Object Oriented Programming I

Topics today:

- Function Declaration
- Local Variables
- Default parameters and function overloading
- Pointers
- Advanced Function Features
- Dynamic Memory Allocation

Recap of Function Definition, Function Call, Return Value

Function Definition

```
returnType FunctionName(parameters) // function head
{
    statements
}
```

- returnType: type of the value returned by the function (**`void`** if nothing is returned)
- FunctionName: name we choose – should describe what the function is doing
- parameters: comma separated list of variable declarations
- statements: fulfill the purpose of the function

Example 1 (Function Definition)

- Write down the definition of a function **SumRand** that takes one parameter **n** of type **int**, computes the sum of **n** random numbers and returns the result.

```
double SumRand(int n)
{
      double sum=0;
      for(int i=0;i<n;i++)
         sum+=rand();
      return sum;
}
```

Function Call

A function call consists of the function name followed by a comma separated list of values for the parameters enclosed in parenthesis

Purpose:

- Passes the values of the parameters to the function
- Executes the function with this input

Example 2 (Function Call)

- Write down a main function in which the function SumRand from Example 1 is tested

```
int main()
{
    cout << SumRand(1000000) << endl;
    system("PAUSE");
}
```

Rules for Function Calls

Case 1: The function returns a value.

Then treat a function call like a value of the return type

Case 2: The function has return type void.

Then the function can only be called by a command like

```
FunctionName(parameters);
```

Local Variables of Functions

Recall: Function Parameters

```
int Multiply(int x, double y)
{
     return x*y;
}
```

- **x** and **y** are function parameters
- They are valid only inside the function body
- Values are assigned to **x** and **y** when the function is called.
- E.g. function call **Multiply(234,43)**

  ⇨ **x=234, y=43**

Do not Re-declare Function Parameters

```
int Multiply(int x, double y)
{
    int x; // wrong!
    return x*y;
}
```

- Function parameters are declared in function head
- Must not be re-declared in function body!
- int x; in functions body “shadows” the parameter x

Problem 1

- Check what happens if you try to use the function **Multiply** from the last slide

Local Variables of Functions

- Local variables are variables belonging to a function. They don't have any meaning outside the function body (out of scope there)
- Function parameters are local variables
  (if we don't use pointers or references)
- Every variable declared inside a function body is a local variable
- If the value of a local variable is changed in the function body, this has no effect on variables outside the function, even if they have the same name as the local variable

Local Variables: Example

- Define a function **Vector** which returns a vector of length 10 with all entries equal to 0
- Test the function inside the main function

Erroneous Version

```
1  vector<int> Vector()
2  {
3      vector<int> result(10);
4      return result;
5  }
6
7  int main()
8  {
9      cout << result << endl;
10     system("PAUSE");
11 }
```

What are the problems?

- **result** is a local variable of the function **Vector** and thus invalid inside the main function -- compiler error
- **cout** cannot be applied to a complete vector, only to its entries – another compiler error

Correct Version

```
1  vector<int> Vector()
2  {
3      vector<int> v(10);
4      return v;
5  }
6
7
8  int main(int argc, char *argv[])
9  {
10     vector<int> test=Vector();
11     for(int i=0;i<test.size();i++)
12        cout << test[i] << "  ";
13     system("PAUSE");
14 }
```

Name of Local Variable used with Different Meaning Elsewhere

```
1  vector<int> Vector()
2  {
3      vector<int> result(10);
4      return result;
5  }
6
7  int main()
8  {
9      vector<int> result = Vector();
10     for(int i=0;i<result.size();i++)
11        cout << result[i] << endl;
12     system("PAUSE");
13 }
```

- This is also correct
- **result** is a local variable of the function **Vector** and thus “invisible” inside the main function
- So, we can declare another vector **result** inside the main function without violating the rule “never declare a variable more than once”

Problem 2

- Declare and initialize a variable **x** inside a function **Test**
- Declare and initialize a variable **x** inside the main function
- Call the function **Test** in the main function
- Experiment to check that there is no conflict and the two variables with the same name have do “not influence each other”

Functions returning more than one value

- A function can have only one return value
- What if we want to return several values – for instance a whole matrix?
- Solution: Return a vector
- Careful: Then the function call has to be assigned to a vector in the main function (otherwise the result is lost)

Problem 3

- Let $v = (v_0, \ldots, v_{n-1})$ and $(w_0, \ldots, w_{n-1})$ be vectors of length n
- The sum is defined entry – wise:

  $v + w = (v_0 + w_0, \ldots, v_{n-1} + w_{n-1})$
- Write a C++ function that adds two vectors with entries of type double and returns the result
- If the sizes of the two vectors are not the same, print an error message to the screen and return an empty vector.
- Test the function.

Function Declaration

Function Declaration

```
returnType FunctionName(parameters);
```

- A function declaration is just the function head plus a semicolon
- A function declaration is also called function prototype
- The parameter names can be omitted. However, the parameter types must not be omitted
- Purpose: we can use a function which is defined in another source file or later in the same file if we put a function declaration before we call the function

Example 5 (Function Declaration)

Task:

- Define a function **IsOdd** which is called in the main function
- We get a compiler error if we define the function **after** the main function
- We only need to include a function declaration before the main function to make it work

Function Definition before main Function (correct)

```
 1 bool IsOdd(int n)
 2 {
 3       if(n%2==1)
 4         return true;
 5       return false;
 6 }
 7
 8 int main()
 9 {
10      cout << IsOdd(10) << endl;
11      system("PAUSE");
12 }
```

Incorrect Version

```
 1 int main()
 2 {
 3      cout << IsOdd(10) << endl;
 4      system("PAUSE");
 5 }
 6
 7 bool IsOdd(int n)
 8 {
 9       if(n%2==1)
10         return true;
11       return false;
12 }
```

Compiler error: in line 3 the compiler doesn't know yet that the function exists

Correct Version with Function Declaration

```
 1 bool IsOdd(int n);
 2
 3 int main()
 4 {
 5     cout << IsOdd(10) << endl;
 6     system("PAUSE");
 7 }
 8
 9 bool IsOdd(int n)
10 {
11      if(n%2==1)
12        return true;
13      return false;
14 }
```

Problem 4

- Add a second empty .cpp source file to your C++ project
- Define a function **Test** in that source file
- Put a declaration of the function **Test** before the main function (in the other .cpp file containing the main function)
- Call **Test** in the main function
- Check that you get a compiler error if you remove the function declaration

Default Function Parameters

Default Parameters

- Some function parameters are only used rarely
- In such a case, a default value for the parameter can suffice
- Default values can be specified in the function declaration or definition
- Default values cannot be re-specified in the function declaration or definition
- Default values are only used by the function if no values are substituted for them in the function call
- There can be several default parameters, but they must be at the end of the list of parameters

Default Parameters Examples

```
1  #include<iostream>
2  #include<iomanip>
3  using namespace std;
4
5  void PrintDouble(double x, int prec=10)
6  {
7       cout << setprecision(prec) << x << endl;
8       // prints x with a precision of prec digits
9  }
10
11 int main()
12 {
13      PrintDouble(0.123456789123);
14      // prints 10 digits; default value 10 is used for prec
15
16      PrintDouble(0.123456789012,3);
17      // prints 3 digits; default value is overridden
18
19      system("pause");
20 }
```

Default Parameters Examples

```
1  #include<iostream>
2  #include<iomanip>
3  using namespace std;
4
5  void PrintDouble(double x, int prec=10);
6
7  int main()
8  {
9      PrintDouble(0.123456789123);
10     system("pause");
11 }
12
13 void PrintDouble(double x, int prec=10)   // error
14 {
15      cout << setprecision(prec) << x << endl;
16 }
```

Compiler error – default parameters cannot be re-specified, even with the same value

Default Parameters Examples

```
1  #include<iostream>
2  #include<iomanip>
3  using namespace std;
4
5  void PrintDouble(int prec=10, double x)   // error
6  {
7       cout << setprecision(prec) << x << endl;
8  }
9
10
11 int main()
12 {
13      PrintDouble(0.123456789123);
14      system("pause");
15 }
```

Compiler error – default parameters must be at the end of the list of parameters

Problem 5

Write and test a function
vector<double> RandVector(int n,double a,double b)
that returns a vector of size n whose entries are random numbers in the interval [a,b].

By default, a vector of random numbers in [0,1] should be returned.

Hint: A random number x in [a,b] can be created by
double x, max = RAND_MAX;
x = (rand()/max)*(b-a)+a;
(needs #include<cstdlib>)

Function Overloading

Function Overloading

- Functions with the same name can be used if they have different kinds of parameters
- “Different” means different in the type or in the number of parameters
- When the function is called, the compiler determines through the parameter values which function has to be used
- Parameters with default values do not count for function overloading
- If type conversion are involved, ambiguities have to be avoided

Function Overloading Examples

```
1  #include<iostream>
2  using namespace std;
3
4  void Print(double x)
5  {
6        cout << x << " (double)" << endl;
7  }
8
9  void Print(int x)
10 {
11       cout << x << " (integer)" << endl;
12 }
13
14 int main()
15 {
16     Print(1.0); // first function is used
17     Print(1);   // second function is used
18     system("pause");
19 }
```

Function Overloading Examples

```
1  #include<iostream>
2  using namespace std;
3
4  void Print(double x)
5  {
6        cout << x << " (double)" << endl;
7  }
8
9  void Print(float x)
10 {
11       cout << x << " (integer)" << endl;
12 }
13
14 int main()
15 {
16     Print(1);    //error
17     system("pause");
18 }
```

Error – compiler cannot decide if 1 should be converted to double or float

Function Overloading Examples

```
1  #include<iostream>
2  using namespace std;
3
4  void Print(double x)
5  {
6        cout << x << " (double)" << endl;
7  }
8
9  void Print(double x, int y = 10)
10 {
11       cout << y << " (integer)" << endl;
12 }
13
14 int main()
15 {
16     Print(1.0); // error
17     system("pause");
18 }
```

Error – default parameters do not count for overloading, compiler cannot determine which function to use

Problem 6

Write and test the functions

int Add(vector<int> a)

double Add(vector<double> a)

(defined at global scope in one .cpp file) that return the sum of the entries of the respective vectors.

Pointers

Pointers

- A pointer is a variable whose value is the memory address of another variable. We say it points to that address.
- Pointers can be used to access the values of variables indirectly and often more efficiently
- Assign **NULL** to pointers for which a suitable address in not yet available (**NULL** pointer)
- Pointers are often used in libraries which are written in C (not C++). To use these libraries, we need pointers.

Pointer Syntax

```
type* pointerName;
```

Declaration of pointer to type.

```
pointerName = &variable;
```

Assigns the memory address of variable to the pointer. The variable must exist already.

```
*pointerName
```

Yields the value of the variable pointed to. This is called dereferencing.

Example: Pointers

```
 1 #include <cstdlib>
 2 #include <iostream>
 3 using namespace std;
 4
 5 int main(int argc, char *argv[])
 6 {
 7     int x=100, y=1000;
 8     int* pointer = NULL;
 9     pointer = &x;
10     cout << "value of x: " << *pointer << endl;
11     cout << "value of x: " << x << endl;
12     *pointer+=50;
13     cout << "new value of x: " << x << endl;
14     pointer = &y;
15     cout << "value of y: " << *pointer << endl;
16     system("PAUSE");
17     return EXIT_SUCCESS;
18 }
```

Explanations see next slide

Explanations

- **Line 8:** declaration of **NULL** pointer to type **int**
- **Line 9:** assigns the address of **x** to the pointer. We say the pointer "points to the address of **x**"
- **Line 10: *pointer** accesses the value of **x** (dereferencing)
- **Line 11**: **x** yields the same value as ***pointer**
- **Line 12:** changes the value of **x** indirectly through pointer
- **Line 13:** verify that the value of **x** has changed
- **Line 14:** assigns the address of **y** to pointer
- **Line 15:** verify that pointer now points to a new address

Pointers and Arrays

- Let **A** be the name of an array with entries of type **int** (similar for other types)
- Then **A** is also a pointer to the first element of the array
- So ***A** is the same as **A[0]**
- But **A** is a constant pointer – its address cannot be changed
- Pointer arithmetic: **int* p=A; p+=n;** has the effect that **p** points to the nth element of **A**
- But **A+=n;** is wrong since **A** is a constant pointer

Example

```
double A[3];
A[0]=234.1; A[1]=1000.2; A[2]=100.3;
double* p =A;
cout << *p << endl; // output 234.1
p+=2;
cout << *p << endl; // output 100.3
p--;
cout << *p << endl; // output 1000.2
```

Problem 7

- Initialize the entries of an array **A** of size 100 to random numbers
- Compute the sum of entries of **A** in two ways:

  1. As usual, sum up the **A[i]**

  2. Define a pointer **p** to the first element of **A**, let **p** run over the addresses of all elements of **A** using pointer arithmetic, and sum up the values ***p**

Passing Parameters to a Function

4 Ways to Pass Parameters

- By value
- By reference
- By pointer
- By constant reference

By pointer was “standard” for C programs, by constant reference is standard for C++ programs

Pass by Value

```
int Add(int a,int b)
{ return a+b;}
…
int c=5; d=10;
cout << Add(c,d) << endl;
```

When **Add(c,d)** is called, the following happens:

- Temporary copies of **c** and **d** are created and passed to the function (pass by value)
- The sum of the values of these temporary copies is returned
- The temporary copies of **c** and **d** are destroyed

Implications of Pass by Value

- If variable is passed by value to a function, its value cannot be changed by the function call
- Possible inefficiency if parameter is “large” (e.g. large array) – creating temporary copies takes time

Problem 8

- Write a function **void Change(int x)** that attempts to increase the value of x by 1. Check that a function call **Change(y)** has no effect on the value of an **`int`** variable **y**.

- Write a function **void Test(vector<int> v)** that does nothing and call it 1000 times in the main function with a vector of size 1000,000 as parameter. Take note of how much time the program execution takes.

Pass by Reference

- If we put **&** after the type of a function parameter, no temporary copy of the parameter is created and the function works directly on the parameter
- This is called pass by reference
- Avoids possible inefficiency, allows function to change value of a parameter
- Example:

```
int Add(int& a, int& b)
{
   return a+b;
}
```

Problem 9

- Write a function **void Change(int& x)** that attempts to increase the value of x by 1. Check that a function call **Change(y)** indeed changes the value of an **int** variable **y**.

- Write a function **void Test(vector<int>& v)** that does nothing and call it 1000 times in the main function with a vector of size 1000,000 as parameter. How does this compare with the previous version without **&** ?

Pass by Pointer

- If we put ***** after the type of a function parameter, the address of a variable is passed to the function, not the variable itself. This is called pass by pointer
- Means in a function call we must pass an address of a variable
- Pass by pointer also avoids possible inefficiency and allows function to change the value of a parameter.

Example:

```
int Add(int* a, int* b)
{
   return *a + *b;
}
int c=5, d=10;
cout << Add(&c,&d);
```

Problem 10

- Write a function **void Change(int* x)** that attempts to increase the value of x by 1. Check that a function call **Change(&y)** indeed changes the value of an **int** variable **y**.
- Write a function **void Test(vector<int>* v)** that does nothing and call it 1000 times in the main function with a vector of size 1000,000 as parameter. How does this compare with the version without ***** ?

Pass by Constant Reference

- If a parameter of a function is declared a constant reference, the function cannot change its value
- This is standard practice to make clear in the function head already that a function does not change parameter values
- Example:

```
int Add(const int& a, const int& b)
{
   return a+b;
}
```

(“const” not necessary, but good practice)

Dynamic Memory Allocation

Arrays of Variable Size

- “Variable size” means that the size can be changed while the program is running (“at runtime”)
- Often we can use vectors instead of arrays if we need variable size, but sometimes the use of arrays cannot be avoided
- Need to know how to create and delete arrays of variable size (“dynamic memory allocation”)

How Dynamic Memory Allocation cannot be done

```
int n;
cin >> n;
int A[n];
```

- This is an attempt to let the user determine the size of **A** at runtime
- Compiler error! (usually)
- Correct is to use the operators **new** and **delete**

New and Delete

```
int *A;
int n;
cin >> n;
A=new int[n];      // now A is array of size n
...                // do something with A
delete [] A;
```

- This is a correct way to create an array of variable size
- The delete command is necessary, otherwise the memory assigned to **`A`** is not released until the program terminates (“memory leak”)

Problem 11

Assume a file **input.txt** contains an unknown number of integers. Write a program that does the following:

- Determine how many integers are in the file, say **n**
- Create an array of size **n** using dynamic memory allocation
- Store the integers from **input.txt** in the array
- Print the first and last entry and the size of the array to the screen
- Apply the program to the files **input1.txt** and **input2.txt** (available in NTULearn)
- Hint: To reset an input stream **in** to the beginning of the file, use **in.clear(); in.seekg(ios::beg);**

Additional Problems

Problem 12 (Recursive Function)

- A C++ function can call itself (recursive function)
- Test the following function:

```
int factorial (int n)
 {
    if (n==0)
        return 1;
    return n*factorial(n-1);
}
```

Problem 13 (Euclidean Algorithm)

- Informally, the Euclidean algorithm for finding the gcd of two positive integers m,n can be described as follows.

  If m=n then return m.
  If m<n use gcd(m,n)=gcd(m,n mod m),
  if m>n use gcd(m,n)=gcd(m mod n,n)
  to recursively compute the gcd with the convention gcd(0,a)=gcd(a,0)=a for all positive integers a.

- Implement this algorithm as a recursive C++ function.

Problem 14

1 Partitions of Integers and $p(n)$

A *partition* of a positive integer n is a decreasing sequence of positive integers with sum n (the sequence does *not* need to be strictly decreasing). The number of different partitions of n is denoted by $p(n)$. Despite its simple definition, the function $p(n)$ has kept mathematicians busy for centuries

Here is a table of partitions and $p(n)$ for $n \leq 5$:

n	$p(n)$	partitions
1	1	1
2	2	2=1+1
3	3	3=2+1=1+1+1
4	4	4=3+1=2+2=2+1+1+1=1+1+1+1
5	7	5=4+1=3+2=3+1+1=2+2+1=2+1+1+1=1+1+1+1+1

Problem 14 (continued)

Let $P(n,k)$ be the number of partitions of n with largest part at most k. For instance, $P(5,2) = 3$, $P(5,3) = 5$, $P(5,4) = 6$ and $P(5,5) = 7$. Note $P(n,1) = 1$ for all $n \geq 1$. To facilitate a recursion we set

$$P(n,k) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{if } k < 1 \text{ or } n < 0 \end{cases}$$

(yes, this is not well defined if $n = 0$ and $k < 1$, but this case does not occur in the computations...). Then

$$P(n,k) = P(n-k,k) + P(n,k-1)$$

for all $k, n \geq 0$ (why?) Use this recursion to write a recursive C++ function `int P(int n, int k)` which returns $P(n,k)$. A recursive function calls itself inside the function body. This is no problem in C++.
Note $p(n) = P(n,n)$. Use this to calculate $p(n)$ for $n \leq 50$.

Hint to check correctness: $p(50) = 204226$.