CC3301
Programación de Software de Sistemas
Profesor: Luis Mateu

- Typedef para punteros a funciones
- Motivación: función integral genérica
- Variables globales
- Scope
- Cast de punteros
- Punteros opacos
- Tipo estático vs. tipo dinámico
- Programación de la función integral genérica
- Conclusiones

Función integral genérica

- Implementación:

```
double integral(double (*pf)(double x),
                        double xi, double xf, int n) {
   double h= (xf-xi)/n;
   double sum= ( (*pf)(xi) + (*pf)(xf) ) / 2;
   for (int k= 1; k<n; k++)
      sum += (*pf)(xi + k*h);
   return sum * h;
}
```

- ¡Horrible!

```
double poli(double x) {     // Ejemplo de uso
   return 4.5*x*x-10*x+3.1;
}
int main( ) {
   printf("%f\n", integral(poli, 0.0, 10.0, 100));
   return 0;
}
```

Typedef para punteros a funciones

- ¿Qué pasa si agregamos typedef a la declaración de un puntero a una función?

```
typedef double (*Fun)(double x);
```

- Fun ya no es puntero a una función: es el tipo de los punteros a funciones que reciben un parámetro real y retornan un real
- El identificador *x* es solo un adorno opcional
- La función integral se puede reescribir:

```
double integral(Fun pf,
                double xi, double xf, int n) {
   double h= (xf-xi)/n;
   double sum= ( (*pf)(xi) + (*pf)(xf) ) / 2;
   for (int k= 1; k<n; k++)
      sum += (*pf)(xi + k*h);
   return sum * h;
}
```

Más integrales

- Se desea programar la función:

```
double   h(double a, double b, double c,
          double xi, double xf, int n);
```

- tal que:

$$h(a, b, c, xi, xf, n) \approx \int_{xi}^{xf} ax^2 + bx + c\, dx$$

  Aproximado con *n* trapecios

- ¿Se puede usar integral? ¿Hay manera de pasarle los valores de *a*, *b* y *c*?

- Solución incorrecta:

```
double poli2(double x) {
   return a*x*x+b*x+c;  // ✗
}
double h(double a, double b, double c,
                double xi, double xf, int n) {
   return integral(poli2, xi, xf, n);
}
```

- Problema: No compila
- Las variables *a*, *b* y *c* no son *visibles* en *poli2*
- Solución de parche: *variables globales*

Scope

- ***Definición***: *Alcance*, *visibilidad* o en inglés *scope* de un *identificador* de variable corresponde a la región del código fuente en donde ese identificador es conocido
- Usar un identificador fuera de su alcance es equivalente a usar un identificador que no está declarado
- La visibilidad de una *variable local* es el código que va inmediatamente después de su aparición en la declaración hasta que se cierra el bloque en que fue declarada
- El identificador de una variable local puede no ser visible, pero la variable sí podría estar viva
- Si se declara una variable con identificador *id*, pero ya existía otra variable declarada con el mismo identificador *id* en un bloque distinto, el identificador previo deja de ser visible hasta que termine la visibilidad del nuevo identificador
- Dado que una variable dinámica *no tiene identificador*, no tiene sentido hablar de su visibilidad

Variables globales

- ***Definición***: Si una variable se declara fuera de toda función, es una *variable global*
- Ejemplo: esta solución para la función *h* es correcta

```
double g_a, g_b, g_c;        // variables globales
double poli2(double x) {
   return g_a*x*x+g_b*x+g_c;  //
}
double h(double a, double b, double c,
                double xi, double xf, int n) {
   g_a= a; g_b= b; g_c= c;
   return integral(poli2, xi, xf, n);
}
```

- g_a, g_b y g_c son variables globales
- *Tiempo de vida*: se crean al inicio de la ejecución del programa y se destruyen cuando este termina
- *Scope*: el código que va inmediatamente después de su aparición en la declaración hasta que termina el archivo en el que fue declarada
- **Si es posible, evite usar variables globales**

Más sobre scope de variables globales

- Si se antepone el atributo *static*, como en:

  static double global_var;

  la variable no es visible desde otros archivos
- Si no lleva *static*, la variable sí puede ser visible desde otro archivo si en ese otro archivo se redeclara con el atributo extern:

  extern double global_var;
- Típicamente el *extern* se usa en archivos de encabezados (.*h*)

Inicialización de variables globales

- Las variables globales se puede inicializar pero solo con valores constantes
- Por ejemplo:

```
int n= 100;
double pi= 3.14159;
```

- Más precisamente se puede inicializar con una expresión que el compilador pueda calcular

```
int m= 2*100;
int o= 100/sizeof(int);
```

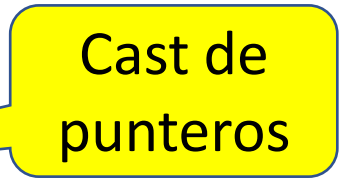
- No se puede:

```
double c= sin(1.0);   // ✗
double pi2= pi;       // ✗
```

Cast de punteros

- Un *cast de punteros* permite cambiar el tipo de la variable a la que apunta un puntero
- Ejemplo:

```
double pi= 3.14159;
double *ptr_double= &pi;
int *ptr_int= (int*)ptr_double;          //
```



- Si no se coloca el *cast*, el compilador reclama con un warning y coloca automáticamente el cast
- Las direcciones almacenadas en *ptr_int* y *ptr_double* son idénticas
- No se realiza ningún tipo de conversión de los datos
- ¿Cuánto vale *ptr_int?

- Respuesta: Ni cerca de 3, es basura en realidad
- Es ``legal´´ usar ptr_int[0] y ptr_int[1] pero rara vez útil

Tipo estático vs. tipo dinámico

- ***Definición***: el tipo estático de un puntero es el tipo declarado en tiempo de compilación
- Ejemplos:

  el tipo estático de *ptr_double* es double*

  el tipo estático de *ptr_int* es int*
- El tipo estático de un puntero no cambia jamás
- ***Definición***: Si en un instante dado la última variable almacenada en la dirección *dir* fue de tipo *T* y un puntero *ptr* contiene la dirección *dir*, el tipo dinámico de *ptr* en ese instante es *T**
- Ejemplo:

  el tipo dinámico de *ptr_int* es double *
- El tipo dinámico de un puntero sí cambia durante la ejecución
- Típicamente coincide con el tipo estático, pero puede diferir

Versión genérica de integral

- La función *integral* recibe un puntero *ptr* que incluirá como argumento en todas las invocaciones de **pf*
- El usuario de *integral* suministra en **ptr* cualquier otro parámetro que necesite para evaluar la función

```
typedef double (*Fun)(void *ptr, double x);
double integral(Fun pf, void *ptr,
                        double xi, double xf, int n) {
   double h= (xf-xi)/n;
   double sum= ( (*pf)(ptr, xi) + (*pf)(ptr, xf) ) / 2;
   for (int k= 1; k<n; k++)
      sum += (*pf)(ptr, xi + k*h);
   return sum * h;
}
```

- Un puntero de tipo void * es un *puntero opaco*: no se sabe nada acerca del tipo de la variable a la que apunta
- El compilador no permite usar * o -> con un puntero opaco
- ¡La función *malloc* retorna un puntero opaco!
- Cumple la misma función que *Object* en Java

Versión preferida para la función h

```
typedef struct { double a, b, c; } Abc;
double poli2(void *ptr, double x) {
   Abc *pabc= (Abc*)ptr;                          // Nota b)
   return pabc->a*x*x+pabc->b*x+pabc->c;
}
double h(double a, double b, double c,
                double xi, double xf, int n) {
   Abc abc= { a, b, c };
   return integral(poli2, &abc, xi, xf, n);      // Nota a)
}
```

Notas

a) Se permite asignar cualquier dirección a un puntero opaco, no es causa de errores

b) El cast es opcional cuando el puntero es void*

Abc *pabc= ptr; // ✔ solo porque *ptr* es *void **

Si el tipo dinámico de ptr no es Abc,*
el resultado de pabc->a no está especificado

Conclusiones

- Cuando el tipo dinámico de *p* no coincide con su tipo estático, es un error leer la variable apuntada con **p* o *p->*
- Durante la ejecución no se hace ningún chequeo para detectar este error
- Su ocurrencia se puede traducir en resultados incorrectos o *segmentation fault*
- ***Es completa responsabilidad de los programadores evitar este tipo de errores***
- Es la lógica del programa la que ayuda a evitar estos errores
- **Sí se puede escribir: ¡La asignación *p = … sirve para cambiar el tipo dinámicos de p!**
- Los *casts de punteros* junto a la aritmética de punteros le dan flexibilidad al lenguaje C, haciendo posible la programación orientada a objetos en C, la programación de colecciones genéricas, la programación **en C** de las funciones *malloc* y *free*, la programación de núcleos de sistemas operativos, etc.
- La *gran desventaja* es la ausencia del chequeo de tipos

Ejercicio desafiante

- Se desea programar la función:

  typedef double (***Fun2**) (double x, double y);

  double e(double xf, double yf, **Fun2 g**, int n);

$$e(xf, yf, g, n) \approx \int_0^{yf} \int_0^{xf} g(x, y)\, dx\, dy$$

Aproximado con *n* trapecios

- **¡Usar integral!**