Programming in C

- C was designed by Dennis Ritchie at AT&T Bell Labs to use as a systems language to implement Unix. It became operational in 1971.
- Knowing Java before learning C makes C very easy.
- Most of the Java syntax is stolen from C.
- Function and procedure headers are *declared* forward as prototypes and fully *defined* later.
- The function definition can replace its declaration.
- C has pointers which are references or addresses
  —Java cleverly pretends not to have them.
- C does not have *classes*. Its object is a primitive datastructure without methods called a **`struct`**.
- C either returns a value or what is also a value, i.e., a pointer like a reference to a value or **`struct`** object:
  —Java pretends to return an object.
- In C all arguments are passed as values. Since it is possible to pass the address of a variable as a value, call by reference is also possible.
- C executables are likely to run faster than their Java equivalents.

Comparing C and Java

- Arithmetic, assignment, boolean relations, **for**, **if**, **while**, **switch**, and function invocation are all similar if not exactly the same.
- Priority of operators is the same.
- Functions and procedures have different headings.
- Strings in C are arrays of characters.
- You need to understand pointers in code like this.

```
1 int *xP;
2 int x;
3   x = 123;
4   xP = &x;
5   *xP *= 3;
6
7   printf ("x = %d\n", x);
```

- What is the value of **x** that is printed?

Syntax in common with Java

Operators in priority order:

- Parentheses, brackets, member selection, and postfix **++**, and **--**:
  (. . .)—function invocation,
  [. . .]—subscript evaluation,
  .—element selection,
  ->—element selection via pointer,
  ++, **--**—as postfix operators,
- Addressing: *****, **&**, **sizeof**,
- Arithmetic: **++**, **--**—as prefix operators,
  *****, **/**, **%**—multiplicative operators
  +, **-**—additive operators.
- Bitwise shift:**<<**, **>>**,
- Relational: **==**, =!, **<**, **>**, **<=**, **>=**
- Bitwise: **&**, ^, **|**, Logical: **&&**, **||**, Ternary conditional:**?:**,
- Assignment: **>>**, **>>=**, **&=**, **|=**, ^**=**, **=**, **+=**, **-=**, ***=**, **/=**, **%=**
  Please check.

Further syntax in common with Java

```
1  if( ){                          switch( ){
2     ... ;}                       case 1:
3  else{                              ...; break;
4     ... ;}                       case 2:
5                                     ...; break;
6  if(){                           default:{
7     ... ;}                          ... ;}
8
9  while( ){                       do{
10    ... ;}                          ... ;}
11                                   while( )
12
13 for(i=1; i<=100; i++){
14    ... ;}
15
16 break;  continue;  return;
```

switch also has a similar syntax, including a **default** case.

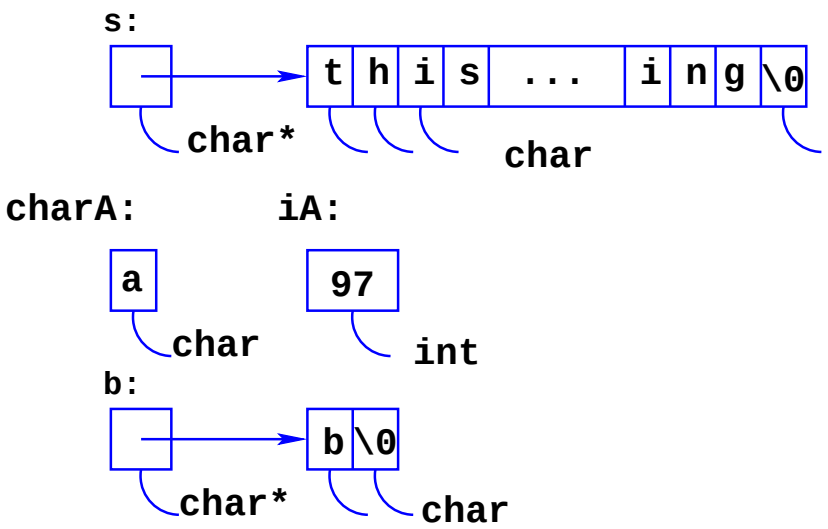
‘Strings’ in C

String functions are provided by the package **string.h**

In C a ‘string’ is an array of **char** ending with the character **’\0’**

Note that **(int)’0’** ⇒ **48**

```
1  char s[] = "this is a string";
2  char charA = 'a'; /* but */
3  char b[] = "b";
4  int iA= 'a';
```



'Strings' in C: the package `string.h`

Include the package **`string.h`** with

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

For `char * dest, src, string` and for `const char * s1, s2, s` and for `int c` and for `size_t n` this provides the functions

```
int strcmp(s1, s2); int strncmp(s1, s2, n); char *stpcpy(dest, src);
char *strcat(dest, src); char *strchr(s, int c); int strcoll(s1, s2);
char *strdup(s); char *strcpy(dest, src); char *strncpy(dest, src, n);
char *strfry(string); size_t strlen(s); char *strncat(dest, src, n);
size_t strcspn(s, reject);
```

`strcmp` and `strncmp` compare strings `s1` and `s2`, yielding an `int` which is `-1` if `s1 < s2`, `0` when `s1 == s2`, and `1` when `s1 > s2`. `strncmp` only compares the first `n` characters.

`strcat` and `strncat` concatenate their arguments. `strlen` returns the length of a string. `strcpy` and `strncpy` copy their arguments.

`strdup(s)` returns a pointer a `malloc`ed duplicate of its argument string `s`.

Why do we prefer C?

- C was designed to avoid using assembly language for systems programming
- C is a high-level language that does low level things:
  —Supports *bit manipulation* with
  `<<, >>, &, |, ~, ^, <<=, >>=, ^=, &=, |=`
  —can *access registers* or memory
  —**`struct`** supports *methodless* complex data type
  —supports *separate compilation* of parts
- Low-level control of machine
  —memory management by *programmer*
  —*programmer* detects and handles run time errors
  —*programmer* controls everything
- Performs better than Java
- C a popular choice for *systems programming*.
- Algol was used by Burroughs, C used by Unix, C++ by Linux.

Goals of Tutorial

- Learn what is 'new' in C:
  —one *learns* C one is not taught to program in C
- Warning about errors that can be avoided:
  —C not very type strict, use **`lint`**
  —room for mistakes
  —C programming requires disciplined programming
- Useful information and example files
  —use `Makefile`
  —compile modules separately
  —compile and execute with `Makefile`
  —**`printf`** for debugging
  —conditional code inclusion using

```
1 #ifdef DEBUG
2 ... Code omitted when no #define DEBUG
3 #endif
```

Hello World

The file `hello.c`

```
1 /* Hello World program */
2 #include <stdio.h>
3
4 int main(void){
5   printf("Hello␣World.\n");
6   return 0;}
```

The `Makefile`

```
1 P = hello
2 all:
3   g++ $(P).c -o $(P)
4   ./$(P)
5 clean:
6   -rm -rf $(P) *.o core
```

Run by typing **`make`** or **`make all`**,
and **`make clean`** gets rid of the rubbish.

Primitive types

- Integer types:
  —**`char`**: for representing characters or one byte data
  —Java can use 16 bits
  —Beware: Solaris the owners of Java sometimes use 8 bit characters
  —**`short`**, **`int`**, **`long`**, **`unsigned int`**:
  are similar to Java's versions of **`int`** also architecture dependent
  —are implicitly defined as signed or explicitly defined as **`unsigned`**
- Floating point types:
  `float` and **`double`** as in Java.
- Boolean is done by using **`short`**, **`char`** or **`int`**

$$\neq 0 \Rightarrow \texttt{true}$$
$$= 0 \Rightarrow \texttt{false}$$

Primitive types: examples

```
1 char c = 'z';
2 char M = 1000;
3 short biggest = 32767;
4 int i = -2345432;
5 unsigned int ui = 1000000000;
6 float pi = 3.141593; /* 7 digits are exact */
7 double _pi = 0.31415926535897 93e+1; /* 15 digits are exact
```

Some of the older C compilers do not allow assignment in a definition.

Using a 32-bit int

```
1 unsigned int smallEnough = 4294967295;
2 unsigned int tooBig = 4294967296;
```

Using a 16-bit int

```
1 unsigned int justFits = 65535;
2 unsigned int justTooBig = 65536;
```

`printf` function

- Syntax: `printf(format, arg1, arg2, ...)`
- Format: text to be displayed and % tokens to be filled by arguments

```
%d int %i int %x hex %u unsigned
%s string %% for a % %ld long %lld long long
%c char %f float %e float %lf double
```

Examples:

```
printf("%s␣has␣%d␣students.\n", "CSC311", 57);
```

⇒ **CSC311 has 57 students.**

Produce output in neat columns:

```
printf("%5d%12s%9.2f\n", 29, "nectarines", 27.35);
```

⇒

```
   29 nectarines 27.35
1234512345678912123456.12
|  5 ||  12 ||9=6+1+2|
```

- Use **`printf`** liberally while debugging.

`enum`: enumerated data-types—back in Java

- `enum months{jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};`
- Each element of `enum` gets an `int` value, such that `jan` $\equiv 0$, `feb` $\equiv 1, \ldots,$ `dec` $\equiv 11$
- `enum`s are manipulated as `int`s
- `months theMonth = feb;`
  `theMonth = (theMonth + n) % 12;`
  `theMonth` becomes the `n`th month after `theMonth`, i.e., `feb` is turned into `aug` if `n` is 6,
  and `aug` is turned into `may` when `n` is 9.
- The bad news is that `theMonth` prints as an integer.
- In `enum months{jan=1, feb=3, mar, apr};`
  the value assigned to `apr` is 5
- `enum` thrown out by James Gosling but later reinstated in Java
- An unintended use for `enum` is to define `int` constants

Pointers

- Similar to references in Java.
- An address of a variable is the location in memory where that variable is stored
- A pointer is a variable containing an address
- **`type*`** means *pointer* to variable of type **`type`**.
- Example

```
1 int x; /* Declares int variable x */
2 int* xPointer; /* xPointer points to rubbish */
3   xPointer = &x; /* xPointer points to int x */
4   *xPointer = 7; /* Now, same as 'x = 7;' */
```

- **`&`** *address* operator
  `&x` is the address of **`x`**
- **`*`** *dereference* operator
  `*xPointer` is the *value* of the thing **`xPointer`** points to, i.e., **`x`**'s value, when we have set **`xPointer = &x;`**

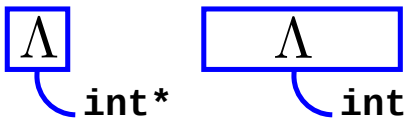
Pointers in C

```
1 int * xPointer;
```

xPointer:

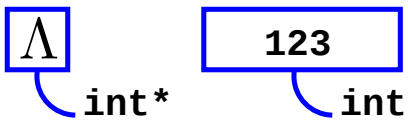

```
1 int x;
```

xPointer: x:




```
1   x = 123;
```

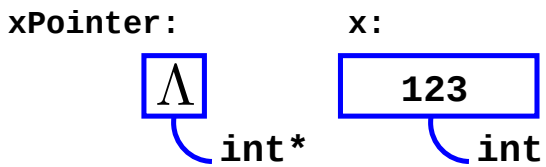
xPointer: x:



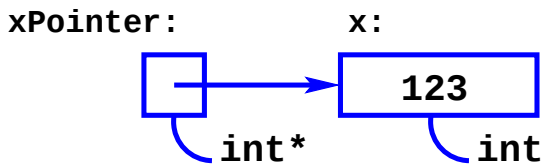
Pointers in C

```
1 int * xPointer;
2 int x;
3   x = 123;
4   xPointer = &x;
5   *xPointer *= 3;
```

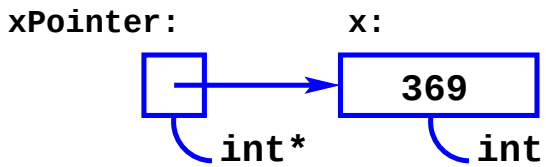
```
1   x = 123;
```




```
1   xPointer = &x;
```




```
1   *xPointer *= 3;
```



More on pointers

It is wrong to use pointers that are uninitialized
—program will crash with a **segmentation fault**
—so, always initialize pointers at declaration,
—either with **NULL** or with a reference to some object.
—before using test if a pointer is not **NULL**

```
1 int* xPointer = NULL;
2   . . .
3   if (xPointer == NULL){
4     printf("xPointer␣is␣NULL.\n");
5     exit(1);}
```

Structures

- The record type of C, like Java classes with only members: no methods.

```
1  enum months{jan=1,feb, mar, apr, may, jun,
2           jul, aug, sep, oct, nov, dec};
3  months theMonth;
4  struct birthday {
5    char* name;
6    int year;
7    enum months month;
8    int day; }; /* Note the ;s */
9  struct birthday mybirthday =
10                {"Wills",1980, mar, 21};
11 char FirsLetter = mybirthday.name[0];
12    /* is a W */
13 theMonth = mybirthday.month;
14    /* is mar */
```

More on structures

- Structures can hold any element with a defined type
- Structures may refer to themselves:

```
1 struct list_node{
2   int data;
3   struct list_node* next;};
```

- Use **->** for dereference and take element:

```
1 struct list_node n = {10, NULL};
2 struct list_node* nPointer = &n;
3 printf("The␣data␣is␣%d\n", nPointer->data);
```

nPointer->data is the same as **(*nPointer).data**, which is the same as **n.data**

Data-type synonyms

- Syntax: **typedef type alias**;
- Example:

```
1 typedef int boole; boole b;
2 typedef int* intPointer;
3 intPointer p; /* p is a pointer to int */
4 typedef struct list_node list_Node;
5 /* list_Node = alias of struct list_node */
6 struct list_node{
7   int data;
8   list_Node* next; /* quite legal */};
```

- Advantage: easier to remember, cleaner code.

`void*` and type conversion

- Type conversion syntax:
  `(new_type)expression_old_type`
- Examples:

```
1 float f = 1.2;
2 int i = (int)f;/* i assigned value 1 */
3 char c = i;/* implicit conversion from int to char */
4 float g = i; /* implicit conversion; g=1.0 */
```

- Extremely useful conversion is to and from `void*`—pointer to unspecified type:

```
1 #include <string.h>
2 char str1[100]; char str2[100];
3 memcpy((void*) str2, (void*) str1, 100);
```

- Always do explicit conversions.

Memory allocation and deallocation

Global variables:

- Declared outside a function.
- Memory allocated statically before execution.
- Initialization done before program execution.
- Deallocated when program terminates.
- Name is unique for the whole program
  —C has flat name space.

More memory allocation and deallocation

Local variables:

- Declared in the body of a function.
- Space allocated when entering the function.
- Initialization before function starts executing.
- Space automatically deallocated when function returns:
  Referring to a local variable by means of a pointer after the function returned leads to unwanted results.
- Names are local to the function.

More memory allocation and deallocation

Heap variables:

- Allocate / free memory explicitly

  Allocate memory: the function **`void* malloc(int)`** behaves like **`new`** in Java, e.g. allocate an integer to which a pointer points
  `iPointer = (int*) malloc(sizeof(int));`

  Free memory: the function **`void free(void*)`** frees memory, e.g. free the memory that the pointer **`iPointer`** points to
  `free(iPointer);`
- The programmer must deallocate memory when it is not in use any more to prevent memory overflow. Java does this for you.
- If you are looking for trouble the deallocate the same memory more than once.

Memory allocation and deallocation: example

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int counter; /* global allocation counter */
4  void main(void){
5  int* iPointer; /* local variable of type int* */
6  /* allocate space to hold an int */
7    iPointer = (int*) malloc(sizeof(int));
8    counter++; /* check if successfull */
9    if (iPointer == NULL)
10     exit(1); /* not enough memory available, exiting */
11   *iPointer = 4; /* store 4 into memory */
12   free(iPointer); /* deallocate memory */
13   counter--; }
```

Functions

- Functions provide modules that easy to code and debug. These modules are reusable. They can be invoked recursively.
- Technically C passes its arguments only by *by value*:
  a copy of the argument is handed to the function, but we can get *by reference* by handing a pointer argument to the function
- So Strictly speaking, C only passes arguments by value, and since a pointer can be passed by value while its corresponding parameter in the invoked function is marked as a pointer, we say C also handles *call by reference*.
- Strictly speaking, only a value is ever returned
  —but this value may be a pointer
  —this is quite dangerous because the object that the pointer points to may be destroyed when the function is exited

Functions—simple example

```
1  #include <stdio.h>
2  /* sum and psum's function declaration or prototype */
3  int sum(int a, int b);
4  int psum(int* a, int* b);
5
6  /* definition of main() */
7  void main(void){
8    /* call sum with arguments 4 and 5 */
9    int total=sum(2+2, 5);
10   printf("The␣total␣is␣%d.\n", total); }
11
12 /* definition of function sum;
13   has to match declaration signature */
14 int sum(int a, int b){ /* arguments passed by value */
15   return (a+b); /* return by value */ }
16
17 /* definition of function psum;
18   matches declaration signature */
19 int psum(int* a, int* b){
20 /* arguments passed by reference */
21   return ((*a)+(*b)); }
```

Why pass by reference?

```
1  #include <stdio.h>
2  void swap(int, int);
3  int main(void){
4  int a = 7, b = 11;
5     swap(a, b);
6     printf("a = %d and b = %d\n", a, b);
7     return 0;}
8
9  void swap(int x, int y){ /* pass by value */
10 int temp;
11    temp = x;
12    x = y;
13    y = temp; }
14
15 $ ./swaptest
16 a = 7 and b = 11
```

swap has *no* effect.

Why pass by reference?

```
1  #include <stdio.h>
2  void swap(int*, int*);
3  int main(void){
4  int a = 7, b = 11; // X
5  int* pa = &a, *pb = &b // int * pa=&a, pb=&b;
6    swap(pa, pb); /* same as swap(&a, &b); */
7    printf("a = %d and b = %d\n", a, b);
8    return 0;}
9
10 void swap(int* pX, int* pY){/* pass as */
11 int temp; /* references */
12   temp = *pX;
13   *pX = *pY;
14   *pY = temp; }
15
16 ./swaptest2
17 a = 11 and b = 7
```

swap works.

Pointer to Function—functions as variables

```
1  #include <stdio.h>
2  void callee(int p){ printf("callee(%d)\n", p); }
3  void g(int x) { printf("g(%d)\n", x); }
4  void f(int x) { printf("f(%d)\n", x); }
5
6  void invoke(void (*f)(int), int q){
7    printf("invoking␣f(%d)\n", ++q);
8    f(q); } /* invoke f with q as argument */
9
10 int main(void){
11   f(1);
12   callee(2);
13   invoke(callee, 3); /* invoke callee(3) */
14   invoke(f, 4); /* invoke f(4) */
15   invoke(g, 5); /* invoke g(5) */
16   return 0; }
```

Module support

```
1 /* include standard I/O library declaration */
2 #include <stdio.h>
3 /* include my declarations */
4 #include "myheader.h"
```

Symbol definition—behaves like final in Java

```
1 #define DEBUG 0
2 #define MAX_LIST 100
3 if (DEBUG)
4   printf("Max␣length␣of␣list␣is␣%d.\n", MAX_LIST);
```

Conditional compilation

```
1 #ifdef DEBUG
2   printf("DEBUG:␣reached␣line␣%d.\n", _LINE_);
3 #endif
```

If DEBUG is defined with *any* value the **printf** will be compiled.

Program with header files

- File `symboltable.h`:

```
1 enum {tableSize = 127};
2 enum varType {int, real, string, boolean};
3 int insertString(char * s);
4 int insertInt(int i);
5 int delete(char * s);
6 int pos = 0;
7 struct table {
8   char * symbol;
9   varType varT} table[tableSize];
```

- Code belongs to `.c` files not to `.h` files.

Program with header files

By convention, function and procedure headers are *declared* first in a header, i.e., a **.h** file—pronounced *dot aich file*.
These headers are known as the *prototypes* of functions and procedures.
The body of their code should be fully *defined* in a separate **.c** file with the same name as the header.

- File `symboltable.c`:

```
1 #include <stdio.h>
2 #include "symboltable.h"
3 int insertString (char * s){
4   table.symbol[pos++] = strdup(s);
5   table.varT = string; }
6 int insertInt(int i){
7   . . . }
8 int delete (char * s){
9   . . . }
```

A program with header files

- File `main.c`:

```
1 #include "symboltable.h"
2 void main(void){
3   . . .
4   insertString(symbol);
5   . . .
6   }
```

- We need to do the following to create and run **`primes`**

```
1 g++ -cg primes.c makePrimes.c list.c
2 g++ -o primes primes.o makePrimes.o list.o
3 ./primes
```

- Have to compile files `symboltable.c` and `main.c` to produce object files `symboltable.o` and `main.o`.
- Must link `symboltable.o`, `main.o`, and necessary system files to produce executable object.
- Use **`make`** to compile.

A more complex `make` file

```
1  CC=gcc
2  CFLAGS=-c -Wall -g
3  LDFLAGS=-g
4  SOURCES=primes.c makePrimes.c list.c
5  OBJECTS=$(SOURCES:.c=.o)
6  P = primes
7
8  all: $(SOURCES) $(P)
9    ./$(P)
10
11 $(P): $(OBJECTS)
12   $(CC) $(LDFLAGS) $(OBJECTS) -o $@
13
14 .c.o:
15   $(CC) $(CFLAGS) $< -o $@
16
17 clean:
18   -rm -rf $(P) a.out core *.o
```

Makefiles

- Read **man make** or the **make** manual.
- **make -n** shows the commands that **make** executes.
- Use **make -f ownMakefile** to run **ownMakefile**
- Learn a great deal about **make** by figuring out what **make -d** produces.

The $ macro, and special macros in make

The **$** macro extracts text from the variable: if
P=anything
then using
$(P) ⇒ **anything**

make has some built in macros called *special macros*
$(CC) ⇒ **cc**, unless you assign something else to it.
$(CFLAGS) are the options built in for **$(CC)**.
$@ is the full name of the current rule's target.
$? is a list of files for the current rule which need to be built again because they have changed since the last **make** was run
$< is the source file of the current rule.

make can substitute
Given the rule **OBJS = list.o makePrimes.o primes.o** then the rule **$(OBJS:.o=.c)** does the substitution of the suffix **.o** with **.c** yielding **list.c makePrimes.c primes.c**

Real programmers are always alert

- Initialize variables before use
- Initialize pointers—a **`segmentation fault`** is often caused by a null pointer especially an uninitialized pointer, or an array that has left its bounds,
- Don't kill pointers while they are still in use
- Do not return local variables of functions by reference
  do not dereference pointers before initialization or after deallocation
- C has no exceptions—error handling is explicit

If you get stuck

- RTFM
- Ask Google
- Ask a friend
- Ask one of the lecturers
- Keep a notebook—in a '**`notes`**' file on the machine—of everything you had to look up before you could do it
- If you are really stuck then ask the question on a linux forum
- RTFM again. Ask Google. Ask a friend. ...

Acknowledgments

These notes are a free adaptation by Reg Dodds of the notes for →
CS 414 / CS 415
by Niranjan Nagarajan
Department of Computer Science
Cornell University
niranjan@cs.cornell.edu
Original Slides: Alin Dobra. ←

I accept the blame for any errors that I have introduced.
References http://*publications.gbdirect.co.uk/c_book/*.