C

robert lupton[1]

23 september 2010

[1]LooChao@gmail.com

Outline

Introduction

C was designed as a system programming language, to remove the necessity of writing operating systems in assembler. It's one of the large family of languages deriving from Algol.

Introduction

C was designed as a system programming language, to remove the necessity of writing operating systems in assembler. It's one of the large family of languages deriving from Algol.

You've seen "hello world" before: CANNOT INCLUDE FILE src/hello.c

Introduction

C was designed as a system programming language, to remove the necessity of writing operating systems in assembler. It's one of the large family of languages deriving from Algol.

You've seen "hello world" before: CANNOT INCLUDE FILE src/hello.c

This example comes from "The C Programming Language" by Brian Kernighan and Dennis Ritchie ("K&R")

Declarations

All variables must be declared before they can be used:

```
char c = 'a';
char *s = "I am a string";
double d = 1;
float f = 1.0f;
int i = 101010101;
long l = 10;
short s = 10;
```

Declarations

All variables must be declared before they can be used:

```
char c = 'a';
char *s = "I am a string";
double d = 1;
float f = 1.0f;
int i = 101010101;
long l = 10;
short s = 10;
```

"string" is spelt char * for reasons that I'll explain in a bit.

Declarations

All variables must be declared before they can be used:

```
char c = 'a';
char *s = "I am a string";
double d = 1;
float f = 1.0f;
int i = 101010101;
long l = 10;
short s = 10;
```

"string" is spelt char * for reasons that I'll explain in a bit. short is actually a shorthand for short int (long means long int). You can also define integral types (char and int) as being signed or unsigned.

Declarations

All variables must be declared before they can be used:

```
char c = 'a';
char *s = "I am a string";
double d = 1;
float f = 1.0f;
int i = 101010101;
long l = 10;
short s = 10;
```

"string" is spelt char * for reasons that I'll explain in a bit. short is actually a shorthand for short int (long means long int). You can also define integral types (char and int) as being signed or unsigned. One especially useful qualifier is const:

```
const unsigned long bad = 0xdeadbeef00000000;
```

Declarations

All variables must be declared before they can be used:

```
char c = 'a';
char *s = "I am a string";
double d = 1;
float f = 1.0f;
int i = 101010101;
long l = 10;
short s = 10;
```

"string" is spelt char * for reasons that I'll explain in a bit. short is actually a shorthand for short int (long means long int). You can also define integral types (char and int) as being signed or unsigned. One especially useful qualifier is const:

```
const unsigned long bad = 0xdeadbeef00000000;
```

An unqualified int is supposed to be the most efficient integral type on your machine, and is what you'd generally use unless there was some reason not to.

Declaring your own types

C allows you to use your own name for a type:

```
typedef unsigned short U16;
```

Declaring your own types

C allows you to use your own name for a type:

```
typedef unsigned short U16;
```

Why would I want to do this?

Declaring your own types

C allows you to use your own name for a type:

```
typedef unsigned short U16;
```

Why would I want to do this?
CCD image data is typically created using a 16-bit A/D converter, so the natural type for a single pixel is a 2-byte integer.
But C doesn't tell me how large an `int` is; I can find out (using `sizeof(int)`) but I don't want to have to change all my declarations when I move to a new system.

Declaring your own types

C allows you to use your own name for a type:

```
typedef unsigned short U16;
```

Why would I want to do this?
CCD image data is typically created using a 16-bit A/D converter, so the natural type for a single pixel is a 2-byte integer.
But C doesn't tell me how large an int is; I can find out (using sizeof (int)) but I don't want to have to change all my declarations when I move to a new system.
Actually, these days, I could say:

```
#include <stdint.h>
typedef uint16_t U16;
```

but even that only works if the processor actually has an unsigned 16-bit type.

Declarations

You can mix code and declarations:

```
int pwv = 0;                                   // Precipitable Water Va
/*
 * Estimate the PWV based on the altitude
 */
...;
/*
 * Given that estimate, find a better value
 */
const int pwv0 = pwv;                          /* initial estimate of p
```

Declarations

You can mix code and declarations:

```
int pwv = 0;                                      // Precipitable Water Va
/*
 * Estimate the PWV based on the altitude
 */
...;
/*
 * Given that estimate, find a better value
 */
const int pwv0 = pwv;                             /* initial estimate of p
```

The ability to declare variables when they are first needed means that you can usually initialize them too; when possible, make them const.

Scope

A variable's *scope* is the part of the programme it may be referenced from; in C, a variable's scope is the nearest set of braces ({}), a *block*. If it isn't in a block, a variable is visible globally (i.e. it'll show up when you run nm on your object file). If this isn't what you wanted, you can:

Scope

A variable's *scope* is the part of the programme it may be referenced from; in C, a variable's scope is the nearest set of braces ({}), a *block*. If it isn't in a block, a variable is visible globally (i.e. it'll show up when you run nm on your object file). If this isn't what you wanted, you can:

- Move it into a function — remember, global variables should usually be avoided

Scope

A variable's *scope* is the part of the programme it may be referenced from; in C, a variable's scope is the nearest set of braces ({}), a *block*. If it isn't in a block, a variable is visible globally (i.e. it'll show up when you run nm on your object file). If this isn't what you wanted, you can:

- Move it into a function — remember, global variables should usually be avoided
- Label it `static` which makes it only visible within the file

Scope

A variable's *scope* is the part of the programme it may be referenced from; in C, a variable's scope is the nearest set of braces ({}), a *block*. If it isn't in a block, a variable is visible globally (i.e. it'll show up when you run nm on your object file). If this isn't what you wanted, you can:

- Move it into a function — remember, global variables should usually be avoided
- Label it static which makes it only visible within the file
- Decide that it really *must* be globally visible, and declare it in a header file:

  ```
  extern int nread;     // Number of times I've read from a fi
  ```

Scope

A variable's *scope* is the part of the programme it may be referenced from; in C, a variable's scope is the nearest set of braces ({}), a *block*. If it isn't in a block, a variable is visible globally (i.e. it'll show up when you run nm on your object file). If this isn't what you wanted, you can:

- Move it into a function — remember, global variables should usually be avoided
- Label it static which makes it only visible within the file
- Decide that it really *must* be globally visible, and declare it in a header file:

```
extern int nread;      // Number of times I've read from a fil
```

It is generally a good idea to declare a variable in as restricted a scope as possible (the "No Globals" rule is a special case of this one).

Control Flow

C has the usual constructs:

- if (...) { ... } else if (...) { ...} else { ... }

Control Flow

C has the usual constructs:

- if (...) { ... } else if (...) { ...} else { ... }
- for (...) { ... }
- while { ... }
- do { ... } while (...);

Control Flow

C has the usual constructs:

- if (...) { ... } else if (...) { ...} else { ... }
- for (...) { ... }
- while { ... }
- do { ... } while (...);
- switch (...) { case XXX: ... break; ... }

Control Flow

C has the usual constructs:

- if (...) { ... } else if (...) { ...} else { ... }
- for (...) { ... }
- while { ... }
- do { ... } while (...);
- switch (...) { case XXX: ... break; ... }
- break, continue

Control Flow

C has the usual constructs:

- if (...) { ... } else if (...) { ...} else { ... }
- for (...) { ... }
- while { ... }
- do { ... } while (...);
- switch (...) { case XXX: ... break; ... }
- break, continue
- goto

If statements

```
if (i < 9) {
    printf("Hello");
} else {
    printf("Goodbye");
}
printf(" world\n");
```

If statements

```
if (i < 9) {
    printf("Hello");
} else {
    printf("Goodbye");
}
printf(" world\n");
```

Legally, you can write this as

```
if (i < 9)
    printf("Hello");
else
    printf("Goodbye");

printf(" world\n");
```

But we don't recommend it;

If statements

```
if (i < 9) {
    printf("Hello");
} else {
    printf("Goodbye");
}
printf(" world\n");
```

Legally, you can write this as

```
if (i < 9)
    printf("Hello");
else
    printf("Goodbye");

printf(" world\n");
```

But we don't recommend it; it's too easy to write

```
if (i < 9)
    printf("Hello");
else
```

For loops

CANNOT INCLUDE FILE src/for.c This only works with a compiler that supports the C99 standard (cc −−std=c99 ...)

For loops

CANNOT INCLUDE FILE src/for.c This only works with a compiler that supports the C99 standard (cc −−std=c99 ...) This is almost equivalent to:

```
#include <stdio.h>

int
main()
{
     int i = 0;
     while (i != 10) {
          printf("Hello world\n");
          ++i;
     }

     return 0;
}
```

Switch I

CANNOT INCLUDE FILE src/printf.c

Switch

```
$ make printf && ./printf
$ cc -g -Wall -O3 --std=c99    printf.c   -o printf
a
b
c
:
<space>
<integer>
%
<space>
d
o
n
e
<newline>
```

(N.b. The call was checkFormat("abc: %d%% done\n");)

Putting that together

CANNOT INCLUDE FILE src/switch.c

Subroutines

Here is a function to add two numbers:

Subroutines

Here is a function to add two numbers:
and here is one to multiply them:

Subroutines

Here is a function to add two numbers:
and here is one to multiply them:
So far, just like Fortran

Subroutines

Here is a function to add two numbers:
and here is one to multiply them:
So far, just like Fortran

Subroutines

Here is a function to add two numbers:
and here is one to multiply them:
So far, just like Fortran
That's a little clumsier — in Fortran (or python) you could have said x**2. And note the extra include file, **math.h**. In the old days you needed to link −lm too, but modern systems seem to be more forgiving.

Recursion

Subroutines may be called recursively — that is, they may call themselves, either directly or indirectly. There is no limit except the capacity of your computer. CANNOT INCLUDE FILE src/factorial.c

Recursion

That may not seem very interesting; it's easy enough to write a loop to calculate factorials.

However, consider a routine

```
integrate ( float a, float b, float (*func)( float x)) 2
```

Recursion

That may not seem very interesting; it's easy enough to write a loop to calculate factorials.

However, consider a routine

`integrate(float a, float b, float (*func)(float x))` [2]

If I need to do a double integral, I can say something like:

```
static float yy;                        // current value of y

static float func(float x) {
   return sin(x)*cos(yy);
}

static float dfunc(float y) {
   yy = y;                              // pass y to func
   return integrate(0, y, func);
}

const double ans = integrate(1, 3, dfunc);
```

to calculate

Complete Example

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

double integrate(const float a, const float b, float (*func)(flo
      const int nstep = 1000;              // number of steps
      const float step = (b - a)/nstep;

      double ans = 0.0;
      float x = a;
      for (int i = 0; i != nstep; ++i) {
         ans += func(x);
         x += step;
      }

      return step*ans;
}

static float yy;                           // current value of y
```

Subroutine Arguments

Question: What does this do?

```
void triple(double x) {
   x *= 3;
}
...
double x = 1;
triple(x);
```

Subroutine Arguments

Question: What does this do?

```
void triple(double x) {
   x *= 3;
}
...
double x = 1;
triple(x);
```

Answer: wastes CPU cycles.
The function triple is passed a *copy* of x, so **nothing** that triple does can affect the program that calls it.

Subroutine Arguments

Question: What does this do?

```
void triple(double x) {
   x *= 3;
}
...
double x = 1;
triple(x);
```

Answer: wastes CPU cycles.
The function triple is passed a *copy* of x, so **nothing** that triple does can affect the program that calls it.
The solution is to pass a *pointer* to x:

```
void triple(double @\color{red}*@x) {
    @\color{red}*@x *= 3;
}
...
double x = 1;
triple(@\color{red}\&@x);
```

Subroutine Arguments

Question: What does this do?

```
void triple(double x) {
   x *= 3;
}
...
double x = 1;
triple(x);
```

Answer: wastes CPU cycles.
The function triple is passed a *copy* of x, so **nothing** that triple does can affect the program that calls it.
The solution is to pass a *pointer* to x:

```
void triple(double @\color{red}*@x) {
    @\color{red}*@x *= 3;
}
...
double x = 1;
triple(@\color{red}\&@x);
```

Fortran *always* passes arguments this way; a famous party trick when I

Prototypes

It is critically important that subroutines' callers and callees agree about the number and types of the arguments (the *signature*). C uses a *prototype* to allow the compiler to check; given

```
void triple(double *x) {
    *x *= 3;
}
```

we put the prototype `void triple(double *x);` in a header (".h") file and `#include` it in the file that defines `triple` and also whenever we call `triple`. [3]
This isn't a lot of extra work and soon becomes second nature.

[3] Most compilers these days will complain if you don't do this.

Arrays

Arrays are declared and subscripted using `[]`.

```
int ids[10];
```

Arrays

Arrays are declared and subscripted using [] .

```
int ids[10];
```

In C99, the dimension need *not* be known at compile time:

```
void foo(const int n) {
    int ids[n];
    ...
    double x = ids[n/2];
}
```

The index starts at *0* (not *1*, as in Fortran) as it specifies the distance from the start of the array.

```
printf("ID0: %d\n", ids[0]);
```

Arrays

Arrays are declared and subscripted using `[]`.

```
int ids[10];
```

In C99, the dimension need *not* be known at compile time:

```
void foo(const int n) {
    int ids[n];
    ...
    double x = ids[n/2];
}
```

The index starts at *0* (not *1*, as in Fortran) as it specifies the distance from the start of the array.

```
printf("ID0: %d\n", ids[0]);
```

This may make more sense when we discuss *pointers*.

n-D Arrays

You can also define n-D arrays:

```
U16 data[4096][2048];
...
U16 const peak = data[y][x];                // the (x, y)th pixel
```

The data is stored row-by-row so the *last* index increases fastest if we access pixels in the order in which they're stored (unlike Fortran arrays, in which the *first* index varies fastest).

n-D Arrays

You can also define n-D arrays:

```
U16 data[4096][2048];
...
U16 const peak = data[y][x];              // the (x, y)th pixel
```

The data is stored row-by-row so the *last* index increases fastest if we access pixels in the order in which they're stored (unlike Fortran arrays, in which the *first* index varies fastest).
n-D arrays aren't as useful as you might think as they are passed to subroutines as *pointers*; for now all you need to know is that you **must** specify all but the first dimension when passing an n-D array to a subroutine:

```
void debias(U16 data[][2048], const int nrow, const int ncol) {
    ...
}
```

`n`-D Arrays

You can also define `n`-D arrays:

```
U16 data[4096][2048];
...
U16 const peak = data[y][x];              // the (x, y)th pixel
```

The data is stored row-by-row so the *last* index increases fastest if we access pixels in the order in which they're stored (unlike Fortran arrays, in which the *first* index varies fastest).

`n`-D arrays aren't as useful as you might think as they are passed to subroutines as *pointers*; for now all you need to know is that you **must** specify all but the first dimension when passing an `n`-D array to a subroutine:

```
void debias(U16 data[][2048], const int nrow, const int ncol) {
   ...
}
```

(hmm, I seem to have typed 2048 twice; was that a good idea. . . ?)

Operator Precedence

operator	associativity
`() [] -> .`	left to right
`! ~ ++ -- - (type) * & sizeof()`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -=` etc.	right to left
`,`	left to right

Macros and the C Pre-Processor (*CPP*)

C provides a simple macro processor which can be used to keep *magic numbers* out of your code[3] instead of

```
char configFile[21];                      // name of configuration
fgets(stdin, fileName, 20);
```

Macros and the C Pre-Processor (*CPP*)

C provides a simple macro processor which can be used to keep *magic numbers* out of your code[3] instead of

```
char configFile[21];                    // name of configuration
fgets(stdin, fileName, 20);
```

you can write

```
#define FILE_LEN 20                     // maximum length for fi

char configFile[FILE_LEN + 1];          // name of configuration
fgets(stdin, fileName, FILE_LEN);
```

Macros and the C Pre-Processor (*CPP*)

C provides a simple macro processor which can be used to keep *magic numbers* out of your code[3] instead of

```
char configFile[21];                    // name of configuration
fgets(stdin, fileName, 20);
```

you can write

```
#define FILE_LEN 20                     // maximum length for fi

char configFile[FILE_LEN + 1];          // name of configuration
fgets(stdin, fileName, FILE_LEN);
```

writing all macros in CAPITALS is a common convention.

Conditional Compilation

In the bad old days, we wrote lots of code like:

```
#if defined(vms)
   return -1;
#elif defined(HAVE_SELECT) && !defined(USE_POLL)
   if(select(ncheck,&mask,(fd_set *)NULL,(fd_set *)NULL,
              (void *)NULL) != 0) {
        ...
   }
#else
  /* Use poll() */
#endif
```

Conditional Compilation

In the bad old days, we wrote lots of code like:

```
#if defined(vms)
   return -1;
#elif defined(HAVE_SELECT) && !defined(USE_POLL)
   if(select(ncheck,&mask,(fd_set *)NULL,(fd_set *)NULL,
             (void *)NULL) != 0) {
         ...
   }
#else
  /* Use poll() */
#endif
```

Fortunately, in these days of Posix (`IEEE 1003`; `ISO/IEC 9945`) there is much less need for of this sort of thing.

Conditional Compilation

In the bad old days, we wrote lots of code like:

```
#if defined(vms)
   return -1;
#elif defined(HAVE_SELECT) && !defined(USE_POLL)
   if(select(ncheck,&mask,(fd_set *)NULL,(fd_set *)NULL,
             (void *)NULL) != 0) {
        ...
   }
#else
  /* Use poll() */
#endif
```

Fortunately, in these days of Posix (`IEEE 1003`; `ISO/IEC 9945`) there is much less need for of this sort of thing. The (or at least my) main use of the CPP is:

```
#define DEBUG 1
...
#if DEBUG
int niter = 0;
```

Good uses for macros I

One standard use for macros is to prevent header files being parsed more than once

```
#if !defined(GREET_H)
#define GREET_H
/* Lots of stuff that should be processed only once */
#endif
```

This use of a macro is known as an *include guard*.

Good uses for macros II

CANNOT INCLUDE FILE src/macros.c

Good uses for macros II

CANNOT INCLUDE FILE src/macros.c

```
$ make macros && macros 1 2 3
cc -g -Wall --std=c99 macros.c -o macros
macros.c:009: Hello world!
macros.c:010: My name is macros and I was called with 3 argument
```

Good uses for macros II

CANNOT INCLUDE FILE src/macros.c

```
$ make macros && macros 1 2 3
cc -g -Wall --std=c99 macros.c -o macros
macros.c:009: Hello world!
macros.c:010: My name is macros and I was called with 3 argument
```

Unfortunately, this code does not conform to the C99 standard (`ISO 9899:1999`):

```
$ cc -g -Wall --std=c99 --pedantic-errors macros.c -o macros
macros.c:3:29: error: ISO C does not permit named variadic macro
macros.c:7:30: error: ISO C99 requires rest arguments to be used
```

Good uses for macros II

A legal version is: CANNOT INCLUDE FILE src/macros2.c Note that we were forced to pass at least one argument (hence the "" in the first call).

Bad uses for macros I

You can use the CPP to pretend that you're writing Algol, not C

```
#define IF      if(
#define THEN    ){
#define ELSE    } else {
#define ELIF    } else if (
#define FI      ;}

IF x == 0 THEN
   printf("zero\n");
ELIF x == 1 THEN
   printf("one\n");
ELSE
   printf("many\n");
FI
```

Bad uses for macros I

You can use the CPP to pretend that you're writing Algol, not C

```
#define IF      if(
#define THEN    ){
#define ELSE    } else {
#define ELIF    } else if (
#define FI      ;}

IF x == 0 THEN
   printf("zero\n");
ELIF x == 1 THEN
   printf("one\n");
ELSE
   printf("many\n");
FI
```

These macros come from the original Bourne shell source code in Unix Version 7.

Bad uses for macros II

How about

```
#define MAX(a,b)        ((a)>(b)?(a):(b))
```

This one also comes from Steve Bourne.

Bad uses for macros II

How about

```
#define MAX(a,b)        ((a)>(b)?(a):(b))
```

This one also comes from Steve Bourne.
Why is this bad? Consider

```
const double fg = MAX(funcs(x), gunks(x));
```

Bad uses for macros II

How about

```
#define MAX(a,b)        ((a)>(b)?(a):(b))
```

This one also comes from Steve Bourne.
Why is this bad? Consider

```
const double fg = MAX(funcs(x), gunks(x));
```

```
double funcs(const double x) {
   static int x = 0.5;
   x = sin(x);
   return x;
}

static int i = 0;
double gunks(const double x) {
   return (x < ++i) ? 5.6 : 5.9;
}
```

Bad uses for macros II

How about

```
#define MAX(a,b)          ((a)>(b)?(a):(b))
```

This one also comes from Steve Bourne.
Why is this bad? Consider

```
const double fg = MAX(funcs(x), gunks(x));
```

```
double funcs(const double x) {
   static int x = 0.5;
   x = sin(x);
   return x;
}

static int i = 0;
double gunks(const double x) {
   return (x < ++i) ? 5.6 : 5.9;
}
```

Both funcs and gunks are called twice; this is always inefficient, but in this case catastrophic as they maintain internal state.

Bad uses for macros II

How about

```
#define MAX(a,b)        ((a)>(b)?(a):(b))
```

This one also comes from Steve Bourne.
Why is this bad? Consider

```
const double fg = MAX(funcs(x), gunks(x));
```

```
double funcs(const double x) {
   static int x = 0.5;
   x = sin(x);
   return x;
}

static int i = 0;
double gunks(const double x) {
   return (x < ++i) ? 5.6 : 5.9;
}
```

Both funcs and gunks are called twice; this is always inefficient, but in this case catastrophic as they maintain internal state.

Bad uses for macros II

How about

```
#define MAX(a,b)        ((a)>(b)?(a):(b))
```

This one also comes from Steve Bourne.
Why is this bad? Consider

```
const double fg = MAX(funcs(x), gunks(x));
```

```
double funcs(const double x) {
   static int x = 0.5;
   x = sin(x);
   return x;
}

static int i = 0;
double gunks(const double x) {
   return (x < ++i) ? 5.6 : 5.9;
}
```

Both funcs and gunks are called twice; this is always inefficient, but in this case catastrophic as they maintain internal state.

Structs

If you routinely write code like:

```
void printObjects(const int n, const int id[],
                  const float xcen[], const float ycen[],
                  const float flux[]);

#define NOBJECT 1000                          // Maximum number of obj

int id[NOBJECT];                              // Object IDs
float xcen[NOBJECT];                          // x-coordinate of centr
float ycen[NOBJECT];                          // y-coordinate of centr
float flux[NOBJECT];                          // object's flux

for (int i = 0; i < n; ++i) {
   id[i] = i;
   xcen[i] = ...;
   ycen[i] = ...;
   flux[i] = ...;
}
```

Structs

A cleaner way to write this is:

```
struct Object {
   int id;                                   // Object IDs
   float xcen;                               // x-coordinate of centr
   float ycen;                               // y-coordinate of centr
};

typedef struct Object Object;

void printObjects(const int n, const Object objs[]);

#define NOBJECT 1000                         // Maximum number of obj

Object objs[NOBJECT];                        // our objects

for (int i = 0; i < n; ++i) {
   objs[i].id = i;
   objs[i].xcen = ...;
   objs[i].ycen = ...;
```

Structs

A cleaner way to write this is:

```
struct Object {
    int id;                                   // Object IDs
    float xcen;                               // x-coordinate of centr
    float ycen;                               // y-coordinate of centr
};

typedef struct Object Object;

void printObjects(const int n, const Object objs[]);

#define NOBJECT 1000                          // Maximum number of obj

Object objs[NOBJECT];                         // our objects

for (int i = 0; i < n; ++i) {
    objs[i].id = i;
    objs[i].xcen = ...;
```

Memory Allocation

You should be a little uneasy about NOBJECT in that last example. A number of questions come to mind:

- How did I know that I only had 1000 objects?
- What would I do if I had more?
- Am I wasting space if I have less?

Memory Allocation

You should be a little uneasy about NOBJECT in that last example. A number of questions come to mind:

- How did I know that I only had 1000 objects?
- What would I do if I had more?
- Am I wasting space if I have less?

and also

- Where is the computer putting all those IDs and positions?

Inside your job

A running program consists of a number of pieces:

Inside your job

A running program consists of a number of pieces:

- `text` Our instructions and read-only data

Inside your job

A running program consists of a number of pieces:

- `text` Our instructions and read-only data
- `data` Initialised global data

Inside your job

A running program consists of a number of pieces:

- `text` Our instructions and read-only data
- `data` Initialised global data
- `bss` Uninitialized global data

Inside your job

A running program consists of a number of pieces:

- `text` Our instructions and read-only data
- `data` Initialised global data
- `bss` Uninitialized global data
- `stack` Memory for variables in subroutines

Inside your job

A running program consists of a number of pieces:

- `text` Our instructions and read-only data
- `data` Initialised global data
- `bss` Uninitialized global data
- `stack` Memory for variables in subroutines
- `heap` Memory available to the programmer

Inside your job

A running program consists of a number of pieces:

- `text` Our instructions and read-only data
- `data` Initialised global data
- `bss` Uninitialized global data
- `stack` Memory for variables in subroutines
- `heap` Memory available to the programmer

All of these are mapped into a single (logical) section of RAM:

0x0	text	data	bss	heap		stack

(`0x0` is how you write a hexadecimal number in C (or C++))

Stack and Heap

It's traditional to think of the heap and stack as growing down and up respectively:

heap	$\downarrow$
stack	$\uparrow$

The Stack

What happens when I call a function? E.g.

```
double integrate(const float a, const float b, float (*func)(flo
      const int nstep = 1000;             // number of steps
      const float step = (b - a)/nstep;
      ...
      ans += func(x);
}

double ans = integrate(1, 3, dfunc);
```

The Stack

What happens when I call a function? E.g.

```
double integrate(const float a, const float b, float (*func)(flo
      const int nstep = 1000;            // number of steps
      const float step = (b - a)/nstep;
      ...
      ans += func(x);
}

double ans = integrate(1, 3, dfunc);
```

We first push the values of `a`, `b`, and `func` onto the stack:

<dfunc>
3.0
1.0

The Stack

What happens when I call a function? E.g.

```
double integrate(const float a, const float b, float (*func)(flo
      const int nstep = 1000;           // number of steps
      const float step = (b - a)/nstep;
      ...
      ans += func(x);
}

double ans = integrate(1, 3, dfunc);
```

We first push the values of `a`, `b`, and `func` onto the stack:

<dfunc>
3.0
1.0

We then make space for `step` and `nstep`

???
???
<dfunc>

The Stack

We initialised `step` to `1000` and `nstep` to `(b - a)/nstep`, so:

`0.002`
`1000`
`<dfunc>`
`3.0`
`1.0`

The Stack

We initialised `step` to `1000` and `nstep` to `(b - a)/nstep`, so:

0.002
1000
<dfunc>
3.0
1.0

Within `integrate` we call `dfunc` with argument `x`; this pushes `x` (`1.0`) onto the stack:

1.0
0.002
1000
<dfunc>
3.0
1.0

The Stack

within `dfunc` we calculate `y` (`1.0`) and call integrate (0, y, func) resulting in:

0.001
1000
<func>
1.0
0.0
1.0
0.002
1000
<dfunc>
3.0
1.0

And so on.

The Stack

When `integrate` has calculated its return value, it puts it somewhere and *pops* the stack:

1.0

0.002
1000
<dfunc>
3.0
1.0

The Stack

When `integrate` has calculated its return value, it puts it somewhere and *pops* the stack:

1.0
0.002
1000
<dfunc>
3.0
1.0

`dfunc` puts *its* return value somewhere, and pops the stack:

0.002
1000
<dfunc>
3.0
1.0

The Stack

When `integrate` has calculated its return value, it puts it somewhere and *pops* the stack:

1.0
0.002
1000
<dfunc>
3.0
1.0

`dfunc` puts *its* return value somewhere, and pops the stack:

0.002
1000
<dfunc>
3.0
1.0

And finally the outer call to `integrate` finishes, saves its value, pops the stack, and we're back where we started.

Subroutine Arguments Redux

It should now be clear that:

- A language can **only** pass variables by value if it wishes to support recursion [4]
- Variables in subroutines are irretrievably lost when the routine returns
- Uninitialized values may have any value
- (Almost) all variables are stored somewhere in the process's memory

Subroutine Arguments Redux

It should now be clear that:

- A language can **only** pass variables by value if it wishes to support recursion [4]
- Variables in subroutines are irretrievably lost when the routine returns
- Uninitialized values may have any value
- (Almost) all variables are stored somewhere in the process's memory

The last bullet suggests a way to work around the first one: we can push the variable's *address* onto the stack and agree to use not the value on the stack, but the value stored at that location.

Subroutine Arguments Redux

It should now be clear that:

- A language can **only** pass variables by value if it wishes to support recursion [4]
- Variables in subroutines are irretrievably lost when the routine returns
- Uninitialized values may have any value
- (Almost) all variables are stored somewhere in the process's memory

The last bullet suggests a way to work around the first one: we can push the variable's *address* onto the stack and agree to use not the value on the stack, but the value stored at that location.
In C, `&x` is `x`'s address, and `*px` means "use the value stored at the address `px`"

Pointers

A variable that holds an address is called a *pointer*; there's nothing magic about it; it just happens that you can apply the $*$ operator if you want to use the value it points to.

Pointers

A variable that holds an address is called a *pointer*; there's nothing magic about it; it just happens that you can apply the * operator if you want to use the value it points to.

```
int i = 0;
int *pi = &i;
printf("i = %d, %d\n", i, *pi);
*pi = 10;
printf("i = %d, %d\n", i, *pi);
```

Pointers

A variable that holds an address is called a *pointer*; there's nothing magic about it; it just happens that you can apply the * operator if you want to use the value it points to.

```
int i = 0;
int *pi = &i;
printf("i = %d, %d\n", i, *pi);
*pi = 10;
printf("i = %d, %d\n", i, *pi);
```

If you haven't ensured that a pointer is set to a valid address, you're going to suffer. If you say

```
int *pi;
*pi = 10;
printf("i = %d\n", *pi);
```

your program may well (and should!) crash.

Arrays

By definition, given any type type

```
int i;
type p[N];
p[i] == *(p + i);
p + i == &p[i]
```

Arrays

By definition, given any type type

```
int i;
type p[N];
p[i] == *(p + i);
p + i == &p[i]
```

i.e. adding an integer to a pointer gives you an address larger by $i * \text{sizeof}(\text{type})$

Arrays

By definition, given any type type

```
int i;
type p[N];
p[i] == *(p + i);
p + i == &p[i]
```

i.e. adding an integer to a pointer gives you an address larger by $i * \text{sizeof}\,(\text{type})$

E.g. float is usually a 4-byte real, so if p is `0xffff0000`, $p + 2$ is `0xffff0008`.

Arrays

By definition, given any type type

```
int i;
type p[N];
p[i] == *(p + i);
p + i == &p[i]
```

i.e. adding an integer to a pointer gives you an address larger by $i * \mathrm{sizeof}(type)$

E.g. float is usually a 4-byte real, so if p is `0xffff0000`, $p + 2$ is `0xffff0008`.

In fact, whenever you refer to an array (p), it is treated as a pointer to the first element ($\&p[0]$). This is why you can't pass an `n`-D array to a subroutine.

Strings

We introduced char * as a way of spelling "string" and we can now see how it works.

We can write `char str[7] = "abcdef";` [5]

Strings

We introduced char $*$ as a way of spelling "string" and we can now see how it works.
We can write char str [7] = "abcdef"; [5]
The statement char *str = "abcdef"; is analogous to float *x = &xx;, and *str is indeed a.

Strings

We introduced `char *` as a way of spelling "string" and we can now see how it works.

We can write `char str[7] = "abcdef";` [5]

The statement `char *str = "abcdef";` is analogous to `float *x = &xx;`, and `*str` is indeed `a`.

The difference between `char *str = "abcdef";` and `char str[7] = "abcdef";` is where the data is actually stored; in the former case it's in the `data` segment, in the latter case it's on the stack.

Structs revisited

Convinced of the value of a `struct` such as

```
struct Object {
   int id;                              // Object IDs
   float xcen;                          // x-coordinate of centr
   float ycen;                          // y-coordinate of centr
};

typedef struct Object Object;
```

I wrote a convenience function:

```
Object *newObject(const int id, const float xcen, const double y
   Object obj;

   obj.id = id;
   obj.xcen = xcen;
   obj.ycen = ycen;

   return &obj;
}
```

Malloc

After some time spent in gdb, I remembered:

- Variables in subroutines are irretrievably lost when the routine returns

and that's exactly what this does:

```
Object *newObject(const int id, const float xcen, const double y
   Object obj;
   ...
   return @\color{red}\&@obj;
}
```

Malloc

After some time spent in gdb, I remembered:

- Variables in subroutines are irretrievably lost when the routine returns

and that's exactly what this does:

```
Object *newObject(const int id, const float xcen, const double y
   Object obj;
   ...
   return @\color{red}\&@obj;
}
```

The solution is to get a pointer to a piece of persistent memory. C provides this via a call to malloc:

```
#include <stdlib.h>

Object *newObject(const int id, const float xcen, const double y
   Object @\color{red}*@obj = malloc(sizeof(Object));
   ...
   return obj;
}
```

Malloc

After some time spent in gdb, I remembered:

- Variables in subroutines are irretrievably lost when the routine returns

and that's exactly what this does:

```
Object *newObject(const int id, const float xcen, const double y
   Object obj;
   ...
   return @\color{red}\&@obj;
}
```

The solution is to get a pointer to a piece of persistent memory. C provides this via a call to malloc:

```
#include <stdlib.h>

Object *newObject(const int id, const float xcen, const double y
   Object @\color{red}*@obj = malloc(sizeof(Object));
   ...
   return obj;
}
```

Pointers-to-Pointers

How about:

```
#define NOBJECT 1000                          // Maximum number of obj

Object *objs[NOBJECT];                        // our objects
```

Pointers-to-Pointers

How about:

```
#define NOBJECT 1000                      // Maximum number of obj

Object *objs[NOBJECT];                    // our objects
```

We know that

```
int i[10];
```

and

```
int *i;
```

are equivalent (except for the question of where the data lives).

Pointers-to-Pointers

How about:

```
#define NOBJECT 1000                    // Maximum number of obj

Object *objs[NOBJECT];                  // our objects
```

We know that

```
int i[10];
```

and

```
int *i;
```

are equivalent (except for the question of where the data lives). We can provide the needed storage with:

```
int *i = malloc(10*sizeof(int));
```

Pointers-to-Pointers

How about:

```
#define NOBJECT 1000                          // Maximum number of obj

Object *objs[NOBJECT];                        // our objects
```

We know that

```
int i[10];
```

and

```
int *i;
```

are equivalent (except for the question of where the data lives). We can provide the needed storage with:

```
int *i = malloc(10*sizeof(int));
```

So:

```
n = ...;
Object **objs = malloc(n*sizeof(Object *)); // our objects
```

Pointers-to-Pointers

How about:

```
#define NOBJECT 1000                        // Maximum number of obj

Object *objs[NOBJECT];                      // our objects
```

We know that

```
int i[10];
```

and

```
int *i;
```

are equivalent (except for the question of where the data lives). We can provide the needed storage with:

```
int *i = malloc(10*sizeof(int));
```

So:

```
n = ...;
Object **objs = malloc(n*sizeof(Object *)); // our objects
```

If you don't know `n` *a priori*, look up the system call realloc .

Pointers-to-Pointers

How about:

```
#define NOBJECT 1000                    // Maximum number of obj

Object *objs[NOBJECT];                  // our objects
```

We know that

```
int i[10];
```

and

```
int *i;
```

are equivalent (except for the question of where the data lives). We can provide the needed storage with:

```
int *i = malloc(10*sizeof(int));
```

So:

```
n = ...;
Object **objs = malloc(n*sizeof(Object *)); // our objects
```

If you don't know `n` *a priori*, look up the system call realloc . There is also calloc, but I never use it — initialising to 0x0 is a blunt weapon.

Free

As I said, malloc returns persistent memory, so it's important to return it to the system when you've finished with it:

```
for (int i = 0; i != n; ++i) {
   free(objects[i]);
}
free(objects);
```

Free

As I said, malloc returns persistent memory, so it's important to return it to the system when you've finished with it:

```
for (int i = 0; i != n; ++i) {
   free(objects[i]);
}
free(objects);
```

Failure to do so results in a *memory leak*.

Free

As I said, malloc returns persistent memory, so it's important to return it to the system when you've finished with it:

```
for (int i = 0; i != n; ++i) {
   free(objects[i]);
}
free(objects);
```

Failure to do so results in a *memory leak*.
Freeing a piece of memory more than once usually has catastrophic consequences; don't do it.

Free

As I said, malloc returns persistent memory, so it's important to return it to the system when you've finished with it:

```
for (int i = 0; i != n; ++i) {
   free(objects[i]);
}
free(objects);
```

Failure to do so results in a *memory leak*.
Freeing a piece of memory more than once usually has catastrophic consequences; don't do it.
Another failure mode is that sometimes malloc can't give you the memory you want. In this case it returns `0`, conventionally written `NULL`.
There is not much you can do when this happens, so a reasonable response is to abort:

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

Object *newObject(const int id, const float xcen, const double y
   Object @\color{red}*@obj = malloc(sizeof(Object));
```

struct and typedef

I wrote

```
struct Object {
    int id;                                     // Object IDs
    ...
};
typedef struct Object Object;
```

struct and typedef

I wrote

```
struct Object {
    int id;                              // Object IDs
    ...
};
typedef struct Object Object;
```

This can be abbreviated

```
typedef struct {
    int id;                              // Object IDs
    ...
} Object;
```

struct and typedef

I wrote

```
struct Object {
   int id;                              // Object IDs
   ...
};
typedef struct Object Object;
```

This can be abbreviated

```
typedef struct {
   int id;                              // Object IDs
   ...
} Object;
```

There are two reasons why you might not always want to omit the name in struct Object {...} :

- structs can contain pointers to themselves:

  ```
  typedef struct List {
      struct List *prev, *next;
      ...;
  ```

struct and typedef

I wrote

```
struct Object {
    int id;                              // Object IDs
    ...
};
typedef struct Object Object;
```

This can be abbreviated

```
typedef struct {
    int id;                              // Object IDs
    ...
} Object;
```

There are two reasons why you might not always want to omit the name in struct Object {...} :

- structs can contain pointers to themselves:

  ```
  typedef struct List {
      struct List *prev, *next;
      ...;
  ```

struct and typedef

I wrote

```
struct Object {
   int id;                              // Object IDs
   ...
};
typedef struct Object Object;
```

This can be abbreviated

```
typedef struct {
   int id;                              // Object IDs
   ...
} Object;
```

There are two reasons why you might not always want to omit the name in struct Object {...} :

- structs can contain pointers to themselves:

```
typedef struct List {
     struct List *prev, *next;
     ...;
```

CCD Data again

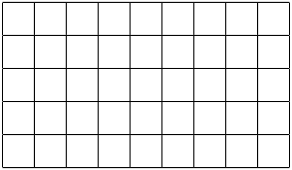
You will recall a code fragment that looked like this:

```
U16 data[4096][2048];
...
U16 const peak = data[y][x];               // the (x, y)th pixel
...
void debias(U16 data[][2048], const int nrow, const int ncol);
```

We are now in a position to do better.

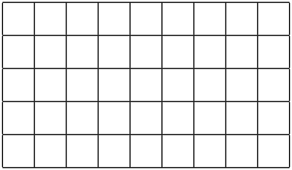
CCD Data again

We need a 2-D `ncol*nrow` array of type `U16`

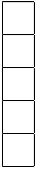


CCD Data again

We need a 2-D `ncol*nrow` array of type `U16`

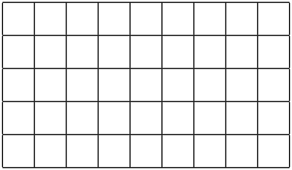


Create an 1-D array `rows` of type `U16 *` and dimension `nrow`

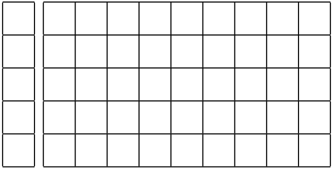


CCD Data again

We need a 2-D `ncol*nrow` array of type `U16`



Create an 1-D array `rows` of type `U16 *` and dimension `nrow`



Make each element of `rows` point to the start of a row, e.g. `rows[1]` points to the first pixel in the second row of data. Then `rows[1][2]` is the value of the `(2, 1)` pixel.

CCD Data again

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

typedef uint16_t Pixel_t;

struct Image {
     Pixel_t **rows;
     int nrow, ncol;
};
typedef struct Image Image;

Image *newImage(const int ncol, int const nrow) {
     Image *im = malloc(sizeof(Image));   // the Image
     assert (im != NULL);

     im->rows = malloc(nrow*sizeof(Pixel_t *)); // pointers to r
     assert (im->rows != NULL);
```

CCD Data again

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

typedef uint16_t Pixel_t;

struct Image {
     Pixel_t **rows;
     int nrow, ncol;
};
typedef struct Image Image;

Image *newImage(const int ncol, int const nrow) {
     Image *im = malloc(sizeof(Image));   // the Image
     assert (im != NULL);

     im->rows = malloc(nrow*sizeof(Pixel_t *)); // pointers to r
     assert (im->rows != NULL);
```

Matrices

Now we know enough to write ourselves a matrix library

```
  typedef struct Matrix {
     float **data;
     int nrow;
     int ncol;
  };

  Matrix *newMatrix(int nrow, int ncol);
...
  Matrix *A = newMatrix(10, 10);
  Matrix *B = newMatrix(10, 10);
```

Matrices

Now we know enough to write ourselves a matrix library

```
  typedef struct Matrix {
     float **data;
     int nrow;
     int ncol;
  };

  Matrix *newMatrix(int nrow, int ncol);
...
  Matrix *A = newMatrix(10, 10);
  Matrix *B = newMatrix(10, 10);
```

However, you can't write:

```
Matrix *sum = A + B;
```

Matrices

Now we know enough to write ourselves a matrix library

```
typedef struct Matrix {
    float **data;
    int nrow;
    int ncol;
};

Matrix *newMatrix(int nrow, int ncol);
...
Matrix *A = newMatrix(10, 10);
Matrix *B = newMatrix(10, 10);
```

However, you can't write:

```
Matrix *sum = A + B;
```

... not until you switch to C++.

[3] In C++ you'd probably use a const variable but scoping rules are different in C, so a macro is appropriate

[4] this isn't quite true; it could pass a limited number of variables by using *registers*