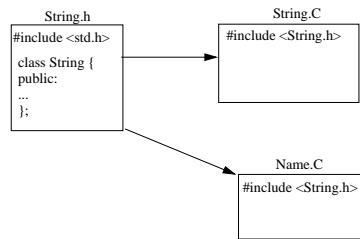


C++ Source File Organization

Organize your program as a collection of “header” files (.h) and implementation files (.C or .cpp or .cc or .cxx).

The .h files contain:

- #include of other .h files
- preprocessor macros (#define)
- constant declarations (consts)
- type definitions (typedefs)
- enumerated types (enums)
- class declaration
- inline function/method definitions
- procedure declarations (prototypes)



The .C files contain:

- #include of .h files
- extern/static data and object definitions
- extern/static functions and procedures
- class “method” implementations

Good programming practice is to pair the name of a .h file with a .C file, for example:

String.h defines common string constants, types, the **String** class declaration, and inline methods
String.C defines the implementation of the **String** class

The user of a class includes the corresponding .h file at compile-time, and links to the already compiled .C file

C++ Header File Organization

```

/* String.h -- defines a common String abstraction in C++
 *
 * Copyright (c) 1996 by Me Too Inc. All Right Reserved
 *
 * Purpose: provide an efficient object-oriented String object.
 *
 * Author: me
 * Date: today
 */

// prevent repeated inclusion
#ifndef _String_h
#define _String_h 1

// include some needed definitions
#include <string.h> // include definitions of "C" strings from the std C library

// define NULL if not already defined
#ifndef NULL
#define NULL 0
#endif

// useful constants
const MAX_STRING_SIZE = 8192;

// enumerated type
enum StringTypes { IA5String, UniversalString };

// useful type aliases
typedef unsigned char octet;

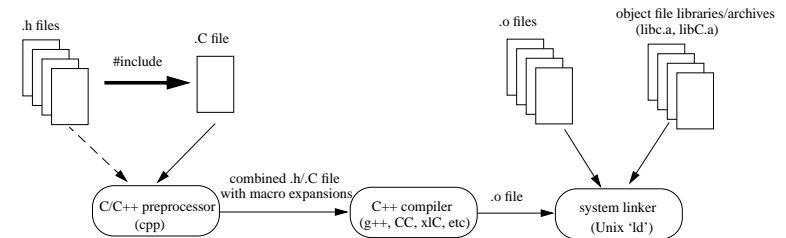
// class declaration and inline method definitions
...

```

C++ Compilation and Linking

3 steps to creating an executable program

1. Preprocessor - strips comments, expands all “#” directives (#include, #define, #ifdef-#endif)
2. Compiler - checks syntax, generates debugging info, performs optimization, generates machine code (.o file)
3. Linker - combines multiple .o files and libraries to create an executable image. Linker is a system utility that is language independent since it works on object files, not source files. But you still need common procedure calling and linkage conventions to work. In practice, C++ code can call C code easily, but it is much harder the other way around because of non-standard “name mangling” of symbols in C++.



String.h continued

```

class String {
public:
    // public clause means all data & methods are "visible" to use of class

    // constructors and destructor
    String(); // "default" constructor
    String(const String& s); // "copy" constructor
    String(const char* s); // conversion constructor -- converts a char* to a String

    ~String(); // destructor

    // other methods
    int length() const;
    ...

private: // private clause hides data & methods from being used
    // this is where the implementation specific stuff goes

    char* _string; // a character pointer to a chunk of heap allocated memory
    int _size; // the size of the chunk

    void copy(const char*); // private internal function to copy a sequence of bytes
    ...
};

// inline String class method definitions
inline int String::length() const
{
    if (_string != NULL)
        return strlen(_string); // C library function to compute the length
    else
        return 0;
}

#endif // _String_h

```

String.C Implementation File

```

/* String.C -- implementation common String class
 *
 * Copyright (c) 1996 by Me Too Inc. All Right Reserved
 *
 * Purpose: provide an efficient object-oriented String object implementation.
 *
 * Author: me
 * Date: today
 */

// a static data item is not visible outside the .C file
static const char* version = "String.C version 1";

// include the String class definitions
#include "String.h"

// provide an implementation of the non-inline String class methods

// constructors
String::String()
{
    _string = NULL;
    _size = 0;
}

String::String(const String& s)
{
    ...
}

String::String(const char* s)
{
    ...
}

```

8/5/00

5 of 12

The Main program

Before you can use your classes and objects, you need to write a ‘main’ program, usually main.C. The system linker/loader uses the symbol ‘main’ as the entry point to your program.

```

// main.C
#include "String.h"

// main can be written with no arguments, or two arguments, or three arguments
int
main() // or main(int argc, char* argv[]) or main(int argc, char* argv[], char* env[])
{
    String s = "Hello world";
    cout << s << '\n';
    exit(0);
}

```

You can then compile and link an executable image. If you use -o, you can name the image, otherwise is defaults to ‘a.out’:

```

% g++ -g -I. main.C String.C -o myprog
% myprog
Hello, world
%

```

Try using -v to see all the steps: preprocessor, compiler, linker. Note that the linker will also link in the lib.c.a library by default. The compiler command (g++, x1C, CC, etc) is just a driver program that runs all the steps by calling other programs.

```

% g++ -v -g -I. main.C String.C -o myprog

```

8/5/00

7 of 12

Compiling under Unix

The compile will run the preprocessor for you automatically.

GNU C++ compiler (g++). The -g flag turns on debugging information for later use by a debugger (gdb). I recommend that you always use the -g flag. The -I flag tells the preprocessor where to look for .h files. The compiler will generate a corresponding .o file (e.g., String.o).

```

% g++ -c -g -I. String.C

```

If you want to see all the steps, use the -v flag for verbose output

```

% g++ -v -c -g -I. String.C

```

Other Unix compilers accept very similar options:

```

aix% x1C -c -g -I. String.C

```

```

sun% CC -c -g -I. String.C

```

```

hpux% CC -c -g -I. String.C

```

There are lots of different compilation options. You can type ‘man g++’ to read them. If you use the -E option, only the preprocessor will run and produce a single file with all the comments stripped and the .h files included. The -S flag will cause the assembly code in a .s file instead of machine code in a .o file, this is sometimes helpful when trying to understand how the compiler ‘mangles’ C++ class and method names into symbol names the linker can use to link together .o files and .a libraries.

8/5/00

6 of 12

Basic C++ (See Appendix A in course text)

C++ is a *statically typed* language, which means that the compiler must be able to infer all type information at compile-time. The language is also *strongly typed*, which means that only well-defined type definitions and conversions are allowed. In general, a statically typed language is more efficient, because type inferencing decisions are made at the early at compile-time instead of late at run-time. A strongly typed language is ‘safe’ because it forces you to specify clearly what types are allowed and how they can be used.

The distinction between what is done at compile-time (statically) versus run-time (dynamically) is fundamental in programming. If you don’t get it straight in your head at the start, you will always have difficulty understanding a language like C++.

C++, like C, defines the same set of built-in types and control structures (conditionals, loops, switch statement). Where it differs most, is the ability to define new types using the ‘class’ construct and the inheritance mechanism for composing classes, in a well-define way, to form new types.

So, the heart and soul of C++ are it’s type definition and composition mechanisms: i.e., classes, template class and template functions, and class inheritance. A key feature of the C++ type system is that it allows *polymorphic types*, and most of the complexity of the language derives from this feature.

All the rest of C++ is pretty much the same as C.

C++ also implement *exception handling*, which is a control flow mechanism for transferring control *and information* from the site of an error to a place in the program that can either handle the error, or gracefully terminate the running program.

8/5/00

8 of 12

Built-in Types

C++ is not a pure OO language because builtin types are not really objects. For efficiency, they are handled specially. That is, they map closely to machine level data elements (byte, half-word, word, double word, single and double precision floating point).

Here they are:

```
char
short
int
long
long long
float
double
long double
```

By default, these are all signed values (MSB is the sign bit). In practice, we often need unsigned integer values:

```
unsigned char
unsigned short
unsigned int
unsigned long
```

It is common practice to use a typedef to define a type alias for these:

```
typedef unsigned char uchar;
typedef unsigned char octet;
typedef unsigned short ushort;
typedef unsigned int uint;
typedef unsigned long ulong;
```

NOTE: some compilers treat char as unsigned by default, so it is best to write ‘signed char’ explicitly if that is the type of value you really want.

Using Built-in Types

```
char x = 'c';
char newline = '\n';
char mode = 0755;
uchar y = 0xff; // 0x or 0X denotes a constant hexadecimal value

short a = 128;
short a = -1 * 128;
ushort b = a; // what happens here?

long = 0L; // L denotes a long constant

float epsilon = .0001;

double pi = 3.1415926535897932;
```

Question: how would you represent a 1024 bit integer value, such as a public key used for encryption?

Type conversion between compatible types is done implicitly by the compiler, and you have to know about it.

```
char c = 'a';
int b = c; // character value is "promoted" to an integer value then assigned

int x = 10;
long z = x * 10000000; // value of x is temporarily promoted to a long
```

Enumerated types are integral types, starting at 0 by default:

```
enum Colors { Red, Green, Blue };
enum Boolean { False = 0x00, True = 0xff };
enum Plugs { Error = -1, Open = 1, Closed = 2 };

int x = False;
```

Pointer Types

Pointers are strongly typed variables for referencing the address of a datum. All the builtin types have corresponding pointer types:

```
char*
short*
int*
long*
float*
double*
void* // this is a special pointer type, which an opaque type that can point at anything
```

Examples

```
char buffer[1024]; // array of 1024 bytes, starting at the address of buffer[0]
```

```
char *q = buffer + 1024; // point at first byte past the end of buffer
```

Arrays and pointers are the same. There is no bounds checking in C/C++. If you run off the end of an array, that's your problem and it is usually a nasty bug to find.

Pointer arithmetic in C/C++ is used heavily. For example, you can do either of the following to loop through a character array:

```
for (int i = 0; i < 1024; i++)
    buffer[i] = 0x00;
```

Or

```
for (char* p = buffer; p != q; p++)
    *p = 0x00;
```

p++ increments the pointer variable p by 1 (sizeof char). *p dereferences the pointer variable, giving you the location of the object whose address the pointer variable holds.

Pointer Types

The ‘++’ operator is the increment operator. it comes in two form: pre-increment and post-increment. It is used to increment the value of any built-in type. C++ allows you to overload this operator as part of a class definition. Pointer arithmetic on built-in types is an “overloaded” operation. The amount of the increment is the size of the type to which the pointer points. There is also a pre-decrement and post-decrement operator ‘--’

```
int array[10]; // 10*4 bytes worth of memory space
```

```
int count = 0;
for (register int *p = array; p < &array[10]; p++)
    *p = ++count;
```

The *register* keyword is a hint to the compiler to try to allocate the pointer in a register for efficiency, although most compilers are smart enough to do this without you asking to.

Questions: What is the increment of p++? What is the increment of ++count?

C/C++ define a special compiler function called *sizeof*, that will always give you the size, in bytes, of any type:

```
sizeof(char)
sizeof(short)
sizeof(long)
sizeof(float)
sizeof(double)
```

What about pointer types?

```
sizeof(char*)
sizeof(short*)
sizeof(int*)
sizeof(void*)
```