Comp151

Definitions & Declarations

Example: Definition

```
/* reverse_print.cpp */

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

int global_var = 23;                    // global variable definition

void reverse_print(const char* s)   // function definition
{
    for (int j = strlen(s) - 1; j >= 0; --j)
        cout << s[j];
    cout << endl;
}
```

Example: Declaration

```
/* use_reverse_print.cpp */

#include <iostream>
using namespace std;

extern int global_var;                           // external variable declaration
extern void reverse_print(const char* s); // external function declaration

int main(int argc, const char* argv[])
{
    float local_var;                             // local variable definition
    local_var = 987.654;

    cout << "global var = " << global_var << endl;
    cout << "local var = " << local_var << endl;
    cout << "input string backwards = ";
    reverse_print(argv[1]);
}
```

Definition

- A **definition** introduces a variable's or a function's name and type.
- A variable definition reserves a number of bytes of memory for the variable.
- A function definition generates code for the function.
- In both cases, definitions cause the compiler to allocate memory to store the variable or function code.
- An object must be defined exactly once in a program.*

*Except inline function definitions (which we'll discuss in a moment).

Declaration

- The **declaration** of a variable (or function) announces that the variable (or function) exists and is defined somewhere else (in the same file, or in a different file). The connection is made when the object files are linked.
- A variable declaration consists of the variable's name and its type preceded by the keyword `extern`.
- A function declaration consists of the function prototype (without the function body) preceded by the keyword `extern`.
- A (forward) class declaration consists of the class name (without the class body) preceded by the keyword `class`.
- A declaration does <u>not</u> generate code, and does <u>not</u> reserve memory.
- There can be <u>any number of declarations</u> for the same object name in a program (as long as they're consistent with each other).
- If a declaration is used in a file different from that with the definition of the object, the <u>linker</u> will insert the real memory address of the object instead of the symbolic name.
- In C++, a variable <u>must</u> be defined or declared to the program before it is used.

A word on class definitions

- A **class definition** defines a type.
- Note that merely defining a type or class does not generate code. (The code for any member function of the class is not generated until the compiler sees an individual member function definition.)
- In this sense, class definitions are unlike variable and function definitions, which cause the compiler to allocate memory to store the variable or function code.
- But class definitions are still like variable and function definitions in the sense that a class must be defined exactly once in a program.

Advantages of Header Files

- In general, a header file provides a centralized location for:
  - external object declarations
  - function declarations & member function declarations
  - type definitions
  - class definitions (but <u>not</u> member function definitions, except inline)
  - inline function definitions & inline member function definitions

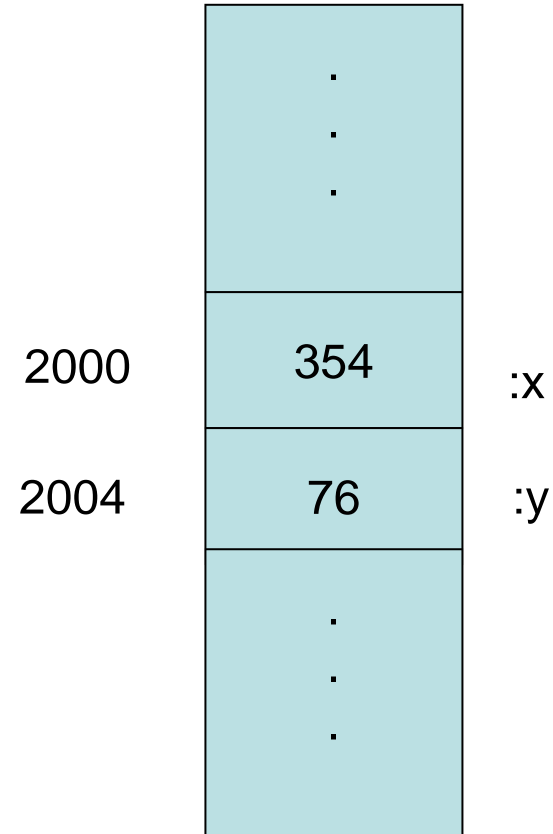
- The advantages are:
  - 1. By including the header files, all files of the same piece of software are guaranteed to contain the same declaration for a global object or function.
  - 2. Should a declaration require updating, only one change to the header file will need to be made.

A question…

```
/* C++ code fragment – uses Date class from separate compilation lecture */

    Date d1(2, 17, 2006);
    Date d2;
    d2 = d1;
    d1.set(2, 20, 2006);
    d2.print();                              // what gets printed here?


/* Java code fragment – looks nearly identical to above C++ code fragment */

    Date d1 = new Date(2, 17, 2006);
    Date d2;
    d2 = d1;
    d1.set(2, 20, 2006);
    d2.print();                              // what gets printed here?
```

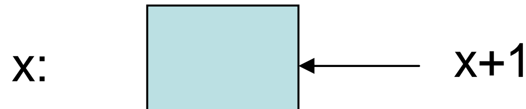
Variables

- A **variable** is a symbolic name assigned to some memory storage.
- The size of this storage depends on the type of the variable, compiler, and platform.
  - e.g., on x86 under Windows, `char` is 1 byte long and `int` is 4 byte long.
- The difference between a <u>variable</u> and a <u>literal constant</u> is that a variable is addressable.



Key distinction: lvalue vs. rvalue

[interpretation of " x = x + 1 "]



- A variable has dual roles, depending on where it appears in the program, it can represent
  - **lvalue**: the location of the memory storage
  - **rvalue**: the value in the storage

- They are so called because a variable represents an lvalue (or rvalue) if it is written to the left (or right) of an assignment statement. Thus, the following are invalid statements in C++:

```
4 = 1;
grade + 10 = new - grade;
```

Not all languages distinguish rvalues from lvalues the way that C++ does!

```
/* C++ code fragment – uses Date class from separate compilation lecture */

    Date d1(2, 17, 2006);
    Date d2;
    d2 = d1;
    d1.set(2, 20, 2006);
    d2.print();                                // prints 2006.02.17


/* Java code fragment – looks nearly identical to above C++ code fragment */

    Date d1 = new Date(2, 17, 2006);
    Date d2;
    d2 = d1;
    d1.set(2, 20, 2006);
    d2.print();                                // prints 2006.02.20
```