University of Arizona

CSc 340

Foundations of Computer Systems

Christian Collberg

March 20, 2001

C Types & Functions

Slide 15–1

Integral Types

- C has several integral (integer) types:

Type	Size (bytes)
`char`	1
`short`	2
`int`	4* (one word)
`long`	4
`enum`	4 (same as int)

- Integral types are signed by default.

Slide 15–2

Integral Types...

- Unsigned types use the `unsigned` modifier, e.g.

  ```
  unsigned int.
  ```

- C does not specify the size of `int`, except that it can't be smaller than `short` or larger than `long`. Typically it's the size of a word, e.g. 4 bytes on a SPARC. 64-bit machines often use `long long` for 64-bit quantities.
- The `enum` type is used to create an enumeration. An enumeration is an `int` that can take on a limited set of values. C considers an enumeration an `int` when type-checking.

Slide 15–3

Integral Types...

- Types can be converted by casting. Suppose you have:

  ```
  short s;
  int i;
  ```

  If you assign one to the other, e.g.

  ```
  i = s;
  s = i;
  ```

  C will implicitly convert the types. Otherwise you can do it explicitly:

  ```
  i = (int) s;
  s = (short) i
  ```

Slide 15–4

Integral Types...

- I prefer explicit conversions so I know that I meant to do it.
- Note that the `"byte"` type is named `char`, because it is typically used to hold a character.
- `char` is an integral type, however, so you can use as a (small) integer to hold values and perform arithmetic.

Slide 15–5

Function Declaration

- A function should be declared before it is used in a C file.
- The declaration provides the C compiler with information about the function's return value and parameters.
- The declaration is often called the function's *prototype*.
- The format is (brackets '`[ ]`' `denote an optional word`):

```
[static|extern] type name(parameter list);
```

Slide 15–6

Function Declaration

- For example

```
int average(int a, int b);
```

- This declaration tells the C compiler that the function `average` returns an integer of type `int`, and has two parameters both of type `int`.
- The parameter names aren't needed in the declaration, but are often useful to the programmer.

Slide 15–7

Function Declaration...

- The optional modifier `static` indicates that the function is private to the current file.
- The optional modifier `extern` indicates that the function is actually defined in another source file.
- Function declarations are often put in a header file.
- If a function isn't declared before it is used, C assumes the function returns an `int`, and doesn't check the parameter types.

Slide 15–8

Function Definition

- A function definition consists of a function header followed by the function body. The header has the same format as the declaration above. If you define a function before it is used, you don't need the declaration although it's usually best to have one anyway.
- The function body consists of variable definitions followed by statements. Variable definitions have the following form:

```
[static] type name-list;
```

Slide 15–9

Function Definition...

- The static modifier indicates that the variable is stored in permanent storage, i.e. the next time the function is called the variable has the same value as the last time.
  1. `type` is the type of the variable.
  2. `name-list` is a comma-separated list of variable names.
  3. `Variables` can be set to initial values, e.g.:

     ```
     int foo = 10;
     ```
  4. Variables that are not initialized have an undefined value.

Slide 15–10

Global Variables

- A variable declared outside of a function is a global variable – it is allocated permanent storage and is accessible at least to the functions in the same source file.
- The modifier static causes the variable to be private to the current file:

```
static intcount = 0;
```

- Variables with initial values are initialized before the program runs.

Slide 15–11

Global Variables...

- The modifier extern indicates that the variable is defined in another source file:

```
extern int count;
```

- The variable must be defined in one (and only one) source file, and it cannot be `static`. Do not put the definition in a header file.

Static Overload

- The C designers loved the word `"static"`. It is used for three different purposes in C:
  1. To make a global variable private to the current file.
  2. To make a function private to the current file.
  3. To allocate a local variable in permanent storage, so it retains its value between function invocations.

Slide 15–12

Call-by-Value Parameters

- All function parameters in C are passed call-by-value. This means that the callee function gets its own copies of its parameters – any changes it makes to the parameters are not visible to the caller.
- Thus the function `swap` on the next slide doesn't do what you want:

Slide 15–13

`swap`, Version I

```
void swap(int x, int y);
void main(void) {
        int a, b;
        a = 10;
        b = 17;
        swap(a, b);
}
void swap(int x, int y) {
        int tmp;
        tmp = x;
        x = y;
        y = tmp;
}
```

- Since `swap` gets copies of `a` and `b`, the values of `a` and `b` in the parent are unchanged after the call to `swap`.

Slide 15–14

`swap`, in C++

- In `C++` you can fix the problem using reference parameters, which are passed call-by-reference:

```
void swap(int &x, int &y);
void main(void) {
   int a=10, b=17;
   swap(a, b);
}
void swap(int &x, int &y){
   int tmp=x;
   x = y; y = tmp;
}
```

- We'll see how to implement `swap` correctly in C in a minute, but to do so we first have to learn about pointers.

Slide 15–15

Slide 15–16

Pointers

- A pointer is a variable that contains the address of another variable. A pointer variable is created using the '`*`' character when declaring it:

```
int *ptr;
```

- In this example `ptr` is a pointer variable that contains the address of a variable of type `int`.
- Declaring a pointer does not allocate storage for the pointer to point to. The above ptr variable initially contains an undefined value (address).

Slide 15–17

Pointers. . .

- The '`*`' character is also used to refer to the value pointed to by the pointer (dereference the pointer). Suppose we want to store the value 10 into memory at the address contained in ptr. Do the following:

```
*ptr = 10;
```

- Similarly, to get the value stored at ptr do the following:

```
y = *ptr + 5;
```

- The value of `y` is now 15.

Slide 15–18

Pointers. . .

- How do you set the value of ptr in the first place? Use the '`&`' character to get the address of a variable, e.g.:

```
int        y;
int        *ptr;
ptr = &y;
*ptr = 21;
```

- The value of `y` is now 21.
- What does the following do?

```
*(&y) = 42;
```

Slide 15–19

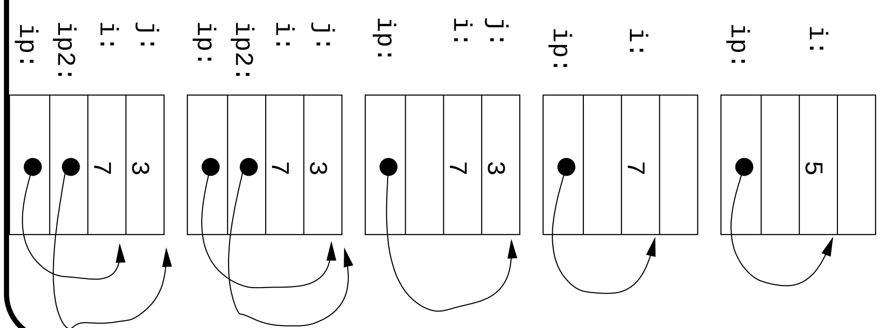
```
int *ip;
int i = 5;
ip = &i;
```

```
*ip = 7;
```

```
int j = 3;
ip = &j
```

```
int *ip2=ip;
```

```
ip = &i;
```



Slide 15–20

Pointer Types

- C cares (sort of) about pointer types; the type of a pointer must match the type of what it points to. The following will cause the C compiler to print a warning:

```
short   x;
int     *ptr;
ptr = &x;
```

- I say it "sort of" cares because while it issues a warning, it goes ahead and compiles the (probably incorrect) code.

Slide 15–21

Pointer Types...

- If you know you want to do the above, you can stop the compiler from complaining by casting the pointer, e.g.:

```
short               x;
int                 *ptr;
ptr = (int *) &x;
```

- Your program may still not do the right thing, however.

Slide 15–22

Pointer Types...

- Casting pointers is a popular thing to do in C, as it provides you with a certain amount of flexibility. The type ⌜`void *`⌝ is a pointer to nothing at all — it contains a memory address, but there is no type associated with it. You can't do:

```
int         x;
void        *ptr = (void *) &x;
*ptr = 10;
```

for example.

Slide 15–23

Pointer Types...

- Instead, you have to cast the pointer to the proper type:

```
int         x;
void        *ptr = (void *) &x;
*((int *) ptr) = 10;
```

Slide 15–24

`void *`

- Why is ⌜`void *`⌝ useful? Consider the `malloc` routine. It returns the address of a block of memory. You can use this memory to store information of any type – an array of characters, an integer, a structure, etc. malloc returns type ⌜`void *`⌝, you cast it into the proper type:

```
extern   void *malloc(unsigned int size);
extern   void free(void *ptr);
int      *ptr;
ptr = (int *) malloc(sizeof(int));
*ptr = 10;
free((void *) ptr);
```

Slide 15–25

Swap Revisited

```
void swap(int *x, int *y);
void main(void){
    int a=10, b=17;
    swap(&a, &b);
}
void swap(int *x, int *y){
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
```

- The parameters to `swap` are pointers to integers. `swap` dereferences these pointers to exchange the values to which they point. `main` passes the addresses of the values to be swapped to `swap`.

Slide 15–26

Pointer Example I

```
#include <stdio.h>
int j, k;
int *ptr;
int main(void) {
  j = 1;
  k = 2;
  ptr = &k;
  printf("\n");
  printf("j has the value %d and is stored at %p\n",j,&j);
  printf("k has the value %d and is stored at %p\n",k,&k);
  printf("ptr has the value %p and is stored at %p\n",ptr,&ptr);
  printf("The value of the integer pointed to by ptr is %d\n",
          *ptr);
}
```

Slide 15–27

Pointer Example I. . .

```
$ gcc ptr.c
$ a.out

j has the value 1 and is stored at 0x8049714
k has the value 2 and is stored at 0x804971c
ptr has the value 0x804971c and is stored at 0x8049718
The value of the integer pointed to by ptr is 2
```

Swap Revisited...

- Java does not have pointers, nor does it have reference parameters. How do you implement swap in Java?