
double sum( const vector<double>& v ) {  
    double total = 0.0;  
    int n = v.size();  
    for (int i=0; i<n; i++) {  
        total += v[i];  
    }  
    return total;  
}
```

- **Rule:** Use `const` and `&` symbols. We need to pass the vector by **reference** because vectors are too big to keep copying all the time.

Remark

- For very small data types (double , int , bool), pass by value is quicker.
- For big data types, pass by reference is quicker.

Using some useful C++ classes

How to write to files?

```
// create an ofstream
ofstream out;

// choose where to write
out.open("myfile.txt");

out << "The first line\n";
out << "The second line\n";
out << "The third line\n";

// always close when you are finished
out.close();
```

Works just like `std::cout` **except** for the **open and closing**.

Using some useful C++ classes

Passing a stream as a parameter Pass a reference to an ostream .

```
void writeHaiku( ostream& out ) {  
    out << "The wren\n";  
    out << "Earns his living\n";  
    out << "Noiselessly.\n";  
}
```

Using some useful C++ classes

Passing a stream as a parameter

```
void testWriteHaiku() {
    // write a Haiku to cout
    writeHaiku( cout );
    // write a Haiku to a file
    ofstream out;
    out.open("haiku.txt");
    writeHaiku( out );
    out.close();
}
```

Why can we do this? Because an `ofstream` is an `ostream`.

Using some useful C++ classes

Working with strings

```
// Create a string
string s("Some text.");
// Write it to a stream
cout << s << "\n";
cout << "Contains "
    << s.size() <<
    " characters \n";
```

Using some useful C++ classes

Working with strings

```
// Change it
s.insert( 5, "more ");
cout << s <<"\n";

// Append to it with +
s += " Yet more text.";
cout << s <<"\n";
// Test equality
if ( s=="Some more text. Yet more text.");
```

Using some useful C++ classes

Technical points about strings

- Using a **string** is better than using a **char*** (C style strings) because they have lots of helpful functions!

Using some useful C++ classes

Working with strings efficiently

Using `+` to build up strings is slow. Don't do this:

```
string s("");
for (int i=0; i<100; i++) {
    s+="blah ";
}
cout << s << "\n";
```

Using some useful C++ classes

Working with strings efficiently Do this:

```
stringstream ss;
for (int i=0; i<100; i++) {
    ss<<"blah ";
}
string s1 =ss.str();
cout << s1 <<"\n";
```

You have to use `#include <sstream>`. A `stringstream` is an `ostream`.

Summing up

We have learnt about:

- Classes with pointer data member variables and the rule of three
- We have studied the class YourVector, which simulates the Vector of the Standard Library
- We have learnt about several useful classes of the Standard Library.

Lecture 7

C++ in Financial Mathematics

What have we learnt?

Classes with non-trivial destructor

Study case: YourVector class

```
class YourVector{
int size;
int* Name;

public: YourVector(); //default constructor
YourVector(int size); //constructor with
parameters
YourVector(const YourVector&); //copy
constructor
~YourVector(); //destructor
YourVector& operator=(const YourVector& v);
//assignment operator
```

What have we learnt?

Classes with non-trivial destructor

Study case: YourVector class

Rule of three: You have to write the destructor, the copy constructor and the assignment operator. The copy constructor and the assignment operator provided by default by the compiler are **not OK!**

Remark: We have seen during the practical a more complex class YourVector which has an additional member variable **capacity**. A benefit of this class is represented by the fact that we can write an efficient function **push_back()**.

Plan

- Inheritance
- Polymorphism

Plan

Inheritance

(2nd main concept relating to OOP.)

Recall, 1st main concept was Encapsulation.

Inheritance

Motivating example

- Consider different types of bank accounts
 - Savings accounts
 - Checking accounts (*Current Account*)
 - Time withdrawal accounts, which are like saving accounts, except that only the interest can be withdrawn
 - If you were designing C++ classes to represent each of these, what **member functions might be repeated** among the different classes? What **member functions would be unique** to a given class?

Question to ask. S → which are common member variables & functions to all Accounts.
L → " " " Specific " " " " " given Account / class.

Inheritance

Motivating example

- To avoid repeating common member functions and member variables, we will create a **class hierarchy**, where the common member functions and variables are placed in a **base class** and specialized ones are placed in **derived classes**.

Inheritance

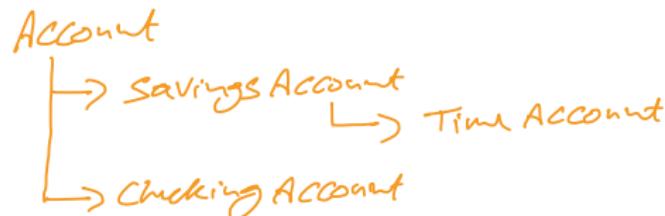
Accounts Hierarchy We will use the accounts class hierarchy as a working example.

- **Account** is the *base class* of the hierarchy.
- **SavingsAccount** is a *derived* class from **Account**.
SavingsAccount has inherited member variables and functions and ordinarily-defined member variables and functions.
- The member variable **balance** in base class **Account** is **protected**, which means:
 - **balance** is not publicly accessible outside the class, but it is accessible in the derived classes.
 - if **balance** was declared as **private**, then **SavingsAccount** member functions could not access it.

(Protected is between Private & Public)

Inheritance

- When using objects of type **SavingsAccount** the inherited and derived members are treated exactly the same and are not distinguishable.
- **CheckingAccount** is also derived from base class **Account**.
- **TimeAccount** is derived from **SavingsAccount**. **SavingsAccount** is its base class and **Account** is its indirect base class.



Inheritance

The Accounts Class Hierarchy

Base class: Account

```
class Account{  
protected:  
double balance; //account balance is protected member variable.  
public:  
Account(): balance(0.0){}; //Default constructor sets balance to zero.  
Account(double bal): balance(bal){}; //Constructor with parameter bal,  
//which initializes balance with parameter bal.  
void deposit(double amt){balance+=amt;}; //deposit function  
double get_balance() const {return balance;}; //Accessor  
//function  
};
```

Inheritance

The Accounts Class Hierarchy

Derived class: SavingsAccount from the Base class

Account

Syntax to say Savings Account is derived from Account.
Class derived class name : public BaseClassName

```
class SavingsAccount: public Account {  
protected:  
    double rate; // periodic interest rate. Specific member  
public:  
    SavingsAccount () : Account () {rate=0.0;} // Default constructor  
    SavingsAccount (double bal, double rate) :  
        Account (bal) {this->rate=rate;} // Constructor with parameters.  
    double compound() { // deposit interest // Specific function  
        double interest=balance*rate; // Compute interest.  
        balance+=interest; // increase balance by interest.  
        return interest;  
    }  
}
```

Inheritance

The Accounts Class Hierarchy

Derived class: SavingsAccount from the Base class Account

function specific to SavingsAccount class.

```
double withdraw(double amt) { // if overdraft
    return 0, else return amount
if (amt>balance) { return 0.0; }
else {
    balance-=amt; // balance update after withdrawing money.
    return amt;
}
};
```

• Derived Class: Savings Account

Class SavingsAccount : Public Account → This is the syntax to say SavingsAccount is derived from base class Account.
We need to write : Public Account to say the type of inheritance is public.

There are several types of inheritance: public, Protected & private.
In this course, we will only use public inheritance as this is mostly used in general.

To write constructor of derived class, we also have to call constructor of the base class. The syntax for eg is:

SavingsAccount() : Account() { rate = 0.0; } → This default constructor first initialises common variable balance then initialise specific member variable to SavingsAccount rate to 0.0.

SavingsAccount(double bal, double rate) : Account(bal) { this->rate = rate; }
This is a constructor with two parameters. Works same way as before, first calls constructor corresponding to base class Account which initialises balance with bal then initialise variable rate with rate. Since, both parameter and member variable has same name rate, we need to use pointer using this->rate.

Inheritance

The Accounts Class Hierarchy

Derived class: CheckingAccount from the Base class
Account

```
class CheckingAccount: public Account{
protected:
double limit; //lower limit for free checking
double charge; //per check charge
public:
CheckingAccount(): Account() { limit= 0. ;
    charge= 0. ; } //default constructor.
CheckingAccount(double bal, double lim ,
    double chg): Account(bal), limit(lim),
    charge(chg) {} //constructor with parameters.
```

Inheritance

Function that allows us to withdraw cash using check
from checking Account.

```
double cash_check(double amt) {
    assert(amt>0); // assert function used to check validity of parameter.
    if (balance<limit && (amt+charge<=balance)) {
        balance-=amt+charge;
        return amt;
    }
    else if (balance>=limit && amt<=balance) {
        balance-=amt;
        return amt;
    } else { return 0.0; } } // If both above if,else are not
                           satisfied we return 0.
```

Assertions in C++

We use assert functionality in code to test assumptions made by programmer & mostly to check if parameters passed to functions are valid. To use assert functionality we must use header file

`#include <cassert>`

Assertions makes debugging easy & efficient but should not be included in the release version of software. For release version, we can disable assertion using NDEBUG macro.

`#define NDEBUG`

There are two types of assert:

(i) assert

This is executed at run time. The syntax is:

`assert(expression);`

If the expression evaluates to true, program continues. But if it evaluates to false, an error message issued along with the program name & line number where it happened & the program terminates.

(ii) static_assert

This is executed at compiled time. The syntax is:

`static_assert(expression, "message");`

If the expression evaluates true, the program continues. But if it evaluates false, a compiler error is issued & the message that was passed as parameter will be displayed & the program terminates.

Inheritance

The Accounts Class Hierarchy

Derived class: TimeAccount from the Base class
SavingsAccount

```
class TimeAccount: public SavingsAccount{
    protected:
double funds_avail; //amount available for
                     withdrawal // this variable is protected.
public:
TimeAccount () :
    SavingsAccount () {funds_avail=0.0;};//Default constructor.
TimeAccount (double bal, double rate):
    SavingsAccount (bal, rate),
    funds_avail(0.0) {};//Constructor with parameter.
// redefines 2 member functions from
SavingsAccount
```

Inheritance

```
double compound() {  
    double interest=SavingsAccount::compound(); //  
    funds_avail+=interest; // Specific member variable of TimeAccount  
    class funds_avail increases by the interest.  
    return interest; }  
  
double withdraw(double amt) {  
    if (amt<=funds_avail) {  
        funds_avail-=amt;  
        balance-=amt;  
        return amt;  
    } else { return 0.0; }  
}  
  
double get_avail() const {return funds_avail;} //  
// accessor function Specific to TimeAccount class  
};
```

↑ Compound() & withdraw() function from SavingsAccount redined

↑ // Member variable interest equal value returned by the function compound() from parent class SavingsAccount.

↑ // accessor function Specific to TimeAccount class

Inheritance

Constructors and Destructors

- **Constructors** of a derived class *call the base class constructor* immediately, **before** doing **anything** else. The only thing you can control is which constructor is called and what the arguments will be.

When a **TimeAccount** is created 3 constructors are called in the following order:

- The **Account** constructor
- The **SavingsAccount** constructor
- The **TimeAccount** constructor.

Inheritance

Constructors and Destructors

- The **reverse is true for destructors**: derived class destructors do their job first and then base class destructors are called automatically.

In our particular example: when an object of the class **TimeAccount** goes out of scope, the destructors are called in the following order:

- The **TimeAccount** destructor.
- The **SavingsAccount** destructor.
- The **Account** destructor.

Inheritance

Overriding Members

- A derived class can redefine member functions in the base class. The function prototype must be identical, not even the use of **const** can be different (otherwise both functions will be accessible).
- For example, in the class **TimeAccount** we have **TimeAccount::compound** and **TimeAccount::withdraw**
- Once a function is redefined it is not possible to call the base class function, unless it is explicitly called as in **SavingsAccount::compound**.

Inheritance

Overriding Members

Example

```
TimeAccount obj; // define object of TimeAccount class.  
cout << obj.Compound(); // calls the function  
// compound defined in the class TimeAccount;  
cout << obj.SavingsAccount::Compound(); //  
// calls the function compound defined in the  
// class SavingsAccount;
```

Inheritance

Public Inheritance

- Note the line **class Savings_Account: public Account**
This specifies that the member functions and variables from **Account** do not change their *public*, *protected* or *private* status in SavingsAccount. This is called *public inheritance*.

There also exist:

- **protected inheritance**: public members in the base class become protected and other members are unchanged.
- **private inheritance**: all members become private.

In this course, we'll only use public inheritance.

Inheritance

The most important aspect of **Inheritance** is the **relationship expressed between the new class and the base class.**

After encapsulation, **Inheritance** is the second essential feature of an object-oriented programming language.

Inheritance

Relationships Among Classes

- If "**C1 is a C2**" class, then C1 should be a derived class (a subclass) of C2. For example, "a savings account is an account". This relationship corresponds in our case to the **public inheritance**.
- If "**C1 has a C2**", then class C1 should have a member variable of type C2. For example, "a cylinder has a circle as its base". Or "a circle **has** a Point as its center". This relationship corresponds to a design technique called **composition**. *(member variable of class cylinder will be of class Point)*
- In the case of "**C1 is implemented as a C2**", then C1 should be derived from C2, but with **private inheritance**. For example, "the stack is implemented as a list". *(stack will derive private inheritance from list)*

Inheritance

Relationships Among Classes: Composition/Inheritance

Inheritance and composition allow to create a new type from existing types, and both embed subobjects of the existing types inside the new type. The main difference between them is the following:

- You use composition to reuse existing types as part of the underlying implementation of the new type.
- You use inheritance when you want to force the new type to be the same type as the base class.

Some Useful C++ Classes

① The Vector Class

```
#include <vector>
```

```
vector<double> myVector; // declaring myVector which is  
object of Vector class of type double.
```

↑
Point ← live in vector class.
int
char
char*
Point*

You can have any type you want even user defined type.
This is achieved in C++ using templates.

The main benefit of class Vector is you can work with it very easily as you do with arrays but it also has the benefit of Dynamic Allocation which allows us to change size of vector but in arrays the size is fixed. Vector also has function that gives you the size.

// Function that adds elements at the end of vector (push-back)

```
myVector.push_back(12.0);  
myVector.push_back(5.0);  
myVector.push_back(3.0);  
cout << myVector[0] << " "; // prints 12.0.  
cout << myVector[1] << " "; // prints 5.0  
cout << myVector[2] << " "; // prints 3.0
```

// Using version of subscript operator which returns value that can be modified.

```
myVector[0] = 0.1;  
myVector[1] = 0.2;  
myVector[2] = 0.3;
```

// Looping through a vector.

```
int n = myVector.size(); // size() function gives size of vector.  
for (int i=0; i < n; i++)  
{ cout << myVector[i] << " "; }
```

```

// Constructors.
vector<double> vector20(20, 1.0); // declaring vector of size 20 with
                                         all elements equal to 1.0.

vector<double> copy(vector20); // Creating another vector by using copy
                                constructor.

copy = myVector; // assignment operator used to assign myVector to copy.

// Using resize() function
myVector.resize(5); // resize length of myVector & contain only 1st 3 elements.
                     If it has less than 5 elements then adds some zeros.
                     Eg: 0.1, 0.2, 0.3, 0, 0.

myVector.resize(2); // 0.1, 0.2.

```

Example: A function which computes the sum of elements of a vector passed as parameter.

[Note: When passing big objects as parameters, we should use reference symbols as this avoids making copies. Making copies of big object inside a function creates nightmares in terms of space and time.]

```

double sum(const vector<double>& v) // Function sum takes
                                         parameter v which contains elements
                                         of type double.

{ double total = 0.0;
  int n = v.size();
  for (int i = 0; i < n; i++)
    { total += v[i]; }
  return total;
}

```

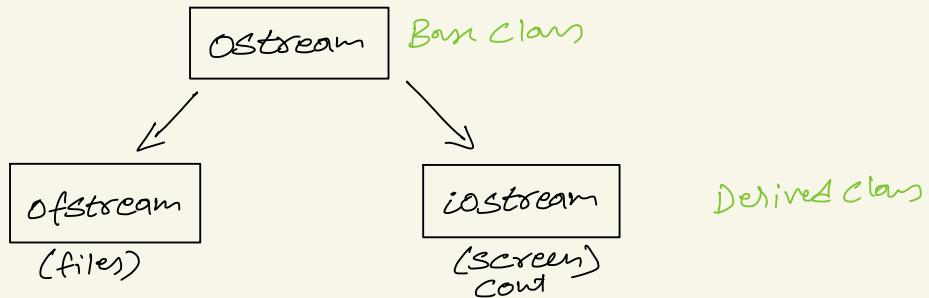
Notice, we only used one parameter & didn't require another parameter of size as before. This demonstrates why the notion of class is so important.

Rule: In a function, for big objects use reference to pass as parameter to avoid copying. For small objects like double, bool, char, int etc passing by value is quicker.

② How to write to files?

ofstream out; // This object is the file we write to.
out.open("myfile.txt"); // Choosing name for file to write to.
out << "Hello World"; // Writing "Hello World" in the file.
out.close(); // Closing file once finished writing.

Note, it works just like cout except for the open and closing.
ofstream is a class contained in input/output library of C++.
ofstream & ifstream are derived from Stream objects.



Passing a stream as a parameter

Void funcexample(ostream &out) // passing object of base class so
that we can use any object of derived class.
{ out << "love C++";}

3

Void USE_function()
{ funcexample(cout); // Points "love C++" on screen.
ofstream out;
out.open("test.txt"); } // Writes "love C++" into the file test.txt.
funcexample(out);
out.close(); }

3

This is the advantage of inheritance. We have one function funcexample() which either points to screen or write to file using inheritance of ifstream or ofstream from base class ostream.

We can also Read from screen or from a file.

Read from screen → cin ↗ istream

Read from file → ifstream

③ Working with Strings (C++ Version)

#include <string> // String class in C++.

// Create a string

String s ("Some text"); // declare s of type String which is initialized with "Some text".

// Write it to a stream.

cout << s << "\n"; // Overloading cout to point s to screen.

cout << s.size() << "characters";

s.insert(5, "more"); // Insert "more" on 5th position of s.

s += "More text"; // Concatenates "More text" to s.

Polymorphism

Polymorphism

Let us consider the following **Musician hierarchy**.

1. **Base class**: *Musician*.

```
class Musician{
public:
void greet(); //declaration of greet fn.
};

void Musician::greet() {cout<<"musician says:
hello\n":} //definition of greet fn.
```

Polymorphism

2. A first derived class: **Trumpeter**

```
class Trumpeter: public Musician{
public:
void greet();
};

void Trumpeter::greet() {cout<<"trumpeter
says:hello\n":} //Overriding function greet() from the
class Musician.
```

Polymorphism

3. A second derived class: Pianist

```
class Pianist: public Musician{
public:
void greet();
};

void Pianist::greet() {cout<<"pianist says:
hello\n":}
```

Polymorphism

Invoking the functions through pointers/ instances

User Part Code

```
int main() {
    Trumpeter t; // a Trumpeter instance t.
    Pianist p; // a Pianist instance P.
    Musician m, *pm; // a Musician instance m and a
                      Musician pointer *pm.
    // 1: invoking through a Musician instance
    m.greet(); // prints "musician says: hello"

    // 2: invoking through Trumpeter and Pianist
    // instances
    t.greet(); // prints "trumpeter says: hello"
    p.greet(); // prints "pianist says: hello"
}
```

Polymorphism

Invoking the functions through pointers/ instances

```
// 3: invoking through a Musician pointer on a
     Musician instance
pm=&m; //pm points to Musician. Initializer pm with address of m.
pm->greet(); // prints "musician says: hello"
// 4: invoking through a Musician pointer on a
     Trumpeter instance and on a Pianist instance
pm=&t; // points to Trumpeter
pm->greet(); // prints "musician says: hello"
pm=&p; // points to Pianist
pm->greet(); // prints "musician says:hello"
}
```

Polymorphism

Remark: Taking the address of an object of a derived class (either using a pointer or a reference) and treating it as the address of the base type is called **upcasting**

In Group 4, we declare a pointer to a **Musician**, and without complaint we can initialise it with the address of a **Trumpeter** or **Pianist** (because **Trumpeter** or **Pianist** are derived from **Musician**) (this is an example of upcasting). Note that the interface of **Musician** must exist in **Trumpeter** or **Pianist**, because **Trumpeter** and **Pianist** are publicly inherited from **Musician**.

Polymorphism

Remark 2:

- The results obtained for the first 3 Groups are as expected. In **Group 4**, invoking the function greet through a Musician pointer has in all the cases as result "musician says: hello" (even if the Musician is actually a Trumpeter or a Pianist):
the result is not the one that we expect!
- Which should be the **correct result**?
If the pointer pm points to a Musician which is a Trumpeter (as in Group 4) we should obtain as message: "Trumpeter says: hello"!, instead of "Musician says: hello"!
- Which is the **solution**?

Polymorphism

The solution is: the keyword "virtual"!

Basing the behavior of an object on its run-time is a task that C++ takes on with *virtual functions*. A virtual function is a special kind of member function. You declare it with the keyword **virtual**.

We now re-write our example of Musician hierarchy using the keyword **virtual**.

Polymorphism

1. Base class

```
class Musician{
public:
virtual void greet(); // virtual function
};

void Musician::greet() {cout<<"musician says:
hello\n":}
```

We declare **greet()** to be a virtual function.

Polymorphism

We have created a base class **Musician** with a **virtual function greet()**. Which is the next step?

- We derive the classes where we override the virtual functions to behave in ways specific to each class.

Polymorphism

2. A first derived class: Trumpeter

```
class Trumpeter: public Musician{
public:
virtual void greet(); // overriding
Musician's virtual function
};

void Trumpeter::greet() {cout<<"trumpeter
says: hello\n":}
```

Polymorphism

3. A second derived class: Pianist

```
class Pianist: public Musician{
public:
virtual void greet(); // overriding
Musician's virtual function
};

void Pianist::greet() {cout<<"pianist says:
hello\n":}
```

Polymorphism

Invoking the functions through pointers/ instances

```
int main() {
    Trumpeter t; // a Trumpeter instance
    Musician m, *pm; // a Musician instance and a
                      Musician pointer
    // 1: invoking through a Musician instance
    m.greet(); // prints "musician says: hello"
    // 2: invoking through a Trumpeter instance
    t.greet(); // prints "trumpeter says: hello"
}
```

Polymorphism

Invoking the functions through pointers/ instances

```
// 3: invoking through a Musician pointer on a
    Musician instance
pm=&m; // points to Musician
pm->greet(); // prints "musician says: hello"

// 4: invoking through a Musician pointer on a
    Trumpeter instance
pm=&t; // points to Trumpeter
pm->greet(); // prints "trumpeter says: hello"
}
```

Polymorphism

Group 1 - We call the function **greet()** on a **Musician** instance - it acts like a **Musician**

Group 2 - We call the function **greet()** on a **Trumpeter** instance - acts like a **Trumpeter**

Group 3 - We make **pm** - a **Musician** pointer -point to **m**. Invoking the member function **greet()**, we see that the object acts like a **Musician**.

Group 4 - we point **pm** at **t**. When we invoke **greet()**, we see the **Trumpeter** greeting!

Polymorphism

Difference between non-virtual and virtual functions.

- In our **first example**: the function **greet()** is **non-virtual**. In this case, the function is invoked based on the *apparent* type of the object. As the object of type **Trumpeter** is accessed through a pointer to a **Musician**, it is apparently a **Musician**; so the **Musician** greeting is invoked.
- In our **second example**: the function **greet()** is **virtual**. In this function, the function is invoked based on the *actual* type of the object. In the final group above, we made **pm** point to a **Trumpeter**; so the **Trumpeter** player is invoked.

Polymorphism

Concept of binding.

- Connecting a function call to a function body is called *binding*.
- When binding is performed before the program is run (by the compiler and linker), it's called *early binding*. In our first example, the problem related to the function **greet** is caused by the *early binding*: the compiler cannot know the correct function to call when it has only the Musician address!
- The solution is called *late binding*, which means that the binding occurs at runtime, based on the type of the object. In the last example, the **late binding occurs for the function greet thanks to virtual keyword!**
- C compilers have only one kind of function call, that's *early binding*.

Polymorphism

To retain

- Creating *late binding*, we get the desired behavior of an object!
- *Late binding* occurs only with **virtual functions** and only when you're using **pointers to the base class** where those virtual functions exist.

Rule : Virtual functions + Pointers to base class.

Polymorphism

Polymorphism is the third essential feature of an object-oriented programming language, after encapsulation and inheritance.

Polymorphism means to use different objects in the same code, only manipulating pointers to the base class Musician! **With polymorphism we can write code in terms of generic Musician and make the code work correctly for any actual Musician.**

The generic Musician is polymorphic - *many forms* - because at any particular point during program execution it can be a Trumpeter, a Pianist etc. **Polymorphism** requires the generic Musician to behave differently at run-time depending on the *actual* type of Musician it is.

Late binding is used in order to implement **Polymorphism!**

Polymorphism

Polymorphism

What we do is create a common base class for all types we want to use together polymorphically. The common **interface** of all the classes is implemented using virtual functions. In our examples above, we used the **Musician** class as the base class holding the common interface.

Polymorphism

Example of polymorphic code only using pointers to the base class Musician.

```
void Musiciangreet (Musician* pm) //Function that receives a parameter  
{  
cout<<"introducing....\n";  
pm->greet ();  
}
```

This routine will work on an instance of Musician or any class derived from Musician because **greet() is virtual!**

Doing this means, we don't have to write function Musiciangreet() for each derived class. We can simply pass for eg the address of Pianist in the function.

Polymorphism

To use C++ Vector library we must #include <vector>

We can create an entire orchestra in a single data structure!

Type of element of vector (Here, it's a pointer to Musician)

```
int main(){
    vector<Musician*> orchestra; // a vector which
    // holds the entire orchestra which is a vector of pointers to Musician.
    orchestra.push_back(new Trumpeter); // Adds Trumpeter to vector
    orchestra.push_back(new Pianist);
    orchestra.push_back(new Violonist);

    for (int i=0; i<orchestra.size(); i++)
        MusicianGreet(orchestra[i]); // Calling MusicianGreet() on elements
    // of orchestra.

    delete v[i]; // Release memory to allocation.

}
```

push-back is a function that adds elements to a vector.

Polymorphism

Object slicing There is a **distinct** difference between passing the address of objects (using pointers) and passing objects by value when using polymorphism. All the examples that we have seen pass addresses and not values!

What happens if you try to upcast without using pointers?

The object is "sliced" until all that remains is the subobject that corresponds to the destination type of your cast.

In previous page, the function call `Musician& get(Orchestra[i]);` will greet accordingly to the objects in `Orchestra`. for eg. if `Orchestra[0]` is `Pianist` then "Pianist says Hello" will print.

Polymorphism

Object slicing

```
void Musiciangreet (Musician pm) // passing Musician by value.  
{  
cout<<"introducing....\n";  
pm.greet ();  
}
```

The function **Musiciangreet()** is passed an object of type **Musician** *by value*. It then calls the virtual function **greet()** for the **Musician**.

Polymorphism

```
Trumpeter t;  
Musician greet(t);
```

You might expect that the above code produce "Trumpeter says...". The result will be "Musician says.."

When a virtual function is invoked through an object, there is no doubt about which version of the function is invoked.

Polymorphism

Rule: Use virtual function+pointers to the base class!

- With greet() defined as virtual in the base class, you can add as many new types as you want without changing the Musiciangreet() function. In a well-designed OOP program, most or all of your functions will follow the model of Musiciangreet() and communicate only with the base-class interface. Such a program is **extensible** because you can add new functionality by inheriting new data types from the common base class. The functions that manipulate the base-class interface will not need to be changed at all to accommodate the new classes.

Polymorphism

Invoking virtual functions dynamically

We have seen that virtual functions are invoked dynamically, that is, the version of the function invoked is based on the run-time type of the object. This allows us to support polymorphism by using pointers to the base class. **But pointers aren't the only way we can use virtual functions in order to obtain a polymorphic behaviour. We can also use virtual functions and references together.**

Polymorphism

Invoking virtual functions dynamically (with references)

```
void Musiciangreeting(Musician &rm) //using reference this time
    { rm.greet() }    // invokes a virtual function
int main()
Musician m;
Trumpeter t;
Pianist p;

Musiciangreeting(m);      // will result in
    invocation of Musician::greet()
Musiciangreeting(t);    // will result in
    invocation of Trumpeter::greet()
Musiciangreeting(p);    // will result in
    invocation of Pianist::greet()
return 0;
```

Polymorphism

Conclusion: Calling a virtual function through an object - rather than through a pointer or reference to an object - always results in the same version being invoked.

Remark: If you are invoking a virtual function through a pointer or reference, you still need to ensure which version is invoked.

```
int main() {
    Musician *pm=new Trumpeter; // really a
                                Trumpeter
    pm->greet(); // invokes Trumpeter::greet()
    pm->Musician::greet(); // invokes
                           Musician::greet()
    pm->Trumpeter::greet(); // error
    delete pm;
    return 0;
}
```

Polymorphism

Conclusion: Calling a virtual function through an object - rather than through a pointer or reference to an object - always results in the same version being invoked.

If you want a function to behave polymorphically: use the `virtual` keyword + pointers or references to the base class.

Polymorphism

Relaxed Overriding Rules

You must always match signatures to override a virtual function. You do have some freedom with the return type. For example, if the original function returns a pointer to some class, the overriding function can return a pointer to a derived class.

Polymorphism

Abstract base classes and pure virtual functions

Often in a design, you want the base class to present *only* an interface for its derived classes. In other words, you don't want anyone to actually create an object of the base class, only to upcast to it, so that its interface can be used. This is accomplished by making that class *abstract* which happens if you give it at least one *pure virtual function*.

What is a pure virtual function?

A pure virtual function is a function which uses the **virtual** keyword and it is followed by **=0**.

If anyone tries to make an object of an abstract class, the compiler prevents them. This is a tool that allows you to enforce a particular design.

Polymorphism

- **Abstract base classes and pure virtual functions**

in derived classes

When an abstract class is inherited,[↑] all pure virtual functions must be implemented. *The pure virtual fn is not defined in base class but has to be defined in derived class.*

- **An interface class (or a pure abstract class)** is a class which contains only pure virtual functions and no member variables. It can be seen as a contract between the designer of the class and the users, in the sense that any class implementing the interface class provides the functionality announced in the interface class.

Polymorphism

The constructor can not be made virtual! *While destructors have to be made virtual.*

Destructors and Virtual destructors What happens if you want to manipulate an object through a pointer to its base class (that is, manipulate the object through its generic interface)? The problem occurs when you want to **delete** a pointer of this type for an object that has been created on the heap with **new**. If the pointer is to the base class, the compiler can only know to call the base-class version of the destructor during **delete**. This is the same problem that virtual functions were created to solve the general case.

Virtual functions work for destructors as they do for all other functions except constructors.

Polymorphism

(Nonvirtual destructor)

(i) Virtual versus non-virtual destructor!

```
//Behavior of virtual vs. non-virtual
//destructor
class Base1{ //Parent Class
public:
    ~Base1() { cout<<"~Base1()\n"; } //destructors
};

class Derived1: public Base1{ //Child Class
public: ~Derived1() {cout<<"~Derived1()\n"; } //destructors
};
```

Polymorphism

(Virtual destructors)

(ii) Virtual versus non-virtual destructor!

```
//Behavior of virtual vs. non-virtual
//destructor

class Base2{    //parent class
public:
    virtual ~Base2() { cout<<"~Base2()\n"; } //destructor
};

class Derived2: public Base2{    //child class
public: ~Derived2() {cout<<"~Derived2()\n"; } //destructor
};
```

Polymorphism

Virtual versus non-virtual destructor!

```
//Behavior of virtual vs. non-virtual
// destructor
int main()
{
Base1 *bp=new Derived 1; // Upcast (version(i))
delete bp;
Base2 * bp2=new Derived2; // Upcast (version(ii))
delete bp2;
}
```

Polymorphism

Virtual versus non-virtual destructor!

When you run the program you'll see:

- **delete bp** only calls the base-class destructor;
- **delete bp2** calls the derived-class destructor followed by the base class destructor, which is the behavior we desire, otherwise we will lose some memory.

Note that:

- Forgetting to make a destructor **virtual** is an insidious bug because it often doesn't directly affect the behavior of your program, but it can quietly introduce a memory leak.
- Even though the destructor, like the constructor, is an "exceptional" function, it is possible for the destructor to be virtual because the object already knows what type it is.

Summing up

We have learnt about:

- **Inheritance** Create a base class and derived classes, where the base class contains the common member variables and functions.
- **Polymorphism**- *Many forms.* You want a function of the base class to behave in a specific way to each of the derived classes. In order to get the polymorphic behaviour: use the keyword **virtual** and **pointers or references** to the base class.

Summing up

- Constructors can not be virtual. The destructor of a class that you intend to use as a base class should be a **virtual destructor**.
- In conclusion; the base class contains the common non-virtual functions (that have the same behaviour for each derived class) and virtual functions (same name, but different behaviour depending on the derived class)!

Pricing in the Black-Sholes Model

Pricing in the Black-Sholes Model

Mathematical background

- The model for stock price evolution is

$$dS_t = \mu S_t dt + \sigma S_t dW_t, \quad (1)$$

and a riskless bond B , grows at a continuously compounded rate r .

- The Black-Sholes pricing theory then tells us that the price of a vanilla option, with expiry T and payoff f , is equal to

$$\text{Price at time zero} = e^{-rT} \mathbb{E}^{\mathbb{Q}}[f(S_T)], \quad (2)$$

where the expectation is taken under the risk-neutral probability measure \mathbb{Q} .

Assume Complete Market \Rightarrow Unique Probability measure \mathbb{Q} .

Pricing in the Black-Sholes Model

Mathematical background

Recall:

- The model for stock price evolution under the probability measure \mathbb{P} is

$$dS_t = \mu S_t dt + \sigma S_t dW_t, \quad (3)$$

- The evolution of the stock price under the probability measure \mathbb{Q} is

$$dS_t = rS_t dt + \sigma S_t dW_t. \quad (4)$$

Pricing in the Black-Sholes Model

Mathematical background

Notation:

- S : Price of the underlying (stock price)
- K : Strike price
- r : Risk free interest rate.
- σ : Volatility.
- t : Current date.
- T : Maturity.

Pricing in the Black-Sholes Model

Mathematical background

Analytical option prices (we give the formulas only for call options, similar formulas for put options).

- The **payoff** of the call option is

$$C_T = \max(S_T - K, 0). \quad (5)$$

- The **analytical option price** has the following functional form:

$$c = S\mathcal{N}(d_1) - Ke^{-r(T-t)}\mathcal{N}(d_2), \quad (6)$$

where

$$d_1 = \frac{\log(\frac{S}{K}) + (r + \frac{1}{2}\sigma^2)(T - t)}{\sqrt{T - t}} = \quad (7)$$

Pricing in the Black-Sholes Model

Mathematical background

$$= \frac{\log(\frac{S}{K}) + r(T - t)}{\sigma\sqrt{T - t}} + \frac{1}{2}\sigma\sqrt{T - t}.$$

and

$$d_2 = d_1 - \sigma\sqrt{T - t}.$$

$\mathcal{N}(\cdot)$ represents the cumulative normal distribution.

Pricing in the Black-Sholes Model

Mathematical background

Computation of Greeks

In trading of options, a number of partial derivatives (called sensitivities or Greeks) of the option price is important.

- **Delta** (*gives hedging strategy*)

The first derivative of the option price with respect to the underlying is called the *delta* of the option price. It is the derivative most people will run into, since it is important in *hedging* options.

$$\frac{\partial C}{\partial S} = \mathcal{N}(d_1);$$

This is the one we use numerically to compute hedging strategy.

Pricing in the Black-Sholes Model

Mathematical background

- **Gamma**

The second derivative of the option wrt the underlying stock.
These are equal for puts and calls

$$\Gamma_c = \frac{\partial^2 c}{\partial S^2} = \frac{\mathcal{N}(d_1)}{S\sigma\sqrt{T-t}}$$

- **Theta**

The partial derivative with respect to time-to-maturity

$$\Theta_c = \frac{\partial c}{\partial(T-t)} = -\frac{\mathcal{N}(d_1)S\sigma}{2\sqrt{T-t}} - rKe^{-r(T-t)}\mathcal{N}(d_2).$$

Pricing in the Black-Sholes Model

Mathematical background

- **Vega**

The partial with respect to volatility

$$\text{Vega}_c = \frac{\partial c}{\partial \sigma} = S\sqrt{T-t}\mathcal{N}(d_1)$$

Pricing in the Black-Sholes Model

Mathematical background

- **Rho**

The partial with respect to the interest rate

$$\text{Rho}_c = \frac{\partial c}{\partial r} = K(T-t)e^{-r(T-t)}\mathcal{N}(d_2)$$

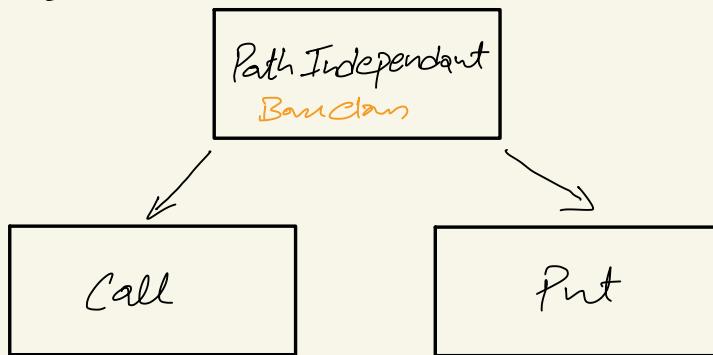
Class:
Black-Scholes
(Underlying Model)

Hierarchy of Claims:
Options
(Financial Contract)

Class:
Monte-Carlo Pricer

Here, we are using just one model (Black-Scholes) but we could also have done hierarchy of claims for Model & include other models like local volatility models etc.

Hierarchy of Claims



Simulation

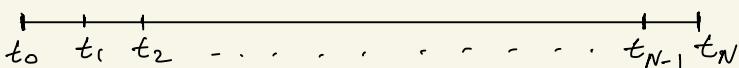
In a discrete version of Geometric Brownian Motion (GBM), the estimate for the price of an option is

$$e^{-rT} \frac{1}{N} \sum_{i=0}^N f(S_T^i)$$

Where
 $N = \text{No. of Simulations}$
 $r = \text{interest rate}$
 $T = \text{Time to Maturity}$
 $f = \text{Payoff function}$
 $S^i = i^{\text{th}} \text{ simulation of path following B&S model.}$

e.g.: The payoff function for
call option: $f(x) = (x - K)^+$
put option: $f(x) = (K - x)^+$

$$S_{t_{i+1}} = S_{t_i} \exp \left[\left(r - \frac{\sigma^2}{2} \right) \delta t + \sigma (W_{t_{i+1}} - W_{t_i}) \right]$$



$$W_{t_{i+1}} - W_{t_i} \sim \sqrt{st} e_i$$

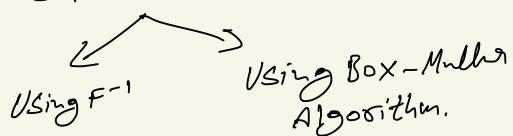
$$e_i \sim N(0, 1)$$

How to Simulate:

→ For each path s_i we need to simulate N steps independently
Normally distributed random variables.

Steps:

- ① Simulate uniformly distributed Random Variable using Mersenne Twister.
- ② Simulate Normally distributed Random Variables.



Pricing in the Black-Sholes Model

Implementation in C++

We will have the class BlackSholesModel and hierarchy of classes the options which will be completely independent & a class MonteCarloPricer.

Pricing in the Black-Sholes Model

Implementation

We define a class called **BlackSholesModel**, which corresponds to the model of the asset price.

```
class BlackSholesModel{  
public: // For simplicity but you should do Private when creating.  
    double stockPrice;  
    double volatility;  
    double drift;  
    double riskFreeRate;  
    double date; // Today's date.  
}
```

In order to simplify the presentation, we have declared the member variables as public. In practice, you have to declare them as private and to use the set and get functions.

Pricing in the Black-Sholes Model

Implementation

The choice of member variables in this class is carefully considered. This class **only** contains the variables associated with the model and not variables associated with the financial contract. We will specify different option contracts in different classes shortly.

Pricing in the Black-Sholes Model

Implementation

We define a separate class for the path independent option class. We'll create a base class **PathIndependentOption** which contains a maturity and a strike, but does not contain any details about the current market data.

The class contains as **public functions**:

- *Common functions*, with the *same behavior* for any PathIndependent Option in the BlackSoles model.
- *Common functions*, with *different behaviors* depending on the type of option. Example: **payoff**, **price**. These methods are declared using the keyword **virtual** which means that these functions may be overriden.

Think
Polymorphism

Pricing in the Black-Sholes Model

Implementation

```
class PathIndependentOption{           //base class.  
private:  
    double maturity;                } //member variables common to all options.  
    double strike;  
public:  
    void SetMaturity(double);  
    double GetMaturity() const;      } } accessor functions  
    void SetStrike(double);  
    double GetStrike() const;  
    virtual double payoff(double StockAtMaturity)  
        const=0; //Pure virtual function. Const as it will not modify member variables.  
    virtual double price(const BlackSholesModel&  
        bsm) const=0; //Pure virtual fn i.e. no implementation in base class but have to be  
                    //implemented in derived class.  
    virtual ~PathIndependentOption() {} }; //destructor.
```

Functions with Polymorphic behaviour is given with virtual.

Pricing in the Black-Sholes Model

(Derived class: CallOption)

Implementation

We now define a new class, named **CallOption**, which is derived from the base class **PathIndependentOption**.

```
class CallOption: public PathIndependentOption
{
public:
    double payoff (double x) const;
    double price (const BlackSholesModel& bsm)
        const;
    ~CallOption () {};
}
```

Pricing in the Black-Sholes Model

Implementation

The function **CallOption::price** is implementing the formula given at the beginning of the lecture.

```
double CallOption::price (const  
BlackSholesModel& bsm) const  
{  
    double S=bsm.stockPrice;  
    double K=strike;  
    double sigma=bsm.volatility;  
    double r=bsm.riskFreeRate;  
    double T=maturity-bsm.date;  
    double  
        numerator=log (S/K) + (r+sigma*sigma*0.5) *T;
```

Pricing in the Black-Sholes Model

Implementation

```
double denominator=sigma*sqrt(T);  
double d1=numerator/denominator;  
double d2=d1-denominator;  
return S*normcdf(d1)-exp(-r*T)*K*normcdf(d2);  
}
```

normcdf(.) has been implemented in the first practical, representing the cumulative standard normal distribution function.

Pricing in the Black-Sholes Model

(Derived Class: PutOption)

Implementation

We now define a new class, named **PutOption**, which is derived from the base class **PathIndependentOption**.

```
class PutOption: public PathIndependentOption
{
public:
    double payoff (double x) const;
    double price (const BlackSholesModel& bsm)
        const;
    ~PutOption () { };
}
```

Pricing in the Black-Sholes Model

Implementation

Remarks:

- In order to compute the price of the Call (resp. Put) option, we have used the analytical formulas.
- What does it happen if we don't have explicit formulas?
Solution: **Monte-Carlo**! This is the reason for which the function **price** has been declared using the keyword **virtual** (if we have explicit formulas, we use them in order to compute the price, otherwise we price by Monte-Carlo!)

Pricing in the Black-Sholes Model

Implementation

We write a new class **MonteCarloPricer** that uses a **BlackScholes** model to simulate stock prices and then uses risk-neutral pricing to price a Path Independent Option by Monte Carlo.

Of course, as we said, pricing a European call or put option by Monte-Carlo is unnecessary since one already knows the BlackScholes formulas. However, implementing Monte-Carlo method for a call (or put) option is a valuable exercise which will help you to extend the ideas slightly to price a path-dependent option for which we have no analytical formula.

Pricing in the Black-Sholes Model

Monte-Carlo

Algorithm

To compute the Black–Scholes price of an option whose payoff is given in terms of the prices at times $t_0, t_1, t_2, \dots, t_n$:

- Simulate stock price paths *in the risk-neutral measure*. i.e. use the algorithm with $\mu = r$.
- Compute the payoff for each price path. *(for large no. of Paths)*
- Compute the discounted mean value.
- This gives an unbiased estimate of the true risk-neutral price.

Pricing in the Black-Sholes Model

Price = Discounted expectation under risk neutral Measure.

Monte-Carlo

Algorithm

More precisely, the main idea of Monte-Carlo is: Given a payoff function f , the price of the option $\gamma := \mathbb{E}^{\mathbb{Q}}[e^{-rT} f(S_T)]$ can be approximated by $\bar{X}_N := e^{-rT} \frac{1}{N} \sum_{i=0}^{i=N} f(S_T^i)$, with N large, where

- N is the number of simulations (this approximation follows by the Law Large Number).
- S^i - the i^{th} simulation of the price of the asset S under the risk neutral probability measure \mathbb{Q} .

\bar{X}_N = Empirical Mean

Pricing in the Black-Sholes Model

Monte-Carlo

Algorithm

The estimator \bar{X}_N gives an approximation of the price. A second important problem is the **estimation of the error**. In order to do this, we use the Central Limit Theorem which gives:

$$\sqrt{N}(\bar{X}_N - \gamma) \rightarrow \mathcal{N}(0, \sigma^2). \quad (8)$$

The diagram consists of two orange arrows pointing downwards from the text "Estimated Price" and "True Price" to the term $\sqrt{N}(\bar{X}_N - \gamma)$ in the equation.

We'll learn how to implement the estimation of the error during the next practical.

A third important problem: Variance reduction techniques!
(see tomorrow the practical).

Pricing in the Black-Sholes Model

Monte-Carlo

Simulation of the stock price paths

- Recall the dynamics of the stock price in continuous time (under probability \mathbb{Q})

$$dS_t = S_t(rdt + \sigma dW_t). \quad (9)$$

- Write the **discrete** geometric Brownian motion under the probability measure \mathbb{Q} , which will be used for simulation.

$$S_{t_{i+1}} = S_{t_i} \exp\left(\left(r - \frac{\sigma^2}{2}\right)(t_{i+1} - t_i) + \sigma(W_{t_{i+1}} - W_{t_i})\right). \quad (10)$$

Pricing in the Black-Scholes Model

S_0 = initial price

Algorithm for Black-Scholes price paths

- Define

$$\delta t_i = t_i - t_{i-1}$$

- Simulate independent, normally distributed ϵ_i , with mean 0 and standard deviation 1 (used to simulate increments of B.M.)
- Define $s_{t_0} = \log(S_0)$ and then for $i \geq 1$

$$s_{t_i} = s_{t_{i-1}} + \left(r - \frac{1}{2}\sigma^2 \right) \delta t_i + \sigma \sqrt{\delta t_i} \epsilon_i$$

$$\sqrt{\delta t_i} \epsilon_i \stackrel{\mathcal{L}}{=} W_{t_i} - W_{t_{i-1}}. \text{ Since, } W_{t_i} - W_{t_{i-1}} \sim N(0, \delta t_i) \text{ & } \epsilon_i \sim N(0, 1)$$

- Define $S_{t_i} = \exp(s_{t_i})$.
- S_{t_i} simulate the stock price (given by (10)) at the desired times.

Pricing in the Black-Sholes Model

Generating random numbers with Monte Carlo

As we have seen in the previous slide, we have to simulate independent, normally distributed ϵ_i , with mean 0 and standard deviation 1. How to do this?

Pricing in the Black-Sholes Model

Generating random numbers with Monte Carlo

- Conventional computers cannot generate true random numbers, they can only generate *pseudo random numbers*.
- Old method to generate uniformly distributed random numbers - using the **rand** function (it isn't a good choice because the sequence of pseudo random numbers it generates start to repeat themselves rather quickly).
- A much better algorithm: **Mersenne Twister**.
- The C++ class **mt19937** allows you to use the Mersenne Twister algorithm. In order to do this, one has to
#include <random >.

Pricing in the Black-Sholes Model

This slide of keyword static to say variable/fn can only be used in current source.cpp file.
1st was static local variable which exists below as program runs & 2nd one is member vari/fns declared static in class and to
Count no. of object of that class which has been created.

Generating random numbers with Monte Carlo

- How to use? First create a global variable of type mt19337 called **mersenneTwister**

```
static mt19337 mersenneTwister;
```

// Creating object mersenneTwister
// belonging to class mt 19337.

mersenneTwister is our random generator.

Remark related to the keyword static in this context: By marking a variable or a functions as **static**, we are saying it can only be used in the current source file. This means that we can reuse the name in other source files if desired. Note that although simply not including things in the header files is the main way of achieving information-hiding, you can go further using the keyword static.

Pricing in the Black-Sholes Model

Generating random numbers with Monte Carlo

- Function which generates n independent uniformly distributed random numbers.

```
vector<double> randuniform (int n) {  
    vector<double> ret(n, 0.0); // Vector of size n will all elements  
    for (int i=0;i<n; i++) {  
        ret[i]=(mersenneTwister() + 0.5) /  
            (mersenneTwister.max () + 1.0); // Normalising s.t. R.V. is between 0 & 1.  
    }  
    return ret; } // Vector of uniform random no. between 0 & 1.
```

Pricing in the Black-Sholes Model

Generating random numbers with Monte Carlo

- To generate a random integer we write **mersenneTwister()**.
This returns a random integer in the range
mersenneTwister.min()=0 to **mersenneTwister.max()**.

Pricing in the Black-Sholes Model

Method 1 works for any dist.

Generating random numbers with Monte Carlo Simulation of gaussian random variables

- **Method 1.** Generate a uniformly distributed random variable u . Let F^{-1} be the inverse of the standard normal cumulative distribution. Then $F^{-1}(u) \sim \mathcal{N}(0, 1)$.
- **Method 2 (Box-Muller algorithm)** Generate two uniformly distributed random variables u_1 and u_2 in the interval $(0, 1)$. Define

$$n_1 = \sqrt{-2.0 \log(u_1)} \cos(2\pi u_2) \quad (11)$$

$$n_2 = \sqrt{-2.0 \log(u_1)} \sin(2\pi u_2). \quad (12)$$

n_1 and n_2 will be independent normally distributed random variables with mean 0 and standard deviation 1.

Pricing in the Black-Sholes Model

Generating random numbers with Monte Carlo Simulation of gaussian random variables

- Function which generates n independent normally distributed random numbers using Method 1. *With $\mu=0, \sigma^2=1$*

```
vector<double> randn(int n)
{
    vector<double> ret=randuniform(n); // Vector of uniform dist
    Random No's between 0 & 1.
    for (int i=0;i<n;i++)
        ret[i]=norminv(ret[i]);
    return ret;
}
```

The function **norminv** representing the inverse of the standard normal cumulative distribution function has been implemented in the first practical.

Pricing in the Black-Sholes Model

Generate price paths in C++

We now wish to add a function `generatePricePath` to class `BlackSholesModel` which takes a final date (`toDate`) and a number of steps (`nSteps`) and generates a random Black–Scholes Price path with the given number of steps.

```
class BlackScholesModel {  
public:  
    ... other members of BlackScholesModel ...  
  
    std::vector<double> generatePricePath(  
        double toDate,  
        int nSteps) const;  
};
```

Uses IP measure to generate path. Uses drift u.

Pricing in the Black-Sholes Model

Generate price paths in C++

Note that the class declaration effectively contains the specification. If you choose good function and variable names, you won't need too many comments.

Pricing in the Black-Sholes Model

Generate price paths in C++

We also want a function `generateRiskNeutralPricePath` which behaves the same, except it uses the \mathbb{Q} -measure to compute the path.

```
class BlackScholesModel {  
public:  
    ... other members of BlackScholesModel ...  
  
    std::vector<double>  
    generateRiskNeutralPricePath(  
        double toDate,  
        int nSteps) const;  
};
```

Pricing in the Black-Sholes Model

Generate price paths in C++

Private helper function

To implement these functions, we introduce a private function that allows you to choose the drift in the simulation of the price path.

This fn has exton parametrs
which is used to simulate path
for specific drift.

(Rewriting class BlackScholesModel)

```
class BlackScholesModel {  
    ... other members of BlackScholesModel ...  
private:  
    std::vector<double>  
    generateRiskNeutralPricePath (  
    generatePricePath  
        double toDate,  
        int nSteps,  
        double drift)  
const;};
```

Pricing in the Black-Sholes Model

Generate price paths in C++

Private helper function

This function is private because we've only created it to make the implementation easier. Users of the class don't need (or even want) to know about it.

This private function is used in implementation of the public functions. The extra parameter is used to simulate price path for specific drift.

Pricing in the Black-Sholes Model

Generate price paths in C++

Implement the helper function

(definition of `private helper fn`)

```
vector<double>
BlackScholesModel::generatePricePath(
    double toDate,
    int nSteps,
    double drift ) const {
    vector<double> path(nSteps+1, 0.0); // vector path of ones used to store results.
    path[0]=stock; // initial price.
    vector<double> epsilon = randn( nSteps ); // simulate vector of N(0,1) for BM increments from our time steps to another using fn randn().
    double dt = (toDate-date)/nSteps; // time increment.
    double a = (drift -
    volatility*volatility*0.5)*dt;
    double b = volatility*sqrt(dt);
```

discrete geometric BM formula used now.

Pricing in the Black-Sholes Model

Generate price paths in C++

Implement the helper function

$$\underbrace{b * \epsilon_{\text{epsilon}}[i]}_{(\sigma \cdot \sqrt{\Delta t} \cdot c_i)} \sim \sigma (W_{t_{i+1}} - W_{t_i})$$

```
double currentLogS = log( stockPrice );
for (int i=0; i<nSteps; i++) {
    double dLogS = a + b*epsilon[i];
    double logS = currentLogS + dLogS;
    path[i+1] = exp( logS );
    currentLogS = logS;
}
return path;
```

Pricing in the Black-Sholes Model

(Notice, generatePricePath() fun has been overloaded.)

Generate price paths in C++

① Implement the public functions

```
vector<double>
BlackScholesModel::generatePricePath(
    double toDate,
    int nSteps ) const {
    return generatePricePath( toDate, nSteps,
drift );
}
```

↑ helper function

This function generatePricePath generates path
under real probability measure \mathbb{P} .

Pricing in the Black-Sholes Model

Generate price paths in C++

② Implement the public functions

```
vector<double> BlackScholesModel::  
    generateRiskNeutralPricePath(  
        double toDate,  
        int nSteps ) const {  
    return generatePricePath(  
        toDate, nSteps, riskFreeRate );  
}
```

Helper function.

Notice that with this design we've avoided writing the same complex code twice.

The function generateRiskNeutralPricePath generates path under Risk-Neutral probability \mathbb{Q} .

Pricing in the Black-Sholes Model

Monte-Carlo specification

We want to write a class called **MonteCarloPricer** that:

- Is configured with **nScenarios**, the number of scenarios to generate and **nSteps**, the number of time steps.
- Has a function **price** which takes a **CallOption** and a **BlackScholesModel**, and computes (by **Monte Carlo**) the price of the **CallOption**.

We'll see that the declaration for **MonteCarloPricer** is pretty much the same thing as this specification.

Pricing in the Black-Sholes Model

Monte-Carlo declaration (in the header file)

```
class MonteCarloPricer {
public:
    /* Constructor */
    MonteCarloPricer();
    /* Number of scenarios */ No. of Paths we simulate.
    int nScenarios;
    /* number of steps */
    int nSteps;
    /* Price a call option */
    double price( const CallOption& option,
                  const BlackScholesModel&
model );
};
```

Pricing in the Black-Sholes Model

①

MonteCarlo.cpp

(definition of constructor)

```
#include "MonteCarloPricer.h"

using namespace std;

MonteCarloPricer::MonteCarloPricer() :
    nScenarios(10000), nSteps(100) {
}
```

Pricing in the Black-Sholes Model

MonteCarlo.cpp

(Function that computes Price)

(For Path independent option, we need just last value of stock & is given by last element of vector Path using Path.back().)

② The implementation of price

```
double MonteCarloPricer::price(
    const CallOption& callOption,
    const BlackScholesModel& model ) {
    double total = 0.0;
    for (int i=0; i<nScenarios; i++) {
        vector<double> path= model.
            generateRiskNeutralPricePath(
                callOption.maturity,
                nSteps );
        double stockPrice = path.back(); stock price to final value
        double payoff = callOption.payoff(
            stockPrice );
        total+= payoff; // N=nScenarios
    }
}
```

$$\sum_{i=0}^{N=nScenarios} (S_T^i - K)^+$$

$f(S_T^i) = \text{Payoff}(S_T^i)$

Pricing in the Black-Sholes Model

MonteCarlo.cpp

The implementation of price

```
double mean = total/nScenarios; //Empirical Mean.  
double r = model.riskFreeRate; //interest rate.  
double T = callOption.maturity -  
model.date; // Time to maturity (duration). The model.date gives todays date.  
return exp(-r*T) *mean; // Price using Monte-Carlo.  
}
```

Note, we used Model.riskFreeRate & Model.date as they were declared public for simplicity. But in practice, these will be declared private & we should use accessor functions to obtain these values.
eg: Model.getData().

Pricing in the Black-Sholes model

The following call option price is 2nd Alternate which uses Monte Carlo method. This defⁿ we are writing here can be used for any type of Option.

Implementation of the method CallOption::price using Monte-Carlo

```
double CallOption::price(
    const BlackScholesModel& model ) const
{
    MonteCarloPricer pricer; // Default constructor called to initialise object
                             // Pricer Belonging to class
                             // MonteCarloPricer.
    return pricer.price( *this, model ); // calls fn Price from
}                                         // MonteCarloPricer.
```

Pointer to current instance of type
CallOption which is
calling the fn price().

Remark:

```
int main()
{
    BlackScholes m;
    CallOption c;
    cout << c.price(m); // here *this points to c.
    PutOption c1;
    cout << c1.price(m); // here *this points to c1.
```

Q: How to extend the pricing by Monte Carlo from a Call option
to any Path-Independent Options (e.g. Put, Digital Call etc.)?

⇒ Replace

double price (const Calloption &, const BlackScholes &) const
by
double price (const PathIndependent &, const BlackScholes &) const
in the Monte-Carlo declaration & definition. The rest of the
code is exactly same. This works because of upcasting.
We can now do:

```
int main()
{
    BlackScholes m;
    CallOption c;
    PutOption p;
    cout << c.price(m); // Call price from Calloption which uses MC price.
    cout << p.price(m); // Put price from Putoption which uses MC price.
```

Pricing in the Black-Sholes Model

MonteCarlo.cpp

Our Monte-Carlo pricer defined above can be used for the pricing of a Call Option. What should we change in order to be able to price any Path Independent Option? *e.g Put, Digital call etc.*
We have simply to replace:

```
double price( const CallOption& option,  
               const BlackScholesModel&  
               model );
```

by

```
double price( const PathIndependentOption&  
               option,  
               const BlackScholesModel&  
               model );
```

Pricing in the Black-Sholes Model

MonteCarlo.cpp

The code developed in the function **price** will work with any Path Independent Option, because the **payoff** has been declared virtual.

Conclusion: Making the above changes, we are now able to price any Path Independent Option (try to add also other class, DigitalCall, Digital Put...).

A general class hierarchy to price path-independent and path-dependent options

Pricing in the Black-Scholes model

Aim: Add the ability to price a path-dependent option.

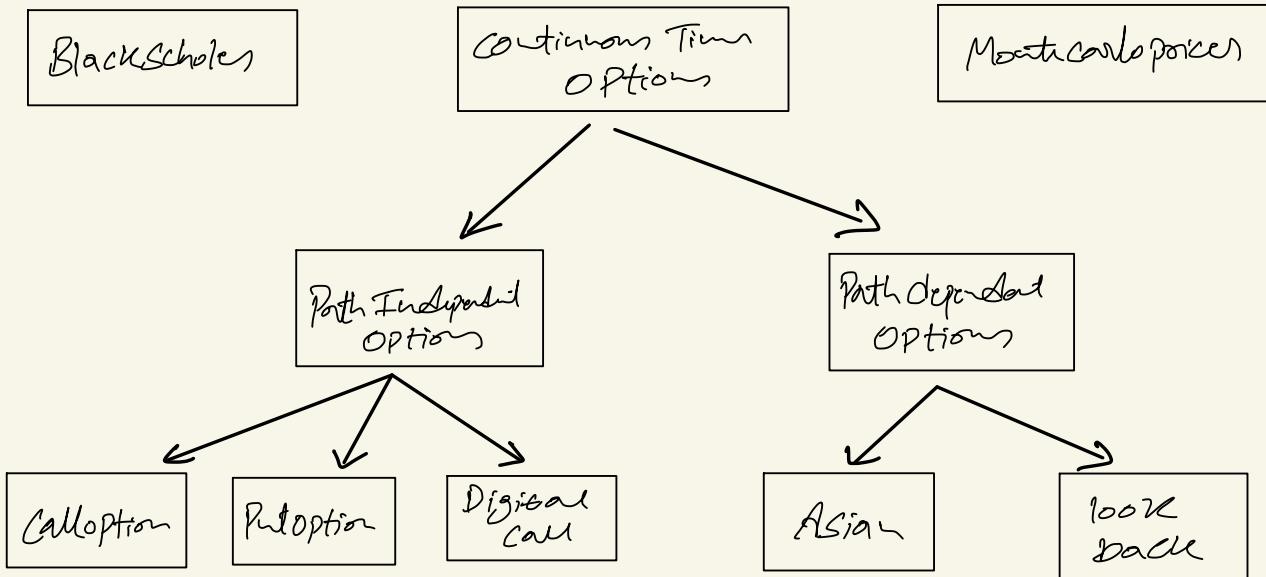
Example: An up-and-out knock-out call option with strike K , barrier B , and maturity T is an option which pays off:

$$\begin{cases} \max\{S_T - K, 0\} & \text{if } S_t < B \text{ for all } t \in [0, T] \\ 0 & \text{otherwise.} \end{cases} \quad (13)$$

What are the changes that we have to make to our previous hierarchy of classes?

Eg. of other Path dependent options : Geometric Asian Options,
Arithmetic Asian options,
lookback options etc.

Steps: Add a Superclass Continuous Time Option.



Pricing in the Black-Sholes model

1. We add a super class **ContinuousTimeOption**. (Base class)

```
class ContinuousTimeOption {  
private:  
    double maturity;  
    double strike;  
public:  
    virtual double price( const  
        BlackScholesModel& bsm ) const=0;  
    virtual double payoff(std::vector<double>&  
        stock)=0; // Handles Pathdependent as we need whole path.  
    virtual ~ContinuousTimeOption() {};//destructor  
    virtual bool isPathDependent() const=0;
```

// Price is set virtual since its implementation uses either explicit formula for call/put & MC otherwise.

Both Path dependent & independent.

Pricing in the Black-Sholes model

```
double GetMaturity() const;  
void SetMaturity( double maturity );  
double GetStrike() const;  
void SetStrike( double strike );  
};
```

*Access
fns.*

- Base class provides basic implementations of methods common to most options.
- The base class has a virtual destructor. Recall that any class used as a base class must have a virtual destructor!

Pricing in the Black-Sholes model

- The payoff function has as parameter a vector (containing the entire path of the stock price), instead of only one value representing the terminal value of the stock.

```
virtual double payoff(std::vector<double>&  
stock)=0;
```

- We have add a virtual method **isPathDependent**.
- How should we adapt the design of our PathIndependentOption class?

Pricing in the Black-Sholes model

2. We write the class **PathIndependentOption** as a derived class from **ContinuousTimeOption**.

```
class PathIndependentOption : public
    ContinuousTimeOption {
public:
    /* Calculate the payoff of the option
     given a history of prices */
    double payoff(const std::vector<double>&
stockPrices) const; //implementation of virtual fn from base class Continuous
                     //timeoption.
    virtual double payoff(double stock)
    const=0; //Depends only on last Stock Price Value.
    bool isPathDependent() const;
    virtual ~PathIndependentOption () {} ; //destructor.
};

function payoff() is overloaded.
```

Pricing in the Black-Sholes model

Implementation of Payoff function from last page.

```
double PathIndependentOption ::payoff(const  
    std::vector<double>& stockPrices) const  
{ return payoff(stockPrices.back()); }  
  
bool PathIndependentOption ::isPathDependent()  
const {  
    return false; }
```

• `back()` gives last element of vector.



Pricing in the Black-Sholes model

- Note that: We have written the implementation of the function **payoff(vector<double> &)** using the function **payoff(double stock)**, which is a virtual function as before.
- No modification is needed to the classes derived from **PathIndependentOption**.

Pricing in the Black-Sholes model

3. We now add a class **PathDependentOption** derived from **ContinuousTimeOption**.

```
class PathdependentOption :  
    public ContinuousTimeOption {  
public:  
  
    virtual ~PathDependentOption() {} //destructor.  
  
    bool isPathDependent() const {return  
        true; };
```

Pricing in the Black-Sholes model

4. We add a class **UpAndOut knock-out** derived from **PathDependentOption** (we'll see the implementation the next practical).

We can easily extend our hierarchy of classes, by adding **Arithmetic Asian Calls**, **Arithmetic Asian Puts** etc.

Pricing in the Black-Sholes model

MonteCarlo.cpp

④

Modification in the MonteCarlo price function

```
double MonteCarloPricer::price(  
    const ContinuousTimeOption& Option,  
    const BlackScholesModel& model ) {  
    double total = 0.0;  
    for (int i=0; i<nScenarios; i++) {  
        vector<double> path= model.  
            generateRiskNeutralPricePath(  
                Option.maturity,  
                nSteps );  
        double payoff = Option.payoff(path);  
        total+= payoff;  
    }  
}
```

Two small changes in me

Pricing in the Black-Sholes Model

MonteCarlo.cpp

Modification in the MonteCarlo price function

```
    double mean = total/nScenarios;  
    double r = model.riskFreeRate;  
    double T = Option.maturity - model.date;  
    return exp(-r*T) *mean;  
}
```

Remark: In order to simplify the code, I have simply put *Option.maturity*. *maturity* is a private member of the class **ContinuousTimeOption**, so we can't access it from outside the class. In your implementation, you have to replace *Option.maturity* by *Option.GetMaturity()*.

Summing up

- We have written a generic **Monte-Carlo pricer** which contains:
 - A class representing the model (the evolution of the asset price): **class BlackSholesModel**
 - A complex hierarchy of classes representing the financial contract (**path-independent** and **path-dependent**).
 - A class representing the Monte-Carlo pricer.