CH-230-A

Programming in C and C++

C/C++

Lecture 11

Dr. Kinga Lipskoch

Fall 2020

The Copy Constructor

To correctly manage by value argument passing for objects it is necessary to define a copy constructor:

- For class X a copy constructor has the form:

```
X::X(const X&);
```

- If defined, this will replace bit-copy (exact, bit by bit copy of an object) when passing by value object parameters
- Its goal is to correctly create a copy of an object starting from an existing one
- copyconstructor.cpp

Compiler Generated Constructors

To summarize, the compiler can generate two types of constructors:

- The default constructor, taking no arguments
  - This is generated only if you do not provide any constructor
- The copy constructor, which performs bit-copy initialization from an existing object
  - This is not generated if you either provide an `X::X(const X&)` implementation or you declare a `private X::X(const X&)` constructor
  - The `private X::X(const X&)` constructor does not need to be implemented

C++ References (1)

A reference is a constant pointer which is automatically dereferenced and that has to be initialized when it is created

- Constant means that it cannot be modified to reference a different entity
  - But you can of course modify what it is pointing to
- Cannot reference NULL

```
int a = 3;
int &reference = a;
reference++;
cout << a;        // prints 4
```

C++ References (2)

- References create a “synonym” (alias)
  - Previous example: acting on a reference is the same that acting on the variable `a`
- The first use of references is for creating functions and methods having out parameters
  - Indeed C++ references can be used even if you do not exploit the object-oriented capabilities of the language
- `outparameters.cpp`

outparameters.cpp

```
#include <iostream>
using namespace std;
/* This function takes two parameters. Only modifications
   done on the first one are visible outside. */
void oldStyle(int *outval, int inval)
{
  cout << "Inside oldStyle" << endl;
  inval++;
  (*outval)++;                      // need to dereference
}
/* Also this function takes two parameters. Again, only
  modifications done on the first one are visible outside. */
void newStyle(int &outval, int inval)
{
  cout << "Inside newStyle" << endl;
  inval++;
  outval++;                         // no need to dereference
}
int main(int argc, char** argv)
{
  int a = 0, b = 0;
  cout << a << " " << b << endl;
  oldStyle(&a, b);                 // needs to take the address
  cout << a << " " << b << endl;
  a = b = 0;                       // reset to initial values
  newStyle(a, b);                  // no specific syntax to pass the parameter
  cout << a << " " << b << endl;
  return 0;
}
```

Passing `const` Object References

The usual way to pass an input parameter to a function or method, is to pass it as a `const` reference

- Improved efficiency + cannot modify
  - No need to create a temporary copy of the object
- No need to define a copy constructor

```
void method(const string& byvaluepar) {
  // use it as a constant object
}
```

- All previous examples should be rewritten according to this indication

Passing Objects by Reference

- Another case: use a reference when you wish to pass an object by reference, i.e., if modifications have to be seen outside

```
void modifyString(string& tomod) {
  tomod.assign("new value");
  // non const as modification has to be seen
}
```

- The use is consistent with basic data types

Dynamic Memory Allocation

- C++ has an operator for dynamic memory allocation
  - It replaces the use of the C `malloc` function
  - Easier and safer
- The operator is called `new`
  - It can be applied both to user defined types (classes) and to native types

Using `new` for Predefined Data Types

- The operator returns a pointer to a specified type
- It automatically calculates the amount of memory necessary
  - It only requires the type and the number of "objects" to hold
- `newarrays.cpp`
- Note: same syntax for pointers to different types (no casting needed)

Dynamic Memory Deallocation

- As `malloc` has the companion function `free`, the operator `new` is coupled with the operator `delete`, which removes an object from memory
- `delete` requires the address of the object(s) to be deleted from the memory

```
int *a, *b;
a = new int;
b = new int[40];
delete a;
delete []b;
```

More on `delete`

- When `delete` is called to remove an object, the destructor is invoked before removing it
- Calling `delete` twice (or more) on the same object will result in an undefined behavior
- Calling `delete` on a `NULL` pointer will do nothing
  - Thus it could be advisable to set a pointer to `NULL` after calling `delete` (further `delete` will have no effect)
- Do not mix calls to `new` / `delete` with `malloc` / `free` – similar purpose, but predictably very bad result

`new` and `delete` for Arrays of Objects

- It is possible to dynamically create arrays of objects (instances of classes)
  - There must be a default (empty) constructor for the class (which will be called for every element of the array)
  - An array of objects must be explicitly deleted, by using the following syntax
    `delete []ptr;`
  - In this way the compiler is able to call the destructor for every element before freeing memory
  - There is no need to specify how many elements
- `Student2.h`
- `Student2.cpp`
- `studentsrevised.cpp`

Even More on `delete`

- When you create objects via `new`, you should destroy them via `delete`
  - All the memory you get from the operating system should be returned
  - Again, your programs must avoid memory leaks
- Most of the bugs in early stage are due to bad / misplaced calls of `delete`
  - Many memory related errors cause severe problems

Constants (1)

- As in C, the keyword `const` is used to define values that do not change
- In C++ the use of constants is wider
  - Constants should be used instead of the preprocessor `#define` directive

```
// avoid #define SIZE 100
const int SIZE = 100;
// use this instead
```

    - Why? Preprocessor directives can hide bugs which are nasty to find
  - Constants can be inserted into header files, name clashes will be detected by the compiler

Constants (2)

- Methods or method parameters can be declared as `const`
  - Does not add information to the outside, but rather force the compiler to check that no modifications are attempted
  - Useful when dealing with temporarily generated objects
  - Useful for efficient parameter passing (more soon)
- `constantparameters.cpp`

`const` Objects

- Again, as classes are types, it is possible to declare `const` objects or to declare a method which accepts a `const` object
  - The syntax is the same
- For a `const` object it is not possible to modify its public data members

Constant Methods

- A constant method is a method which does not alter the object. Thus
  - It cannot modify data members
  - It may call only other `const` methods
- Constant methods are the only methods which can be called for constant objects
- `constclass.cpp`

Multiple Inclusions

- Class declarations go to header files
- Header files will be included in all the `.cpp` files that need their declarations
- What if a header file is included twice?
  - A repeated class declaration is an error
    - But a repeated function declaration is not (as long as the declarations are the same)
- Should the programmer take care of not including the file twice?
  - Almost impossible in big projects

Conditional Compilation

- The preprocessor can be used to avoid multiple inclusions
- The `#ifdef`, `#ifndef`, `#else`, `#endif` directives allow to exclude some parts of the code according to specified conditions
- They are to be used with the `#define` that you already know

The Structure of a Header File

```
/* Student.h */
#ifndef _STUDENT_H
#define _STUDENT_H
class student {
  /* your class declaration */
};
#endif // this matches the initial #ifndef
```

How Does This Work?

- The first time the header is included the symbol `_STUDENT_H` is not defined
  - Then the class declaration is compiled and then the symbol is defined
- In all the subsequent inclusions the symbol is already defined and then the class declaration is skipped
- You must always protect (or guard) your header files with this mechanism