Tipos de Datos Recursivos

Josefina Sierra Santibáñez

15 de mayo de 2018

Introducción

La **recursividad** no sólo **se puede aplicar** a la definición de procedimientos (i.e. funciones o acciones), sino también **a la definición de estructuras de datos**.

- Los elementos de la estructura de datos se distribuyen en un conjunto de items conectados unos con otros.
- Los items se denominan **nodos**.
- Cada nodo contiene un elemento de la estructura de datos y al menos un enlace a otro nodo del mismo tipo (aquí es donde se usa la **recursividad**).
- Se define un valor nulo para cualquier tipo de nodo, que no contiene ningún elemento y sirve para indicar el final de la estructura (i.e. el **caso directo** de la recursividad).

Introducción

Las estructuras de datos formadas por **nodos enlazados** tienen dos propiedades muy útiles.

- No es necesario conocer a priori el número máximo de elementos de la estructura para reservar la memoria necesaria para almacenarla. **Se puede asignar memoria para nuevos nodos de manera dinámica**, a medida que se van añadiendo elementos a la estructura.
- Se pueden **insertar y borrar elementos** en cualquier punto **sin necesidad de mover los demás elementos** (esto no ocurre en un vector). Basta con **modificar algunos enlaces entre nodos**.

Para definir los nodos de estas estructuras utilizaremos

- el **constructor de tipos** `struct`, que permite representar tuplas de datos heterogéneos.
- el **constructor de tipos** `puntero`, que utilizaremos para implementar los **enlaces entre los nodos**.

Punteros y Gestión Dinámica de la Memoria

Consideramos que todo objeto de un programa está almacenado en la memoria del ordenador donde se ejecuta dicho programa, y que ocupa una serie de celdas de memoria contiguas o no.

Un puntero de un cierto tipo **puede contener la dirección de la primera celda de memoria de un objeto de dicho tipo**.

- Dado un tipo `T1`, el operador * permite definir el tipo **puntero a T1** del modo siguiente **T1***

- Cuando **una variable de tipo puntero a T1 apunta a un objeto**, se puede acceder a dicho objeto utilizando la expresión ***x**

```
struct T1 {
  int camp1;
  bool camp2;
};
T1 tup;      // variable de tipo T1
T1* x;       // variable de tipo puntero a T1
x = &tup;    // x apunta al objeto tup
(*x).camp2 = false;// equivale a x->camp2=false;
    // mismo efecto que tup.camp2 = false;
```

Punteros y Gestión Dinámica de la Memoria

Cuando se declara una variable de tipo puntero (e.g. `T1* x;`) dicha variable no tiene un valor definido. Si se intenta consultar su valor se produce un error. Para inicializar dicha variable se puede

- hacer que apunte a un objeto de tipo T1 ya existente,
- reservar memoria para que apunte a un objeto nuevo de T1, o
- asignarle el valor `nullptr`.

```
T1* z = &tup; // Asigna a z la direccion de tup.
              // Ahora z apunta al objeto tup.
T1* v = new T1; // Reserva memoria para un nuevo
// objeto de tipo T1 y hace que v apunte el.
v->camp1 = false;
T1* y = v; // y apunta al mismo objeto que v
v = nullptr; // v no apunta a ningun sitio
v->camp2 = 6; // produce un error
y->camp2 = 6;//y sigue apuntando al nuevo objeto
delete y; // libera la memoria del objeto al que
// apunta y. Solo para objetos creados por new.
```

Punteros y Cadenas de Nodos Simplemente Enlazados

Una cadena de nodos simplemente enlazados es una **cadena construida utilizando punteros**. Las cadenas de este tipo no tienen un tamaño fijo, **su tamaño puede crecer o disminuir** durante la ejecución.

Una cadena de nodos simplemente enlazados contiene un conjunto **nodos** (que dibujamos mediante cajas) conectados a través de **punteros** (que dibujamos mediante flechas).

Cada nodo contiene un **elemento** en el campo dato y un **puntero** en el campo enlace que puede apuntar a otros nodos del mismo tipo.

```
struct Nodo {
  int dato;
  Nodo* enlace; // uso de recursividad
};
// El puntero head representa la cadena
Nodo* head = nullptr; // Es una cadena vacia
head = new Nodo; // Extendemos con nodo nuevo
head->dato = 2;
head->enlace = nullptr; // terminamos la cadena
```

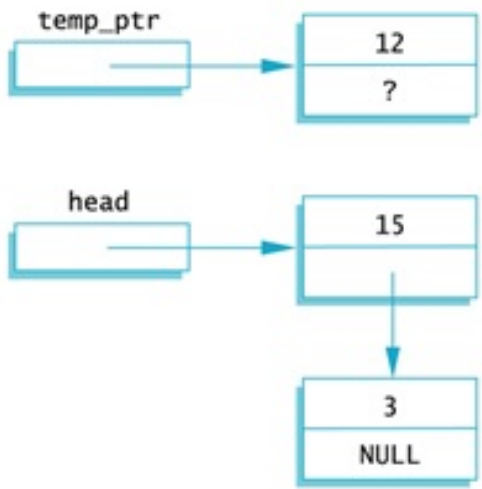
Punteros y Cadenas de Nodos Simplemente Enlazados

Construcción de una cadena de nodos simplemente enlazados. El primer nodo contiene el dato 12, el segundo 15 y el último 3.

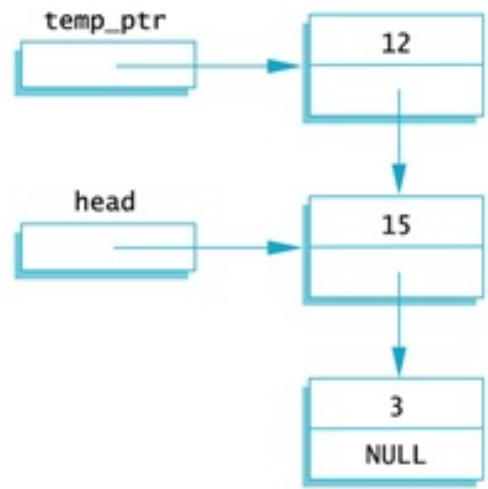
```
struct Nodo {
  int dato;
  Nodo* enlace;
};
Nodo* head = nullptr;
head = new Nodo;
head->dato = 12;
head->enlace = new Nodo;
(head->enlace)->dato = 15;
(head->enlace)->enlace = new Nodo;
((head->enlace)->enlace)->dato = 3;
((head->enlace)->enlace)->enlace = nullptr;
```

La variable de tipo puntero a Nodo **"head"** actua como el **nombre de la cadena** de nodos enlazados. Al contener la dirección del primer nodo de la cadena, posibilita el acceso a los demás nodos.

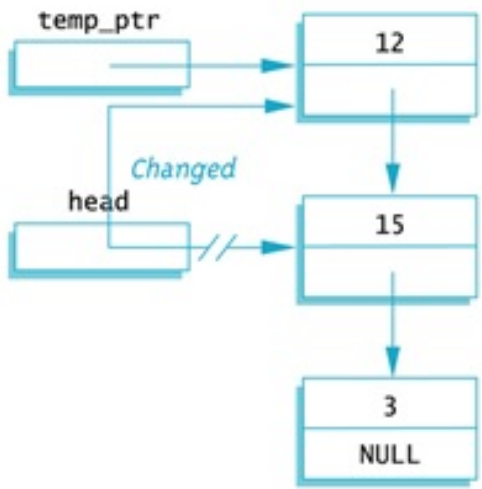
1. Set up new node



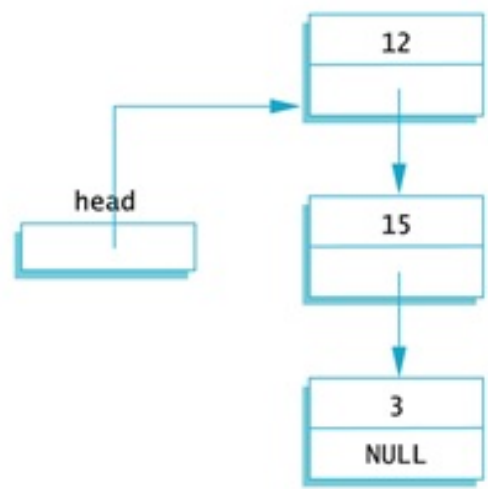
2. temp_ptr->link = head;



3. head = temp_ptr;



4. After function call



Punteros y Cadenas de Nodos Simplemente Enlazados

- La caja con la etiqueta **"head"** no es un nodo sino una **variable de tipo puntero** que puede apuntar **a** un objeto de tipo **Nodo**.
- Para modificar el valor del dato almacenado en el primer nodo de la cadena de nodos enlazados de la figura se puede utilizar la instrucción `head->dato = 3;`
- El puntero del último nodo de la figura contiene el valor `nullptr`. En C++ la **constante nullptr** se usa para **marcar el final de una cadena de nodos enlazados**.
- La constante `nullptr` se puede asignar a una variable puntero de cualquier tipo.

Cadenas de Nodos Simplemente Enlazados

Una cadena de nodos simplemente enlazados es una cadena de nodos en la que cada nodo tiene un campo que es un puntero que apunta al siguiente nodo de la cadena.

El primer nodo de una cadena de nodos enlazados se conoce como **la cabeza** de la cadena.

El último nodo no tiene un nombre especial, pero tiene una propiedad especial: el valor de **su campo de tipo puntero es nullptr**.

En el siguiente ejemplo creamos una cadena **"head"** con un solo nodo.

```
struct Nodo {
  int dato;
  Nodo* enlace;
};
Nodo* head = new Nodo;
head->dato = 3;
head->enlace = nullptr;
```

Insertar un Nodo en la Cabeza de una Cadena

El primer argumento de la siguiente acción es una variable de tipo puntero a Nodo que apunta a la cabeza de una cadena de nodos simplemente enlazados. Este parámetro se pasa por referencia. El segundo argumento es el valor que queremos almacenar en el nuevo nodo.

```
void insert_cab(Nodo*& head, int v)
/*Pre: head apunta a la cabeza de una cadena
de nodos enlazados */
/*Post: Se ha insertado un nuevo nodo que
contiene el valor v en la cabeza de la cadena*/
```

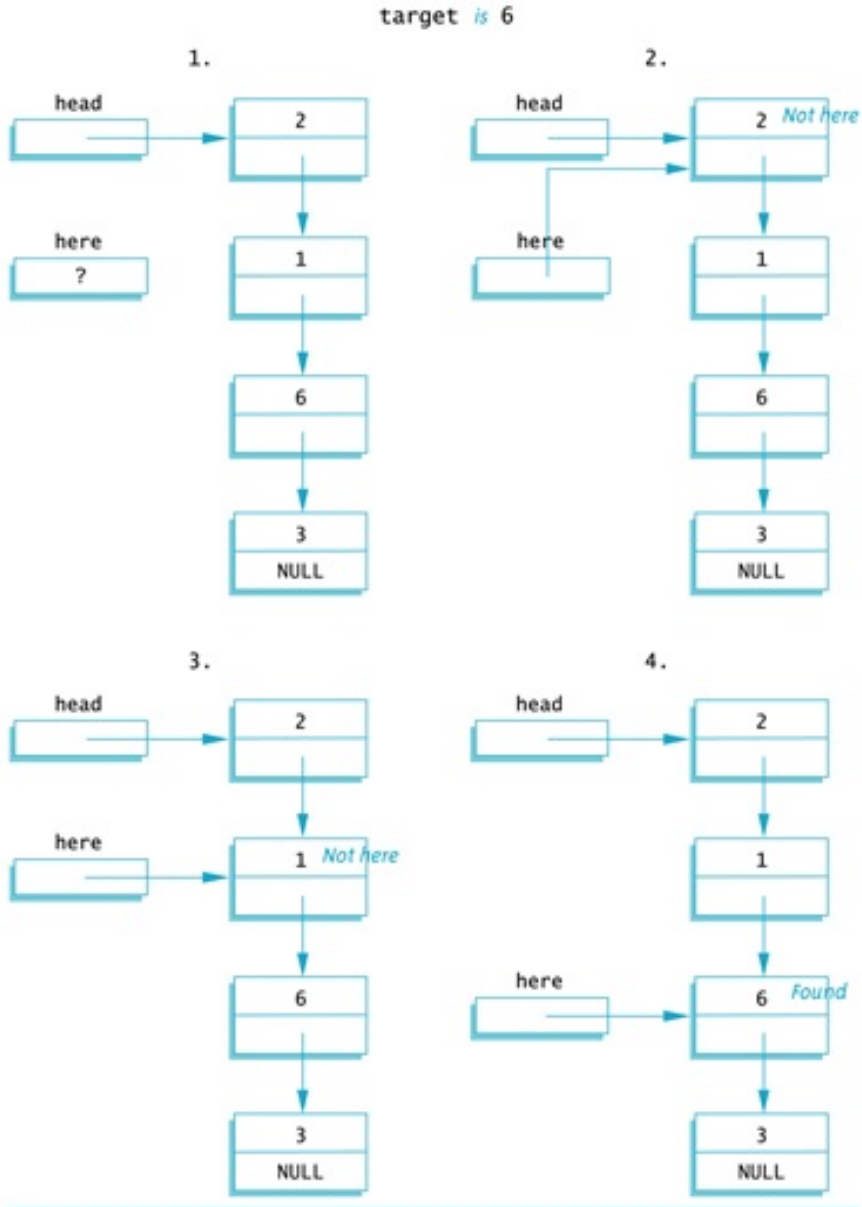
Siempre debe haber una variable de tipo puntero a nodo (e.g. **head**) que apunte a la cabeza de una cadena. Cuando se define una función o acción que tiene como argumento la cadena de nodos enlazados se usa esta variable como **argumento para representala**.

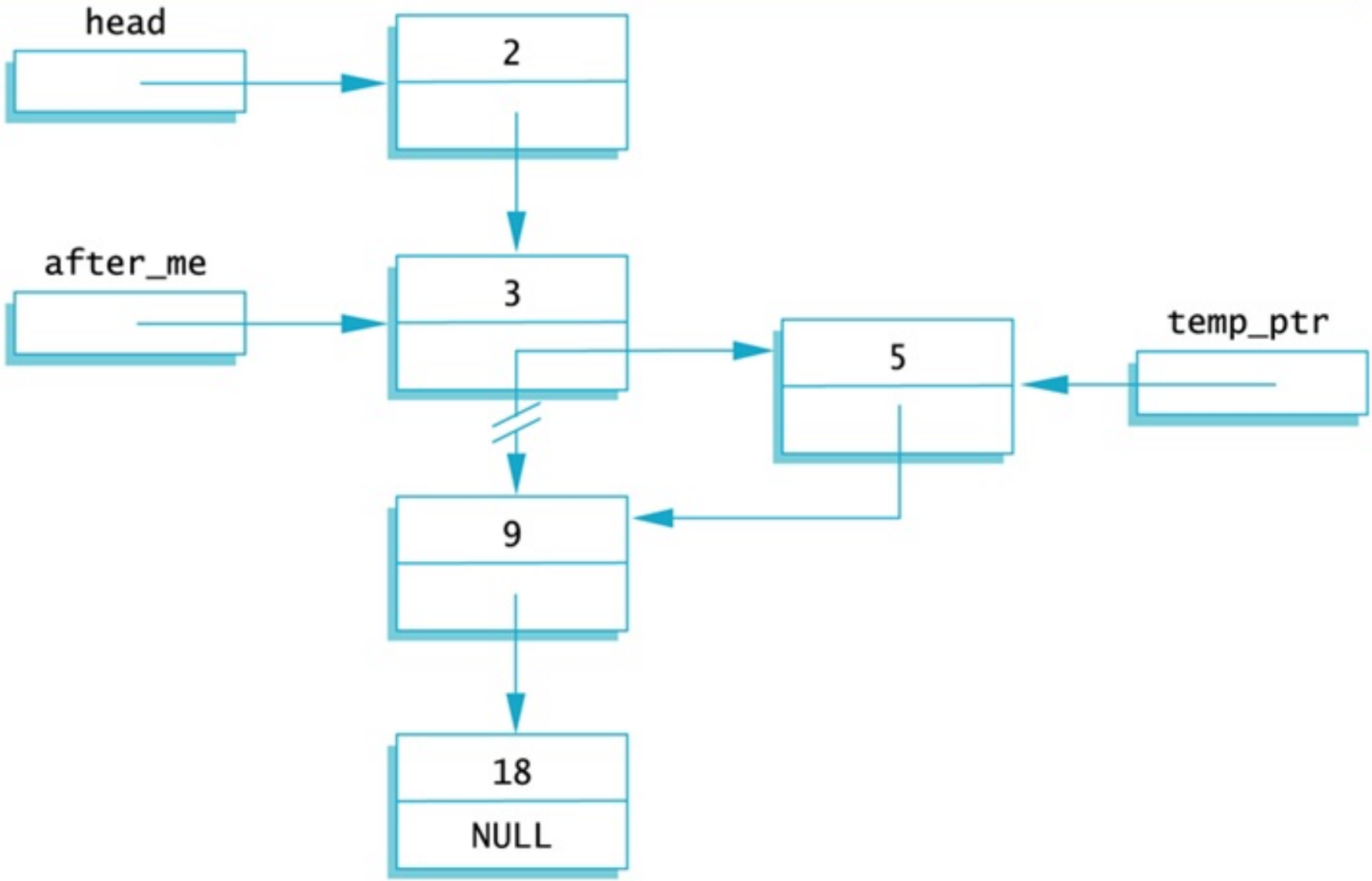
Para representar la cadena vacía se asigna `nullptr` a la variable de tipo puntero que se utiliza para referirse a la cadena, por ejemplo, mediante la instrucción `head = nullptr;`

Operaciones con Cadenas de Nodos Simplemente Enlazados

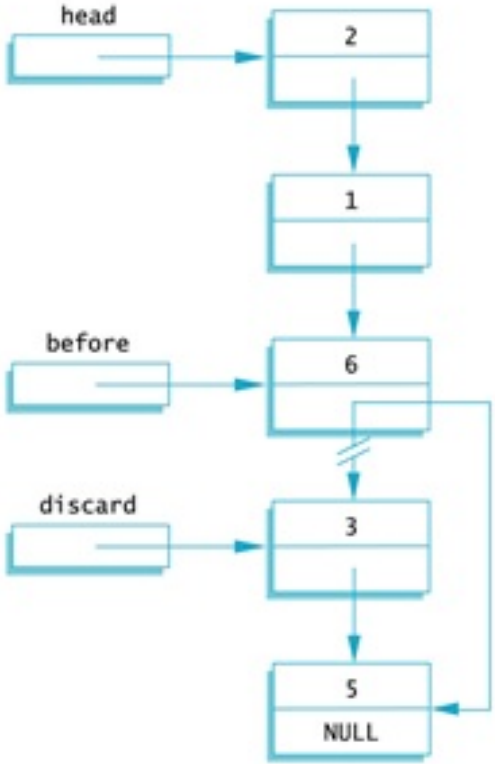
Las figuras que se muestran en los ejemplos han sido extraidas del libro "Problem solving with C++" de Walter Savitch, Pearson, 2009.

- Añadir un nodo a la cabeza de una cadena de nodos (ver figura de la página 8 y anyade_cab.cc).
- Escribir una cadena de nodos enlazados (ver anyade_cab.cc).
- Buscar un elemento en una cadena de nodos enlazados (ver figura de la página 13 y busca.cc).
- Insertar un elemento detrás de un nodo en una cadena de nodos enlazados (ver figura de la página 14 y inserta_detras.cc).
- Leer una cadena de nodos enlazados (ver inserta_detras.cc).
- Borrar un nodo de una cadena de nodos enlazados (ver figura de la pagina 15 y eliminar.cc).
- Crear una copia de una cadena de nodos enlazados (ver copia.cc).

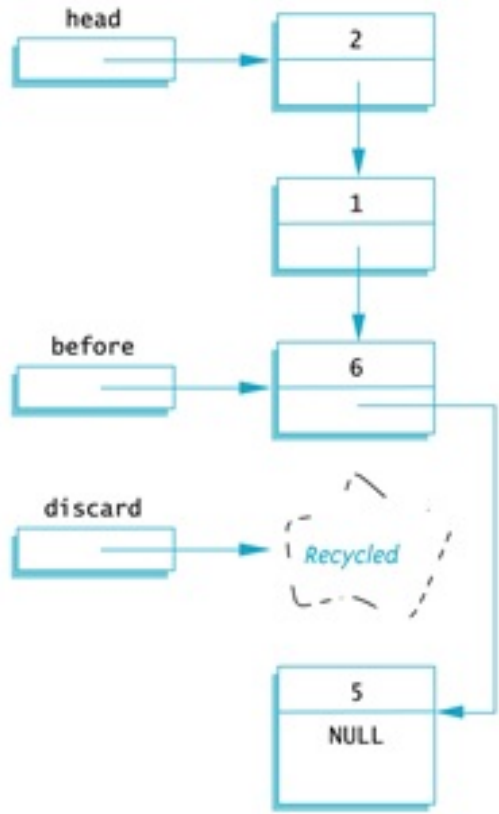

target is 6
1.
head
here
?
2
1
6
3
NULL
2.
head
here
2
Not here
1
6
3
NULL
3.
head
here
2
1
Not here
6
3
NULL
4.
head
here
2
1
6
Found
3
NULL


head
2
after_me
3
temp_ptr
5
9
18
NULL

2. before->link = discard->link;



3. *delete* discard;



Plantillas de Funciones

La siguiente plantilla (en Inglés **template**) para la acción `swapG` permite intercambiar los valores de dos variables de cualquier tipo, siempre que las dos variables sean del mismo tipo.

La definición y la declaración de una **plantilla** comienza con la línea `template<class T>` que se denomina el **prefijo de la plantilla** (en C++11 se puede usar `template<typename T>`), e informa al compilador de que la definición o declaración de función que sigue es una **plantilla** y de que `T` **es un parámetro de tipo**.

```
template<class T>
void swapG(T& var1, T& var2) {
    T temp = var1;
    var1 = var2;
    var2 = temp;
}
```

El parámetro de tipo `T` puede sustituirse por cualquier tipo, sea una clase o no. En el cuerpo de la definición de la función el parámetro `T` se usa como cualquier otro tipo.

Plantillas de Funciones

- La definición de una plantilla para una función es, de hecho, una colección de definiciones de funciones. Cada una de esas definiciones se obtiene sustituyendo el parámetro de tipo `T` por el nombre de un tipo.
- El compilador no construye una definición de la función para cada posible tipo. Pero se comporta como si hubiera construido todas esas definiciones.
- El compilador construye una definición separada para cada tipo con el que se usa la plantilla, pero no para los tipos con los que no se usa.
- Una función definida como una plantilla se invoca igual que cualquier otra función. El compilador construye la definición de la función a partir de la plantilla de la función.
- Sólo se genera una definición para un tipo determinado independientemente del número de llamadas que se hagan a la plantilla con dicho tipo.

Plantillas de Funciones

- Muchos compiladores no permiten la compilación separada de plantillas, por tanto, es necesario incluir la definición de una plantilla en el código que la usa.
- Esto puede hacerse utilizando la directiva `#include`, escribiendo la definición de la plantilla en un archivo, e incluyendo via `#include` dicho archivo en otro archivo que use la plantilla.
- Se pueden definir plantillas de funciones que tienen más de un parámetro de tipo. Por ejemplo, `make_pair` es una plantilla de una función con dos parámetros de tipo `T1` y `T2`

```
template <class T1, class T2>
  pair<T1,T2> make_pair(T1 x, T2 y);
```

- No puede haber parámetros de tipo sin usar. Todos los parámetros de una plantilla deben usarse en la función.

Plantillas de Clases

- La sintáxis de las plantillas de clases es básicamente la misma que la sintáxis de plantillas de funciones. Se coloca la siguiente línea delante de la definición de la plantilla de una clase.

```
template<class T>
class Par { ... };
```

- El parámetro de tipo `T` se usa en la definición de la clase como cualquier otro tipo.
- Como en el caso de las plantillas de funciones se puede usar otro identificador que no sea una palabra clave en lugar de `T`.
- Una vez que se ha definido la plantilla de una clase se pueden declarar objetos de la clase. La declaración debe especificar el tipo que se utiliza en lugar de T. Por ejemplo, el siguiente código declara el objeto `notas` como un Par de números reales y el objeto `equipo` como un Par de strings.

```
Par<string> equipo; // nombres de estudiantes
Par<double> notas; //notas de los estudiantes
```

Plantillas de Clases

- Los métodos de una plantilla de una clase se definen igual que los métodos de las clases normales. La única diferencia es que las definiciones de los métodos son también plantillas.
- El nombre de una plantilla de una clase puede usarse como tipo para el parámetro o el valor de retorno de una función.

```
void escribe(const vector<int>& v);
```

  Se especifica el tipo que se usa (en este caso `int`) en lugar del parámetro de tipo `T`.
- El nombre de una plantilla de una clase puede usarse como tipo para el parámetro o el valor de retorno de una plantilla de función. En este caso serviría para cualquier tipo de valores `T`.

```
template<typename T>
void escribe(const vector<T>& v);
```

- Las clases Pila, Cua, Llista y Arbre de los apuntes de PRO2 son ejemplos de plantillas de clases.