CS 2SC3 and SE 2S03 Fall 2009

05 Case Study: C Programming Language

William M. Farmer

Department of Computing and Software
McMaster University

18 November 2009

The C Programming Language

- Developed by Dennis Ritchie in 1972 at AT&T Bell Labs.
- Intermediate level language designed for system programming for the Unix operating system.
- A single paradigm programming language: imperative.
- Usually has a single mode of execution: compilation to native machine code.
- Notable characteristics:
  - Weak typing.
  - Low-level access to memory.
  - Extensive use of explicit pointers.
  - Preprocessor for macro definition.
  - Major functionality provided by library routines.
  - Very high execution speed.

Characteristics of C (King)

- C is a low-level language.
  - Access to low-level concepts.
  - Provides control of memory management.
- C is a small language.
  - Lacks high-level design mechanisms like classes and modules.
  - Lacks modern programming mechanisms like exceptions and pattern matching.
- C is a permissive language.
  - Weak typing.
  - Minimal error checking.

Strengths of C (King)

- Efficiency (space and time).
- Portability.
- Power.
- Flexibility.
- Standard library.
- Integration with Unix.

Weaknesses of C (King)

- C programs can be error-prone.
- C programs can be difficult to understand.
- C programs can be difficult to modify.

Outline

1. Form of a program.
2. Variables and references.
3. Basic values.
4. Procedures.
5. Control structures.
6. Pointers.
7. Records and arrays.
8. C Preprocessor
9. Memory management.
10. Evaluation strategies.
11. Higher-order procedures.
12. Recursion.

C: The Structure of a Simple Program

- Here is a simple C program:

```
# include <stdio.h>
int main ()
{
  printf("Hello!\n");
  return 0;
}
```

- `main` is the procedure that starts the execution when the program is invoked.
- `main` takes no input and returns a value of type `int` as output.

C: Native Code Compilation

- A C native-code compiler such as `gcc` compilers C source code files to native code object files and links these object files to produce standalone executables.
- Example: `gcc -o prog prog.c`

C: Variables

- A variable in C is bound to a reference implemented as a memory address.
- Hence, every variable in C has:
  - A name.
  - A type.
  - A location (the address of the reference).
  - A value (the value held by the reference).
- The statement

  ```
  int i;
  ```

  declares a variable with name `i` and type `int`.
- The statement

  ```
  const int two = 2;
  ```

  declares a constant with name `two`, type `int`, and value `2`.

References in C

- References are implemented in C as memory addresses.
- A reference of type t is a memory address of a location that can hold a value of type t.
- In C, a variable of type t is bound to a reference of type t.
- Constructor: `int x;` constructs a reference of type `int` and binds `x` to it.
- Value selector: `x` selects the referenced value (of the reference `x` is bound to).
- Address selector: `&x` selects the address (of the reference `x` is bound to).
- Mutator: `x = 3;` sets the referenced value (of the reference `x` is bound to) to `3`.

C: Numbers

- C contains several primitive numeric types; the most important are:
  - `char`, a type of machine integers representing characters.
  - `int`, a type of medium-size machine integers.
  - `double`, a type of double-precision floating point numbers.
- The sizes of numeric types in C vary across hardwares and compilers.
- Numeric types of C are not strictly typed:
  - The types share a common set of arithmetic operators (`+`, `-`, `*`, `/`, `%`).
  - Numeric values of the wrong type are automatically coerced to the right type.
- Danger: Numeric value coercion may lead to incorrect or unexpected results.

C: Booleans

- C does not have a primitive boolean type.
- The standard C library with header `<stdbool.h>` provides a type `bool` with expressions `true` and `false` denoting the two truth values.
- In C, the proposition operators are:
  - Negation (`!`).
  - Sequential and (`&&`).
  - Sequential or (`||`).
- C has the following set of operators for forming equations and inequalities: `==`, `!=`, `<`, `>`, `<=`, `>=`.

C: Conditionals

- In C, a conditional expression if(A, b, c) is written as

  A `?` *b* `:` *c*

- In C, a conditional statement "If *A* then do *b* else do *c*" is written as

  ```
  if A
    b;
  else
    c;
  ```

- The else part of the conditional statement is optional.

C: Procedures (1/2)

- Procedures in C are called functions.
- The definition of a function has the following form:

$$
\begin{array}{l}
t\ f\ (t_1\ p_1, \ldots, t_n\ p_n) \\
\{ \\
\quad B \\
\}
\end{array}
$$

- t is the type of the value that is returned by a `return` statement in the body B.
- f is the name of the function.
- $t_1\ p_1, \ldots, t_n\ p_n$ is the parameter list of the function. t_i is the type of the parameter p_i.
- B is the body of the function consisting of a list of definitions and statements.
- $t\ f\ (t_1\ p_1, \ldots, t_n\ p_n);$ is the function header or function prototype for the function.

C: Procedures (2/2)

- A function prototype for a function f declares f with its type (which is given indirectly).
- Every function in C must be declared before it can be applied. (Synonyms for "applied" are "called" and "invoked".)
- Unlike OCaml, all executable code in a C program is contained in some function body.
- Unlike OCaml, a C function cannot be defined inside another function.
- Unlike OCaml, the application of a C function is only weakly type checked.
- Like OCaml, functions in C can be defined by recursion (but tail-recursive functions are usually not executed in constant space).

Control Structures

- C has the following syntax for blocks:

  $\{\text{stmt}_1 \; \cdots \; \text{stmt}_n\}$

  Note: An atomic statement ends with a `;`.

- C has the following syntax for while loops:

  `while (`expr`)` stmt

- C has the following syntax for for loops:

  `for (`expr_1`;` expr_2`;` expr_3`)` stmt

  which is equivalent to

  expr_1`;`
  `while (`expr_2`) {`
  stmt
  expr_3`;`
  `}`

Pointers in C

- A pointer in C is a variable of a reference type.
  - Reference types in C are called pointer types.
  - The value of a pointer is a memory address that refers or points to a value.
- Constructor:
  - `int * ip = NULL;`
  - `int * ip = &i;`

  creates a pointer of pointer type `int *`.
- Selector for dereferencing:
  - `*ip`

  denotes the value that the pointer `ip` points to.
- Mutator for dereferencing:
  - `*ip = i;`

  sets the value that the pointer `ip` points to.
- Pointers are used extensively in C.
  - Dangerous and tricky, they must be used very carefully!

Pointer Arithmetic in C

- The `sizeof` operator takes a variable x or type t as input returns an integer that is the number of 8-bit bytes reserved for x or t.
- A pointer p of type t can be viewed as an index into a giant array of cells of size `sizeof`(t):
  - $p + 1$ (pointer addition) is the next index into this giant array.
  - $p - 1$ (pointer subtraction) is the previous index into this giant array.
- Pointer arithmetic is not valid with `void` pointers because values of type `void` do not have a fixed size.
- Pointer arithmetic provides a powerful and uniform mechanize for accessing memory, but it can lead to dangerous and undesired memory access.

Records in C

- Records are called structures in C.
- Structure type declaration:

  ```
  typedef struct {
    [const] t_1 field-name_1;
      ⋮
    [const] t_n field-name_n;
  } struct-type;
  ```

  where `const` is an optional type qualifier that makes the field immutable.
- Constructor:

  struct-type struct-name = {expr$_1$, ..., expr$_n$};
- Selector:

  struct-name.field-name;
- Mutator:

  struct-name.field-name = expr;

Arrays in C.

- Constructor:

  ```
  int a[5];
  int b[5] = {10,20,30,40,50};
  ```

  create arrays of type `int[5]` of length 5.

- Selector:

  `a[`index`]`

  where index is $0, \ldots, 4$.

- Mutator:

  `a[`index`] =` expr`;`.

  where index is $0, \ldots, 4$.

- Note: An array in C is not an ordinary variable: it can not be directly modified by assignment:
  - `a = b;` gives an error.

Pointers and Arrays

- Arrays are implemented in C like constant pointers that point to a fixed amount of space.
- Suppose

```
int a[5];
int * ip;
ip = a;
```

- Then

```
ip == &a[0]
ip + 3 == &a[3]
*(ip + 3) == a[3]
ip[3] == a[3]
```

- Since an array is accessed via a pointer, it is possible to access memory outside of the array (called a buffer overflow).
- Buffer overflows are the cause of many insidious bugs and dangerous security breaches.

Question 1

Consider the following C code:

```
int a[5] = = {2,3,5,7,11};
int * b;
b = a;
```

How are `a` and `b` the same?

A. They are both allocated the same amount of space.
B. They can both be assigned a new value.
C. They both have the same type.
D. They are both bound to the same location.
E. None of the above.

Arrays of Unspecified Length

- An array of unspecified length can be declared if the array is initialized.
  - Example: `int a[] = {0,1,2};`
- An array parameter of a function is usually declared as an array of unspecified length.
  - Example: `int f(int a[]) { ... }`
  - A parameter of this kind is the same as a parameter declared to be a pointer of type `int *`.
- Since an array parameter of a function is treated as a pointer, the length of the argument array is usually passed as a separate parameter to the function.
  - Example: `int f(int a[], int n) { ... }`

Strings in C

- A string in C is a array of elements of type `char` that end with the null character `'\0'`.
- Constructors:

  $"c_1 c_2 \cdots c_n"$

  $\{c_1, c_2, \ldots, c_n, \texttt{'\textbackslash 0'}\}$

- A variable of type `char` array can hold a string.

```
char x[] = "abc";
```

- The C library with header `<string.h>` contains various string processing functions.

The C Preprocessor

- The C preprocessor is a code translator that is applied to a C code file just before the file is compiled.
- The behavior of the preprocessor is controlled by directives that start with the `#` character.
- The most common directives are:
  - `#include` for specifying a code file (usually a header file with the extension `.h`) whose contents are included in the file by the preprocessor.
  - `#define` for defining macros that are expanded in the file by the preprocessor.
  - `#if`, `#endif`, etc. for specifying conditional compilation of the file.
- The preprocessor also removes comments and unnecessary white space in the file.

Header Files in C

- In C, a header file is a code file with the extension `.h` that is intended to be inserted in several other code files.
- Header files are inserted using
  - `#include <filename>`

  for library header files and
  - `#include "filename"`

  for other header files.
- Header files are used to share code between code files and often contain:
  - Macro definitions
  - Type definitions
  - Function prototypes
  - External variable declarations.

Program Structure in C

- A C program consists of a set of header files (with extension `.h`) and source files (with extension `.c`).
- For each source file `file.c`, there should be a header file `file.h` that contains the prototypes of the functions defined in `file.c`.
  - `file.h` should be included in every file that uses the functions defined in `file.c`.
  - It is good form to include `file.h` in `file.c` itself.
  - `file.h` should also contain the definitions of the macros, types, and external variables that are needed to use the functions in `file.c`.
- A module (say, for a stack) is implemented in C as pair of a header file (`stack.h`) and a source file (`stack.c`):
  - The header file serves as the module's interface.
  - The source file serves as the module's implementation.

Question 2

In an object-oriented programming language like C# or Java, what data structure is a kind of module?

A. A class.

B. An object.

C. A hash table.

D. A method.

E. An array.

Question 3

In C, what kind of code file does not need a corresponding header file?

A. One that implements a module.
B. One that contains only functions.
C. One that contains only macros.
D. One that contains the `main` function.
E. One that will never be used by any other code file.

Macros

- A macro is a notational definition used in code that is expanded when the code is interpreted or compiled.
- PL/I, Lisp, C, and C++ are examples of programming languages with macros.
- In source code, a macro can serve as an abbreviation or an alternate syntax form.
  - Macros can make code much easier to read and maintain.
  - A set of macros can define a "little language".
- Macros are expanded using the call by name evaluation strategy.

Macros in C

- In C, a macro definition has a simple form

  `#define` m d

  and a parameterized form

  `#define` m`(`x_1`, ... ,`x_n`)` d

  where $n \geq 1$ and d is a finite sequence of tokens.
- Macros are not typed and can thus lead to type mismatches.
- Simple macros are quite often used to define constants.
- Parameterized macros are more generic and faster than functions and use the call by name evaluation strategy.
- Poorly devised macros (e.g., without sufficient parentheses) may work fine in most but not all situations.
- C macros should always be employed with great care!

Program Memory

A machine code program has four kinds of memory:

1. Processor registers.
2. Static memory.
3. The stack (also called the call stack or execution stack).
4. The heap.

Persistence

- The persistence of an entity (e.g., a variable) is the period of time the entity is available to a running program.
- Examples:
  - The persistence of a running procedure begins when it is called and ends when it returns a value.
  - The persistence of a local variable declared in a procedure normally has the same persistence as the procedure.
  - The persistence of a local variable declared in a procedure can be from when the procedure is first declared to the termination of the program. (These are called local static variables in C.)
  - The persistence of a global variable is from when it is first declared to the termination of the program.

Static Memory Allocation

- Static memory allocation is done at compile time.
  - Performed by the compiler.
  - The size of static memory does not change during run time.
- Static memory includes:
  - Program code.
  - Data that needs to be available for the lifetime of the program.
  - Global constants.
- There is no run time allocation overhead.
- Data structures held in static memory persist for the lifetime of the program.
- Static memory is not deallocated during run time.

Automatic Memory Allocation

- Automatic memory allocation is done at run time whenever a procedure is called.
  - A frame is pushed on the stack when the procedure is called.
  - The frame is popped from the stack when the procedure finishes.
- Stack memory includes:
  - Return address.
  - Local variables.
- Allocation overhead is modest.
- Data structures held in stack memory persist only for the lifetime of the procedure call.
- Stack memory is automatically deallocated when a procedure call finishes and the stack frame is popped.

Dynamic Memory Allocation

- Dynamic memory allocation is done at run time when a data structure is created in the heap.
- The heap memory includes:
  - Dynamic data structures.
  - Data structures that need to persist longer than procedure calls.
- Allocation overhead is high.
- Data structures held in heap memory persist until the memory is deallocated.
- In OCaml, heap memory is implicitly deallocated by garbage collection.
- In C, heap memory is explicitly deallocated.

Heap Memory Allocation and Deallocation in C

- Heap memory is allocated using the operator `malloc`:

```
type * ptr = malloc(sizeof(type));
```

- Heap memory is deallocated using the operator `free`:

```
free(ptr);
```

- Problems:

  1. Unneeded heap memory is not freed by the programmer or the pointer to heap memory is lost (memory leak).
  2. Heap memory is accessed after it is freed.

Evaluation Strategies

- Let p be a procedure with parameters $x_1, \ldots, x_n$ that is applied to arguments $a_1, \ldots, a_n$.
- An evaluation strategy is a set of rules for evaluating $p(a_1, \ldots, a_n)$, an application of p to $a_1, \ldots, a_n$.
- There are several different evaluation strategies.
- A programming language employs one or more evaluation strategies.
- A programming language may also be able to simulate evaluation strategies that is does not directly support.
- The three main types of evaluation strategies are:
  1. Call by value.
  2. Call by reference.
  3. Call by name.

Call by Value

- The most common evaluation strategy is call by value.
- Call by value works as follows on $p(a_1, \ldots, a_n)$:
  1. The arguments $a_1, \ldots, a_n$ are evaluated resulting in values $v_1, \ldots, v_n$.
  2. The values of the parameters $x_1, \ldots, x_n$ of p are set to the values $v_1, \ldots, v_n$.
- Call by value is used by both OCaml and C.
- In OCaml, step 2 is done by binding the parameters to the values.
- In C, step 2 is done by copying the values to new memory locations and then binding the parameters to these memory locations.
  - This is a costly operation if the values occupy a large amount of space.
- In OCaml and C, call by value is relaxed for boolean expressions and conditions.

Call by Reference

- The evaluation strategy call by reference works as follows:
  1. The arguments $a_1, \ldots, a_n$ are evaluated resulting in values $v_1, \ldots, v_n$.
  2. The parameters $x_1, \ldots, x_n$ of p are bound to references for the values $v_1, \ldots, v_n$.
- Call by reference is more space- and time-efficient than C-style call by value, but widens the access to the values.
- Call by reference is not directly supported in OCaml or C.
- In functional programming languages like OCaml, call by reference is used internally — so a parameter bound to a mutable value v behaves as if it were bound to a reference for v.
- Call by reference can be simulated in OCaml by using references and in C by using pointers.

Call by Name

- The evaluation strategy call by name works as follows:
  1. The arguments $a_1, \ldots, a_n$ are not evaluated.
  2. The arguments $a_1, \ldots, a_n$ are directly substituted for the parameters $x_1, \ldots, x_n$ in the body of p.
- Call by name is used to implement:
  - Lazy evaluation (or delayed evaluation).
  - Macro expansion (for example, as in C).

Higher-Order Procedures

- A higher-order function is a function that either takes functions as input or returns functions as output.
- A higher-order procedure is a procedure that represents a higher-order function.
- Higher-order functions are directly represented in OCaml.
- Higher-order functions are represented in C using function pointers, i.e., pointers that point to the address of a function.
- Higher-order procedures are invaluable for building complex procedures from simpler procedures.

Function Pointers in C

- A function name is bound to the starting address in memory of the code that implements the function.
- A function pointer is a variable that holds the address of a function.
- Function pointers are used to indirectly store functions and to pass functions to other functions as input and output values.
- The syntax for declaring a function pointer is:

  $$t \ \texttt{(*fun\_ptr)(}t_1 \ p_1, \ldots, t_n \ p_n\texttt{);}$$

  Note: The parameter names $p_1, \ldots, p_n$ are optional.
- The syntax for applying the function that a function pointer references is:

  $$\texttt{(*fun\_ptr)(}a_1, \ldots, a_n\texttt{)}$$

Polymorphic Procedures

- A procedure is polymorphic if it can be applied to different types.
- In OCaml, polymorphic procedures are defined automatically when input and output types are not fully specified.
  - The execution of polymorphic procedures in OCaml is type safe.
- In C, polymorphic procedures are defined using the `void *` type.
  - The `void *` acts as a universal type.
  - The execution of polymorphic procedures in C is not type safe.
- The use of polymorphic procedures allows code to be more generic, more powerful.

Semantics of Recursive Procedures

A recursive procedure can be understood as:

1. Declarative definition: A definition of a function with an infinite body.
2. Operational definition: A definition of a special-purpose computer.
3. Fixed point definition: An implicit definition of a function f that satisfies an equation of the form $f = H(f)$.

Implementation of Recursive Procedures

- Recursive procedures are usually implemented using the call stack.
  - The stack contains one frame for each call of the recursive procedure.
  - The nesting depth of recursive calls does not need to be calculated before execution.
- If the nesting depth of recursive calls is infinite, the procedure will run until the stack space is exhausted.

Quality Issues

- Termination is shown using a well-founded ordering.
  - For example, a strictly decreasing natural number value.
- Correctness can be proved using induction.
- Efficiency:
  - In some cases, recursion can be highly inefficient in the use of space (e.g., in standard implementations of C).
  - In some cases, recursion can be executed in constant space (e.g., with tail recursive procedures in Scheme or OCaml).

Tail Recursion

- A procedure is tail recursive if nothing is left to do after each recursive call in the procedure body.
- Tail recursive procedures can be made to execute in constant space:
  - In some programming languages, e.g., Scheme and OCaml, the compiler ensures that tail recursive procedures execute in constant space.
  - In other programming languages, tail recursive procedures can be redefined using iteration (which executes in constant space).
- Loops can be replaced with the use of tail recursion.