

TCC CSCI-2843 C++ Programming Language

Lecture 1 Notes

History of Object-Oriented Programming

First languages were low-level and platform-specific

- Machine Language – Numeric codes to directly manipulate hardware and operations.

- Assembly – Close to one-to-one match with statements and machine language operations.

Languages then became platform-independent and provided higher-level abstractions

- FORTRAN, COBOL, and later Pascal, C

- Functions and procedures for program decomposition

 - Argument passing 'by reference' and 'by value'

 - Functions that return values

- Richer primitive data types

 - Multiple integer types and sizes

 - Reals (floating point)

 - Character strings

 - Arrays

- User-defined data structures

 - 'record's in Pascal

 - 'struct's in C

- Program control flow

 - 'if' and 'if/else' for conditional execution

 - 'while', 'until', and 'for' loops for iteration

- Operators

 - 'Normal' infix math operators, '+', '-', '*', '/', '%'

 - Logical operators, '&&', '||'

 - Relational operators '<', '>', '==', '!='

 - Bit operators, '&', '|', '^'

- Separate Compilation

 - Programs may be composed of multiple source files

 - Possible to put common data, types, and functions in libraries for reuse

At this stage, known as 'Procedural' or 'Structured' programming, there are two major areas of study in Computer Programming:

- Data Structures – organization of information in memory

 - Arrays

 - Lists

 - Trees

 - User-defined data structures

- Algorithms – steps coded in a language to solve problems

Sorting

Searching

Object-oriented programming combines data structures and algorithms to make ‘smart structures’ or ‘objects’. The three themes in OO are

- Encapsulation
- Inheritance
- Polymorphism

We will be focusing on Encapsulation.

Two basic camps of OO programming:

- Smalltalk – Lives on in languages like Java and Ruby.
- Simula – Where C++ got its main OO concepts from.

Of course, most of C++’s basic syntax was inherited from ANSI C (for better or worse).

From C to C++

C++ inherited the basics from C, so you will need to know a good deal of how a C program is organized in order to cope with C++ and all that it adds and changes.

Scoping

C and C++ organize programs using source files (aka. compilation units) in which to organize types, data, and functions.

Two main issues – visibility and lifetime

This relates to the memory model

- Code segment – stores executable code, usually read-only
- Data segment – holds externs and statics
- Stack segment – hold locals for functions and procedures
- Freestore (‘heap’ in C vernacular) for dynamic – will discuss later

At the compilation unit (c and .cpp) level

extern (global)

Visible to all source files that declare a variable **extern**

Lifetime of program

static (at file level)

Visible only in compilation unit in which it is declared

Lifetime of program

local (**auto**)

Anywhere there is an open and close brace is a new scope

Created on stack in “activation records”

Visibility is from the declaration to the end of the scope in which it is defined (braces { })

Example is a local variable within a function’s scope

Lifetime is the same as its visibility.

Declaration vs. Definition

C and C++ have a “one definition rule”.

A variable or function may be **declared** as many times as the programmer wants, but may be **defined** only once.

A function may be declared:

```
void f(int);
```

which would indicate that **somewhere** there is a function named, f, that takes an **int** and returns nothing.

The function may be **defined** in the same or any other source file, such as

```
void f(int i) {
    ...
}
```

If a function or variable is **used** before it is **defined**, you will get a **compiler** error.

If a function or variable is **defined** more than once by the time the program is **linked**, then you will get a **linker** error.

If a function or variable is **declared** (no matter how many times), but never **defined** in any source files by the time the program is **linked**, you will get a **linker** error.

Variable Instantiation and Initialization

Extern Example:

```
int i = 0;

void f() {
    i = 1;
}
int main() {
    i = 2;
    f();
    return 0;
}
```

This **defines** an **extern int** named *i* and initializes it to 0. Note the initialization happens when the program is loaded into memory.

An **extern** may only be defined and initialized only once in one source file.

All other source files that want to reference this external variable must include the **declaration**:

```
extern int i;
```

Static Example:

Suppose the following was in a source file, xyz.cpp:

```
static int i = 0;

void f() {
    i = 1;
}
int main() {
    i = 2;
    f();
    return 0;
}
```

The variable, *i*, now is visible only to functions within xyz.cpp and **no other source file**.

What C++ Added to C

C++ not only supports another paradigm of programming, but also provides several facilities that are not found in C. These can be seen as being part of encapsulation, but are not “required” to qualify as object-oriented.

Function Overloading

In C, two functions within the same scope may not have the same name – even if their arguments are different. In C++, two functions with the same name may be visible in the same scope as long as the compiler can unambiguously determine the function to call. We will see example of this as we continue.

Operator Overloading

Most of the operators may be overloaded based on user-defined types so that objects may appear “built-in”. We will see some examples shortly.

Namespaces

C++ provides namespaces that allow us to name scopes and organize types, data, and functions.

Qualify access using scope resolution operator (::)

```
namespace A {
    int i = 10;

    void f() {}
}

namespace B {
    int i = 20;

    void g() {}
}

int main() {
    int k = A::i;
    int a = B::i;

    A::f();
    B::g();

    return 0;
}
```

How do we select the type, data, and functions we want? The **using** keyword.

We can use an entire namespace

```
using namespace A;
f();
```

Or can use specific type or function

```
using A::f;
f();
```

All standard C++ provided types and objects are in namespace `std`.

From Structured Programming to OO

Example of a structure

```
struct person {
    char last_name[80];
    char first_name[80];
};
```

used like

```
#include <stdio.h>
#include <string.h>

int main() {
    struct person p;

    strcpy(p.first_name, "Fred");
    strcpy(p.last_name, "Murtz");

    printf("%s %s\n", p.first_name, p.last_name);

    return 0;
}
```

What can go wrong?

- What happens if a nul-terminated character array (C string) is not initialized?
- What happens if it is not nul-terminated?
- What happens if you try to put too many characters into the array?
- What happens if you give a mismatched format specifier in a printf function?

Note that a string literal (a string in double-quotes, "") is still a nul-terminated array of characters set up by the compiler, even in C++.

The Deficiencies of “structured programming”

70’s and early 80’s were all about structured or procedural programming

The emphasis was on data structures and algorithms but these two topics only intersected in theory. The implementation was very much separate.

We still used procedures to manipulate “structures” that were otherwise unintelligent chunks of memory.

This is difficult to guarantee data integrity and consistency and made programming more difficult that it had to be.

Older techniques relied on the presence of “programmer discipline” to be correct, and that is clearly an oxymoron.

Transition to “OO”

Objects are a “smarter” combination of data structures and algorithms (procedures or functions)

We now can give those “chunks of memory” behavior and guarantees in addition to mere structure

We first move from structs to classes so we can use more intelligent types:

Strings

We’ll start with the std::string class

```
#include <string>

using std::string;
```

```

int main() {
    string s = "hello";

    string t = " world";

    string u = s + t;

    return 0;
}

```

Things to note

- Different header format – no ‘.h’
- Dynamically allocated and managed – no need to worry about running out of room or truncation.
- No need to worry about nul-termination
- Guaranteed to be a valid string
- Supports the use of operators (such as assignment, concatenation, etc...).

Console IO

Another example is the type-safe I/O streaming facilities of C++. Our first exposure will be to simply write on the console.

```

#include <iostream>

using std::cout;

int main() {
    cout << "Hello world\n";

    int i = 10;

    cout << "i = " << i << '\n';

    return 0;
}

```

So our example from before can now be written using classes as

```

#include <iostream>

using std::cout;
using std::string;

class person {
public:
    string first_name;
    string last_name;
};

int main() {
    person p;

    p.first_name = "Fred";
    p.last_name = "Murtz";

    cout << p.first_name << " " << p.last_name << '\n';

    return 0;
}

```

This is considerably safer than the C struct, mainly due to the use of `std::string` class. Note that we can use an assignment instead of `strcpy` and the stream output is completely type-safe using the operator `<<`.

Note that in C, we needed to say `'struct person p'` to instantiate a person, but in C++, a class is just like a built-in type, so we can just say: `'person p'`

Member Functions

We can add functions to classes so that our data structures now take on behavior:

```
class person {
public:
    string first_name;
    string last_name;

    string full_name() { return first_name + " " + last_name; }
};
int main() {
    person p;

    p.first_name = "Fred";
    p.last_name = "Murtz";

    cout << p.full_name() << '\n';

    return 0;
}
```

Note the use of string concatenation with the **operator+** instead of the old, unsafe `strcat`.

The `full_name` function is a member of the class, so can be selected for calling through the `'.'` operator **just like data members**. This gives us the ability to give the class itself behavior and not rely on separate functions to operate on just chunks of memory.

Constructors

With our person class, if we do not assign the members values, we get empty strings. This is an improvement over the C struct which could cause a crash with uninitialized C strings. A `std::string` somehow can initialize itself – how? Through constructors.

Constructors are special member functions that control initialization. They are functions that have the same name as the class and never have return types (not even **void**)

```
class person {
public:
    string first_name;
    string last_name;

    person(string fn, string ln) {
        first_name = fn;
        last_name = ln;
    }
    string full_name() { return first_name + " " + last_name; }
};
```

Now a person may be used in the manner

```
int main() {
    person p("Fred", "Murtz");

    cout << p.full_name() << '\n';

    return 0;
}
```

Now instead of “declaring a variable”, we are “instantiating an object” through a constructor call. This points out the main difference between plain data structures and objects. Objects come with behavior and guarantees, whereas data structures are mere chunks of memory that are manipulated by explicit function calls that may or may not put the structure in a valid state.

Note that this looks like a combination of a traditional declaration coupled with a function call. The function call is to the constructor that matches the arguments given. The use of the constructor **guarantees** initialization of the members. If there is only one argument to the constructor, we may use the ‘=’ to initialize the object:

```
string s("hello");
```

is semantically the same as

```
string s = "hello";
```

Default Constructor

A default constructor is a constructor that takes no arguments. These are called when a programmer does not provide any arguments on the object’s instantiation.

The `std::string` class has such a default constructor that simply makes the string an empty, but valid string, thus giving us the guarantee that it will not crash our program if we try to use it, unlike an uninitialized C string.

One ramification of this is that the statements:

```
string s;  
s = "hello";
```

is not the same as

```
string s = "hello";
```

The former is a **default** construction of the object, `s`, followed by an **assignment**. This means that in effect, the object is being initialized twice. **Try to avoid doing this**. The latter shows the proper way to ensure the object is initialized without doing double-duty.

Trivial Default Constructor

If no default constructor is provided and there are no other constructors provided, the compiler will attempt to generate a default constructor, known as the trivial default constructor. The behavior of a trivial default constructor is such that it will attempt to default construct all of the members of the class. If this is not possible or if any other constructor for the class is provided, then no trivial default constructor will be provided and no object of that type may be default constructed.

Initialization Lists

In our previous example, the `person` class had a constructor that used assignments in the body of the constructor to initialize the members. The question is – what was the state of these members **before** the assignment. The answer is – they were **default constructed** before the body of the constructor was executed.

The rule is that **all** members are constructed before the body of the constructor is executed.

Having member default constructed only to turn around and assign right on top of them seems wasteful (and is) and should be avoided. How? Initialization lists allow members to be constructed using constructor arguments **before** the constructor body is executed. An initialization list is a comma separated list of constructions placed after a colon (‘:’) after the constructor arguments and before the opening brace.

```
class person {  
public:  
    string first_name;  
    string last_name;
```

```

    person(string fn, string ln) :
        first_name(fn),
        last_name(ln)
    {}
    ...
};

```

The body of the constructor is now empty, since the members are initialized in the initialization list.

You can still perform whatever other operations you need in the body, such as validations, I/O, etc...

It is important to realize that members are constructed in the **order they are declared** not in the order they appear in an initialization list.

Multiple Constructors

A class may provide multiple constructors, due to the ability to overload functions. This means that you may provide a default constructor and several other constructors, all of which take in different numbers or type of arguments. As long as they are unambiguous, you can have as many as you want.

We've already seen an example of this with `std::string`, which provides a default constructor that initialized the string to be empty, but valid, as well as a constructor that takes a string literal (a string in quotes), and also provides a copy-constructor in addition to others that you can read up on as needed.

Accessor functions

public and private

So far, we've only used public members, but one aspect of encapsulation is that we may **hide** members of a class so that only member functions can access them and anyone who is using our class cannot.

```

class person {
private:
    string last_name;
    string first_name;

public:
    person(string fn, string ln) :
        last_name(ln),
        first_name(fn)
    {}
    ...
};

```

If the members are inaccessible to the public, then how are they to access them? The answer is to provide public member functions that control access to the members:

```

class person {
private:
    string last_name;
    string first_name;
public:
    person(string fn, string ln) :
        last_name(ln),
        first_name(fn)
    {}
    string get_last_name() { return last_name; }
    string get_first_name() { return first_name; }
    ...
};

```

We can use accessor functions to control how to set members as well:

```

class person {

```

```
...  
void set_last_name(string ln) { last_name = ln; }  
void set_first_name(string fn) { first_name = fn; }  
...  
};
```

If we give accessors to both set and get a member's value, then why make it private?

- We can perform validations to insure the integrity of values passed in.
- We can perform computations on the value passed on a set or on a value returned from a get.
- There may not be a one-to-one correspondence of accessor value and members (see full_name member on previous example).
- We might, at a later time, change the internal representation of the data without wanting to change the public interface.

In general, private members allow the class author to control access and integrity of the data.

Encapsulation Review

The first object-oriented topic we are discussing is encapsulation. This involves the following C++ features:

- Namespaces
- Classes not just 'structs'
- Member functions
- Constructors – controlling and guaranteeing initialization
- Data hiding – public and private
- Accessor functions – controlling access