



The C Programming Language

“C is a strongly typed,
weakly checked,
programming language.”

—Dennis Ritchie

This chapter is a brief overview of the C programming language and its standard library from the point of view of someone who knows C++. It lists the C++ features missing from C and gives examples of how a C programmer can cope without those. C/C++ incompatibilities are presented, and C/C++ interoperability is discussed. Examples of I/O, list manipulation, memory management, and string manipulation are included as illustration.

27.1 C and C++: siblings	27.4 Free store
27.1.1 C/C++ compatibility	27.5 C-style strings
27.1.2 C++ features missing from C	27.5.1 C-style strings and <code>const</code>
27.1.3 The C standard library	27.5.2 Byte operations
27.2 Functions	27.5.3 An example: <code>strcpy()</code>
27.2.1 No function name overloading	27.5.4 A style issue
27.2.2 Function argument type checking	27.6 Input/output: <code>stdio</code>
27.2.3 Function definitions	27.6.1 Output
27.2.4 Calling C from C++ and C++ from C	27.6.2 Input
27.2.5 Pointers to functions	27.6.3 Files
27.3 Minor language differences	27.7 Constants and macros
27.3.1 <code>struct</code> tag namespace	27.8 Macros
27.3.2 Keywords	27.8.1 Function-like macros
27.3.3 Definitions	27.8.2 Syntax macros
27.3.4 C-style casts	27.8.3 Conditional compilation
27.3.5 Conversion of <code>void*</code>	27.9 An example: intrusive containers
27.3.6 <code>enum</code>	
27.3.7 Namespaces	

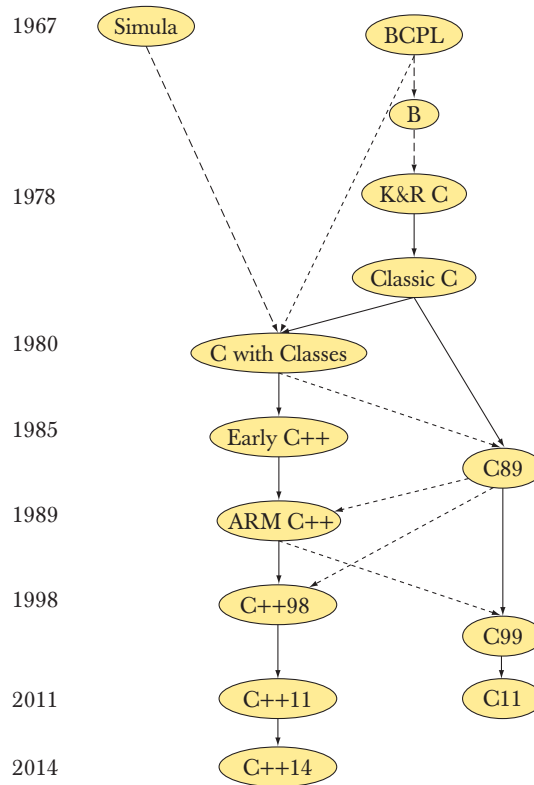
27.1 C and C++: siblings

The C programming language was designed and implemented by Dennis Ritchie at Bell Labs and popularized by the book *The C Programming Language* by Brian Kernighan and Dennis Ritchie (colloquially known as “K&R”), which is arguably still the best introduction to C and one of the great books on programming (§22.2.5). The text of the original definition of C++ was an edit of the text of the 1980 definition of C, supplied by Dennis Ritchie. After this initial branch, both languages evolved further. Like C++, C is now defined by an ISO standard.

We see C primarily as a subset of C++. Thus, from a C++ point of view, the problem of describing C boils down to two issues:

- Describe where C isn’t a subset of C++.
- Describe which C++ features are missing in C and which facilities and techniques can be used to compensate.

Historically, modern C and modern C++ are siblings. Both are direct descendants of “Classic C,” the dialect of C popularized by the first edition of Kernighan and Ritchie’s *The C Programming Language* plus structure assignment and enumerations:



The version of C that is used today is still mostly C89 (as described in the second edition of K&R), and that’s what we are describing here. There is still some Classic C in use and some C99, but that should not cause you any problems when you know C++ and C89.

Both C and C++ were “born” in the Computer Science Research Center of Bell Labs in Murray Hill, New Jersey (for a while, my office was a couple of doors down and across the corridor from those of Dennis Ritchie and Brian Kernighan):



Both languages are now defined/controlled by ISO standards committees. For each, many supported implementations are in use. Often, an implementation supports both languages with the desired language chosen by a compiler switch or a source file suffix. Both are available on more platforms than any other language. Both were primarily designed for and are now heavily used for hard system programming tasks, such as

- Operating system kernels
- Device drivers
- Embedded systems
- Compilers
- Communications systems

There are no performance differences between equivalent C and C++ programs.

Like C++, C is very widely used. Taken together, the C/C++ community is the largest software development community on earth.

27.1.1 C/C++ compatibility

It is not uncommon to hear references to “C/C++.” However, there is no such language, and the use of “C/C++” is typically a sign of ignorance. We use “C/C++” only in the context of C/C++ compatibility issues and when talking about the large shared C/C++ technical community.

C++ is largely, but not completely, a superset of C. With a few very rare exceptions, constructs that are both C and C++ have the same meaning (semantics) in both languages. C++ was designed to be “as close as possible to C, but no closer”:

- For ease of transition
- For coexistence

Most incompatibilities relate to C++’s stricter type checking.

An example of a program that is legal C but not C++ is one that uses a C++ keyword that is not a C keyword as an identifier (see §27.3.2):

```
int class(int new, int bool);    /* C, but not C++ */
```

Examples where the semantics differ for a construct that is legal in both languages are harder to find, but here is one:

```
int s = sizeof('a');           /* sizeof(int), often 4 in C and 1 in C++ */
```

The type of a character literal, such as `'a'`, is `int` in C and `char` in C++. However, for a `char` variable `ch` we have `sizeof(ch)==1` in both languages.

Information related to compatibility and language differences is not exactly exciting. There are no new neat programming techniques to learn. You might like `printf()` (§27.6), but with that possible exception (and some feeble attempts at geek humor), this chapter is bone dry. Its purpose is simple: to allow you to read and write C if you need to. This includes pointing out the hazards that are obvious to experienced C programmers, but typically unexpected by C++ programmers. We hope you can learn to avoid those hazards with minimal grief.

Most C++ programmers will have to deal with C code at some point or another, just as most C programmers will have to deal with C++ code. Much of what we describe in this chapter will be familiar to most C programmers, but some will be considered “expert level.” The reason for that is simple: not everyone agrees about what is “expert level” and we just describe what is common in real-world code. Maybe understanding compatibility issues can be a cheap way of gaining an unfair reputation as a “C expert.” But do remember: real expertise is in the use of a language (in this case C), rather than in understanding esoteric language rules (as are exposed by considering compatibility issues).

References

- ISO/IEC 9899:1999. *Programming Languages – C*. This defines C99; most implementations implement C89 (often with a few extensions).
- ISO/IEC 9899:2011. *Programming Languages – C*. This defines C11.
- ISO/IEC 14882:2011. *Programming Languages – C++*.
- Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. ISBN 0131103628.
- Stroustrup, Bjarne. “Learning Standard C++ as a New Language.” *C/C++ Users Journal*, May 1999.
- Stroustrup, Bjarne. “C and C++: Siblings”; “C and C++: A Case for Compatibility”; and “C and C++: Case Studies in Compatibility.” *The C/C++ Users Journal*, July, Aug., and Sept. 2002.

The papers by Stroustrup are most easily found on my publications home page.

27.1.2 C++ features missing from C

From a C++ perspective, C (i.e., C89) lacks a lot of features, such as

- Classes and member functions
 - Use `struct` and global functions.

- Derived classes and virtual functions
 - Use **structs**, global functions, and pointers to functions (§27.2.3).
- Templates and inline functions
 - Use macros (§27.8).
- Exceptions
 - Use error codes, error return values, etc.
- Function overloading
 - Give each function a distinct name.
- **new/delete**
 - Use **malloc()/free()** and separate initialization/cleanup code.
- References
 - Use pointers.
- **const**, **constexpr**, or functions in constant expressions
 - Use macros.
- **bool**
 - Use **int**.
- **static_cast**, **reinterpret_cast**, and **const_cast**
 - Use C-style casts, e.g., **(int)a** rather than **static_cast<int>(a)**.



Lots of useful code is written in C, so this list should remind us that no one language feature is absolutely necessary. Most language features – even most C language features – are there for the convenience (only) of the programmer. After all, given sufficient time, cleverness, and patience, every program can be written in assembler. Note that because C and C++ share a machine model that is very close to the real machine, they are well suited to emulate varieties of programming styles.

The rest of this chapter explains how to write useful programs without those features. Our basic advice for using C is:

- Emulate the programming techniques that the C++ features were designed to support with the facilities provided by C.
- When writing C, write in the C subset of C++.

- Use compiler warning levels that ensure function argument checking.
- Use lint for large programs (see §27.2.2).

Many of the details of C/C++ incompatibilities are rather obscure and technical. However, to read and write C, you don't actually have to remember most of those:

- The compiler will remind you when you are using a C++ feature that is not in C.
- If you follow the rules above, you are unlikely to encounter anything that means something different in C from what it means in C++.

With the absence of all those C++ facilities, some facilities gain importance in C:

- Arrays and pointers
- Macros
- **typedef** (the C and C++98 equivalent to simple using declarations; see §20.5, §A.16)
- **sizeof**
- Casts

We give examples of a few such uses in this chapter.

I introduced the `//` comments into C++ from C's ancestor BCPL when I got really fed up with typing `/* . . . */` comments. The `//` comments are accepted by most C dialects including C99 and C11, so it is probably safe just to use them. Here, we will use `/* . . . */` exclusively in examples meant to be C. C99 and C11 introduced a few more C++ features (as well as a few features that are incompatible with C++), but here we will stick to C89, because that's far more widely used.

27.1.3 The C standard library

Naturally, a C++ library facility that depends on classes and templates is not available in C. This includes

- **vector**
- **map**
- **set**
- **string**
- The STL algorithms: e.g., **sort()**, **find()**, and **copy()**
- **iostreams**
- **regex**

For these, there are often C libraries based on arrays, pointers, and functions to help compensate. The main parts of the C standard library are

- `<stdlib.h>`: general utilities (e.g., `malloc()` and `free()`); see §27.4)
- `<stdio.h>`: standard I/O; see §27.6
- `<string.h>`: C-style string manipulation and memory manipulation; see §27.5
- `<math.h>`: standard floating-point mathematical functions; see §24.8
- `<errno.h>`: error codes for `<math.h>`; see §24.8
- `<limits.h>`: sizes of integer types; see §24.2
- `<time.h>`: date and time; see §26.6.1
- `<assert.h>`: debug assertions; see §27.9
- `<ctype.h>`: character classification; see §11.6
- `<stdbool.h>`: Boolean macros

For a complete description, see a good C textbook, such as K&R. All of these libraries (and header files) are also available in C++.

27.2 Functions

In C:

- There can be only one function of a given name.
- Function argument type checking is optional.
- There are no references (and therefore no pass-by-reference).
- There are no member functions.
- There are no inline functions (except in C99).
- There is an alternative function definition syntax.

Apart from that, things are much as you are used to in C++. Let us explore what that means.

27.2.1 No function name overloading

Consider:

```
void print(int);           /* print an int */
void print(const char*);  /* print a string */ /* error! */
```

The second declaration is an error because there cannot be two functions with the same name. So you'll have to invent a suitable pair of names:


```
void print_int(int);           /* print an int */
void print_string(const char*); /* print a string */
```

This is occasionally claimed to be a virtue: now you can't accidentally use the wrong function to print an `int`! Clearly we don't buy that argument, and the lack of overloaded functions does make generic programming ideas awkward to implement because generic programming depends on semantically similar functions having the same name.

27.2.2 Function argument type checking

Consider:

```
int main()
{
    f(2);
}
```

A C compiler will accept this: you don't have to declare a function before you call it (though you can and should). There may be a definition of `f()` somewhere. That `f()` could be in another translation unit, but if it isn't, the linker will complain.

Unfortunately, that definition in another source file might look like this:

```
/* other_file.c: */

int f(char* p)
{
    int r = 0;
    while (*p++) r++;
    return r;
}
```

The linker will not report that error. You will get a run-time error or some random result.

How do we manage problems like that? Consistent use of header files is a practical answer. If every function you call or define is declared in a header that is consistently **#included** whenever needed, we get checking. However, in large programs that can be hard to achieve. Consequently, most C compilers have options that give warnings for calls of undeclared functions: use them. Also, from the earliest days of C, there have been programs that can be used to check for all kinds of consistency problems. They are usually called *lint*. Use a lint for every nontrivial C program. You will find that lint pushes you toward a style of C usage that is rather similar to using a subset of C++. One of the observations that led

to the design of C++ was that the compiler could easily check much (but not all) of what lint checked.

You can ask to have function arguments checked in C. You do that simply by declaring a function with its argument types specified (just as in C++). Such a declaration is called a *function prototype*. However, beware of function declarations that do not specify arguments; those are *not* function prototypes and do not imply function argument checking:

```

int g(double);      /* prototype — like C++ function declaration */
int h();           /* not a prototype — the argument types are unspecified */

void my_fct()
{
    g();            /* error: missing argument */
    g("asdf");    /* error: bad argument type */
    g(2);         /* OK: 2 is converted to 2.0 */
    g(2,3);       /* error: one argument too many */

    h();          /* OK by the compiler! May give unexpected results */
    h("asdf");   /* OK by the compiler! May give unexpected results */
    h(2);       /* OK by the compiler! May give unexpected results */
    h(2,3);     /* OK by the compiler! May give unexpected results */
}

```



The declaration of **h()** specifies no argument type. This does not mean that **h()** doesn't accept arguments; it means "Accept any set of arguments and hope they are correct for the called function." Again, a good compiler warns and lint will catch the problem.

C++	C equivalent
void f(); // preferred	void f(void);
void f(void);	void f(void);
void f(. . .); // accept any arguments	void f(); /* accept any arguments */

There is a special set of rules for converting arguments where no function prototype is in scope. For example, **chars** and **shorts** are converted to **ints**, and **floats** are converted to **doubles**. If you need to know, say, what happens to a **long**, look it up in a good C textbook. Our recommendation is simple: don't call functions without prototypes.

Note that even though the compiler will allow an argument of the wrong type to be passed, such as a `char*` to a parameter of type `int`, the use of such an argument of a wrong type is an error. As Dennis Ritchie said, “C is a strongly typed, weakly checked, programming language.”

27.2.3 Function definitions

You can define functions exactly as in C++ and such definitions are function prototypes:

```
double square(double d)
{
    return d*d;
}

void ff()
{
    double x = square(2);           /* OK: convert 2 to 2.0 and call */
    double y = square();           /* argument missing */
    double y = square("Hello");   /* error: wrong argument type */
    double y = square(2,3);       /* error: too many arguments */
}
```

A definition of a function with no arguments is not a function prototype:

```
void f() { /* do something */ }


void g()
{
    f(2);    /* OK in C; error in C++ */
}
```

Having

```
void f();    /* no argument type specified */
```


mean “`f()` can take any number of arguments of any type” seemed really strange. In response, I invented a new notation where “nothing” was explicitly stated using the keyword `void` (*void* is a four-letter word meaning “nothing”):

```
void f(void); /* no arguments accepted */
```

 I soon regretted that, though, since that looks odd and is completely redundant when argument type checking is uniformly applied. Worse, Dennis Ritchie (the father of C) and Doug McIlroy (the ultimate arbiter of taste in the Bell Labs Computer Science Research Center; see §22.2.5) both called it “an abomination.” Unfortunately, that abomination became very popular in the C community. Don’t use it in C++, though, where it is not only ugly, but also logically redundant.

C also provides a second, Algol60-style function definition, where the parameter types are (optionally) specified separately from their names:

```
int old_style(p,b,x) char* p; char b;
{
    /* ... */
}
```

 This “old-style definition” predates C++ and is not a prototype. By default, an argument without a declared type is an **int**. So, **x** is an **int** parameter of **old_style()**. We can call **old_style()** like this:

```
old_style();           /* OK: all arguments missing */
old_style("hello", 'a', 17); /* OK: all arguments are of the right type */
old_style(12, 13, 14);  /* OK: 12 is the wrong type, */
                       /* but maybe old_style() won't use p */
```

The compiler should accept these calls (but would warn, we hope, for the first and third).


Our recommendation about function argument checking:

- Use function prototypes consistently (use header files).
- Set compiler warning levels so that argument type errors are caught.
- Use (some) lint.

The result will be code that’s also C++.

27.2.4 Calling C from C++ and C++ from C

You can link files compiled with a C compiler together with files compiled with a C++ compiler provided the two compilers were designed for that. For example, you can link object files generated from C and C++ using your GNU C and C++ compiler (GCC) together. You can also link object files generated from C and C++ using your Microsoft C and C++ compiler (MSC++) together. This is common and useful because it allows you to use a larger set of libraries than would be available in just one of those two languages.

 C++ provides stricter type checking than C. In particular, a C++ compiler and linker check that two functions **f(int)** and **f(double)** are consistently defined

and used – even in different source files. A linker for C doesn't do that kind of checking. To call a function defined in C from C++ and to have a function defined in C++ called from C, we need to tell the compiler what we are doing:

```
// calling C function from C++:

extern "C" double sqrt(double);    // link as a C function

void my_c_plus_plus_fct()
{
    double sr = sqrt(2);
}
```

Basically **extern "C"** tells the compiler to use C linker conventions. Apart from that, all is normal from a C++ point of view. In fact, the C++ standard **sqrt(double)** usually is the C standard library **sqrt(double)**. Nothing is required from the C program to make a function callable from C++ in this way. C++ simply adapts to the C linkage convention.

We can also use **extern "C"** to make a C++ function callable from C:

```
// C++ function callable from C:

extern "C" int call_f(S* p, int i)
{
    return p->f(i);
}
```

In a C program, we can now call the member function **f()** indirectly, like this:

```
/* call C++ function from C: */

int call_f(S* p, int i);
struct S* make_S(int, const char*);

void my_c_fct(int i)
{
    /* ... */
    struct S* p = make_S(x, "foo");
    int x = call_f(p, i);
    /* ... */
}
```

No mention of C++ is needed (or possible) in C for this to work.

The benefit of this interoperability is obvious: code can be written in a mix of C and C++. In particular, a C++ program can use libraries written in C, and C programs can use libraries written in C++. Furthermore, most languages (notably Fortran) have an interface for calling to/from C.

In the examples above, we assumed that C and C++ could share the class object pointed to by **p**. That is true for most class objects. In particular, if you have a class like this,

```
// in C++:
class complex {
    double re, im;
public:
    // all the usual operations
};
```

you can get away with passing a pointer to an object to and from C. You can even access **re** and **im** in a C program using a declaration:

```
/* in C: */
struct complex {
    double re, im;
    /* no operations */
};
```



The rules for layout in any language can be complex, and the rules for layout among languages can even be hard to specify. However, you can pass built-in types between C and C++ and also classes (**structs**) without virtual functions. If a class has virtual functions, you should just pass pointers to its objects and leave the actual manipulation to C++ code. The **call_f()** was an example of this: **f()** might be **virtual** and then that example would illustrate how to call a virtual function from C.

Apart from sticking to the built-in types, the simplest and safest sharing of types is a **struct** defined in a common C/C++ header file. However, that strategy seriously limits how C++ can be used, so we don't restrict ourselves to it.

27.2.5 Pointers to functions

What can we do in C if we want to use object-oriented techniques (§14.2–4)? Basically, we need an alternative to virtual functions. For most people, the first idea that springs to mind is to use a **struct** with a “type field” that describes what kind of shape a given object represents. For example:

```

struct Shape1 {
    enum Kind { circle, rectangle } kind;
    /* ... */
};

void draw(struct Shape1* p)
{
    switch (p->kind) {
    case circle:
        /* draw as circle */
        break;
    case rectangle:
        /* draw as rectangle */
        break;
    }
}

int f(struct Shape1* pp)
{
    draw(pp);
    /* ... */
}

```

This works. There are two snags, though:

- For each “pseudo-virtual” function (such as **draw()**), we have to write a new **switch**-statement.
- Each time we add a new shape, we have to modify every “pseudo-virtual” function (such as **draw()**) by adding a case to the **switch**-statement.

The second problem is quite nasty because it means that we can’t provide our “pseudo-virtual” functions as part of a library, because our users will have to modify those functions quite often. The most effective alternative involves pointers to functions:

```

typedef void (*Pfct0)(struct Shape2*);
typedef void (*Pfct1int)(struct Shape2*,int);

struct Shape2 {
    Pfct0 draw;
    Pfct1int rotate;
    /* ... */
};

```

```

void draw(struct Shape2* p)
{
    (p->draw)(p);
}

void rotate(struct Shape2* p, int d)
{
    (p->rotate)(p,d);
}

```

This **Shape2** can be used just like **Shape1**.

```

int f(struct Shape2* pp)
{
    draw(pp);
    /* ... */
}

```

With a little extra work, an object need not hold one pointer to a function for each pseudo-virtual function. Instead, it can hold a pointer to an array of pointers to functions (much as virtual functions are implemented in C++). The main problem with using such schemes in real-world programs is to get the initialization of all those pointers to functions right.

27.3 Minor language differences

This section gives examples of minor C/C++ differences that could trip you up if you have never heard of them. Few seriously impact programming in that the differences have obvious work-arounds.

27.3.1 struct tag namespace

In C, the names of **structs** (there is no **class** keyword) are in a separate namespace from other identifiers. Therefore, every name of a **struct** (called a *structure tag*) must be prefixed with the keyword **struct**. For example:

```

struct pair { int x,y; };
pair p1;          /* error: no identifier pair in scope */
struct pair p2;  /* OK */
int pair = 7;    /* OK: the struct tag pair is not in scope */
struct pair p3;  /* OK: the struct tag pair is not hidden by the int */
pair = 8;        /* OK: pair refers to the int */

```


Amazingly enough, thanks to a devious compatibility hack, this also works in C++. Having a variable (or a function) with the same name as a **struct** is a fairly common C idiom, though not one we recommend.

If you don't want to write **struct** in front of every structure name, use a **typedef** (§20.5). The following idiom is common:

```
typedef struct { int x,y; } pair;
pair p1 = { 1, 2 };
```

In general, you'll find **typedefs** more common and more useful in C, where you don't have the option of defining new types with associated operations.

In C, names of nested **structs** are placed in the same scope as the **struct** in which they are nested. For example:

```
struct S {
    struct T { /* ... */};
    /* ... */
};

struct T x; /* OK in C (not in C++) */
```

In C++, you would write

```
S::T x; // OK in C++ (not in C)
```


Whenever possible, don't nest **structs** in C: their scope rules differ from what most people naively (and reasonably) expect.

27.3.2 Keywords

Many keywords in C++ are not keywords in C (because C doesn't provide the functionality) and can be used as identifiers in C:

C++ keywords that are not C keywords				
alignas	class	inline	private	true
alignof	compl	mutable	protected	try
and	concept	namespace	public	typeid
and_eq	const_cast	new	reinterpret_cast	typename
asm	constexpr	noexcept	requires	using
bitand	delete	not	static_assert	virtual

C++ keywords that are not C keywords (<i>continued</i>)				
bitor	dynamic_cast	not_eq	static_cast	wchar_t
bool	explicit	nullptr	template	xor
catch	export	operator	this	xor_eq
char16_t	false	or	thread_local	
char32_t	friend	or_eq	throw	

 Don't use these names as identifiers in C, or your code will not be portable to C++. If you use one of these names in a header file, that header won't be useful from C++.

Some C++ keywords are macros in C:

C++ keywords that are C macros				
and	bitor	false	or	wchar_t
and_eq	bool	not	or_eq	xor
bitand	compl	not_eq	true	xor_eq

In C, they are defined in `<iso646.h>` and `<stdbool.h>` (**bool**, **true**, **false**). Don't take advantage of the fact that they are macros in C.

27.3.3 Definitions

C++ allows definitions in more places than C89. For example:

```

for (int i = 0; i<max; ++i) x[i] = y[i];    // definition of i not allowed in C

while (struct S* p = next(q)) {           // definition of p not allowed in C
    /* ... */
}

void f(int i)
{
    if (i< 0 || max<=i) error("range error");
    int a[max];    // error: declaration after statement not allowed in C
    /* ... */
}

```

C (C89) doesn't allow declarations as initializers in **for**-statements, as conditions, or after a statement in a block. We have to write something like

```

int i;
for (i = 0; i<max; ++i) x[i] = y[i];

struct S* p;
while (p = next(q)) {
    /* ... */
}

void f(int i)
{
    if (i< 0 || max<=i) error("range error");
    {
        int a[max];
        /* ... */
    }
}

```

In C++, an uninitialized declaration is a definition; in C, it is just a declaration so that there can be two of them:

```

int x;
int x;      /* defines or declares a single integer called x in C; error in C++ */

```

In C++, an entity must be defined exactly once. This gets a bit more interesting if the two **ints** are in different translation units:

```

/* in file x.c: */
int x;

/* in file y.c: */
int x;

```

No C or C++ compiler will find any fault with either **x.c** or **y.c**. However, if **x.c** and **y.c** are compiled as C++, the linker will give a “double definition” error. If **x.c** and **y.c** are compiled as C, the linker accepts the program and (correctly according to C rules) considers there to be just one **x** that is shared between code in **x.c** and **y.c**. If you want a program where a global variable **x** is shared, say so explicitly:

```

/* in file x.c: */
int x = 0;      /* the definition */

/* in file y.c: */
extern int x;  /* a declaration, not a definition */

```

Better still, use a header file:

```

/* in file x.h: */
extern int x;           /* a declaration, not a definition */

/* in file x.c: */
#include "x.h"
int x = 0;             /* the definition */

/* in file y.c: */
#include "x.h"
/* the declaration of x is in the header */

```

Better still, avoid the global variable.

27.3.4 C-style casts

In C (and C++), you can explicitly convert a value **v** to a type **T** by this minimal notation:

(T)v



This “C-style cast” or “old-style cast” is beloved by poor typists and sloppy thinkers because it’s minimal and you don’t have to know what it takes to make a **T** from **v**. On the other hand, this style of cast is rightfully feared by maintenance programmers because it is just about invisible and leaves no clue about the writer’s intent. The C++ casts (*named casts* or *template-style casts*; see §A.5.7) were introduced to make explicit type conversion easy to spot (ugly) and specific. In C, you have no choice:

```

int* p = (int*)7;           /* reinterpret bit pattern: reinterpret_cast<int*>(7) */
int x = (int)7.5;          /* truncate double: static_cast<int>(7.5) */

typedef struct S1 { /* ... */ } S1;
typedef struct S2 { /* ... */ } S2;
S2 a;
const S2 b;                /* uninitialized consts are allowed in C */

S1* p = (S1*)&a;           /* reinterpret bit pattern: reinterpret_cast<S1*>(&a) */
S2* q = (S2*)&b;           /* cast away const: const_cast<S2*>(&b) */
S1* r = (S1*)&b;           /* remove const and change type; probably a bug */

```

We hesitate to recommend a macro (§27.8) even in C, but it may be an idea to express intent like this:

```
#define REINTERPRET_CAST(T,v) ((T)(v))
#define CONST_CAST(T,v) ((T)(v))

S1* p = REINTERPRET_CAST (S1*,&a);
S2* q = CONST_CAST(S2*,&b);
```

This does not give the type checking done by `reinterpret_cast` and `const_cast`, but it does make these inherently ugly operations visible and the programmer's intent explicit.

27.3.5 Conversion of `void*`

In C, a `void*` may be used as the right-hand operand of an assignment to or initialization of a variable of any pointer type; in C++ it may not. For example:

```
void* alloc(size_t x);           /* allocate x bytes */

void f (int n)
{
    int* p = alloc(n*sizeof(int)); /* OK in C; error in C++ */
    /* ... */
}
```

Here, the `void*` result of `alloc()` is implicitly converted to an `int*`. In C++, we would have to rewrite that line to

```
int* p = (int*)alloc(n*sizeof(int)); /* OK in C and C++ */
```

We used the C-style cast (§27.3.4) so that it would be legal in both C and C++.

Why is the `void*`-to-`T*` implicit conversion illegal in C++? Because such conversions can be unsafe:

```
void f()
{
    char i = 0;
    char j = 0;
    char* p = &i;
    void* q = p;
    int* pp = q;           /* unsafe; legal in C, error in C++ */
    *pp = -1;             /* overwrite memory starting at &i */
}
```

Here we can't even be sure what memory is overwritten. Maybe **j** and part of **p**? Maybe some memory used to manage the call of **f()** (**f**'s stack frame)? Whatever data is being overwritten here, a call of **f()** is bad news.

Note that (the opposite) conversion of a **T*** to a **void*** is perfectly safe – you can't construct nasty examples like the one above for that – and those are allowed in both C and C++.

Unfortunately, implicit **void***-to-**T*** conversions are common in C and possibly the major C/C++ compatibility problem in real code (see §27.4).

27.3.6 enum

In C, you can assign an **int** to an **enum** without a cast. For example:

```
enum color { red, blue, green };
int x = green;          /* OK in C and C++ */
enum color col = 7;    /* OK in C; error in C++ */
```

One implication of this is that we can use increment (**++**) and decrement (**--**) on variables of enumeration type in C. That can be convenient but does imply a hazard:

```
enum color x = blue;
++x;          /* x becomes green; error in C++ */
++x;          /* x becomes 3; error in C++ */
```

“Falling off the end” of the enumerators may or may not have been what we wanted.

Note that like structure tags, the names of enumerations are in their own namespace, so you have to prefix them with the keyword **enum** each time you use them:

```
color c2 = blue;          /* error in C: color not in scope; OK in C++ */
enum color c3 = red;     /* OK */
```

27.3.7 Namespaces

There are no namespaces (in the C++ sense of the word) in C. So what do you do when you want to avoid name clashes in large C programs? Typically, people use prefixes or suffixes. For example:

```
/* in bs.h: */
typedef struct bs_string { /* ... */ } bs_string; /* Bjarne's string */
typedef int bs_bool;      /* Bjarne's Boolean type */
```

```

/* in pete.h: */
typedef char* pete_string;      /* Pete's string */
typedef char pete_bool ;       /* Pete's Boolean type */

```

This technique is so popular that it is usually a bad idea to use one- or two-letter prefixes.

27.4 Free store

C does not provide the **new** and **delete** operators dealing with objects. To use the free store, you use functions dealing with memory. The most important functions are defined in the “general utilities” standard header **<stdlib.h>**:

```

void* malloc(size_t sz);      /* allocate sz bytes */
void free(void* p);          /* deallocate the memory pointed to by p */
void* calloc(size_t n, size_t sz); /* allocate n*sz bytes initialized to 0 */
void* realloc(void* p, size_t sz); /* reallocate the memory pointed to by p
to a space of size sz */

```

The **typedef size_t** is an unsigned type also defined in **<stdlib.h>**.

Why does **malloc()** return a **void***? Because **malloc()** has no idea which type of object you want to put in that memory. Initialization is your problem. For example:

```

struct Pair {
    const char* p;
    int val;
};

struct Pair p2 = {"apple", 78};
struct Pair* pp = (struct Pair*) malloc(sizeof(Pair));    /* allocate */
pp->p = "pear";      /* initialize */
pp->val = 42;

```

Note that we cannot write

```

*pp = {"pear", 42};      /* error: not C or C++98 */

```

in either C or C++. However, in C++, we would define a constructor for **Pair** and write

```

Pair* pp = new Pair("pear", 42);

```

In C (but not C++; see §27.3.4), you can leave out the cast before `malloc()`, but we don't recommend that:

```
int* p = malloc(sizeof(int)*n);    /* avoid this */
```

Leaving out the cast is quite popular because it saves some typing and because it catches the rare error of (illegally) forgetting to include `<stdlib.h>` before using `malloc()`. However, it can also remove a visual clue that a size was wrongly calculated:

```
p = malloc(sizeof(char)*m);    /* probably a bug — not room for m ints */
```



Don't use `malloc()/free()` in C++ programs; `new/delete` require no casts, deal with initialization (constructors) and cleanup (destructors), report memory allocation errors (through an exception), and are just as fast. Don't `delete` an object allocated by `malloc()` or `free()` an object allocated by `new`. For example:

```
int* p = new int[200];
// ...
free(p);    // error

X* q = (X*)malloc(n*sizeof(X));
// ...
delete q;    // error
```

This might work, but it is not portable code. Furthermore, for objects with constructors or destructors, mixing C-style and C++-style free-store management is a recipe for disaster.

The `realloc()` function is typically used for expanding buffers:

```
int max = 1000;
int count = 0;
int c;
char* p = (char*)malloc(max);
while ((c=getchar())!=EOF) {    /* read: ignore chars on eof line */
    if (count==max-1) {        /* need to expand buffer */
        max += max;          /* double the buffer size */
        p = (char*)realloc(p,max);
        if (p==0) quit();
    }
    p[count++] = c;
}
```


For an explanation of the C input operations, see §27.6.2 and §B.11.2.

The `realloc()` function may or may not move the old allocation into newly allocated memory. Don't even think of using `realloc()` on memory allocated by `new`.

Using the C++ standard library, the (roughly) equivalent code is

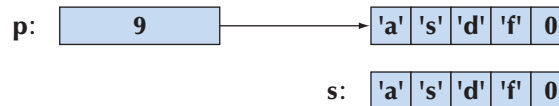
```
vector<char> buf;
char c;
while (cin.get(c)) buf.push_back(c);
```

Refer to the paper “Learning Standard C++ as a New Language” (see the reference list in §27.1) for a more thorough discussion of input and allocation strategies.

27.5 C-style strings

In C, a string (often called a *C string* or a *C-style string* in C++ literature) is a zero-terminated array of characters. For example:

```
char* p = "asdf";
char s[] = "asdf";
```



In C, we cannot have member functions, we cannot overload functions, and we cannot define an operator (such as `==`) for a `struct`. It follows that we need a set of (nonmember) functions to manipulate C-style strings. The C and C++ standard libraries provide such functions in `<string.h>`:

```
size_t strlen(const char* s);           /* count the characters */
char* strcat(char* s1, const char* s2); /* copy s2 onto the end of s1 */
int strcmp(const char* s1, const char* s2); /* compare lexicographically */
char* strcpy(char* s1, const char* s2); /* copy s2 into s1 */

char* strchr(const char* s, int c);     /* find c in s */
char* strstr(const char* s1, const char* s2); /* find s2 in s1 */

char* strncpy(char*, const char*, size_t n); /* strcpy, max n chars */
char* strncat(char*, const char, size_t n); /* strcat with max n chars */
int strncmp(const char*, const char*, size_t n); /* strcmp with max n chars */
```

This is not the full set, but these are the most useful and most used functions. We will briefly illustrate their use.



We can compare strings. The equality operator (`==`) compares pointer values; the standard library function `strcmp()` compares C-style string values:

```
const char* s1 = "asdf";
const char* s2 = "asdf";

if (s1==s2) { /* do s1 and s2 point to the same array? */
             /* (typically not what you want) */
}

if (strcmp(s1,s2)==0) { /* do s1 and s2 hold the same characters? */
}

```

The `strcmp()` function does a three-way comparison of its two arguments. Given the values of `s1` and `s2` above, `strcmp(s1,s2)` will return `0`, meaning a perfect match. If `s1` was lexicographically before `s2`, it would return a negative number, and if `s1` was lexicographically after `s2`, it would return a positive number. The term *lexicographical* means roughly “as in a dictionary.” For example:

```
strcmp("dog","dog")==0
strcmp("ape","dodo")<0 /* "ape" comes before "dodo" in a dictionary */
strcmp("pig","cow")>0 /* "pig" comes after "cow" in a dictionary */

```

The value of the pointer comparison `s1==s2` is not guaranteed to be `0` (**false**). An implementation may decide to use the same memory to hold all copies of a character literal, so we would get the answer `1` (**true**). Usually, `strcmp()` is the right choice for comparing C-style strings.

We can find the length of a C-style string using `strlen()`:

```
int lgt = strlen(s1);

```

Note that `strlen()` counts characters excluding the terminating `0`. In this case, `strlen(s1)==4` and it takes 5 bytes to store `"asdf"`. This little difference is the source of many off-by-one errors.

We can copy one C-style string (including the terminating `0`) into another:

```
strcpy(s1,s2); /* copy characters from s2 into s1 */

```

It is your job to be sure that the target string (array) has enough space to hold the characters from the source.

The `strncpy()`, `strncat()`, and `strncmp()` functions are versions of `strcpy()`, `strcat()`, and `strcmp()` that will consider a maximum of `n` characters, where `n` is their third argument. Note that if there are more than `n` characters in the source string, `strncpy()` will not copy a terminating 0, so that the result will not be a valid C-style string.

The `strchr()` and `strstr()` functions find their second argument in the string that is their first argument and return a pointer to the first character of the match. Like `find()`, they search from left to right in the string.

It is amazing both how much can be done with these simple functions and how easy it is to make minor mistakes. Consider a simple problem of concatenating a user name with an address, placing the @ character in between. Using `std::string` this can be done like this:

```
string s = id + '@' + addr;
```

Using the standard C-style string function we can write that as

```
char* cat(const char* id, const char* addr)
{
    int sz = strlen(id)+strlen(addr)+2;
    char* res = (char*) malloc(sz);
    strcpy(res,id);
    res[strlen(id)+1] = '@';
    strcpy(res+strlen(id)+2,addr);
    res[sz-1]=0;
    return res;
}
```

Did we get that right? Who will `free()` the string returned from `cat()`?

TRY THIS



Test `cat()`. Why 2? We left a beginner's performance error in `cat()`; find it and remove it. We "forgot" to comment our code. Add comments suitable for someone who can be assumed to know the standard C-string functions.

27.5.1 C-style strings and `const`

Consider:

```
char* p = "asdf";
p[2] = 'x';
```

This is legal in C but not in C++. In C++, a string literal is a constant, an immutable value, so `p[2]='x'` (to make the value pointed to "asdf") is illegal. Unfortunately, few compilers will catch the assignment to `p` that leads to the problem. If you are lucky, a run-time error will occur, but don't rely on that. Instead, write

```
const char* p = "asdf";    // now you can't write to "asdf" through p
```

This recommendation applies to both C and C++.

The C `strchr()` has a similar but even harder-to-spot problem. Consider:

```
char* strchr(const char* s, int c);    /* find c in constant s (not C++) */

const char aa[] = "asdf";             /* aa is an array of constants */
char* q = strchr(aa, 'd');             /* finds 'd' */
*q = 'x';                              /* change 'd' in aa to 'x' */
```

Again, this is illegal in C and C++, but C compilers can't catch it. Sometimes this is referred to as *transmutation*: it turns **consts** into non-**consts**, violating reasonable assumptions about code.

In C++, the problem is solved by the standard library declaring `strchr()` differently:

```
char const* strchr(const char* s, int c);    // find c in constant s
char* strchr(char* s, int c);                // find c in s
```

Similarly for `strstr()`.

27.5.2 Byte operations

In the distant dark ages (the early 1980s), before the invention of `void*`, C (and C++) programmers used the string operations to manipulate bytes. Now the basic memory manipulation standard library functions have `void*` parameters and return types to warn users about their direct manipulation of essentially untyped memory:

```
/* copy n bytes from s2 to s1 (like strcpy): */
void* memcpy(void* s1, const void* s2, size_t n);

/* copy n bytes from s2 to s1 ( [s1:s1+n) may overlap with [s2:s2+n) ): */
void* memmove(void* s1, const void* s2, size_t n);

/* compare n bytes from s2 to s1 (like strcmp): */
int memcmp(const void* s1, const void* s2, size_t n);
```

```

/* find c (converted to an unsigned char) in the first n bytes of s: */
void* memchr(const void* s, int c, size_t n);

/* copy c (converted to an unsigned char)
   into each of the first n bytes that s points to: */
void* memset(void* s, int c, size_t n);

```

Don't use these functions in C++. In particular, **memset()** typically interferes with the guarantees offered by constructors.

27.5.3 An example: **strcpy()**

The definition of **strcpy()** is both famous and infamous as an example of the terse style that C (and C++) is capable of:

```

char* strcpy(char* p, const char* q)
{
    while (*p++ = *q++);
    return p;
}


```

We leave to you the explanation of why this actually copies the C-style string **q** into **p**. Post-increment is described in §A.5: The value of **p++** is the value of **p** before increment.

TRY THIS



Is this implementation of **strcpy()** correct? Explain why.

If you can't explain why, we won't consider you a C programmer (however competent you are at programming in other languages). Every language has its own idioms, and this is one of C's. 

27.5.4 A style issue

We have quietly taken sides in a long-standing, often furiously debated, and largely irrelevant style issue. We declare a pointer like this:

```

char* p;           // p is a pointer to a char

```

and not like this:

```

char *p;          /* p is something that you can dereference to get a char */

```

The placement of the whitespace is completely irrelevant to the compiler, but programmers care. Our style (common in C++) emphasizes the type of the variable being declared, whereas the other style (more common in C) emphasizes the use of the variable. Note that we don't recommend declaring many variables in a single declaration:

```
char c, *p, a[177], *f();  /* legal, but confusing */
```

Such declarations are not uncommon in older code. Instead, use multiple lines and take advantage of the extra horizontal space for comments and initializers:

```
char c = 'a';           /* termination character for input using f() */
char* p = 0;          /* last char read by f() */
char a[177];          /* input buffer */
char* f();            /* read into buffer a; return pointer to first char read */
```

Also, choose meaningful names.

27.6 Input/output: stdio

There are no **iostreams** in C, so we use the C standard I/O defined in **<stdio.h>** and commonly referred to as **stdio**. The **stdio** equivalents to **cin** and **cout** are **stdin** and **stdout**. **Stdio** and **iostream** use can be mixed in a single program (for the same I/O streams), but we don't recommend that. If you feel the need to mix, read up on **stdio** and **iostreams** (especially **ios_base::sync_with_stdio()**) in an expert-level textbook. See also §B.11.

27.6.1 Output

The most popular and useful function of **stdio** is **printf()**. The most basic use of **printf()** just prints a (C-style) string:

```
#include<stdio.h>

void f(const char* p)
{
    printf("Hello, World!\n");
    printf(p);
}
```

That's not particularly interesting. The interesting bit is that `printf()` can take an arbitrary number of arguments, and the initial string controls if and how those extra arguments are printed. The declaration of `printf()` in C looks like this:

```
int printf(const char* format, . . . );
```

The `. . .` means “and optionally more arguments.” We can call `printf()` like this:

```
void f1(double d, char* s, int i, char ch)
{
    printf("double %g string %s int %d char %c\n", d, s, i, ch);
}
```

Here, `%g` means “Print a floating-point number using the general format,” `%s` means “Print a C-style string,” `%d` means “Print an integer using decimal digits,” and `%c` means “Print a character.” Each such format specifier picks the next so-far-unused argument, so `%g` prints `d`, `%s` prints `s`, `%d` prints `i`, and `%c` prints `ch`. You can find the full list of `printf()` formats in §B.11.2.

Unfortunately, `printf()` is not type safe. For example:

```
char a[] = { 'a', 'b' };    /* no terminating 0 */

void f2(char* s, int i)
{
    printf("goof %s\n", i);    /* uncaught error */
    printf("goof %d: %s\n", i); /* uncaught error */
    printf("goof %s\n", a);   /* uncaught error */
}
```

The effect of the last `printf()` is interesting: it prints every byte in memory following `a[1]` until it encounters a 0. That could be a lot of characters.

This lack of type safety is one reason we prefer `iostreams` over `stdio` even though `stdio` works identically in C and C++. The other reason is that the `stdio` functions are not extensible: you cannot extend `printf()` to print values of your own types, the way you can using `iostreams`. For example, there is no way you can define your own `%Y` to print some `struct Y`.

There is a useful version of `printf()` that takes a file descriptor as its first argument:

```
int fprintf(FILE* stream, const char* format, . . . );
```

For example:

```
fprintf(stdout,"Hello, World!\n"); // exactly like printf("Hello, World!\n");
FILE* ff = fopen("My_file","w"); // open My_file for writing
fprintf(ff,"Hello, World!\n"); // write "Hello, World!\n" to My_file
```

File handles are described in §27.6.3.

27.6.2 Input

The most popular stdio functions include

```
int scanf(const char* format, . . . ); /* read from stdin using a format */
int getchar(void); /* get a char from stdin */
int getc(FILE* stream); /* get a char from stream */
char* gets(char* s); /* get characters from stdin */
```

The simplest way of reading a string of characters is using **gets()**. For example:

```
char a[12];
gets(a); /* read into char array pointed to by a until a '\n' is input */
```



Never do that! Consider **gets()** poisoned. Together with its close cousin **scanf("%s")**, **gets()** used to be the root cause of about a quarter of all successful hacking attempts. It is still a major security problem. In the trivial example above, how would you know that at most 11 characters would be input before a newline? You can't know that. Thus, **gets()** almost certainly leads to memory corruption (of the bytes after the buffer), and memory corruption is a major tool of crackers. Don't think that you can guess a maximum buffer size that is "large enough for all uses." Maybe the "person" at the other end of the input stream is a program that does not meet your criteria for reasonableness.

The **scanf()** function reads using a format just as **printf()** writes using a format. Like **printf()** it can be very convenient:

```
void f()
{
    int i;
    char c;
    double d;
    char* s = (char*)malloc(100);
    /* read into variables passed as pointers: */
    scanf("%i %c %g %s", &i, &c, &d, s);
    /* %s skips initial whitespace and is terminated by whitespace */
}
```


Like `printf()`, `scanf()` is not type safe. The format characters and the arguments (all pointers) must match exactly, or strange things will happen at run time. Note also that the `%s` read into `s` may lead to an overflow. Don't ever use `gets()` or `scanf("%s")`!

So how do we read characters safely? We can use a form of `%s` that places a limit on the number of characters read. For example:

```
char buf[20];
scanf("%19s",buf);
```

We need space for a terminating 0 (supplied by `scanf()`), so 19 is the maximum number of characters we can read into `buf`. However, that leaves us with the problem of what to do if someone does type more than 19 characters. The “extra” characters will be left in the input stream to be “found” by later input operations.

The problem with `scanf()` implies that it is often prudent and easier to use `getchar()`. The typical way of reading characters with `getchar()` is

```
while((x=getchar())!=EOF) {
    /* ... */
}
```

`EOF` is a stdio macro meaning “end of file”; see also §27.4.

The C++ standard library alternative to `scanf("%s")` and `gets()` doesn't suffer from these problems:

```
string s;
cin >> s;      // read a word
getline(cin,s); // read a line
```

27.6.3 Files

In C (or C++), files can be opened using `fopen()` and closed using `fclose()`. These functions, together with the representation of a file handle, `FILE`, and the `EOF` (end-of-file) macro, are found in `<stdio.h>`:

```
FILE *fopen(const char* filename, const char* mode);
int fclose(FILE *stream);
```

Basically, you use files like this:

```
void f(const char* fn, const char* fn2)
{
    FILE* fi = fopen(fn, "r");      /* open fn for reading */
    FILE* fo = fopen(fn2, "w");    /* open fn2 for writing */
}
```

```

if (fi == 0) error("failed to open input file");
if (fo == 0) error("failed to open output file");

    /* read from file using stdio input functions, e.g., getc() */
    /* write to file using stdio output functions, e.g., fprintf() */

    fclose(fo);
    fclose(fi);
}

```

Consider this: there are no exceptions in C, so how do we make sure that the files are closed whichever error happens?

27.7 Constants and macros

In C, a **const** is never a compile-time constant:

```

const int max = 30;
const int x;      /* const not initialized: OK in C (error in C++) */

void f(int v)
{
    int a1[max];    /* error: array bound not a constant (OK in C++) */
                  /* (max is not allowed in a constant expression!) */
    int a2[x];     /* error: array bound not a constant */

    switch (v) {
    case 1:
        /* ... */
        break;
    case max:    /* error: case label not a constant (OK in C++) */
        /* ... */
        break;
    }
}

```

The technical reason in C (though not in C++) is that a **const** is implicitly accessible from other translation units:

```

/* file x.c: */
const int x;      /* initialize elsewhere */

```

```
/* file xx.c: */
const int x = 7;      /* here is the real definition */
```

In C++, that would be two different objects, each called **x** in its own file. Instead of using **const** to represent symbolic constants, C programmers tend to use macros. For example:

```
#define MAX 30

void f(int v)
{
    int a1[MAX];      /* OK */

    switch (v) {
    case 1:
        /* ... */
        break;
    case MAX:      /* OK */
        /* ... */
        break;
    }
}
```

The name of the macro **MAX** is replaced by the characters **30**, which is the value of the macro; that is, the number of elements of **a1** is **30** and the value in the second case label is **30**. We use all capital letters for the **MAX** macro, as is conventional. This naming convention helps minimize errors caused by macros.

27.8 Macros

Beware of macros: in C there are no really effective ways of avoiding macros, but their use has serious side effects because they don't obey the usual C (or C++) scope and type rules. Macros are a form of text substitution. See also §A.17.2.

How do we try to protect ourselves from the potential problems of macros apart from (relying on C++ alternatives and) minimizing their use?

- Give all macros we define **ALL_CAPS** names.
- Don't give anything that isn't a macro an **ALL_CAPS** name.
- Never give a macro a short or "cute" name, such as **max** or **min**.
- Hope that everybody else follows this simple and common convention.

The main uses of macros are

- Definition of “constants”
- Definition of function-like constructs
- “Improvements” to the syntax
- Control of conditional compilation

In addition, there is a wide variety of less common uses.

We consider macros seriously overused, but there are no reasonable and complete alternatives to the use of macros in C programs. It can even be hard to avoid them in C++ programs (especially if you need to write programs that have to be portable to very old compilers or to platforms with unusual constraints).

Apologies to people who consider the techniques described below “dirty tricks” and believe such are best not mentioned in polite company. However, we believe that programming is to be done in the real world and that these (very mild) examples of uses and misuses of macros can save hours of grief for the novice programmer. Ignorance about macros is not bliss.

27.8.1 Function-like macros

Here is a fairly typical function-like macro:

```
#define MAX(x, y) ((x)>=(y)?(x):(y))
```

We use the capital **MAX** to distinguish it from the many functions called **max** (in various programs). Obviously, this is very different from a function: there are no argument types, no block, no return statement, etc., and what are all those parentheses doing? Consider:

```
int aa = MAX(1,2);
double dd = MAX(aa++,2);
char cc = MAX(dd,aa)+2;
```

This expands to

```
int aa = ((1)>=( 2)?(1):(2));
double dd = ((aa++)>=(2)?( aa++):(2));
char cc = ((dd)>=(aa)?(dd):(aa))+2;
```

Had “all the parentheses” not been there, the last expansion would have ended up as

```
char cc = dd>=aa?dd:aa+2;
```

That is, `cc` could easily have gotten a different value from what you would reasonably expect looking at the definition of `cc`. When you define a macro, remember to put every use of an argument as an expression in parentheses.

On the other hand, not all the parentheses in the world could save the second expansion. The macro parameter `x` was given the value `aa++`, and since `x` is used twice in `MAX`, `a` can get incremented twice. Don't pass an argument with a side effect to a macro.

As it happens, some genius did define a macro like that and stuck it in a popular header file. Unfortunately, he also called it `max`, rather than `MAX`, so when the C++ standard header defines

```
template<class T> inline T max(T a,T b) { return a<b?b:a; }
```

the `max` gets expanded with the arguments `T a` and `T b`, and the compiler sees

```
template<class T> inline T ((T a)>=( T b)?( T a):( T b)) { return a<b?b:a; }
```

The compiler error messages are “interesting” and not very helpful. In an emergency, you can “undefine” a macro:

```
#undef max
```

Fortunately, that macro was not all that important. However, there are tens of thousands of macros in popular header files; you can't undefine them all without causing havoc.

Not all macro parameters are used as expressions. Consider:

```
#define ALLOC(T,n) ((T*)malloc(sizeof(T)*n))
```

This is a real example that can be very useful for avoiding errors stemming from a mismatch of the intended type of an allocation and its use in a `sizeof`:

```
double* p = malloc(sizeof(int)*10);    /* likely error */
```

Unfortunately, it is nontrivial to write a macro that also catches memory exhaustion. This might do, provided that you define `error_var` and `error()` appropriately somewhere:

```
#define ALLOC(T,n) (error_var = (T*)malloc(sizeof(T)*n), \
                    (error_var==0)\
                    ?(error("memory allocation failure"),0)\
                    :error_var)
```

The lines ending with `\` are not a typesetting problem; it is the way you break a macro definition across lines. When writing C++, we prefer to use **new**.

27.8.2 Syntax macros

You can define macros that make the source code look more to your taste. For example:

```
#define forever for(;;)
#define CASE break; case
#define begin {
#define end }
```



We strongly recommend against this. *Many* people have tried this idea. They (or the people who maintain their code) find that

- Many people don't share their idea of what is a better syntax.
- The "improved" syntax is nonstandard and surprising; others get confused.
- There are uses of the "improved" syntax that cause obscure compile-time errors.
- What you see is not what the compiler sees, and the compiler reports errors in the vocabulary it knows (and sees in source code), not in yours.

Don't write syntactic macros to "improve" the look of code. You and your best friends might find it really nice, but experience shows that you'll be a tiny minority in the larger community, so that someone will have to rewrite your code (assuming it survives).

27.8.3 Conditional compilation

Imagine you have two versions of a header file, say, one for Linux and one for Windows. How do you select in your code? Here is a common way:

```
#ifdef WINDOWS
#include "my_windows_header.h"
#else
#include "my_linux_header.h"
#endif
```

Now, if someone had defined **WINDOWS** before the compiler sees this, the effect is

```
#include "my_windows_header.h"
```

Otherwise it is

```
#include "my_linux_header.h"
```

The `#ifndef WINDOWS` test doesn't care what `WINDOWS` is defined to be; it just tests that it is defined.

Most major systems (including all operating system variants) have macros defined so that you can check. The check whether you are compiling as C++ or compiling as C is

```
#ifdef __cplusplus
    // in C++
#else
    /* in C */
#endif
```

A similar construct, often called an *include guard*, is commonly used to prevent a header file from being `#include`d twice:

```
/* my_windows_header.h: */
#ifndef MY_WINDOWS_HEADER
#define MY_WINDOWS_HEADER
    /* here is the header information */
#endif
```

The `#ifndef` test checks that something is not defined; i.e., `#ifndef` is the opposite of `#ifdef`. Logically, these macros used for source file control are very different from the macros we use for modifying source code. They just happen to use the same underlying mechanisms to do their job.

27.9 An example: intrusive containers

The C++ standard library containers, such as `vector` and `map`, are non-intrusive; that is, they require no data in the types used as elements. That is how they generalize nicely to essentially all types (built-in or user-defined) as long as those types can be copied. There is another kind of container, an *intrusive container*, that is popular in both C and C++. We will use an intrusive list to illustrate C-style use of `structs`, pointers, and free store.

Let's define a doubly-linked list with nine operations:

```
void init(struct List* lst);          /* initialize lst to empty */
struct List* create();              /* make a new empty list on free store */
void clear(struct List* lst);       /* free all elements of lst */
void destroy(struct List* lst);     /* free all elements of lst, then free lst */

void push_back(struct List* lst, struct Link* p); /* add p at end of lst */
void push_front(struct List*, struct Link* p);   /* add p at front of lst */
```

```

/* insert q before p in lst: */
void insert(struct List* lst, struct Link* p, struct Link* q);
struct Link* erase(struct List* lst, struct Link* p); /* remove p from lst */

/* return link n "hops" before or after p: */
struct Link* advance(struct Link* p, int n);

```

The idea is to define these operations so that their users need only use **List***s and **Link***s. This implies that the implementation of these functions could be changed radically without affecting those users. Obviously, the naming is influenced by the STL. **List** and **Link** can be defined in the obvious and trivial manner:

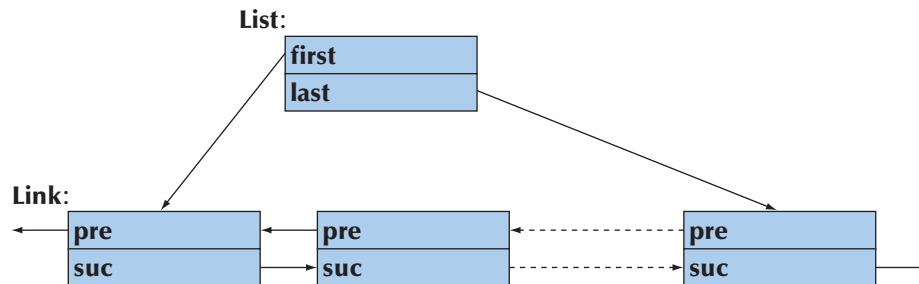
```

struct List {
    struct Link* first;
    struct Link* last;
};

struct Link { /* link for doubly-linked list */
    struct Link* pre;
    struct Link* suc;
};

```

Here is a graphical representation of a **List**:



It is not our aim to demonstrate clever representation techniques or clever algorithms, so there are none of those here. However, do note that there is no mention of any data held by the **Links** (the elements of a **List**). Looking back at the functions provided, we note that we are doing something very similar to defining a pair of abstract classes **Link** and **List**. The data for **Links** will be supplied later. **Link*** and **List*** are sometimes called handles to opaque types; that is, giving **Link***s and **List***s to our functions allows us to manipulate elements of a **List** without knowing anything about the internal structure of a **Link** or a **List**.

To implement our **List** functions, we first **#include** some standard library headers:

```
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
```

C doesn't have namespaces, so we need not worry about **using** declarations or **using** directives. On the other hand, we should probably worry that we have grabbed some very common short names (**Link**, **insert**, **init**, etc.), so this set of functions cannot be used "as is" outside a toy program.

Initializing is trivial, but note the use of **assert()**:

```
void init(struct List* lst)    /* initialize *lst to the empty list */
{
    assert(lst);
    lst->first = lst->last = 0;
}
```

We decided not to deal with error handling for bad pointers to lists at run time. By using **assert()**, we simply give a (run-time) system error if a list pointer is null. The "system error" will give the file name and line number of the failed **assert()**; **assert()** is a macro defined in **<assert.h>** and the checking is enabled only during debugging. In the absence of exceptions, it is not easy to know what to do with bad pointers.

The **create()** function simply makes a **List** on the free store. It is a sort of combination of a constructor (**init()** initializes) and **new** (**malloc()** allocates):

```
struct List* create()        /* make a new empty list */
{
    struct List* lst = (struct List*)malloc(sizeof(struct List));
    init(lst);
    return lst;
}
```

The **clear()** function assumes that all **Links** are created on the free store and **free()**s them:

```
void clear(struct List* lst)  /* free all elements of lst */
{
    assert(lst);
    {
```

```

    struct Link* curr = lst->first;
    while(curr) {
        struct Link* next = curr->suc;
        free(curr);
        curr = next;
    }
    lst->first = lst->last = 0;
}
}

```

Note the way we traverse using the **suc** member of **Link**. We can't safely access a member of a **struct** object after that object has been **free()**d, so we introduce the variable **next** to hold our position in the **List** while we **free()** a **Link**.

If we didn't allocate all of our **Links** on the free store, we had better not call **clear()**, or **clear()** will create havoc.

The **destroy()** function is essentially the opposite of **create()**, that is, a sort of combination of a destructor and a **delete**:

```

void destroy(struct List* lst)  /* free all elements of lst; then free lst */
{
    assert(lst);
    clear(lst);
    free(lst);
}

```

Note that we are making no provisions for calling a cleanup function (destructor) for the elements represented by **Links**. This design is not a completely faithful imitation of C++ techniques or generality – it couldn't and probably shouldn't be.

The **push_back()** function – adding a **Link** as the new last **Link** – is pretty straightforward:

```

void push_back(struct List* lst, struct Link* p) /* add p at end of lst */
{
    assert(lst);
    {
        struct Link* last = lst->last;
        if (last) {
            last->suc = p;          /* add p after last */
            p->pre = last;
        }
        else {
            lst->first = p;        /* p is the first element */
            p->pre = 0;
        }
    }
}

```

```

    }
    lst->last = p;           /* p is the new last element */
    p->suc = 0;
  }
}

```

However, we would never have gotten it right without drawing a few boxes and arrows on our doodle pad. Note that we “forgot” to consider the case where the argument **p** was null. Pass 0 instead of a pointer to a **Link** and this code will fail miserably. This is not inherently bad code, but it is *not* industrial strength. Its purpose is to illustrate common and useful techniques (and, in this case, also a common weakness/bug).

The **erase()** function can be written like this:

```

struct Link* erase(struct List* lst, struct Link* p)
/*
   remove p from lst;
   return a pointer to the link after p
*/
{
  assert(lst);
  if (p==0) return 0;           /* OK to erase(0) */

  if (p == lst->first) {
    if (p->suc) {
      lst->first = p->suc;       /* the successor becomes first */
      p->suc->pre = 0;
      return p->suc;
    }
    else {
      lst->first = lst->last = 0; /* the list becomes empty */
      return 0;
    }
  }
  else if (p == lst->last) {
    if (p->pre) {
      lst->last = p->pre;     /* the predecessor becomes last */
      p->pre->suc = 0;
    }
    else
      lst->first = lst->last = 0; /* the list becomes empty */
    return 0;
  }
}

```

```

    }
    else {
        p->suc->pre = p->pre;
        p->pre->suc = p->suc;
        return p->suc;
    }
}

```

We will leave the rest of the functions as an exercise, as we don't need them for our (all too simple) test. However, now we must face the central mystery of this design: Where is the data in the elements of the list? How do we implement a simple list of names represented by a C-style string? Consider:

```

struct Name {
    struct Link lnk;    /* the Link required by List operations */
    char* p;          /* the name string */
};

```

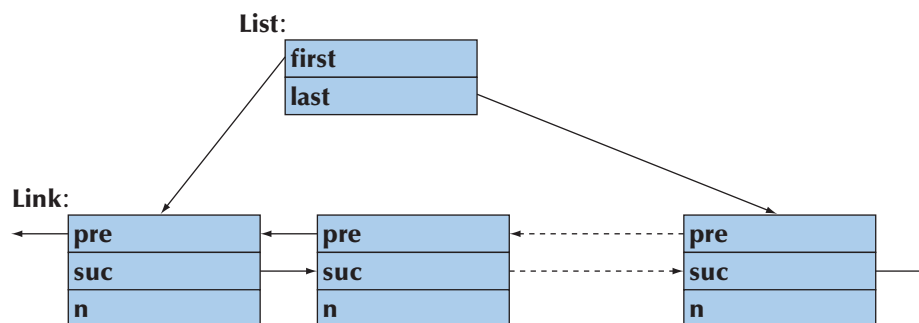
So far, so good, though how we get to use that **Link** member is a mystery; but since we know that a **List** likes its **Links** on the free store, we write a function creating **Names** on the free store:

```

struct Name* make_name(char* n)
{
    struct Name* p = (struct Name*)malloc(sizeof(struct Name));
    p->p = n;
    return p;
}

```

Or graphically:



Now let's use that:

```
int main()
{
    int count = 0;
    struct List names;      /* make a list */
    struct List* curr;
    init(&names);

    /* make a few Names and add them to the list: */
    push_back(&names,(struct Link*)make_name("Norah"));
    push_back(&names,(struct Link*)make_name("Annemarie"));
    push_back(&names,(struct Link*)make_name("Kris"));

    /* remove the second name (with index 1): */
    erase(&names,advance(names.first,1));

    curr = names.first;    /* write out all names */
    for (; curr!=0; curr=curr->suc) {
        count++;
        printf("element %d: %s\n", count, ((struct Name*)curr)->p);
    }
}
```

So we “cheated.” We used a cast to treat a **Name*** as a **Link***. In that way, the user knows about the “library-type” **Link**. However, the “library” doesn’t know about the “application-type” **Name**. Is that allowed? Yes, it is: in C (and C++), you can treat a pointer to a **struct** as a pointer to its first element and vice versa.

Obviously, this **List** example is also C++ exactly as written.

TRY THIS



A common refrain among C++ programmers talking with C programmers is, “Everything you can do, I can do better!” So, rewrite the intrusive **List** example in C++, showing how to make it shorter and easier to use without making the code slower or the objects bigger.



Drill

1. Write a “Hello, World!” program in C, compile it, and run it.
2. Define two variables holding “Hello” and “World!” respectively; concatenate them with a space in between; and output them as **Hello World!**.
3. Define a C function that takes a **char*** parameter **p** and an **int** parameter **x** and print out their values in this format: **p is "foo" and x is 7**. Call it with a few argument pairs.

Review

In the following, assume that by C we mean ISO standard C89.

1. Is C++ a subset of C?
2. Who invented C?
3. Name a highly regarded C textbook.
4. In what organization were C and C++ invented?
5. Why is C++ (almost) compatible with C?
6. Why is C++ only *almost* compatible with C?
7. List a dozen C++ features not present in C.
8. What organization “owns” C and C++?
9. List six C++ standard library components that cannot be used in C.
10. Which C standard library components can be used in C++?
11. How do you achieve function argument type checking in C?
12. What C++ features related to functions are missing in C? List at least three. Give examples.
13. How do you call a C function from C++?
14. How do you call a C++ function from C?
15. Which types are layout compatible between C and C++? (Just) give examples.
16. What is a structure tag?
17. List 20 C++ keywords that are not keywords in C.
18. Is **int x;** a definition in C++? In C?
19. What is a C-style cast and why is it dangerous?
20. What is **void*** and how does it differ in C and C++?
21. How do enumerations differ in C and C++?
22. What do you do in C to avoid linkage problems from popular names?
23. What are the three most common C functions from free-store use?
24. What is the definition of a C-style string?
25. How do **==** and **strcmp()** differ for C-style strings?

26. How do you copy C-style strings?
27. How do you find the length of a C-style string?
28. How would you copy a large array of `ints`?
29. What's nice about `printf()`? What are its problems/limitations?
30. Why should you never use `gets()`? What can you use instead?
31. How do you open a file for reading in C?
32. What is the difference between `const` in C and `const` in C++?
33. Why don't we like macros?
34. What are common uses of macros?
35. What is an include guard?

Terms

<code>#define</code>	Dennis Ritchie	non-intrusive
<code>#ifdef</code>	FILE	opaque type
<code>#ifndef</code>	fopen()	overloading
Bell Labs	format string	printf()
Brian Kernighan	intrusive	strcpy()
C/C++	K&R	structure tag
compatibility	lexicographical	three-way comparison
conditional compilation	linkage	void
C-style cast	macro	void*
C-style string	malloc()	

Exercises

For these exercises it may be a good idea to compile all programs with both a C and a C++ compiler. If you use only a C++ compiler, you may accidentally use non-C features. If you use only a C compiler, type errors may remain undetected.

1. Implement versions of `strlen()`, `strcmp()`, and `strcpy()`.
2. Complete the intrusive `List` example in §27.9 and test it using every function.
3. “Pretty up” the intrusive `List` example in §27.9 as best you can to make it convenient to use. Do catch/handle as many errors as you can. It is fair game to change the details of the `struct` definitions, to use macros, whatever.
4. If you didn't already, write a C++ version of the intrusive `List` example in §27.9 and test it using every function.
5. Compare the results of exercises 3 and 4.

6. Change the representation of **Link** and **List** from §27.9 without changing the user interface provided by the functions. Allocate **Links** in an array of links and have the members **first**, **last**, **pre**, and **suc** be **ints** (indices into the array).
7. What are the advantages and disadvantages of intrusive containers compared to C++ standard (non-intrusive) containers? Make lists of pros and cons.
8. What is the lexicographical order on your machine? Write out every character on your keyboard together with its integer value; then, write the characters out in the order determined by their integer value.
9. Using only C facilities, including the C standard library, read a sequence of words from **stdin** and write them to **stdout** in lexicographical order. Hint: The C sort function is called **qsort()**; look it up somewhere. Alternatively, insert the words into an ordered list as you read them. There is no C standard library list.
10. Make a list of C language features adopted from C++ or C with Classes (§27.1).
11. Make a list of C language features not adopted by C++.
12. Implement a (C-style **string**, **int**) lookup table with operations such as **find(struct table*, const char*)**, **insert(struct table*, const char*, int)**, and **remove(struct table*, const char*)**. The representation of the table could be an array of a **struct** pair or a pair of arrays (**const char*[]** and **int***); you choose. Also choose return types for your functions. Document your design decisions.
13. Write a program that does the equivalent of **string s; cin>>s;** in C; that is, define an input operation that reads an arbitrarily long sequence of whitespace-terminated characters into a zero-terminated array of **chars**.
14. Write a function that takes an array of **ints** as its input and finds the smallest and the largest elements. It should also compute the median and mean. Use a **struct** holding the results as the return value.
15. Simulate single inheritance in C. Let each “base class” contain a pointer to an array of pointers to functions (to simulate virtual functions as free-standing functions taking a pointer to a “base class” object as their first argument); see §27.2.3. Implement “derivation” by making the “base class” the type of the first member of the derived class. For each class, initialize the array of “virtual functions” appropriately. To test the ideas, implement a version of “the old **Shape** example” with the base and derived **draw()** just printing out the name of their class. Use only language features and library facilities available in standard C.
16. Use macros to obscure (simplify the notation for) the implementation in the previous exercise.

Postscript

We did mention that compatibility issues are not all that exciting. However, there is a lot of C code “out there” (billions of lines of code), and if you have to read or write it, this chapter prepares you to do so. Personally, we prefer C++, and the information in this chapter gives part of the reason for that. And please don’t underestimate that “intrusive **List**” example – both “intrusive **Lists**” and opaque types are important and powerful techniques (in both C and C++).