Lenguaje C

Historia

- Desarrollado por Dennis Ritchie entre 1969 y 1973 en los Laboratorios AT&T Bell.
- Diseñado para ser compilado de una manera directa.
  - Sus construcciones se pueden mapear directamente a instrucciones de máquina (ensamblador).
- A la vez, que sea independiente de plataforma.

Beneficios

- Se puede compilar en varias plataformas requiriendo pocas modificaciones.
- Provee acceso de bajo nivel a memoria.
- Requiere de mínimo soporte de ejecución.

Mínima Gramática

```
#include <stdio.h>
int a = 1;
int sum(int b){
   return a+b;
}
int main(){
   printf(“hello world, %d \n”, sum(2));
   return 1;
}
```

Compilar

Compilación

- Para poder proseguir, se requiere poder compilar.
- Cada plataforma requiere de diferentes módulos para llevar a cabo esto.
- En el caso de Linux, hay un compilador magnífico, no sólo por ser muy robusto y bien desarrollado, sino que también es gratis.

Nuestro amigo: GCC

- Para instalarlo en Ubuntu:

  sudo apt-get install build-essential

- Es muy probable que ya lo tengan instalado.

- Funciona así:

  gcc -o programa codigo.c

  - *programa* siendo el nombre del programa compilado
  - *codigo.c* siendo el nombre del archivo con el código.

Incluir librerias

- Vamos a ver más adelante que se requieren añadir librerias (específicamente para cuando lleguemos a JACK).
- Estas librerias, aunque instaladas, se le tiene que especificar a gcc cuales son necesarias, y, dado el caso, donde encontrarlas en el sistema.

Incluir librerias

- Si ya están instaladas en lugares conocidos como /lib, /usr/lib, etc., entonces es sólo:

  gcc -o programa codigo.c -ljack

  - *libjack* es el nombre de la libreria añadida
    - Es una l (ele minúscula), seguido por el nombre de la libreria sin el inicial “lib”.

Incluir librerias

- Si están instaladas en otro lugar:

  gcc -o programa codigo.c -ljack -L/var/otroslibs

  - */var/otroslibs* es un directorio (adicional a los conocidos) donde gcc debe buscar por librerias
    - Es un L (ele mayúscula), seguido por el directorio

Incluir Headers

- Archivos con terminación “.h” que son descriptores de la librería (headers).
- Algunas veces, dichos headers no se encuentran en el mismo lugar que la librería, por lo que se le tiene que decir a gcc donde buscarlos.

  gcc -I/var/otrosheaders -o programa codigo.c

  - Es una i mayúscula seguido por un directorio.
  - Nótese que van ANTES que el -o, las librerías después.

Estilo de Gramática

- El lenguaje C ha tenido varias modificaiones menores durante los años.
- Se le puede indicar a gcc cual versión del lenguaje se quiere utilizar.

  gcc -std=gnu99 -o programa codigo.c

  - Std es el argumento del estilo, y gnu99 es la versión del lenguaje que estaremos usando mayormente.

Nuestro mejor amigo: Make

- Ya que la línea de compilación puede llegar a ser muy larga y complicado, build-essentials incluye una utilería llamada *make*.
- Al correr make, éste busca un archivo llamado Makefile o makefile en dicho directorio, y corre los procesos de compilación descritos ahí.

Makefile básico

```
all:
	gcc -o programa codigo.c
```

Makefile Básico

- La línea que contiene el comando de gcc ***DEBE*** comenzar con un tabulador, si no llora.
- El “all” de al principio es una etiqueta genérica a la cual make se va por defecto y lleva a cabo los comandos de compilación justo debajo.
- Se pueden agregar otras etiquetas con otros comandos...

Makefile Poco Menos Básico

```
all:
	gcc -o programa codigo.c

jack:
	gcc -o programa codigo.c -ljack
```

Makefile Poco Menos Básico

- Si se corre “make” se correrá solo lo que está debajo de “all” en el makefile hasta justo antes de la siguiente etiqueta.
  - “jack” en este caso.
- Si se corre “make jack” se correrá lo que está debajo de “jack” en el makefile hasta justo antes de la siguiente etiqueta.
  - Hasta el final del makefile en este caso.

Ahora sí, vamos a divertirnos...

Mínima Gramática

```
#include <stdio.h>
int a = 1;
int sum(int b){
   return a+b;
}
int main(){
   printf(“hello world, %d \n”, sum(2));
   return 1;
}
```

Explicación de Cada Parte

- Include
- Variables Globales
- Funciones
- Main

Cada Parte

```
#include <stdio.h>
int a = 1;
int sum(int b){
   return a+b;
}
int main(){
   printf(“hello world, %d”, sum(2));
   return 1;
}
```

Include/Libreria

Variables Globales

Funciones

Main

Libreria/Include

- Librerias son código que complementa las funcionalidades nativas de C.
  - stdio.h – funciones para manejar hilos de datos, incluyendo datos, dispositivos, archivos, etc.
  - string.h – funciones para manejar arreglo de caracteres (aka. “string”)
  - time.h – funciones para manejar información de horarios y fechas.
  - etc.

Libreria/Include

```
#include <stdio.h>
#include <string.h>
#include <time.h>

#include <time>
```

Funciones

- Una función es una estructura que recibe argumentos y *regresa* un resultado.

```
int function1(int arg1, char arg2);
```

Tipo de Funciones

- Es definido por el tipo de dato que regresa:

  int function1(...)

  char function2(...)

  short function3(...)

Return

```
int function1(int b){
   b = 0;
   return b;
}
int function2(){
   return 0;
}
```

Void

- Excepción: cuando no regresa nada:

  void function4(...)

```
void function4(int a, int b){
   int c = a + b;
}
```

¿Void?

- Puede “regresar” su valor indirectamente por medio de:
  - variables globales
  - escribir a pantalla
  - apuntadores

Alcance de Variables

- Globales
- Locales

Variables Globales

- Definidas fuera de las funciones.
- Pueden ser accedidas/manipuladas en cualquier lado del código.

Variables Locales

- Definidas dentro alguna función.
- Su valor y nombre son “olvidados” al momento en la que la función termine.
- *Precaución:* evitar que una variable local y una global tengan el mismo nombre.
  - C llora al compilar.
  - Y si no llora, aunque normalmente le da prioridad a la local, no es consistente.

Tipo de Variables Nativas

- char: usualmente utilizado para representar letras, ocupa sólo un byte.
- int: representa números enteros, ocupa dos bytes.
- float: representa números con una parte decimal, con precisión de hasta 7 digitos, ocupa 4 bytes.
- double: representa números con una parte decimal, con precisión de hasta 15 digitos, ocupa 8 bytes.

Subtipos de Variables Nativa

- unsigned: sólo números positivos, lo cual dobla el rango del lado positivo:
  - int: -32,768 a 32,767
  - unsigned int: 0 a 65,535
- long: dobla la ocupación de bytes, por lo que el rango se intensifica considerablemente:
  - long int: -2,147,483,648 a 2,147,483,647
  - long unsigned int: 0 a 4,294,967,295

Tipos de Variables No-nativas: Struct y Typedef

- Una librería puede presentar nuevos tipos de variables, usualmente llevadas acabo por una combinación entre *struct* y el uso de *typedef.*
- *struct*: define una variable compuesta de otras ya conocidas.
- *typedef*: define el nombre de una variable con la etiqueta de otra.

struct

```
struct point{
   int x;
   int y;
};
typedef struct point point;

point a = {.x = 1, .y = 2};
int b = a.x * a.y;
```

struct

```
typedef struct {
    int x;
    int y;
} point;

point a = {.x = 1, .y = 2};
int b = a.x * a.y;
```

Variables No-nativas

- No pueden ser precedidas por los sub-tipos unsigned y long.
  - C llora también.
- Deben ser definidas en el área de las variables globales.

Variables No-nativas (C++)

- **Clases.**
  - No vamos entrar en detalle en este momento.
- Son “super-variables” que contienen internamente:
  - Múltiples variables (ala *struct*).
  - Múltiples funciones.
    - Ejemplo importante: Eigen (implementa la biblioteca de manipulación de matrices que vamos a utilizar)

Main

- Es una función especial, la cual es la primera en correrse al mandarse a llamar el programa compilado.
- Como toda función, regresa información, las cuales son códigos de error.
  - Si regresa 0, fue todo exitoso.
  - Si regresa algo diferente, hubo error.
    - El tipo de error es interpretado diferente por cada sistema operativo, así como el mismo programa.

main

```
int main(){
   printf(“hello world”);
   return 0;
}
```

Escribir a Pantalla

- El método nativo de C para escribir a pantalla es printf:

  printf(“cadena a imprimir: %d, %d”,a,b);

  - El primer argumento es la cadena a imprimir, en los cuales va incluido el formato de los siguientes argumentos precedido por un “%”
  - Los siguientes argumentos serán evaluados, y sus valores serán formateados en la cadena a imprimir dependiendo de como hayan sido posicionados en la cadena a imprimir

Escribir a Pantalla

```
int a = 1;
int b = 2;
printf(“cadena a imprimir: %d, %d”,a,b);
```

- Lo impreso:

```
cadena a imprimir: 1, 2
```

Escribir a Pantalla

- Hay muchos tipos de formatos, los mas utilizados:

  d: enteros

  f: flotantes

  s: string (serie de caractéres)

- También hay varios símbolos útiles precedidos por un “\”:

  \n : fin de línea

  \t : tabulador

Escribir a Pantalla

printf(“cadena1”);

printf(“cadena2”);

- Impreso:

cadena1cadena2

Escribir a Pantalla

```
printf(“cadena1\n”);
printf(“cadena2\n”);
```

- Impreso:

```
cadena1
cadena2
```

Cout (C++)

- Para aquellos experimentados en C++, seguramente recordarán el uso del cout para escribir a pantalla.

  cout << “cadena1” << endl;

- Se puede utilizar, pero printf:
  - Es nativo de C.
  - Da mucho control del formato de la cadena impresa.
    - También C++, pero requiere de más trabajo en mi opinión.

Métodos Matemáticos

Aritméticas

a + b

a - b

a * b

a / b

a % b

Trigonométricas

- Incluidas en math.h

  sin, cos, tan

  asin, acos, atan

Math.h

- Al incluir math.h

  #include<math.h>

- También se le debe decir a gcc donde encontrarlo por medio de añadir “libm”:

  gcc -o programa codigo.c -lm

  - “libm” es la libreria de matematicas del que math.h es encabezado

Trucos

```
int a = 0;
a = a + 1;
```

Trucos

```
int a = 0;
a = a + 1;
```

“a” es igual a “1”

Trucos

```
int a = 0;
a++;
```

Trucos

```
int a = 0;
a++;
```

“a” es igual a “1”

Trucos

```
int a = 0;
a--;
```

Trucos

```
int a = 0;
a--;
```

“a” es igual a “-1”

Trucos

```
int a = 0;
a += 2;
```

Trucos

```
int a = 0;
a += 2;
```

“a” es igual a “2”

Trucos

```
int a = 0;
a -= 2;
```

Trucos

```
int a = 0;
a -= 2;
```

“a” es igual a “-2”

Trucos

```
int a = 2;
a /= 2;
```

Trucos

int a = 2;

a /= 2;

“a” es igual a “1”

Condicionales

If

```
if (conditional){
   command1...
   command2...
}
```

Condicionales

>

<

>=

<=

!=

If

```
if (1 > 0){
   command1...
   command2...
}
```

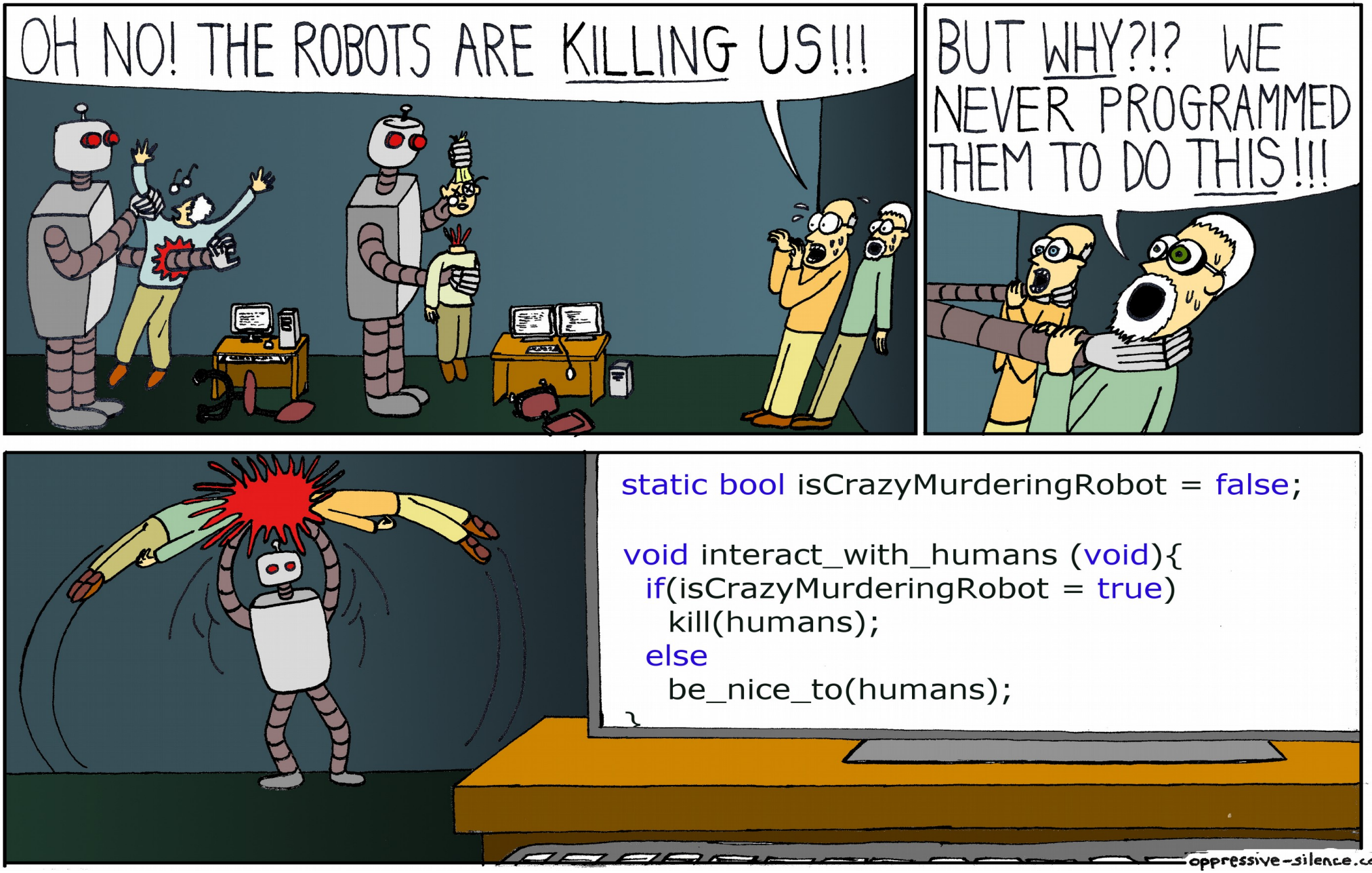
Condicionales

- Cuando una condicional no se cumple, regresa un 0 como valor, y un 1 cuando sí.
- Pero para que el if permita que se entre, lo que tiene dentro de su paréntesis debe regresar algo *diferente* a 0.
  - Se pueden meter adentro del if algo que regrese un 0 y funciona igual.

If

```
int func(){
   return 2;
}

if ( func() ){
   command1...
   command2...
}
```

¡ADVERTENCIA!



Resultado de Operaciones

- La operación de asignar un valor a una variable regresa un resultado de éxito o no éxito.
- Dicho resultado se puede utilizar en una condicional.

Resultado de Operaciones

```
int c = 1;
if(c == 3){
    printf(“true”);
}else{
    printf(“false”);
}
```

Este ejemplo imprimirá “false”, porque c no es igual a 3.

Resultado de Operaciones

```
int c = 1;
if(c = 3){
    printf(“true”);
}else{
    printf(“false”);
}
```

Este ejemplo imprimirá “true”, porque se le asigna exitosamente a c el valor de 3.

Condicionales Compuestas

- Si (esto) ***Y*** (esto)

  &&

  - Son dos *ampersands*
  - (i > 2) && (i < 4)
    - Admitiría sólo cuando i == 3

Condicionales Compuestas

- Si (esto) ***O*** (esto)

  ||

  - Son dos *pipes*
    - Usualmente localizado justo abajo de la tecla ESC en teclados latinoamericanos
  - (i <= 2) || (i >= 4)
    - Admitiría sólo cuando i != 3

Condicionales Compuestas

- Se pueden combinar && y II en una sola condicional.
  - Se recomienda utilizar paréntesis para sanidad mental.

    ((i<=2) || (i>=4)) && (i == 3)
    - Nunca se cumple:

      ((i<=2) || (i>=4)) sólo es verdadero cuando i !=3

      (i == 3) sólo es verdadero en la situación contraria

      && requiere que los dos se cumplean en la misma situación para que toda la condicional sea verdadera

If, Else

```
if (1 > 0){
   command1...
}else{
   command2...
}
```

If, Else if, Else

```
if (1 > 0){
   command1...
}else if (1 < 0){
   command2...
}else{
   command3...
}
```

If, Else if, Else if, Else

?????

Switch

```
switch(a){
   case 1:
      command1...
      break;
   case 2:
      command2...
      break;
   case 3:
      command3...
      break;
   default:
      command4...
      break;
}
```

Bucles

While

- Es como un if, pero en vez de llevar a cabo sus comandos una vez, lo hace repetidamente ***mientras*** que la condicional se cumpla.

While

```
int a = 0;
while(a < 0){
   command1...
}
```

¿Qué problema ven aquí?

While

```
int a = 0;
while(a < 0){
   command1...
}
```

¿Qué problema ven aquí?

“a” no es menor a “0”, por lo que no va a entrar al bucle

Y, ¿así?

```
int a = 0;
while(a <= 0){
   command1...
}
```

¿Qué problema ven aquí?

Y, ¿así?

```
int a = 0;
while(a <= 0){
   command1...
}
```

¿Qué problema ven aquí?

“a” no cambia de valor, por lo que el bucle es infinito

CTRL+C es un buen amigo...

¿Cómo le hacemos para que corra command1 dos veces?

Así...

```
int a = 0;
while(a < 2){
   command1...
   a++;
}
```

O, ¿así?

```
int a = 2;
while(a > 0){
   command1...
   a--;
}
```

Do while

- Es muy parecido al while.
- La única diferencia es que revisa la condicional **después** de haber hecho los comandos dentro del bucle.
  - While lo hace *antes*.
- Por lo tanto, siempre corre lo que está adentro del bucle por lo menos una vez.

Do While

```
int a = 2;
do{
   command1...
   a--;
}while(a < 0);
```

For

- Es un tipo de while, pero que ya tiene en su definición la inicialización, condicional, y que hacer cada vez que se corra lo dentro del bucle.

For

```
for( int i = 0 ; i < 10 ; i++ ){
   command1...
}
```

For

- Las variables utilizadas en la definición del for son accesibles adentro del bucle.

For

```
for( int i = 0 ; i < 10 ; i++ ){
   printf(“contador a %d \n”,i);
}
```

Break, Continue

- Se puede interrumpir el proceso del bucle ya sea utilizando:
  - break: interrumpe y salte del bucle
  - continue: interrumpe y brinca al siguiente paso del bucle

Break

```
for( int i = 0 ; i < 10 ; i++ ){
   if (i == 2){
      break;
   }else{
      printf(“contador a %d \n”,i);
   }
}
```

Continue

```
for( int i = 0 ; i < 10 ; i++ ){
   if (i == 2){
      continue;
   }else{
      printf(“contador a %d \n”,i);
   }
}
```

Apuntadores

Definición de Apuntadores

- Una variable que apunta a una dirección de la memoria.
- Se identifican por su famoso asterisco:

  int *a;

  - Aunque “a” parezca un entero, su valor es sólo la dirección de memoria donde **comienza** un entero.

Uso de Apuntadores

- Variables Grandes
- Variables Complejas
- Arreglo de Variables

Variables Grandes

```
void func1(int b){
   b = 1;
}
int main(){
   int a = 0;
   func1(a);
   printf(“%d”,a);
}
```

Variables Grandes

- Normalmente, al pasar un argumento a una función, se pasa **una copia** del valor.
- Esto puede ser un problema si tenemos una variable que ocupa mucho espacio que queremos pasar como argumento a una función.
- A menos que...

Variables Grandes

```
void func1(int *b){
   *b = 1;
}
int main(){
   int a = 0;
   func1(&a);
   printf(“%d”,a);
}
```

Variables Grandes

- La función sólo recibe apuntadores,
  - Al asignar el valor de 1 a “*b”, se lo está asignando en la dirección de memoria que se le ha pasado.
- Dentro del main, al hacer “&” se le dice al compilador que no se está mandando el valor de la variable, si no su dirección.

Variables Complejas

- Hay librerías que tienen sus variables no nativas tan complejas/grandes, que es más fácil manipularlas como apuntadores.
  - Un puerto de JACK es del tipo jack_port_t.
  - Para crear uno, se utiliza la función jack_port_register, que regresa un valor del tipo jack_port_t *

Variables Complejas

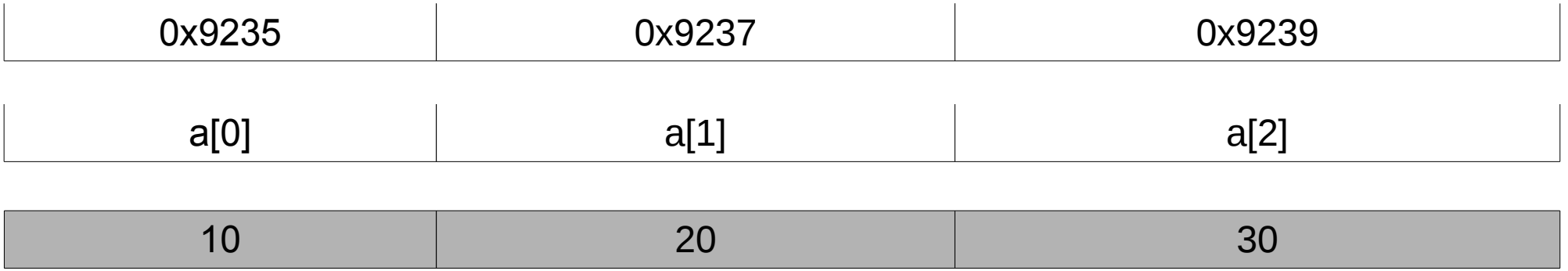
```
#include <jack/jack.h>

jack_port_t   *input_port;

input_port = jack_port_register (...);
```

Arreglo de Datos

- Un arreglo es una serie de secciones de memoria del mismo tamaño.
- Para acceder al valor guardada alguna de las secciones, se requiere de un apuntador del **inicio** de la serie, y un número identificador de la sección para saber donde está ubicada.

Arreglo de Datos

```
int a[3] = {10, 20, 30};
```



Inicialización de Arreglo de Datos

```
#include <stdlib.h>
int *a;
int main(){
   a = malloc(3 * sizeof(int));
   a[0] = 10;
   a[1] = 20;
   a[2] = 30;
}
```

Malloc

- Es posible inicializar un arreglo de datos sin saber su tamaño final.
- **malloc** hace espacio en la memoria dado un número de bytes.
  - Requiere de stdlib.h
- **sizeof** indica el número de bytes que ocupe un tipo de variables.

```
malloc(3 * sizeof(int))
```

Regreso de Malloc

- El compilador es suficientemente inteligente para saber que el número de bytes los tiene que “arreglar” en dos en dos por el tipo de apuntador al cual está siendo guardado.

```
int *a = malloc(3 * sizeof(int));
```

- Esto es equivalente a:

```
int a[3];
```

Regreso de Malloc (cont.)

- Pero de vez en cuando el compilador no es tan inteligente y se le tiene que hacer un *cast* a lo que regresa para hacerle saber explícitamente qué es lo que esta regresando:

```
int *a = (int *)malloc(3*sizeof(int));
```

Regreso de Malloc

- Por cierto:

```
int *a = (int *)malloc(3 * sizeof(int));
```

- Es equivalente a:

```
int a[3];
```

¿Para que entonces malloc?

- Por que es posible que tengamos una variable global que es un arreglo de datos, y queramos que sea accedida por todo el código, pero que no sepamos su tamaño hasta después de comenzar el programa.
- Por ejemplo:
  - Una variable global que nos dé acceso a todos los puertos de entrada de JACK, pero no sepamos al principio cuántos puertos tenemos accesibles.

Arreglo de Variables Complejas

```
#include <stdlib.h>
#include <jack/jack.h>

jack_port_t   **input_port;
int main(){
   int num_port;

   /* código que obtiene cuantos puertos de JACK hay disponibles */
   /* y lo guarda en “num_port” */

   input_port = (input_port**)malloc (num_port * sizeof (jack_port_t *));
}
```

¿Quién puede explicar este código?

Arreglo de Variables Complejas

```
#include <stdlib.h>
#include <jack/jack.h>

jack_port_t   **input_port;
int main(){
   int num_port;

   /* código que obtiene cuantos puertos de JACK hay disponibles */
   /* y lo guarda en “num_port” */

   input_port = (input_port**)malloc (num_port * sizeof (jack_port_t *));
}
```

*input_port es un **arreglo** de **variables complejas**, por eso el doble apuntador*

*se reserva la memoria para una cantidad “**num_port**” de variables complejas tipo “**jack_port_t***”

Inicialización de Arreglo de Variables Complejas

Para registrar un puerto se utiliza la función jack_port_register.

¿Qué código le agregarían para que todos los puertos se registren?

Que pase al frente el/la valiente...

Arreglo de Arreglos

int a[2][3];

¿Cómo creen que se acceda cada celda?

Escriban el código que llene la matriz “a” con los siguientes valores.

	0	**1**	**2**
0	1	2	3
1	2	3	6

Arreglo de Arreglos

- Dado que tuvieran a la matriz “a” definida como una variable global así:

```
int **a;
```

- Hagan el código que le dé las dimensiones de 2 x 3 como el ejemplo pasado.

Argumentos

Forma Básica

- La función de main también puede recibir argumentos:

  int main(int argc, char *argv[])

- **argc** es el número de argumentos recibidos
  - Siempre, por lo menos, recibe uno: el nombre del programa.
- **argv** es un arreglo de arreglo de caracteres que indican los strings que se pasaron al programa separados por espacios

Ejemplo

./programa argumento1 argume2

argc → 3

argv[0] → ./programa

argv[1] → argumento1

argv[2] → argume2

getopt

- Hay formas más sofisticadas de manejar argumentos
  - No requieren de que el usuario ponga los argumentos en el orden correcto.
  - Asigna valores por defecto.
- **Getopt** es una opción atractiva.
  - Pero no es esencial para nuestros propósitos.
  - Se los dejo de tarea opcional investigarlo.

Conversión de Datos

- Los datos son siempre entregados como strings (serie de caracteres).
- Si se necesita que se conviertan a tipos de variables numéricas, stdlib.h proporciona las siguientes funciones:

  atoi : de string a int

  atof : de string a float

Conversión Inversa

- sprintf es una función parecida a printf, pero en vez de escribir a pantalla, escribe a un string:

```
char *cadena[5];
sprintf(cadena, “%d”,10);
printf(“%s \n”, cadena);
```

Ya que estamos en el tema...

- printf y sprintf pueden manejar varias maneras de moldear los datos a un string:

  “%d”,10 → 10

  “%05d”,10 → 00010

  “%5d”,10 → 10 (rellena con espacios)

  “%f”, 2.519 → 2.519000 (por defecto 6 después)

  “%.2f”, 2.519 → 2.52

Problema

Problema

- Hacer un programa en C que:
  - Obtenga tres argumentos y verifique que son números enteros.
    - Si no, que los redondee al entero más cercano, y entregue en pantalla una advertencia de lo que se ha hecho.
  - Verifique que el tercer número sea múltiplo de ambos los dos primeros.
    - Dicha verificación se debe llevar a cabo en una función aparte de main.
  - Continuación en la siguiente diapositiva...

Problema Cont...

- Si no es múltiplo, no hacer nada.
- Si es múltiplo, crear una matriz que tenga:
  - Tantos renglones como el primer argumento.
  - Tantas columnas como el segundo argumento.
- Hacer una función que reciba como argumento el apuntador de la matriz, y sus dimensiones (cuantos renglones y cuantas columnas), y asigne el valor de cada celda con el siguiente valor:
  - El tercer argumento del programa, entre el número del renglón, y multiplicado por el número de la columna.