Universidad Nacional del Litoral
Facultad de Ingeniería y Ciencias Hídricas
Departamento de Informática

PROGRAMACIÓN ORIENTADA A OBJETOS

Asignatura correspondiente al plan de estudios de la carrera de Ingeniería Informática

UNIDAD 1

Punteros

Ing. Pablo Novara

2018

UNIDAD 1

Punteros

Variables estáticas

Hemos aprendido en Fundamentos de Programación que una forma de almacenar un valor en memoria consiste en definir una variable del tipo adecuado. Definir una variable implica indicarle al compilador el tipo de dato exacto y un identificador (nombre) con el que haremos referencia a la misma. El compilador se encarga de garantizar la existencia de dicha variable cuando corresponda. Esto es, determina de alguna manera en qué lugar de la memoria se almacenará el valor mientras se ejecuten las acciones del ámbito de validez de la variable. Tomemos por ejemplo el siguiente programa para obtener las raíces reales de una ecuación cuadrática aplicando la resolvente:

```
int main() {
	float a,b,c;
	cin >> a >> b >> c;
	float disc = b*b-4*a*c;
	if (disc>=0) {
		float sqrt_d = sqrt(dist), denom = 2*a;
		float r1 = (-b+sqrt_d)/denom,
		      r2 = (-b-sqrt_d)/denom;
		cout << r1 << endl << r2 << endl;
	} else {
		cout << "Las raices no son reales!" << endl;
	}
	cout << "Que tenga un buen día" << endl;
}
```

En este ejemplo, se pueden identificar las variables *a*, *b*, *c*, *disc*, *sqrt_d*, *denom*, *r1* y *r2*. Todas son de tipo flotante, los cual le indica al compilador (entre otras cosas) que debe reservar 4 bytes para cada una de ellas. Sin embargo, no todas están disponibles durante toda la ejecución. Cada variable está declarada dentro de un scope o ámbito de validez. Las variables *a*, *b*, *c* y *disc* están disponibles desde que son creadas hasta la finalización del programa, por estar definidas en la función principal (main). Pero las variables *sqrt_d*, *denom*, *r1* y *r2* sólo existen dentro de la rama por verdadero del condicional *if*. Por ejemplo, si en una ejecución la condición del *if* resulta falsa, estas variables no se crearán ni destruirán. Por otro lado, aunque resulte verdadera, al llegar al último *cout*, las variables ya no estará accesibles, ya habrán sido destruidas.

Por el momento, crear una variable implica solamente asegurar un lugar en la memoria para guardar su valor, y destruirla implica que ese lugar ya no será utilizado para esa variable (por lo cual podría ser reutilizado luego para otra). En la unidad siguiente, cuando estudiemos los conceptos de clase y objeto veremos

que las acciones de crear y destruir una variable pueden ser mucho más complejas, pero por el momento nos enfocaremos solamente en la gestión de la memoria.

En este mecanismo para crear variables, el concepto de scope o ámbito de validez es de suma importancia, dado que determina su ciclo de vida. Y es información que se desprende del análisis del código fuente, sin requerir su ejecución. Por esto, se les llama variables estáticas, y también por esto es que el compilador puede encargarse de reservar y liberar la memoria necesaria para sus valores automática y transparentemente. Es decir, que con solo definir una variable ya podemos dar por sentado que tendrá un espacio de memoria asignado, y no necesitamos hacer nada para liberar ese espacio cuando la variable ya no se utilice, sino que el mecanismo de scopes le indica al compilador cuando debe liberarlo automáticamente.

Repasemos el ejemplo para explicitar cuando se crean y destruyen las variables del mismo:

```
int main() {
      float a,b,c; // se crean a, b y c
      cin >> a >> >> c;
      float disc = b*b-4*a*c; // se crea disc
      if (disc>=0) {
            float sqrt_d = sqrt(dist), // se crea sqrt_d
                  denom = 2*a;         // se crea denom
            float r1 = (-b+sqrt_d)/denom, // se crea r1
                  r2 = (-b-sqrt_d)/denom; // se crea r2
            cout << r1 << endl << r2 << endl;
      } // se destruyen sqrt_d, denom, r1 y r2
      else {
            cout << "Las raices no son reales!" << endl;
      }
      cout << "Que tenga un buen día" << endl;
} // se destruyen a, b, c y disc
```

Punteros

Ya sabemos que cada variable se corresponde con alguna posición en la memoria de la PC. Podemos obtener esa posición mediante el operador &:

```
int f;
cout << &f; // el & podría leerse como
            // "la dirección de memoria de…"
```

El ejemplo mostrará algo como *0x7ffe7119833c*[1]. Esto es un número que indica en qué posición de la memoria *empieza* la variable *f* (ya que un *int*, por ejemplo, ocupa en realidad 4 posiciones/bytes). Las posiciones se miden como si la memoria fuera un arreglo gigante, pero los números se muestran en formato hexadecimal (notación con base 16, en lugar de base 10 como estamos acostumbrados), de ahí el prefijo *0x* y la presencia de algunas letras (entre *'a'* y *'f'*) como dígitos del número de posición.

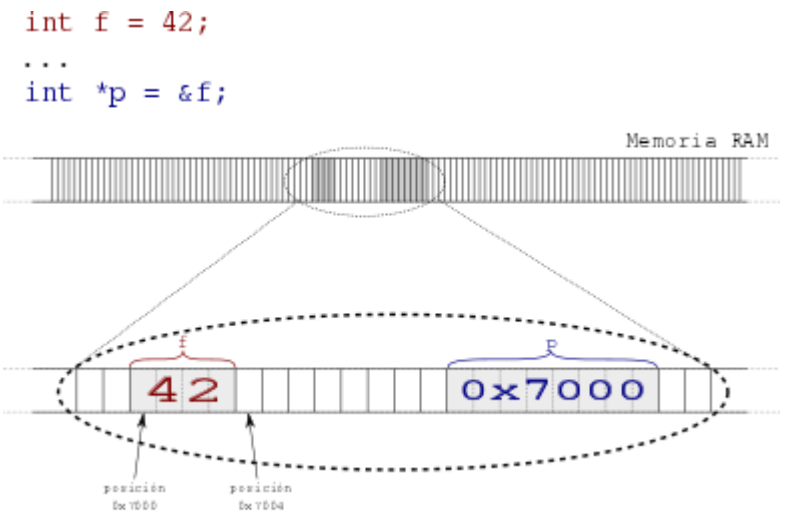
Esta posición, puede también almacenarse en una variable. Es decir, en otro lugar de la memoria podemos escribir ese número. A esta nueva variable se la denomina *puntero*.

Definición: Un puntero es una variable cuyo contenido es una dirección de memoria.

Supongamos que guardamos esa dirección en una variable *p*:

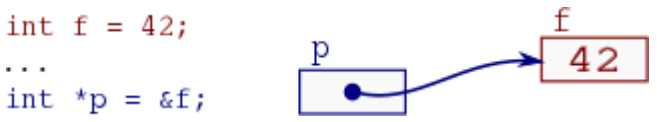
```
int f;
int *p = &f;
cout << p;
```

p contiene por valor, a la posición de memoria que le fue asignada a *f*. Se dice que *p* apunta a *f*. En C++, para declarar una variable de tipo puntero se debe anteponer el asterisco (*) al identificador de la variable. En el ejemplo, *p* (definida como *int**) es un puntero a entero. Es decir, puede contener una dirección de memoria, en la cual debería encontrarse un valor entero.



Gráficamente se suele representar de forma simplificada como sigue:

[1] No podemos predecir ni controlar la posición, sino que esta será elegida en conjunto entre el compilador y el sistema operativo.



Notar que el tipo de un puntero incluye al tipo de la variable apuntada. Todos los punteros guardan direcciones de memoria, por lo cual, en principio no es necesario saber a qué apuntaremos para saber cómo guardar ese valor. Pero, dado que más adelante intentaremos utilizar el valor apuntado, el compilador necesitará saber el tipo de dicho valor para saber qué operaciones serán válidas sobre ese valor. Por ejemplo, no podemos usar el valor apuntado por un puntero de tipo *string** para realizar una división (/), ya que el operador de división sólo puede aplicarse sobre tipos numéricos, y no sobre operandos del tipo *string*.

Finalmente, queda mostrar cómo operar sobre el valor apuntado por un puntero. Si disponemos de un puntero *p*, aplicándole el operador unario * obtenemos su valor:

```
int f = 0;
int *p = &f;
*p = 42; // el * podría leerse como "el dato apuntado por"
cout << f; // muestra 42
```

En este ejemplo, *p* apunta al valor de la variable *f*, por lo que **p* equivale a *f*. Es decir, el identificador *f* representa un entero *f* en alguna posición de la memoria determinada por el compilador. *p* contiene dicha posición, por lo tanto al modificar el dato apuntado por *p* estamos modificando el valor de la variable *f*. Se denomina *desreferencia* a la aplicación del operador * sobre un puntero.

Nota: hemos visto dos usos para cado uno de los símbolos *&* (declaración de un alias y obtención de la dirección de memoria de una variable), y * (declaración de un puntero, y desreferencia). La clave para saber cuándo significa una cosa y cuándo otra, está en diferenciar cuándo se está usando para especificar un tipo (por ejemplo, en la declaración de una variable nueva, o de un argumento en el prototipo de una función), en cuyo caso serán para declarar un alias o un puntero; y cuándo se están aplicando sobre variables o expresiones ya existentes, en cuyo caso serán para obtener una dirección de memoria o desreferenciar un puntero.

```
int *ptr = foo(); // en el tipo "int *", * indica puntero
int &alias = x; // en el tipo "int &", & indica alias

cout << *ptr; // aplicado al puntero ptr, * significa
              // "el dato apuntado por..."
cout << &x; // aplicado a la variable x, & significa
            // "la dirección de memoria de..."
```

Variables dinámicas

Habiendo introducido el concepto de puntero y la sintaxis relacionada, podemos ahora presentar un segundo mecanismo para crear y destruir variables, reservando y liberando respectivamente la memoria necesaria. Se basa en los operadores *new* y *delete*:

```
int *p = new int; // se reserva memoria para
                  // un nuevo valor entero
*p = 42;
cout << *p;
delete p; // se libera la memoria reservada
          // previamente con new
```

El operador *new* reserva un bloque de memoria para una nueva variable del tipo que se indique a continuación de dicho operador, y retorna la dirección donde comienza dicha memoria. Es decir, la expresión *new int* reserva memoria para un nuevo valor de tipo *int*, y retorna un puntero que apunta a ese nuevo valor.

A diferencia de lo que ocurría con las variables estáticas, la memoria reservada con *new* no se libera automáticamente cuando la ejecución sale el ámbito en donde se creó. En este caso, el programador es el responsable de decidir en qué momento liberar dicha memoria. Para ello se utiliza el operador *delete* seguido por un puntero apuntando al comienzo del bloque de memoria que se debe liberar.

Es importante recalcar que en un programa correctamente desarrollado, por cada new que se ejecuta, se ejecutará posteriormente un *delete*. Es un error reservar memoria con *new* y no liberarla luego con *delete*. En ese caso, la memoria quedará reservada aunque el programa ya no la utilice, y por ello no estará disponible para otras operaciones[2]. De esta forma, el consumo de memoria del programa crecerá más de lo necesario durante la ejecución, y en muchos casos indefinidamente hasta agotar la memoria.

```
void foo(int a, int b, int c) {
      int *sum = new int;
      *sum = a+b+c;
      return *sum;
} // error de lógica: la memoria de sum nunca será liberada
```

También es un error aplicar dos veces o más delete sobre una misma variable, o aplicar delete sobre una dirección de memoria que no fue generada por *new*.

Lo peligroso de estos errores es que normalmente no se manifiestan en la compilación, y muchas veces tampoco en la ejecución. Olvidar un delete no se notará a menos que el programa termine por agotar toda la memoria disponible. Realizar un delete sobre una dirección inválida ocasiona lo que se conoce como comportamiento indefinido. En este caso el programa puede generar un error

[2]Se denomina a este problema *memory-leak* (goteo de memoria).

que detenga la ejecución, continuar la ejecución con normalidad, o continuar y luego detenerse en otro punto diferente al que verdaderamente ocasionó el problema. Es por esto que se debe tener especial cuidado al utilizar memoria dinámica en C++[3].

El puntero nulo

Existe una dirección de memoria especial, que nunca podrá ser utilizada para una variable, y por lo tanto se utiliza para asignar a un puntero cuando el mismo aún no apunta a una dirección de memoria válida. Por ejemplo:

```
int *p;
cout << p << endl;
*p = 42;
cout << *p << endl;
```

El puntero *p* es una variable cuyo valor no ha sido inicializado, por lo que contendrá *basura*. Entonces, con **p* (el valor guardado en la dirección *basura*) estamos tratando de acceder a un lugar en la memoria que podría corresponder a otra variable del programa, o (más usualmente) no corresponderle a este programa. Aquí caemos nuevamente en comportamiento indefinido, por lo que el error podría no manifestarse. Si el puntero *p* se inicializa con la dirección nula, tenemos garantías de que el programa se detendrá con un error al intentar desreferenciar el puntero.

```
int *p = nullptr;
cout << p << endl; // muestra 0x0
cout << *p << endl; // el programa se detiene
```

El puntero nulo representa la dirección *0x0*, y se lo ingresa en el código mediante la constante *nullptr*[4].

Se debe tomar entonces como regla práctica lo siguiente: al definir un puntero, o bien se le debe asignar una dirección válida (con *new* o con el operador *&*), o bien se le debe asignar *nullptr*. Más aún, luego de liberar un bloque de memoria, también es conveniente asignarle *nullptr* al puntero para evitar volver a utilizar dicha dirección por error:

```
int *p = nullptr;
... // si usamos *p el programa se detendrá
```

[3] Veremos más adelante en esta materia que C++ ofrece mecanismos para reducir la complejidad de este problema, mediante abstracciones de más alto nivel que ocultan el verdadero uso de new y delete, y aseguran su correcta aplicación. Sin embargo, deberemos ser capaces de entender cómo funcionan por dentro estos mecanismos, y cómo programar nuestras propias abstracciones en caso de ser necesario.

[4] *nullptr* se introdujo en C++ en 2011. En versiones anteriores se utilizaba *NULL*. Y peor aún, se pueden encontrar código donde directamente se utiliza *0*. Si bien *0* y *NULL* siguen siendo válidos, es recomendable utilizar siempre *nullptr*.

```
p = new float;
... // podemos usar *p sin problemas
delete p;
p = nullptr;
... // si usamos *p el programa se detendrá
```

Finalmente, otra ventaja de asignar *nullptr* a un puntero, es que al depurar podremos reconocer fácilmente esa dirección (*0x0*). Pues en caso contrario será muy difícil determinar si una dirección dada es válida o *basura*.

Arreglos otra vez

En Fundamentos de Programación aprendimos a utilizar el tipo *vector<T>* para representar arreglos. Este tipo no es nativo del lenguaje, sino que se agrega como biblioteca (aunque esta biblioteca sea estándar). Es decir, está construido a partir de los elementos más básicos del lenguaje, de forma que hace uso de ellos pero esconde su complejidad y ofrece una interfaz maś simple. Aunque no se vea en el uso, por dentro, el tipo *vector* debe encargarse de alguna forma de gestionar la memoria necesaria para almacenar los elementos del arreglo.

En la unidad siguiente veremos cómo se construye un tipo de datos como *vector* para que estas operaciones de gestión de memoria queden “escondidas” y se ejecutan automáticamente al crear o destruir una variable de ese tipo. Ahora veremos cómo son los mecanismos explícitos que el tipo puede utilizar por dentro para gestionar la memoria del arreglo.

Arreglos estáticos

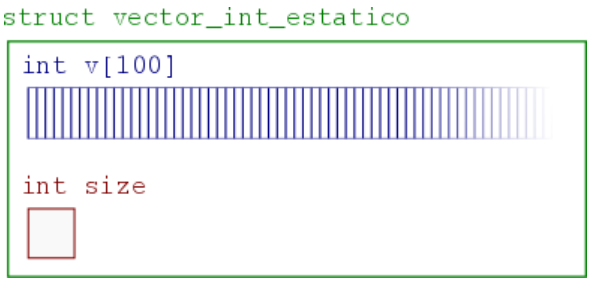
En C++, un arreglo estático se declara como sigue:

```
int v[100]; // arreglo de 100 enteros
```

Esto genera un arreglo denominado *v* con lugar para 100 datos de tipo *int*, que se corresponden con las posiciones de 0 a 99. A diferencia de lo que ocurría con vector, este arreglo tiene una limitación importante: el tamaño debe ser constante. Esto implica que no podemos utilizar una variable para indicar el tamaño, y que una vez generado el arreglo no se podrá modificar su tamaño.

Entonces, al crear un arreglo de esta forma, si no se conoce antes de la ejecución el tamaño exacto necesario, no queda otra opción que sobredimensionarlo, desperdiciando espacio en la mayoría de los casos. Lo que podría hacer un tipo como vector por dentro para ocultar este problema es guardar junto al arreglo un contador que indique cuántos elementos se están utilizando efectivamente:

Es importante notar que en este struct todos los datos del arreglo están dentro del struct[5]:



El tipo *vector* entonces, podría actualizar automáticamente la variable *size* al crear o redimensionar el vector. Esto no es lo que en realidad hace *vector*[6], ya que utiliza por dentro memoria dinámica.

Finalmente, vale destacar que si en una expresión utilizamos el nombre del arreglo sin incluir índices, el mismo representa un puntero cuya dirección es la dirección donde comienzan los datos del arreglo (donde comienza el primer elemento).

```
int v[100];
cout << &(v[0]) << "  " << v; // veremos 2 veces
                              // la misma dirección
```

Arreglos dinámicos

En C++, un arreglo dinámico se genera como sigue:

```
int *v = new int[100];
```

Al utilizar el operador *new[...]*, podemos indicar el tipo *int[100]* para reservar un bloque de memoria para contener un arreglo de *100* enteros, en lugar de un solo entero. Notar que el tipo de puntero en el cual se asigna la dirección de memoria que retorna *new* no varía respecto al primer ejemplo. Es decir, no incluye nada que indique que se trata de un arreglo, y no de un solo entero. Esto se debe a que, al igual que pasaba con los arreglos estáticos, la dirección es en realidad la del primer elemento.

Pero sí se debe tener en cuenta al momento de liberar la memoria, ya que para un arreglo se debe utilizar *delete[ ]* (con un par de corchetes vacíos) en lugar de *delete*:

---

[5] Este detalle será de especial interés cuando guardemos los datos de un arreglo en un archivo, en la unidad 5.

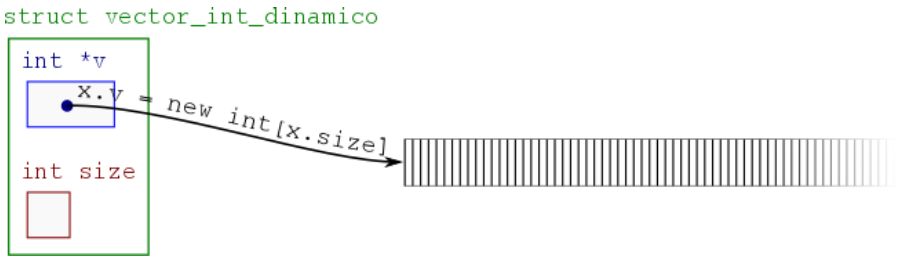
[6] Hay en la biblioteca estándar de C++ un tipo que efectivamente genera un arreglo con memoria estática, el tipo *std::array*.

```
delete [] v;
```

Una ventaja de este mecanismo es que el tamaño no necesita ser constante, sino que puede ser una variable cuyo valor se lee o calcula durante la ejecución. Una desventaja es que no existe forma de preguntar, dado un puntero que apunta a un arreglo, cuan grande es el arreglo. Es decir, el puntero indica el comienzo, pero no el final. Nuevamente, la solución es agrupar junto con el puntero un contador que guarde dicho valor:

```
struct vector_int_dinamico {
      int *p;
      int size;
}
int main() {
      // ejemplo de inicialización de un arreglo
      // dinámico de 100 elementos
      vector_int_dinamico v;
      v.size = 100;
      v.p = new int[v.size];
```

En este caso, los datos del arreglo no estarán dentro del struct, sino fuera del mismo. Dentro del struct solo tendremos el puntero a los mismos:



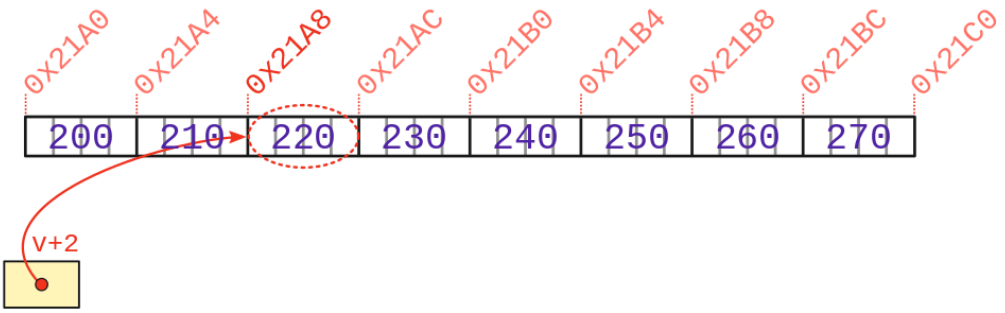
Este sí es, simplificando mucho, el mecanismo que utiliza el tipo vector que empezamos a ver en Fundamentos de Programación, solo que *vector* realiza los *new*s y *delete*s necesarios automáticamente, y se encarga además de mantener sincronizada la variable *size*.

Aritmética de punteros

Se pueden utilizar los operadores de suma y resta con operandos de tipo puntero y entero:

```
int *v = new int[10];
v[2] = 42;
cout << *( v+2 ); // muestra 42
delete []v;
```

Cuando a un puntero *v* se le suma o resta un entero *x* (2 en el ejemplo), se obtiene un nuevo puntero, cuya dirección es la que corresponde a un elemento ubicado *x* elementos a la derecha o izquierda de *v*. Es decir, si *v* apunta al primer elemento del arreglo, *v+1* apunta al segundo, *v+2* al tercero, etc.



Notar que *v+1* no es la posición de memoria que le sigue a *v*, ya que cada elemento del arreglo ocupa en realidad 4 posiciones de memoria (cada int ocupa 4 bytes). Por ejemplo:

```
int *v = …; // supongamos que v = 0x21A0;
cout << v << endl; // vemos 0x70000
cout << v+1 << endl ; // vemos 0x21A4
cout << v+2 << endl ; // vemos 0x21A8
cout << v+3 << endl ; // vemos 0x21AC
```

Entonces, las expresiones **(v+i)* y *v[i]* (donde *v* es un puntero e *i* un entero) son totalmente equivalentes. Análogamente, la resta de dos punteros da un entero con la cantidad de posiciones que hay entre una dirección de memoria y otra. El compilador determina cuántos bytes ocupa cada elemento a partir del tipo de puntero (si es *char** será 1, si es *int** serán 4, si es *double** serán 8, etc).

Esto puede utilizarse, por ejemplo, para calcular la posición de un elemento a partir de su dirección de memoria:

```
int * buscar(const int *v, int size, int valor) {
      // busca un valor en el arreglo v
      for (int i=0; i<size; ++i)
            // si lo encuentra retorna un ptr al encontrado
            if (v[i] == valor) return v+i;
      // si no lo encuentra retorna null
      return nullptr;
}
int main() {
      int v[100];
      … llenar el arreglo con datos …
      int *pos = buscar(v,100,42);
      if (pos==nullptr)
            cout << "El 42 no está en el arreglo.";
      else
            cout << "El 42 está en la pos: " << pos-v;
}
```

Arreglos multidimensionales

Es simple crear y utilizar arreglos estáticos multidimensionales:

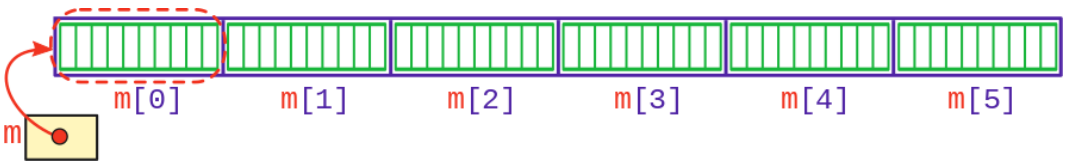
```
int m[3][2] = { { 0, 1, 2 }, { 3, 4, 5 } } ;
for (int i=0; i<3; ++i)
    for (int j=0; j<2; j++)
      cout << m[i][j];
```

Se debe notar que cada dimensión lleva su propio par de corchetes. Es decir, la posición de la fila 2, columna 1 es *[2][1]*, y no *[2,1]*. De igual forma, se pueden agregar más dimensiones (Por ejemplo, 4 dimensiones: *int m2[3][5][12][8];*).

En el caso de los arreglos dinámicos, la solución es más complicada, ya que el tipo de puntero para un arreglo multidimensional es diferente:

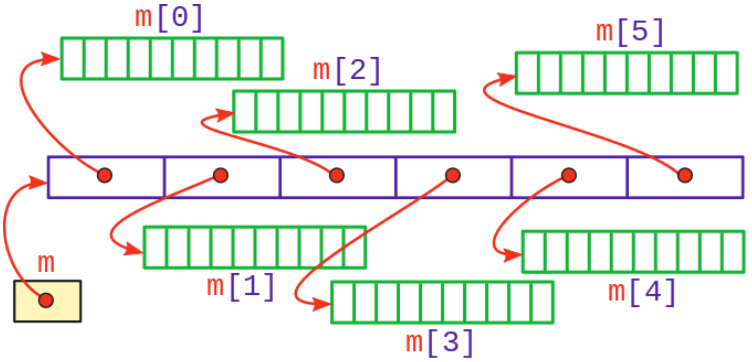
```
int (*m)[10] = new int[n][10];
… usar la matriz con m[i][j] como siempre …
delete [] m;
```

En este ejemplo de arreglo bi-dimensional, una de las dimensiones puede ser variable (*n*) mientras que la otra deberá ser constante (*10*) por formar parte del tipo de puntero (que es *int(*)[10]*). El puntero *m* aquí apunta a *toda* la primer fila de la matriz (que puede verse como un arreglo de 10 elementos), y no solo al primer elemento (aunque ambas direcciones son la misma).



La alternativa es crear un arreglo de punteros, y asignarle a cada uno una fila de la matriz:

En este caso, tendremos disponible también las *mismas* (en cantidad y numeración) posiciones y podremos acceder a ellas con *m[i][j]*, pero en memoria todas estas posiciones no estaran contiguas y ordenadas por fila. Cada fila podría estar arbitrariamente en cualquier posición, y el *arreglo de punteros* al que apunta *m* servirá de *índice* para saber efectivamente dónde está cada una de ellas.



Esta técnica permite que ambos tamaños sean variables, pero complica notablemente la gestión de la memoria (reserva y liberación), por lo que usualmente dicho código estará oculto en el interior de tipos de datos avanzados similares a *vector* (como el tipo *matrix* que utilizamos en Fundamentos de Progamación). Por esto, no será necesario memorizar los detalles de sintaxis de esta sección, ya que casi nunca deberemos recurrir a ellos. Pero un buen programador debe ser consciente de que esto es lo que hay detrás de ciertos tipos de datos, y saber dónde buscar si en alguna ocasión le toca desarrollar uno propio.

Ventajas/desventajas del uso de memoria dinámica

El uso de memoria dinámica implica mayor complejidad y responsabilidad para el programador. Además, por el funcionamiento del conjunto compilador+sistema operativo, es mucho más lento obtener memoria dinámicamente que estáticamente[7]. Entonces, ¿cuándo será conveniente utilizar new y delete?

Existen tres problemas que se resuelven solo con el uso de memoria dinámica:

[7] La reserva de memoria estática se resuelve durante la compilación, mientras que la de memoria dinámica durante la ejecución. Casi nunca notaremos esta diferencia, a menos que en un programa necesitemos crear y destruir cientos de miles de variables por segundo; pero si no se tiene cuidado este podría ser el caso en muchos tipos de sistemas tales como videojuegos o cualquier otro software de simulación y/o visualización en tiempo real.

1. Cuando **queremos un ciclo de vida diferente** al determinado por el scope (alcance) que la contiene. Por ejemplo, si queremos que un función reserve un bloque de memoria y lo retorne al programa cliente. Dado que todas las variables locales de la función se destruyen cuando esta finaliza, la única forma de garantizar que una posición de memoria sobreviva para llegar directamente al programa cliente es utilizando memoria dinámica.

```
int* crea_vector(int n, int vmin, int vmax) {
      int *v = new int[n];
      for (int i=0; i<n; i++)
            v[i] = rand()%(vmax-vmin+1) + vmin;
      return v;
}
void muestra_vector(int *v, int n) {
      for (int i=0; i<n; ++i)
            cout << i << " = " << v[i] << "  ";
}
void destruye_vector(int *v) {
      delete [] v;
}
int main() {
      int n; cin >> n;
      int *v = crea_vector(n, 0, 100);
      muestra_vector(v, n);
      destruye_vector(v);
}
```

   En este ejemplo, la memoria para el arreglo es creada en una función (*crea_vector*), utilizada en otra (*muestra_vector*), y destruida en una tercera (*destruye_vector*), por lo que su ciclo de vida no está determinado por el scope de ninguna de las tres.

2. Cuando **necesitamos mucha memoria**. Un programa divide la memoria disponible en secciones, y utiliza cada sección de forma diferente. Por ejemplo, hay una sección en la que se encontrará el código objeto del programa (las instrucciones). Las variables estáticas y dinámicas serán alojadas en diferentes secciones[8]. Las estáticas en una sección denominada stack (pila), y las dinámicas en una denominada heap (montículo). El tamaño del stack es fijado por el sistema operativo, y será generalmente pequeño, mientras la mayor parte de la memoria del sistema se asignará al heap. Por ejemplo, en un sistema GNU/Linux típico el stack será de 8MB, y en un sistema Windows generalmente de 1MB. El heap, en cambio, representa casi todo el resto de la memoria disponible (miles de MB). Por esto, en el stack solo debemos poner estructuras relativamente pequeñas, mientras que en el heap podemos alojar estructuras de datos muchísimo más grandes.

[8] Esta diferencia es la que hace que un mecanismo sea más rápido que el otro, pues en cada sección la memoria disponible se administra de forma diferente.

```
int main(int argc, char *argv[]) {
    const long long n = 1e9; // 10M
    int v[n]; // error en tiempo de ejecución
    cout << "Hola Mundo";
}
```

En este ejemplo, el vector requiere 40MB (10M elementos * 4 bytes cada uno), y esto es mucho más grande que el stack en cualquier sistema operativo, por lo que hará que el programa se detenga y muestre un error al intentar generar el arreglo. La siguiente versión funciona sin problemas:

```
int main(int argc, char *argv[]) {
    const long long n = 1e9; // 10M
    int *v = new int[n];
    cout << "Hola Mundo";
    delete [] v;
}
```

3. Cuando **no sabemos el tamaño de los datos** en tiempo de compilación. Si la creación de una variable, o su tipo o tamaño dependen de datos que se generan durante la ejecución (por ejemplo, si el tamaño de un arreglo está dado por un entero que ingresa el usuario), entonces no es posible utilizar memoria estática, ya que para que su reserva pueda planificarse en tiempo de compilación, el compilador necesita conocer en ese momento el tipo[9] y tamaño exacto.

```
int main(int argc, char *argv[]) {
    int n;
    cin >> n;
    int v[n]; // error al compilar, n debe ser const
    cout << "Hola Mundo";
    ...
}
```

Este ejemplo no compila[10], ya que el tamaño de un vector estático no puede ser variable. Si se califica a *n* con *const* desaparecerá el error en la declaración de *v*, pero ya no podremos leer *n* con *cin*. La única solución viable es la siguiente:

```
int main(int argc, char *argv[]) {
    int n;
    cin >> n;
    int *v = new[n];
    cout << "Hola Mundo";
    ...
    delete [] v;
}
```

[9] No hemos visto aún ejemplos donde se pueda variar el tipo, y no disponemos aún de preconceptos necesarios para introducirlos, pero los veremos en la unidad 3.

[10] Muchos de los compiladores actuales compilarán sin problemas este ejemplo porque en realidad transformarán el código internamente para utilizar el mecanismo de *new* y *delete* como en el segundo ejemplo en estos casos; por lo que no será realmente un arreglo estático aunque lo parezca.

Punteros y funciones

Punteros como argumentos de funciones

Ahora que sabemos cómo obtener y utilizar la dirección de memoria de una variable, podemos preguntarnos si es conveniente o necesario aplicar este conocimiento al diseñar funciones. Analicemos el siguiente ejemplo:

```
void incrementar_v1(int &x) { ++x; }
void incrementar_v2(int *p) { ++(*p); }
int main() {
      int cont = 0;
      cout << cont ; // muestra 0
      incrementar_v1 ( cont );
      cout << cont; // muestra 1
      incrementar_v2 ( &cont );
      cout << cont; // muestra 2
}
```

Las funciones *incrementar_v1* e *incrementar_v2* realizan exactamente la misma operación (de hecho, el compilador generará exactamente el mismo código objeto para ambas): incrementar en 1 una variable de tipo *int*. Son dos formas de implementar un pasaje por referencia. ¿Qué implica optar por una u otra en un programa? Además de la comodidad (subjetiva) de cada sintaxis, hay dos argumentos, uno a favor de cada una.

1. A favor de *incrementar_v2* (que utiliza punteros), tenemos que para invocarla desde el main se requiere utilizar el operador & para obtener la dirección de memoria del contador. Esto puede parecer una incomodidad, pero deja en claro (explícito) que hay un pasaje por referencia. En el caso de *incrementar_v1* (que utiliza alias), la llamada se ve exactamente igual que la de una función que recibe argumentos por copia.

2. En contra de *incrementar_v2* (que utiliza punteros), se tiene que a esta función se le puede pasar un puntero inválido como *nullptr*, y se compilará igual sin errores, dejando el problema para la ejecución. En este sentido, *incrementar_v1* (que utiliza alias) es menos propensa a errores, ya que es mucho más difícil obtener un argumento inválido que pase la compilación sin errores.

Como se observó en la unidad de Fundamentos de Programación donde se introdujo el concepto de función y se mencionaron lineamientos para el diseño de prototipos, en general se debe buscar evitar diseñar funciones donde se utilice la referencia para devolver un resultado. Por esta razón, el punto nro 2, en contra de la versión que utiliza punteros, tendrá mayor peso que el punto nro 1. Es decir, no deberíamos utilizar punteros para realizar un pasaje por referencia, a menos que sea la intención admitir al puntero nulo. En este último caso, el uso de punteros le permite al programa cliente omitir un argumento:

```
tuple<float,float,bool>
resolvente(float a, float b, float c, float *disc=nullptr) {
      float d, denom = 2*a;
      if (disc==nullptr) d = b*b-4*a*c;
      else               d = *disc;
      if (d<0) { // raices imaginarias
            float preal = -b/denom;
            float pimag = sqrt(-d)/denom;
            return make_tuple(preal,pimag,false);
      } else { // raices reales
            float sqrt_d = sqrt(d);
            float r1 = (-b+sqrt_d)/denom;
            float r2 = (-b-sqrt_d)/denom;
            return make_tuple(r1,r2,true);
      }
}
```

En este ejemplo, el argumento *disc* es opcional. Si en el programa cliente ya se tiene el discriminante calculado por algún paso previo a la llamada a esta función, se le puede pasar como argumento para evitar que la función deba volver a calcularlo. Si no se tiene, se pasa *nullptr*, y la función lo calcula a partir de los coeficientes *a*, *b* y *c*. Un valor por defecto para un *disc* de tipo *float* (no puntero) no podría ser suficiente en este caso, ya que cualquier valor real podría ser efectivamente un valor del discriminante, y entonces no habría forma de diferenciar el caso en que no se pasa argumento alguno.

Entonces, puede ser una buena práctica recibir un argumento de tipo puntero cuando “no hay argumento” (*nullptr*) sea una opción válida para la lógica de la función.

Punteros como valores de retorno en funciones

Algo que en general siempre se debe tratar de evitar es generar funciones que solo reserven o liberen memoria, pero no ambas. Es decir, funciones que tengan new, pero no delete o delete pero no new, aunque no sea un memory-leak:

```
int *foo(int *v1, int *v2, int n) {
      int *p = new int[n];
      …
      return p;
}
```

En esta función se reserva memoria, y se le delega al programa cliente la responsabilidad de liberarla. Esto en general no es buena idea, ya que si el cliente no analiza en detalle la implementación, y lee solo su prototipo (caso maś habitual cuando es una función de una biblioteca desarrollada por otro programador), entonces el problema no es evidente. Es decir, a partir del prototipo no podemos determinar si el puntero que retorna apunta a algún valor que ingresó como parte de los argumentos (por ejemplo, a un elemento de v1), o

apunta a una variable o a un arreglo nuevo. Entonces, la responsabilidad adquirida luego de la llamada a la función, la de aplicar delete o delete[] (tampoco está claro cual corresponde), no es evidente, lo que llevará a un uso incorrecto con mayor frecuencia.

Por esto, se puede tomar como convención que las funciones solo retornan punteros a valores contenidos en los argumentos, y nunca cuando reservan memoria. Para el segundo caso tendremos tipos de datos especiales, como vector.

Otros tipos de punteros

Punteros a structs

Un puntero puede apuntar a un struct:

```
struct Alumno {
string nombre, apellido;
      int dni;
      vector<int> notas;
}
int main() {
      Alumno *a = new Alumno;
      (*a).nombre = "Juan";
      (*a).apellido = "Perez";
      ....
}
```

No hay diferencias conceptuales entre un puntero a un struct o un puntero a un int, pero sí hay un agregado a nivel de sintaxis que suele utilizarse para simplificar la notación. En el ejemplo, *a* es "puntero a Alumno", entonces debemos aplicar el operador de desreferencia (*) para obtener un "Alumno". Como Alumno es un struct, utilizamos el punto(.) para acceder a sus miembros individualmente. Los paréntesis se incluyen para clarificar la precedencia entre los operadores . y *. Existe una forma alternativa para acceder a un miembro de un struct al que referenciamos mediante un puntero:

```
int main() {
      Alumno *a = new Alumno;
      a->nombre = "Juan";
      a->apellido = "Perez";
      ....
}
```

El operador -> desreferencia el puntero y accede a un miembro. Es decir, "*(*x).foo*" es totalmente equivalente a "*x->foo*". En general, se prefiere la segunda sintaxis.

Punteros a funciones

Se pueden declarar y manipular punteros a funciones. Esto es, punteros que en lugar de apuntar a datos, apuntan a algoritmos:

```
bool compara_por_dni(Alumno a1, Alumno a2) {
      return a1.dni<a2.dni;
}
bool compara_por_nombre(Alumno a1, Alumno a2) {
      return a1.nombre<a2.nombre;
}
int main() {
      Alumno a[100];
      …
      if (…) sort(a,a+100,compara_por_dni);
      else   sort(a,a+100,compara_por_nombre);
      …
}
```

En este ejemplo hay dos funciones que sirven para determinar si un *Alumno* va antes que otro, según dos criterios diferentes. La función *sort* (estándar de C++, incluida en la cabecera <*algorithm*>) ordena un arreglo. Además de recibir punteros indicando los extremos del arreglo, puede recibir un tercer puntero indican con qué función se determina el orden relativo de dos alumnos. Así, la misma implementación de la función *sort* se puede utilizar para ordenar poor diferentes criterios.

Un puntero a una de estas funciones de comparación se declararía como: *bool (*p)(Alumno, Alumno)*. Es decir, como un prototipo pero con el nombre entre paréntesis y con un *. Luego, si se asigna un dirección válida a *p*, se puede invocar a la funciń que apunte como si *p* fuera la propia función: *p(a1,a2)*.
No analizaremos más detalles de la sintaxis específica de punteros a funciones, ya que en la práctica esta sintaxis no se utiliza directamente, sino que dichos punteros se utilizan a través de *templates*, tema que desarrollaremos más adelante en la unidad 6.

Punteros void

Por último, a los punteros declarados como "*void**" se les puede asignar cualquier dirección de memoria, proveniente de otros tipos de punteros. A estos punteros no se los puede desreferenciar ni utilizar en operaciones como *delete*, ya que al convertir un puntero cualquiera a *void** se pierde la información del tipo de dato al que apunta, información que es necesaria para la mayoría de las operaciones sobre punteros. Su uso es entonces limitado y/o poco conveniente, y dado que veremos más adelante alternativas mejores para esos pocos

escenarios (nuevamente *templates*, en la unidad 6), tampoco perderemos tiempo analizando ejemplos de uso para este tipo de punteros.

¿Qué sigue?

Como se remarcó en la mayoría de los casos, la gestión manual de memoria se utiliza cada vez menos de forma directa. Lo que se tiende a utilizar en la práctica son tipos de datos especiales que encapsulan, automatizan y esconden estas operaciones, como en los tipos *vector, matrix* y *string* que ya hemos utilizado anteriormente. La biblioteca estándar de C++ incorpora muchos tipos de datos auxiliares para tal fin, varios de los cuales serán estudiados en este curso[11]. Sin embargo, no es tan infrecuente tener que construir uno de estos tipos de datos, o necesitar conocer el mecanismo que hay detrás de ellos para realizar ciertas operaciones con los mismos, o para aprovechar mejor sus capacidades.

Nos alcanzará entonces, para casi todas las aplicaciones, con tener presentes todos los conceptos relacionados a la gestión de memoria, pero solo un subconjunto muy básico de la sintaxis específica relacionada al uso de punteros.

[11] En este curso en particular no tendremos tiempo de analizar en detalle los denominados *smart-pointers* (punteros inteligentes), pero sobre el final del mismo habremos aprendido todos los preconceptos necesarios para entender mucho más fácilmente su utilidad y funcionamiento. Cuando hayamos avanzado y desarrollado dichos preconceptos, compartiremos material adicional sobre este tema.