TCC CSCI-2843 C++ Programming Language

Additional Notes

1 The C and C++ Languages

1.1 History and Goals

C++ inherited much of its basic syntax from C. C was designed to be a high-level language with the ability to access low-level features of a computer. C has been declared a "portable assembler" due to its near-machine-level access while maintaining platform-independence. At the same time, however, it supports structured programming and separate compilation. The net result is a language that covers a wide array of programming needs, but sacrifices hardly any efficiency or optimization doing so.

C++ gains all the low-level access that C provides, but adds object-oriented and generic programming facilities for even higher-level programming capabilities. In C++, we can address programming problems at the following levels:

- System (near-or-at-hardware level) Programming
- Procedural/Structured Programming
- Object-oriented Programming
- Generic Programming

It is because of this that C++ is known as a **multi-paradigm** language and is one of the few that attempts to be so and succeed. Other languages that attempt and succeed at some level are

- Eiffel
- Objective-C
- Ada
- D

These notes fill in possible gaps in the C language that are critical for understanding some C++ features and also how most C and C++ programs are organized, compiled, and linked. Where appropriate, the notes will point out C features that are now considered anachronistic where newer C++ features and functionality are preferred.

1.2 Synopsis

The topics covered in these notes will be

- Types
- Variables
- Constants
- Pointers
- Arrays
- "C" Style Character Strings
- Functions
- Scope
- Compilation Units in C and C++
- Basic Preprocessor Usage
- Introduction to Libraries

2 Types

C and C++ support primitive types that are portable to most platforms ranging from 8-bit micro-controllers to modern 64 bit, multi-core processors. The types that C and C++ have supported since the beginning are

bool
char
signed char
unsigned char
short
unsigned short
long
unsigned long
int
unsigned int
long
unsigned long
long long
unsigned long long
float
double

The C++0X standard has introduced new types into C++, many of which have already been added to C to aid in numeric programming and to keep up with modern architectures and programming techniques:

char8_t
char16_t
char32_t
int8_t
int16_t
int32_t
int64_t
uint8_t
uint6_t
uint32_t
uint64_t

2.3 Character Types

There are four character types:

char
unsigned char
signed char
wchar_t

These are distinct types. The first three are inherited from C and are only guaranteed to hold a character of the "C implementation character set", which is a subset of ASCII, or (practically speaking) ASCII itself. The upshot of this is that you can theoretically only hold 7-bit characters in these types and stay portable. In practice, we freely store ASCII in **char** and some go out on a limb and store 8-bit character sets in **unsigned char**.

The standard does not specify whether **char** is signed or unsigned, so never rely on its signedness. If you want a signed character (in essence, an eight bit, signed integer) you can specify **signed char**. If the platform does not support a specific character type or signedness that you explicitly specify, you should receive an error or at least a warning, but those platforms with these restrictions are few and are usually special-purpose.

The fourth, **wchar_t**, was an earlier attempt at providing **wide character support** to C++. However, it was ill-defined whether it was intended to support 16-bit Unicode and its 16-bit transformations or the full Unicode standard, which requires 21 bits. Some platforms have implemented this as 16 bits while other have implemented this as 32 bits, making it non-portable with few standard guarantees.

The C++0x standard provides new character types to solve this ambiguity:

char8_t
char16_t
char32_t

Note there is no concept of signedness – these types are for character manipulation, and even though they are integral types (and can have integral arithmetic performed upon them), they should never be negative, so therefore should never need to be declared as "**unsigned**".

The **char8_t** type is capable of holding 8-bit (or fewer) character sets such as the C++ standard character set, ASCII, or any one of the ISO-8859 character sets that require 8 bits. Note that ISO-8859-1 is actually the first 256 characters of the Unicode standard, which (coincidentally) contains the 128 character ASCII standard as a subset. It may also hold UTF-8 transformation points to store transformed characters from the entire Unicode range.

The **char16_t** type is capable of holding 16-bit character sets such as UCS-16, which is the first 65536 characters of the Unicode standard, otherwise known as the Basic Multilingual Plane (BMP). It may also be used to hold UTF-16 Unicode transformation points to store transformed characters from the entire Unicode range.

The **char32_t** type is capable of holding characters from the full Unicode standard, known as UCS-4, **without** any transformations (although a "transformation" known as UTF-32 **does** exist).

2.4 Integer Types

The guarantees of the C standard (and thus C++ as well) for the size of unsigned integers are

1 ≤ **sizeof**(**unsigned char**)
≤ **sizeof**(**unsigned short**)
≤ **sizeof**(**unsigned int**)
≤ **sizeof**(**unsigned long**)
≤ **sizeof**(**unsigned long long**)

and for integers

1 ≤ **sizeof**(**signed char**)
≤ **sizeof**(**short**)
≤ **sizeof**(**int**)
≤ **sizeof**(**long**)
≤ **sizeof**(**long long**)

This is necessary to be portable to as many platforms as possible. The downside of this is that porting from one machine to another might change the limits of the values capable of being held by a given integer type.

To help mitigate this, new integer types with guaranteed sizes have been added to the standard:

int8_t
int16_t
int32_t
int64_t
uint8_t
uint6_t
uint32_t
uint64_t

This gives certain guarantees that **if** the type is supported on that platform, then the size and therefore the ranges are known. You still will not have 64-bit integers supported on an 8-bit microcontroller.

On **most** 32 or 64 bit platforms supporting **two's complement**, signed and unsigned, binary integers, the following holds:

N	min	max
int8_t	-128	127
int16_t	-32768	32767
int32_t	-2147483648	2147483647
int64_t	-9223372036854775808	9223372036854775807
uint8_t	0	255
uint16_t	0	65536
uint32_t	0	4294967295
uint_64_t	0	18446744073709551615

We still say “most” because there are always some odd architectures that insist on different representations than the “norm”.

On **most** 32 bit and **some** 64 bit platforms the following holds:

char is the same as **int8_t**
short is the same as **int16_t**
int is the same as **long**, which is the same as **int32_t**
long long is the same as **int64_t**
unsigned char is the same as **uint8_t**
unsigned short is the same as **uint16_t**
unsigned int is the same as **unsigned long**, which is the same as **uint32_t**
unsigned long long is the same as **uint64_t**

On **some** 64 bit platforms

int is the same as **int64_t**
unsigned int is the same as **uint64_t**

other combinations are possible, however and it is always a good idea to check on the platform you are using to see what is what.

3 Getting Started

3.5 Declarations and Definitions

C and C++ have a "one definition rule". A variable or function may be **declared** as many times as the programmer wants, but may be **defined** only once.

If a function or variable is **used** before it is **defined**, you will get a **compiler** error.

If a function or variable is **defined** more than once by time the program is **linked**, then you will get a **linker** error.

If a function or variable is **declared** (no matter how many times), but never **defined** in any source files by the time the program is **linked**, you will get a **linker** error.

3.5.1 Variables

To define a variable, we give it a type and a name:

```
int i;
```

This reserves memory for an integer variable named i.

We have the option to initialize a variable upon definition:

```
int i = 10;
```

It is generally good practice to initialize a variable upon definition.

3.5.2 Constants

To ensure that a value cannot change, we can declare it as "constant" using the **const** keyword:

```
const int max_headoom = 100;  // this int cannot change.
```

3.5.3 Pointers

A pointers variable merely holds the address of another variable. Each variable that gets created exists in a numbered "slot" in memory. That number is its **address**. To take the address of a variable, we use the **address-of operator**, "&". To declare a pointer to a variable of a given type, we prefix its name by an asterisk in its declaration/definition. To see this in action, consider

```
int a = 0;
int* b = &a;
```

Now b holds the address of a. We can get a's value through **dereferencing** b by applying the **dereferencing operator**, "*" to b:

```
int c = *b;
```

This dereferences b, thus retrieving the value of what it points at, which just happens to be a, and uses that as the value which is used to initialize c.

Pointers may be set to any arbitrary integral value, but of course accessing random spots in memory is a good way to crash your program. A special pointer value of zero indicates a **null pointer**. We signify this as

```
int* a = 0;
```

or using the old C macro

```
int* a = NULL;
```

The C++0X standard has added the keyword **nullptr** to specifically mean "null pointer":

```
int* a = nullptr;
```

Pointers are not initialized upon definition, so it is a good idea to always initialize a pointer, even if that is only to **null**. Accessing an uninitialized pointer is like accessing a random location in memory – sure to give incorrect results and probable crashes.

3.5.4 const Pointers

A pointer may be defined to point to something const:

```
int i = 0;
const int* p = &i;
```

This means that while p can change, we cannot assign to whatever it points to, effectively making the pointer "read only". Because p itself can change, we can reassign it to point to other **int**:

```
int j = 0;
p = &j;     // perfectly legal
*p = 10;   // still illegal – cannot change the int that p points to
```

To make a pointer constant, we need to put the **const** in a slightly different place:

```
int i = 0;
int* const p = &i;
```

This makes p unable to refer to any other integer but i. Note that we **can**, in this case, still change the value to which p points through dereferencing:

```
*p = 10;   // legal
```

To make a pointer that points to a value that cannot change **and** at the same time make the pointer unable to point to anything else, we can use two **const**s

```
int i = 0;
const int* const p = &i;
```

Note that the first **const** can either precede or follow the type, so that we could have written this

```
int const* const p = &i;
```

however, It is more common to place the initial **const** before the type.

3.5.5 Arrays

Arrays are contiguous blocks of memory that hold a number of items of a given type that can be addressed through an integral index. We must specify the number of items we wish to hold on the array definition:

```
int a[6];
```

This defines an area of memory, named "a", that holds six integers contiguously as depicted in Figure 3-1.

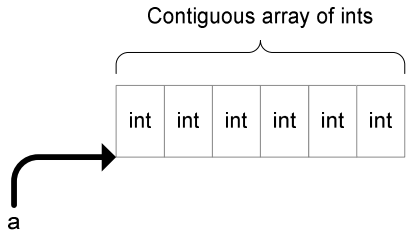

Figure 3-1

It is important to note that a is nothing more that an **address** from which all of the items in the array can be accessed. In essence, it is the address of the first element of the array. To access one of these integers, we use the **array index operator**, "[]" with an integer index:

```
a[0] = 1;
int b = a[3];
```

Note that the indices start at zero, so this sets the first integer in a to zero and initializes b to be whatever was in the fourth entry.

Since the name of an array is nothing more that an address, it essentially be treated like a **pointer**:

```
int a[6];

int* b = a;
```

The pointer "b" now refers to the first element of the array, just like the array name "a" does as depicted in Figure 3-2.

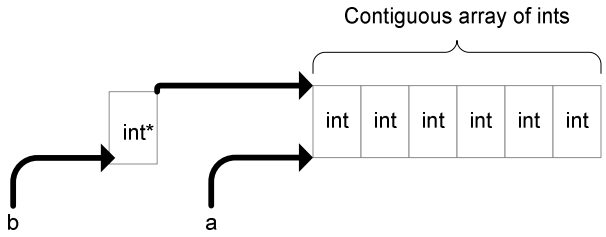

Figure 3-2

To make this useful, we must introduce **pointer arithmetic**, which is the ability to add or subtract integral values to and from a pointer to arrive at another address.

With pointer arithmetic, we may add and subtract integral values to and from a pointer to arrive at a new address;

```
int a[10];
int* b = a;
int* c = a + 2;
```

This is shown in Figure 3-3.

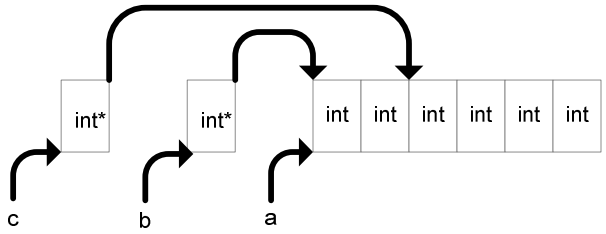

Figure 3-3

Note that this results in c pointing to the second **integer** past the beginning of the array **not** just the second **byte**. This is because with pointer arithmetic the address is always calculated by taking the integral offset and **scaling** by the size of the type of the pointer. In the above example, the pointer, c, points to an address that is **two times the size** of an **int** past b and **not** just two **bytes** past b.

For an array, we can see how the index operator must evaluate a[3] using pointer arithmetic since

```
a[3]
```

yields the same results as

```
*(a + 3)
```

as depicted in Figure 3-4.

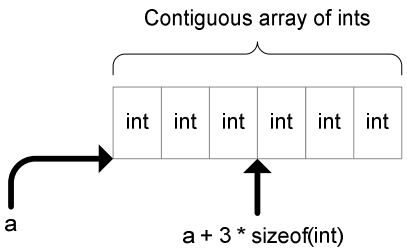

Figure 3-4

Since a is tantamount to a pointer, and since we could apply the indexing operator ("[]")to an array, it makes sense that we can apply that operator to a pointer with the same results:

```
int* b = a;
b[0] = 1;
```

In other words, we can treat **any** pointer as an array in addition to treating **any** array like a pointer. This underlines the fact that array access is nothing more that pointer arithmetic with a little syntactic sugar.

3.5.6 "C" Strings (nul-terminated Character Arrays)

To represent strings in the "C" language, programmers stored sequences of characters in arrays of **char**. Unfortunately, as mentioned in the discussion on arrays, arrays do not store their sizes, nor is there anyway to indicate that a character array is only partially used by a string that does not fill up the array. This led to the convention that all "C" strings must end with an ASCII nul character (character '\0'). Even a string constant like

```
"hello world"
```

is actually an array of characters that is usually used as a const char* that contains a "hidden" nul character:

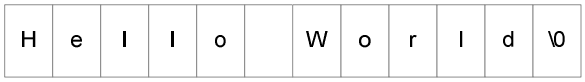

Figure 3-5

The number of characters stored in this example is 12 instead of 11. In order to find the length of this string, one must scan for the nul character, counting characters until it is found, which is precisely what the strlen function does.

```
const char* text = "Hello world";
int size = strlen(text);
```

This will assign the value or 11 to size.

There are a plethora of other "C" string manipulation functions available under the header file <cstring>. However, "C" strings are inherently difficult to use and can easily lead to illegal memory accesses due to overflowing of arrays or leaving off nul-terminations. In C++, we prefer to use the higher-level and safer std::string class. Note that even these facilities must cope with the legacy of string constants being represented as nul-terminated character arrays.

3.6 Functions

3.6.7 Introduction

Functions are executable blocks of code that have the ability to take in zero or more arguments and return a value. There are two ways to declare a function in C++. The first states the return type first, then the name of the function, followed by argument declarations enclosed in parenthesis. An example is

```
double f(double x);
```

Note this merely **declares** that there is a function f that takes a single argument of type **double** and returns a **double**. To **define** a function, we must provide a **body** containing zero or more statements:

```
double f(double x) { return x + 1.0; }
```

This is equivalent to the mathematical notation:

$$f(x) = x + 1.0 \text{ for } x \in \mathbb{R}$$

The major difference is that we must indicate the return type of the function, whereas this is implied in the mathematical notation. Functions that do not return a value have a return type of **void**. You cannot define a variable of type **void** – it merely is a type to indicate that there is no need to reserve storage for any value since there is none.

A function is just an area of code that can have control transferred to it and return right back whence it was called, possibly yielding a value (if not **void**) in the process. One key issue is to realize that how we call a function is to name the function, then apply the function call operator (parenthesis) to it, possibly passing arguments to it.

Functions may have variables that are only known inside of the body of the function. These are known as **local** or **auto** variables.

```
void f() {
    int x; // this x is only known within the function, f
}
```

The term "auto" refers to their storage classification of "automatic", which means that space is allocated for local variables when the function is called and that space is "automatically" reclaimed when the function returns. Arguments are merely a specific type of local variable. This concept is covered more fully in §Scope3.7 when we talk about "Scope".

3.6.8 Passing Arrays as Arguments

When passing an array into a function, we catch it as a pointer.

```
void f(int* v) {
}
void g() {
    int a[10];
    f(a);
}
```

Note that when we pass an array, we do **not** have anyway to note its size on the receiving end. The function must innately know what the size is, have it passed in as another argument, or use a sentinel value to note where the end of the array is, such as in nul-terminated characters arrays used as "C" strings.

3.6.9 Pointers to Functions (Function Pointers)

Functions can be pointed to by assigning the function to a **function pointer**. A function may be invoked through a function pointer by simply applying the function operator "()" to the function pointer. Consider

```
int f(double x) { … }

int main() {
    int (*my_f)(double) = f;

    int i = my_f(10.0);
    return 0;
}
```

This calls the function pointed to by my_f (which points to f), passing in 10.0.

3.6.10 The main Function

The entry point into a C++ program is the main function. A minimal main function looks like this

```
int main() {
    return 0;
}
```

If you typed this into a source file, compiled and linked it, then ran the resulting executable, it would work fine – it wouldn't **do** anything, but it would be legal and cause no problems. The return type of main is an **int** and the return value is often used by the operating system to determine the overall state of the program. The convention is that a zero return value indicates "no error" or "ok" and anything else is an error code that may be checked for further diagnostics.

The C++ standard does, however, state that main has the special ability to forgo the return and a zero will be understood as the return value. This is not true of any other function. Therefore, the shortest legal C++ program possible is

```
int main() { }
```

There are two optional arguments to the main function that give the number of command line arguments and an array of their values:

```
int main(int argc, char* argv[]) { … }
```

The argument values are given as an array of pointers to nul-terminated strings. The first argument is always the name under which the program was run, therefore the argument count will **always** be one or greater and the array will **never** be empty.

3.7 Scope

C and C++ organize programs using source files (aka. compilation units) in which to organize types, data, and functions. We have options as to where variables and functions are located and which parts of the program have **visibility** to them. There is also the concept of the **lifetime** of a variable, which also depends on where it is defined.

This relates to the memory model

- Code segment – stores executable code, usually read-only
- Data segment – holds externs and statics
- Stack segment – hold locals for functions and procedures
- Freestore ('heap' in C vernacular) for dynamic – will discuss later

At the compilation unit (.c and .cpp) level

extern (global)

Visible to all source files that declare a variable **extern**

Lifetime of program

static (at file level)

Visible only in compilation unit in which it is declared

Lifetime of program

local (**auto**)

Anywhere there is an open and close brace is a new scope

Created on stack in "activation records"

Visibility is from the declaration to the end of the scope in which it is defined (braces { })

Example is a local variable within a function's scope

Lifetime is the same as its visibility.

C++ inherited the way source files are structured and assembled into object files and executables from C. A C++ compiler compiles a **source file** to an **object file** that holds the binary, translated representation of the source code. This file is **not** an executable, but rather a file that holds **object code** generated from

the compilation of functions and descriptions the data that is used by the source file. For the code to be executed, it must first be **linked** to resolve addresses for data and functions.

Figure 3-6 shows an example of a single source file that has no declared external data that is compiled and linked to become an executable.

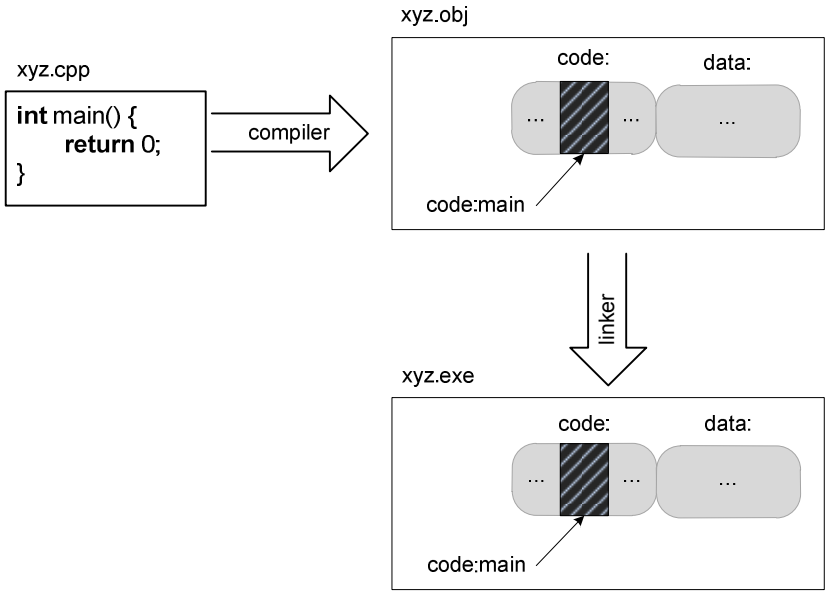

Figure 3-6

By convention, the extension, ".obj", is used on Microsoft platforms, while almost **everyone else** in the world, uses simply ".o".

After an object file is produced, the **linker** is invoked to finalize all object files into an **executable** image. On Microsoft platforms, executable have a ".exe" extension. On almost all other platforms, they are simply files without an extension and are marked as "executable".

Note that the code generated from the main function (as with all object code) is put in the code section. If the linker finds no main function (or, in general, any other referenced function), you will get an **unresolved function** linker error.

3.7.11 Using Externs

```
int i = 0;

void f() {
    …
}
int main() {
    …
    return 0;
}
```

This **defines** an **extern int** named i and initializes it to 0. Note the initialization happens when the program is loaded into memory.

Figure 3-7 shows what happens when we compile and link this program into an executable.

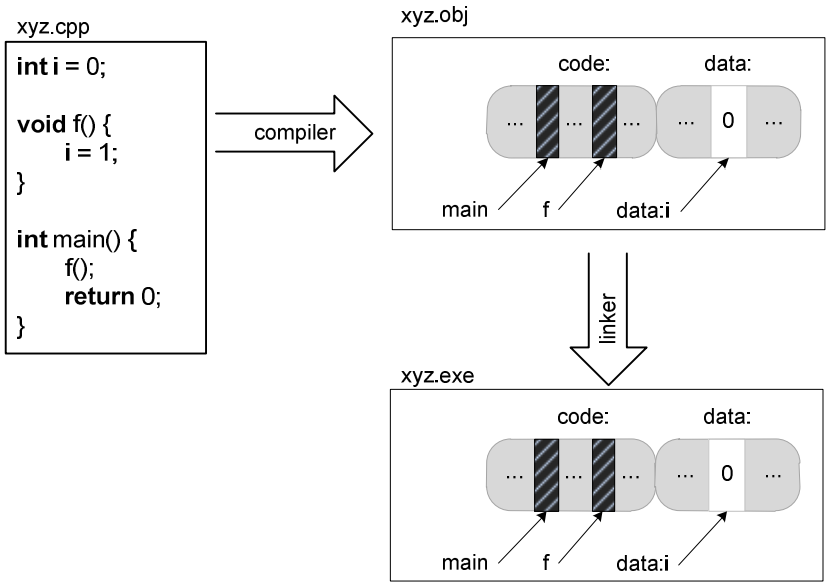

Figure 3-7

The final address of the function, f, is not known until link-time, just as the final address of the external **int**, i, is not known until then as well.

It is worthy to remark that even though it is not explicit, the function, f, is also **extern**. It **can** be made explicit, if desired:

```
extern void f() {
    …
}
```

An **extern** variable may only be defined and initialized once in one source file.

All other source files that want to reference this external variable must include the **declaration**:

```
extern int i;
```

Now we get a little more complicated and see why we separate the compile and link steps. In Figure 3-8, we have an external **int** and an external function that are referenced between two source files. Note the difference between the **declaration** of each and the **definition** of each, keeping in mind the "one definition rule" of C and C++.

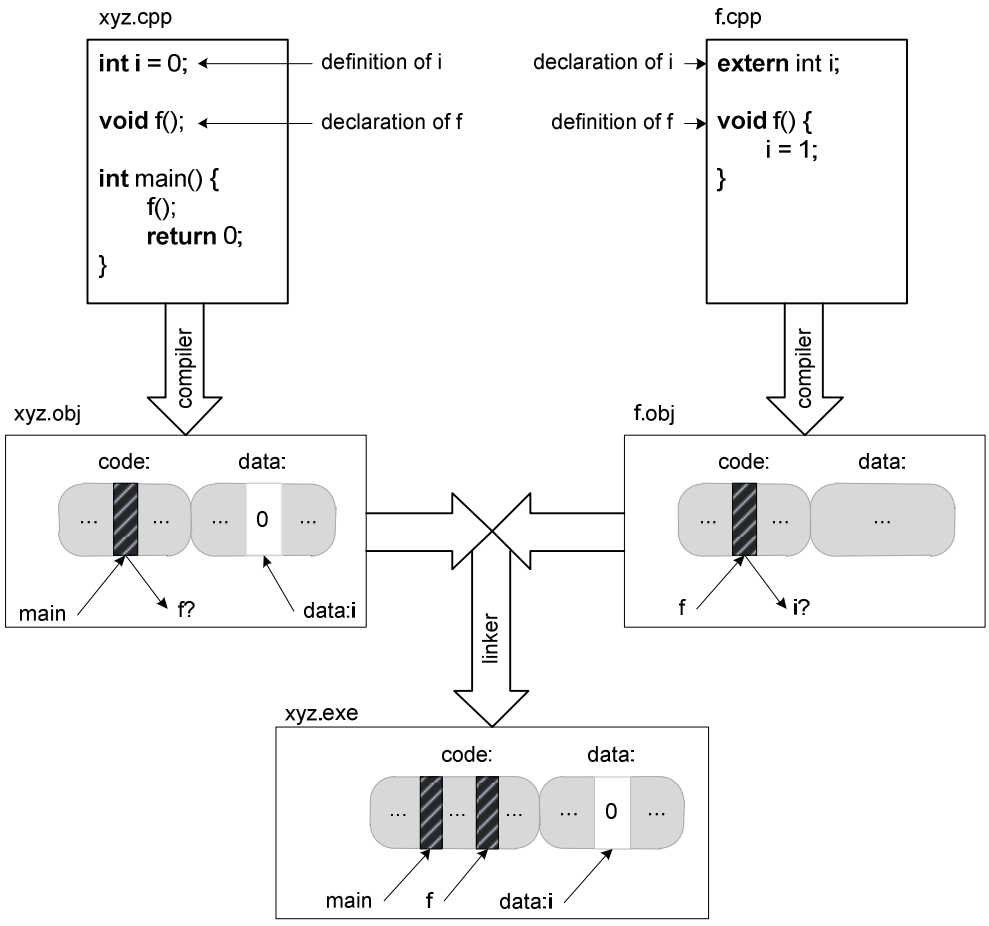

Figure 3-8

This shows that the two object files, xyz.obj and f.obj both hold information about what the compiler knew when it compiled the code. The xyz.obj object file contains information that **somewhere** there is declared to be a function named "f" with the indicated signature, but it has no other knowledge of **where** it is, let alone **what** it is composed of. Likewise, f.obj only knows that somewhere there is an integer named i, but knows not where it will be in the executable.

It is only at link-time that the addresses of the externals are known and (assuming the linker is fed the two object files as input) will be able to **resolve** the addresses so that the appropriate code will reference the appropriate address in the final executable image. In this example, the call to f in main is resolved to be a function call to wherever the code for f is placed in the code segment and the reference to i in f will be resolved to access the final address in the data segment.

3.7.12 Static Example:

Suppose the following was in a source file, xyz.cpp:

```
static int i = 0;

void f() {
    i = 1;
}
int main() {
    i = 2;
    f();
    return 0;
}
```

The variable, i, now is visible only to functions within xyz.cpp and **no other source file**.

3.8 Statements

Statements in C++ consist of expressions that can evaluate and assign values, or flow of control statements that alter the path a program takes.

3.8.1 Expressions

Believe it or not, except for flow of control statements (such as **if**, **while**, **do**, **for**), then everything else is an expression, but can still be used as statements. For example, the following is perfectly valid C/C++ code:

```
int main() {
    int a;
    a;
    3 + 3;
    a + 10;
    return 0;
}
```

All this does is evaluate some expressions whose values are never used.

3.8.2 Assignment

Assignment takes the form of

l-value = expression

Assignment itself is an **expression** – meaning that it not only assigns a value to something, but it yields a value itself. That is why we can write something like

```
int a = 0;
int b = 1;
int c = 2;
a = b = c = 3;
```

Note the order of evaluation (associativity) for the assignment operator is from **right to left**, meaning that the rightmost assignment is evaluated first, then the next rightmost, etc… At the end of the above assignment, all of the variables will have the value of 3.

3.8.3 Operators

The following is a list of all the C++ operators, along with their associativities.

Precedence	Associativity	Operator	Description
1	L to R	(…)	Grouping
		.	Member Operator
		[]	Index
		()	Function call
2	R to L	!	Logical not
		~	Bit-wise not
		++	Increment
		--	Decrement
		-	Unary minus
		+	Unary plus
		@	Dereference

		&	Unary reference
3	R to L	**	Power
4	L to R	*	Multiply
		/	Divide
		%	
5	L to R	+	Add
		-	Subtract
6	L to R	<<	Left shift (also used as output operator)
		>>	right shift (also used as input operator)
7	L to R	<	Less than
		>	Greater than
		<=	Less or equal to
		>=	Greater or equal to
8	L to R	==	Equal to
		!=	Not equal to
9	L to R	&	Bit-wise and
10	L to R	^	Bit-wise xor
11	L to R	|	Bit-wise or
12	L to R	&&	Logical and
13	L to R	||	Logical or
14	R to L	? :	Conditional
15	R to L	=	Assignment
		+=	Add to and assign
		-=	Subtract from and assign
		*=	Multiply by and assign
		/=	Divide by and assign
		%=	Modulus of and assign
		~=	Bit-wise not and assign
		&=	Bit-wise and and assign
		^=	Bit-wise xor with and assign
		|=	Bit-wise or and assign
		<<=	Left shift by and assign
		>>=	Right shift by and assign
		,	Sequencing operator
16	L to R		

4 Compilation Units

4.1 Compiling and Linking Source Files

C++ also inherited the way source files are structured and assembled into object files and executables. A C++ compiler compiles a single source file directly to an **object file** that holds the binary, translated representation of the source code. This file is **not** an executable, but rather a file that holds **object code** generated from the compilation of functions and descriptions the data that is used by the source file. For the code to be executed, it must first be **linked** to resolve addresses for data and functions.

4.2 Single Source File

Figure 4-1 shows an example of a single source file that has no declared external data that is compiled and linked to become an executable.

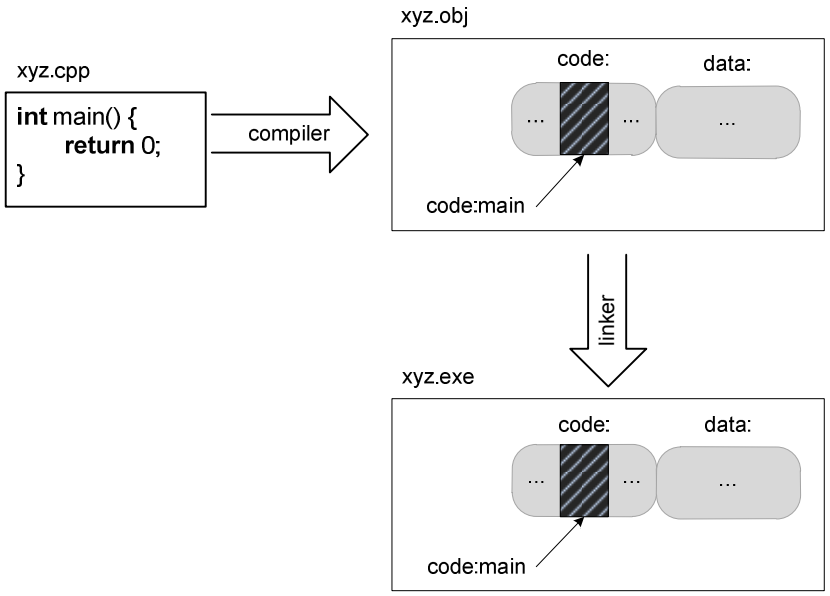

Figure 4-1

By convention, the extension, ".obj", is used on Microsoft platforms, while almost **everyone else** in the world, uses simply ".o".

After an object file is produced, the **linker** is invoked to finalize all object files into an **executable** image. On Microsoft platforms, executable have a ".exe" extension. On almost all other platforms, they are simply files without an extension and are marked as "executable".

Note that the code generated from the main function (as with all object code) is put in the code section. If the linker finds no main function (or, in general, any other referenced function), you will get an **unresolved function** linker error.

4.3 Single Source with External Data

In the previous example, we had no external data, so nothing (of ours) was put into the executable image in the "data" section. In Figure 4-3 we see a global (**extern**) integer defined in the source file that is compiled into an object file and linked into an executable. Note how the object file contains the external definition, but its final address is known only at "link time".

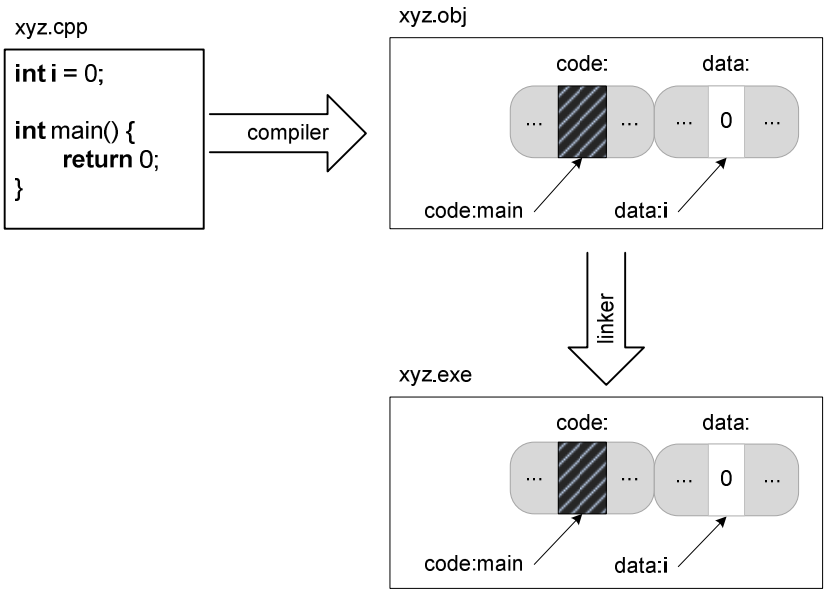

Figure 4-2

4.4 Single Source with a Function and External Data

Moving in a small increment, we now introduce a function into our single source file in Figure 4-3.

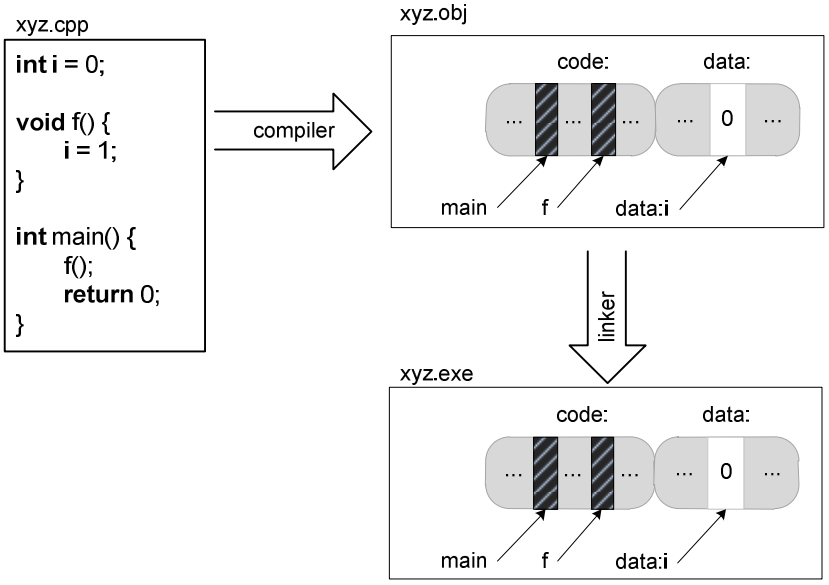

Figure 4-3

The final address of the function, f, is not known until link-time, just as the final address of the external **int**, i, is not known until then as well.

It is worthy to remark that even though it is not explicit, the function, f, is also extern. It **can** be made explicit, if desired:

```
extern void f() {
    …
}
```

4.5 Multiple Sources with Functions and External Data

Now we get a little more complicated and see why we separate the compile and link steps. In Figure 4-4, we have an external **int** and an external function that are referenced between two source files. Note the difference between the **declaration** of each and the **definition** of each, keeping in mind the "one definition rule" of C and C++.

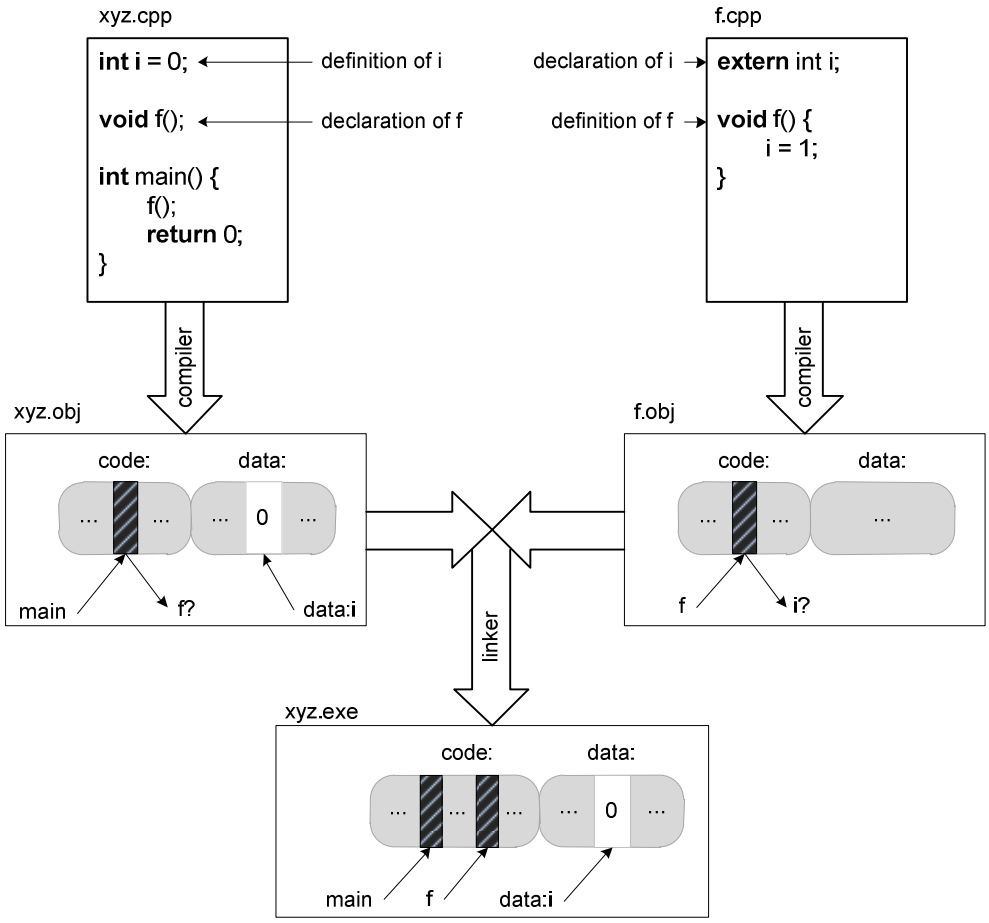

Figure 4-4

This shows that the two object files, xyz.obj and f.obj both hold information about what the compiler knew when it compiled the code. The xyz.obj object file contains information that **somewhere** there is declared to be a function named "f" with the indicated signature, but it has no other knowledge of **where** it is, let alone **what** it is composed of. Likewise, f.obj only knows that somewhere there is an integer named i, but knows not where it will be in the executable.

It is only a link-time that the addresses of the externals are known and (assuming the linker is fed the two object files as input) will be able to **resolve** the addresses so that the appropriate code will reference the appropriate address in the final executable image. In this example, the call to f in main is resolved to be a function call to wherever the code for f is placed in the code segment and the reference to i in f will be resolved to access the final address in the data segment.

4.6 The Preprocessor

The C (and thus C++) preprocessor is a utility that is invoked **before** the compilation step to allow **textual manipulation** of a source file. It is important to realize that the preprocessor does not know **anything** about types, objects, functions, or any other unit of information except characters in a file. In fact, the preprocessor can usually be invoked separately from the compiler to process text files that are not even source files.

4.6.1 Header Files

One of the main usages of the preprocessor in C++ is to include header files that contain **declarations** of types, functions, and **extern**s and **definitions** of inline functions. Figure 4-5 shows the processing of two source files that use a common header in such a fashion:

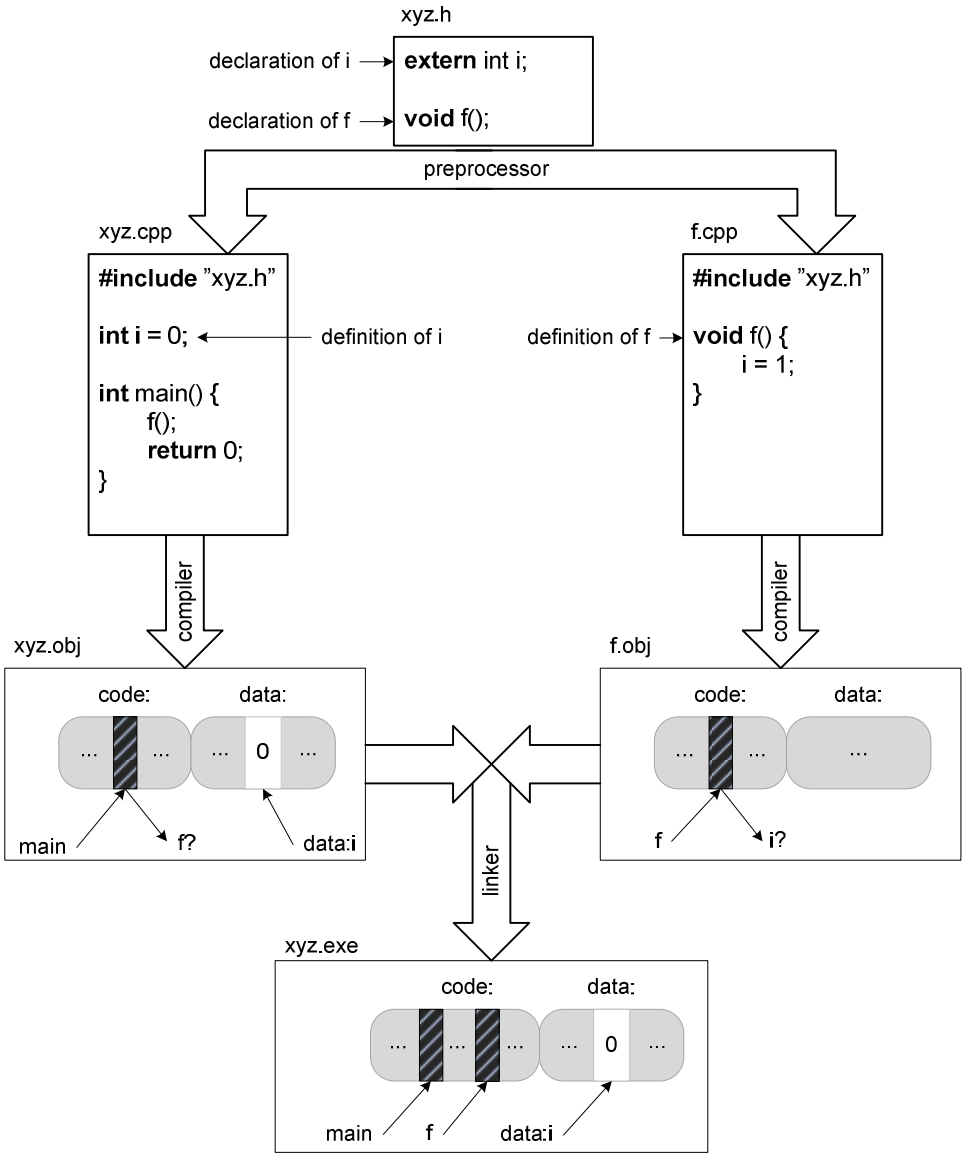

Figure 4-5

In both of these examples, the preprocessor merely places the text from the file, "xyz.h" as if it were part of the files, "xyz.cpp" and "f.cpp". For instance, to the compiler, the "xyz.cpp" file actually looks like:

```
extern int i;

void f();

int i = 0;

int main() {
    f();
    return 0;
}
```

Just as if this was the actual contents of the file.

4.6.2 Macros

One of the other purposes of the preprocessor is to provide macro processing facilities. This allows certain preprocessor directives to be replaced with text as either a direct textual replacement or by macros with arguments with the ability to perform limited textual manipulation.

As will be pointed out in turn, C++ offers language features that replace most of the uses of the preprocessor. These features are implemented by the compiler and thus **are** type-aware, giving better control and diagnostics.

4.6.3 Preprocessor Constants

Preprocessor allow us to define a preprocessor symbol that will have its associated text replaced everywhere that symbol is used:

```
#define MAX 10

if (i > MAX) {
    …
}

int j = MAX;
```

The preprocessor will textually substitute the text "10" everywhere the symbol MAX is found, such that the compiler will see the above like the following:

```
if (i > 10) {
    …
}

int j = 10;
```

Why did we need this facility? To be able to name certain constants and values to be more recognizable and understandable and to provide a method to easily change such values if need be. In the above example, if we wished to change MAX to, say 15, we can do so in exactly one spot and have the preprocessor put the correct value into the compiled code without us having to do a "search and replace" ourselves, which could be tedious and error-prone.

In C++, we have the ability to define constants, via **const**, to perform this exact functionality:

```
const int MAX = 10;
```

with the advantage that a const defined in this fashion is known to the **compiler** that knows the type and semantics surrounding using the symbol MAX instead of relying on lowly textual substitution.

4.6.4 Macros with Arguments

It is possible to declare and pass arguments to macros to provide function-like preprocessing effects:

```
#define ROUND(x) ((int) ((x) + .5)))
```

If this was used in code like the following

```
double r = 1.7;

int i = ROUND(r);
```

the preprocessor would see the symbol, ROUND, and note that it requires a single argument. The argument given is the **text**, "r" (remember that the preprocessor has no idea of types, variables, functions, etc...). The text "r" is bound to the macro's argument, x. The macro argument, x, is used as yet another text symbol to perform textual replacement within the macro's content itself, ending with the preprocessor handing off the following text to the compiler:

```
double r = 1.7;

int i = ((int) ((r) + .5)));
```

The advantage here is much like the advantage of taking a tedious or repetitious task and parameterizing it into functions. The additional advantage is that there is **no function call**, resulting in a faster evaluation of small expressions.

Why all the extra parenthesis? Experienced C programmers know that since the processor is purely **textual,** there are any number of ways to make it misbehave that do not yield proper or expected code in the resulting text passed to the compiler. If we wrote our macro without safeguards like this

```
#define ROUND(x) (int) (x + .5)
```

then we could abuse the compiler in ways that are sometimes baffling to both the compiler and other programmers:

```
int j = ROUND("hello");
int k = ROUND(r = i);
```

In the first example, we hand the following to the compiler:

```
int j = (int) ("hello" + .5);
```

The preprocessor knows no difference, but the compiler will complain about illegal pointer arithmetic. Without type-checking on the parameters, the actual error of passing in a **const char*** where a **double** is expected cannot be detected.

The second example is even more insidious, since even the compiler will not complain:

```
int k = (int) (r = i + .5);
```

This will unexpectedly and perversely assign the value of i + .5 to r **as a side-effect** as well as performing the rounding. Good luck interpreting such errors yourself, let alone explaining this to your colleagues.

These are mild in comparison to some of the "tricks" that you can perform with the preprocessor. It has become a good practice to surround the whole macro body in parentheses as well as every mention of a macro argument, but even this cannot stop bad programming practice (read "abuse") of the preprocessor. In general **avoid using the preprocessor** when better alternatives are available and **don't partake in abusive preprocessor "tricks"**.

C++ provides **inline functions** to replace macros with arguments and provide better type-safety and diagnostics, while still allowing for the factoring out of the function call:

```
inline int round(double x) { return int(x + .5); }
```

Now, the compiler is aware of the function's argument type(s) and the return type, and can treat a call to round as any other function. Internally, the function body will be inlined and no function call will be produced. Moreover, there is no way to perform subtle or inconspicuous "tricks" without using straight-forward, up-front C++ code.

4.6.5 __FILE__ and __LINE__

Another legitimate use of the preprocessor in C++ is to access the pre-defined preprocessor symbols that allow the retrieval of the file and line number that is currently being processed. The macros

```
__FILE__
```

and

```
__LINE__
```

are of types, **const char*** and **int**, respectively. Note there are **two** underscores before **and** after FILE and LINE. The __FILE__ macros expands out to a nul-terminated C-string whose contents contain the name of the file being processed. So if the following were in a file named "xyz.cpp"

```
#include <iostream>
using std::cout;
int main() {
    string file = __FILE__;
    cout << file << "@" << __LINE__ <<'\n';
    return 0;
}
```

then the output would be

```
xyz.cpp@5
```

This is useful in debugging as well as run-time logging of errors.

C++0X has added another preprocessor directive similar to __FILE__ and __LINE__, named __FUNC__, which will expand out to a string literal containing the name of the function in which the directive is used.

4.7 Libraries

Libraries are merely files that contain (potentially) multiple object files. They provide a more "packaged" way of providing the functionality of a set of code over providing multiple object files. Not only is a good container, but linkers are built to recognize library structure to access the object files within without any user intervention. In addition, most library formats provide an internal indexing scheme for finding the contents of the object files to provide more efficient (faster) linking and possibly browsing capabilities to programmers.

Libraries only provide the **object** files – in order for other source code to use the libraries, it must have the declarations of what are in the object files. That is why there are usually header files that accompany a library that contain the **declarations** for the functions, types, and (sigh) global data needed by the code held within the library. The process of creating and managing libraries is platform-dependant and often is a general facility that allows a library to hold code, data, text, or any other type of file.

The program used to create a library is generally referred to a "librarian", but that terminology is not used often. On Microsoft platforms, libraries use a ".lib" extension, while on most Unix-like systems, the extension, ".a" is used (short for "archive"). Figure 4-6 **Error! Reference source not found.**shows how source code is compiled into object files, which are then placed in a (typically indexed) library.

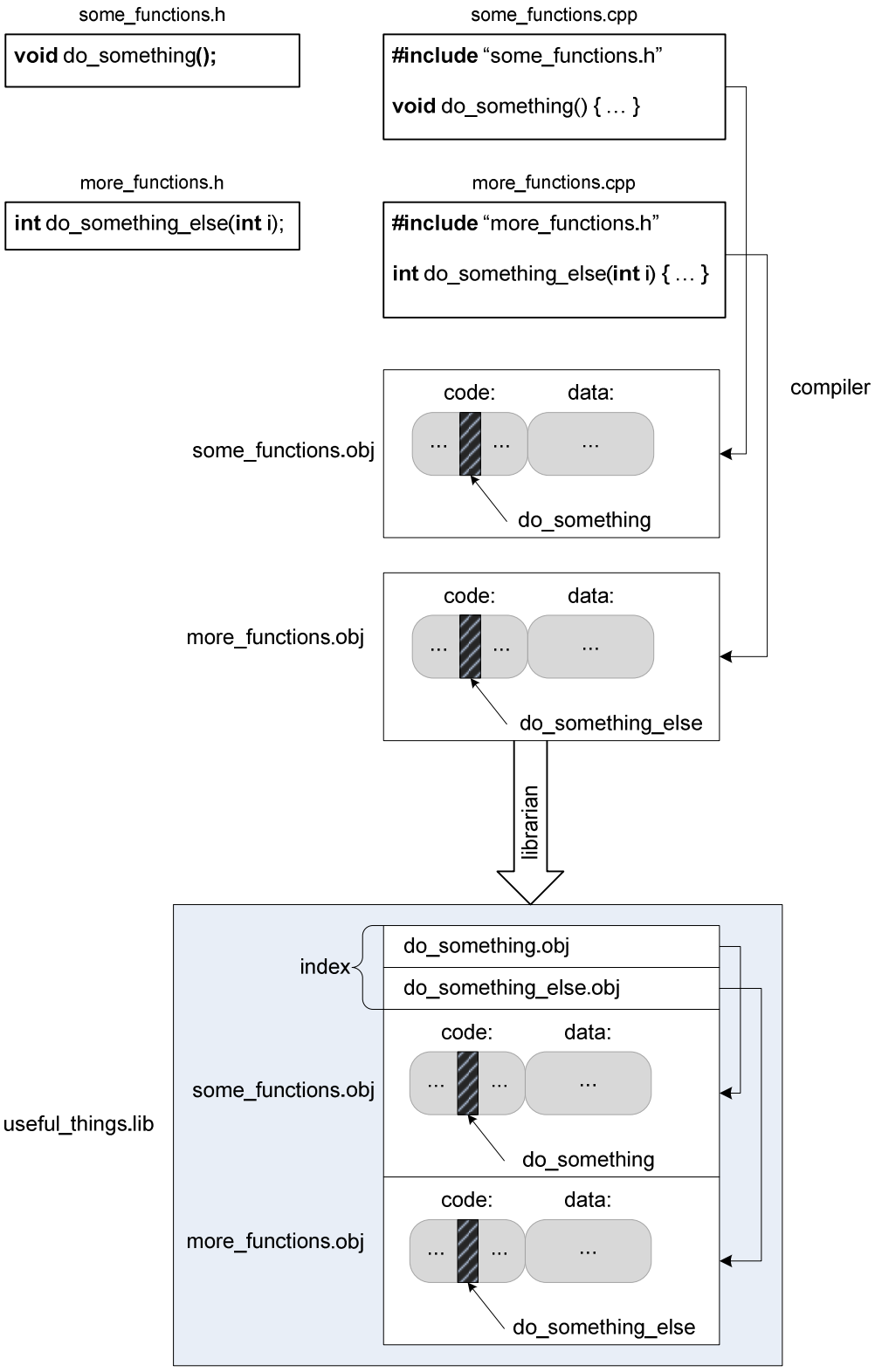

Figure 4-6

When a linker is handed a library, it behaves as if the object files in the library were passed to it as if they were separate object files. If the library is indexed, this can be used to significantly speed up linking.