University of Waterloo
David R. Cheriton
School of Computer Science

Unit 3 – An Introduction to C

We introduce the C programming language. C is a general purpose programming language developed by Dennis Ritchie in 1969-73, for use in the UNIX operating system.

Both C and UNIX are in widespread use today.

C exposes the machine architecture and memory model like few other languages and hence is important in our understanding of computer programming.

Intro to C

- C is often referred to as a “System implementation language”.
- AT&T Bell Labs developed the UNIX operating system in the early 1970s, and developed C as its implementation language.
- UNIX was the first operating system (OS) that made it off of mainframes, to minicomputers, and then to microcomputers.
- C is considered a “low-level” language, especially in its early versions. It allows direct access to the machine, and often reflects the idiosyncrasies of the computers of the 1970s!
- This means you can easily “crash” a program using C, possibly even your computer.

Intro to C

“C gives you enough rope to shoot yourself in the foot.”

A good reference is essential:

- K. N. King: C Programming, A Modern Approach (2nd Edition).

We will only introduce a few of the C language features in class.

You *must* follow up with reading the King text. Start with:

- Chapter 2: “C Fundamentals”
- Chapter 15: “Writing Large Programs”

Racket Variables vs. C Variables

Racket

```
;; Define a variable
(define i 5)
;; Mutate the value of i
(set! i 10)
```

C

```
// Declare a variable
int i;
// Initialize its value
i = 5;
// Assign a new value to i
i = 10;
```

- *Statements* in C end in a semicolon, ";"
- *Comments* in C - two styles
  - After a *//* until the end of the line
  - Between */* and */*
- In C, variables must be declared with a `type`
- Declaration and initialization can also be combined:

  `int i = 5;`

Although seemingly similar, the implementation of variables in Racket and C are very different!

Racket Variables vs. C Variables

In Racket, `(define i 5)` binds 5 to `i`; i.e. `i` *means* 5.

- Variables are always defined with an initial value
- Variable type is determined by its value (dynamically)
- "Variables" may change but seldom[1] do

In C, `int i = 5;` obtains a *memory location* for `i` and writes `int` value 5 into it.

- All variables in C must be declared (once) and given a `type` *before* they are used
- The `type` determines how to interpret the *memory* for `i`
- Declared variables should be initialized (but don't have to be)

[1] Mutation is supposed to be the exception rather than the rule.

Mutation in C

Mutation is so ubiquitous in C, nobody thinks about it.

- Racket programmer to C programmer:
  "How do you do mutation in C?"
- C programmer: <blank stare>

In C, mutation is accomplished with an `assignment` statement.

- A single "=" is the `assignment` operator
- "`i = 10;`" is similar to "`(set! i 10)`"
- Overwrites an `int` value in the memory location assigned `i`
- The "label" `i` always references the same *memory* location, but the value located there may change

Careful! "==" is the *equality operator* (more on that later).

Memory

- C runs on machines with Random Access Memory (RAM), which is modelled as a sequence of numbered *bytes*, where each byte holds 8 *bi*nary digi*ts* (or *bits*).
- The bytes in RAM are numbered (0, 1, 2 ...) and the "number" of each byte is the *address* of the byte.

address	contents
4	10010010 11101000 00011010 11110000
8	00001101 00000001 11011111 01001010
12	11110100 10101110 00010100 00000000
...	...
256	
260	
264	
...	

In this course, the size of each `int` is 32 bits or 4 bytes.

address	4	5	6	7
	10010010	11101000	00011010	11110000

Memory Locations

Previously, we said that in C, "`int i = 5;`"

- Obtains a memory location for `i`
- Writes `int` value 5 into it

In addition, C will:

- Remember the `type` of `i` (`int`)
- Obtain exactly 4 bytes for `i`
- Keep track of the memory location (the *address*) of where those 4 bytes are located in RAM

Every variable has an address.

Giving meaning to bits

- Just knowing the address of a variable is not enough to interpret it: you also have to know the `type`
- For example, 4 bytes (32 bits) stored in memory can be interpreted in many different ways:
  - An integer
  - A machine instruction
  - The address (memory location) of another variable
  - The colour value of a pixel
  - 4 characters (each one requiring 1 byte)
  - ...
- The *type* of a variable tells us how many bytes of information are stored at that address, *and* how to interpret those bytes

`int`

The type `int` is an "integer type".

- An `int` can be positive or negative
- Can represent integers from -2147483648 to 2147483647 or -2^{31} to $2^{31}-1$
- Arithmetic is performed modulo 2^{32} ➡ *Don't go out of bounds!*

Suppose `int a = -2147483648; int b = 2147483647;`

- What is `a-1`? `2147483647` ➡ *OVERFLOW!*
- What is `b+1`? `-2147483648`
- What is `b*2`? `-2`

The type `int` in C is

- Not as general as integers in Racket (no arbitrary precision)
- But all C arithmetic operations `+ - * /` execute in a constant number of operations: ➡ Very Fast!

More on `int`

Suppose we have declared

```
int a = 13;
int b = 5;
```

- `a / b` is integer division of `a` by `b` (i.e. rounds down)
  - `13 / 5` ➡ `2`
- `a % b` is remainder of integer division of `a` by `b` (pronounced “`a` mod `b`”)
  - `13 % 5` ➡ `3`

Integer comparison operations and expressions

If we have "`int a; int b;`", C provides the following operations:

- `a < b`, `a > b`
- `a <= b`, `a >= b`
- `a == b`, `a != b`

Operations are written "infix" (`a < b`) in C, as opposed to prefix (`(< a b)`) in Racket.

N.B. "`equal?`" is written as "==" (and **definitely not** "=")

Boolean operators

- "and" ➡ &&
- "or" ➡ ||
- "not" ➡ !

Boolean Expressions

Expressions are “natural” combinations of comparison operations, arithmetic expressions, and boolean operations such as

- `(i>3) && (i != 5)` $\Leftrightarrow$ `(and (> i 3) (not (= i 5)))`
- `(i/4) > (i%4)` $\Longleftrightarrow$ `(> (quotient i 4) (remainder i 4))`
- ...
- C has complicated *precedence rules* for which operations are done first (e.g., `*`, `/` before `+`, `-`, etc.)
  - Check King, Chapter 4.1, for some common ones
  - Better to just use parentheses (...) to control evaluation order

- In C, boolean values are represented by `int`s:
  - *false* is `0`
  - *true* is “not false” ➡ `non-zero`

Function definitions

Racket

```
(define (timestwo x)
  (* x 2))
```

C

```
int timestwo(int x) {
  return x * 2;
}
```

Function definitions in C,

- Start with a *header* describing
  - `type` of value the function will `return` (i.e. *produce* in Racket) (`void` is used if no value is to be returned)
  - Name of the function
  - List of all parameter names and types (also may be `void` if there are no parameters)
- Followed by the *body*
  - Code statements within `{...}`

In C, if your function returns a value, you must explicitly tell it to!

Return statement

In C, a `return` statement is used to explicitly return (produce) a value.

- Statements end in "`;`" and *do not have values* as in Racket
- `return` $\langle val \rangle$ stops evaluation of function and returns $\langle val \rangle$
- `type` of returned value must match header

- If we "reach the end" of a function without doing a `return`, your program may have unexpected results (unless the function returns `void`)
- RunC will help you identify such errors with a warning or error message

Compound statement

We can create a *compound* statement by putting a series of statements in `{...}`.

- Takes many statements and combines them into one
- Much like `(begin ...)` in Racket
- We can also create local variables for use within that `{...}`
- Statements are executed in sequence

Racket

```
(define (timestwo x)
  (define factor 2)
  (* x factor))
```

C

```
int timestwo(int x) {
  int factor = 2;
  return x * factor;
}
```

Global and local variables

Racket

```
(define g 10)
(define (foo x)
  (define j (* x 5))
  (* x j g))
```

C

```
int g = 10;

int foo(int x) {
  int j = x * 5;
  return x * j * g;
}
```

Variable scope is similar in Racket and C.

- `g` has a *global* scope: it is accessible anywhere within the module
- `j` is *local* in scope: it is only accessible within the body of `foo`

In imperative programming, global variables should be used sparingly (unless storing constant values).

Conditional Statements

The `if` statement in C is similar to `cond` expression in Racket, except that it does not produce a value unless `return` is used.

- ```
  if ( boolean expression ) {
      statements
  }
  ```

- ```
  if ( boolean expression ) {
      statements
  } else {
      statements
  }
  ```

- ```
  if ( boolean expression ) {
      statements
  } else if ( boolean expression ) {
      statements
  } else {
      statements
  }
  ```
  ```

Conditional Statements: Examples

- ```
  if ( answer == 42 ) {
     printf("What is the question?");
  }
  ```

- ```
  if ( i % 2 == 0 ) {
     printf("i is even\n");
  } else {
     printf("i is odd\n");
  }
  ```

- ```
  if (!((i > j) && (j > k)) || (m == 42)) {
   //some code
  } else if (i == j) {
   //some code
  } else {
   //some code, possibly print an error message
  }
  ```
  ```

Repetition: Recursion

```
(define (helper n acc)
  (cond
    [(= n 0) acc]
    [else (helper (sub1 n) (+ n acc))]))

(define (sum-first n)
  (helper n 0))
```

This is guaranteed to be efficient in Racket, but not in C.

```
int helper(int n, int acc) {
  if (n == 0) {
    return acc;
  } else {
    return helper(n-1, acc+n);
  }
}

int sumfirst(int n) {
  return helper(n, 0);
}
```

Trace: Recursion

Recursion

- `sumfirst(3)` calls `helper(3, 0)`
- which calls `helper(2, 3)`
- which calls `helper(1, 5)`
- which calls `helper(0, 6)`
- `helper(0, 6)` then `return`s 6 to `helper(1, 5)`
- `helper(1, 5)` then `return`s 6 to `helper(2, 3)`
- `helper(2, 3)` then `return`s 6 to `helper(3, 0)`
- `helper(3, 0)` then `return`s 6 to `sumfirst(3)`
- `sumfirst(3)` then `return`s 6 to where it was called from

Repetition: Iteration

- Iteration is the more standard idiom in C
- We can still use recursion, but tend to not as much

```c
int helper(int n, int acc) {
  while(n > 0) {
    acc = acc + n;
    n = n - 1;
  }
  return acc;
}

int sumfirst(int n) {
  return helper(n, 0);
}
```

Trace: Iteration

Iteration

- `sumfirst(3)` calls `helper(3, 0)`
- Initially `n: 3` and `acc: 0`
- After each iteration of the `while` loop:
  - Iteration 1, `n: 2` and `acc: 3`
  - Iteration 2, `n: 1` and `acc: 5`
  - Iteration 3, `n: 0` and `acc: 6`
  - `while` condition is false
- `helper` then `return`s 6 to `sumfirst`
- `sumfirst` then `return`s 6 to where it was called from

Iteration: No Wrapper

Since we are using a loop instead of accumulative recursion, we no longer need a wrapper function:

```
int sumfirst(int n) {
  int acc = 0;
  int i = n;
  while(i > 0) {
    acc = acc + i;
    i = i - 1;
  }
  return acc;
}
```

This method of iteration is so common in C that there is a more compact `for`m...

For loops

C also has `for` loops, which provide a more compact form for some common iterations.

```
for(<expr1>;<expr2>;<expr3>) {
  //statements
}
```

```
for(i=n; i>0; i=i-1) {
  acc = acc + i;
}
```

Can be rewritten as:

```
<expr1>
while(<expr2>) {
  //statements
  <expr3>;
}
```

```
i = n;
while(i > 0) {
  acc = acc + i;
  i = i - 1;
}
```

Generally have specific roles for *<expr1>*, *<expr2>*, *<expr3>*:

- *<expr1>* is the *initializer*: set up some variables
- *<expr2>* is the *condition*: test to continue
- *<expr3>* is the *updater*: update for next iteration

Common `for` loops

- Counting up from `0` to `n-1`:

```
for (i = 0; i < n; i = i + 1) {...}
```

- Counting up from `1` to `n`:

```
for (i = 1; i <= n; i = i + 1) {...}
```

- Counting down from `n-1` to `0`:

```
for (i = n-1; i >= 0; i = i - 1) {...}
```

- Counting down from `n` to `1`:

```
for (i = n; i > 0; i = i - 1) {...}
```

Code organization in C

C has primitive version of Racket's modules, based solely on source files.

- *Interface/specification* goes in a **header** file: `foo.h`

```
int foo(int i);
// PRE: True
// POST: Returns i + 1
```

- *Implementation* code goes in a different file: `foo.c`

```
int foo (int i) {
  return i + 1;
}
```

- Somebody (the client) wants to use foo in their file?

```
#include "foo.h"
// ...
int eight = foo(7);
```

Header “.h” files

The *Header* file only contains information you want to share with the client.

- *Prototype* statements to declare the functions that the client may use
  - For example: `int foo(int i);`
  - Declares that the function `foo` exists (its definition will be in the implementation file) and shows its *signature*
    - Specifies return type, function name, parameter types
  - Like a contract but part of the C language (not a comment)
- Additional interface/specification information in comments
  - Preconditions, postconditions, side-effects, purpose, examples, etc.

C Code ".c" files

The *Implementation* file has all of the code for the module, and may be hidden from the client.

- Interface information goes in an ".h" file; e.g. `foo.h`
- Implementation code goes in a ".c" file; e.g. `foo.c`
- Both files should have the same name, but different extension

To make the code in `foo.c` available to a client, the client should

```
#include "foo.h"
```

- *Directives* in C start with `#`, and are "special instructions"
- `#include` is similar to `require` in Racket
- use `#include "foo.h"` for header files in the same directory
- use `#include <stdio.h>` for header files in a "standard" location (typically, headers provided by C)

Separation of Concerns

Interface, implementation and usage are all separated:

foo.h

```c
int foo(int i);
```

foo.c

```c
int foo(int i) {
  return i + 1;
}
```

client.c

```c
#include "foo.h"
// ...
int eight = foo(7);
```

Function Signatures

- Whenever a function call is encountered, C must already "know" the *signature* (contract) of the function
- C must have "seen" either the declaration or the definition for the function

- When C encounters the special `#include` directive, it "pastes" the contents of the ".h" file into the current ".c" file
- Because the function declarations are in the ".h" file, C then "knows" the signature for those functions

- **Every** function in C must be declared (or defined) before it can be called.
- When two functions call each other (i.e., mutual recursion), you need to declare a function first, and then define it later.

Special function main

In order to execute, C must know where to start.

- One of your ".c" files must define a function `main`:

  ```
  int main(void) {
    /*Statements happen when your program
      is executed.*/
    return 0;
  }
  ```

- `main` returns an `int`
- The statement "`return 0;`" tells the operating system (which called `main`) that everything worked fine
- When you write a C module, you must write another program with a `main` function in it for testing
- In RunC, have the file with `main` open in gEdit when you run

C versus Racket - Definition of function timestwo

In C: header file `timestwo.h`

```
int timestwo(int x);
//PRE: True
//POST: returns 2 * x
```

In C: source file `timestwo.c`

```
#include "timestwo.h"

int timestwo(int x) {
  return x * 2;
}
```

In Racket: file `timestwo.rkt`

```
#lang racket
(provide timestwo)
```

```
;;timestwo:  Int -> Int
;;    PRE: True
;;    POST: produces 2 * x
(define (timestwo x)
  (* x 2)
)
```

C versus Racket: Program that uses timestwo

Somebody (the client) wants to use `timestwo`? In a file `myprog.c`...

```c
// file: myprog.c

#include <stdio.h>
#include "timestwo.h"

int main(void) {
  printf("2 * 7 is %d\n",
         timestwo(7));
  return 0;
}
```

Somebody (the client) wants to use `timestwo`? In a file `myprog.rkt`...

```racket
#lang racket
;; file:  myprog.rkt

(require "timestwo.rkt")

(printf "2 * 7 is ~a\n"
   (timestwo 7))
```

Types in C: char

The type `char` is a “character type”.

- `char`
  - Generally uses 1 byte to store a single character
  - Stored in ASCII ➡ an integer representation
    - Range: from 0 to 255
    - Some characters are platform specific
  - Single characters can be used in single quotes to represent their integer value
  - `char c = 'a';` is equivalent to `char c = 97;`
  - Arithmetic and comparison operations are similar to `int`s
    `printf("%c is the %dth letter\n", c, c-'a'+1);`
  - *Be careful!* `'a' != "a"`

Booleans

There is no actual boolean type in C – only `int`s

- Comparison operators (like >, <, ...) and logical operators (like `&&` and `||`) return "`0`" for false and non-zero for true
- C99 has a type `_Bool`, which is really just another name for `int`, but with values restricted to 1 and 0
- You can `#include <stdbool.h>` which defines the nicer name `bool` for `_Bool`, and defines constants like `true` and `false`

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

int main(void) {
  bool getout = false;
  int i=0;
  while (!getout) {
    printf("i=%d\n",i);
    i = i + 1;
    if (i>50) {
      getout = true;
    }
  }
  return 0;
}
```

`printf`

The `printf` function is the most common output function in C.

- Part of a built-in module, so we must: `#include <stdio.h>`
- Similar to Racket, but different syntax - we don't use `~a`
- *Side-effect:* Prints to "standard output"
- `"%d"` is for printing an `int` (in **d**ecimal, base 10)
- `"%c"` is for printing a **c**haracter

  ```
  printf("The ASCII value of %c is %d\n", c, c);
  ```
- `"%p"` is for printing a memory address (more later)
- Other format strings for different types; see King, Chapter 3

`scanf`

The `scanf` function is the most common input function in C.

- We must make it available: `#include <stdio.h>`
- For now, one form only:
  `scanf("%d",&i);`
- Returns a “status value”, but ignore that for now
- *Side-effect:* Reads an integer from “standard input”, puts the value into variable `i`
- Note “magic” syntax “`&i`”
  - We will talk about this later, but be sure to put it in exactly as written!

Interactive driver program

```c
// file: myprogram.c

#include <stdio.h>

int main(void) {
  int num1, num2;
  printf("Enter two integers and then press enter: ");
  scanf("%d",&num1);
  scanf("%d",&num2);
  printf("You entered %d and %d\n",num1,num2);
  return 0;
}
```

Testing in RunC

- Test files:
  - myprogram.in.1, myprogram.in.2, ...
  - myprogram.expect.1, myprogram.expect.2, ...
- RunC runs your program once for each .in. file as if you typed the contents in at the prompt
- compares against the corresponding .expect. file, alerts you to differences

A Guessing Game in C

```
// file: newgame.h
// Provides:  startgame, guess

void startgame(int n);
// PRE: n >= 1
// POST: The game is initialized
// startgame(n) starts a new game ...

char guess(int k);
// PRE: k >= 1
//      The game has been initialized
// POST: returns one of 'r' or 'h' or 'l' for
//       'r'ight, too 'h'igh, or too 'l'ow
```

Guessing Game in C – Implementation

```
// newgame.c
// ==========================
#include <stdlib.h>
#include "newgame.h"

int secret;   // secret number

void startgame(int n) {
  secret = (rand() % n) + 1;
}

char guess(int g) {
  if (g == secret) {
    return 'r';
  } else if (g < secret) {
    return 'l';
  } else {
    return 'h';
  }
}
```

Guessing Game: Naïve Player

```c
// File playnewgame.c
// ==================

#include "newgame.h"
#include <stdio.h>

int main(void) {
  int count = 0;
  char result = 'x';
  startgame(10); // Play game of size 10

  while(result != 'r') { // While wrong
    count = count + 1;
    result = guess(count);
  }
  printf("I won after %d guesses!\n", count);
  return 0;
}
```

Guessing Game: Cheating Player

```c
// File playnewgame.c
// ==================

#include "newgame.h"
#include <stdio.h>

extern int secret; // cheating!

int main(void) {
  char result = 'x';

  startgame(10);
  result = guess(secret);

  if (result == 'r') {
    printf("I won after 1 guess!\n");
  } else {
    printf("Inconceivable!\n");
  }
  return 0;
}
```

Revisiting the Simple Passport Office

```
// passport.h module

int next_ticket(void);
// PRE: true
// POST: increments & returns the ticket #

int next_serve(void);
// PRE: true
// POST: increments & returns the service #
```

```
//passport.c implementation
#include "passport.h"

int ticket = 0;
int serving = 0;

int next_ticket(void) {
  ticket = ticket + 1;
  return ticket;
}
int next_serve(void) {
  serving = serving + 1;
  return serving;
}
```

Jumping the Line

```c
// cheater.c
#include <stdio.h>
#include "passport.h"

extern int serving; // cheating!

int main(void) {
  int myticket;

  while (next_ticket() < 3684) {}  // simulate long line

  myticket = next_ticket();
  printf("my ticket is: %d\n", myticket);
  printf("now serving: %d\n", next_serve());
  serving = myticket - 1;  // cheating!
  printf("now serving: %d\n", next_serve());
}
```

```
my ticket is: 3685
now serving: 1
now serving: 3685
```

Variable and function hiding with static

The extern keyword makes global variables in other ".c" files (modules) available in the current file.

To ensure that a global variable is only available within the current file (module), the static keyword must be used:

```
//passport.c implementation
//...
static int ticket = 0;   // now hidden
static int serving = 0;
```

The static keyword can also be used with functions:

```
int visible_function(int x) {
  ...
}

static int hidden_function(int x) {
  ...
}
```

Structures: Racket vs. C

Racket

```
;; define a structure
(define-struct posn (x y))
```

```
;; declare a posn
(define p (make-posn 3 4))
```

```
;; accessing fields
(define j (posn-x p))
```

```
;; with #:mutable
(set-posn-y! p 10)
```

C

```
// define a structure
struct posn {
  int x;
  int y;
}; // <--- note the ; here!
```

```
// declare a posn
struct posn p = {3,4};
struct posn q; // uninit.
```

```
// accessing fields
int j = p.x;
```

```
// mutation
p.y = 10;
```

Structures in C

- the type of the structure is "`struct structname`"
- the type of each field must be declared
- there is no C equivalent of `posn?`, because the type of the variable is always known
- The == (equal) operator does not work with structures (you must write your own)
- functions can return structures, and structures can be arguments:

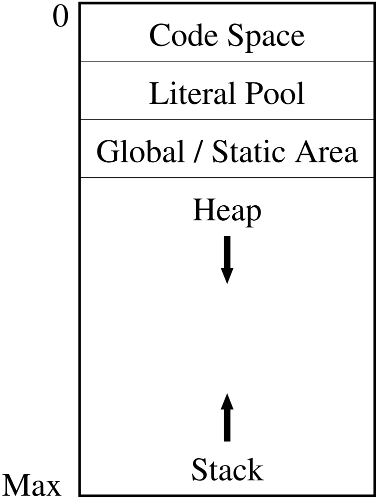
```
struct posn scale(struct posn p, int factor) {
  struct posn newposn;
  newposn.x = p.x * factor;
  newposn.y = p.y * factor;
  return newposn;
}
```

(later, we will see a more common way to use structures with functions)

C Memory Partitioning

Previously, a simplified view of memory was introduced. In practice, memory is *partitioned* into different *regions*:

- Global/Static (global and static variables)
  - These persist for the life of your program
  - Static variables are only visible within module or function
  - Uninitialized variables are initialized to 0
- Stack (function parameters and local variables)
  - Temporary storage for variables
  - Uninitialized variables have unknown values
- Heap
  - Available storage that you can use
- Literal Pool
  - Will be discussed later
- Code Space
  - Where your program resides


0
Code Space
Literal Pool
Global / Static Area
Heap
Stack
Max

Implementing function calls in C

When a function is called, C "automatically" allocates memory to store:

- parameters
- local variables
- the location (address) from which the function was called (and hence where to resume upon return)

This is naturally implemented as a stack.

- A *stack frame* stores the information for a function on the stack
- Stack frame on top belongs to the function currently executing
- When a function completes its execution, its stack frame is popped and control is returned to the location where the function was initially called (the function that is now on top)

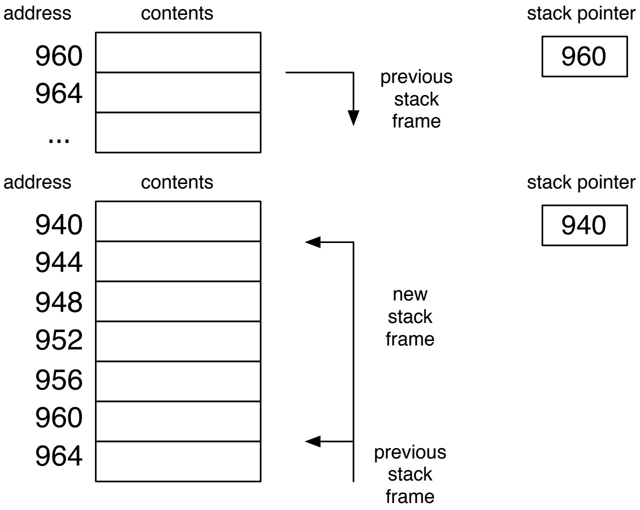
Managing the stack

- To manage the stack, C maintains a single value, the *stack pointer*, which is an address of the next available position on the stack.
- Sometimes the stack grows toward higher-numbered memory addresses and sometimes toward lower-numbered ones. The choice depends on the compiler and architecture.
- In this course, stacks grow toward lower-numbered memory locations (as this is the case with most modern CPUs).

Managing the stack (2)

Suppose the *stack pointer* currently has value 960 and a *stack frame* holding 4 `ints` and the return address is to be pushed onto the stack.

- The stack frame requires:
  - 4 `ints` ➡ 4 * 4 bytes = 16 bytes
  - return address ➡ 32-bit memory address = 4 bytes
  - Total: 20 bytes
- Subtract 20 from the stack pointer to make it 940
- Stack frame will use locations 944, 948, 952, 956, and 960
- If another function is called from this function, a new stack frame will be pushed on the stack (starting at 940)
- When the current function returns, the stack frame is popped by setting the stack pointer back to 960 (from 940)


address
contents
960
964
...
previous stack frame
stack pointer
960
address
contents
940
944
948
952
956
960
964
new stack frame
previous stack frame
stack pointer
940

- Each function call requires a stack frame to allocate space for parameters and local variables declared in the function.
- Rather than providing detailed memory diagrams, draw a box to represent a stack frame, with its information inside.
- The following program has been executed:
  - The stack frame for `main` is first on the stack
  - `i` is initialized to 3
  - `main` is about to call `sum(i)`

```
int sum(int n) {
    int r = 0;
    if (n != 0) {
        r = n + sum(n - 1);
    }
    return r;
}
int main(void) {
    int i = 3;
    printf("%d\n", sum(i));
    ...
```

main
i : 3
return addr: to OS

When `sum(i)` is encountered, the argument value of `i` is 3.

- A new stack frame is allocated
- The argument is copied into the parameter: 3 is copied into `n`
- The body of `sum` is executed

sum n : 3 r: 0 return addr: to main l.2
main i : 3 return addr: to OS

In the body of `sum`, the expression `sum(n - 1)` is encountered.

- The value of `n` is 3, so the argument to `sum` evaluates to 2
- A new stack frame is allocated for `sum(2)`, value 2 is copied into `n` and execution of `sum(2)` commences.

We speed up the movie.

sum n : 3 r : 0 return addr: to main l.2
main i : 3 return addr: to OS

sum n : 2 r: 0 return addr: to sum l.3
sum n : 3 r: 0 return addr: to main l.2
main i : 3 return addr: to OS

sum n : 1 r: 0 return addr: to sum l.3
sum n : 2 r: 0 return addr: to sum l.3
sum n : 3 r: 0 return addr: to main l.2
main i : 3 return addr: to OS

sum n : 0 r: 0 return addr: to sum l.3
sum n : 1 r: 0 return addr: to sum l.3
sum n : 2 r: 0 return addr: to sum l.3
sum n : 3 r: 0 return addr: to main l.2
main i : 3 return addr: to OS

- When `n` is 0, the `if` condition is false, and `return r` is executed.
- The value of `r` becomes the value of the `sum` function in the computation being resumed.

sum
n : 0
r: 0
return addr: to sum l.3

sum
n : 1
r: 0
return addr: to sum l.3

sum
n : 2
r: 0
return addr: to sum l.3

sum
n : 3
r: 0
return addr: to main l.2

main
i : 3
return addr: to OS

sum
n : 1
r: 1
return addr: to sum l.3

sum
n : 2
r: 0
return addr: to sum l.3

sum
n : 3
r: 0
return addr: to main l.2

main
i : 3
return addr: to OS

sum
n : 2
r: 3
return addr: to sum l.3

sum
n : 3
r: 0
return addr: to main l.2

main
i : 3
return addr: to OS

sum
n : 3
r: 6
return addr: to main l.2
main
i : 3
return addr: to OS

main
i : 3
return addr: to OS

- The value 6 was returned for `sum(3)` to `main`.
- That value is printed, and `main` returns to the OS.
- Tedious though this whole process was, it is essential to understanding the behaviour of the recursive C function `sum`.

- Modelling recursion in CS135 Racket (Intermediate Student with Lambda) was easier, because of the lack of mutation and the ability to use the substitution model
- Modelling recursion in C is harder because we are required to get closer to what actually happens on the machine
  - We need to understand the run-time stack

- Without recursion, one does not need to talk about stacks.
  - We could have a separate area of memory for each function
  - Early imperative languages (e.g. FORTRAN) did not support recursion
  - Those wishing to use it had to build stacks themselves

`static` local variables

Earlier, we learned how the `static` keyword is used to identify variables and functions that are *local* to the module (file).

(Unfortunately) C also uses the `static` keyword to identify **local** variables (within a function) that are *persistent*.

Within a function, a variable declared as `static` is stored in the global/static region, and *not* on the stack.

Just like regular local variables, `static` local variables are only visible within the function. The significant differences are that they are only initialized once (at the start of the program) and that they keep their values between function calls (persistence).

Example: `static` local variables

```
#include <stdio.h>

int foo(int x) {
   static int y=0;
   y=y+x;
   return y;
}

int main(void) {
   printf("%d\n", foo(1));
   printf("%d\n", foo(2));
   printf("%d\n", foo(3));
   printf("%d\n", foo(100));
   return 0;
}
```

```
1
3
6
106
```

Allocating memory

- Every variable in C is at some location in RAM with its own address
- The statement `int i = 0;` does **2** things:
  (1) Reserves four consecutive bytes of memory
  (in global/static area or on the stack)
  (2) Sets the value of the bytes at that location to the binary representation of zero, i.e.
  `00000000 00000000 00000000 00000000`
- In C, we can ask *where* a variable is stored using: the *address of* operator, `&`
- `&i` is the address of the variable `i`

Addresses

```c
#include <stdio.h>

int i = 10;
int j = 20;

int main(void){
  int m = 30;
  int n = 40;
  static int s = 50;

  printf("The address of i is: %p\n",&i);
  printf("The address of j is: %p\n",&j);
  printf("The address of m is: %p\n",&m);
  printf("The address of n is: %p\n",&n);
  printf("The address of s is: %p\n",&s);

  return 0;
}
```

Example output: (`%p` displays in *hexadecimal* format)

```
The address of i is: 0x8062080
The address of j is: 0x80620c0
The address of m is: 0xbf9c7c60
The address of n is: 0xbf9c7ca0
The address of s is: 0x8062100
```

Pointers

- In C, we can store an *address* in a variable.
- The variable that stores an address is called a *pointer*.

```
...
int i = 10;
int *ptr = &i;
printf("The address of i is: %p\n",&i);
printf("The value of ptr is: %p\n",ptr);
...
```

```
The address of i is: 0x804a014
The value of ptr is: 0x804a014
```

- To declare a pointer, use * before the variable name.
- See King, Chapter 11.

Dereferencing

```
int i = 10;
int *ptr = &i;
```

- Pointer `ptr` "points at" the address of `i`
- The pointer type (`int *ptr`) and the type it points at (`int i`) must match
- We can access the **contents** of the location that a pointer "points at" by using *dereferencing*
- `int j = *ptr;`

  `*ptr` means "The value of the `int` located at the address stored in `ptr`."
- A `*` in front of a variable *declaration* means "this is a pointer", while a `*` in front of a variable *in an expression* means "dereference this pointer"
  - A `*` in-between two expressions still means *multiplication*, so be careful!

```c
#include <stdio.h>

int i = 10;

int main(void){

  int *ptr = &i;

  printf("The address of i is: %p\n", &i);
  printf("The value of i is: %d\n\n",  i);

  printf("The address of ptr is: %p\n",               &ptr);
  printf("The value of ptr is: %p\n",                  ptr);
  printf("The value of what ptr points at is: %d\n", *ptr);

  return 0;
}
```

```
The address of i is: 0x8062080
The value of i is: 10

The address of ptr is: 0xbfe73140
The value of ptr is: 0x8062080
The value of what ptr points at is: 10
```

Mutation and Aliasing

- Mutation is allowed.

```
int i = 10;
int *ptr = &i;

printf("The value of i is: %d\n",i);
*ptr = 55;
printf("The value of i is: %d\n",i);
```

```
The value of i is: 10
The value of i is: 55
```

- The ability to access the same location in memory through two or more different variables is called *aliasing*

Passing Values to Functions

In Racket, before a function is evaluated, each of the arguments must be a *value*.

The same rule applies in C: each of the arguments passed to a function must be a value.

This convention is known as “pass by value”.

We have now seen how when a function is called, **copies** of the argument **values** are pushed on to the stack.

If a function changes the value of one of the parameters, it modifies the *copy on the stack*, and cannot change the value of the original variable passed in.

```c
#include <stdio.h>

void foo(int j) {
  j = j + 1;
}

int main(void) {
  int i = 10;
  printf("i before: %d\n",i);
  foo(i);
  printf("i after: %d\n",i);
  return 0;
}
```

```
i before: 10
i after: 10
```

What if we *want* a function to change the value of a variable?

We can pass in a **pointer** to the variable we wish to change.

```c
#include <stdio.h>

void foo(int *j) {    // <-- now accepts a pointer
  *j = *j + 1;        // <-- dereferences the pointer
}

int main(void) {
  int i = 10;
  int ptr = &i;
  printf("i before: %d\n",i);
  foo(ptr);  // <-- passing ptr by value
  printf("i after: %d\n",i);
  return 0;
}
```

```
i before: 10
i after: 11
```

This still follows the "pass by value" convention. The **value** of `ptr` was passed to the function `foo` and we *did not change* `ptr`. However, the *contents* of what `ptr` points to (`i`) were changed.

In practice, a separate pointer variable is not required: the `&` operator is used:

```
#include <stdio.h>

void swap(int *i, int *j) {
  int tmp = *i;
  *i = *j;
  *j = tmp;
}

int main(void) {
  int i = 10;
  int j = 20;
  printf("i,j before: %d,%d\n", i, j);
  swap(&i, &j); // <--- note use of &
  printf("i,j after: %d,%d\n", i, j);
  return 0;
}
```

```
i,j before: 10,20
i,j after: 20,10
```

Back to `scanf`

Now we can see why we pass the *address* of the variable to `scanf`

```
#include <stdio.h>
// Reads and prints integers from standard input
// until there are no more
int main(void){
  int num;
  while (scanf("%d", &num) == 1) {
    printf("%d\n", num);
  }
}
```

- `scanf` actually returns an integer: the number of items read
- If you try to read 1 item and `scanf` returns 0 or `EOF` (a special value), there is no more data to be read
- In the RunC interactions window, you can indicate the end of input by typing Control-D.

Uninitialized Pointers

With pointers, it is now *much* easier to "crash" your program.

```
int main(void) {
  int *p;        // uninitialized pointer: p = ??
  *p = 5;        // could crash your program
  ...
```

In our RunC environment, there are special checks in place to try and catch pointer misuse, but that's not always possible.

If a pointer hasn't been initialized, or is otherwise invalid, it is very good practice to set the value pointer to `NULL`: `int *p = NULL;`

`NULL` is defined in `<stdlib.h>`, and is a *sentinel value* guaranteed to be an address that nothing could point to. In all modern C environments, `NULL` is zero.

It's often very good practice (especially in "real world" systems) to check to make sure a pointer is not NULL before using it.

```
  if (ptr != NULL) {...}
```

Functions that Return Pointers

You can have a function that `return`s a pointer type:

```
int *function_name(...) {...}
```

However, you must **never** return a pointer to data that was allocated on the stack, as that memory becomes invalid as soon as the function `return`s.

```
int *very_bad(int i) {
  return &i;  // NEVER do this!
}

int *also_very_bad(int i) {
  int j = 10;
  return &j;  // NEVER do this!
}
```

Pointers to Structures

When the parameters of a function are large structures, it can be inefficient to copy the entire contents of the structure on to the stack every time the function is called.

In practice, it's more common to use a *pointer* to a structure.

Passing a pointer to a structure is also common when a function wants to change (mutate) the fields within a structure.

```
void posn_add_x(struct posn *p, int i) {
  (*p).x = (*p).x + i;
}
```

The `(*p).x` notation is awkward, and occurs often enough that C has a special operator `p->x`, where `p->x` is equivalent to `(*p).x`

```
void posn_add_x(struct posn *p, int i) {
  p->x = p->x + i;
}
```

Structure Pointers in Racket

So far we haven't discussed pointers in Racket.

Most higher-level languages like Racket hide pointers from the programmer, avoiding many of the pointer pitfalls that we will see.

Structures in "full" Racket are actually manipulated through pointers:

```
(define-struct posn (x y) #:mutable)

(define p1 (make-posn 1 2))
(define p2 p1)  ;; pointer assignment!  p2 points to p1

(posn-x p2)
(set-posn-x! p1 10)
(posn-x p2)
```

```
1
10
```

Structure Parameters in Racket

Like C, all Racket function parameters are consumed as *values*, but with structures the *value of the pointer* is consumed by the function.

Because of this, we are able to mutate the fields of a structure parameter inside a Racket function:

```
(define-struct posn (x y) #:mutable)

;; posn-add-x: Posn! Int -> Void
(define (posn-add-x p i)
  (set-posn-x! p (+ (posn-x p) i)))
```

(we have already seen this, but now it makes more sense)

Stepping Back: our Approach to Learning C

- So far we have focused on a few introductory topics in C
- In future units, we will tackle more advanced C topics such as dynamic memory, arrays and strings
- For the remainder of this unit, we will discuss some more intermediate uses of C including some techniques that are common in the “real world”
- There are many topics in C that we will not use in this course and not talk about much (or not at all), including: the `switch` statement, `#define` macros, `unions`, `enums` and many more...
- Even for the topics we did cover, we skipped over some of the detail
- Ultimately, if you wish to become proficient at C, you should read the textbook in detail

Named Constants

Programs often have programmer-defined constants in them.

- Literals (like 10, 'a', etc.) can appear anywhere a value of the appropriate type is legal
- But literals might need to change, and can be scattered (and repeated) throughout code

So far in C, all declared variables can be changed.

- Might declare `int maxval=100;` but could then say `maxval=20;`
- Use a `const` modifier when declaring a variable: variable must be initialized, but can't be modified thereafter: it's *constant*
- For example:

```
const int max_nodes = 17;
const char first_letter = 'A';
```

- Constants improve code maintainability, clarity and possibly improve performance

Use of Named Constants

Constants may be used within any scope (i.e., global, static global, or local)

```
const int important_global = 42;         // A
static const int internal_const = 43;    // B

int foo(int x) {
  const int adder = 3;                   // C
  return x + adder;
}
```

(A) Available to this module, and other modules via `extern`

- All global variables should be constants (though lots of software doesn't adhere to this).
- In CS136, all global variables must be constant!

(B) Available to this module, but not externally.

- Better to use named constants rather than literals for all but the most trivial values (0,1, ...)

(C) Constant is available only within function (or local scope).

Do Loops

We introduced the `while` loop:

```
while (<expr>) {
   ...
}
```

that checks the boolean expression *before* executing the loop.

There is also a form that will always execute the loop code *at least once*, and then check the boolean expression at the end to see if the loop should be repeated:

```
do {
   ...
} while (<expr>)
```

More on `for` loops

In `for` loops, you can have a variable declared right in the `for` statement, and it's scope only exists within the for loop:

```
for (int i=0; i<100; i=i+1) { // note the int
  // i only exists within this loop
}
```

You can also have compound statements within a `for` loop:

```
for (i=0, j=100; i<100; i=i+1, j=j-1) {
  // note the commas above
}
```

This can be terribly misused, but is convenient on occasion.

Manoeuvring through loops

C has two statements which change the way loops are executed

- `break;` causes the program to leave the *innermost* loop it is executing and to execute the statement immediately following that loop

```
for(int i=0; i<10; i=i+1) {
  for (int j=0; j<10; j=j+1) {
    if (i+j>5) {
      break;
    }
    printf("*");
  }
  printf("\n");
}
```

```
******
*****
****
***
**
*
```

- We can always avoid using a `break` using an `if` statement (and some programmers think we should!)

Manoeuvring through loops

C has two statements which change the way loops are executed

- `continue;` causes the program to immediately start the next iteration of the *innermost* loop (skipping the rest of the current iteration)

```
int main(void) {
  for(int i=0; i<5; i++) {
    for (int j=0; j<5; j++) {
      if ( (i+j) % 3 == 0) {
        continue; }
      printf("%d", j);
    }
    printf("\n");
  }
}
```

```
124
0134
023
124
0134
```

- We can always avoid using a `continue` with an `if` statement (and many programmers think we should!)

Loop Mistakes

A misplaced "`;`" can cause a lot of headaches in loops:

```
while (i > 0); {
  printf("%d\n",i);
  i = i - 1;
}
```

this is an infinite loop (if `i > 0`) because technically only the *very next statement* is repeated in a while loop. Because of the misplaced "`;`" above, the first statement after the while loop is actually an empty statement!

For a similar reason, the following will also become an infinite loop:

```
while (i > 0)
  printf("%d\n",i);
  i = i - 1;
```

this is because only the `printf` statement is repeated. This is why it is always good coding style to use compound statements `{...}` with loops and `if` statements.

Operators and side effects

Increment (`++`) and decrement (`--`) operators

- Operators `++` and `--` have the *side-effect* of incrementing or decrementing a variable

```
int i=5;
i++;  // increment i
printf("i --> %d\n",i);
```

- Operators `++` and `--` also have a *value*

```
printf("5*(i--) --> %d\n",5*(i--));
```

- Moreover `i++` is different from `++i` (!!!)
  - `i++` is a *post-increment*: `i++` has the same value as `i`, and then `i` is incremented
  - `++i` increments `i` then returns its new value
- Best to use `i++` and `i--` just for their side-effects

Update and assignment operators

C also has update operators `+=`, `-=`, `*=`, ...

- Equivalent to performing the operator on the LHS variable
- "`i+=5;`" is equivalent to `i=i+5;`
- While it's just "syntactic sugar" it is quite mnemonic and recommended when appropriate

Assignment statements as operators

- Assignment "=" is also an operator, returning the *value* being assigned, and having the *side effect* of doing the assignment
- For example:

```
int i,j;
j = 3+(i=5);
```

assigns 5 to `i` and 8 to `j`.

- What about:

```
...
if (i=0) { // this is bad! should be ==
  printf("Don't divide by zero!\n");
  return -1;
}
printf("10/i=%d\n", 10/i);
...
```

Use assignment (=) only as a statement!!!