Learning C from Java[±]

Java is a soft cushion at the bottom of the sea;
C is a stony floor at the bottom of a cliff.

Differences

This is a list of differences between Java and C, and assumes that the reader knows less about the latter. It's worth familiarising yourself with all the points, even if you don't fully understand them, then you may be aware of the cause of any problem you might encounter.

Many features of C, particularly its standard library facilities, are not dealt with in any great depth, since you can look those up in help files or manual pages, or a good reference book.

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, while avoiding or accounting for aspects of the standard which are implementation-defined.

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

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

---

[±] From Steven Simpson's web site: http://www.comp.lancs.ac.uk/~ss/java2c/

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 reference, or accessing beyond the bounds of an array).

Notwithstanding effective JIT optimization, C programs will usually run faster, however, since:

- dynamic memory management (which is often not required) is fully under the programmer's control, and
- there are no checks for illegal operations (but a well-written program won't attempt them anyway),

… although these require greater responsibility from the programmer.

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, an unsigned integral type at least as wide as `unsigned long`, typically (and always at least) 64 bits. The `int` is often omitted.

`signed long long int`
`signed long long`
`long long int`
`long long`

In C99, a signed integral type of the same size as `unsigned long long int`. This is often just called `long long`.

`float`

A single precision floating-point type.

`double`

A double precision floating-point type.

`long double`

An extended double precision floating-point type.

`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.

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.

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.

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` (NB: ‘`struct`’ is part of the 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 initialised where it is defined:

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

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

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

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

In C99, a structure initialisation 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 */
```

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 }
```

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` (NB: ‘`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, although it is possible for a reference to refer to any class derived from its own. A reference of type `java.lang.Object` can refer to any class of object, since all classes are originally derived from `java.lang.Object`.

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.

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 */
{
  /* ... */
}
```

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):

```
void (*signal(int signum, void (*handler)(int)))(int);
```

Erm, what? It becomes a little clearer when POSIX (an Operating System standard which incorporates the C standard) declares it:

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Now we can see that the function's second argument has the same type as its return value, and that that type is, in fact, a pointer-to-function 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.

Declarations and definitions

C programs are built from collections of functions (which have behaviour) 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 behaviour.

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, should that be encountered first.

```
/* 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

Global objects also have distinct declarative and definitive forms. A definition may be accompanied by an initialiser, *e.g.*

```
int globval = 34; /* initialised */
int another;      /* uninitialised */
```

while a declaration should not have an initialiser, 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

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 initialisation 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 */
}
```

Empty parameter 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, and is not recommended.

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, internationalisation, *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**.

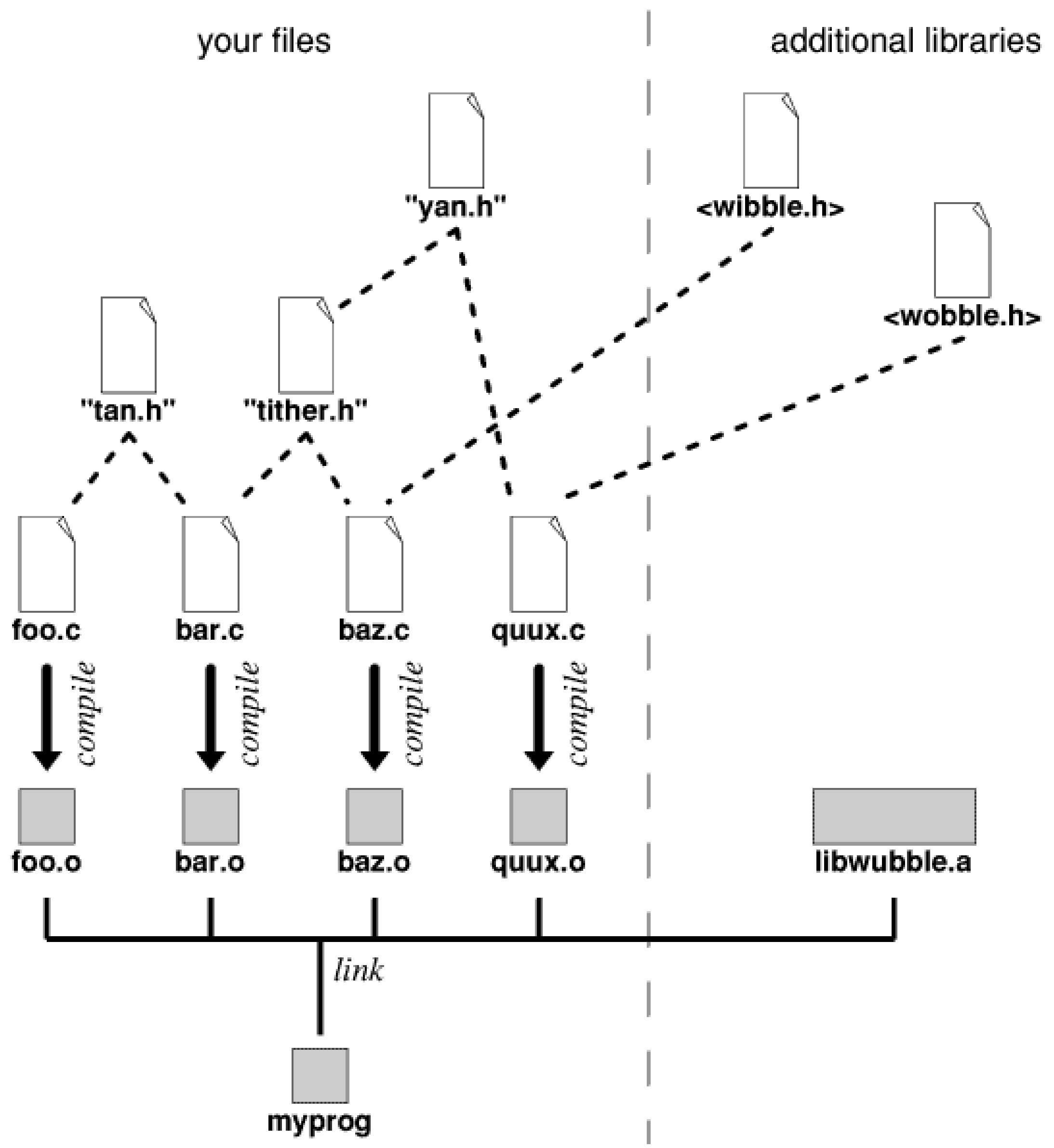
For example, the object file generated from the code below would contain references to the names `pow` (because it is invoked as a function) and `errno` (because it is accessed as a global variable), and would also provide a representation of the name `func` (because a definition of the function is provided).

```
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.

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, but the following purposes don't usually apply to it anyway.

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. (See a good book on C for the limitations of this.)

- 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 `#include`d more than once:

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

typedef int myInteger;

#endif
```

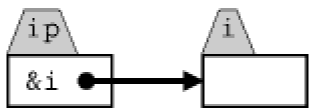
You should normally protect all your header files in this way.

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;
```

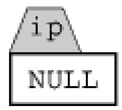
…then `i` is of type `int`, so the expression `&i` must be of type `int *`. `ip` is of type `int *`, so you can assign `&i` to it. `&j` is of type `int *`, so it can be compared with `&i`, and so on.

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;
```

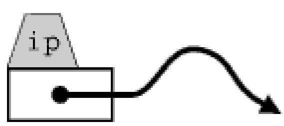


It is permissible to use pointers as integer expressions treated as boolean expressions to detect a null pointer (null means 'false' in this context). For example:

```
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 initialised 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:

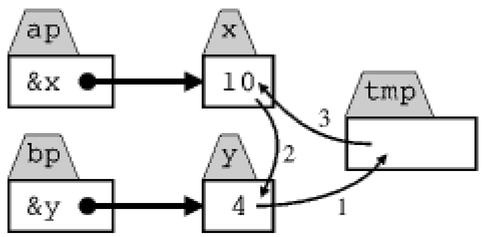
```
void badswap(int a, int b)
{
  int tmp = b;
  b = a;
  a = tmp;
  /* a and b are swapped but they're only copies */
}

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.

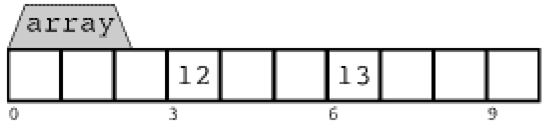
```
int *ip;
float *fp;
void *vp;

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.

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;
```



Initialising arrays

Arrays may be initialised 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 initialiser. The initialiser 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.

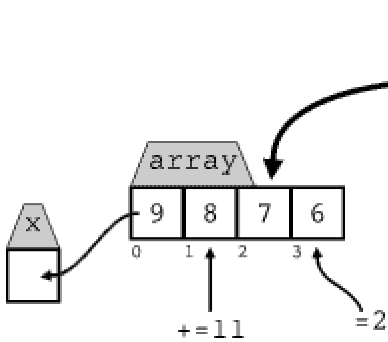
In C99, you can specify which elements of an array are initialised:

```
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` can not 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

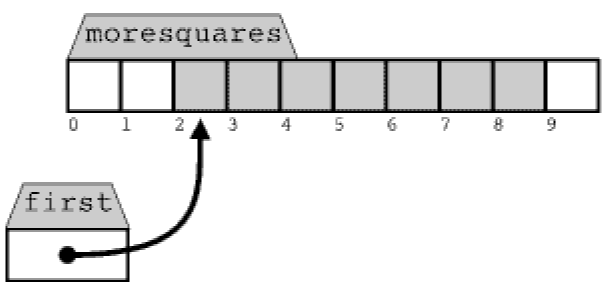
Arrays are effectively passed to functions by reference. The array name evaluates to a pointer to the first element, so the function's parameter has a type of 'pointer-to-element-type'. For example, given the function:

```
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 `char`s. `sizeof squares[0]` returns the size (in `char`s) of one element, and since all the elements are of the same size, the ratio of these two `sizeof`s is the number of elements in the array:

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

(For arrays of `char`s, 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).

`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 initialised, 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.

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 redclaring 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.

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 `char`s.

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 initialised like other arrays:

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

Note that the second initialiser 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 behaviour, 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.

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 initialised. 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.

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.**

`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.

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 summarised 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