Chapter 2.
Making And Using Objects

C++ Object Oriented Programming

Pei-yih Ting

93/02 NTOU CS

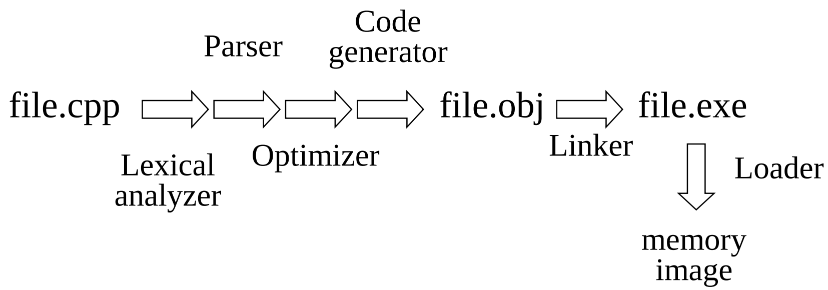
Content

- Program compilation and linking
- Type checking system
- Separate compilation
- Declaration vs. Definition
- #include preprocessor directive
- Using libraries
- Namespace
- Using iostream and iomanip
- String class
- File I/O with fstream
- Container: vector

Interpreter vs. Compiler

- Interpreter
  - translate each source code executed into **machine activities**
  - retranslate each line executed, skip those not encountered
  - Pros: rapid development(modification/debugging/interaction), easy to automate
  - Cons: slow for computation intensive jobs
  - BASIC, PERL
- Compiler
  - Translate source codes into **machine instructions**
  - Translate the whole program only once
  - Slow development cycle as a tradeoff to fast execution
- Combination
  - Python, JAVA: have intermediate language, platform indep.

C++ Compilation/Linking/Execution



C++ Compilation/Linking/Execution

- Preprocessor: process #preprocessor directives

ex.
```
#define PI 3.14159
#ifndef _WIN32
#define SQUARE(x) ((x)*(x))
#pragma warning (disable:4768)
#include <iostream>
```

> - Save typing
> - Increase readability
> - Parser does not see them
> - Debugger does not see them
> - **Introduce subtle bugs**

- Lexical analyzer: breaks the source code into small units
- Syntactical parser: organizes into a parse tree according to the grammar
- Optimizer: produces smaller or faster code
- Code generator: generate object(target) machine code (*.obj) according to the parse tree
- Linker: resolves variables or routines references outside each independent object module, produce a relocatable execution code
- Loader: loads the executable from file into the memory

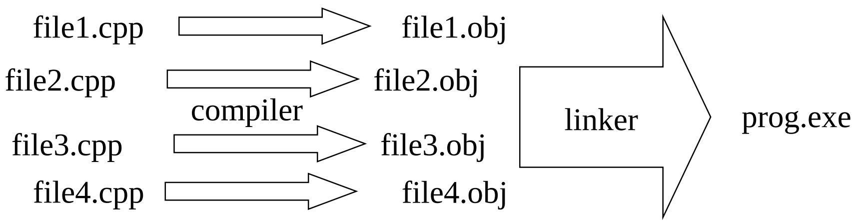
Type Checking

- Static Type Checking: ensure that each grammatical objects have the correct type by the parser and enforce type conversion, ex. proper data types of function arguments
  - Weak type checking: Perl
  - Strong type checking: C/C++, PASCAL

  In C/C++ you can disable type checking by **coercion**.
- Dynamic type Checking: perform type checking at runtime, more powerful but adds overhead to program execution
  - Java
  - C++ RTTI (run time type information)
  - MFC Runtime class

Separate Compilation

- Each module is put in a separate file, which is a small, manageable, independently tested piece
- Editing of one module does not involve other module: avoid editing mistakes
- Compilation of one module only checks the grammar syntax and does not involve other modules: large project can proceed independently, better encapsulation
- C/C++ compiler provides the functionality of separate compilation



Declaration vs. Definition

- Function: the atomic unit of code in C/C++
  - must be put in a single file
  - has a name, some parameters and a return value.
- Each file contains **definitions** of:
  - Data: the type of that identifier and allocates memory
  - Function: the name, the parameters, the return values and the codes
- To access a function (to call a function) or a variable defined in another file
  - you must **declare** the function or the variable first in that file, so the compiler knows what the identifier represents, can perform the type checking, and can deal with it (conversion)
  - Ex. declarations:
    ```
    int func1(int, float, char *);
    extern int x;
    ```

> **You can declare an identifier many times but you can only define it once.**

Declaration vs. Definition

- Declare.cpp // Declaration & definition examples page84

```
extern int i; // Declaration without definition
extern float f(float); // Function declaration

float b;  // Declaration & definition
float f(float a) {  // Definition
  return a + 1.0;
}

int i; // Definition
int h(int x) { // Declaration & definition
  return x + 1;
}

int main() {
  b = 1.0;
  i = 2;
  f(b);
  h(i);
}
```


#include preprocessor directive

- Insert the contents of the specified file in the place of your #include statement for the following compilation
  - #include <header.h>
    search the current directory and the specified directories in the 'include search path'
  - #include "header.h"
    search the current directory for the header file
- To see what is included:
  - Compiler options: using VC as example: cl /E or cl /P
  - View the included file in the system directory: ex.
    C:\Program Files\Microsoft\VC98\include
- C++ convention
  - #include <iostream.h>   old versions
  - #include <iostream>
    using namespace std;   new template versions

#include preprocessor directive

- #include <stdio.h> ⇒ #include <cstdio>
  #include <stdlib.h> ⇒ #include <cstdlib>
- Where are those functions defined in the include files?
  - Their object codes are in the library: ex. libc.lib
    use lib tool to view the contents of *.lib files
  - All C/C++ compilers direct the linker to search the standard library automatically

Using Libraries

- Libraries contains one or many object modules
- To use a library:
  - Include the header file of an object file in a library
  - Use the functions and variables in the object module of a library
  - Link the library into the executable program
- How the linker searches a library
  - It searches the 'unresolved references' one library by on library
  - If there are many repeated definitions of a certain function, the first library containing the definition is used.
    - The order of libraries supplied to the linker is crucial
    - You can preempt the use of a library function by inserting a version of yours
  - Once an unresolved reference is located in an object module of a library, the object module is extracted out of that library and combined to the executable.

Using Libraries (cont'd)

- How do the following two usages differ?
  - First scenario:

    ```
    cl /Femain.exe main.c tool1.c tool2.c tool3.c tool4.c tool5.c
    ```

  - Second scenario:

    ```
    cl /c /Fotool1.obj tool1.c
    cl /c /Fotool2.obj tool2.c
    cl /c /Fotool3.obj tool3.c
    cl /c /Fotool4.obj tool4.c
    cl /c /Fotool5.obj tool5.c
    lib /out:tool.lib tool1.obj tool2.obj tool3.obj tool4.obj tool5.obj
    cl /Femain.exe main.c tool.lib
    ```


Namespace: avoid name collisions

- Namespace definition

```
namespace Foo {
   int var;
   int foofun(int x) {
      return x;
   }
}
```

- using directive

```
int main() {
   using namespace Foo;
   int testvar;
   return foofun(testvar+Test::testvar+
                  var+Foo::var);
}
```

- using declaration

```
using Foo::foofun;
foofun(10);
```

```
namespace Test {
   int testvar;
   int testfun(int x) {
      return x;
   }
}
….
namespace Test {
   int anotherVar;
   int anotherFun();
}
…
int Test::anotherFun() {
   return 1;
}
```


Using iostream and iomanip

- C++ standard input/output modules

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

void main() {
   int value;
   cout << “Howdy!" << endl;
   cout << “Please input an integer “;
   cin >> value;
   std::cout << setw(4) << value << endl;
}
```

Note: the difference with scanf("%d", &value);

Miscellaneous

- Calling other programs: system()

```
#include <cstdlib>
using namespace std;
int main() {
   system("hello.exe"); // executes hello.exe and returns to the program
}
```

- __FILE__,__LINE__, __DATE__, __TIME__:

  compiling file, line #, date, and time as string into the program
  - cout << __FILE__ << " " << __DATE__ << __TIME__ << endl;
  - assert() macro's output
- ctime(), time.h: getting the system time

Standard C++ ***string*** class

- Properties:**(more abstract and convenient character array implementation)**
  - Dynamic memory management
  - Character array copying
  - Concatenation
- Header file: #include <string>
- Namespace: using namespace std;
- Ex.

```
string s1, s2;                      // empty strings
string s3 = "Hello, World.";        // initialized
string s4("I am");                  // initialized
char dest[100];
s2 = "today";                       // string copy
s1 = s3 + "  " + s4;                // concatenation of strings
s1 += " 8 ";                        // appending to a sting
cout << s1 + s2 + "!" << endl;      // extended ostream
strcpy(dest, s1.c_str());           // convert to char array
cin >> s1;
cout << s1.length();                // or s1.size();
if (s3 + s4 == "Hello, World.I am") …
```


Reading/writing files with *fstream*

- Avoid the complexity of C's file operations
- Header file: #include <fstream>
- Namespace: using namespace std;
- Ex.

explicit open not required

```
ifstream in("Scopy.cpp"); // open for reading
ofstream out("Scopy2.cpp");          // open for writing
string s;
while (getline(in, s)) // delimiter '\n' deleted automatically
   out << s << "\n";
```

  - getline() returns false upon reaching the end of the input and the returned string become empty

fstream cont'd

- Ex.

```
ifstream in("FillString.cpp");          // open for reading
string s, line;
while (getline(in, line)) s += line + "\n";
cout << s;
```

  - Exercises:
    - Concatenate 2 files with filename headers
    - Add line # to each line in a file
    - Search a specific string and print out with line #

line.find("hello");

- The delimiter of getline() function can be changed
  - Ex. getline(in, s, ';');

Standard C++ *vector* class

- Container object:
  - hold any kind of object
  - dynamically adjust its memory size
- Vector class is a template: can be efficiently applied to different types ex. vector of strings, vector of integers…
- Declaration: vector<string> obj;
- Header file: #include <vector>
- Namespace: using namespace std;
- Interfaces:
  - push_back()
  - insert()
  - size()
  - indexing: a[4] like an array

vector

```
// cl -GX Fillvector.cpp
// copy an entire file into a vector of strings
#include <fstream>
#include <iostream>
#include <vector>
#include <string>
using namespace std;
int main() {
    ifstream inf(“Fillvector.cpp");
    string line;
    vector<string> lines;
    while (getline(inf, line))
        lines.push_back(line);       // add the line to the end
    cout << lines.size() << endl;
    for (int i=0; i<lines.size(); i++)
        cout << i << ":" << lines[i] << endl;
    return 0;
}
```

The object *line* is copied into *lines* and the main program can destroy the *line* object afterwards.

The container object *lines* is going to handle all memories at the destruction.

Summary

- OOP:
  - Programming based on objects, to be more exact, based on the interface of objects
  - Suitable encapsulation hides the detailed implementations of an object and exhibits only the interface of a certain simple abstract model
  - Usages of an object:
    - Include a header file
    - Create the objects
    - Send messages to them
    - Get their responses