C++ Programming for Engineers

Petar Jervic
Francisco Luro
Andrew Moss (course-resp)

May 2, 2012

Modularization

- Modular programming – last lecture
- Compilation units
- Preprocesing overview
- Compiling
- Linking
- Accesing code from other units
- Accesing data from other units
- Extern vs Static
- Debugging in VS

Modular Programming

Sofware design technique

- Break the program into components, called modules
- Each module is (ideally) independent from the rest
  - Vector, Stack, Linked list,...
  - Modules can be re-used to solve multiple problems
- Each module must be seen as a black box
  - Only interact with the module through its interface
  - Documentation is *very* important
  - Never rely on implementation details!
- Benefits
  - Maintainance
  - Reduced development time
  - Compilation time
  - Work on smaller independent problems

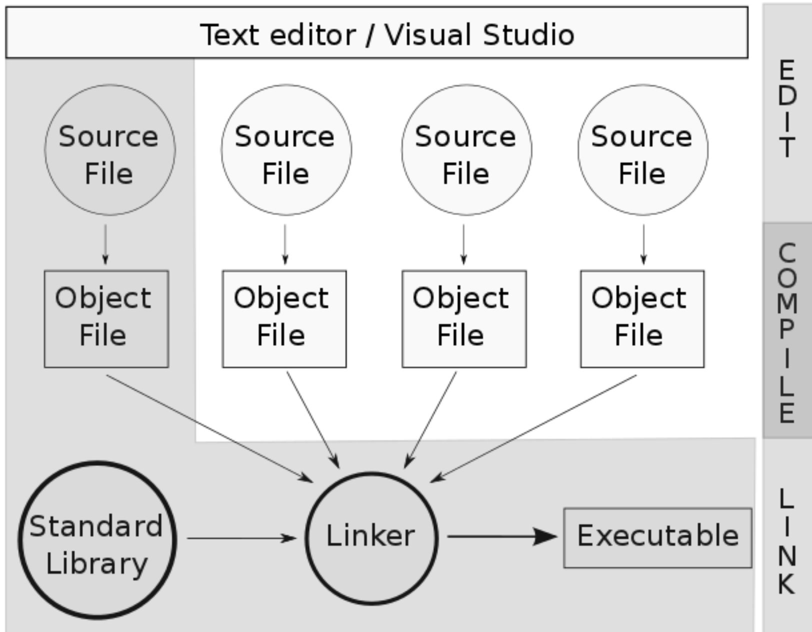
Compilation Units

How to achieve modularization ?

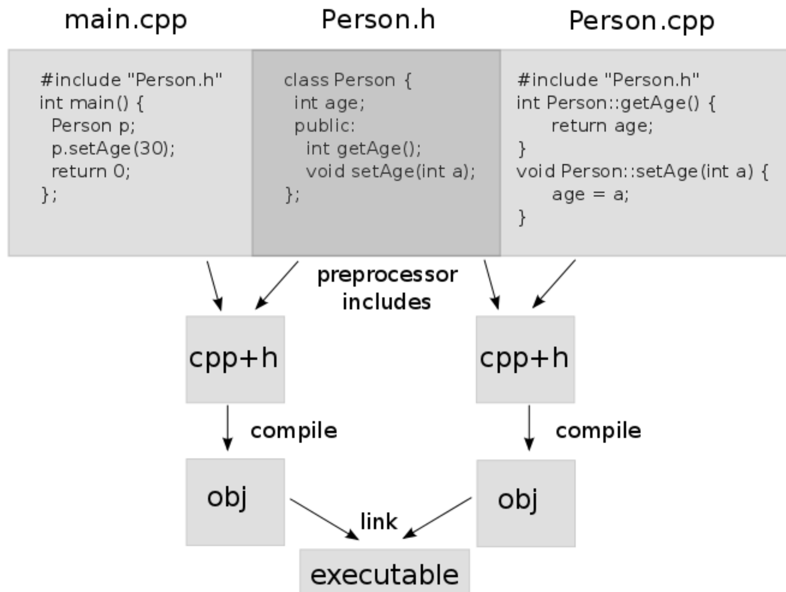
- Separate source code in different files (compilation units)
- Each file can be compiled separately
  - We only need to know about the interfaces of other modules used
- Combine the compiled units
  - Libraries
    - Static vs Dinamic libraries
  - Executable

Process overview

Edit -> (Compile, Compile, ...) -> Link -> Execute


Text editor / Visual Studio
Source File
Source File
Source File
Source File
Object File
Object File
Object File
Object File
Standard Library
Linker
Executable
EDIT
COMPILE
LINK

Preprocessing



Preprocesing (cont)

Define constants and symbols

Before preprocessing

```
#define PI 3.1415
#define MAX(p,q) ((p)>(q))?(p):(q)
int main() {
    double f = 2*PI;
    int a = 0,b = 1;
    int c = MAX(a,b);
}
```

After preprocessing – before compile!

```
int main() {
    double f = 2*3.1415;
    int a = 0,b = 1;
    int c = ((a)>(b))?(a):(b)
}
```

Compiling

- Transform code from source to binary
- Compile a single unit on the command line

```
General syntax: cl.exe /c File.cpp
cl.exe /c Person.cpp
cl.exe /c main.cpp
```

- Each will produce a *.obj* file (binary)
- When compiling, we only need access to the headers
- References to variables and functions defined elsewhere will be left unresolved until the linking stage.
  - Try removing setAge(int) from Person.cpp and both files will still compile fine.

Linking

- The different compiled units are put together
  - Can produce a library or an executable.
- Often linking happens implicitly

Compile first, then link

```
cl.exe /c Person.cpp
cl.exe /c main.cpp
cl.exe Person.obj main.obj
```

Compile and link in one go

```
cl.exe Person.cpp main.cpp
```

- If any reference (variable or function) is missing when linking, then is a Linking error.
  - Try commenting Person::setAge() in Person.cpp and then try to link main.o and Person.o.

Declaration/Definition

Until now, we have used the term *declaration* when talking about variables needed in our program.

- In C++, a variable declaration is only the act of describing the variable.
- The variable definition will actually instruct the compiler to allocate memory for this variable.
- Usually, these two events happen at the same time.
  - Local/Global variables are declared and defined in the same sentence.
- We can have many declarations but one definition!

Define/Declare

```
int globalVar;
int functionA() {
    int a = 0;
    return a;
}
```

Declaration/Definition and Extern

The extern keyword will specify that the definition is in another compilation unit. (external linkage)

- Therefore, only declaring the variable or function.
- Knowing only about the declaration is enough to compile, but not to link.
- Function declarations (only prototypes) are extern by default.

main.cpp

```
#include <iostream>
extern int globalCounter;
int count();

int main() {
  for (int i=0;i<5;i++)
    int a = count();

   std::cout << "Called ";
   std::cout << globalCounter;
   std::cout << " times!";
   return 0;
}
```

count.cpp

```
int globalCounter;
int count() {
  return globalCounter++;
}
```

out

```
Called 5 times!
```

Static keyword

Depending on the context it has different meanings

- Variable defined static at file scope, is like a global variable, but can only be accessed anywhere within the file (internal linkage)
- Other compilation units cannot use extern to access it.

main.cpp

```
#include <iostream>
int count();

int main() {
 int a=0;
 for (int i=0;i<5;i++)
   a = count();

  std::cout << "Called ";
  std::cout << a;
  std::cout << " times!";
  return 0;
}
```

count.cpp

```
static int globalCounter;
int count() {
 return globalCounter++;
}
```

out

```
Called 5 times!
```

Static keyword (2)

- Variable defined static inside a function, is like a global variable, but can only be accessed from within the function.
- The variable memory is created and initialized only once!

main.cpp

```
#include <iostream>
int count();
int main() {
  for (int i=0;i<5;i++)
    int a = count();

  std::cout << "Called ";
  std::cout << count();
  std::cout << " times!";
  return 0;
}
```

count.cpp

```
int count() {
  // static function scope
  static int memory;
  return memory++;
}
```

out

```
Called 5 times!
```

Static keyword (3)

- A data member of a class defined as static implies that the value will be shared among all the objects (instances) of that class.
  - Usually called *class variable*

main.cpp

```
#include <iostream>
#include "Person.h"
using namespace std;
int main() {
   for (int i=0;i<10;i++) {
      Person p = Person();
   }

    cout << "Population ";
    cout << Person::population;
    return 0;
};
```

Person.h

```
class Person {
  int age;
  public:
   Person();
   // declaration
   static int population;
   int getAge();
   void setAge(int a);
};
```

Person.cpp

```
#include "Person.h"

// definition
int Person::population=0;
Person::Person() {
  population++;
  age = 0;
}
int Person::getAge() {
     return age;
}
void Person::setAge(int a) {
     age = a;
}
```

output

```
Population: 10
```

Debuggin in VS

Demo...