Programming Expertise
C++ Classes

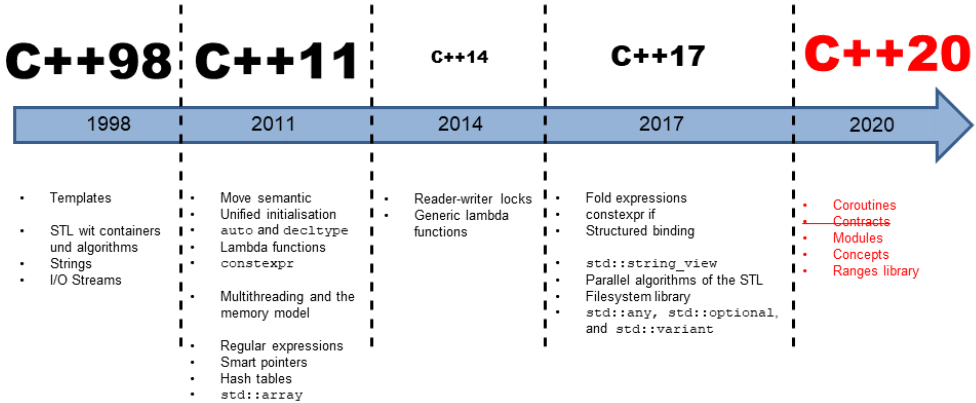
Kappel & Groth
2023-06-15

2 Pointers, References and C++ Classes

Outline

Special function features 5
Call by references 11
Constant function arguments 18
Function overloading 20
Lambda functions 29
Using arrays 34
Classes 41
Class constructors 63
Exercise C++ Classes 80

C++ History



https://www.heise.de/developer/artikel/Das-naechste-grosse-Ding-C-20-4560939.html

http://www.modernescpp.com/index.php/c-20-an-overview

Modern C++

C++98 .. C++17

```
#include <iostream>
#include <vector>
#include <algorithm>
std::vector<int> v {1,3,4};
std::sort( v.begin(), v.end() );
std::cout << "Values: " << v[0] <<
       " " << v[1] << "!" << std::endl;
```

C++20

```
import std.core
import std.format
auto v {1,3,4};
std::ranges::sort( v );
std::cout << std::format("Values {} {}!\n", v[0],v[1]);
```

Special function features: inline functions

- should be only used if really speed is a problem
- body of `inline` functions is copied into the caller scope
- speed improvements as number of jumps between functions is reduced
- but size increase as same code is copied into multiple places
- useful for very short functions
- `inline` is a proposal for the compiler
- don't use `#define` macros, better use inline

```
#include <iostream>

inline int Double (int x) ;
int main() {
    int x = 5;
    std::cout << "Hello  double of x is " <<
        Double(x) << std::endl;
    return(0);
}
inline int Double (int x) {
    return(x*2);
}
```

$ g++ -o inline.cpp.bin -std=c++17 -fconcepts inline.cpp &&
./inline.cpp.bin

```
Hello  double of x is 10
```

⇒ Non-essential feature - I do not recommended to use it!

Recursive Functions

$\Rightarrow$ small elegant, but often inefficient

```
#include <iostream>
int fib (int x) {
    if (x < 3) {
        return(1) ;
    }
    return(fib(x-2)+(fib(x-1)));
}
int main () {
    for (int x = 1; x < 8; x++) {
        std::cout << "Fibonacci  of " << x
            << "=" << fib(x) << std::endl;
    }
    return(0);
}
```

```
$ g++  -o fibrec.cpp.bin -std=c++17 -fconcepts fibrec.cpp &&
./fibrec.cpp.bin
Fibonacci  of 1=1
Fibonacci  of 2=1
Fibonacci  of 3=2
Fibonacci  of 4=3
Fibonacci  of 5=5
Fibonacci  of 6=8
Fibonacci  of 7=13
```

Iterative functions

$\Rightarrow$ often more efficient but less elegant

```
#include <iostream>
int fib (int x) {
    int y=1; int z=1;
    int tmp=0;
    if (x < 3) {
        return(1) ;
    }
    for (int i = 3; i <= x; i++) {
        tmp=y;
        y=z;
        z=tmp+y;
    }
    return(y+z);
}
```

```
int main () {
    for (int x = 1; x < 8; x++) {
        std::cout << "Fibonacci  of " << x
            << "=" << fib(x) << std::endl;
    }
    return(0);
}
```

$ g++ -o fibit.cpp.bin -std=c++17 -fconcepts fibit.cpp && ./fibit.cpp.bin

```
Fibonacci  of 1=1
Fibonacci  of 2=1
Fibonacci  of 3=3
Fibonacci  of 4=5
Fibonacci  of 5=8
Fibonacci  of 6=13
Fibonacci  of 7=21
```

$\Rightarrow$ convert recursive into iterative functions if speed is an issue.

Pointers and References

- Questions: how to return more than one value/item?
- Answer 1: in principle you can't – but you can write functions which are called with references as arguments.
- Answer 2: you can return a data structure which keeps more than one value, such as array, vector, list, map etc.
- Calls:
  - call by value - argument variable can't be changed
  - call by reference - argument variable can be changed
    - using pointers
    - using references (C++ only)
- Call by references:
  - to change the input variable (not so recommended)
  - for efficiency reasons (no copy of large data structures)

Call by references using pointers

```
#include <iostream>
void swap (int *n, int *m) ;

int main () {
    int x = 4;
    int y = 5;
    std::cout << "Before swap x=" << x << " - y="
         << y << std::endl;
    swap(&x,&y);
    std::cout << "After  swap x=" << x << " - y="
        << y << std::endl;
    return 0;
}
```

```
void swap (int *n, int *m) {
    int temp=0;
    temp=*m;
    *m=*n;
    *n=temp;
    return;
}
```

$ g++ -o pswap.cpp.bin -std=c++17 -fconcepts pswap.cpp && ./pswap.cpp.bin

```
Before swap x=4 - y=5
After  swap x=5 - y=4
```

Remember if x is a pointer:

- `*x` is the value and `x` is the memory address
- but assignment is like this: `*x = value`

Remember if x is a normal variable:

- `x` is the value and `&x` is the memory address (pointer)

Call by references using references

References:

A reference variable is an alias, that is, 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. [Tutorialspoint — C++ References, 2019]

```
#include <iostream>
void swap (int &rn, int &rm) ; // only here special symbols
int main () {
    int x = 6;
    int y = 7;
    std::cout << "Before swap x=" << x << " - y="
         << y << std::endl;
```

```
    swap(x,y); // easier to read
    std::cout << "After  swap x=" << x << " - y="
        << y << std::endl;
    return 0;
}
void swap (int &rn, int &rm) { // only here special symbols
    int temp=0;
    temp=rm;
    rm=rn;
    rn=temp;
    return;
}
```

$ g++ -o rswap.cpp.bin -std=c++17 -fconcepts rswap.cpp &&
./rswap.cpp.bin

```
Before swap x=6 - y=7
After  swap x=7 - y=6
```

⇒ from 8 stars and 2 &'s with pointers
to now 4 &'s with references!
⇒ references here only required in the function argument list are easier to use
⇒ some uncertainty: will this function change my variables - `swap(x,y)` ?
⇒ see later for `const`
⇒ but can't provide pointer artithmetics
⇒ but can't be reassigned (next example)
⇒ Hint: Restrict the use to function arguments ...

References can't be reassigned

```
#include <iostream>
int main () {
    int x = 6;
    int y = 7;
    int &rx = x; // rx is a reference on x
    std::cout << "x=" << x << " rx=" << rx << " y="
         << y << std::endl;
    rx=y; // oops, don't do this, x is changed ...
    std::cout << "x=" << x << " rx=" << rx << " y="
         << y << std::endl;
    return 0;
}
```

```
$ g++  -o reass.cpp.bin -std=c++17 -fconcepts reass.cpp && ./reass.cpp.bin
x=6 rx=6 y=7
x=7 rx=7 y=7
```

const function arguments

- sometimes you call functions per reference for efficiency reasons (large data structures normally) but not to modify the arguments
- to make clear that a variable even given as reference did not change its values you can use the `const` keyword
- any change to the variable will produce a compile error
  $\Rightarrow$ use const if you don't want to change the variable just like to copy the address of the object into the function

```
void foo(const int &x) // x is a const reference
{
    x = 6;  // compile error: a const reference cannot
    // have its value changed!
}
```

Some recommendations

Prefere references over pointers but:
⇒ Just use references in parameter lists.
⇒ Only use them for large data structures or if you like to modifiy the variable.
⇒ When passing structs or classes (use const if read-only).
⇒ When passing an argument by reference, always use a const reference unless you need to change the value of the argument.
When not to use pass by reference:
⇒ When passing fundamental types that don't need to be modified (use pass by value).
A tutorial: `https://www.learncpp.com/cpp-tutorial/73-passing-arguments-by-reference/`

Function overloading:

In some programming languages, function overloading or method overloading is the ability to create multiple functions of the same name with different implementations. Calls to an overloaded function will run a specific implementation of that function appropriate to the context of the call, allowing one function call to perform different tasks depending on context. [Wikipedia — Function overloading, 2019]

- add function for two values
- integers
- floats
- doubles

- ...
  ⇒ we need many functions ...
  ⇒ polymorphic functions

```
#include <iostream>
// evil C-way
#define dadd(x,y) ((x+y))
// tedious polymorphic way
int add (int x, int y) { return(x+y); }
float add (float x, float y) { return(x+y); }
double add (double x, double y) { return(x+y); }
double add (int x, float y) { return(x+y) ; }
// ... and more
int main () {
    int ix = 6;
    int iy = 7;
    float fx = 16.2;
    float fy = 17.2;
    double dx = 26.2;
    double dy = 27.2;
```

```
    std::cout << "Using tedious three functions ..." << std::endl;
    std::cout << "ix+iy=" << add(ix,iy) <<
                 " - fx+fy=" << add(fx,fy) <<
                 " - dx+dy=" << add(dx,dy) << std::endl;
    std::cout << "Using a C #define ..." << std::endl;
    std::cout << "ix+iy=" << dadd(ix,iy) <<
                 " - fx+fy=" << dadd(fx,fy) <<
                 " - dx+dy=" << dadd(dx,dy) << std::endl;
    return 0;
}
```

$ g++ -o pmfunc.cpp.bin -std=c++17 -fconcepts pmfunc.cpp && ./pmfunc.cpp.bin

```
Using tedious three functions ...
ix+iy=13 - fx+fy=33.4 - dx+dy=53.4
Using a C #define ...
ix+iy=13 - fx+fy=33.4 - dx+dy=53.4
```

⇒ C++17 we could use $auto$, auto as return type is allowed:

```
#include <iostream>
auto add (auto x, auto y) {
    // only one return type possible
    return(x+y);
}
// will compile with new compilers?  (chicken and egg problem)?
// just for illustr., usually we should use int or long here ...
auto afib (auto x) {
    if (x < 3) {
        return(1) ;
    }
    return(afib(x-2)+(afib(x-1)));
}
int main () {
    for (int x = 1; x < 8; x++) {
        std::cout << "Fibonacci  of " << x
```

```
            << "=" << afib(x) << std::endl;
    }
    std::cout << add(12,13) << std::endl;
    std::cout << add(12,13.1) << std::endl;
    return(0);
}
```

```
$ g++  -o afib.cpp.bin -std=c++17 -fconcepts afib.cpp && ./afib.cpp.bin
Fibonacci  of 1=1
Fibonacci  of 2=1
Fibonacci  of 3=2
Fibonacci  of 4=3
Fibonacci  of 5=5
Fibonacci  of 6=8
Fibonacci  of 7=13
25
25.1
```

Polymorphic functions with templates

```
#include <iostream>
template <typename T>
T add (T a, T b) {
  T result;
  result = a+b;
  return(result);
}
int main () {
    int ix = 6;
    int iy = 7;
    float fx = 16.2;
    float fy = 17.2;
    double dx = 26.2;
    double dy = 27.2;
    std::cout << "Template version:" << std::endl;
```

```
    std::cout << "ix+iy=" << add(ix,iy) <<
                 " - fx+fy=" << add(fx,fy) <<
                 " - dx+dy=" << add(dx,dy) <<  std::endl;
    return 0;
}
```

$ g++ -o tfunc.cpp.bin -std=c++17 -fconcepts tfunc.cpp && ./tfunc.cpp.bin

```
Template version:
ix+iy=13 - fx+fy=33.4 - dx+dy=53.4
```

$\Rightarrow$ but `add(dx,fy)` would not work!! $\Rightarrow$ but auto declared add would work!

Templates or auto? I prefer $auto$ in the Moment

Polymorphic functions with templates II

```
#include <iostream>
template <typename T, typename S>
T add (T a, S b) { // two types
  T result;
  result = a+b;
  return(result);
}
int main () {
    int ix = 6;
    int iy = 7;
    float fx = 16.2;
    float fy = 17.2;
    double dx = 26.2;
    double dy = 27.2;
    std::cout << "Mixing types:" << std::endl;
```

```
    std::cout << "ix+iy=" << add(ix,iy) <<
                 " - fx+fy=" << add(fx,fy) <<
                 " - dx+dy=" << add(dx,dy) <<
                 " - ix+fy+dy=" << add(add(ix,fy),dy) <<
                    std::endl;
    return 0;
}
```

$ g++ -o tfunc2.cpp.bin -std=c++17 -fconcepts tfunc2.cpp && ./tfunc2.cpp.bin

```
Mixing types:
ix+iy=13 - fx+fy=33.4 - dx+dy=53.4 - ix+fy+dy=50
```

⇒ only those functions required in the program will be created by the compiler, not all possible ones ...

⇒ `add(ix,fy)` would now work but would return an integer !!

⇒ still I prefer $auto$ ...

Lambda functions - C++11

Lambda: an unnamed function object capable of capturing variables in scope (function closure).

Syntax: `[ captures ] ( params ) { body }`

```
#include <iostream>
#include <algorithm>
#include <vector>

int main () {
 std::vector<int> v = {1,2,3,7,9,12,123};
  for (auto i : v) {
       std::cout << i << " " ;
  }
  std::cout << std::endl;
```

```
  std::transform(v.begin(), v.end(), v.begin(),
     [](int i) { // this is a comment inside lambda
            if (i > 3) {
                return 3;
            } else {
                return i;
            }
  });
  for (auto i : v) {
       std::cout << i << " " ;
  }
  std::cout << std::endl;
}
```

$ g++ -o lambda.cpp.bin -std=c++17 -fconcepts lambda.cpp && ./lambda.cpp.bin

```
1 2 3 7 9 12 123
1 2 3 3 3 3 3
```

Lambda captures

You can capture by both reference and value, which you can specify using & and = respectively:

+ [epsilon] - capture epsilon by value
+ [&epsilon] - capture by reference
+ [&] - captures all variables used in the lambda by reference
+ [=] - captures all variables used in the lambda by value
+ [&, epsilon] - captures variables like with [&], but epsilon by value
+ [=, &epsilon] - captures variables like with [=], but epsilon by reference

Lambda captures example

```
#include <iostream>
#include <algorithm>
#include <vector>

int main () {
 std::vector<int> v = {1,2,3,7,9,12,123};
 int limit = 10;
  for (auto i : v) {
       std::cout << i << " " ;
  }
  std::cout << std::endl;
  std::transform(v.begin(), v.end(), v.begin(),
      [limit](int i) {
          if (i > limit) {
              return limit;
```

```
            } else {
                return i;
            }
    });
    for (auto i : v) {
         std::cout << i << " " ;
    }
    std::cout << std::endl;
}
```

$ g++ -o lambda2.cpp.bin -std=c++17 -fconcepts lambda2.cpp && ./lambda2.cpp.bin

```
1 2 3 7 9 12 123
1 2 3 7 9 10 10
```

Using arrays

$\Rightarrow$ C-like (not recommended ...)

```
#include <iostream>

int main () {
   // an array with 5 rows and 2 columns.
   int a[5][2] = { {0,0}, {1,2}, {2,4}};
   // output each array element's value
   for ( int i = 0; i < 3; i++ )
      for ( int j = 0; j < 2; j++ ) {
         std::cout << "a[" << i << "][" << j << "]: ";
         std::cout << a[i][j]<< std::endl;
      }
   return 0;
}
```

```
$ g++  -o array.cpp.bin -std=c++17 -fconcepts array.cpp && ./array.cpp.bin
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
```

⇒ single and multidimensional arrays as in C

⇒ arrays can have only one type as in R

⇒ C++ has as well template facilities to work with arrays independently of a type

⇒ later in the course we will create a matrix class using vectors of vectors.

Passing arrays to functions (C vs C++)

```
#include <iostream>
#include <vector>
using namespace std;
// C-style with with size
double getAverage(int arr[], int size) {
  int sum = 0;
   for (int i = 0; i < size; ++i) { sum += arr[i]; }
   return(double(sum) / size);
}
double getAverage (vector<int> arr) { // C++ style
    double average = 0;
    for (auto i : arr) { average+=i; }
    return(average/arr.size());
}
int main () {
```

```
    int balance[5] = {1000, 2, 3, 17, 50};
    double avg;
    // pass pointer to the array as an argument.
    avg = getAverage( balance, 5 ) ;
    // output the returned value
    cout << "Average value is: " << avg << endl;
    vector<int> v = { 2, 3, 4, 5 };
    cout << "Average value vector is: " << getAverage(v) << endl;
    return 0;
}
```

```
$ g++  -o arraypass.cpp.bin -std=c++17 -fconcepts arraypass.cpp &&
./arraypass.cpp.bin
Average value is: 214.4
Average value vector is: 3.5
```

Which container???

We have map, unorderd map, multimap, unordered multimap, set, array, vector ...

Hint: Until having problems with memory and time use *vector* and *map* as data containers, you can ignore the others in 99% of the cases. Other containers can be imitated, for instance *multimaps* can imitated as *map* with *vectors*!

Matrix

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

typedef std::vector<std::vector<int> >  Matrix ;
int main() {
   Matrix mt = {
    {0,1,0,0},
    {0,0,0,0},
    {0,0,0,0},
    {1,0,1,0}};
    cout << mt[0][0] << endl;
    cout << mt[0][1] << endl;
    return(0);
}
```

```
$ g++   -o matrix.cpp.bin -std=c++17 -fconcepts matrix.cpp &&
./matrix.cpp.bin
0
1
```

Structures and Classes

- A `struct` is as user defined data type combining data items of different kinds.
- You can see a structure as a C++ class with only public variables but no methods.

```
#include <iostream>
#include <cstring>
using namespace std;
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};
```

```
int main() {
   // struct here is optional
   struct Books Book1;  // Declare Book1 of type Book
   struct Books Book2;  // Declare Book2 of type Book

   // book 1 specification
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;
```

```
    // Print Book1 info
    cout << "Book 1 title : " << Book1.title <<endl;
    cout << "Book 1 author : " << Book1.author <<endl;
    cout << "Book 1 subject : " << Book1.subject <<endl;
    cout << "Book 1 id : " << Book1.book_id <<endl;

    // Print Book2 info
    cout << "Book 2 title : " << Book2.title <<endl;
    cout << "Book 2 author : " << Book2.author <<endl;
    cout << "Book 2 subject : " << Book2.subject <<endl;
    cout << "Book 2 id : " << Book2.book_id <<endl;

    return 0;
}
```

$ g++ -o struct.cpp.bin -std=c++17 -fconcepts struct.cpp && ./struct.cpp.bin

```
Book 1 title : Learn C++ Programming
Book 1 author : Chand Miyan
Book 1 subject : C++ Programming
Book 1 id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Yakit Singha
Book 2 subject : Telecom
Book 2 id : 6495700
```

Structures as function arguments

```
#include <iostream>
#include <cstring>
using namespace std;
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};
void printBook( Books book );
int main() {
   Books Book1;  // Declare Book1 of type Book
   Books Book2;  // Declare Book2 of type Book

   // book 1 specification
```

```
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;
   printBook(Book1);
   printBook(Book2);
}

void printBook( Books book ) {
   cout << "Book title : " << book.title <<endl;
```

```
    cout << "Book author : " << book.author <<endl;
    cout << "Book subject : " << book.subject <<endl;
    cout << "Book id : " << book.book_id <<endl;
}
```

$ g++ -o structfunc.cpp.bin -std=c++17 -fconcepts structfunc.cpp && ./structfunc.cpp.bin

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

Class:

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class. [Tutorialspoint — C++ Classes and Objects, 2019]

- a `class` with just some public properties is like a `struct`
- `public` properties/methods can be accessed directly from any code outside of the class
- `protected` properties or methods can be accessed from classes/objects inheriting from that class
- `private` properties and methods can be used only inside of this class.
- access operator for member properties and methods is the dot

struct Books $\Rightarrow$ class Books I

```
#include <iostream>
#include <cstring>
using namespace std;
class Books {
public:
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;

};
void printBook( Books book ) ;
int main() {
   Books Book1;  // Declare Book1 of type Book
   Books Book2;  // Declare Book2 of type Book
```

```
   // book 1 specification
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   printBook(Book1);
   printBook(Book2);
   return 0;
}
```

```
void printBook( Books book ) {
   cout << "Book title : " << book.title << endl;
   cout << "Book author : " << book.author << endl;
   cout << "Book subject : " << book.subject << endl;
   cout << "Book id : " << book.book_id << endl;
}
```

$ g++ -o class.cpp.bin -std=c++17 -fconcepts class.cpp && ./class.cpp.bin

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

struct Books ⇒ class Books II

```
#include <iostream>
#include <string>
using namespace std;
class Books {
public:
   string  title   ="";
   string  author  ="";
   string  subject ="";
   int   book_id   = 0;
   void printBook() {
      cout << "Book title : " << title << endl;
      cout << "Book author : " << author << endl;
      cout << "Book subject : " << subject << endl;
      cout << "Book id : " << book_id << endl;
   }
};
```

```
int main() {
   Books Book1;  // Declare Book1 of type Book
   Books Book2;  // Declare Book2 of type Book

   // book 1 specification
   Book1.title   = "Learn C++ Programming";
   Book1.author  = "Chand Miyan";
   Book1.subject = "C++ Programming";
   Book1.book_id = 6495407;

   // book 2 specification
   Book2.title   = "Telecom Billing";
   Book2.author  = "Yakit Singha";
   Book2.subject = "Telecom";
   Book2.book_id = 6495700;
   cout << "Class Example 2 ...\n";
```

```
    Book1.printBook();
    Book2.printBook();
    return 0;
}
```

$ g++ -o class2.cpp.bin -std=c++17 -fconcepts class2.cpp && ./class2.cpp.bin

```
Class Example 2 ...
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

adding functions to a class

Member functions:

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object. [Tutorialspoint — C++ Class Member Functions, 2019]

$\Rightarrow$ let's create a class Animal with properties and methods, as they are more interesting than books, they can do more things than books can do ...

class Animal

```
#include <iostream>
#include <string>
using namespace std;
class Animal {
  public:
     int age = 0;
     // declare and define in class definition
     void eat() {
        cout << "I'm eating generic food. " << endl ;
     }
     // declare here only - define later
     void run (float km);
  protected:
    float km = 0.0;
    string name =  "Max Musteranimal";
}; // semicolon
```

```
// define here
void Animal::run (float akm) {
    cout << "I'm, " << name << ", running now: " << akm << "km" <<
    km=km+akm;
    cout << "Total  runnings: " << km << "km" << endl;
}

int main () {
    Animal animal;
    animal.eat();
    animal.run(3);
    animal.run(2.1);
}
```

$ g++ -o classanimal.cpp.bin -std=c++17 -fconcepts classanimal.cpp && ./classanimal.cpp.bin

```
I'm eating generic food.
```

```
I'm, Max Musteranimal, running now: 3km
Total  runnings: 3km
I'm, Max Musteranimal, running now: 2.1km
Total  runnings: 5.1km
```

Getter and setter functions:

Public variables are usually discouraged, and the better form is to make all variables private and access them with getters and setters. [Stackoverflow — Public or private variables, 2019]

```
class Animal {
    public:
        void setAge (int age);
        int getAge () const ;
        // const means does not change anything
        // within the object, here it is optional
    protected:
        int itsAge = 0;
};
```

http://www.gotw.ca/publications/c++cs.htm

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

class Animal {
  public:
     // declare and define in class definition
     void eat() {
        itsWeight += 0.5;
        cout << "I'm eating generic food. " << endl ;
     }
     // getter and setter methods
     int getAge () const { return (itsAge); }
     void setAge (int age ) { itsAge=age; return;}
     float getWeight () const { return (itsWeight); }
     // declare here only - define later
```

```
    void run (float km);
  protected:
    float itsKm = 0.0;
    string itsName =  "Maxi Musteranimalin";
    int itsAge = 0;
    float itsWeight = 0.0;
};

// define here
void Animal::run (float km) {
    cout << "I'm running now: " << km << "km" << endl;
    itsKm += km;
    itsWeight -= km*0.1;
    cout << "Total  runnings: " << itsKm << "km" << endl;
}
```

```
int main () {
    Animal animal;
    animal.eat();
    animal.run(3);
    animal.eat();
    animal.run(2.1);
    cout <<  "Current weight: " << animal.getWeight() << endl;
}
```

$ g++ -o classanimalsetter.cpp.bin -std=c++17 -fconcepts classanimalsetter.cpp && ./classanimalsetter.cpp.bin

```
I'm eating generic food.
I'm running now: 3km
Total  runnings: 3km
I'm eating generic food.
I'm running now: 2.1km
Total  runnings: 5.1km
Current weight: 0.49
```

Constructors

Constructor:

A class constructor is a special member function of a class that is executed whenever we create new objects of that class. A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

```
#include <iostream>
using namespace std;
class Animal {
  public:
     // constructor 1
     Animal(int age) { itsAge = age ;}
     // constructor 2 (standard)
     Animal() { itsAge=0;}
     void eat() {
        itsWeight += 0.5;
        cout << "I'm eating generic food. " << endl ;
     }
     // getter and setter methods
     int getAge () { return (itsAge); }
     void setAge (int age ) { itsAge=age; return;}
     float getWeight () { return (itsWeight); }
```

```
    // declare here only - define later
    void run (float km);
  protected:
    float itsKm = 0.0;
    char itsName[50] =  "";
    int itsAge = 0;
    float itsWeight = 0.0;
};

// define here
void Animal::run (float km) {
    cout << "I'm running now: " << km << "km" << endl;
    itsKm += km;
    itsWeight -= km*0.1;
    cout << "Total  runnings: " << itsKm << "km" << endl;
}
```

```
int main () {
    Animal animal;
    animal.eat();
    animal.run(3);
    animal.eat();
    animal.run(2.1);
    cout <<"Current weight: " <<animal.getWeight() <<endl;
    cout <<"Animals age: " <<animal.getAge()<< endl;
    Animal animal2(12);
    cout <<"Current weight: " <<animal2.getWeight() <<endl;
    cout <<"Animals age: " <<animal2.getAge() << endl;
}
```

$ g++ -o classconstr.cpp.bin -std=c++17 -fconcepts classconstr.cpp && ./classconstr.cpp.bin

```
I'm eating generic food.
I'm running now: 3km
```

```
Total  runnings: 3km
I'm eating generic food.
I'm running now: 2.1km
Total  runnings: 5.1km
Current weight: 0.49
Animals age: 0
Current weight: 0
Animals age: 12
```

Outside constructor definitions

```
#include <iostream>
using namespace std;
// just declarations within the class body
// public part is the published interface for programmers
class Animal {
  public:
     // constructor 1
     Animal(int age);
     // constructor 2
     Animal();
     int getAge();
  protected:
    int itsAge = 0;
};
```

```
// the implementation, details
// normally not required to be read by the programmer
int Animal::getAge () {
    return(itsAge);
}

// no argument constructor
Animal::Animal () {
    cout << "simple animal creation ..." << endl;
}

// parameterized constructor
Animal::Animal (int age) {
    cout << "complex animal creation ..." << endl;
    itsAge=age;
}
```

```
// use case
int main () {
    Animal animal;
    cout <<"Animal 1's age: " <<animal.getAge() << endl;
    Animal animal2(12);
    cout <<"Animal 2's age: " <<animal2.getAge() << endl;
}
```

$ g++ -o classout.cpp.bin -std=c++17 -fconcepts classout.cpp && ./classout.cpp.bin

```
simple animal creation ...
Animal 1's age: 0
complex animal creation ...
Animal 2's age: 12
```

Using Initialization Lists

without list:

```
class Animal {
  public:
     // constructor 1
     Animal(int age) { itsAge = age ;}
  protected:
    int itsAge = 0;
};
```

same but with initialization list:

```
class Animal {
  public:
     // constructor 1
     Animal(int age): itsAge(age) { }
  protected:
    int itsAge = 0;
};
```

When to use initialization lists

1) For initialization of non-static const data members:
const data members must be initialized using initializer lists. In the following example, 't' is a const data member of class "Test" and is initialized using an initializer list.

```
#include <iostream>
#include <string>
using namespace std;
class Animal {
  public:
    Animal(string name):name(name) {} //Initializer list
    string getName() const { return name; }
  private:
    const string name ; // should not change after birth
};
```

```
int main() {
    Animal fido("Fido the Collie");
    cout << fido.getName() << endl;
    return 0;
}
```

$ g++ -o classinitlist.cpp.bin -std=c++17 -fconcepts classinitlist.cpp && ./classinitlist.cpp.bin

```
Fido the Collie
```

⇒ other example of use: students matrikel number

⇒ initialize at students creation but never, ever allowed to be changed!!

2) For initialization of reference members:
Reference members must be initialized using initializer lists. In the following example, 't' is a reference member of class "Test" and it is initialized using an initializer list.

```
// Initialization of reference data members
#include <iostream>
#include <string>
using namespace std;

class Animal {
   int &age;
public:
   Animal (int &age):age(age) {}
   int getAge() const { return age; }
};
```

```
int main() {
    int age = 2;
    Animal fido(age);
    cout << fido.getAge() <<endl;
    age = 3;
    cout << fido.getAge() <<endl;
    return 0;
}
```

```
$ g++   -o classinitlistref.cpp.bin -std=c++17 -fconcepts classinitlistref.cpp
&& ./classinitlistref.cpp.bin
2
3
```

⇒ There are more, more complex cases:

`https://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/`

⇒ Athough the reference approach above is legal C++ code, I did not recommend to program this way as it breaks encapsulation.

$\Rightarrow$ Only useful may be if you you make references to huge external data structures.

Summary

function features:

- (inline - many features, but don't use all of them)
- recursive vs iterative
- call by value vs call by references
- lambda functions
- overloading functions, templates and auto for C++
- define directives are deprecated in C++
- auto a useful choice for polymorphic functions (?)
- with auto be sure to return only one type return(x) and return(y) in the same function if x is integer and y is float will not work

structs and classes:

- differences, similarities
- creation of classes
- class member variables
- class member functions (methods)
- constructors and initialization
- const member functions and const variables

Next week

- string class
- regular expressions
- destructors
- inheritance
- virtual functions
- abstract classes
- static variables and methods

Exercise C++ Classes

Task 1 - Login and Workspace:

Exercise and Homework will be updated probably on Tuesday or Wednesday!

Task 4 - Commandline arguments

- have a look at using C++ applications with commandline arguments here: `https://www.geeksforgeeks.org/command-line-arguments-in-c-cpp/`
- if the users supplies a commandline argument belonging to a menu point your application should start this exercise with all default settings and thereafter it should go back into menu mode
- to achieve this add an additional argument to the menu method entry which is a single letter char, which defaults to 'x'
- delegate the commandline argument to the menu function
- if an invalid character was entered just the standard menu should start

Homework

- complete the tasks above
- extend the calculator with an option to display single letter coded amino acids codes where the user should type three letter codes as answer.
- so the user should get a single letter code and guess the triple letter code.
- if the user guessed wrong, after some time the letter will shown him again
- this stops until all single letter codes were guessed correctly or after a limited number of guesses
- you might used the c++ data structure `std::map` for instance if you like
- see for some use cases: `https://www.geeksforgeeks.org/searching-map-using-stdmap-functions-c/`
- or this tutorial: `https://thispointer.com/stdmap-tutorial-part-1-usage-detail-with-examples/`

References

Tutorialspoint — C++ References. Tutorialspoint, SIMPLYASLEARNINMG, 2019. URL https://www.tutorialspoint.com/cplusplus/cpp_references.htm. [Online; accessed 4-June-2019].

Wikipedia — Function overloading. Wikipedia, The Free Encyclopedia, 2019. URL https://en.wikipedia.org/wiki/Function_overloading. [Online; accessed 4-June-2019].

Tutorialspoint — C++ Classes and Objects. Tutorialspoint, SIMPLYASLEARNINMG, 2019. URL https://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm. [Online; accessed 4-June-2019].

Tutorialspoint — C++ Class Member Functions. Tutorialspoint, SIMPLYASLEARNINMG, 2019. URL https://www.tutorialspoint.com/

cplusplus/cpp_class_member_functions.htm. [Online; accessed 4-June-2019].

Stackoverflow — Public or private variables. Stackoverflow – Learn, Share, Build, 2019. URL https://stackoverflow.com/questions/14399929/should-i-use-public-or-private-variables. [Online; accessed 4-June-2019].