Learning C from Java[1]

Differences between C and Java

The following is a list of differences between Java and C, and assumes that the reader knows less about the latter.

1. Speed of execution *vs.* portability

Java source and binaries are entirely portable (subject to availability of appropriate libraries), since the source format is standardized, and the binaries run on a software emulation of a standardized processor, which also slows execution. In C, binaries are not usually portable from one platform to another, because they use the platform's native hardware processor directly, and so run faster, but C source can be portable with little modification if it adheres to the ISO C standard (ISO/IEC 9899:1990, or C90), again subject to the available libraries.

There is now a new standard (ISO/IEC 9899:1999, or C99) adding some new features. These are pointed out where appropriate.

2. Speed of execution *vs.* speed/ease of development

Java may initially be seen as a slow language, since it is compiled from source into a bytecode, a low-level machine code for a non-existent processor, which then has to be interpreted by a software emulation of that processor. All of this takes time, but this is less significant in modern JVM implementations, which are able to compile some or all of the program's bytecode into native code at execution time (Just In Time, JIT). For long running programs (e.g. GUI-based applications, servers), this initial cost of translation is amortized by the speed improvement achieved subsequently. For short-lived programs, like shell commands, this advantage cannot be taken.

Java programs may be easier to develop since:

- ✓ Dynamic memory management is handled largely automatically, and
- ✓ Diagnostic exceptions are thrown for illegal operations (such as accessing through a null object reference, or accessing beyond the boundary of an array).
- ✓ Despite effective JIT optimization, C programs will usually run faster, however, since:
    - o Dynamic memory management (which is often not required) is fully under the programmer's control, and there are no exception checks for illegal operations (but a well-written program won't attempt them anyway), although these require greater responsibility from the programmer.

[1] This short version of C language tutorial for Java programmers is based on freely available materials in the Web and written by anonymous authors. This tutorial also include introduction to make utility and writing Makefile.

3. Primitive types

In C, the primitive types are referred to using a combination of the keywords `char`, `int`, `float`, `double`, `signed`, `unsigned`, `long`, `short` and `void`. The allowable combinations are listed below, but their meanings depend on the compiler and platform in use, unlike Java.

- `unsigned char`

  The narrowest unsigned integral type, typically (and always at least) 8 bits wide.

- `signed char`

  The narrowest signed integral type, of the same width as unsigned char.

- `char`

  An integral type equivalent to one or other of the signed/unsigned variants, but its *signedness* is implementation-dependent. C treats it as a distinct type, though.

- `unsigned short int`
- `unsigned short`

  An unsigned integral type at least as wide as unsigned char, typically (and always at least) 16 bits. The `int` is usually omitted.

- `signed short int`
- `signed short`
- `short int`
- `short`

  A signed integral type of the same width as `unsigned short int`. This is often just called `short`.

- `unsigned int`
- `unsigned`

  An unsigned integral type at least as wide as `unsigned short`, and wider than the `char` types. 16- or 32-bit widths are common.

- `signed int`
- `signed`
- `int`

A signed integral type of the same size as unsigned int. This is often just called int.

- unsigned long int
- unsigned long

An unsigned integral type at least as wide as unsigned int, typically (and always at least) 32 bits. The int is often omitted.

- signed long int
- signed long
- long int
- long

A signed integral type of the same size as unsigned long int. This is often just called long.

- unsigned long long int
- unsigned long long

In C99, unsigned long is typically (and always at least) 64 bits. The int is often omitted.

- signed long long int
- signed long long
- long long int
- long long

This is often just called long long.

- float

A single precision floating-point type. Typical size is 4 bytes.

- double

A double precision floating-point type. Typical size is 8 bytes.

- long double

An extended double precision floating-point type. The size of long double can be 16, 12, 8 bytes depending on the underlying system.

- void

An empty type. It has no value, and cannot be accessed. As in Java, C functions with no return value are defined to return void. Unlike Java, a function with no parameters has void in its parameter list.

Note that there is no `boolean` type. Instead, the test conditions of `if`, `while` and `for` statements, and the operands of the logical operators (`!`, `&&` and `||`), are integer expressions with a boolean interpretation: zero means false, non-zero means true. The relational operators (==, !=, <=, >=, < and >) and logical operators return 0 for false and 1 for true. For example, the following C code

```
int t = (x > y);
if (t) { printf("True!\n");}
  else { printf("False!\n"); }
```

is equivalent to

```
int t;
if (x > y) { t = -1; }
  else { t = 0; }
if (t != 0) { printf("True!\n"); }
  else { printf("False!\n"); }
```

However, since the C language standards allow the result of a comparison to be any non-zero value, the statement `if(t==1){...}` is not equivalent to `if(t){...}` or to `if(t!=0){...}`.

Since C standards mandate that 0 be interpreted as the null pointer when used in a pointer context or cast to a pointer, one can test whether a pointer is non-null by the statement `if(p){...}` instead of `if(p!=NULL){...}` — although some code styles discourage this shorthand. One can also write `if(x){...}` when x is a floating-point value, to mean `if(x!=0){...}`.

For convenience, many programmers and C header files use C's `typedef` facility to define a Boolean type (which may be named Boolean, boolean, bool, etc.). The Boolean type may be just an alias for a numeric type like `int` or `char`. In that case, programmers often define also macros for the true and false values. For example,

```
typedef int bool;
#define FALSE 0
#define TRUE (-1)
...
bool f = FALSE;
...
if (f) { ... }
```

The defined values of the TRUE and FALSE macros must be adequate for the chosen Boolean type. Note that, on the now common two's complement computer architectures, the signed value -1 is converted to a non-zero value (~0, the bit-wise complement of zero) when cast to an unsigned type, or assigned to an unsigned variable.

Another common choice is to define the Boolean type as an enumerated type (enum) allows naming elements in the language of choice. For example, the following code uses the English names FALSE and TRUE as possible values of the type *boolean*. In this case, care must be taken so that the false value is represented internally as a zero integer:

```
typedef enum { FALSE, TRUE } boolean;
...
boolean b = FALSE;
...
if (b) { ... }
```

Again, since a true result may be any non-zero value, the tests if(t==TRUE){...} and if(t), which are equivalent in other languages, are not equivalent in C.

In C99, there is a boolean type bool (which is really just a very small integer type) and symbolic values true and false (i.e. just 1 and 0), but the other integer types work just as well as before. In C99, there is _Bool data type. It is large enough to store the values 0 and 1. When any scalar value is converted to _Bool, the result is 0 if the value is 0, otherwise 1. If the <stdbool.h> is #included, the macros bool, true and false can be used to refer to _Bool, 1 and 0, respectively:

```
#include <stdbool.h>
int main()
{
  bool b = false;
  b = true;
}
```

4. Comments

Java allows the use of these forms of comment:

```
/* a multiline
   comment */
// a single line comment
```

C only allows the former. It is not wise to use such comments to temporarily disable sections of code, since they do not nest. Use the preprocessor (see later) instead:

```
  /* enabled code */
#if 0
  /* disabled code */
#endif
  /* enabled code */
```

In C99, the one-line comment is allowed.

5. Structures instead of classes

C does not allow you to declare class types (as you can in Java using the class construct), but you can declare C structures using the `struct` construct. A C structure is like a Java class that only contains public data members — there must be no functions, and all parts are visible to any code that knows the declaration. For example:

```
struct point {
  int x, y;
};
```

This declares a type called `struct point` ('`struct`' is part of the structure type name; `point` is known as the structure type's *tag*).

Members of a C structure are accessed using the . operator, as class members can be in Java:

```
struct point location;

location.x = 10;
location.y = 13;
```

A structure object may be initialized where it is defined:

```
struct point location = { 10, 13 }; /* okay; initialization (part of definition) */

location = { 4, 5 }; /* illegal; assignment (not part of definition) */
```

In C99, you can create anonymous structure objects to perform a compound assignment:

```
location = (struct point) { 4, 5 }; /* legal in C99 */
```

In C99, a structure initialization can specify which members are being set:

```
struct point location = { .y = 13, .x = 10 }; /* legal in C99 */
```

Unlike Java, where class variables are references to objects, C structure variables are the objects themselves. Assigning one to another causes copying of the members:

```
struct point a = { 1, 2 };
struct point b;

b = a;    /* copies a.x to b.x, and a.y to b.y */
b.x = 10; /* does not affect a.x */
```

6. Enumerations

An enumeration defines several symbolic integer constants with unique values in a convenient way. The following declares a new type `enum light`, and defines the symbols `RED` for 0, `REDAMBER` for 1, `GREEN` for 2, and `AMBER` for 3:

```
enum light { RED, REDAMBER, GREEN, AMBER };
```

The first symbol is assigned the value 0, and each subsequent symbol is assigned the next integer. However, a symbol can be assigned a particular value:

```
enum light { RED = 3, REDAMBER, GREEN = 1, AMBER };
```

This also implies that `REDAMBER` is 4, and that `AMBER` is 2.

If a new type is not required, the tag can be omitted:

```
enum { RED, REDAMBER, GREEN, AMBER };
```

The symbols can be used in any expression, and may be assigned to any integral type, not just the enum type. For this reason, the tag is rarely used.

Symbolic constants in Java usually have this form:

```
public static final int RED = 0;
```

```
public static final int REDAMBER = 1;
public static final int GREEN = 2;
public static final int AMBER = 3;
```

However, Java 1.5 has introduced a new enum family of classes, which achieves the above with greater type-safety, and a few other nice facilities:

```
public enum LightState { RED, REDAMBER, GREEN, AMBER }
```

7. Unions

C allows an area of memory to be occupied by data of several types, though only one at a time, using a union. Unions are syntactically similar to structures:

```
union number {
  char c;
  int i;
  float f;
  double d;
};
```

This declares a type called union number ('union' is part of the name; number is known as the union's *tag*).

Members of a C union are accessed using the . operator, just as structure members are accessed:

```
union number n;
int j;

n.i = 10;
j = n.i;
```

Only the member to which a value was last assigned contains valid information to be read. There is no way to determine that member implicitly, so the programmer must take steps to identify it, for example, by using a separate variable to indicate the type:

```
union number n;
enum { CHAR, INT, FLOAT, DOUBLE } nt;
```

```
n.i = 10;
nt = INT;

switch (nt) {
case CHAR:
  /* access n.c */
  break;
case INT:
  /* access n.i */
  break;
case FLOAT:
  /* access n.f */
  break;
case DOUBLE:
  /* access n.d */
  break;
}
```

Java does not have unions.

8. Single namespace for functions and global variables

Each class in Java defines a namespace which allows functions and variables in separate, unrelated classes to share the same name. When identifying a function or variable in Java, the namespace must be expressed, or implied using an import directive; for example, the method `java.lang.Integer.toString()` is distinct from `java.lang.Long.toString()`. Java packages allow distinct classes and interfaces to share the same name; for example, the name Object could refer to either `java.lang.Object` or `org.omg.CORBA.Object`.

In C, all functions are global, and must share a single namespace (*i.e.* one per program). Global variables can also be declared and defined, and they also share that namespace. Care must be taken in choosing names for functions in large projects, and often a strategy of using a common prefix for groups of related functions is employed, *e.g.* WSA prefixes most of the WinSock functions.

Note that other namespaces exist in C: a single namespace is shared by the tags of all structures, unions and enumerations; each structure and union holds a unique namespace for its members; each block statement holds a namespace for local variables.

9. Lack of function name overloading

In Java, two functions in the same namespace may share the same name if their parameter types are sufficiently different. In C, this is simply not the case, and all function names must be unique.

```
void myfunc(int a)
{
  /* ... */
}

void myfunc(float b) /* error: myfunc already defined */
{
  /* ... */
}
```

10. Type aliasing

New names or aliases for existing types may be created using `typedef`. For example:

```
typedef int int32_t;
```

This allows `int32_t` to be used anywhere in place of `int`, and such aliases are often used to hide implementation- or platform-specific details, or to allow the choice of a widely-used type to be changed easily.

`typedef` are also useful for expressing complex compound types. For example, a prototype for the standard-library function signal has the following, rather cryptic form (in ISO C).

As an example, let us first see

```
void (*signal(int sig, void (*func)(int)))(int);
```

What is this??

`void (*)(int)`

type pointer to (function with one `int` parameter returning `void`)

`*signal()`

`signal` is a function returning pointer.

`(*signal)()`

`signal` is a pointer to function

`void (*signal())(int);`

`signal` is a function returning pointer to (function with one `int` param returning `void`)

`void (*signal(int sig, void (*func)(int)))(int);`

`signal` is a function returning pointer to (function with one `int` param returning `void`). The first parameter is type `int`, the second parameter is type: `void (*)(int)` (i.e., pointer to function with one `int` parameter returning `void`).

The original declaration is equivalent to: `void (*signal(int, void (*)(int)))(int);`
The parameters of `` `signal' `` don't have to be named, similarly parameter to `` `func' `` was not named.

Note that the type of the second parameter of `` `signal' `` is the same as the type that `` `signal' `` returns. In other words, pointer to (function with one `int param`, returning `void`).

It's easier to understand if you declare a type:

```
typedef void (*ptr_void_fn_int)(int);
ptr_void_fn_int signal(int, ptr_void_fn_int);
```

`ptr_void_fn_int` is type "pointer to (to function with one `int` parameter returning `void`.

`` `signal' `` function for a given signal number sets the new signal handler you supply (which is type `ptr_void_fn_int`) and returns you the previous handler (which naturally has to be the same type).

Note that a `typedef` is *syntactically* similar to a variable declaration, with the new type name appearing in the place of the variable name.

There is no equivalent of type aliasing in Java.

11. Declarations and definitions

C programs are built from collections of functions (which have behavior) and objects (which have values; variables are objects), the natures of which are indicated by their types. C compilers read through source files sequentially, looking for names of types, objects and functions being referred to by other types, objects and functions.

A declaration of a type, object or function tells the compiler that a name exists and how it may be used, and so may be referred to later in the file. If the compiler encounters a name that does not have a preceding declaration, it may generate an error or a warning because it does not understand how the name is to be used.

In contrast, a Java compiler can look forward or back, or even into other source files, to find definitions for referenced names.

A definition of an object or function tells the compiler which module the object or function is in (see "Program modularity"). For an object, the definition may also indicate its initial value. For a function, the definition gives the function's behavior.

- **Functions and their prototypes**

In Java, the use of a function may appear earlier than its definition. In C, all functions being used in a source file *should* be declared somewhere earlier than their invocations in that file, allowing the compiler to check if the arguments match the function's formal parameters. A function declaration (or *prototype*) looks like a function definition, but its body (the code between and including the braces ('{' and '}') is replaced by a semicolon (similar to a native method, or an interface method, in Java). If the compiler finds a function invocation before any declaration, it will try to infer a declaration from the invocation, and this may not match the true definition. A proper declaration can be inferred from a function definition.

```
/* a declaration; parameter names may be omitted */
int power(int base, int exponent);

/* From here until the end of the file, we can make calls to power(),
   even though the definition hasn't been encountered. */

/* a definition; parameter names do not need to match declaration */
int power(int b, int e)
{
  int r = 1;
  while (e-- > 0)
    r *= b;
  return r;
}
```

- **Global objects (variables)**

Global objects also have distinct declarative and definitive forms. A definition may be accompanied by an initialize. For example,

```
int globval = 34; /* initialized */
int another;      /* uninitialized */
```

while a declaration should not have an initializer, and should be preceded by extern.

```
extern int globval;
extern int another;
```

(extern can also appear before a function declaration, but it is optional.)

- **Local objects (variables)**

For local objects in C, the definition and declaration are not distinguished. Unlike Java, all local variables must be defined at the beginning of their enclosing block, before any statements are reached. **This restriction does not apply in C99.**

```
{
  int x; /* a definition */

  x = 10; /* a statement */

  int y; /* illegal; follows a statement */
}
```

Furthermore, an iteration variable in a for loop cannot be declared within the initialization of the statement:

```
{
  for (int x = 0; x < 10; x++) { /* illegal */
    /* ... */
  }
}
```

This restriction does not apply in C99.

- **Scope**

All declarations have scope, which is the part of the program in which the declared name is valid. 'File scope' means *from the declaration to the end of the file*, and applies to types, functions and global objects.

'Block scope' means *from the declaration to the end of the block statement in which it is declared*. This always applies to local objects (and formal parameters), but can also apply to types, functions and global objects. All of the following declarations have block scope, and can be used by the trailing statements, but not beyond:

```
{
  /* a local type */
  typedef int MyInteger;

  /* a local variable */
  MyInteger x;

  /* global variable */
  extern int y;

  /* function (extern is implicit) */
  int power(int base, int exponent);

  /* statements... */
}
```

Unlike Java, a local variable in an inner block may hide one in an outer block by having the same name:

```
{
  int x;

  {
    int x; /* hides the other */
  }

  /* first one visible again */
}
```

12. Empty parameters lists

In Java, a function that takes no parameters is expressed using (). In C, such a function should be expressed with (void) in its declaration and definition. However, it is still invoked with ():

```
/* prototype/declaration */
int myfunc(void);

/* definition */
int myfunc(void)
{
  /* ... */
}
```

```
/* invocation */
myfunc();
```

The form () is permitted in declarations, but it means "unspecified arguments" rather than "no arguments". This tells the compiler to abandon type-checking of arguments where that function is invoked.

13. Program modularity

Java programs, particularly large ones, are usually built in a modular fashion that supports code re-use. The source code is spread over several source files (.java), and is used to generate Java byte-code in class files (.class) which are identified by the class they support, **so in Java, there is a direct relationship between the name of a class and the file containing the code for that class**. These are combined at run-time to produce the executing program. Java's standard library of utilities for file access, GUIs, internationalization, *etc.*, is a practical example of modular coding.

A large C program may also be split into several source files (usually with a .c extension), and compilation of each of these produces an *object* file of (usually) the same name with a different extension (.o or .obj). These are the modules of C that can be combined to form an executable program. An object file contains named representations of the functions and global data defined in its source file, and allows them to refer to other functions and data by name, even if in a separate module. **In C, there doesn't have to be any relationship between the names of functions and variables and the names of the modules that contain them.**

A final executable program is produced by supplying all the relevant modules (as object files) to a *linker* (which is often built into the compiler). This attempts to resolve all the referred names into the memory addresses required by the generated machine code, and **linking will fail if some names cannot be resolved, or if there are two representations of the same name**.

For example, the object file generated from the code below would contain references to the names pow (because it is invoked as a function) and errno (because it is accessed as a global variable), and would also provide a representation of the name func (because a definition of the function is provided).

```
extern int errno;

void func(void)
{
  double pow(double, double);
  double x = 3.0, y = 12.7, r;
  int e;

  r = pow(x, y);
```

```
  e = errno;

  /* ... */
}
```

Like Java, C comes with a standard library of general-purpose support routines, an implementation of which is supplied with your compiler. Its source code is not usually required, since it has already been compiled into object files for your system, and these will be used automatically when linking.

Other pre-compiled libraries may also exist (*e.g.* to support sockets), but it will normally be necessary to link with them explicitly to use them.

Here is an illustration of a program built from several components:

The source code consists of 4 source files (`foo.c`, `bar.c`, `baz.c`, `quux.c`) and 3 header files for preprocessing (`"yan.h"`, `"tan.h"`, `"tither.h"`; see "File inclusion"). The program also uses some header files (`<wibble.h>`, `<wobble.h>`) from an additional library. Compiling each of the source files in turn generates the object files `foo.o`, `bar.o`, `baz.o`, `quux.o`, and these are linked with an archive of pre-compiled objects (`libwubble.a`) from the library to produce an executable program `myprog`.

14. Preprocessing

Each C source file undergoes a lexical preprocessing stage which serves several purposes, including conditional compilation and macro expansion. The main purpose is to allow common declarations of types, global data and functions to be conveniently and consistently made available to modules which need to access them. In general, the preprocessor is able to insert, remove or replace text from the source code as it is supplied to the compiler (the original source code doesn't change).

There is no equivalent of preprocessing in Java.

- **File inclusion**

When a large C program is split over several modules, code in one module may need to make references to named code in another, or may use types that the other module uses. The usual way to achieve this is to precede the reference with a declaration that shows what the name means. Some example declarations:

```
/* this declares the type struct point */
struct point {
  int x, y;
};

/* this declares the global variable errno */
```

```
extern int errno;

/* this declares the function getchar */
int getchar(void);
```

It would be tedious to repeat such declarations in each source file that requires them (particularly if they need to be modified as the program develops), but these could instead be placed in a separate file (usually with a .h extension), and inserted automatically by the preprocessor when it encounters an #include directive embedded in the source code, for example:

```
#include "mydecls.h"
```

These *header* files are also preprocessed, and so may contain further #include (or other) directives.

Header files containing declarations for the standard library are also available to the preprocessor. These are normally accessed with a variant of the #include directive:

```
/* include declarations for input/output routines */
#include <stdio.h>
```

You should normally use the "" form for your own headers rather than <>.

Do not put definitions of functions or variables in header files — it may result in multiple definitions of the same name, so **linking will fail**. Header files should normally only contain types, function prototypes, variable declarations, and macro definitions. Note that inline functions are exceptional.

- **Macros**

The preprocessor allows macros to be defined which serve a number of purposes. Some macros are used to hold constants or expressions:

```
#define PI 3.14159
double pi_twice = PI * 2;
```

PI will be replaced by the numeric value wherever it is used.

Some macros take arguments:

```
#define MAX(A,B) ((A) > (B) ? (A) : (B))
```

that provide a convenient way to emulate functions without the overhead of a real function call. There are limitations of this macro.

Some macros are merely defined to exist:

```
#define JOB_DONE
```

and are used in conditional compilation.

- **Conditional compilation**

The preprocessor allows code to be compiled selectively, depending on some condition. For example, if we assume that the macro __unix__ is defined only when compiling for a UNIX system, and that the macro __windows__ is defined only when compiling for a Windows system, then we could provide a single piece of code containing two possible implementations depending on the intended target:

```
int file_exists(const char *name)
{
#if defined(__unix__)
  /* use UNIX system calls to find out if the file exists */
#elif defined(__windows__)
  /* use Windows system calls to find out if the file exists */
#else
  /* don't know what to do - abort compilation */
#error "No implementation for your platform."
#endif
}
```

The most common use of conditional compilation, though, is to prevent the declarations in a header file from being made more than once, should the file be inadvertently #included more than once:

```
/* in the file mydecls.h */
#if !defined(mydecls_header)
#define mydecls_header

typedef int myInteger;
```

```
#endif
```

You should normally protect all your header files in this way.

15. Pointers instead of references

All variables of non-primitive types in Java are references. C has no concept of 'reference', but instead has pointers, which Java does not.

A pointer is an address in memory of some ordinary data. A variable may be of pointer type, *i.e.* it holds the address of some data in memory.

```
/* we'll assume we're inside some block statement, as in a function */

int i, j; /* i and j are integer variables */
int *ip;  /* ip is a variable which can point to an integer variable */

i = 10;
j = 20;   /* values assigned */

ip = &i;  /* ip points to i */
*ip = 5;  /* indirectly assign 5 to i */
ip = &j;  /* ip points to j */
*ip += 7; /* j now contains 27 */
i += *ip; /* i now contains 32 */
```

The & operator obtains the address of a variable (the syntax ensures that there is no conflict with the bit-wise 'and' operator). The * operator *dereferences* the pointer (again, the syntax ensures that there is no conflict with the multiplication operator). A dereferenced pointer can be used on the left-hand side of an assignment, *i.e.* it is a *modifiable lvalue* ('el-value'), as in the two examples above.

- **Pointer types**

For every type, there is a pointer type. Since there is an `int` type, there is also a pointer-to-`int` type, written `int *`. `float *` is the pointer-to-`float` type. When assigning a pointer value to a variable, or comparing two pointer values, the types must match. Given these declarations:

```
int i, j;
float f;
int *ip;
```

```
float *fp;
```

ip is of type int *, so you can assign &i to it. &j can be compared with &i. But &f is of type float *, so it cannot be assigned to ip, or compared with ip, &i or &j.

- **Null and undefined pointers**

A valid value for a pointer may be null (it equals 0), indicating that it points to no object. **Do not dereference a null pointer.** Many of the standard header files define a macro for a null pointer, NULL, which some programmers may prefer.

```
#include <stdlib.h>

int *ip;
ip = NULL;
```

It is permissible to use pointers as integer expressions treated as boolean expressions to detect a null pointer (null means 'false' in this context). For example:

```
int *ip;

if (ip) {
  /* ip is not null */
}

if (!ip) {
  /* ip is null */
}
```

Direct comparisons are also possible (*e.g.* ip != NULL). If a pointer variable has been neither initialized nor assigned the address of a real object, it could be pointing anywhere, or be null. **Do not dereference such an undefined pointer.**

- **Dangling pointers**

In Java, an object will remain in existence so long as there is a reference to it. In C, an object may go out of existence even if there are pointers to it — the programmer is entirely responsible for ensuring that pointers contain valid addresses (either 0, or the address of an existing object) when used. This badly written function returns a pointer to an integer variable:

```
int *badfunc(void)
{
  int x = 18;

  return &x; /* bad - x won't exist after the call has finished */
}
```

The pointer returned by `badfunc()` is invalid.

- **Passing arguments by reference**

In Java, all primitive types are passed to functions by value — the function is unable to change values of variables in the invoking context. All reference types are passed by reference — the function can alter the public contents of the referenced object.

In C, *almost all* types are passed by value, and so no variables supplied as arguments can be altered by a function. It can only alter its local copy of the variables. However, by passing a pointer to the variable, the function is able to dereference its copy of the pointer, and indirectly assign to the variable. Consider these two functions which are intended to swap the values of two variables:

```
void badswap(int a, int b)
{
  int tmp = b;
  b = a;
  a = tmp;
  /* a and b are swapped but they're only copies */
}

void goodswap(int *ap, int *bp)
{
  int tmp = *bp;
  *bp = *ap;
  *ap = tmp;
}

/* assume we're in a function body */
int x = 10, y = 4;

printf("1: x = %d y = %d\n", x, y); /* print state of variables */
badswap(x, y);
```

```
/* x and y are copied, and the copies are swapped
   so x and y are unchanged */
printf("2: x = %d y = %d\n", x, y);

goodswap(&x, &y);
/* pointers tell goodswap() where we store x and y */
printf("3: x = %d y = %d\n", x, y);
```

This reports:

```
1: x = 10 y = 4
2: x = 10 y = 4
3: x = 4 y = 10
```

…indicating that badswap had no effect on the variables given as arguments.

- **Pointers to structures and unions**

A pointer to a variable of structure type may exist. Accessing a member of the structure is straight-forward: dereference the pointer, and apply the . operator. However, parentheses are needed to ensure the correct meaning, but a short form also exists (and is widely used) for convenience:

```
struct point loc;
struct point *locp = &loc;

(*locp).x = 10; /* correct */
*locp.x = 10;   /* incorrect; same as *(locp.x)  */
locp->x = 10;   /* correct, shorter form */
```

Syntactically, pointers to unions are accessed identically.

- **Pointers to functions**

Functions also have addresses, for which there are pointer-to-function types expressing the parameters and return type. The pointers can be passed to or returned from other functions just as other data can.

```
void goodswap(int *, int *);
void (*swapfunc)(int *, int *); /* a pointer called swapfunc */
int x, y;
```

```
swapfunc = &goodswap;      /* now it points to a function  with matching parameters */
(*swapfunc)(&x, &y);       /* invokes goodswap(&x, &y) */
```

Since pointers to functions are just values like any other, they can be passed to and returned from functions, so that 'behaviour' becomes just another form of data.

- **Pointers to pointers**

A pointer may point to variable which itself holds another pointer, and this is expressed in the pointer's type:

```
int    i;         /* i holds an integer */
int   *ip   = &i;  /* ip points to i */
int  **ipp  = &ip; /* ipp points to ip */
int ***ippp = &ipp; /* ippp points to ipp */
/* et cetera */
```

The fact that the pointed-to object also holds a pointer does not fundamentally change the behaviour of the pointer that points to it. It just allows a further level of indirection — in practice, you rarely need more than a couple of levels.

- **Generic pointers**

It is sometimes necessary to store or pass pointers without knowing what type they point to. For this, you can use the generic pointer type void *. You can convert between the generic pointer type and other pointer types (but not pointer-to-function types) whenever you need to:

```
int x;
int *xp, *yp;
void *vp;

xp = &x;

vp = xp;   /* types are compatible */

/* later... */

yp = vp;   /* types are compatible */
```

A generic pointer cannot be dereferenced, nor can pointer arithmetic be applied to it.

```
x = *vp;  /* error: cannot dereference void **/
vp++;     /* error: cannot do arithmetic on void **/
```

The generic pointer type simply allows you to tell the compiler that you're taking responsibility for a pointer's interpretation, and so no error messages or warnings are to be reported when assigning. It is the programmer's responsibility to ensure that the pointer value is interpreted as the correct type.

```
int *ip;
float *fp;
void *vp;

fp = ip;  /* error: incompatible types */
vp = ip;  /* okay */
fp = vp;  /* no compiler error, but is misuse */
```

Generic pointers are used with dynamic memory management, among other things.

16. Arrays and pointer arithmetic

Arrays in Java are reference types with automatic allocation of memory. In C, arrays are groups of variables of the same type in adjacent memory. Allocation for dynamic arrays is handled by the programmer. An array of integers may look like this:

```
int array[10]; /* numbered 0 to 9 */
int i = 6;

array[3] = 12;
array[i] = 13;
```

- **Initializing arrays**

Arrays may be initialized when defined:

```
int myArray[4] = { 9, 8, 7, 6 };
```

The size is optional in this case, since the compiler sees that there are four elements in the initializer. The initializer must not be bigger than the size if specified, but it can be smaller. Either way, the size

must be known at compile time — it must not be an expression in terms of the values of other objects or function calls.

In C99, you can specify which elements of an array are initialised:

```
int myArray[4] = { [2] = 7, [0] = 9, [1] = 8, [3] = 6 };
```

- **Array-pointer relationship**

The address of an array element can be taken, and simple arithmetic can be applied to it. Adding one to the address makes it point to the next element in the array. Subtracting one instead makes it point to the previous element.

```
int myArray[4] = { 9, 8, 7, 6 };
int *aep = &myArray[2];
int x, i;

*(aep + 1) = 2;   /* set myArray[3] to 2 */
*(aep - 1) += 11; /* set myArray[1] to 19 */
x = *(aep - 2);   /* set x to 9 */
```

By definition, `*(aep + i)` is equivalent to `aep[i]`, and in many contexts, an array name such as `myArray` evaluates to the address of the first element, which is how expressions such as `myArray[2]` work (it becomes `*(myArray + 2)`). The code above could be written as:

```
int myArray[4] = { 9, 8, 7, 6 };
int *aep = &myArray[2];
int x, i;

aep[1] = 2;    /* set myArray[3] to 2 */
aep[-1] += 11; /* set myArray[1] to 19 */
x = aep[-2];   /* set x to 9 */
```

Note that an array name such as `myArray` cannot be made to point elsewhere:

```
int myArray[4];
int i;
int *ip;
```

```
ip = myArray; /* okay: myArray is a legal expression; ip now points to myArray[0] */
myArray = &i; /* error: myArray is not a variable */
```

- **Passing arrays to functions**

Arrays are effectively passed to functions by reference. The array name evaluates to a pointer to the first element, so the function's parameter has a type of 'pointer-to-element-type'. For example, given the function:

```
void fill_array_with_square_numbers(int *first, int length)
{
  int i;

  for (i = 0; i < length; i++)
    first[i] = i * i;
}
```

we could write code such as:

```
int squares[4], moresquares[10];
void fill_array_with_square_numbers(int *first, int length);

fill_array_with_square_numbers(squares, 4);
fill_array_with_square_numbers(moresquares + 2, 7);
```

The second call only fills part of the array moresquares.

Note that the programmer must take steps to indicate the length of the array, in this case by defining the function to take a length argument (an alternative would be to identify a special value within the array to mark its end). The second call only has elements 2 to 8 set (an array of length 7).

- **Array length**

Because squares above is the ***name*** *of an array*, we can obtain its length using `sizeof squares`, which returns the total size as a number of chars. `sizeof squares[0]` returns the size (in chars) of one element, and since all the elements are of the same size, the ratio of these two sizeofs is the number of elements in the array:

```
fill_array_with_square_numbers(squares, sizeof squares / sizeof squares[0]);
```

(For arrays of chars, the divisor can be omitted, since `sizeof(char)` is defined to be 1.)

However, this technique doesn't work if the argument to sizeof is a pointer to the first element of an array: consider that such a pointer looks identical to a pointer to a single object, as far as the compiler is concerned — they don't contain any information about the length. This is why the example function above requires the length as a separate argument: within the function, `sizeof` first would only give the size of a pointer to an integer, not the length of the array.

- **Arrays as function parameters**

Note that a function parameter of array type isn't treated as an array, but as a pointer (the array syntax is allowed, but only pointer semantics are implemented). The following declaration is equivalent to the one above:

```
void fill_array_with_square_numbers(int first[], int length);
```

Within the definition of this function, `sizeof` first will still equal `sizeof(int *)`, even if we place a length inside the square brackets (such a value is ignored anyway).

17. const instead of final

Java uses the keyword final to indicate 'variables' which can only be assigned to once (usually where they are declared). C uses the keyword const with an object declaration to indicate a constant object that can (and must) be initialized, but cannot subsequently be assigned to — it is not a variable, but it still has an address and a size (so you can write `&obj` or `sizeof obj`).

```
double sin(double); /* mathematical function sine */

const double pi = 3.14159;
double val;

val = sin(pi); /* legal expression */
pi = 3.0;      /* illegal; not a modifiable lvalue */
```

- **Pointers to const objects**

`const` is useful when declaring functions that take pointers or arrays as arguments, but do not modify the dereferenced contents:

```
int sum(const int *ar, int len)
{
```

```
  int s = 0, i;

  for (i = 0; i < len; i++)
    s += ar[i];
  return s;
}

int array[] = { 1, 2, 4, 5 };

int total = sum(array, 4);
```

The const assures us that the invocation will not attempt to assign to `*array` (or `array[1]`, `array[2]`, *etc*).

- **const pointers**

Pointers themselves can be declared const just like other objects. In these cases, the pointer can't be made to point elsewhere, but what it points to can be modified (assuming that that isn't further const-qualified). Careful positioning of the keyword const is required to distinguish constant pointers from pointers to constants:

```
int array = { 1, 2, 4, 5 };
int *ip = array;              /* a pointer to an integer */
int *const ipc = array;       /* a constant pointer to an integer */
const int *const icpc = array; /* a constant pointer to a constant integer */

ipc[0] = ipc[1] + ipc[2];     /* okay */
ip += 2;                      /* okay */
ipc += 1;                     /* wrong; pointer is constant */
icpc[1] += 4;                 /* wrong; pointed-to object is constant */
```

This example shows a modifiable array whose members are being accessed through four pointers with slightly different types.

18. Inline functions

C99 supports inline functions. The programmer can indicate to the compiler that a function's speed is critical by making it inline:

```
inline int square(int x)
```

```
{
  return x * x;
}
```

If this definition is in scope, and you make a call to it, the compiler may choose not to actually go through the overhead of calling the function, but effectively place a copy of it inside the calling function.

Inline function definitions can (and often should) appear in header files instead of their prototypes. A normal ('external') definition must still be provided — for example, some part of your program may try to obtain a pointer to the function, and only a normal definition can provide that.

If the inline definition is in scope, an equivalent external definition can be generated from it by simply re-declaring the function with extern:

```
extern int square(int x);
```

If the inline definition isn't in scope, you could provide a normal definition which doesn't actually match the inline definition — but this could lead to confusing behaviour.

19. Characters and strings

A Java variable of type char can hold any Unicode character. In C, the char type can represent any character in a character set that depends on the type of system or platform for which the program is compiled. This is usually a variation of US ASCII, but it doesn't have to be, so beware. In particular, it could be a multibyte encoding, where a larger set of characters are represented by several char objects, e.g. UTF-8; a basic set of characters, however, are always represented as single chars.

Java strings are objects of class java.lang.String or java.lang.StringBuffer, and represent sequences of char.

Strings in C are just arrays of, or pointers to, char. Functions which handle strings typically assume that the string is terminated with a null character '\0', rather than being passed length parameter. A character array can be initialized like other arrays:

```
char word[] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' };
char another[] = "Hello!";
```

Note that the second initializer is a shorter form of the first, including the terminating null character. Such a string literal can also appear in an expression. It evaluates to a pointer to the first character.

```
const char *ptr;
```

```
ptr = "Hello!";
```

ptr now points to an anonymous, statically allocated array of characters. Attempting to write to a string literal like this has undefined behavior, so the use of const ensures that such attempts are detected while compiling.

Utilities for handling character strings are declared in <string.h>. For example, the function to copy a string from one place to another is declared as:

```
char *strcpy(char *to, const char *from);
```

and may be used like this:

```
#include <string.h>

char words[100];

strcpy(words, "Madam, I'm Adam.");
```

Like many of the other <string.h> functions, strcpy *assumes* that you have already allocated sufficient space to store the string.

20. Dynamic memory management

Dynamic memory management is built into Java through its new keyword and its garbage collector. In C, it is available through two functions in <stdlib.h> which are declared as:

```
void *malloc(size_t s); /* reserve memory for s chars */
void free(void *);      /* release memory reserved with malloc() */
```

(size_t is an alias for an unsigned integral type.)

malloc(s) returns a pointer to the start of a block of memory big enough for s chars. It returns a generic pointer which can be assigned to a pointer of any type. The memory is not initialized. All such allocated memory must be released when it is no longer required by passing a pointer to its start to free(). Only pointer values returned by malloc() can be passed to free().

You can find out the amount of memory needed to store an object of a particular type using sizeof(*type*). For an array, multiply this by the required size of the array.

```
long *lp;
long *lap;

lp = malloc(sizeof(long));
lap = malloc(sizeof(long) * 10);

/* now we can access *lp as a long integer,
   and lap[0]..lap[9] form an array */

free(lap);
free(lp);

/* now we can't */
```

malloc() returns a null pointer (0) if it cannot allocate the requested amount of memory.

21. Lack of exceptions

Java supports exceptions to cover application-defined mistakes as well as more serious system or memory-access errors (such as accessing beyond the bounds of an array).

In C, application-defined error conditions are normally expressed through careful definition of the meaning of values returned by functions. More serious errors, such as an attempt to access memory that hasn't been allocated in some way, may go unnoticed (because the behaviour is undefined). Write-access to such memory may cause corruption of critical hidden data, which only results in an error at a later stage, so the original cause of the error may be difficult to trace. **Just because some activity is illegal in C, it doesn't mean that you will necessarily be told about it, either by the compiler or by the running program.**

22. main() function

In a Java application, execution begins in a static method (void `main(String[])`) of a specified class. In C, execution also begins at a function called main, but it has the following prototype:

```
int main(int argc, char **argv);
```

The parameters represent an array of character strings that form the command that ran the program. `argv[0]` is usually the name of the program, `argv[1]` is the first argument, `argv[2]` is the second, ..., `argv[argc - 1]` is the last, and `argv[argc]` is a null pointer. For example, the command

```
myprog wibbly wobbly
```

may cause main to be invoked as if by:

```
char a1[] = "myprog";
char a2[] = "wibbly";
char a3[] = "wobbly";

char *argv[4] = { a1, a2, a3, NULL };

main(3, argv);
```

The parameters are optional (you can replace them with a single void), but main always returns `int` in any portable program. Returning 0 tells the environment that the program completed successfully. Other values (implementation-defined) indicate some sort of failure. `<stdlib.h>` defines the macros `EXIT_SUCCESS` and `EXIT_FAILURE` as symbolic return codes.

23. Standard library facilities

Java comes with a rich and still-developing set of classes to support I/O, networking, GUIs, *etc*, to access a process's environment.

Similarly, the C language has a core of facilities to access its environment. These functions, types and macros form C's Standard Library. It is necessarily limited in order to support maximum portability (it provides no GUI facilities, for example), but it is largely fixed and stable. Access to other facilities (GUI, networking) is through additional libraries that are usually specific to your platform.

The headers of the C Standard Library are briefly summarized below:

`<stddef.h>`

Some essential macros and additional type declarations

`<stdlib.h>`

Access to environment; dynamic memory allocation; miscellaneous utilities

<stdio.h>

Streamed input and output of characters

`<string.h>`

String handling

<ctype.h>

Classification of characters (upper/lower case, alphabetic/numeric *etc*)

<limits.h>

Implementation-defined limits for integral types

<float.h>

Implementation-defined limits for floating-point types

<math.h>

Mathematical functions

<assert.h>

Diagnostic utilities

<errno.h>

Error identification

<locale.h>

Regional/national variations in character sets, time formats, *etc*

<stdarg.h>

Support for functions with variable numbers of arguments

<time.h>

Representations of time, and clock access

<signal.h>

Handling of exceptional run-time events

<setjmp.h>

Restoration of execution to a previous state

C95 additionally provides the following headers:

<iso646.h>

Alphabetic names for operators

<wchar.h>

Manipulation of wide-character streams and strings

<wctype.h>

Classification of wide characters (upper/lower case, alphabetic/numeric *etc*)

C99 additionally provides the following headers:

<stdbool.h>

The boolean type and constants

<complex.h>

The complex types and constants

<inttypes.h>
<stdint.h>

Integer types of specific or minimum widths

<fenv.h>

Access to the floating-point environment

<tgmath.h>

Type-generic mathematics functions

Makefile tutorial

1. A Simple Makefile Tutorial 1

Makefiles are a simple way to organize code compilation. This tutorial does not even scratch the surface of what is possible using *make*, but is intended as a starters guide so that you can quickly and easily create your own makefiles for small to medium-sized projects.

- **A Simple Example**

Let's start off with the following three files, hellomake.c, hellofunc.c, and hellomake.h, which would represent a typical main program, some functional code in a separate file, and an include file, respectively.

hellomake.c	hellofunc.c	hellomake.h

#include
 	
int main() {
  // call a function in another file
  myPrintHelloMake();
 	
  return(0);
}
	#include
	#include
	
	void myPrintHelloMake(void) {
	
	  printf("Hello makefiles!\n");
	
	  return;
	}
		/*
		example include file
		*/
		
		void myPrintHelloMake(void);

Normally, you would compile this collection of code by executing the following command:

```
gcc -o hellomake hellomake.c hellofunc.c -I.
```

This compiles the two .c files and names the executable `hellomake`. The `-I.` is included so that `gcc` will look in the current directory (`.`) for the include file `hellomake.h`. Without a makefile, the typical approach to the test/modify/debug cycle is to use the up arrow in a terminal to go back to your last compile command so you don't have to type it each time, especially once you've added a few more .c files to the mix.

Unfortunately, this approach to compilation has two downfalls. First, if you lose the compile command or switch computers you have to retype it from scratch, which is inefficient at best. Second, if you are only making changes to one .c file, recompiling all of them every time is also time-consuming and inefficient. So, it's time to see what we can do with a makefile.

The simplest makefile you could create would look something like:

Makefile 1

```
hellomake: hellomake.c hellofunc.c
    gcc -o hellomake hellomake.c hellofunc.c -I.
```

If you put this rule into a file called Makefile or makefile and then type make on the command line it will execute the compile command as you have written it in the makefile. Note that make with no arguments executes the first rule in the file. Furthermore, by putting the list of files on which the command depends on the first line after the :, make knows that the rule hellomake needs to be executed if any of those files change. Immediately, you have solved problem #1 and can avoid using the up arrow repeatedly, looking for your last compile command. However, the system is still not being efficient in terms of compiling only the latest changes.

One very important thing to note is that there is a tab before the gcc command in the makefile. There must be a tab at the beginning of any command, and make will not be happy if it's not there.

In order to be a bit more efficient, let's try the following:

Makefile 2

```
CC=gcc
CFLAGS=-I.

hellomake: hellomake.o hellofunc.o
    $(CC) -o hellomake hellomake.o hellofunc.o -I.
```

So now we've defined some constants CC and CFLAGS. It turns out these are special constants that communicate to make how we want to compile the files hellomake.c and hellofunc.c. In particular, the macro CC is the C compiler to use, and CFLAGS is the list of flags to pass to the compilation command. By putting the object files--hellomake.o and hellofunc.o--in the dependency list and in the rule, make knows it must first compile the .c versions individually, and then build the executable hellomake.

Using this form of makefile is sufficient for most small scale projects. However, there is one thing missing: dependency on the include files. If you were to make a change to hellomake.h, for example, make would not recompile the .c files, even though they needed to be. In order to fix this, we need to tell make that all .c files depend on certain .h files. We can do this by writing a simple rule and adding it to the makefile.

Makefile 3

```
CC=gcc
CFLAGS=-I.
DEPS = hellomake.h

%.o: %.c $(DEPS)
```

```
	$(CC) -c -o $@ $< $(CFLAGS)

hellomake: hellomake.o hellofunc.o
	gcc -o hellomake hellomake.o hellofunc.o -I.
```

This addition first creates the macro DEPS, which is the set of .h files on which the .c files depend. Then we define a rule that applies to all files ending in the .o suffix. The rule says that the .o file depends upon the .c version of the file and the .h files included in the DEPS macro. The rule then says that to generate the .o file, make needs to compile the .c file using the compiler defined in the CC macro. The -c flag says to generate the object file, the -o $@ says to put the output of the compilation in the file named on the left side of the :, the $< is the first item in the dependencies list, and the CFLAGS macro is defined as above.

As a final simplification, let's use the special macros $@ and $^, which are the left and right sides of the :, respectively, to make the overall compilation rule more general. In the example below, all of the include files should be listed as part of the macro DEPS, and all of the object files should be listed as part of the macro OBJ.

Makefile 4

```
CC=gcc
CFLAGS=-I.
DEPS = hellomake.h
OBJ = hellomake.o hellofunc.o

%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)
	gcc -o $@ $^ $(CFLAGS)
```

So what if we want to start putting our .h files in an include directory, our source code in a src directory, and some local libraries in a lib directory? Also, can we somehow hide those annoying .o files that hang around all over the place? The answer, of course, is yes. The following makefile defines paths to the include and lib directories, and places the object files in an obj subdirectory within the src directory. It also has a macro defined for any libraries you want to include, such as the math library -lm. This makefile should be located in the src directory. Note that it also includes a rule for cleaning up your source and object directories if you type make clean. The .PHONY rule keeps make from doing something with a file named clean.

Makefile 5

```
IDIR =../include
CC=gcc
CFLAGS=-I$(IDIR)

ODIR=obj
LDIR =../lib

LIBS=-lm

_DEPS = hellomake.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))

_OBJ = hellomake.o hellofunc.o
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))


$(ODIR)/%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)
	gcc -o $@ $^ $(CFLAGS) $(LIBS)

.PHONY: clean

clean:
	rm -f $(ODIR)/*.o *~ core $(INCDIR)/*~
```

So now you have a perfectly good makefile that you can modify to manage small and medium-sized software projects. You can add multiple rules to a makefile; you can even create rules that call other rules. For more information on makefiles and the make function, check out the GNU Make Manual, which will tell you more than you ever wanted to know (really).

2. A Simple Makefile Tutorial 2

make is a UNIX program that helps programmers efficiently build projects. The goal of each project is usually one or more executable programs. Each executable program is called a *target*. Thus, to create and executable program called pizza, a user would type

```
make pizza
```

in a UNIX shell. If all of the object code exists, and is up-to-date, the **make** program would invoke the appropriate linker, which would then produce the executable. However, if the object code is out-of-date,

or does not even exist, the **make** program will invoke the appropriate compiler (making up-to-date object code from the project's source code), followed by a call to the linker.

Targets can also refer to other file operations. For example, it is common practice to invoke

```
make clean
```

to delete object files, executable programs, and core dumps from a project directory. (Usually you want to do this before you distribute (or submit) a project.) Similarly, one might use

```
make all
```

to compile every library and link every executable within a complex project. More formal projects might use

```
make install
```

to copy (or move) every executable, compiled library, and header file to an appropriate subdirectory in the file system. (The **make** program should also compile and link as needed.

Users can interact with the make program with a number of command line arguments. For example,

```
make -d pizza
```

will print lots of debugging information as it tries to make the pizza program. More information about these arguments can obtained from the make man page: Just type

```
man make
```

Unfortunately, the **make** program is not very intelligent, and thus needs instructions from the developer on how each target is created. By default, **make** will first look for a special file in the current directory called either makefile or Makefile. A *makefile* is a description file that details how each target should be processed. Let's look at a sample makefile for my pizza project, which defines five targets:

```
# Robert's pizza project.
# (Characters from the first "#" to the end of the line are ignored.)

pizza.o: pizza.c pizza.h                    # Line  4
   gcc -c pizza.c                           # Line  5

pizza: pizza.o                              # Line  7
   gcc -o pizza pizza.o                     # Line  8

all: pizza                                  # Line 10
```

```
clean:                              # Line 12
    rm pizza pizza.o                    # Line 13

install: pizza                      # Line 15
    mv pizza /usr/local/bin             # Line 16
    cp pizza.h /usr/local/include       # Line 17
    echo "Your pizza is ready!"         # Line 18
```

Note how simple the syntax is. The name of each target begins a new line, and is followed by a colon. After the colon is a list of file names that the target depends on. (These are often called *prerequisites*. *It is important to emphasize that each continuation line, that is lines 5, 8, 13, 16, 17, and 18, begin with a* TAB *character*.

Suppose our directory contained only the following files

```
makefile    pizza.c     pizza.h
```

After typing make all, the **make** program would perform the following recursive tasks:

First **make** will try to create the file named pizza, because pizza is a prerequisite to the target all on line 10.

In order to create pizza, **make** will parse line 7. Since pizza.o is a prerequisite to pizza, **make** will needs to create pizza.o before it can create pizza.

The target pizza.o has two prerequisites: pizza.c and pizza.h. (If these did not exist, **make** would fail, as neither of these file names is specified as a target in the makefile.) Since both of these files exist in the current directory, line 5 is executed. Assuming the syntax of the source files is correct (and **gcc** exists), **gcc** will create the object file pizza.o.

Returning to Line 7, **make** can proceed with the task of creating file pizza. Line 8 is executed, and **gcc** will link pizza.o to create our pizza program.

Returning to Line 10, **make** will now attempt to execute line 11. However, this line is blank, so make will exit.

- **Makefile macros**

The **make** program supports many other features that allow one to create more versatile description files. The most useful of these are macros. A macro is symbol that can be defined within a makefile to represent

a list of other symbols. For example, we might wish to create a symbol CC that exapnds to "gcc." We can accomplish this with the single line

```
CC = gcc
```

We could then write our pizza makefile as

```
# Robert's pizza project.
# (Characters from the first "#" to the end of the line are ignored.)

CC = gcc                                # Line  4

pizza.o: pizza.c pizza.h                # Line  6
   ${CC} -c pizza.c                     # Line  7

pizza: pizza.o                          # Line  9
   ${CC} -o pizza pizza.o               # Line 10

all: pizza                              # Line 12

clean:                                  # Line 14
   rm pizza pizza.o                     # Line 15

install: pizza                          # Line 17
   mv pizza /usr/local/bin              # Line 18
   cp pizza.h /usr/local/include        # Line 19
   echo "Your pizza is ready!"          # Line 20
```

The macro CC is expanded by typing ${CC}, as shown on lines 7 and 10. (One can use parentheses instead of brackets). If you wanted to use a different compiler for your project, you would now only have to edit line 4.

Once a macro has been defined, you can override its definition in the command line. For example, if we want to build our project with the compiler /usr/bin/cc, we could invoke the command

```
make all "CC = /usr/bin/cc"
```

- **Suffix Rules**

Large projects use many source files, hence they require many object (.o) files for the build. It is cumbersome to write a pair of lines like lines 6 and 7 above, for each individual object file. To alleviate

this chore, the **make** program will assume a set of standard rules, called *suffix rules*, that describe how each object file should be generated from the corresponding source. First, it's important to specify which suffixes are important to your project. These are stored in the .SUFFIXES macro, which should be placed in the description file. For example,

```
.SUFFIXES: .o .c
```

The default rule for generating *anyfile*.o from *anyfile*.c is written as

```
.c.o:
    $(CC) $(CFLAGS) -c $<
```

The first line specifies that a filename with a .c extension is a prerequisite to the corresponding filename with a .o extension. (This is syntactically different from before: here, no characters should appear after the colon.) In the second line, the macro $< evaluates to the prerequisite filename. Thus if we were to replace lines 6 and 7 by the above suffix rule, all would be well.

We also introduced the symbol CFLAGS which can be used to modify the manner in which the source is compiled. For example, if you wanted to use an interactive source debugger, such as gdb, you might invoke **make** by

```
touch *.c
make all CLFAGS=-g
```

If instead we wanted to optimize the program, we would enter

```
touch *.c
make all CLFAGS=-O
```

(The **touch** command sets the modification time of each listed filename to the current instant. Why is it necessary here?)

Most implementations of **make** have built-in, or default, suffix rules. If you want to know what they are, type

```
make -p -f/dev/null
```

- **Suffix Replacement Macros**

Some implementations of **make** support the following string substitution macro. Suppose the macro SOURCE was defined to expand to a list of file names, e.g.,

```
SOURCE = pizza.c pepperoni.c cheese.c mushrooms.c anchovies.c olives.c
```

Then the statement

```
OBJECTS = $(SOURCE:.c=.0)
```

would be equivalent to

```
OBJECTS = pizza.o pepperoni.o cheese.o mushrooms.o anchovies.o olives.o
```

Thus, every occurrence of the string ".c" in SOURCE, is replaced by the string ".o".

- **Putting things all together**

Here's a more sophisticated version of our original makefile

```
# Robert's pizza project.                    # Line 1
SOURCE  = pizza.c pepperoni.c cheese.c\
        mushrooms.c anchovies.c olives.c      # Line 2
HEADERS = pizza.h topping.h                   # Line 3
OBJECTS = ${SOURCE:.c=.o}                     # Line 4

.PREFIXES = .c .o                             # Line 6
CC       = gcc
CFLAGS   = -O -I${HOME}/include               # Line 8
RM       = /usr/bin/rm -f                     # Line 9
MV       = /usr/bin/mv -f                     # Line 10
CP       = /usr/bin/cp -f                     # Line 11

.c.o:                                         # Line 13
   ${CC} -c ${CFLAGS} $<                      # Line 14

pizza: ${OBJECTS}                             # Line 16
   ${CC} -o $@ ${OBJECTS} -lm                 # Line 17

all: pizza                                    # Line 19

clean:                                        # Line 21
   -${RM} pizza *.o core                      # Line 22

install: pizza                                # Line 24
   ${MV} pizza /usr/local/bin                 # Line 25
   ${CP} ${HEADERS} /usr/local/include        # Line 26
   @echo "Your pizza is ready!"               # Line 27
```

- **Internal macro symbols**

$? The list of prerequisites that is younger than the target. (Cannot be used in a suffix rule.)

$@ Name of the current target. (If the target is a library, this expands to the library name.)

$$@ Name of the current target. (Can only be used in a prerequisite list.)

$< The prerequisite file name in a suffix rule.

$* In a suffix rule, this expands to the root name of the file.

$% Expands to a .o file if the target is a library.