

C

Robert Lupton¹

23 September 2010

¹LooChao@gmail.com

Outline

- 1 Introduction
 - types

- 2 Test
 - test2

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 V
/*
 * 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 V
/*
 * 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 fi
```

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).

test

1

test

2