

Basic C++

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 implements **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.

C++ Source File Organization

When implementing classes in C++, each class should be defined in a separate header file. When writing large programs, Organize your program as a collection of “header” files (.h) and implementation files (.C or .cpp or .cc or .cxx). The header files containing the class definitions are included in the implementation files, which contain the implementation of non-inline class methods.

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
- non-inline class “method” implementations

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

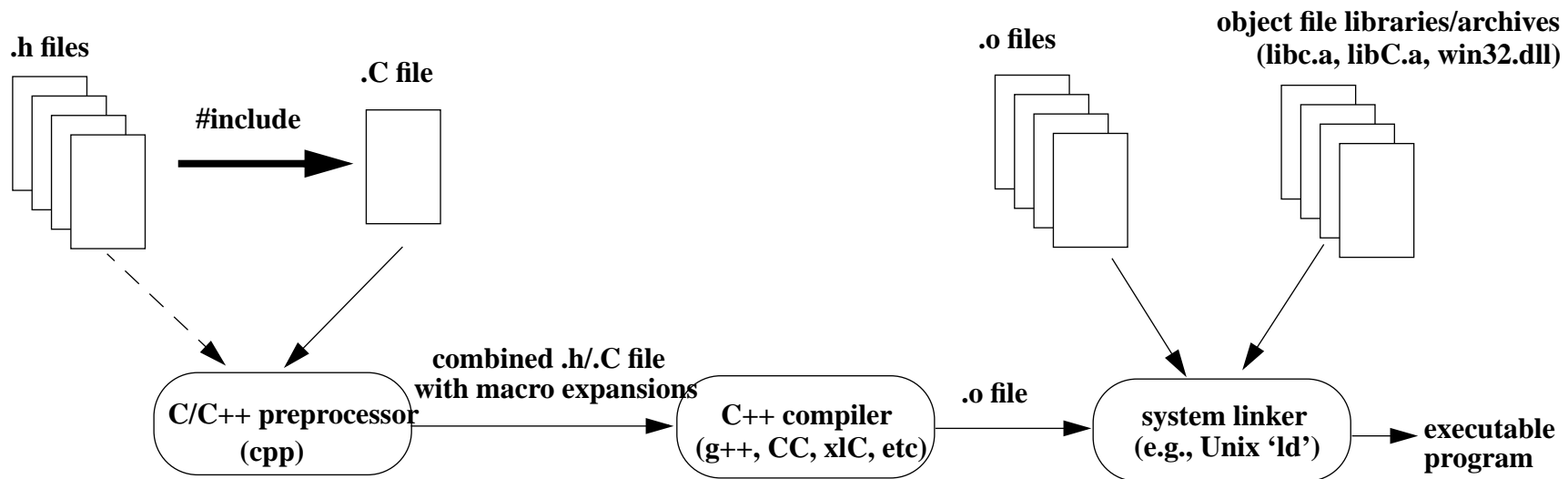
String.h defines common string constants, types, the String class declaration, and inline methodsString.C defines the implementation of the String class The user of a class includes the corresponding .h file at compile-time, and links to the already compiled .C file

Notes on 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 (a .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++.

It is often advisable to combine several .o files into an library archive file (.a file on Unix, .lib or .dll on Windows).



C++ Built-in Types

C++ is not a “pure” object-oriented 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 are the builtin types. (the type ‘bool’ is not defined as builtin by all C++ compilers).

```
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 (8-bit characters) by default, so it is best to write ‘signed char’ explicitly if that is the type of 7-bit character value you really want.

Objects are Defined by a Class

A **class** defines a new *type*. All objects are defined by a class, and so we say “an object is of type *X*”, where *X* is the name of a class. Once defined, that type can be used to declare instances (or objects) of that type, pointers to objects, references to objects, and functions/procedures that takes objects as arguments or return objects as results.

A class defines the data (called **instance variables**) and the operations (called **methods** or **member functions**). An object is thus an instance of class. A class is a compile-time construct. An object is an run-time entity. The C++ compiler does “type checking” to make sure that your program correctly uses an object so that their is some level of confidence that your program is correct at run-time.

The String class defines a commonly used type of object, namely, a string object. A string object contains instance variables used to implement an array of characters and a length, along with operations that users of a string object use to construct, manipulate, and destruct a String object.

```
class String {
private:

    // "private" means instance variables are hidden from users of a String object

    char* _string;    // pointer to an array of bytes for the string
    int   _length;    // the length of the array of bytes, not including '\0'

public:

    // instance variables & methods visible to users (and subclasses)
    ...
};
```

Object Instantiation

Every object has a machine **address**, no matter what area of memory it is allocated from. A **global** or **static** object has an address from the static address space. A **stack** allocated object has the address of a stack location. A **heap** allocated object has an address taken from the dynamically managed heap area. No matter where the object “lives”, the memory address uniquely identifies the object. So, a special pointer, called the **this** pointer is how each object is uniquely identified since the contents of the this pointer is the machine address of an object in memory.

Note that the **this** pointer is a constant value. That is to say, you cannot legitimately change the address of an object by assigning a new value to the **this** pointer for an object.

The implicit **this** variable of every object contains the address of the memory location where the object was allocated. The this variable is implicitly passed as the first argument to every non-static, non-friend member function defined for an object by it's class.

```
void some_fun()
{
    String hello = "hello, world"; // hello is the name associated with a stack location

    String *str = new String(hello); // str is the name of a stack location, which
                                    // holds the address of a String object in the heap

    int x = hello.length(); // calls String::length(this=&hello)
    int y = str->length(); // calls String::length(this=str)
    ...
}
```

Q: The this pointer itself cannot be changed, but the object to which it points can be changed by a non-const member function. So, What is the type of the “this” pointer for a non-const member function for a given class X ?

```
const X* this;
X* const this;
const X* const this;
```

Object Instantiation

So, object can be instantiated statically (in static data area), automatically (on the stack), or dynamically (in the heap)

```
// constructors for external/static objects are executed at program startup and
// destructed on exit from the program

const String version("1.0"); // global, but immutable

static const String rcsid = "$Header$"; // local to this compilation unit

some_fun()
{
    // local object constructors are implicitly called on entry to a scope

    String s; // invokes String::String() default constructor

    String hello = "Hello, world"; // String::String(const char*) constructor

    String *str = new String(hello); // String::String(const String&) constructor
    ...
    delete str; // String::~~String() destructor
} // String objects 's' and 'hello' implicitly destroyed
```

On exit from a scope, an object **destructor** is implicitly called for each stack allocated object defined in that scope.

Q: What to static and heap allocated objects when a scope is exited?

Object Construction, Assignment, and Destruction

In general, a class should define a default constructor, a copy constructor, an assignment operator, and a destructor.

A **default constructor** takes no arguments and is used to define an “instance” of some class that is initialized to some default state:

```
String s;    // an empty string object
```

A **non-default constructor** takes one or more arguments, and initializes an object using those arguments:

```
String a = "hello,world";    // initialize string object using a string literal
String b(buffer, size);      // initialize string object from a buffer of size bytes
```

A **copy constructor** is used to instantiate a new object as a copy of an existing object:

```
String x = a;    // copy the string from object a into a new object x
```

By default, assignment in C++ is a bitwise (shallow) copy. However, the **assignment operation** for an object can be overloaded to do a “deep copy”. So,

```
String h = "hello, world";
String g = "goodbye, world";
...
g = h;    // what does this do?
```

A **destructor** is a special function that allows an object to “clean up” before the storage for the object is reclaimed. A destructor method is identified by a special type signature using a ‘~’ followed by the classname. For example, the String destructor is given the special type signature ~String().

A destructor is rarely called explicitly. For stack allocated objects, the destructor is called implicitly by the C++ run-time when the scope in which the object was declared is exited. For heap allocated objects, the destructor is executed when **operator delete** is called on a pointer to the object.

Example: A Simple String Class

```
class String {  
  
private:  
    char* _string;  
    int   _length;  
public:  
    // default constructor takes no arguments  
  
    String() : _string(0), _length(0) {} // initialization via an "initializer list"  
  
    // a copy constructor takes one argument, namely a reference to a String object  
  
    String(const String& s) : _string(0) { // "copy" constructor  
        _length = s._length;  
        if (_length > 0) {  
            _string = new char[s._length + 1];  
            strcpy (_string, s._string);  
        }  
    }  
  
    // a non-default constructor that takes a constant string literal argument  
  
    String(const char* s) : _string(0), _length(0) {  
        if (s != 0) {  
            _length = strlen(s);  
            _string = new char[_length + 1]; // dynamic array creation  
            strcpy (_string, s);  
        }  
    }  
  
    ...  
}
```

```
// a non-default constructor that takes a constant string literal argument

String(const char* s, int size) : _string(NULL), _length(0) {
    if (s != NULL) {
        _length = size;
        _string = new char[_length + 1]; // dynamic array creation
        memcpy (_string, s, size);
        _string[_length] = '\\0'; // end string with a terminating NULL character
    }
}

// overloaded assignment operator--protects against self assignment

String& operator= (const String& s) {
    if (this != &s) { // protect against self assignment
        if (_string)
            delete [] _string; // delete old string

        // allocate a new _string and s._string
        _length = s._length;
        _string = new char[_length + 1]; // dynamic array creation
        strcpy (_string, s._string);
    }

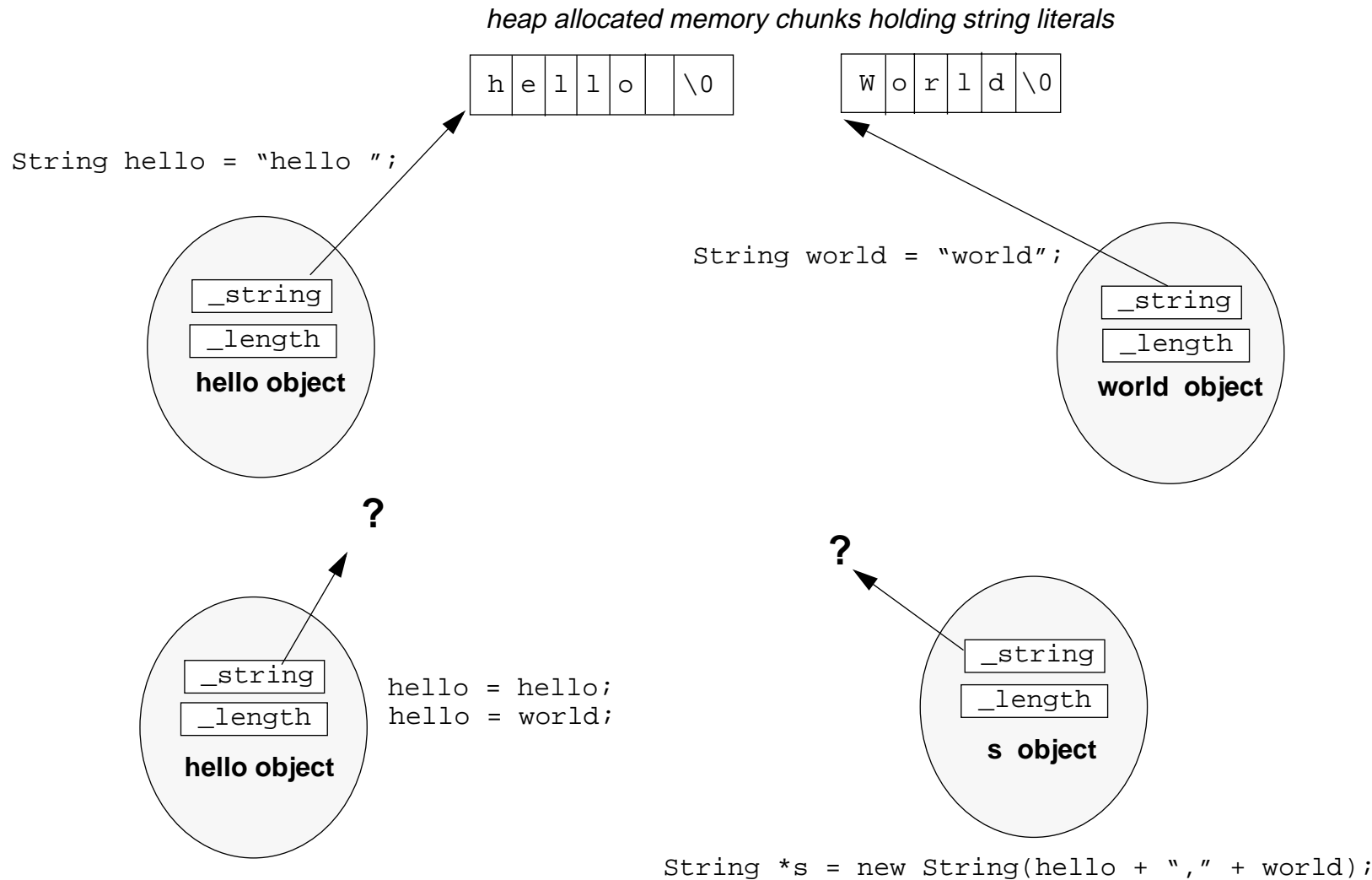
    return *this; // return "self"
}

// destructor
~String() { delete [] _string; } // [] signifies deleting of an array
...
};
```

“Accessor” and “Mutator” Member Methods

```
class String {  
private:  
  
    // ... as before  
  
public:  
    ... // constructors as before  
  
    // “accessor” methods do not modify an object’s instance variables  
  
    int length() const { return _length; } // same as return this->_length  
  
    // “mutator” methods modify an object’s instance variables  
  
    void capitalize();  
  
    // overloaded operators make objects appear as “builtin” types */  
    String& operator+(const String& s) { ...; return *this; }  
  
    // type conversion operators “convert” an object to a related type  
  
    const char* c_str() const { return _string; } // same as return this->_string  
  
    // friend functions can access private/protected data elements, but have no  
    // implicit ‘this’ pointer. You have to explicitly pass a string object by  
    // value or by reference  
  
    friend ostream& operator<<(ostream& os, const String& s) { return os << s._string; }  
};
```

Object Construction versus Object Assignment



What is the sizeof the hello object?