CS2014
Systems Programming

Lectures:

Stephen Farrell

stephen.farrell@cs.tcd.ie

Teaching Assistant:

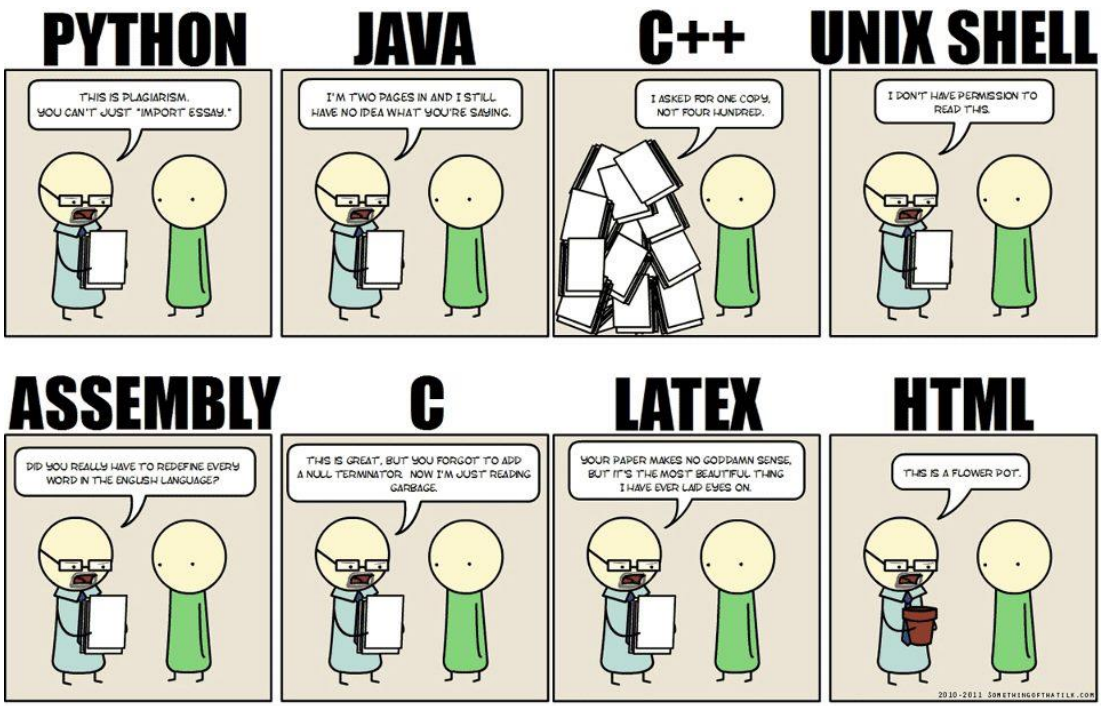
Christian Cabrera

cabrerac@scss.tcd.ie

Review – Programming

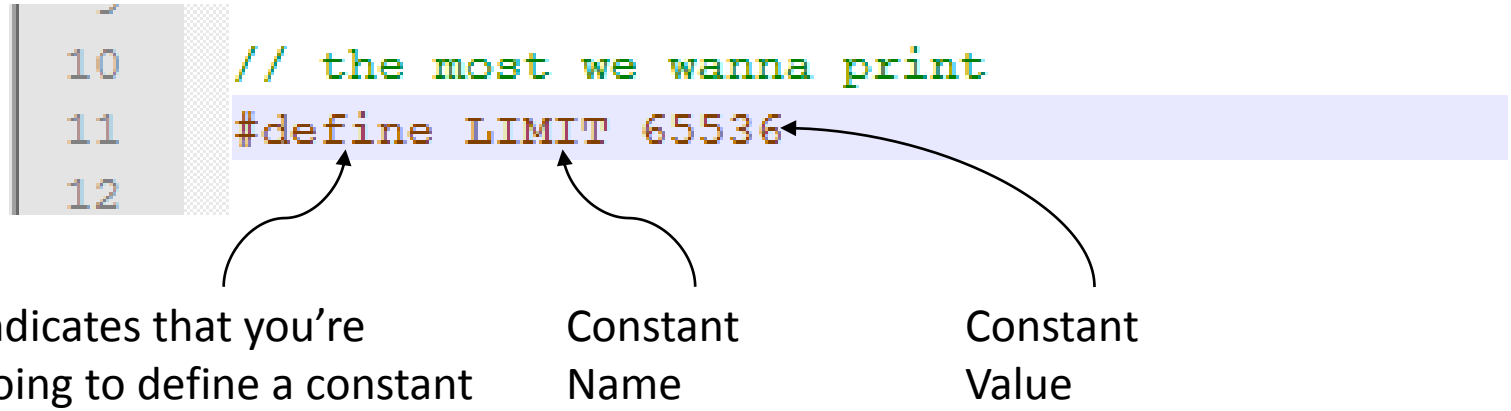
- Problem solving process where you:
  1. Analyse the problem
  2. Design a solution -> Algorithm
  3. Code the solution into an executable computational program
  4. Testing, debugging

  * Google it…



Review – Constants

- Constants are data items that does not change at execution time. Their value cannot be modified after execution



Review –Variables

- Variables are data items that change at execution time. They can be modified after their definition.

```
// various vars as copied from mbedtls-2.6.0/programs/pkey/gen_key.c
int rv;
mbedtls_entropy_context entropy;
mbedtls_ctr_drbg_context ctr_drbg;
const char *pers = "cs2014-coin";
mbedtls_pk_context key;
int pubkeylen=CC_BUFSIZ;
unsigned char pubkey[CC_BUFSIZ];
unsigned char *pubkeyp;
cs2014coin_t thecoin;
int done=0;
unsigned char hashbuf[CC_BUFSIZ];
unsigned char noncebuf[CC_BUFSIZ];
mbedtls_md_context_t sha_ctx;
int hilen;
unsigned char hival[CC_BUFSIZ];
unsigned char *bp; // general pointer into buffer
unsigned char *np; // pointer just beyond end of nonce in hashed input
int nonce_iterations=0; /// used while guessing
unsigned long htonlout;
unsigned char sigbuf[CC_BUFSIZ];
size_t siglen;
```

Optional

<data type> <variable name> = <value>;

Basic data types:

- Integers: Numbers that can be both positive or negative: char, int, short, long
- Unsigned integers: Numbers that can only be positive: unsigned char, unsigned int, unsigned short, unsigned long
- Floating point numbers: Real numbers: float, double

Review –Variables

- Strings are arrays of characters in C. There are different ways to define them:

```c
#include <stdio.h>

int main() {
    char * name = "John Smith";

    char name[] = "John Smith";

    char name[11] = "John Smith";


return 0;
}

```

Only reading!!!

The compiler calculates the size

The size is predefined

- Useful functions related to strings: strlen(), strncmp(), strncat()…

Review –Variables

- Variables are local to the scope in which they are defined, but:
    - They can be declared as static to increase their scope up to file containing them. You just need to put the word static before the data type
- Global variables can be accessed outside the file too.

Review – Output

- If you want to show the variables value:

```
printf("%d too small\n",newnumber);
```

It will print the value of the variable “newnumber” that is an integer

Review – Output

- If you want to show the variables value:

```
printf("%d too small\n",newnumber);
```

It will print the value of the variable “newnumber” that is an integer

You need to use the right format specifier which holds the place for the actual value of the variable.

%d – int
%ld – long
%f – float
%lf – double
%c – char
%s – string
%x – hexadecimal;

```
1 #include <stdio.h>
2
3 int main() {
4     int number = 5;
5     number = number * 5;
6     char name[] = "John Smith";
7     printf("this is an integer %d, now a string %s", number, name);
8
9 return 0;
10 }
```

Review – Input

- If you want to read the variables value from the user:

```c
#include "stdio.h"

int main(void)
{
    int a;

    printf("Please input an integer value: ");
    scanf("%d", &a);
    printf("You entered: %d\n", a);

    return 0;
}
```

Review – Input

- If you want to read the variables value from the user:

```c
#include "stdio.h"

int main(void)
{
    int a;
    printf("Please input an integer value: ");
    scanf("%d", &a);
    printf("You entered: %d\n", a);

    return 0;
}
```

Declare the variable to be assigned

Assign the variable with the input, use the same format specifier and do not forget the “&”!

Review – Input

- If you want to read a file:

```
int main(int argc,char *argv[])
{
    int number=10;
    FILE *fout=NULL;
    char *fname;

    if (argc==3) {
        int newnumber=atoi(argv[1]);
        if (newnumber<=0) {
            printf("%d too small\n",newnumber);
            usage(argv[0]);
        }
        if (newnumber>LIMIT) {
            fprintf(stderr,"%d too big\n",newnumber);
            usage(argv[0]);
        }
        number=newnumber;
        fname=argv[2];
    } else {
        usage(argv[0]);
    }

    if ((fout=fopen(fname,"w"))==NULL) {
        fprintf(stderr,"can't open output file %s\n",fname);
        usage(argv[0]);
    }
```

Review – Input

- If you want to read a file:

```
int main(int argc,char *argv[])
{
    int number=10;
    FILE *fout=NULL;
    char *fname;

    if (argc==3) {
        int newnumber=atoi(argv[1]);
        if (newnumber<=0) {
            printf("%d too small\n",newnumber);
            usage(argv[0]);
        }
        if (newnumber>LIMIT) {
            fprintf(stderr,"%d too big\n",newnumber);
            usage(argv[0]);
        }
        number=newnumber;
        fname=argv[2];
    } else {
        usage(argv[0]);
    }

    if ((fout=fopen(fname,"w"))==NULL) {
        fprintf(stderr,"can't open output file %s\n",fname);
        usage(argv[0]);
    }
```

Declare pointer of type FILE

Opening file

Review – Input

- If you want to read a file:

```
int main(int argc,char *argv[])
{
    int number=10;
    FILE *fout=NULL;
    char *fname;

    if (argc==3) {
        int newnumber=atoi(argv[1]);
        if (newnumber<=0) {
            printf("%d too small\n",newnumber);
            usage(argv[0]);
        }
        if (newnumber>LIMIT) {
            fprintf(stderr,"%d too big\n",newnumber);
            usage(argv[0]);
        }
        number=newnumber;
        fname=argv[2];
    } else {
        usage(argv[0]);
    }

    if ((fout=fopen(fname,"w"))==NULL) {
        fprintf(stderr,"can't open output file %s\n",fname);
        usage(argv[0]);
    }
```

Declare pointer of type FILE

Opening modes

Opening file

Opening Modes in Standard I/O

File Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e, Data is added to end of file.	If the file does not exists, it will be created.
ab	Open for append in binary mode. i.e, Data is added to end of file.	If the file does not exists, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exists, it will be created.

Review – Input

- If you want to read a file:

```
for (int i=0;i!=number;i++) {
    unsigned char rndcount=rndbyte();
    unsigned char randj;
    fprintf(fout,"%d,%02x",i,rndcount);
    if (!rndcount) { // we *can* get zero!
        fprintf(fout,"\n\n");
    } else {
        for (int j=0;j!=(rndcount-1);j++) {
            randj=rndbyte();
            if (!(j%8)) fprintf(fout,"\n");
            fprintf(fout,"%02x,",randj);
        }
        // and add the last one too
        randj=rndbyte();
        if ((rndcount%8)==1) fprintf(fout,"\n%02x\n\n",randj);
        else fprintf(fout,"%02x\n\n",randj);
    }
}

fclose(fout);

return(0);
}
```

Writing in file with the right format specifier as in “printf”

Writing in file with the right format specifier as in “printf”

Closing file, do not forget it!

Review – Arrays

- Arrays are special variables that can hold more than one value using the same variable (e.g., strings)
- You need to use the index to use the array, the index value is from 0 to array length - 1

Review – Arrays

- Arrays are special variables that can hold more than one value using the same variable (e.g., strings)
- You need to use the index to use the array, the index value is from 0 to array length - 1

```
13  #include <stdio.h>
14
15  int main() {
16    int grades[3];
17    int average;
18
19    grades[0] = 80;
20    grades[1] = 85;
21    grades[2] = 90;
22
23
24
25    average = (grades[0] + grades[1] + grades[2]) / 3;
26    printf("The average of the 3 grades is: %d", average);
27
28    return 0;
29  }
```

Array of integers definition, specifying the length

Assigning values to the array for each index

Using the stored values to compute an average

Review – Loops

- What if you want to print the “Hello world” five times?

```
13  #include <stdio.h>
14
15  main() {
16
17      printf( "Hello, World!\n");
18      printf( "Hello, World!\n");
19      printf( "Hello, World!\n");
20      printf( "Hello, World!\n");
21      printf( "Hello, World!\n");
22
23  }
```

Review – Loops

- What if you want to print the “Hello world” five times?

```
13  #include <stdio.h>
14
15  main() {
16
17      printf( "Hello, World!\n");
18      printf( "Hello, World!\n");
19      printf( "Hello, World!\n");
20      printf( "Hello, World!\n");
21      printf( "Hello, World!\n");
22
23  }
```

What if you want to print the “Hello world” 1000 times???

- A practical solution is to use a loop that is a code block that runs multiple times according to given conditions.

Review – Loops

- The while loop continues executing the while block as long as the condition in the while holds.

```
 1 #include <stdio.h>
 2
 3 int main() {
 4     int n = 0;
 5     while (n < 10) {
 6         n++;
 7     }
 8     while (1) {
 9     /* do something */
10     }
11
12 return 0;
13 }
14
```

Code executed 10 times

Code executed infinitely

Review – Loops

```c
#include <stdio.h>

int main() {
    int array[] = {1, 7, 4, 5, 9, 3, 5, 11, 6, 3, 4};
    int i = 0;

    while (i < 10) {
        if(array[i] < 5){
            i++;
            continue;
        }
        if(array[i] > 10){
            break;
        }
        printf("%d\n", array[i]);
        i++;
    }
    return 0;
}
```

Iterate over the elements of an array

Go back to the start of the while block

Break the while loop, even though the while loop never finishes

Review – Loops

- The for loop requires an iterator variable. The loop initializes the iterator, checks if the iterator reached its final value, and increments the iterator

```c
#include <stdio.h>

int main() {
  int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  int factorial = 1;

  for(int j=0; j<10; j ++){
    factorial = factorial * array[j];
  }

  printf("10! is %d.\n", factorial);
}
```

Iterate over the elements of an array

Review – Functions

- Functions are "self contained" modules of code that accomplish a specific task.
- Functions usually "take in" data, process it, and "return" a result.
- Functions can be used over and over and over again. Functions can be "called" from the inside of other functions.

Review – Functions

Return Type

Function Name

Arguments

```
int cs2014coin_make(int bits, unsigned char *buf, int *buflen)
{
    // various vars as copied from mbedtls-2.6.0/programs/pkey/gen_key.c
    int rv;
    mbedtls_entropy_context entropy;
    mbedtls_ctr_drbg_context ctr_drbg;
    const char *pers = "cs2014-coin";
    mbedtls_pk_context key;
    int pubkeylen=CC_BUFSIZ;
    unsigned char pubkey[CC_BUFSIZ];
    unsigned char *pubkeyp;
    cs2014coin_t thecoin;
```

Implementation

Review – Functions

Return Type

Function Name

Arguments

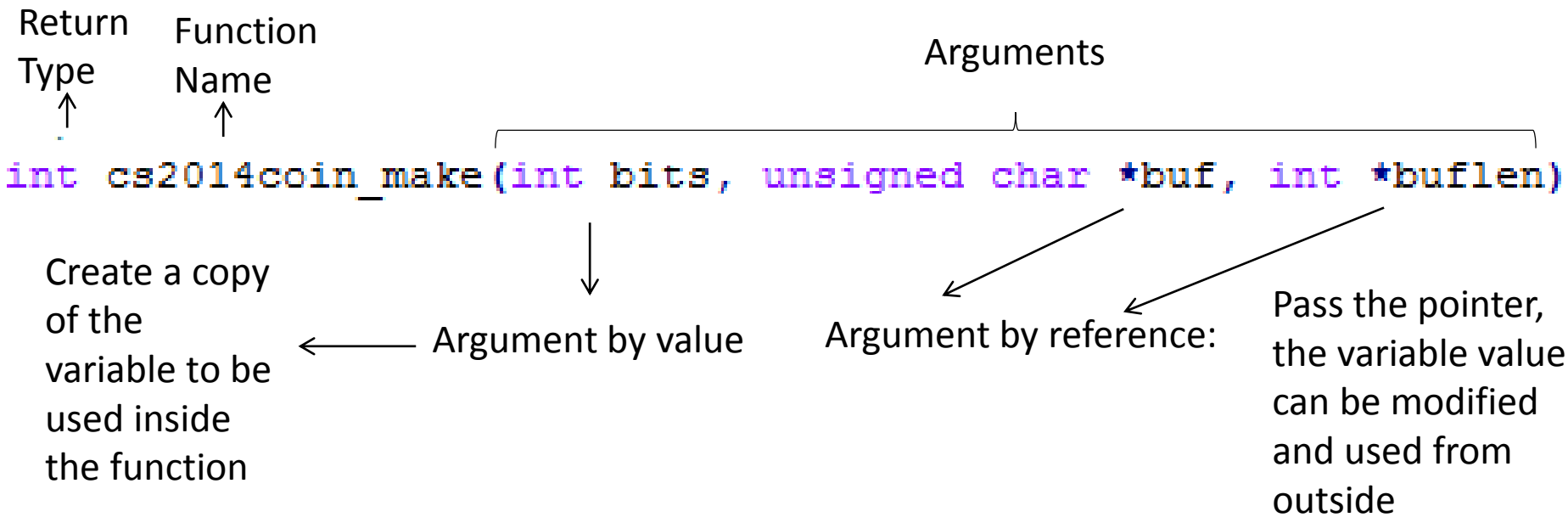
```
int cs2014coin_make(int bits, unsigned char *buf, int *buflen)
{
    // various vars as copied from mbedtls-2.6.0/programs/pkey/gen_key.c
    int rv;
    mbedtls_entropy_context entropy;
    mbedtls_ctr_drbg_context ctr_drbg;
    const char *pers = "cs2014-coin";
    mbedtls_pk_context key;
    int pubkeylen=CC_BUFSIZ;
    unsigned char pubkey[CC_BUFSIZ];
    unsigned char *pubkeyp;
    cs2014coin_t thecoin;

    FILE *fp;
    int byteswritten;
    int actualsize=CS2014COIN_BUFSIZE;
    rv=cs2014coin_make(bits,coinbuf,&actualsize);
    if (rv) {
        fprintf(stderr,"Error (%d) making coin (%s) - exiting\n
```

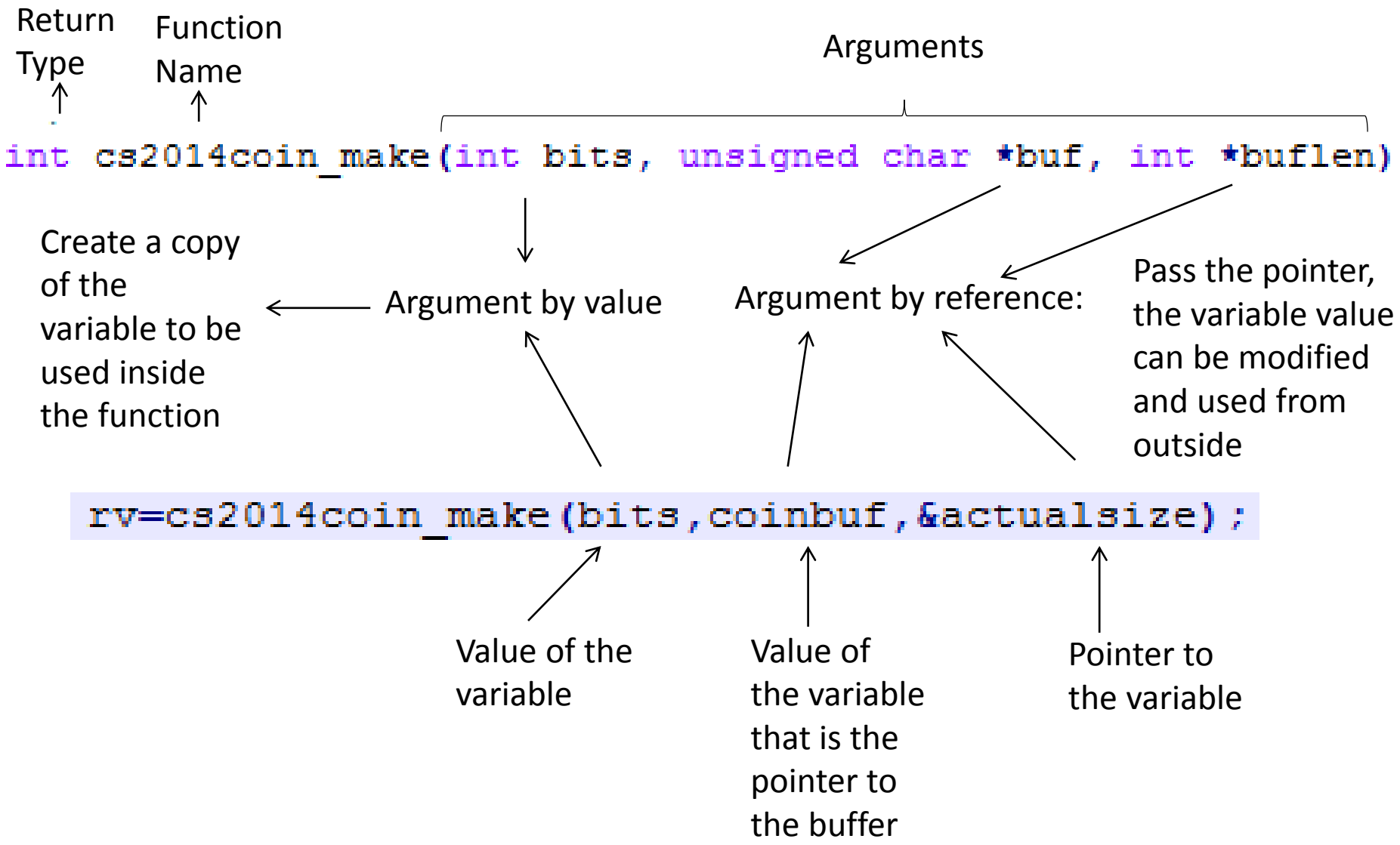
Implementation

Function call

Review – Functions



Review – Functions



Review – Recursion

- Recursion means "defining a problem in terms of itself".
- This can be a very powerful tool in writing algorithms.
- Recursion comes directly from Mathematics, where there are many examples of expressions written in terms of themselves. For example, the Fibonacci sequence is defined as: F(i) = F(i-1) + F(i-2)

Review – Recursion

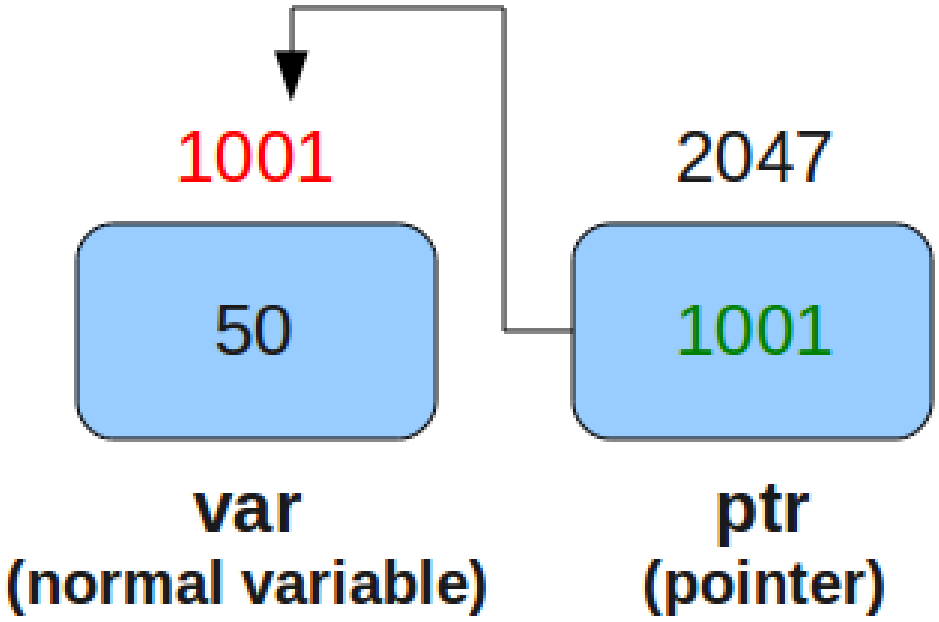
```
void incr_nonce(unsigned char *ptr)
{
    unsigned char ch=*(ptr-1);
    if (ch==255) {
        incr_nonce(ptr-1);
        *(ptr-1)=0;
    } else {
        *(ptr-1)=(ch+1);
    }
    return;
}
```

Recursive Step

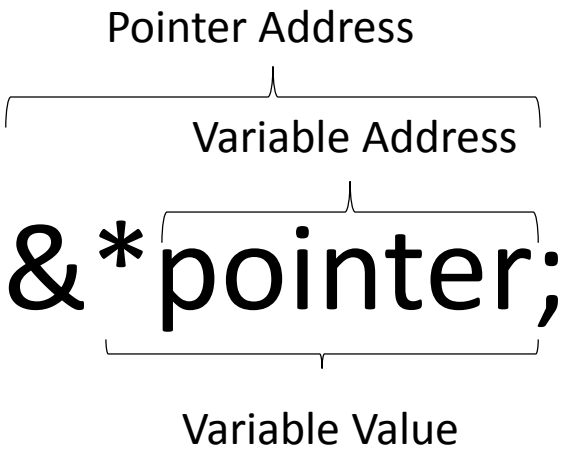
Terminating case

Review – Pointers

- Pointers are also variables which hold a memory address that points to a value, instead of holding the actual value itself.



Review – Pointers




```
 2
 3 int main() {
 4   int n = 10; ——→ Variable n with value 10
 5
 6   int * pointer_to_n = &n; ——→ Pointer to n
 7
 8   *pointer_to_n += 1; ——→ Increment the value of n using the pointer
 9
10   /* testing code */
11   if (pointer_to_n != &n) return 1;
12   if (*pointer_to_n != 11) return 1;
13   printf("Done!\n");
14   return 0;
15 }
```

Review – Pointers

- Arithmetic operations over pointers: increment, decrement, compare address values

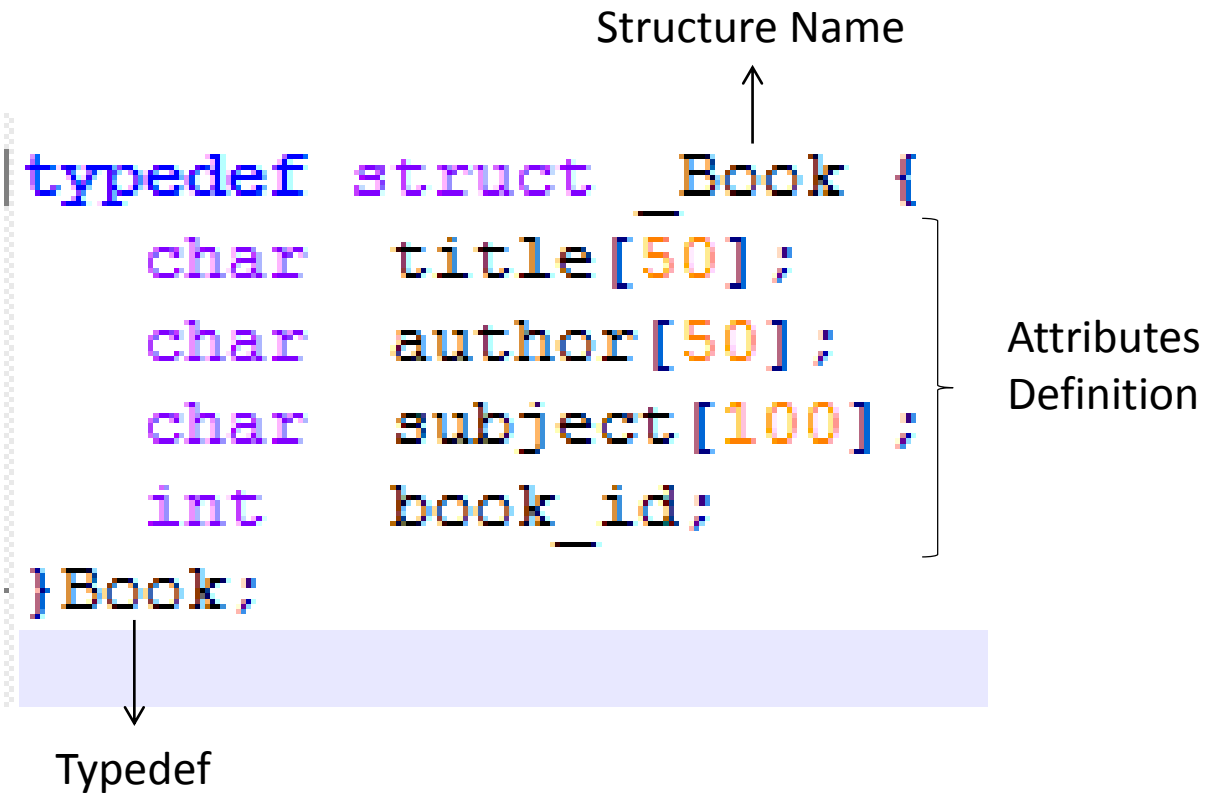
```
int hilen;
unsigned char hival[CC_BUFSIZ];
unsigned char *bp; // general pointer into buffer
// prepare hash input
hilen=0;
bp=hival;
// hand encoding, sure why not
memset(bp,0,CC_BUFSIZ);
htonlout=htonl(thecoin.ciphersuite);
memcpy(bp,&htonlout,4);
hilen += 4;
bp=hival+hilen; ⟶ Increment address pointed by bp
```

Review – Structures

- A structure is a composition of different variables of different data types , grouped under a same name.
- Structures group several pieces of related information together
- E.g.: Suppose you want to keep track of your books in a library, you can track the next attributes of a book:
  - Title
  - Autor
  - Subject
  - Book id

Review – Structures

- Book:
  - Title
  - Autor
  - Subject
  - Book id



Review – Structures

```c
typedef struct _Book {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
}Book;

int main( ) {

   Book book1;        /* Declare book1 of type Book*/

   /* book 1 specification */
   strcpy( book1.title, "C Programming");
   strcpy( book1.author, "Nuha Ali");
   strcpy( book1.subject, "C Programming Tutorial");
   book1.book_id = 6495407;

   /* print Book1 info */
   printf( "Book 1 title : %s\n", book1.title);
   printf( "Book 1 author : %s\n", book1.author);
   printf( "Book 1 subject : %s\n", book1.subject);
   printf( "Book 1 book_id : %d\n", book1.book_id);

   return 0;
}
```

Linked Lists

- Arrays are useful, but they have some limitations:

Linked Lists

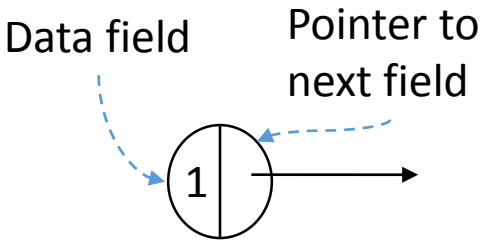
- Arrays are useful, but they have some limitations:
  - Static memory allocation at compilation time. You cannot allocate additional space at run time
  - Insertion and deletion operations are difficult, you need to move all the array after the operation. It is not efficient with large size arrays
  - Bound checking, C is not going to give you an exception if you access a value that exceeds the array size, you will get garbage
  - Wastage of memory

Linked Lists

- Linked lists provide dynamic memory allocation, easy insertion/deletion operations, and bound checking.
- Linked lists are sequences of nodes

Linked Lists

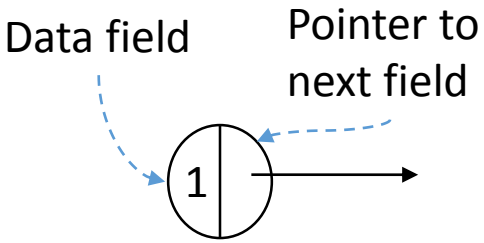
- Linked lists provide dynamic memory allocation, easy insertion/deletion operations, and bound checking.
- Linked lists are sequences of nodes



What can we use to represent it in C?

Linked Lists

- Linked lists provide dynamic memory allocation, easy insertion/deletion operations, and bound checking.
- Linked lists are sequences of nodes



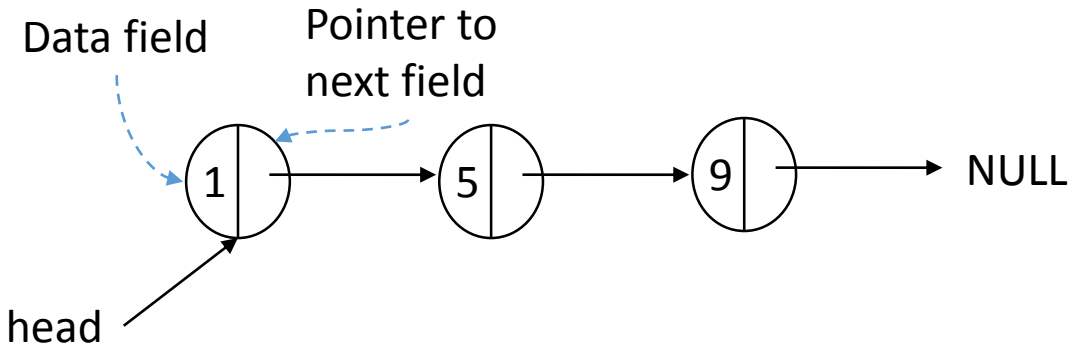
What can we use to represent it in C?

```
typedef struct node {
    int val;
    struct node * next;
} node_t;
```

Structures and pointers!

Linked Lists

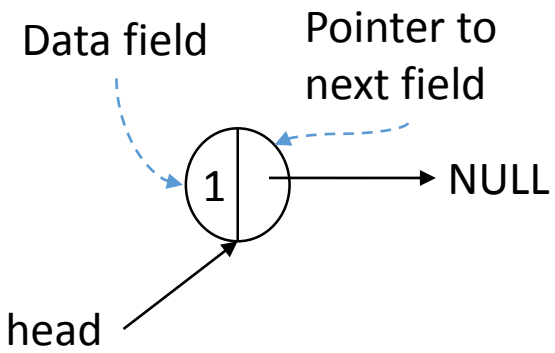
- Linked lists provide dynamic memory allocation, easy insertion/deletion operations, and bound checking.
- Linked lists are sequences of nodes




```
typedef struct node {
    int val;
    struct node * next;
} node_t;
```

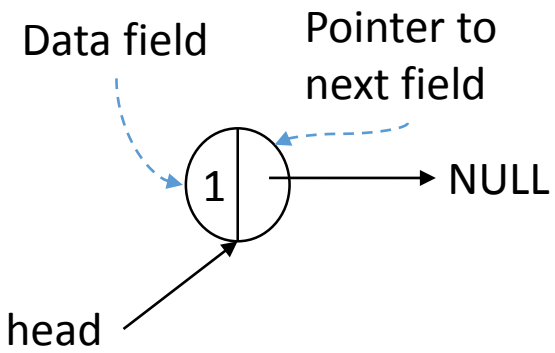
Linked Lists

- Creating the list



Linked Lists

- Creating the list




```
typedef struct node {
    int val;
    struct node * next;
} node_t;
```

```
node_t * head;
head = malloc(sizeof(node_t));
head->val = 1;
head->next = NULL;
```

Linked Lists

- Printing the list:

```
void print_list(node_t * head) {
    printf("Current List:\n");
    node_t * current = head;
    while (current != NULL) {
        printf("%d\n", current->val);
        current = current->next;
    }
}
```

- Length of the list:

```
int length(node_t ** head) {
   int length = 0;
   struct node *current;

   for(current = head; current != NULL; current = current->next) {
      length++;
   }

   return length;
}
```

- Is the list empty?

```
int isEmpty(node_t * head) {
  if (head == NULL){
    return 0;
  }
  else {
    return 1;
  }
}
```

Linked Lists

- Adding elements to the list, two options:

```
void add_element_end(node_t * head, int val) {
    node_t * current = head;
    while (current->next != NULL) {
        current = current->next;
    }
    current->next = malloc(sizeof(node_t));
    current->next->val = val;
    current->next->next = NULL;
}
```

At the end

```
void add_element_beginning(node_t ** head, int val){
  node_t * new_node;
    new_node = malloc(sizeof(node_t));

    new_node->val = val;
    new_node->next = *head;
    *head = new_node;
}
```

At the beginning

Linked Lists

- Removing elements from the list:

```c
int remove_first_element(node_t ** head) {
    int retval = -1;
    node_t * next_node = NULL;

    if (*head == NULL) {
        return -1;
    }

    next_node = (*head)->next;
    retval = (*head)->val;
    free(*head);
    *head = next_node;

    return retval;
}
```

Removing first element

```c
int remove_last_element(node_t * head) {
    int retval = 0;
    /* if there is only one item in the
    list, remove it */
    if (head->next == NULL) {
        retval = head->val;
        free(head);
        return retval;
    }

    /* get to the last node in the list */
    node_t * current = head;
    while (current->next->next != NULL) {
        current = current->next;
    }

    /* now current points to the last
    item of the list, so let's remove
    current->next */
    retval = current->next->val;
    free(current->next);
    current->next = NULL;
    return retval;
}
```

Removing last element

Linked Lists

- Removing elements from the list:

```c
int remove_first_element(node_t ** head) {
    int retval = -1;
    node_t * next_node = NULL;

    if (*head == NULL) {
        return -1;
    }

    next_node = (*head)->next;
    retval = (*head)->val;
    free(*head);
    *head = next_node;

    return retval;
}
```

Removing first element

```c
int remove_last_element(node_t * head) {
    int retval = 0;
    /* if there is only one item in the
    list, remove it */
    if (head->next == NULL) {
        retval = head->val;
        free(head);
        return retval;
    }

    /* get to the last node in the list */
    node_t * current = head;
    while (current->next->next != NULL) {
        current = current->next;
    }

    /* now current points to the last
    item of the list, so let's remove
    current->next */
    retval = current->next->val;
    free(current->next);
    current->next = NULL;
    return retval;
}
```

Removing last element

Linked Lists

- Removing elements from the list:

```
int remove_by_index(node_t ** head, int n) {
    int i = 0;
    int retval = -1;
    node_t * current = *head;
    node_t * temp_node = NULL;

    if (n == 0) {
        return remove_first_element(head);
    }

    for (int i = 0; i < n-1; i++) {
        if (current->next == NULL) {
            return -1;
        }
        current = current->next;
    }

    temp_node = current->next;
    retval = temp_node->val;
    current->next = temp_node->next;
    free(temp_node);

    return retval;
}
```

Removing by index

Linked Lists

- Find by value:

```
node_t* find_by_value(node_t * head,int val) {

   //start from the first link
   struct node* current = head;

   //if list is empty
   if(head == NULL) {
      return NULL;
   }

   //navigate through list
   while(current->val != val) {

      //if it is last node
      if(current->next == NULL) {
         return NULL;
      } else {
         //go to next link
         current = current->next;
      }
   }

   //if data found, return the current Link
   return current;
}
```