Introducción al Cómputo

Más control de flujo

Operador condicional

- El operador condicional tiene la sintaxis:

```
cond ? expr1 : expr2;
```

Si **cond** es verdadero, evalúa y toma el resultado de **expr1**, si **cond** es falso evalúa y toma el resultado de **expr2**

```
if (y < z)
   x = y;
else
   x = z;
```

```
// equivalente a:
// x = (y < z) ? y : z;
```

Bloque switch

- Variación del bloque if para casos múltiples

```
switch(expr) {   // expr define el caso de entrada
    case Caso1:  // los casos son constantes
        Accion1_1;
        ...
        break;   // si no existiese, continuaría
    case Caso2:
        Accion2_1;
        ...
        break;   // si no existiese, continuaría
    ...
    default:     // puede no existir un default
        AccionD_1;
        ...
        break;
}
```

Funciones srand/rand

- Funciones que permiten la generación de números pseudo-aleatorios utilizando el algoritmo de congruencia lineal:

$$x_{n+1} = (a\, x_n + c) \bmod m$$

donde a, c y m son constantes convenientemente seleccionadas

`srand(seed)` => impone x_0

`rand()` => retorna el siguiente valor en la secuencia (en el rango **`[0,RAND_MAX]`**)

Ejemplo de srand/rand

Ejem5_1.cpp

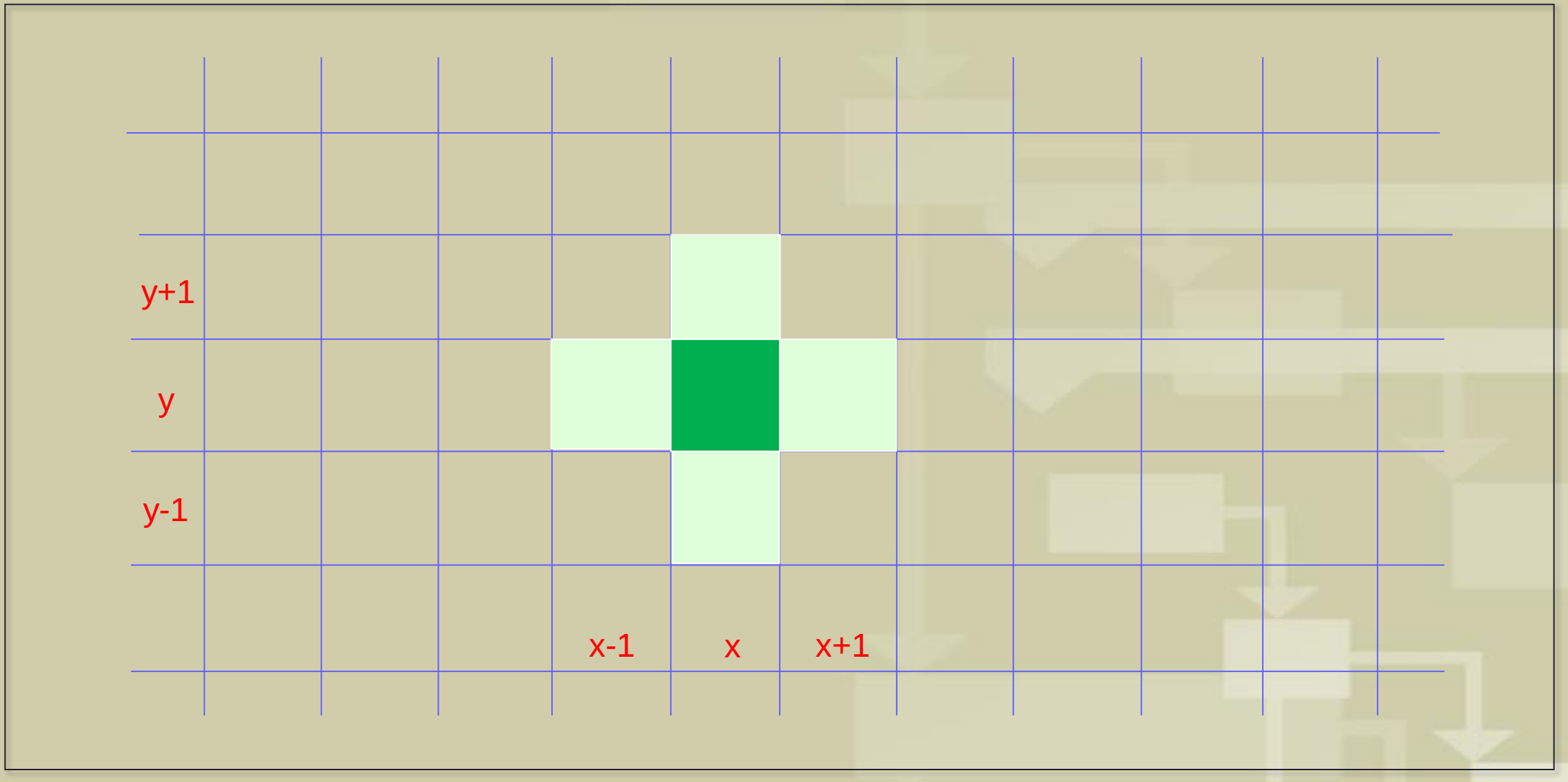
```
#include "icom_helpers.h"
// uso de srand/rand
int main()
{
    srand(32); // El argumento de srand define la secuencia (valor inicial).
               // Si se "semilla" el generador siempre con el mismo valor
               // se obtiene siempre la misma secuencia
    for(int i = 0; i < 10; ++i)
        cout << rand() << ' ';
    cout << '\n';

    srand(time(0)); // Si el argumento de srand depende del tiempo
                    // se obtienen secuencias distintas
    for(int i = 0; i < 10; ++i)
        cout << rand() << ' ';
    cout << '\n';

    return 0;
}
```

Ejemplo de srand/rand

Ejecución de Ejem5_1

```
$./Ejem5_1
143 23988 3067 18799 11241 6380 9660 4148 14533 30482
32679 13826 13787 22725 10009 31641 20885 24410 10317 21083
$
$
$./Ejem5_1
143 23988 3067 18799 11241 6380 9660 4148 14533 30482
101 14641 1333 9373 18430 30015 4799 18468 17717 24944
$
$
$./Ejem5_1
143 23988 3067 18799 11241 6380 9660 4148 14533 30482
199 9414 12970 10376 187 16745 32637 28954 21545 4342
$
$
```

Caminante al azar



Caminante al azar

Ejem5_2.cpp

```
#include "icom_helpers.h"
// simula el comportamiento de un caminante al azar. Approach medio holístico…

enum Direction {          // enumeración de las posibles direcciones (la última es auxiliar)
    LEFT, RIGHT, UP, DOWN, NONE, NUM_DIRS
};

struct Entity {           // Tipo definido por el usuario para representar a la entidad
    int x, y;             // representación: 2 enteros para representar la posición
    void move(Direction d) {      // método para mover la entidad en una dirección determinada
        switch(d) {
            case LEFT:  x--; break;
            case RIGHT: x++; break;
            case UP:    y++; break;
            case DOWN:  y--; break;
            case NONE:  break;
           default: break;  // si llega hasta acá debería disparar una excepcion
         }
    }
    void print() {        // método para imprimir la posición y distancia al origen
      cout << "X: " << x << " Y: " << y << " Dist.: " << sqrt(x*x+y*y) << '\n';
    }
};
```

Caminante al azar (cont)

```
const int NUM_PASOS = 10000;

int main() {
    Entity e = { 0, 0 };

    srand(time(0));           // Inicializa generador pseudo-aleatorio

    for(int contador = 0; contador < NUM_PASOS; contador++)
        e.move(Direction(rand() % NUM_DIRS));

    e.print();
    return 0;
}
```

`rand() % NUM_DIRS` da como resultado un entero.
Como el método move de Entity recibe una Direction, la expresión:
`Direction(rand() % NUM_DIRS)` construye una dirección a partir de un entero

Funciones

- Sientan la base para la reutilización de código.
- Definen los módulos constructivos de resolución de subproblemas en el paradigma de DIVIDE y CONQUISTA.
- Permiten producir programas que son más fáciles de escribir, leer, entender, depurar, modificar y mantener.
- Las funciones deben ser **declaradas** y **definidas**.

Funciones

- La ***declaración*** (también llamada ***prototipo)*** de una función consiste en indicar el nombre de la función, el tipo del valor retornado (**`void`** si no retorna nada) y la cantidad y tipo de los argumentos que deben ser suministrados al llamar a la función.

```
int main();
void exit(int);
double sqrt(double);
double pow(double, double);
vector<int> fibonnacci(int);
```

- Es muy importante para la validación o adecuación (cuando es posible) de argumentos:

```
double sr2 = sqrt(2);          // conversión
double sr3 = sqrt("tres");     // error
```

Funciones

➢ La ***definición*** de una función es una declaración seguida por el cuerpo de la función.

```
void printMessage() {
    cout << "Hello world!\n";
}

int main() {
    printMessage();
    printMessage();
    return 0;
}
```

Argumentos de funciones

- Los argumentos que recibe una función sirven para parametrizar su comportamiento (en general no resuelven un único problema, sino una familia de problemas).

```
void triangular(int n) {
    int triang = 0;
    for( int i = 1; i <= n; i++ )
        triang += i;
    cout << "triangular de " << n << " es " <<  triang << '\n';
}

int main() {
    triangular(10);
    triangular(25);
    return 0;
}
```

El diseño de **`triangular`** la hace poco reutilizable, para ver el resultado hay que ver la consola. Impide el uso del resultado en otros cálculos

Variables locales o automáticas

- Las variables definidas dentro de una función se conocen como variables locales o automáticas.
- Son creadas automáticamente cada vez que la función es activada.
- Estas variables son sólo accesibles desde la función que las definió.
- El valor inicial es asignado cada vez que se ejecuta la función.
- Variables locales no inicializadas tienen valores no controlados.
- Los argumentos de una función son variables automáticas que se inicializan con los valores que se les dieron al llamarse la función.

Valor de retorno de una función

- Las funciones que no hayan sido declaradas como **`void`** deben retornar un valor del tipo declarado.
- Las funciones declaradas **`void`** no pueden retornar un valor.
- El valor a retornar se indica con la sentencia **`return`**.
- Puede haber múltiples sentencias **`return`** en una función.

```
int  f1() { }              // error
void f2() { }              // ok
int  f3() { return 1; }    // ok
void f4() { return 1; }    // error
int  f5() { return; }      // error
void f6() { return; }      // ok

int fac(int n)  { return (n > 1) ? n * fac(n-1) : 1; }

int fac2(int n) {
    if( n > 1 )
       return n * fac2(n-1);
    return 1;
}
```

Valor de retorno de una función

```
int triangular(int n) {
    int triang = 0;
    for( int i = 1; i <= n; i++ )
        triang += i;
    return triang;
}

int main(void) {
    int result = triangular(10) + triangular(25);
    cout "Resultado: " << result << '\n';
    return 0;
}
```

Mejor diseño de la función, se limita al mecanismo (algoritmo) para calcular el triangular, la política de para qué se usará ese resultado es ajena a la función

Funciones internas a un UDT (métodos)

- Se pueden definir funciones internas a un tipo definido por el usuario.
- Estas funciones tienen acceso a los atributos del objeto sobre el que se activó el método.

```
struct Entity {
    int x, y;
    void move(Direction d) {
        switch(d) {
            case LEFT:  x--; break;
            case RIGHT: x++; break;
            //...
       }
    }
    //...
}
```

```
int main() {
    Entity e1 = {0,0};
    Entity e2 = {10,10};

    e1.move(LEFT);
    //...
    e2.move(TOP);
    //...
    return 0;
}
```

Dispositivos predefinidos de I/O

- Dispositivos de I/O: un programa tiene 3 dispositivos predefinidos: de entrada, de salida y de error.
- Por defecto:
  - Entrada: teclado
    - El objeto **`cin`** esta asociado al dispositivo standard de entrada
  - Salida: consola
    - El objeto **`cout`** esta asociado al dispositivo standard de salida
  - Error: consola
    - El objeto **`cerr`** esta asociado al dispositivo standard de errores

Redireccionamiento

- Los sistemas operativos proveen mecanismos de redireccionamiento:
  - Dispositivos de entrada (<)
  - Dispositivo de salida (>, >>)
  - Dispositivo de errores (2>, 2>>)

Redireccionamiento

Redirecciona la salida del programa al archivo “resultados.dat”

```
$ ./prog > resultados.dat
```

Redirecciona la entrada del programa desde archivo “datos.dat”

```
$ ./prog < datos.dat
```

Redirecciona entrada, salida y error

```
$ ./prog < input.dat > output.dat 2> errores.dat
```

Ejemplos de I/O

Ejem5_3.cpp

```
// copia entrada a salida
#include "icom_helpers.h"

int main() {
    int c;
    while((c = cin.get()) != EOF) {
        cout.put(c);
    }
}
```

Ejemplos de I/O

Ejem5_4.cpp

```
// Contador de caracteres
#include "icom_helpers.h"

int main() {
    int nc = 0;
    while(cin.get() != EOF)
        nc++;
    cout << "se leyeron "<< nc << " caracteres\n";
    return 0;
}
```

Ejemplos de I/O

Ejem5_5.cpp

```
// Contador de lineas
#include "icom_helpers.h"

int main() {
    int nl = 0;
    int c;
    while((c = cin.get()) != EOF)
        if(c == '\n')
            nl++;
    cout << "se leyeron "<< nl << " lineas\n";
    return 0;
}
```