C and C++ in 5 days

Philip Machanick

Computer Science Department
University of the Witwatersrand
2050 Wits
South Africa
philip@concave.cs.wits.ac.za

please contact the author for commercial use

contents

Preface

C was developed in the 1970s to solve the problem of implementing the UNIX operating system in a maintainable way. An unexpected consequence was that UNIX also became relatively portable. Consequently, some think of UNIX the first computer virus, but this is erroneous. There are major technical differences between UNIX and a virus.

C was designed a time when computer memories were small, especially on the low-end computers for which UNIX was originally designed. At the same time, compiler-writing techniques were not as well developed as they are today. Most of the code optimization technology of the time was oriented towards making FORTRAN floating-point programs as fast as possible, and tricks used in modern compilers to keep registers in variables as long as possible, and to minimize the number of times and array index must be computed—to give two examples—were still to be developed.

As a consequence, to make C viable for operating system development, the language has many features which are unsafe, and with today's compiler technology, unnecessary. Even so, only the best compilers, typically found on UNIX systems, implement really good code generation, and typical PC compilers are not as good. Part of the reason is to be found in the instruction set of the Intel 80*x*86 processor line, which has very few general-purpose registers, and a large range of equivalent instructions to choose from in some circumstances.

These notes introduce C with a modern style of programming, emphasizing avoidance of the most risky features, while explaining where their use may still be appropriate. The intended audience is experienced programmers who may be used to a more structured language, such as Pascal, Ada or Modula–2; differences from such languages are noted where appropriate or useful.

As a bridge to C++, object-oriented design is introduced with C as a vehicle. This illustrates how object-oriented design is a separate concept from object-oriented languages—even though an object-oriented language is clearly the better implementation medium for an object-oriented design.

The notes are divided into 14 parts, each of which is followed by a hands-on or discussion session. The first half is about C, concluding with object-oriented design and how it relates to C. C++ is introduced as a better way of implementing object-oriented designs.

The notes are intended to supply enough material to absorb in a week; some sources of further information include:

Brian W Kernighan and Dennis M Richie. *The C Programming Language* (2nd edition), Prentice-Hall, Englewood Cliffs, NJ, 1988. ISBN 0–13–110362–8.

Margaret A Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990. ISBN 0–201–51459–1.

Stanley B Lippman. *C++ Primer* (2nd edition), Addison-Wesley, Reading, MA, 1989. ISBN 0–201–17928–8.

Grady Booch. *Object-Oriented Design with Applications*, Addison-Wesley, Reading, MA, 1991. ISBN 0–201–56527–7.

acknowledgement

András Salamon proof read this document and suggested some clarifications.

Part 1—Overview

file structure

C source files are organized as compilable files and *headers*. A header file contains *declarations*; a compilable file imports these declarations, and contains *definitions*.

A definition tells the compiler what code to generate, or to allocate storage for a variable whereas a declaration merely tells the compiler the type associated with a name. Headers are generally used for publishing the interface of separately compiled files.

In UNIX it's usual to end compilable file names with "`.c`" and headers with "`.h`".

A compilable file imports a header by a line such as (usually for system headers):

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

or (usually for your own headers):

```
#include "employees.h"
```

The difference between the use of `<>` and `""` will be explained later.

When the header is imported, it's as if the `#include` line had been replaced by the contents of the named file.

> **caution—**in C, this is only a convention—but one that should not be broken: the header *could* contain anything, but it *should* only contain declarations and comments if your code is to be maintainable

simple program

Only a few more points are needed to write a program, so here's an example:

```
#include <stdio.h>
void main(int argc, char *argv[])
{ int i;
   for (i=0; i < argc; i++)
      printf("command line argument [%d] = %s\n", i, argv[i]);
}
```

The first line imports a system header, for standard input and output. The second line is the standard way of declaring a main program. A main program can return a result of type `int`, though this one doesn't actually return a value, hence the `void`.

The main program has two *arguments*, the first of which is a count of command-line arguments from the command that started the program. The second is a pointer to an array of strings each of which is a separate command-line argument. By convention, the first string is the name of the program. Note the syntax: a "*" is used to declare a pointer, and an empty pair of square brackets is used to denote a variable-sized array.

The next thing to notice is the use of curly brackets for a **begin-end** block.

The main program declares a variable `i`, and uses it as a loop control variable.

Note the convention of counting from zero, and using a < test to terminate the loop. This convention is useful to adhere to because C arrays are indexed from zero.

The `for` loop control actions must be in parentheses, and the initialization, test and increment are separated by semicolons.

The body of the loop is a single statement in this case, so no `{}` are needed to group statements into the body.

The body of the loop uses library function `printf()` to produce output. The first string is used to format the arguments that follow. The `%d` in the format string causes the next argument to be printed as a decimal integer, and the `%s` causes the final argument to be printed as a string. A "\n" terminates a line of output.

what does the program do? No prizes for the answer.

a few details

In C, there is no distinction between functions and procedures. There is a distinction between statements and expressions, but it is usually possible to use an expression as a statement. For example, the following is legal C code (note use of `/* … */` for comments):

```
void main(int argc, char *argv[])
{  int i;
   i+1;  /* this is an expression */
}
```

This is a silly example, but there are cases where the result of an expression is not needed, just its side effect (i.e., what it changes in the global environment).

Functions that do not return a result are declared to return type `void`, which is a general non-type also used to specify pointers that can't be dereferenced, among other things.

A few examples of expressions that might be considered statements in other languages:

- assignment—done with "`=`" in C—so for example, it's possible to do a string of initializations in one go (comparison for equality uses "`==`"):
  ```
  int i,j;
  i = j = 0;  /* j = 0 is an expression: returns new value of j */
  ```
- procedure call—always a function call in C, even if no value is returned (on the other hand a value-returning function can be called as if it were a procedure call, in which case the value is thrown away)
- increment (`var_name++`) and decrement (`var_name--`): the exact behaviour of these constructs is too complex for an introduction; they are explained more fully later

Unlike some languages (e.g., LISP, Algol-68) that treat everything as expressions, most other statements in C cannot be used to return a value. These include selection (`if` and `switch`), loops (`while`, `for` and `do…while`—the latter like a Pascal **repeat**) and `{}` blocks.

hands-on—enter the program on page 2

aims: learn to use the editor and compiler; get a feel for C syntax.

caution: C is case-sensitive—be careful to observe capitalization (for example: `r` and `R` are different variable names—but someone maintaining your code won't thank you if you exploit this feature)

part 2—Language Elements

functions

Functions in C do not have to have a return type specified: the default is `int`. It is a good convention however to put the type in even in this case. Functions that are called as procedures (i.e., return no value) are declared as returning `void`.

A function—as we shall see later—can be stored as a variable or passed as a parameter, so a function has a type like any other value.

The complete specification of the function's type is given by a *prototype*, specifying the function's return type, name and argument types, for example:

```
void sort (int data[], int n);
```

It is permissible to leave out the names of the arguments:

```
void sort (int [], int);
```

This is not good practice: names make the purpose of the arguments more obvious.

Prototypes are usually used in headers, but can be used if the function is called before it's defined. As we shall see later, prototypes are also used in C++ class declarations.

In C, parameter passing is by *value*: values of arguments are copied to the function. To pass by reference (Pascal **var** parameters), you create a pointer to the parameter in the call. This is done using the `&` operator, which creates a pointer to its operand. For example:

```
void swap (int *a, int *b)
{  int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
/* called somewhere: */
int first, second;
/* give them values, then: */
swap (&first, &second);
```

Inside the function, `a` and `b` are *pointers* to `int`s (`int*`). To access their values in the function they must be *dereferenced*. The Pascal dereference operator is `^`; C's is `*`. A notational convenience: you write a variable in a declaration the same way as you write it when you dereference it. This makes it easy to remember where to put the `*`.

In the call, the variables `first` and `second` are not of a pointer type, so a pointer to the values they hold has to be created explicitly using the `&` operator.

What would happen if the example changed as follows?

```
void swap (int a, int b)
{  int temp;
   temp = a;
   a = b;
   b = temp;
}
/* called somewhere: */
int first, second;
/* give them values */
swap (first, second);   /* what does this actually do? */
```

To return a value from a function:

```
return value;     /* immediately exits function */
```

Functions returning `void` can use `return` with no value to exit immediately (e.g. on discovering an error condition).

Unlike in Pascal, functions can only be global, though their names can be restricted to file scope by declaring them `static` (see Part 6 for more on `static`).

types

A few C types have sneaked in without introduction.

Now it's time to be more explicit.

We've already seen the types `int` (**integer**) and `int*` (**pointer to integer**).

These types correspond to integer types in most other languages though as usual it is possible for different compilers on the same machine to have different conventions as to the size of a given type.

In C it is often assumed that an `int` is the size of a machine address, though when we reach the portability section in Part 6 we shall see that this can be a problem.

Type `char` is a single byte integer type.

Integer types can be qualified by `short`, `long` or `unsigned` (or any combinations of these that make sense and are supported by the compiler—e.g., `short long` doesn't make sense). You can leave "`int`" out if a qualifier is used. The Standard is vague about which must be supported and the sizes they may be. Often, `long` and `int` are the same. (Originally, `int` was 16 bits and `long` 32 bits; when the transition to 32 bits was made, many compiler writers left `long` as 32 bits.)

There is no **boolean** type: any integer value other than zero tests as true.

> **exercise:** look up the sizes for your compiler
> **even better exercise:** use `sizeof(type_name)` to find out
>
> **note:** sizeof has to have parentheses if called on a type name, but not if called on an expressions, e.g., `sizeof sortdata`

Reals are represented by type `float`. Inconsistently, the type for a double-precision `float` is `double` (and *not* `long float`). Extended precision reals are `long double`.

To tell the compiler an integer constant is a `long`, it has an `L` at the end of the number (not necessarily capital[†], but a lower-case `l` looks too much like the digit `1` to be good idea); floating-point constants default to double unless they end in an `F` (or `f`).

Interesting to the low-level hacker: constants in hexadecimal start with zero `X`:

`0x8086`

has the same value as the decimal constant

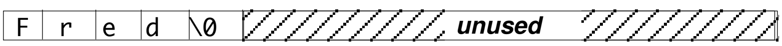
`32902` (or `32902L`, if your compiler uses `long` for values too big for 16 bits)

Variables may also be specified as either `volatile` or `register`. If a variable can be seen by more than one process (e.g., part of memory used by a memory-mapped I/O device), it can be declared `volatile` to warn the compiler not to hold it in a register. This is a relatively new feature and not all compilers may implement it yet. Specifying a variable as `register` (keep it in a register as much as possible) is obsolete on better compilers: it just gets in the way of the code generator's strategy.

Character values are given by a character in single quotes; you can specify them in octal or hexadecimal, as well as a limited number of special symbols, like `'\n'` for new line.

Strings are enclosed in double quotes, and are stored in an array of `char` terminated by the null character (which you can specify as `'\0'`—a zero after a backslash). Example:

```
char  name[100];
strcopy(name,  "Fred");  /* library routine to copy a string */
```



> **caution:** if you declare an array of `char` to store a string, make sure it's 1 bigger than the longest string you need to store to allow for the terminating null character

[†] If you ask me it makes more sense for variables to be case insensitive and to insist on a capital L in this case rather than vice-versa.

statements

C has the usual collection of assignments, selection, iteration and procedure call statements. We will not dwell on details of syntax here; more detail will be considered in Part 3.

> **caution:** it's easy to write `if (i=0)` instead of `if (i==0)`. What's the effect of this error? C has no **boolean** type. If `i` is an `int`, the compiler won't report an error: `i=0` is a valid `int` expression. Good compilers issue a warning: send yours back if it doesn't

There are two selection statements, `if` and `switch`:

```
if (n==0)   /* note "()" but no "then" */
   printf("no data\n");
else     /* else is optional */
{        /* use {} for more than one statement */
   average = total / n;
   printf("Average = %d\n", average);
}
```

or the same thing with a `switch`:

```
switch (n)  /* again the () is needed */
{           /* the cases must be enclosed in {} */
case 0:  /* can be any constant int expression */
   printf("no data\n");
break;
default: /* in effect case "anything" */
   average = total / n; /* note no {} needed */
   printf("Average = %d\n", average);
break;   /* not strictly necessary */
}
```

The `switch` is quite complicated. The `break` is used to quit; without it you fall through to the next `case`. There can be more than one `case`, e.g.,

```
case 'a': case 'e': case 'i': case 'o': case 'u':
   printf("vowel\n");
break;
```

This is a degenerate case of falling through with no `break`.

A `break` after the last `case` (`default` in the first example) isn't needed, but ending every `case` with a `break` makes it easier to be avoid errors and to modify the code later.

> **caution:** the rules of always putting in a `break` and never falling through from one `case` to another derive from many years of experience of maintaining code—don't break them

Loops present slightly fewer possibilities for getting into trouble.

A `while` is reasonably straightforward:

```
while (i > 0)
   i--;  /* decrement i */
```

As is a `do-while`:

```
do
{  i--;
}  while (i>0);
```

exercise: how does the behaviour of the above two loops differ?

The `for` loop is a touch more complicated. Its control consists of initialization, test and increment:

```
for (i=first(); i<last() && i> cut_off(); i++)
```

```
    ;  /* do something */
```

is one example (notice how parameterless functions are called with empty parentheses).

> **aside:** in C **and**is written `&&`; **or** is `||`; "single" versions `&` and `|` are bitwise operations; `!` is **not** (`1` if operand is `0`, `0` otherwise)

All loops can be broken by `break` (exits the innermost loop enclosing the `break`) or `continue` (goes to the control computation, skipping the rest of the innermost loop body). These constructs are a bit better than unrestricted **goto**; see hints on usage later.

hands-on—a larger program

Given the partially written program below, fix the indicated bug, and add code to count the number of negative, zero and positive numbers in the data read in. You should use `switch`, `if` and at least one loop construct

```
#include <stdio.h>

/* bug: should check if s is zero */
int sign (int s)
{  return abs(s)/s;
}

main ()
{  int data [10],
       i, n,
       negatives, zeros, positives;
   n = sizeof data / sizeof (int);
   negatives = zeros = positives  = 0;
   printf("Enter %d numbers : ", n);
   /* need loop on i from 0 to n-1 around this */

   /* read in the data */
      scanf("%d", &data[i]);
   /* now count negatives , zeros, positives */










   printf("negatives=%d,zeros=%d,positives=%d\n",
      negatives, zeros, positives);
}
```

part 3—Style and Idioms

switch

We've been through most of the important features of the `switch` already. Perhaps the most important point to emphasize is that this statement can easily become very large and clumsy, and a disciplined approach is necessary.

If you find yourself programming with large complex `switch` statements, it's time to clean up you design: look for a simpler way of expressing the problem.

When we look at object-oriented programming, we shall see an alternative: the use of *dynamic dispatch*, in which the type of the object determines what kind of action should be carried out. Remember this point for now: if you are using C++ and end up with a lot of big clumsy `switch` statements, reconsider your design.

loops

When we look at arrays and pointers, some interesting strategies for writing loops to go through an array quickly will come up. In this section we'll stick to variants on loop behaviour different from languages such as Pascal.

In situations such as operating systems, event-driven user interfaces and simulations, where termination is unusual (system shutdown, quitting application, or special action to clean up the simulation) it's useful to have a loop which goes on forever, with termination almost an exception (error) condition.

C doesn't have an explicit construct for this but many C programmers simply write

```
while (1)
{  /* do all kinds of things */
   if (good_reason_to_quit())
      break;
}
```

This kind of use of `break` is acceptable programming style. However you should take care not to abuse this feature, as loops with multiple exit points are hard to debug and maintain. Ideally the loop should either have exactly one exit point, or any additional exit points should only handle very rare conditions. Otherwise, you are back to unstructured programming as if you had used an unrestricted **goto** construct.

The `continue` statement is less commonly used, and on the whole it is probably better to use an `if` to skip over unwanted parts of a loop. I have never used `continue` and have never encountered a situation where I felt it could be useful. If you find a need for it, rethink the problem. It's not good style to use a rarely used feature—you will confuse maintainers.

arguments

Parameter passing in C seems superficially very simple but there are some serious traps and pitfalls. The worst of these is the way the pass by value rule interacts with the way arrays are implemented.

In C, an array is in fact a pointer which (usually) happens to have had memory allocated for it automatically. Consequently, when an array is passed as a parameter, what is actually copied on the function call is not the whole array but just a pointer to its first element. This is very efficient compared with copying the whole array but it also means it's easy to write a function that alters an array, forgetting that the original is being altered, and not a copy.

> **caution:** there's no such thing in C as passing a *whole* array by value: only a pointer is copied and the actual elements are overwritten if the function changes them
>
> **double caution:** I lied: if an array is a field in a `struct` (see Part 4), it *is* copied when the `struct` is passed as a parameter

pointers and returning values

When you want to change the value of an actual parameter (the one in the call), you must send a pointer to the parameter, rather than its value. This wouldn't be a problem if C had strict type checking, but since it doesn't things can go wrong.

If you are using an older C compiler that doesn't conform to the ANSI standard, it may take a very lax view of what is or isn't a pointer, or of mixing different types of pointers. Luckily recent compilers conforming to the ANSI standard of 1989 do better type checking, and C++ is even more strict.

Even with newer compilers however, there are cases where type checking isn't done. The worst case is with functions designed to take a variable number of arguments, such as `scanf()`, which does formatted input. For example, in the code

```
int i;
scanf ("%d",i);   /* should be &i */
```

`scanf()` should read an integer in decimal format into `i` from standard input, but `i` should in fact be `&i`. As a result of this error, `scanf()` uses `i` as a pointer with unimaginable consequences. A compiler cannot in general detect such errors. C++ has another I/O mechanism, so these routines don't have to be used—but in C it's vital to check parameters in functions taking a variable number of arguments very carefully.

It's a good strategy to put all input (where this problem is worst) in one place to make checking easier.

arrays, pointer arithmetic and array arguments

Brief mention has been made of how C treats arrays as pointers. The array definition

```
int a[100];
```

has almost† the same effect as defining a pointer and allocating space for 100 `int`s:

```
int *a = (int*) malloc(sizeof(int)*100);
```

A few points about `malloc()`:

- it returns `void*`, and so must be coerced to the correct pointer type—hence the `(int*)` which converts the type of the expression to pointer to integer
- it must be given the correct size in bytes (remember: a string needs one extra byte for the terminating null character)
- the resulting memory cannot be assumed to be initialized
- if memory cannot be allocated, the `NULL` pointer is returned; in principle every call to `malloc()` should check for this. It may not always be necessary (e.g. if you have carefully calculated how much free memory you have)—but this is not something to be casual about
- memory allocated by `malloc()` can be deallocated by `free(ptr_name)`: be careful to call `free()` only on a valid pointer which has been allocated by `malloc()`

Back to arrays: an array access `a[i]` is equivalent to *pointer arithmetic*. C defines addition of a pointer and an `int` as returning an address that integral number of units from the original address (the unit is the size of the object pointed to). For example:

```
int *a;      /* discussion below assumes sizeof(int) == 4 */
double *d;  /* and sizeof(double) == 8 */
a = (int*) malloc (sizeof(int)*100);
d = (double*) malloc (sizeof(double)*20);
a ++;        /* short for a = a + 1 */
d += 2;      /* short for d = d + 2 */
```

results in `a`'s value changing by 4 (to point to the next `int`), while `d`'s changes by 16 (2 `double`s further on in memory).

In terms of pointer arithmetic, since an array's value is a pointer to its first element, `a[i]` is equivalent to

† The main difference: the array's value, while a pointer, isn't an *l*-value (more on *l*-values in Part 13).

```
*(a+i)
```

Because pointer arithmetic is commutative, the above can be rewritten as

```
*(i+a)
```

or—bizarrely:

```
i[a]
```

What use is this? If you have a loop accessing successive array elements, such as

```
int i;
double data[1000];
for (i=0; i<1000; i++)
   data[i] = 0.0;
```

an inefficient compiler generates an array index operation once every time through the loop. An array index operation requires multiplying the index by the size of each element, and adding the result to the start address of the array.

Consider the following alternative code:

```
int i;
double data[1000], *copy, *end;
end = data+1000;
for (copy = data; copy < end; copy ++)
   *copy  = 0.0;
```

On one compiler I tried, the latter code executed 40% fewer instructions in the loop, and did no multiplies, which it did to do array indexing. But this was with no optimization. I turned optimization on, and the “array” version was two instructions shorter!

caution: this could lead you astray. Here’s a hacker version of the loop (it has no body) for which my optimizing compiler generates exactly the same number of instructions as the readable version:

```
/* declarations and end as before */
for (copy=data; copy<end; *(copy++)=0.0) ;
```

Why then does pointer arithmetic persist as a feature?

Sometimes it’s necessary for performance reasons to write your own low-level code, for example, a memory manager. In such a situation, you may have to write very general code in which the size of units you are dealing with is not known in advance. In such cases the ability to do pointer arithmetic is very useful.

However if you find yourself doing pointer arithmetic because your compiler doesn’t generate good code for array indexing, look for a better compiler.

This feature is in the language for reasons of efficiency that have been superseded by better compiler technology, but there are still rare occasions where it is useful.

caution: pointer arithmetic is hard to debug and hard to maintain. Make sure you really do have good reason to use it and there is no alternative before committing to it—and once you have decided to use it, isolate it to as small a part of your code as possible: preferably a single file containing all your low-level hard-to-understand code

hands-on—sorting strings

Here is some code to sort integers. Modify it to sort strings. Use the followin:

```
#include <string.h>
/* from which use
    int strcmp(char* s, char* t) returns
      <0 if s < t, 0 if s ==t , >0 if s > t
   you may also need
   void strcpy(char* s, char* t) copies t to s
*/
```

declare your string array as follows:

```
char strings [10][255]; /* 10 strings of up to 255 chars each */
```

and read them in as follows

```
char strings [10][255]; /* NB not [10,255] */
int i;
printf("Enter 10 strings, max 255 chars each:\n");
for (i=0; i < n; i++)
   scanf("%s", strings[i]);
```

```
#include <stdio.h>
void swap (int data[], int i, int j)
{  int temp;
   temp = data[i];
   data[i] = data[j];
   data[j] = temp;
}

void sort (int data[], int n)
{  int i,j;
   for (i = 0; i < n-1; i++)
      for (j = i + 1; j > 0; j--)
         if (data[j-1] > data[j])
            swap (data, j-1, j);
}

void main()
{  int sort_data [10],
      i, n;
   n = sizeof(sort_data)/sizeof(int);
   printf ("Enter %d integers to sort :", n);
   for (i=0; i<n; i++)
      scanf ("%d", &sort_data[i]);
   sort (sort_data, n);
   printf ("Sorted data:\n\n");
   for (i=0; i<n; i++)
      printf("%d ",sort_data[i]);
   printf(".\n");
}
```

caution: this isn't as easy as it looks—to do this as specified requires a good understanding of the way 2-dimensional arrays are implemented in C; it's actually much easier if the strings are allocated using pointers, so you can swap the pointers much as the `int`s are swapped above. Here's how to initialize the pointers:

```
char *strings [10];
int i, n = 10, str_size = 255; /* in practice read n from file */
for (i = 0; i < n; i++)
      strings [i] = (char*) malloc (255);
```

part 4—Structured Types

struct

The C `struct` is essentially the same as a **record** type in languages like Pascal, but there are some syntactic oddities.

A `struct` is declared as in the following example:

```
struct Employee
{  char *name;
   int   employee_no;
   float salary,  tax_to_date;
};
```

Variables of this type can be defined as follows:

```
struct Employee secretary, MD, software_engineer;
```

Note that the name of the type is `struct Employee`, not just `Employee`.

If you need to define a mutually recursive type (two `structs` that refer to each other), you can do a forward declaration leaving out the detail, as in a tree in which the root holds no data node, and each other node can find the root directly:

```
struct Tree;
struct Root
{  struct Tree *left, *right;
};

struct Tree
{  struct Tree *left, *right;
   char *data;
   struct Root *tree_root;
};
```

caution: the semicolon after the closing bracket of a `struct` is essential even though it isn't needed when similar brackets are used to group statements

double caution: a "*" must be put in for *each* pointer variable or field; if left out the variable is not a pointer (e.g., `right` in both cases above *must* be written as `*right` if a pointer type is wanted)

typedef

C is not a strongly typed language. Aside from generous conversion rules between various integer types, C's named types—declared by `typedef`—are essentially shorthand for the full description of the type. This is by contrast with more strongly typed languages like Modula–2 and Ada, where a new named type is a new type—even if it looks the same as an existing type.

This is also a useful opportunity to introduce the `enum`—roughly equivalent to enumerated types of Pascal, Ada and Modula–2, but with a looser distinction from integer types.

A `typedef` is written in the same order as if a variable of that type were declared, but with the variable name replaced by the new type name:

```
typedef int cents;
```

introduces a new type that is exactly equivalent to `int`, which for reasons of maintainability has been given a new name.

Back to `enums` now that we have the `typedef` mechanism. An `enum` introduces a symbolic name for an integer constant:

```
enum Boolean {FALSE, TRUE};
```

establishes symbolic names for the values `0` and `1`, which can give programmers accustomed to typed languages a greater sense of security. Unfortunately this is not a default in system headers, where `#define` is usually used to define `TRUE` and `FALSE` (using the preprocessor). This can lead to a conflict of incompatible definitions. More on the preprocessor in Part 5.

It is also possible to specify the values of the enum names, for example,

```
enum vowels {A='a', E='e', I='i', O='o', U='u'};
```

Once you have specified values for an initial group of names, if you stop supplying names, the rest continue from the last one specified:

```
enum digit_enum {ZERO='0',ONE,TWO,THREE/*etc.*/};
```

You can now make a variable of one of these types, or better still, make a type so you don't have to keep writing enum:

```
enum digit_enum number = TWO; /*initializer must be a constant*/
typedef enum digit_enum Digits;
Digits a_digit;   /* look: no enum */
```

For a more complicated example of something we'll look at in more detail later in Part 5, here is how to declare a type for a function pointer:

```
typedef int (*Compare) (char*, char *);
```

This declares a type called Compare, which can be used to declare variables or arguments which point to a function returning int, and taking two pointers to char (probably strings) as arguments. More detail in Part 5.

Back to something more immediately useful: to reduce typing, it is common practice to supply a struct with a type name so that instead of writing

```
struct Tree sort_data;
```

it's possible to write

```
typedef Tree Sort_tree;
Sort_tree sort_data;
```

In fact, it is common practice when declaring a struct to give it a type name immediately:

```
struct tree_struct;
typedef struct root_struct
{  struct tree_struct *left, *right;
} Root;

typedef struct tree_struct
{  struct Tree *left, *right;
   char name[100];
   Root *tree_root;
} Tree;
```

From here on, it's possible to declare variables of type Root or Tree, without having to put in the annoying extra word struct.

As we shall see, C++ classes are a much cleaner way of defining structured types.

Some notation for access fields of structs:

```
Root family_tree;
family_tree.left = (Tree*) malloc(sizeof(Tree));
strcpy(family_tree.left->name, "Mum");
(*family_tree.left).left = NULL; /* -> better */
```

Note the use of -> to dereference a pointer to a struct and access a field in one go instead of the more cumbersome (*name).name.

putting it together: array of struct

Using structured types, pointers, and arrays, we can create data of arbitrary complexity.

For example, we can make a mini-employee database using the Employee struct at the start of this Part. If we assume the number of employees is fixed at 10, we can store the whole database in memory in an array.

Using typedefs to clean things up a bit:

```
typedef struct emp_struct
```

```
{  char  *name;
   int   employee_no;
   float salary,  tax_to_date;
}  Employee;

typedef  Employee  Database  [10];
Database  people  = /*initializer:  real  DB  would  read  from  disk*/
   {  {"Fred",  10,  10000,  3000},
      {"Jim",  9,  12000,  3100.5},
      {"Fred",  13,  1000000,  30},
      {"Mary",  11,  170000,  40000},
      {"Judith",  45,  130000,  50000},
      {"Nigel",  10,  5000,  1200},
      {"Trevor",  10,  20000,  6000},
      {"Karen",  10,  120000,  34000},
      {"Marianne",  10,  50000,  12000},
      {"Mildred",  10,  100000,  30000}
   };
```

We'll now use this example for an exercise in putting together a more complicated program.

hands-on—sorting employee records

Starting from the toy employee record database, rewrite the sorting code from the string sort to sort database records instead, sorting in ascending order on the employee name. If any employees have the same name, sort in ascending order of employee number.

part 5—Advanced Topics

preprocessor

It's now time to look in more detail at what `#include` does, and introduce a few more features of the C preprocessor. As mentioned before, the effect of `#include` is to include the named file as if its text had appeared where the `#include` appears.

Two major questions remain to be answered:

- the difference between names enclosed in <> and `""`
- how to avoid including the same header twice if it's used in another header

Usually, <> is for system or library includes, whereas `""` is for your own headers. The reason for this is the order of searching for files: files in <> are searched for among system or library headers first, before looking where the current source file was found, whereas the opposite applies when `""` is used.

> **caution:** the current source file may not be the one you think it is: it's the file containing the `#include` that brought this file in. If that file is a header file that isn't in the directory of the original compilable file, you could end up bringing in a header with the right name but in the wrong directory. In cases of mysterious or bizarre errors, check if your compiler has the option of stopping after preprocessing, so you can examine the preprocessor output

Avoiding including the same header twice is usually a matter of efficiency: since headers shouldn't cause memory to be allocated or code to be generated (declarations not definitions), bringing them in more than once shouldn't matter.

The mechanism for preventing the whole header being compiled a second time requires introducing more features of the preprocessor.

Before the C compiler sees any compilable file, it's passed through the preprocessor. The preprocessor expands `#include`s and *macros*. Also, it decides whether some parts of code should be compiled, and (usually) strips out comments.

Macros and conditional compilation are the key to achieving an include-once effect.

A preprocessor macro is introduced by `#define`, which names it and associates text with the name. If the name appears after that, it's expanded by the preprocessor. If the macro has parameters, they're substituted in. This happens *before* the compiler starts: the macro's text is substituted in as if you replaced it using a text editor.

The next important idea is conditional compilation using `#if` or `#ifdef`, as in

```
#define PC 1  /* PC, a preprocessor symbol, expands as 1 */
#if PC
#include <pc.h>
#else
#include <unix.h>
#endif
```

This sequence defines a preprocessor macro, `PC`, which expands to `1`. Then, the value of `PC` is tested to decide which header to use. You can usually specify preprocessor symbols at compile time, and most compilers have built-in symbols (specifying which compiler it is, whether it's C or C++, etc.).

The usual way of doing this is to use `#ifdef`, which checks if a preprocessor symbol is defined without using its value—or `#ifndef`, which does the opposite. This is typically the way header files are set up so they are only compiled once, even if (through being used in other headers) they may be included more than once:

```
/* File: employees.h */
#ifndef employees_h     /* first thing in the file */
#define employees_h

/* declarations, includes etc. */

#endif   /* last thing in the file */
```

The effect of this is to ensure that for any compilable file importing `employees.h`, it will only be compiled once. Using the name of the file with the "." replaced by an underscore to make it a legal preprocessor name is a common convention, worth adhering to so as to avoid name clashes.

This is not as good as one would like because the preprocessor must still read the whole file to find the `#endif`, so some compilers have mechanisms to specify that a specific include file should only be looked at once.

> **caution:** it seems a good idea to put the test for inclusion around the instead of inside the header. This saves the preprocessor from having to read the whole file—but you end up with a big mess because you must put the test around every `#include`

Here's an example of the other use of macros, to define commonly used text:

```
#define times10(n) 10*n
```

As we'll see later, C++ largely does away with the need for preprocessor macros of this kind by providing inline functions.

function pointers

Function pointers derive from the idea of functions a data type, with variables able to refer to different functions, as long as argument and return types match. Modula–2 also has this feature; Pascal and Ada don't (some extended Pascals do, and you can pass procedures as parameters in Pascal, but not Ada). As we'll see later, C++ classes are a cleaner way of achieving the generality offered by function pointers, but they are important to understand because they're often used in C library and system routines.

To see why this is a useful feature, we'll look at how the sort routine we used in the last two exercises could be generalized to handle both data types we used.

To do this, we need to reconstruct the array we used into an array of pointers, so the sort routine can index the array without knowing the size of each element.

This introduces the need to dynamically allocate memory, which we'll do using the system routine `malloc()`, which had a brief guest appearance in Part 3, where we looked at arrays and pointer arithmetic.

To make `malloc()` return a pointer of the correct type, a type cast (type name in parentheses) of its result is needed. `malloc()` has one argument: the number of bytes required. Special operator `sizeof` (special because it operates on a type) is useful to get the size. If you call it with an expression rather than a type, this expression isn't evaluated: the compiler determines its type and hence its size.

> **caution:** be sure you are asking for `sizeof()` of the object being pointed to, not the size of the pointer, which will usually be way too small; one exception: `sizeof` an array type is the size of the whole array even though an array is normally equivalent to a pointer

Back to the problem: we need a comparison function the sort can use to check the order of two employee records, strings etc. The sort also needs to swap array elements.

Prototypes for functions for this purpose look like this:

```
/* -1 if data[s]<data[t]; 0 if equal, else +1 */
int compare (int *data[], int s, int t);
/* swap two elements in the given array */
void swap (int *data[], int s, int t);
```

You have to supply functions conforming to the types of these prototypes for the sort. Within the functions, the type of `data` can be cast to the required type. Here are `typedef`s for the function pointers (type names `comp_ptr` and `swap_ptr`):

```
typedef int (*comp_ptr)(int *data[],int s,int t);
typedef void (*swap_ptr)(int *data[],int s,int t);
```

Parentheses around the type name are needed to stop the `*` from applying to the return type (making it `int*`).

The sort can then be defined by the prototype:

```
void sort (int *data[], int n, comp_ptr compare,
   swap_ptr swap);
```

and can call the function pointers as follows:

```
if (compare(data, i, j) > 0)
   swap(data, i, j);
```

Older compilers required dereferencing the function pointer, as in

```
if ((*compare)(data, j-1, j) > 0)
```

but newer ones allow the simpler call notation

To sort employee records, you have to supply swap and compare routines:

```
int comp_employee (int *database[], int i, int j);
void swap_employee (int *data[], int i, int j);
/* names of arguments don't have to be the same */
```

and call the sort as follows:

```
sort ((int**)my_data, no_employees, comp_employee,
   swap_employee);
/*arrays are pointers: int**, int*[] same type*/
```

traps and pitfalls

Something not fully explained in previous sections is the C increment and decrement operators. These are in 2 forms: pre- and post-increment (or decrement in both cases).

Preincrement is done before the variable is used: an expression with `++i` in it uses the value of `i+1`, and `i` is updated at the same time. An expression with postincrement `i++` uses the value of `i` before it changes, and afterwards `i`'s value is replaced by `i+1`. Perfectly clear? No. The problem comes in expressions with more than one increment on the same variable. Does "afterwards" (or before in the case of prefix versions) mean after that place in the expression, or after the whole expression?

What is the result of the following, and what value do `i` and `j` have at the end?

```
i = 0;
j = i++ + i++;
```

Under one interpretation, `i` is incremented immediately after its value is used, so the expression for `j` evaluates in the following sequence:

```
i's value 0 used for j; i++ makes i's value 1
i's value 1 used for j: j's final value is 1; i++ makes i's value
2
```

Under another interpretation, i is incremented at the end of the statement:

```
i's value 0 used for j; no i++ yet so i's value stays on 0
i's value 0 used for j: j's final value is 0; both i++ push i to 2
```

caution: if you write code like this, immediately after you are fired the person assigned to maintaining your code after you leave will resign

Here's a start towards the sort code, with a main program at the end:

```
/* file sort.h */

#ifndef sort_h
#define sort_h
typedef int (*comp_ptr) (int *data[], int s, int t);
typedef void (*swap_ptr) (int *data[], int s, int t);

void sort (int *data[], int n, comp_ptr compare, swap_ptr swap);

#endif /* sort_h */
```

```
/* file employee.h */
#ifndef employee_h
#define employee_h

typedef struct emp_struct
{  char name[100];
   int   employee_no;
   float salary, tax_to_date;
} Employee;

typedef Employee *Database[10];

int comp_employee (int *database[], int i, int j);
void swap_employee (int *data[], int i, int j);

/* read in database (for this exercise fake it) */
void init_database (Database employees,
   int no_employees);

/* print out the database */
void print_database (Database people, int no_employees);

#endif /* employee_h */
```

```
/* file main.c */
#include "sort.h"
#include "employee.h"

void main(int argc, char *argv[])
{  const int no_employees = 10;
   Database people;
   init_database (people, no_employees);
   print_database (people, no_employees);
   sort((int**)people,no_employees, comp_employee, swap_employee);
   print_database (people, no_employees);
}
```

part 6—Programming in the Large

The distinction between programming in the small and programming in the large arises from a desire to avoid having implementation detail interfere with understanding how various (possibly separately written) parts of a program fit together and interact.

file structure revisited

Having seen the way the preprocessor works we are now in a better position to look at how multi-file programs are put together.

Header files are the glue used to tie independently compiled files together. These files may be files you are responsible for, library files, or files other programmers are responsible for on a large project.

Unlike some later languages such as Ada or Modula–2, there is no mechanism in C to enforce using equivalent interfaces for all separately compiled files. Also, there is no mechanism in the language to ensure that separately compiled files are recompiled if any header they import is recompiled.

There are tools external to the language to help overcome these limitations.

In UNIX, the `make` program is usually used to rebuild a program: a `makefile` contains a specification of dependencies between the files that are compiled, headers, libraries, etc. Writing a `makefile` is beyond the scope of these notes. However it is worth noting that there is a short cut in most versions of UNIX: the `makedepend` program can be used to create most of the dependencies in the `makefile`. Most PC-based interactive environments automate the "make" process, but command-line driven programming tools often still have this problem. A related problem is that C does not require a type-sensitive linker, so it's possible (e.g. as a result of a bug in your includes) to link files which have an erroneous expectation of function arguments, or types of global data.

C++ goes a long way towards fixing the type-safe linkage problem, so there is a case for using a C++ compiler even for plain C programs.

For large, multi-programmer projects, many platforms have tools to manage access to source files, so only one programmer has write permission on a file at a time. Such tools are not part of C, and can usually be used with other languages and tools available on the system. UNIX's `make` is also general-purpose, not only for C and C++.

maintainability

C—through its lax type checking, permissive file structure and support for low-level hacking—has much potential for producing unmaintainable code.

If include files are used purely as module interfaces, with as few global declarations as possible, module level maintainability can approach that of Modula–2 or Ada.

C allows variables to be declared global to a file, but not visible in other compiled files. Such a variable is declared outside functions, with the keyword `static`. If global state information is needed, it should ideally be restricted to a single file in this way.

For the opposite effect—a globally visible variable—leave out `static`; to make it visible elsewhere, declare it as `extern` in a header imported by files that access it.

An `extern` is a declaration, not definition: it doesn't cause memory allocation.

> **caution:** global variables are reasonably safe within one file but making them global to the entire program is bad for maintainability. Preferably use access functions to read or update the variables from another file if global state *must* be shared across files. You can enforce this convention by always declaring a global using `static`. As noted in part 2, functions can also be `static`

portability

Related to maintainability is portability.

If machine-specific features of your program are isolated to one file, and that file kept a small as possible, portability is not too difficult to achieve.

Some key problems:

- many C programmers assume `ints` are the same size as pointers, and both are 32 bits or 4 bytes; this is causing problems for example in porting UNIX to new processors with 64-bit addresses
- some operating systems have case-sensitive file names. The UNIX file system is case sensitive, while those of DOS, Windows and Macintosh aren't; this can cause problems if you are not careful about typing header file names using all lower-case, and try to move a program to UNIX
- integer types, especially `char` and `int`, can be different sizes even across different compilers on the same machine ; if you rely on their size, use `sizeof` to check they are what you expect (ideally embed this in your code to make it general)
- path name conventions differ—on UNIX, the separator is a "/", on DOS, a "\", on Macintoshes, a ":"; if portability across these operating systems is an issue, it may be useful to separate out `#includes` that must pick up files from a different directory, and put them in a file of their own:

```
/* file data.h */
#ifndef data_h    /* contraction of if !defined(data_h) */
#define data_h
#  ifdef UNIX     /* whatever symbol predefined by your compiler
*/
#     include "../data.h"
#  elif defined(DOS)    /* again */
#     include "..\data.h"
#  else  /* fall through to Macintosh */
#     include "::data.h"
#  endif /* only one endif needed when elif used */
#endif /* data_h */
```

In general: creating one such file for each file that must be found in another directory is a reasonable strategy if you expect to need to port your program to other operating systems. Note my indentation to highlight nesting of the `#ifs`—in general it's bad practice to deeply nest conditional compilation.

hiding the risky parts

Another important point related to both portability and maintainability is avoiding using risky features such as pointer arithmetic and type casts throughout your code. Ideally, they should be isolated into one file as noted before.

As we shall see, C++ offers better mechanisms for hiding details, but a disciplined approach to C can pay off.

Putting together the strategies for portability and maintainability, putting machine-dependent or otherwise potentially troublesome code in only one place is a good start. If this is taken a step further and global variables are never exported (always use `static`), potential trouble can be isolated.

performance *vs.* maintainability

In a performance-critical program, it is tempting to ignore these rules and sprinkle the code liberally with clever tricks to attempt to wring out every last bit of performance.

A good book on algorithm analysis will reveal that this is a futile effort: most programs spend a very large proportion of their time in a very small part of their code. Finding a more efficient algorithm is usually much more worthwhile than hacking at the code and making it unmaintainable.

> **example:** the sort we have been using in our examples takes approximately n^2 operations to sort n data items. A more efficient algorithm, such as *quicksort*, takes roughly $n \log_2 n$ operations. If n=1000, n^2 = 1-million; $n \log_2 n$ is about 100 times less. The detail of quicksort is more complex than our simple sort, so the actual speedup is less than a factor of 100, but you'd do a lot better if you are sorting

1000 items to start with quicksort and optimize it—while still keeping it maintainable—than use all kinds of tricks to speed up our original sort

Also recall the lesson of the optimizing compiler: there are some very good compilers around, and before you attempt to using some of C's less maintainable programming idioms, try the optimizer—or switch to a better compiler.

caution: if you must do strange, hard-to-understand things for good performance, make sure that you really are attacking parts of the code that contribute significantly to run time—and preferably isolate that code to one file, with good documentation to aid maintenance

hands-on—porting a program from UNIX

The following is a simple floating point benchmark written by a student a few years ago for a UNIX platform. See how easily you can get it working on your compiler

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define N 1000

float first[N], second[N], result[N];
int i, j, interations = 1000;
clock_t start, end, elapsed;
void main ()
{ for (i=0; i<N; i++) /* initialize */
   {  first[i] = random();
      second[i] = random();
   }
   start = clock (); /* start timer */
   for (i=0; i<interations ; i++)
     for (j=0; j < N; j++)
        result[j] = first[j] * second[j];
   end = clock ();
   printf ("Timing Ended.\n\n");
   elapsed = end - start;
   printf ("Time : %fs\n",(float)(elapsed)/CLOCKS_PER_SEC);
}
```

Alternatively, if you are working on a UNIX platform, try porting the following which compiles and runs on a Macintosh compiler:

```
#include <time.h>
#include <stdio.h>
/* print current date and time */
void main ()
{ clock_t now;
   struct tm * mac_time;
   char *time_str;
   now = time (NULL); /* time now */
   mac_time = localtime(&now);
   time_str = asctime(mac_time);
   printf ("Now : %s\n", time_str);
}
```

part 7—Object-Oriented Design

identifying objects

The most important element of design is abstraction. Most design methods in some or other form include ways of layering detail, so as little as possible needs be dealt with at a time.

A key aspect of achieving abstraction in an object-oriented design is encapsulating detail in classes. The idea of encapsulation is that a unit of some sort puts a wall around implementation detail, leaving only a public interface visible. This is the general idea called *information hiding*. Where classes are useful is in allowing information hiding to be implemented in a hierarchical fashion, using *inheritance*. Inheritance is the mechanism by which objects are derived from others, sharing attributes they have in common, while adding new ones, or overriding existing ones.

In the design phase, it is useful to try to find objects which can be related by similarity—or differences. Inheritance makes it possible to start with a very general class, and specialize it to specific purposes. If this kind of decomposition can be found, an object-oriented program is relatively easy to construct, and even non-object-oriented languages such as C can benefit from such a design. In particular, an object-oriented design can guide in the use of header files in a disciplined way, as well as discouraging the use of global variables.

For example, in a simulation of *n* bodies interacting through gravitational attraction, it turns out that groups of bodies far away can be grouped together and treated as a single body. Such a clustered body has most of the properties of a normal body, except it contains a list of other bodies which have to be updated at each round of the simulation. A possible decomposition: make a class for ordinary bodies, while extending it for the clustered body.

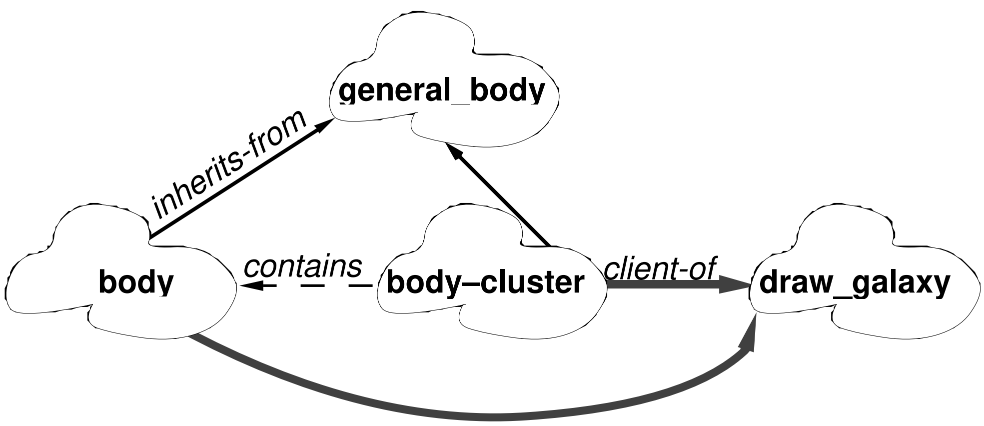
object relationships

Once having identified the major objects and which ones can be described in terms of specialization from a more general one, other relationships can be shown using a suitable notation to distinguish them. Some objects are contained in composite objects, or some are clients of others that provide a service. For example, as illustrated below, a drawing object could provide a service to the bodies in the *n*-body simulation.

Once the first pass of decomposition is complete, the next step is to look in more detail at each object. In terms of the design model, it's necessary to identify *state* (history, or data the object stores), and *behaviours* the object is responsible for. By contrast, most older design models treat procedural and data decomposition separately.

A good strategy as each stage of the design is complete is to implement a toy version of the program with as much detail as has been designed so far; this makes it easier to be sure the design is correct. If any problems are found in implementation, it's possible to go back and correct them at an early stage. This kind of iterative approach is not unique to object-oriented design.

As with other design methodologies, it's important not too fill in too much detail at once: you should proceed from an overall design to designing individual components,



always keeping the level of detail under control so you can understand everything you are working on.

entities *vs.* actions

Some other decomposition techniques emphasize either data or program decomposition. In an object-oriented design, depending on the context, either starting with real-world entities or with actions carried out by them, can be a natural approach.

The *n*-body simulation is an example of an entity-based decomposition.

An example of an action-based decomposition is a event-driven simulation. For example, if a new computer architecture is being designed, it is common to run extensive simulations of programs executing on a simulator of the new design. The simulation is typically run by scheduling *events*, each of which represents an action that would have taken place in a real execution on the still-to-be-built hardware. Examples of events:

- a memory read
- an instruction execution
- an interrupt

Also, such a simulation has a lot of global state which is however only of interest to specific events (e.g., the state of memory is interesting to a memory read).

An object-oriented design based on events helps to focus on the common features across events, and which data is common to which events, leading to an effective approach to decomposition and information hiding.

example: event-driven program

A typical event-driven user interface such as Microsoft Windows or Macintosh responds to events from the user.

Such events are conceptually very similar to events in an event-driven simulation.

The typical style of program for such an interface consists of a loop containing a call to a system event manager to find out if there is an event to process. If there is an event, it's dispatched using a large **case** or `switch` statement (with a case for each type of event the program handles).

In terms of object-oriented design, events which have features in common should be grouped together and common parts used as a basis for designing a higher-level abstraction. Specific events are then derived from these general ones. Attention can also be paid to which events need which parts of the global state of an application. A similar approach is needed to define entities, such as documents, fonts, etc. Finally, interactions between different types of events must be defined, and related to entities.

An event-driven application is interesting as an example where insisting on doing the entire design using either entity-based or action-based decomposition is not helpful. Entities such as documents are useful to model as objects. In fact the application can be though of as breaking down into two major components: views and manipulators. Views are such things as documents (in both their internal representation and their display on the screen), and contents of clipboards and inter-application sharing mechanisms. On the other hand, as we have already seen, events—which are a kind of action—are also useful to model. A natural description of the behaviour of the application is in terms of interaction between entities and actions.

design task—simple event-driven user interface

For purposes of this design we will restrict the problem to user events consisting purely of mouse downs (only one button), keystrokes, update and null events (sent at regular intervals if the user does nothing).

Rather than look at a specific application, the design is for a generic framework, in which the detail of each kind of event handler is left out.

The design should include: mouse down in a window (front most or not), mouse down in a menu item, with consequences: printing the front most document, saving the front most document, quitting or other action added by the application.

An update event results in redrawing a window.

Think about the global state that's needed, which objects can be related, and where to hide data.

An idea to consider: have an internal representation of the application data, which can be displayed via various *views*: this is a useful abstraction for sharing code between printing and drawing windows

part 8—OOD and C

language elements

We have already seen the features of C we need to translate an object-oriented design to code. To encapsulate data, we can use a `struct`, though it does not enforce information hiding. To include actions on the data, we can store function pointers in the `struct`.

To implement inheritance is more of a problem, since there is no mechanism in C to add new data fields to a struct. Nonetheless, we can get quite a fair part of the way without this, since the ability to change function pointers at least allows behaviour to be changed. To extend a data representation though there's no mechanism in the language: we can use an editor to duplicate common parts of `struct`s, and use type casts to allow pointers to point to more than one type of `struct`—at the risk of errors not detectable by the compiler. Another option—not recommended—is to become a preprocessor macro hacker.

One additional language feature can in principle be used as well: the ability to define `static` data within a function. The effect of this is to preserve the value of the data between function calls (this is really the same as a `static` variable at file scope, except it's only visible inside the function). This feature would allow us to store information global to a "class" (not stored with each object). However, there is a problem: since we can change any of the functions stored in the function pointers, we would have to be careful to keep track of such global-to-the-class data as a static in a function when we change a function pointer.

This last feature is not necessary in the example to follow.

example

The example follows from the design developed in Part 7.

Let's consider the relationship between just two parts of the specification: printing and redrawing windows.

These both involve essentially similar operations: iterating over the internal representation and rendering a view. Therefore it makes sense to have a general approach to rendering a view that can be extended to handle either case, with as much re-use between the two cases as possible.

We can make a general `View` class which renders by iterating over a representation. For our purposes, we will fake the rendering action to keep the example simple. In C, this can all be implemented along the following lines:

```
#include <stdio.h>
typedef struct view_struct
{  char data[100];   /* real thing would have something useful */
} View;

/* function pointer type */
typedef void (*Renderer) (View current_view);

typedef struct render_struct
{  Renderer render_function;
} Render;

void Renderer_print (View current_view)
{  printf("printing\n");
}

void Renderer_window (View current_view)
{  printf("redraw\n");
}
```

A `View` could call one of these as follows (highly simplified to illustrate the principles):

```
Render *print_view = (Render*) malloc (sizeof(Render));
Render *window_view = (Render*) malloc (sizeof(Render));
```

```
View  with_a_room;
print_view->render_function  =  Renderer_print;
window_view->render_function  =  Renderer_window;
print_view->render_function(with_a_room);
```

Of course the actual code would be much more complex, since the `View` would have to contain a detailed representation of the objects to render, and the renderers would have to have detailed code for drawing or printing.

group session: finalize the design

Having seen how part of the design could be implemented in C, refine the design making sure you have a clean abstraction.

To avoid too many complications in implementation, make sure your object hierarchy is not very deep, and try not to have too much variation in data representation (state) between objects in the same hierarchy. Also try to define a small subset of the design which you can be reasonably sure of implementing in an afternoon

hands-on—implementation

Now implement the design in C. For simplicity, rather than the actual event generation mechanism, whenever the program requires another event, read a `char` from the keyboard:

```
char  new_event;
new_event  =  getchar();
```

Take ‘q’ as quit, ‘p’ as print, ‘r’ as redraw, ‘k’ as keydown and ‘m’ as mousedown.

Doing this will save the effort of implementing an event queue (needed in the real thing since many events are generated as interrupts, outside the control of the user program)—your toy program will consume the event as soon as it’s generated

part 9—Object-Oriented Design and C++

OOD summary

Object-oriented design requires information hiding through encapsulation in objects and sharing common features through inheritance.

In a language like C, some aspects of the design can be implemented directly, while others can be implemented through programmer discipline.

Inheritance in particular would be useful to implement with compiler support.

objects in C++

In C++, a new concept is introduced: the `class`. For compatibility with C, a `struct` can be used as a class, with some minor differences. From now on, however, we shall use classes since the syntax is more convenient.

Classes support information hiding, and allow encapsulation of data with functions that operate on the data. Inheritance is supported, as is redefining built-in operators for new classes. Data and functions in classes are called *members*. Functions—including member functions—can be *overloaded*, i.e., it's possible to have more than one function with the same name as long as the compiler can tell them apart by the types of their arguments, or the class of which they are a member. Overloaded functions may *not* be distinguished by the type they return.

A simple class declaration looks like this:

```
class Point
{public:
   Point (int new_x, int new_y);
   ~Point ();
   void draw ();
private:
   int x, y;
};
```

This defines a new type, `class Point`. Unlike with C `structs`, it isn't necessary to use the word `class` when declaring or defining a variable of the new type, so there's no need to do a `typedef` to give a class a single-word name. The keyword `public:` means following members are visible outside the class. The member with the same name as the class, `Point()`, is a *constructor*, which is called when a new variable is created, by a definition, or after allocation through a pointer by the `new` operator. `~Point()` is a *destructor*, which is automatically called when a variable goes out of scope, or if allocated through a pointer, the `delete` operator is called on it.

Keyword `private:` is used to make following members invisible to the rest of the program, even classes derived from `Point`. Parts of a class can be made accessible only to other classes derived from it by preceding them with `protected:`.

Here is an example of two definitions of points, with their position:

```
Point origin(0,0), of_no_return(1000000,1000000);
```

When to use `private:`, `public:` and `protected:`? Only member functions that are part of the external specification—or interface—of the class should be made public. These usually include the constructor and destructor (which can be overloaded with different arguments), access functions to allow setting the internal state or reading it, without making the internal representation known, and operations on the class.

Private members should be anything else, except secrets of the class that are shared with derived classes. When in doubt, use private—this isolates information to one class, reducing possibilities for things to go wrong.

> **caution:** C++ allows you to break the principle of encapsulation. Don't be tempted. Making data members public is a sure way of writing unmaintainable code. Instead, use access functions if other classes need to see the data; this can be done efficiently using `inlines`—see "differences from C" below

The interaction between storage allocation and constructors is important to understand. When a new instance of a class is created (either as a result of a definition or of a `new`), initially all that is created is raw uninitialized memory. Then, the constructor is called—and only after the constructor finishes executing can the object properly be said to exist.

Another key feature of C++ classes is *inheritance*. A *derived* class is defined by extension from a *base* class (in fact possibly more than one through *multiple inheritance*).

The notation is illustrated with a simple example:

```
class Shape
{public:
   Shape (Point new_origin);
   ~Shape ();
   virtual void draw () = 0;
private:
   Point origin;
};

class Circle : public Shape
{public:
   Circle (Point new_origin, int new_radius);
   ~Circle ();
   virtual void draw ();
private:
   int radius, area;
};
```

A few points (we'll save the detail for Part 10): class `Circle` is declared as a public derived class of `Shape`: that means the public and protected members of `Shape` are also public or protected respectively in `Circle`. In a private derived class, public and protected members of its base class become private, i.e., they can't be seen by any derived class (unless given explicit permission with a `friend` declaration). A *virtual* function is one that can be replaced dynamically if a pointer to a specific class in fact points to one of its derived classes. If a class has virtual functions, it has a virtual function table to look up the correct member function. The line

```
virtual void draw () = 0;
```

in class `Shape` means `Shape` is a *abstract class*, and creating objects of that class is an error that the compiler should trap. In such a case, the function is called a *pure virtual function*. Only classes that are derived from it that define `draw` (or are derived from others that define `draw`) may be instantiated. (This is a bit like having a function pointer in a C `struct`, and setting it to the `NULL` pointer—but in such a case the C compiler won't detect the error of trying to call a function through a `NULL` pointer.)

Since `Circle` is not an abstract class, we can do the following:

```
Circle letter(origin, 100);
Shape *i_m_in = new Circle(of_no_return, 22);
i_m_in->draw (); /* calls Circle::draw */
```

The double-colon is the C++ *scope* operator, is used to qualify a member explicitly if it isn't clear which class it belongs to, or to force it to belong to a specific class. Notice how member function `draw()` is called through an object: in a member function, the current object can be accessed through a pointer called `this` (you seldom need to use `this` explicitly: a member function can use members of its class directly).

Note that the compiler should complain if you try something like:

```
Shape blob(origin);
```

with a message along the lines of

```
Error: cannot create instance of abstract class 'Shape'
```

stream I/O

One additional point is worth explaining now: input and output using the `iostream` library. It defines three widely used standard streams: `cout`, `cin` and `cerr`, based on the UNIX convention of standard out, standard in and standard error. Output to a stream is strung together with the << operator, while input is strung together with >> (also used—as in C—respectively, as left and right shift bit operators).

Use of the default streams (C++ has end-of-line comments, started by //):

```
#include <iostream.h>
void main ()
{  cout << "Enter a number : ";           // no line break
   cin >> i;
   cerr << "Number out of range" << endl; // endl ends line
}  /* can also use C-style comment */
```

if you need to use files, use

```
#include <fstream.h>
```

A stream can be associated with a file for output (`ios::out` is an `enum` value):

```
ofstream my_out ("file.txt", ios::out);
```

and used to write to the file:

```
my_out << "A line of text ended by a number : " << 100 << endl;
```

To read the file:

```
ifstream my_in ("file.txt", ios::in);
char data[100];
my_in >> data;
```

You can also explicitly open the file, if you didn't connect the `ifstream` or `ofstream` object to a file when you defined it:

```
#include <stdlib.h>  /* for exit () */
ifstream my_in;
my_in.open ("file.txt", ios::in);
if (!my_in)
{  cerr << "open failed" << endl;
   exit (-1);     // kill program returning error code
}
// use the file ... then finally:
my_in.close ();
```

If you need to do both input and output on a file, declare it as class `fstream`; open with `ios::in|ios::out` which combines the two modes using a bitwise **or**.

> **caution:** I/O is one of the most system-dependent features of any language. Streams should work on any C++ but file names are system-specific (.e.g., DOS's "\" path separator, *vs.* UNIX's "/")

differences from C

Classes, aside from supporting object-oriented programming, are a major step towards taking types seriously. Some see C++ as a better C—if you use a C++ compiler on C code and fix everything it doesn't like the chances are you will unearth many bugs.

Classes bring C++ much closer to having types as in a modern language such as Ada or Modula–2, while adding features such as inheritance that both languages lack.

Classes, templates (done briefly in Part 14) and inline functions reduce the need to define arcane code using the preprocessor. Here's an example of an inline function:

```
inline int times10 (int n)
{  return 10 * n;
}
```

The `inline` directive asks the compiler to attempt to substitute the function in directly, rather than to generate all the overhead of a procedure call. The following two lines should cause the same code to be generated:

```
a = times10 (b);
a = 10 * b;
```

It’s particularly useful to use an `inline` as an access function (e.g. for private members of a class). Compilers don’t always honour an `inline`: it’s only a directive. Early compilers didn’t inline complex functions where the procedure call overhead was minor compared with the function; this is less common now.

Since an `inline` doesn’t generate code unless (and wherever) it’s used, it should be in a header file. I prefer to keep inlines separate from the `.h` file, since they are implementation detail rather than part of the interface of a class. My file structure is

File `part_of_program.h`: contains classes, ends with

```
#include "part_of_program.inl"
```

File `part_of_program.inl`: contains inlines
File `part_of_program.c++`: starts with

```
#include "part_of_program.h"
```

Make sure the compiler has seen the `inline` directive before you call the function. This can be a problem if an inline calls another, if you use my file strategy. One fix is to put a prototype of the problem function at the top of the file, with an `inline` directive. Another is to put the `inline` directive into the class declaration, but this is bad practice. Inlining is implementation detail, better not made into part of the class’s interface.

Another bad practice: you can put the body of a member function in the class declaration, in which case it’s assumed to be inlined. This again puts part of the implementation into the publicly advertised interface of the class.

caution: if an inline is called before the compiler sees it’s an inline, it generates a normal call. This causes an error when the compiler sees the `inline` directive, or later when the linker tries to find non-existent code (the compiler doesn’t generate code for an inline, but substitutes it in directly). If you get link errors, this is one thing to check

C++ requires type-safe linking. This isn’t as good as it could be since it assumes a UNIX-style linker, with no type information from the compiler. Instead, *name mangling* is implemented by most C++ compilers: argument types of functions and their class are added to the name, so overloaded versions of the function can be distinguished, and only functions with correct argument types are linked. This is not guaranteed to find all errors; you still need to be sure to use consistent header files, and to use a mechanism such as `make` to force recompilation when necessary.

One other significant addition to C is the reference type, `typename&`. This differs from a pointer in that you don’t have to explicitly dereference it—the compiler generates pointer manipulation for you. Reference types can be used as arguments in function calls, with the same effect as Pascal **var** parameters:

```
void swap (int &a, int &b)
{  int temp;
   temp = a;
   a = b;
   b = temp;
}
//usage (correct, unlike Part 2 example):
swap (first, second);
```

In UNIX, C++ compilable files usually end in `.C` or `.c++`. PC compilers use `.cpp` more often. Headers usually end in `.h`, though `.hpp` is sometimes used on PCs

hands-on—simple example

Don't worry too much about the detail of the below code—we'll look at detail in the Part 10. For now, concentrate on how classes are used: fill in the main program as suggested, and see what output results

```
#include <iostream.h>
#include <string.h>

class View
{public:
   View (const char new_name[]);
   ~View ();
private:
   char name[100];   /* real thing would have something useful */
};

class Render
{public:
   virtual void draw (View *to_draw) = 0;
};

class Print : public Render
{public:
   virtual void draw (View *to_draw);
};

class Update : public Render
{public:
   virtual void draw (View *to_draw);
};

void Print::draw (View *to_draw)
{  cout << "Print" << endl;
}

void Update::draw (View *to_draw)
{  cout << "Update" << endl;
}

View::View (const char new_name[])
{  strncpy(name, new_name, sizeof(name)-1); // copy max. 99 chars
}

void main()
{  View *window = new View ("window");
   Render *renderer;
   renderer = new Print;
   renderer->draw (window);
   // based on the above, create an object of class Update
   // by replacing the Word Print by Update in the last 3
   // lines - now try to relate this to the object-oriented design
   //  exercise
}
```

part 10—Classes in More Detail

constructors and destructors

A constructor is a special function that is called automatically. As defined, it does not return a value (i.e., doesn't contain a statement `return expression;`). Its effect is to turn uninitialized memory into an object through a combination of compiler-generated code and the code you write if you supply your own constructor. If you don't supply a constructor, compiler supplies one—a *default* constructor, with no arguments. The additional code the compiler generates for your own constructor is calls to default constructors for objects within the object which don't have their own constructor, and code to set up the virtual function table (if necessary).

Here is an example of a constructor:

```
Shape::Shape (Point new_origin) : origin(new_origin)
{
}
```

The first thing that's interesting is the use of the scope operator `::` to tell the compiler that this is a member of the class `Shape` (through writing `Shape::`). Then there's the initializer for `origin` after the colon. Why could we not do the following?

```
Shape::Shape (Point new_origin)
{  origin = new_origin;
}
```

This is not allowed because `origin` hasn't had a constructor called on it. Look back at class `Point`. The only constructor defined for `Point` has to take two `int` arguments. Once we define a constructor, the default parameterless constructor is no longer available (unless we explicitly put one in). However there is another compiler-supplied constructor, the *copy* constructor, which allows you to initialize an object from another. That's what `origin(new_origin)` does. Initializers for contained classes follow a colon after the arguments, separated by commas. `Shape` is very simple, so there's very little for the constructor to do, so here's a more interesting example:

```
Circle::Circle (Point new_origin, int new_radius) :
   Shape (new_origin), radius (new_radius)
{   area = PI*radius *radius; // PI usually in <math.h>
}
```

`area` could also be initialized in the header, but it's easier to read if initialization that doesn't require its own constructor is in the body of the constructor. I would usually write the above as follows, with only the base class initialized in the heading:

```
Circle::Circle (Point new_origin, int new_radius) :
   Shape (new_origin)
{  radius = new_radius;
   area = PI*radius *radius;
}
```

Any base classes that have a default constructor need not have the constructor explicitly called, but if you want to pass parameters to the constructor of a base class, it must be done using this mechanism.

Another thing to notice: once you've passed the scope operator in `Circle::`, you can refer to class members without further qualification.

A destructor is only needed if some kind of global state needs to be undone. For example, if an object contains pointers, the constructor will probably allocate memory for them (though this could happen later, e.g., if the object is the root of a tree or node of a list), and the destructor should see to it that it is deallocated—otherwise the memory is not reclaimed, resulting in a *memory leak*. (In general a memory leak is a gradual disappearance of available memory through failing to reclaim memory from pointers that are either no longer active, or have had their value changed.)

Destructors are automatically called in the correct order if an object is of a derived class. The destructor for `Circle` is empty and could have been left out, but since it's in the class specification we must define it:

```
Circle::~Circle ()
```

```
{
}
```

inheritance and virtual functions

Consider the following:

```
void Circle::draw ()
{
   // call some system routine to draw a circle
}

Shape *i_m_in = new Circle(of_no_return, 22);
i_m_in->draw ();
```

Where `draw()` is called, the compiler generates code to find the correct function in the *virtual function table* stored with object `*i_m_in`, and calls `Circle::draw()`.

What happens to virtual function calls in a constructor? Until all constructors terminate (the current one may have been called by another constructor), the virtual function table may not be set up. To be safe, the virtual function is called like a regular member function—based on the class currently under construction.

On the other hand, destructors can both be virtual and can call virtual functions with the expected semantics: the compiler ensures that the virtual function table is valid.

information hiding

Classes are a much more robust way of hiding information than C's limited capabilities of hiding local variables inside functions and hiding names at file scope by making them `static`. These mechanisms are still available in C++, but with the additional machinery of the class mechanism, names visible to a whole file are seldom necessary.

Furthermore, low-level detail can be hidden by putting it in a base class. The machine-specific parts of the class can be made private, preventing direct access to them by even classes derived from them.

We shall now look at a more robust mechanism for hiding global state than C's file-scope `statics`.

static members

If a member of a class has the word `static` before it in the class declaration, it means there is only one instance of it for the whole class in the case of a data member, or in the case of a function, that it can be called without having to go through an object.

For example, to extend the class for shapes, it would be useful to have a global count of all shapes that have been created. To do this, we need a count that is stored only once, not for every shape, and a function to look up the count (because we don't make data members public):

```
// in the header file:
class Shape
{public:
   Shape (Point new_origin);
   ~Shape ();
   virtual void draw () = 0;
   static int get_count ();
private:
   Point origin;
   static int count;
};
```

```
// in the compilable file:

int Shape::count(0); // could also use "count = 0"

int Shape::get_count ()
{  return count;
}
```

The line `int Shape::count(0);` is needed because a static member must be defined. A class declaration is like a `typedef` in the sense that it doesn't cause memory to be allocated. Non-static members have memory allocated for them when a variable of the class is defined, or operator `new` is used, but static data members must be explicitly defined in this way. We must also change the constructor and destructor to keep the count (since `count` is private, this can only be done by Shape):

```
Shape::Shape (Point new_origin) : origin(new_origin)
{  count++;
}

Shape::~Shape ()
{  count--;
}
```

Now, whenever a new object of any class derived from `Shape` is created, `count` is incremented, and decremented whenever an object of such a class ceases to exist. You can look up the count as follows:

```
#include <iostream.h>
// assume the above classes etc.
// can leave out arguments to main if not used
void main()
{  Circle letter(origin, 100), round(letter);
   Shape *i_m_in = new Circle(of_no_return, 22);
   cout << "Number of shapes is " << Shape::get_count () << endl;
}
```

which results in the output

```
Number of shapes is 2
```

hands-on—adding to a class

Define a class `Double_circle` which contains an offset and its `draw()` calls `Circle::draw()` twice—once with the radius increased by the offset—and use it in a simple test program (you may fake drawing by writing out the radius). Use the `Circle` class defined in this section, adding the missing part of `draw()`

Hint: set the radius to its new value before calling `Circle::draw()` the second time, and reset it afterwards

part 11—style and idioms

access functions

It's generally bad style to put data members into the public interface of a class. To do so is bad for maintenance, and is not much better than unrestricted use of global variables.

It's much better practice, if parts of the data of a class need to be accessed elsewhere, to use access functions. If the representation is changed later, only the access functions need change, not every place in the code where the data is accessed. Also, the places where values change are easier to find, which makes for better maintainability. For example:

```
// shapes.h
#ifndef shapes_h
#define shapes_h
class Circle : public Shape   // class Shape as before
{public:
   Circle (Point new_origin, int new_radius);
   ~Circle ();
   virtual void draw ();
   int get_radius ();
   int get_area ();
   void put_radius (int new_radius);
private:
   int radius, area;
};
#include "shapes.inl"
#endif /* shapes_h */
```

```
// shapes.inl
inline int Circle::get_radius ()
{  return radius;
}
inline int Circle::get_area ()
{  return area;
}
inline void Circle::put_radius (int new_radius)
{  radius = new_radius;
   area = PI*radius*radius;
}
```

```
// file main.c++
#include "shapes.h"
void main()
{  Circle letter(origin, 100);
   letter.put_radius (200);
}
```

protected *vs.* private

This has been mentioned before but is worth repeating. As much of a class as possible should be private. Private members are hidden even from derived classes. Protected is better than public, but derived classes can still see protected members.

This relationship applies if a derived class is derived publicly, as in our examples. If a class is derived privately, no names are visible in the derived class, for example:

```
class Circle : private Shape // etc.
```

usage of constructors

Constructors are mostly automatically invoked. There are a few points about constructors that are important to understand, aside from those mentioned already.

An array of objects can only be defined using default (parameterless) constructors. If you have an array declaration, you'll get a compiler error if you defined constructors with arguments, and haven't supplied a default constructor (only if you supply no constructor is a default constructor supplied by the compiler).

If you have common code across several constructors, or want to re-initialize an object, can you call a constructor explicitly? No. You have to separate out the initialization code you want to re-use into an ordinary member function.

In C, a type cast looks like this:

```
(int*) string_var;   /* turn array of char into pointer to int */
```

In C++, you can do the same thing. But you can also define type conversions by defining a constructor taking a given type as an argument:

```
#include <string.h>
class P_string
{public:
   P_string ();
   P_string (const char *c_string);
private:
   char p_string[256];
};
P_string::P_string ()
{
}
P_string::P_string (const char *c_string)
{  int length = strlen(c_string);
   char *start = &p_string[1];       // &p_string[1] is its address
   if (length >= 256)
      p_string[0] = 255;             // need 1 byte for length
   else
      p_string[0] = length;
   strncpy(start, c_string, 255);    // copy at most 255 chars
}
P_string dialog_name = P_string ("Save File");  // for example
```

Constructor `P_string("Save File")` does type conversion from null character-terminated C string to Pascal string (first byte is the length—common on Mac and PC).

A very useful property of a destructor: it's automatically called for non-pointer variables, even if you `return` from a function, or `break` from a loop or `switch`.

The following is useful to ensure that bracketing operations are properly paired (e.g., set the graphics state, do something, restore it to its previous state):

```
class State
{public:
    State ();
    ~State ();
private:
   State_data state_info;
};
State::State ()
{  // save previous state, then set new state
}
State::~State ()
{  //reset to stored state
}
void update_window ()
{  State graphics_state;
   // do stuff to the window
   if (some_condition)
      return;
}
```

The destructor is called in *two* places: before the `return`, and before the final `}`. If you used explicit calls to save and restore state, it would be easy to forget the restore in one of these cases.

hands-on—implementing a simple design

Choose a manageable part of your event-driven interface design, and implement it in C++, using classes. You should find it a lot easier than trying to implement an object-oriented design in C. In particular, a deep hierarchy with many additions to the state at each level should no longer present a big problem

part 12—Advanced Features

mixing C and C++

Sometimes it's useful to link separately compiled C code with a C++ program. You are likely to run into problems if the main program is not written in C++, because most C++ compilers insert initialization code into the main program. Otherwise, the major problem to overcome is the expectation of the C++ compiler that type-safe linking is used. The `extern "C"` mechanism is supplied to solve this problem:

```
extern "C"
{  // {} needed only if more than 1 declaration
#include "sort.h"
}
```

Everything bracketed this way is exempt from type-safe linkage (i.e., names aren't mangled—the mechanism for type-safe linking). The C++ compiler generates calls to the functions declared in `sort.h` without mangling names.

overloading operators

One of the more advanced features of C++, also found in a few recent languages such as Ada (and some older ones like Algol–68), is the ability to define new behaviours for built-in operators.

In C++, aside from obvious operator symbols such as `+`, `-`, etc., some other things are operators, including assignment (`=`) `new`, `delete`, and array indexing (`[]`).

For example, if you do not like the limitations of built-in array indexing and want to define your own, you can create a class containing the array data and indexing operations of your own design. One reason to do this: as with many other languages, C++ is limited as to its support for freely specifying all dimensions of a multi-dimensional array at run-time. The reason for this is that the conventional array indexing operation needs to multiply by all but the last dimension to find the actual place in memory that an array element occupies.

With C's capability of using pointers and arrays interchangeably, this problem can usually be worked around by implementing multi-dimensional arrays as arrays of pointers. C++'s class mechanism provides a cleaner way of hiding the detail of this, allowing you to use code that looks like an ordinary array indexing operation once you have worked out the detail of your array class's index operation.

Here is how you could declare such an indexing operation (we shall extend this to a 3-dimensional array class in the next hands-on session):

```
class Array1D
{public:
   Array1D (int new_max);
   ~Array1D ();
   int& operator [] (int ind);
private:
   int *data;
   int max;
};
```

Note the `&`: it specifies that the return type of the operator is a *reference* to `int`, which means that it is effectively a pointer to the actual data item. However the compiler automatically dereferences the pointer as necessary. The reason for doing this is to make it possible to use the index operator on the left-hand-side of an assignment, as in

```
Array1D scores(100); //not []: 100 is constructor arg
int i;
for (i = 0; i < 100; i++)
   scores[i] = 0;
```

Here is how `operator[]` could be defined:

```
int& Array1D::operator [] (int ind)
{
#ifdef BOUNDS_CHECK
```

```
    if ((ind<0)||(ind>=max))
        ; // insert error message
    else
#endif /* BOUNDS_CHECK */
        return data[ind];
}
```

The constructor has to allocate `data`. I've put in an option of checking the index against the array bounds, which usually isn't available in C or C++ (an array is a pointer, so the compiler may not know the bounds: the "array" could be an arbitrary piece of memory). If you don't want bounds checking you can compile without defining `BOUNDS_CHECK`. To improve performance, you can inline the operator—in which case it's no less efficient than the usual `operator[]`, but with the option of bounds checking.

memory management

It's also possible to redefine the built-in operators `new` and `delete`. This is useful because the standard strategy for memory allocation may not always be efficient enough for every application.

For example, I once encountered a situation where someone was reading a large amount of data off disk, sorting it in memory then writing it back. The data was too large to fit in memory, but he relied on the operating system's virtual memory to allow him to get away with this. While the sort was running, he noticed the disk was constantly busy, indicating that there was a very high number of page faults. He re-examined his sorting strategy, which should have been very efficient:

```
1. divide the possible keys on which the sort is being done
   into a number of buckets
2. read the data sequentially, putting each item directly into
   the right bucket (quick if you know the range of key values)
3. write out the buckets sequentially to disk
```

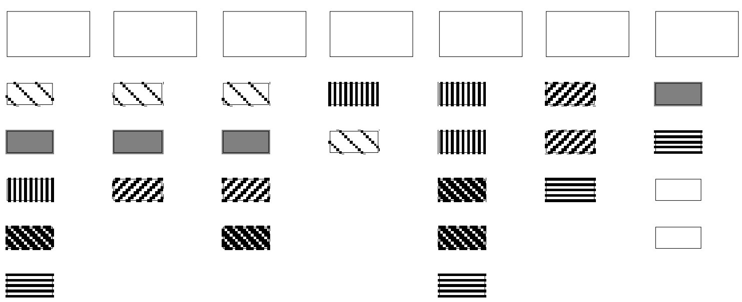
What was happening was his memory allocator was allocating data in the order it was read from disk, so by step 3, data in each bucket was scattered all over memory. The figure below illustrates the problem.

Once he realized this was the problem, he wrote his own memory allocator that allocated a large chunk of memory for each bucket, and when a new data item was added to a bucket, it was given memory allocated for the bucket.

The result? A 100-fold speedup.

Here is how you can write your own versions of `new` and `delete`:

```
#include <stddef.h>  /* sometimes needed to overload new */
#include <new.h>     /* usually needed to overload new */
class Bucket_data;   // can now use Bucket_data*
class Bucket
{public:
    void add_data (Bucket_data *new_data);
    Bucket_data *get_new ();
    void recycle_old (Bucket_data *old_data);
private:
    Data_list data, free_list; // some detail to work out
```



big boxes are buckets; smaller boxes are shaded to show order of arrival of bucket contents

```
};
class Bucket_data
{public:
   Bucket_data (Bucket *new_owner);
   static void* operator new (size_t size, Bucket &owner);
   static void operator delete (void *old_data, size_t size);
private:
   Bucket *owner;
};
```

An implementation could look like this:

```
void* Bucket_data :: operator new (size_t size, Bucket &owner)
{  return owner.get_new ();
}
void Bucket_data::operator delete (void *old_data, size_t size)
{  ((Bucket_data*) old_data)->owner->
      recycle_old((Bucket_data*)old_data);
   old_data = NULL;
}
```

Usage:

```
Bucket_data *new_data = new (pail) Bucket_data (&pail);
```

The first (pail) is the last argument to operator new (the compiler automatically puts in the size), and the later (&pail)—the & makes a pointer to pail—is passed to the constructor, Bucket_data (Bucket *new_owner).

> Exercise: fill in the detail. Bucket::get_new () should use ::new to grab a large chunk of memory when it runs out, otherwise just return the next piece of what's left of the chunk

multiple inheritance

Sometimes it's useful to base a class on more than one other class. For example, we would like to add a capability of printing error messages to our buckets, with a default message for each bucket. This is a useful capability to add to other things, so let's create a separate Error class and make a new class built up out of it and Bucket:

```
class Error
{public:
   Error (const char* new_message);
   void print_err (const char* message = "none");
private:
   char *default_message;
};

class Error_bucket : public Bucket, public Error
{public:
   Error_bucket (const char* new_default = "Hole in bucket");
};
```

Note the default argument in the Error_bucket constructor: if an Error_bucket is created with no argument for the constructor, it's as if the argument had actually been "Hole in bucket". The constructors and implementation are straightforward:

```
Error::Error (const char* new_message)
{  default_message = (char*)new_message;
}

// no constructor call for Bucket: has default constructor
Error_bucket::Error_bucket (const char* new_default) :
   Error (new_default)
{
}

void Error::print_err (const char* message)
```

```
{  if (strcmp(message,"none") == 0)
      cerr << default_message << endl;
   else
      cerr << message << endl;
}
```

The following:

```
   Error_bucket beyond_pale, holy_bucket ("Leaky");
   beyond_pale.print_err ();
   holy_bucket.print_err ();
   holy_bucket.print_err ("Fixed");
```

results in this output:

```
Hole in bucket
Leaky
Fixed
```

LISP programmers call little classes designed to be added to new classes *mixins*.

cloning

Sometimes it's useful to be able to make a new object based on an existing one, without knowing what class the original is.

One way of doing this is to define a `clone()` member function:

```
class Bucket
{public:
   void add_data (Bucket_data *new_data);
   Bucket_data *get_new ();
   void recycle_old (Bucket_data *old_data);
   virtual Bucket* clone ();
private:
   Data_list data, free_list;
};
class Error_bucket : public Bucket, public Error
{public:
   Error_bucket (const char* new_default = "Hole in bucket");
   virtual Bucket* clone ();
};
```

The two versions of clone look like this:

```
Bucket* Bucket :: clone ()
{  return new Bucket(*this);  // note use of copy constructor
}

Bucket* Error_bucket :: clone ()
{  return (Bucket*) new Error_bucket(*this);
}
```

And a call like this:

```
Bucket *kicked = new Error_bucket("kicked over"), *spilt;
spilt = kicked->clone ();
```

would result in a new the creation of a new object of class `Error_bucket`, copied from the object pointed to by `kicked`.

There are variations on cloning: *deep* cloning doesn't copy any pointers, but always makes a completely new object, including allocating new memory and copying any contained objects; shallow cloning only copies the outermost level, which may mean more than one pointer is pointing to the same piece of memory (called an *alias*).

> **caution:** there's no direct way in C++ to force a member function to be redefined for every derived class. It's easy to forget to redefine the `clone()` virtual function in a class and clone the wrong type of object. Use cloning with care, and not for deep class hierarchies

double caution: an alias is bad news—one thing it can result in for example is calling `delete` more than once on the same piece of memory with probably disastrous consequences on the internal state of memory allocation/deallocation

hands-on—3-D array class

Let’s put some of these ideas together now, and define a 3-dimensional array class, capable of storing objects of any class in a hierarchy that has a clone member function. The following is a start:

```
// 3-dimensional array - size set at allocation, check bounds
// #ifdef BOUNDS_CHECK, each dimension indexed 0..initial_max-1
// supply example of object to clone for elements if all the
// same class, otherwise the elements initialized as NULL

class Array2D;
class Array3D;

class Array1D
{public:
   Array1D (int new_max, Bucket *example);
   ~Array1D ();
   Bucket*& operator [] (int ind);
   friend class Array2D;   // let 2D see 1D's private members
private:
   Bucket **data;
   int max;
};

class Array2D
{public:
   Array2D (int new_max_y, int new_max_z, Bucket *example);
   ~Array2D ();
   Array1D& operator [] (int ind);
   friend class Array3D;
private:
   int get_z ();  // 3D can't see Array1D's max
   Array1D **rows;
   int max;
};

class Array3D
{public:
   // supply example to clone from if all to be same type and
   // allocated when array is allocated
   Array3D (int new_max_x, int new_max_y, int new_max_z,
      Bucket *example = NULL);
   ~Array3D ();
   Array2D& operator [] (int ind);
   int get_max_x ();
   int get_max_y ();
   int get_max_z ();
private:
   Array2D **planes;
   int max;
};
```

part 13—Design Trade-Offs

case study—vector class

Another useful class in many applications is one for vectors, including vector arithmetic, such as addition. To keep things simple, we'll stick to a vector of three dimensions, and only look at a small number of possible operations.

defining operators *vs.* functions

The ability to define your own overloaded versions of built-in operations in C++ makes it tempting to always use them when the possibility arises. However, this can sometimes lead to complications, especially the temporary problem described below. However, before going into problems, here is an example of defining a simple vector operation, -= as both a function and an operator. The operator negates its argument and returns a reference to it, so the expression could appear on the left-hand-side of an assignment.

```
const int n_dim = 3;

class Vector
{public: // constructor sets all to zero if no args
     Vector (float first=0.0, float second=0.0, float third=0.0);
     Vector& operator -=(float scalar);
     void decrement (float scalar);
private:
   float data[n_dim];
};

Vector::Vector(float first,float second,float third)
{  data[0] = first;
   data[1] = second;
   data[2] = third;
}

Vector& Vector::operator -=(float scalar)
{  int i;
   for (i = 0; i < n_dim; i++)
      data[i] -= scalar;
   return *this;
}

void Vector::decrement (float scalar)
{  int i;
   for (i = 0; i < n_dim; i++)
      data[i] -= scalar;
}
```

An example of usage

```
Vector velocity (100.0, 37.0, 500.6);
velocity -= 25;
velocity.decrement (20);
```

illustrates how notationally convenient overloading operators can be.

when to inline

Overloading operators is a good topic under which to discuss the issue of when to inline more thoroughly.

Although inlining generally gives a performance advantage, it has some drawbacks. Unless the function (or operator) is smaller than the overhead of setting up a conventional call, the overall size of the program is bigger, since the code is duplicated. Also, the compiler has to process the inline's source code more often: it has to be `#included` into every file that uses it, instead of compiled once, then not seen again until link time. This is slows compilation. If you inline often, you'll frequently run into the problem mentioned in Part 9 (confusing link errors). Finally, many debuggers lose track of where you are in the source if you inline a lot, and other tools such as profilers have less information at run time.

That's not to say you should never use inlines. Once you have written your program and are starting to tune it for performance, you can start to work out which function calls are too expensive, and try inlining them. Remember the lesson of the sorting algorithm: optimizing only makes sense once you know you have the most efficient design.

the temporary problem

An additional problem with operators is that many require returning a value to be consistent with the built-in operator. In a case where the value has to be an *l*-value (capable of appearing on the left-hand-side of an assignment), it's possible to return a reference to `this`, as in the `-=` example.

However if the value returned is meant to be a completely new value, as in the result of an addition, it must be stored somewhere. In the case of built-in operators, that somewhere is generated by the compiler (a temporary space in memory, or more likely, a register)—and merged into the target of the assignment if possible. If you write your own operator, the compiler can't manage temporary values as efficiently, resulting in unnecessary construction of a new object, copying and deletion of the temporary.

By contrast, if you use a function for operations such as addition, you can use a technique such as making the current object the destination for the result.

We can add addition to the vector class, to illustrate the alternative styles:

```
Vector Vector::operator +(Vector other)
{  Vector result; // no constructor: default is all zeroes
   for (int i = 0; i < n_dim; i++)
      result.data[i] += data[i] + other.data[i];
   return result;
}

void Vector::add (Vector first, Vector second)
{  for (int i = 0; i < n_dim; i++)
      data[i] += first.data[i] + second.data[i];
}
```

which could be used as follows:

```
Vector velocity (100.0, 37.0, 500.6), accel (-1.0, 1.0, 0.0),
      final_velocity;
final_velocity = velocity + accel;
final_velocity.add (velocity, accel);
accel -= 10;
final_velocity.add (final_velocity, accel);
```

hands-on—vector class using operators

Extend the vector class to include a few common operations, like multiply by scalar (`operator *=`), inner product (`operator *`) and indexing (`operator []`).

Experiment with both implementing and using these operations as operators, as well as functions

part 14—More Advanced Features and Concepts

templates

Templates are a relatively late addition to the language and do not work properly on all compilers. Nonetheless they are a useful concept and worth explaining.

A template is a parametrized type. The sorts of Parts 3 to 5 started as a sort for strings, which became a sort for employee records, and finally a more general one with function pointers. Imagine how much better it would be if we could define a *generic* sort, which would work on any data type we could compare and exchange. C++ has templates for this purpose. Ada has a similar feature called generics. To do this in Pascal or Modula–2, you have to use a text editor to create multiple versions of a routine such as a sort, whereas in Ada or C++, the compiler can do this for you.

In C++ you can define a generic sort as follows:

```
template<class T> void sort (T data[], int n)
{  int i,j;
   for (i = 0; i < n-1; i++)
      for (j = i + 1; j > 0; j--)
         if (compare(data, j-1, j) > 0)
            swap(data, j-1, j);
}
// assume operator== and operararor< defined on T
template<class T> int compare(T data[], int i, int j)
{  if (data[i] < data[j])
      return -1;
   else if (data[i] == data[j])
      return 0;
   else
      return 1;
}
template<class T> void swap(T data[], int i, int j)
{  T temp = data[i];
   data[i] = data[j];
   data[j] = temp;
}
```

Some examples of usage:

```
int data[] = {0,1,4,3,45,2,1,4,6,89};
float money[] = {1.20,1.50,0.59,500.55,89,5};
sort (data, int(sizeof data / sizeof(int)));
sort (money, int(sizeof money / sizeof(float)));
```

The compiler automatically generates versions of `sort()` for `int` and `float` arrays when it sees the two calls.

It’s also possible to parametrize a class. For example, the vector class of Part 13 could be generalized to make vectors of general objects (some detail left out):

```
template<class T> class Vector
{public:
    T& operator [](int ind);
    void add (Vector<T> first, Vector<T> second);
private:
   T data[n_dim];
};
template<class T>T& Vector<T>::operator[](int ind)
{  return data[ind];
}
template<class T>void Vector<T>::add(Vector<T>first,
   Vector<T> second)
{  for (int i = 0; i < n_dim; i++)
      data[i] += first.data[i]+second.data[i];
}
```

Here are some examples of usage:

```
Vector<int> pos, offset;
```

```
Vector<float> vel, acc;
for (int i = 0; i < n_dim; i++)
{  pos[i] = 10-i; // use operator[]
   offset[i] = -1;
}
pos.add(pos,offset);
vel.add(vel,acc);
```

exceptions

Exceptions are another late addition to the language. Since they are not fully implemented in all compilers, I'll give a quick overview rather than detail.

The essential idea is that you `try` to execute a piece of code. If it fails (either through a built-in exception like floating-point overflow or one you `throw`), you fall through to a `catch` which handles the exception:

```
class Overflow
{// whatever state you want to store about overflows
};

try
{  Overflow status;
// code that causes an exception results in:
   throw status;
}
catch (Overflow &overflow_info)
{  // use overflow_info to handle the exception
}
```

virtual base classes

With multiple inheritance, if the same base class appears more than once in the hierarchy, it is duplicated. If you only want it to appear once, you declare it as a *virtual base class*. For example:

```
class Error
{public:
   Error (const char* new_message);
   void print_err (const char* message = "none");
private:
   char *default_message;
};
class Error_bucket : public Bucket, public Error
{public:
   Error_bucket (const char* new_default = "Hole in bucket");
};
class Error_spade : public Spade, public Error
{public:
   Error_spade (const char* new_default = "Hole in bucket");
};
class Error_beach : public Error_spade, public Error_bucket;
```

will result in an object of class Error_beach having two places to store errors. If this is not desired, the following will fix the problem:

```
class Error_bucket : public Bucket, virtual public Error // etc.
class Error_spade : public Spade, virtual public Error   // etc.
```

future feature: name spaces

A big problem with mixing class libraries from various sources is that "natural" choices of names tend to be duplicated.

For example, it's very common to have class hierarchies descended from a common ancestor with a name like `Object`, or `T_object`. Also, conventions for making symbolic names for "boolean" values are not standardized. Most use the C convention:

```
#ifndef  TRUE
#  define TRUE 1
#  define FALSE 0
#endif
```

or something along those lines, but some define a "boolean" `enum`, and libraries that do this may be hard to mix with others that use a slightly different strategy.

A proposal which is likely to be added to the language is a way of giving a name to a collection of names—a *name space*. Other languages like Ada and Modula–2 have module or package mechanisms which are slightly more robust than C++'s naming conventions, but the problem of name management in large programs exists even with these languages.

Look out for name spaces in future C++ compilers.

libraries *vs.* frameworks

Reusability is one of the selling points of object-oriented programming.

Libraries are a traditional way of making code reusable. A library is a collection of type, class and procedure definitions, designed for greater generality than code written for a special purpose. Examples of libraries include FORTRAN floating-point libraries like IMSL, the Smalltalk-80 class library, and linkable libraries typically distributed with compilers to handle routine tasks like I/O.

Some advocate going a step further, and pre-writing a large part of an application, trying to keep the code as general as possible. Functionality like updating windows and printing is supplied in very general form, and you fill in the details to make a real application.

This *application framework* approach has advantages and disadvantages. The biggest drawback is you have to understand the programming style of the framework designer. This can be a major task. Some have claimed it takes about 3 months to feel at home with MacApp, for example (one of the earlier frameworks, for writing Macintosh applications). On the other hand once you've understood the framework, you don't have to worry about many details that don't change across most applications.

My view is that a compromise is the best strategy. A good library that you can use whatever the style of program can be designed around a relatively simple application framework. This framework should be designed so it can be learnt quickly, and only implements very common functionality, or features which are tricky to get right. When you start to use it, you will tend to mostly use it as a library, gradually graduating to using it more like a framework—particularly once you start enhancing the framework with your own tricks.

i n d e x