Multidimensional arrays

Declaration:

\[ T \ a[\text{NROWS}][\text{NCOLS}]; \]

Subscripting:

\[ a[i][j] \equiv \ast((\ast(a+i))+j) \]

Other ways of writing this:

\[ \ast(a[i]+j) \]

\[ \ast(\ast(a+i))[j] \]

\[ \ast(&a[0][0] + \text{NCOLS}\ast i + j) \]

Multidimensional arrays as arguments to functions

Again, a pointer to the first element is passed.

\[
\text{void } \text{foo}(T\ (*a)[\text{NCOLS}])
\]

\[
\{
\ldots
\}
\]

\[
\text{main}()
\]

\[
\{\ 
T\ a[\text{NROWS}][\text{NCOLS}];
\text{foo}(a);
\}
\]

Equivalent forms:

\[
\text{void } \text{foo}(T\ a[])[\text{NCOLS}])
\]

\[
\text{void } \text{foo}(T\ a[\text{NROWS}][\text{NCOLS}])
\]

More about pointers

The type of the pointer depends on what it points to.

The representation of a pointer could vary depending on its type.

E.g., assuming that all pointers have the same size is not valid.

(Believing that

\[
\ast\ast\text{size}(\ast * ) = \ast\ast\text{size}(\text{char } * )
\]

is wrong.)

This has implications concerning:

- generic pointers (void *)
- null-pointers.

Generic pointers

Generic pointers, (void *), may point to an object of any (non function) type, thus:

- Pointers of any type may be converted to (void *).
- (void *) may be converted to a pointer of any type.
- (void *) must be the largest pointer available.
- In general, the representation must be changed when converting to/from (void *).

The conversion is often done automatically:

\[
\text{int } a;
\]

\[
\text{void } *b;
\]

\[
b = &a;
\]
Generic pointers (cont.)

Automatic conversion is not always possible. The following is wrong:

```c
list cons(h,t)
    void *h;
    list l;
{...}
...
int x;
...
l = cons(&x, l);
```

In this case, a prototype would have helped:

```c
list cons(void *h, list l);
```

The null pointer

Null is a special pointer that, for each pointer type:

- is distinct from every other pointer of this type,
- is not the address of any object.

Syntactically the null pointer is written 0.

Note:

- The compiler must be able to infer that a pointer is expected (and its type).
- The null pointer is not necessarily represented internally by the address 0!

The macro NULL, defined to 0 or (void *)0 is often used.

The null pointer (cont.)

The following is a typical example of when an explicit cast is necessary:

```c
execl("/usr/bin/ls", "ls",
    "-l", "mydir",
    (char *)NULL);
```

0/NULL on its own is OK for:

- initialization
- assignment
- comparison.
- function call with a visible prototype and a fix number of arguments.

Explicit cast necessary when:

- function call, no visible prototype.
- function call, variable number of args.

Pointers to functions

It is sometimes useful to pass functions as arguments to other (higher order) functions.

In C, this is done by passing pointers to functions.

Example: numerical integration.

![Numerical integration diagram]
Pointers to functions (cont.)

#include <math.h>
double trapezoidal(double a, double b, double (*f)(double), int n)
{
    int i;
    double sum, h = (b-a)/n;
    sum = (*f)(a)/2 + (*f)(b)/2;
    for (i=1; i<n; i++)
        sum += (*f)(a + i*h);
    return h*sum;
}

main()
{
    printf("%f\n", trapezoidal(0.0, 3.1415926535, sin, 50));
}

Exercise 1

Understanding types can be tricky in C.

Ground rules:

() and [] have higher priority than *, operators closest to the name go first, parentheses for grouping.

Work backwards, from inside out.

What are declared below?

char x1[5][5];
char *x2[5];
char **x3;
char (*x4)[5];
char *x5(int);
char (*x6)(void);
char (*x7(int, char*))[5];
char (*x8[5])(char);
void (*sigset (int sig, void (*disp)(int)))(int);

Macros

The preprocessor, cpp, allows macros:

#define N 100
#define MASK 0xFF

Macros may also have parameters:

#define max(i,j) ((i) > (j) ? (i) : (j))

Consider the following code:

foo = 3 * max(a & MASK, N);

It will be rewritten as follows before compilation:

foo = 3 * ((a & 0xFF) > (100) ? (a & 0xFF) : (100));

Note the importance of the parentheses in the definition of max!

The dangers of macros

A macro may look like a function, but it is not:

#define sq(x) ((x)*(x))

... 
a = sq(i++);

After pre-processing:

a = ((i++)*(i++))

Macros do not obey the normal scope rules:

#define max 1000
... 
int foo(int a, int b)
{
    int max = 0;
    ...
}
Conditional compilation
Several uses, e.g.:
• Portable code:
  #ifdef SOLARIS2
    /* Do Solaris 2 stuff */
  #else
    /* Do normal stuff */
  #endif
• Debugging aids:
  #ifdef DEBUGMODE
    printf("x=%d,y=%d\n",x,y);
  #endif
• 'Commenting out' code.
  #if 0
    tidy(); /* Tidy /tmp */
  #endif

Including files
The #include-directive comes in two forms:
• #include <file.h>: to include standard library header files.
• #include "file.h": to include files by giving a relative or absolute path.
Examples:
  #include <stdio.h>
  #include <string.h>
  #include <math.h>
  #include <stdarg.h>
  #include "foo.h"
  #include ".../fie.h"

Records (struct)
Records in C are called structures:

struct book {
  char title[50], author[50];
  int page_count, price;
  float weight;
} b1, b2;

This declares a book structure as well as two variables of this type.

Structures are ‘real values’ in that
• structures may be passed (by value) as arguments to functions
• structures may be returned from functions
• structures may be copied by assignment, e.g. b1 = b2;

Accessing structure fields
Dot-notation is used for accessing fields:

strcpy(b1.title, "Flatland");
strcpy(b1.author, "Abbott");
b1.page_count = 327;
b1.price = 239;
b1.weight = 0.85;
printf("Title: %s\n", b1.title);

Suppose bp is a book pointer, e.g.:

struct book *bp = &b1;

Then fields may be accessed in two ways:
(*bp).page_count = 517;
or
bp->page_count = 517;
Structures as function parameters

```c
void print_book(struct book b) {
    printf("Title: %s\n", b.title);
    printf("Author: %s\n", b.author);
    printf("Pages: %d\n", b.page_count);
    printf("Price: %d kr\n", b.price);
    printf("Weight: %f kg\n", b.weight);
}

void read_book(struct book *bp) {
    scanf("%s", bp->title);
    scanf("%s", bp->author);
    scanf("%d", &bp->page_count);
    scanf("%d", &bp->price);
    scanf("%f", &bp->weight);
}

main() {
    struct book b[50];
    int i;
    for (i = 0; i < 50; i++) {
        read_book(&b[i]); print_book(b[i]);
    }
}
```

Defining new type names

It is often useful to define a name for a complicated type or for abstraction purposes.

```c
typedef int bool;

typedef struct book {
    char title[50], author[50];
    int page_count, price;
    float weight;
} book, *bookptr;
```

Now the prototypes for `print_book` and `read_book` may be written as:

```c
void print_book(book b);
void read_book(bookptr b);
```

Variable declarations:

```c
bool cond1, cond2;
book b1, b2;
```

Enumeration types

C also have a kind of enumeration types:

```c
enum colour {red, green, blue};
enum colour col1, col2 = green;
```

In ANSI C, this is really just a convenient way of writing

```c
const red=0, green=1, blue=2;
int col1, col2 = 1;
```

Of course, typedefs may be used:

```c
typedef enum (...) colour;
colour col1, col2 = green;
```

Unions

A union type permits values of different types to be stored in the same variable (but not at the same time!):

```c
typedef union {
    int i;
    float f;
} int_or_float;

int_or_float i_or_f;
i_or_f.i = 10;
i_or_f.f = 17.3;
```

Somehow we must keep track of the kind of value that is stored in the union.
Unions (cont.)

typedef enum {book, video} obj_kind;
typedef enum {vhs, betamax, secam} system;
typedef char string[50];
typedef struct {
    obj_kind kind_f;
    string title_f;
    union {
        int page_count_f;
        struct {
            int length_f;
            system format_f;
        } v_info_f;
        bv_info_f;
    } bv_info_f;
} object;

However, accessing is a bit tedious:

object o;
o.bv_info_f.v_info_f.format_f = vhs;

Macros may be defined to facilitate, e.g.:

#define format(obj) 
    ((obj).bv_info_f.v_info_f.format_f)

format(o) = vhs;
if (format(o) != vhs) ...

Exercise 2

Write the remaining access macros for the object structure on the previous slide.

Use these macros to write a function that prints an object. Obviously, only relevant fields should be printed, e.g.:

Alice’s Adventures in Wonderland
127 pages

Much ado about nothing
111 min
VHS

Use the following prototype:

void print_obj(object o)

Dynamic memory allocation

Dynamic memory allocation and deallocation in C is performed by malloc and free:

#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

Typical usage:

object *op;
if ((op=(object *)malloc(sizeof(object))) == NULL) {
    printf("Failed to allocate memory!\n");
    exit(1);
} ...
free((void *) op);

Linked lists

A typical linked list:

9 <--- 2 <--- 7

Suppose we would like to handle lists of integers. We will call this type intlist. A typical set of operations:

il_empty
il_isempty
il_cons
il_head
il_tail
il_free

The list above would be created by the following calls:

il_cons(9, il_cons(2, il_cons(7, il_empty)))
Implementation of intlist

The type:

```c
typedef struct il_node {
    int head_f;
    struct il_node *tail_f;
} *intlist;
```

We decide to implement `il_empty`, `il_head` and `il_tail` by using macros:

```c
#define il_empty ((intlist) NULL)
#define il_head(is) ((is)->head_f)
#define il_tail(is) ((is)->tail_f)
```

The implementation of `il_isempty` is very simple:

```c
int il_isempty(intlist is) {
    return is == il_empty;
}
```

Cons is also straight forward:

```c
intlist cons(int i, intlist is) {
    intlist new;
    if ((new=malloc(sizeof(struct il_node))) == NULL) {
        printf("cons failed\n");
        exit(1);
    }
    il_head(new) = i;
    il_tail(new) = is;
    return new;
}
```

Il_free is a little more subtle: once freed, memory must not be referenced!

```c
void il_free(intlist is) {
    intlist n, rest;
    for (n = is; n != il_empty; n = rest) {
        rest = il_tail(n);
        free(n);
    }
}
```

Using intlist

Here is an example of how to use intlist. Integers are read from standard input and stored on a list until EOF is encountered. The list is then printed and finally deallocated.

```c
main() {
    intlist is = il_empty, is1;
    int i;
    while (scanf("%d", &i) == 1) {
        is = cons(i, is);
        if (!il_isempty(is)) {
            is1 = il_tail(is);
            il_free(is);
        }
    }
    printf("%d ", il_head(is1));
    il_free(is);
}
```
Modular program development

What?
The division of a large program into several relatively independent parts, modules.

Why?
• Structure.
• Parallel development.
• Separate compilation.
• Software reuse, libraries.

A program to read and sort integers

The problem:
Write a program that reads any number of integers. These should then be sorted and printed both in ascending and descending order.

We decide to separate the program into three modules:
1. A module to manage lists of integers of arbitrary length. Already available!
2. A module for sorting and reversing lists.
3. The main program.

The dependence structure

Thus we have the modules main, ilutils, and intlist with the following dependences:

Problems: the declarations of any external functions on which a module depend must be available in each module.

Interfaces and bodies

Idea:
• Separate a module into a body (.c) and an interface (.h).
• The interface should contain information about all externally visible entities:
  • type definitions
  • macros
  • function declarations
  • variable and constant declarations.
• Include the interface wherever its definitions and declarations are needed.
• Also include the interface in the file containing the module body.
The intlist and ilutils interfaces

/* intlist.h */

typedef struct il_node {
    int    head_f;
    struct il_node *tail_f;
} *intlist;

#define il_empty ((intlist) NULL)
#define il_head(is) ((is)->head_f)
#define il_tail(is) ((is)->tail_f)

extern int il_isempty(intlist);
extern intlist cons(int, intlist);
extern void il_free(intlist);

/* ilutils.h */

extern intlist sortlist(intlist);
extern intlist reverselist(intlist);

Note that ilutils.h depends on intlist.h. Solution:
• either make sure intlist.h is always included before ilutils.h
• or include intlist.h in ilutils.h!
The latter is usually the more convenient solution, but an interface may now be included more than once.
Solution: use conditional compilation.

The final interfaces

/* intlist.h */
#ifndef INTLIST_INCLUDED
#define INTLIST_INCLUDED
#include <stddef.h>

typedef struct il_node {
    int    head_f;
    struct il_node *tail_f;
} *intlist;

#define il_empty ((intlist) NULL)
#define il_head(is) ((is)->head_f)
#define il_tail(is) ((is)->tail_f)

extern int il_isempty(intlist);
extern intlist cons(int, intlist);
extern void il_free(intlist);
#endif

/* ilutils.h */
#ifndef ILUTILS_INCLUDED
#define ILUTILS_INCLUDED
#include "intlist.h"

extern intlist sortlist(intlist);
extern intlist reverselist(intlist);
#endif

Compilation and linking

Two steps to produce an executable file:
1. Compilation of source code to object files (.o). Performed by the preprocessor (cpp) and compiler (cc).
2. Linking all object files and any necessary libraries into a single, executable file. Performed by the linker (ld).

```plaintext
c. c -c a.c b.c
cpp/cc  a.o b.o
cpp/cc b.o
dl foo
```

```plaintext
cc -l libc
cc -l l1hm
```
The make dependences

A file $X$ is make dependent on a file $Y$ if $X$ must be remade when $Y$ changes.

```
make
Given what basically is a textual encoding of the make dependences, `make` will make sure that all files are up to date.
A file is *out of date* if:
- it is missing,
- it is dependent on files that are newer than itself, or
- it is dependent on files that are *out of date*.
```

The Makefile

The dependences are usually stored in `Makefile`:

```
sortints: main.o intlist.o ilutils.o
    cc -c sortints main.o intlist.o ilutils.o
main.o: main.c ilutils.h intlist.h
    cc -c main.c
ilutils.o: ilutils.c ilutils.h intlist.h
    cc -c ilutils.c
intlist.o: intlist.c intlist.h
    cc -c intlist.c
```

Some refinements

Use predefined macros:

```
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) -c
LINK.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS)
```

Then:

```
CC = cc
CFLAGS = -g
OBJ = main.o ilutils.o intlist.o
sortints: $(OBJ)
    $(LINK.c) -o sortints $(OBJ)
main.o: main.c ilutils.h intlist.h
    $(COMPILE.c) main.c
ilutils.o: ilutils.c ilutils.h intlist.h
    $(COMPILE.c) ilutils.c
intlist.o: intlist.c intlist.h
    $(COMPILE.c) intlist.c
```

```
clean:
    /bin/rm -f $(OBJ) sortints core
```
Some further refinements when using Sun’s make

Sun’s make and compilers are clever enough to figure out most dependences themselves!

>>> .KEEP_STATE:
>>> CC = cc
>>> CFLAGS = -g
>>> OBJ = main.o ilutils.o intlist.o
>>> sortints: $(OBJ)
>>> $(LINK.c) -o sortints $(OBJ)

>>> clean:
>>> /bin/rm -f $(OBJ) sortints core

Unfortunately, this is not standard, and such makefiles cannot be used on other systems.

Exercise 3

Write a makefile that takes care of compiling the following files:

main.c:
#include "foo.h"

foo.h: no #includes.

foo.c:
#include "fie.h"
#include "foo.h"

fie.h: no #include.

fie.c:
#include "fie.h"

The executable program is made from the files main.o, fie.o, foo.o, and the mathematical library, and should be called fum.

Solution exercise 2

#define kind(obj) ((obj).kind_f)
#define title(obj) ((obj).title_f)
#define page_count(obj) ((obj).bv_info_f.page_count_f)
#define length(obj) ((obj).bv_info_f.v_info_f.length_f)
#define format(obj) ((obj).bv_info_f.v_info_f.format_f)

void print_obj(object o) {
    printf("%s\n", title(o));
    if (kind(o) == book) {
        printf("%d pages\n", page_count(o));
    } else {
        printf("%d min\n", length(o));
        switch (format(o)) { case vhs: printf("VHS\n"); break;
        case betamax: printf("Betamax\n"); break;
        case secam: printf("SECAM\n"); break;
        }
        printf("\n");
    }
}

Solution exercise 3

OBJ = main.o foo.o fie.o
fum: $(OBJ)
$(LINK.c) -o fum $(OBJ) -lm
main.o: main.c foo.h
$(COMPILE.c) main.c
foo.o: foo.c foo.h fie.h
$(COMPILE.c) foo.h
fie.o: fie.c fie.h
$(COMPILE.c) fie.h

clean:
/bin/rm -f $(OBJ) fum core