```
auto sum1(auto ... param){
  return (0 + ... + param);
}

auto sum2(auto p1, auto... ps){
  return p1 + sum2(ps...);
}
  // 1 + sum2(2, 3, 4, 5)
        2 + sum2(3, 4, 5)
            3 + sum2(4, 5)
                4 + sum2(5)
                    5 + sum2();

  auto sum2(){
    return 0;
  }

auto sum3(auto p1, auto... ps){
  if constexpr(sizeof...(ps)==0){
    return p1;
  } else {
    return p1 + sum3(ps...);
  }
}
```

Lookup en 2 Fases

Paso de parámetros a función

```
void f(int n){   // paso por valor
  //...
}
```

La función f se puede invocar con temporales y con variables, aunque se hace una copia de ellas y se transfiere

```
f(5);
int n=5;
f(n);
std::cout << n << "\n"; //5
```

El paso por valor conviene cuando no queremos modificar la variable usada en la invocación pero a la función le conviene tener una copia del valor que puede modificar sin preocupaciones

```
int p(int n){   // cantidad de dígitos.
  int r=0;
  do{
    r+=1;
    n/=10;
  }while(n!=0);
  return r;
}
```

```
void g(int &n){
  //...
}
```

No se puede invocar con temporales.
Se puede invocar con variables, con const.

```
g(5);
```

```
int n = 5;
g(n); ✓

const int n = 5;
g(n); X
```

```
const int n = 5;
g(const_cast<int &>(n)); (Comportamiento indefinido)
```

El paso por referencia es útil para que la función pueda modificar la variable que se usada en la invocación

```
void g(int& n) {
    n += 1;
}
assert(g(5) == 6);
```

paso por referencia

```
auto h(const int& n) {
    //...
}
```

Se pueden usar ter-temporales y lvalues variables.

```
h(5) ✓

int n = 5;
h(n); ✓

const int n = 5;
h(n); ✓
```

El paso por referencia a const se usa para generar copias y prometer que la variable usada en la invocación no se modificará.

```
void mal(std::string s) {
    std::cout << s;
}
```

```
void h(const std::string& s) {
    std::cout << s;
}

int&& n;
```

Enumeradores

Un enumerador es un tipo entero definido por el usuario que sirve para nombrar constantes.

```
enum dia {
    LUN,
    MAR,
    MIE,
    JUE,
    VIE,
    SAB,
    DOM;
};
```

```
constexpr int LUN = 0;
constexpr int MAR = 1;
constexpr int MIER = 2;
```

Se pueden declarar variables de un enum.

```
dia d = LUN;
dia e = MAR;
std::cout << d; // 0
dia a = 2; ✓   // a == MIER
dia b = 24; ? (posiblemente compila)
```

Se pueden elegir el valor inicial de un enum.

```
enum prueba {
  x = 5,
  y,          // 6
  z           // 7
};
```

```
int main()
  std::cout << LUN << "\n";
   dia a = ME;
  std::cout << d << "\n";
   dia b = dia(5);
};
```

((11 enum dia b = 5)

sizeof(d) == 4;

El tipo subyacente de un enum por omisión es int. Este tipo se puede especificar al declarar el enum.

```
<stdint.h>
enum dia : int8_t {      sizeof(...) == 1,
  //...
};
```

Por omisión ocurren conversiones implícitas entre enumeradores e int.

```
std::cout << LUN + 1 << "\n"; // (0) + 1 = 1
```

```
enum class dia {
    LUN, MAR, MIE
};
```

```
dia d = dia::LUN;
```

Declarar un enum en una enum class provoca que el enumerador ya no participe en operaciones aritméticas sin hacer un cast explícito, y también provoca que los enumeradores del enum ya no sean globales

Se necesita el prefijo nombre_enum::

```
enum class prueba {
    XX, YY;
};
```

```
prueba p = XX; ✗
prueba p = prueba::XX; ✓
std::cout << int(p) + 1; //bien
```

Estructuras

Una estructura es un tipo definido para el usuario que almacena variables, posiblemente de tipos distintos

↳ variable miembro

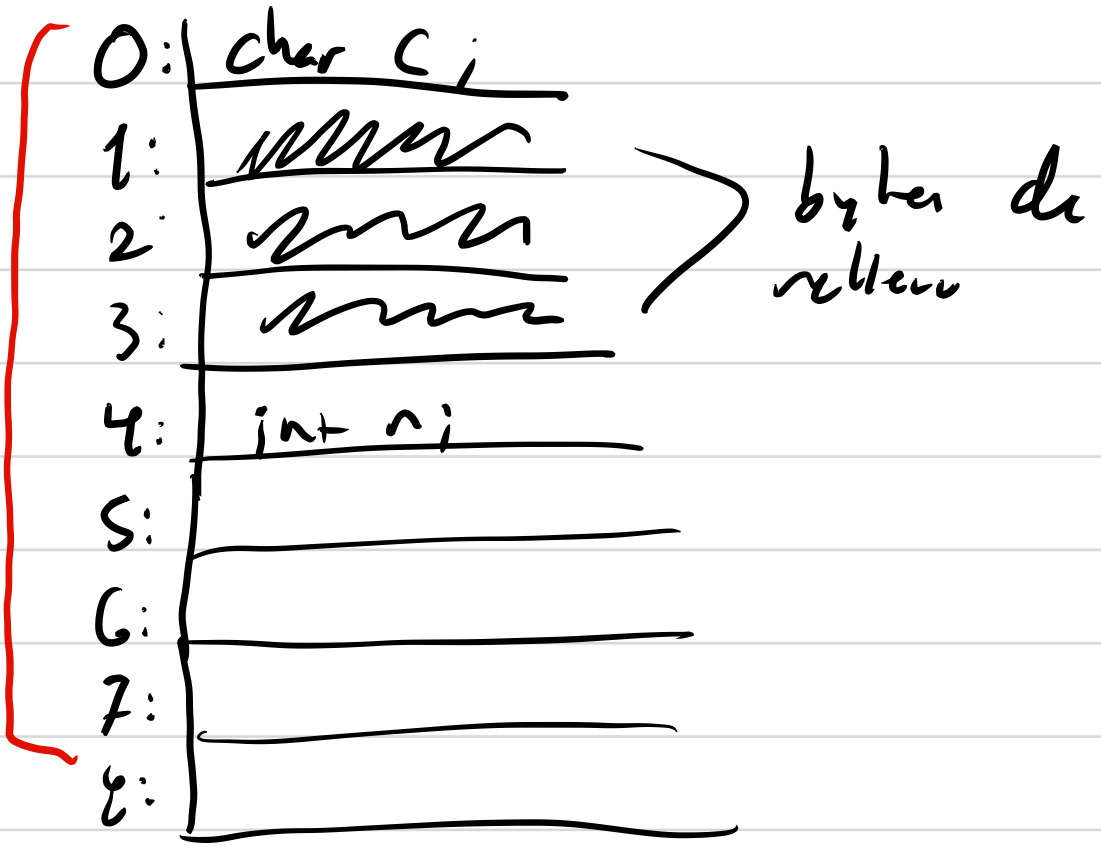
```
struct Fecha {
    int dia;
    int mes, anio;
    std::string mensaje;
};

Fecha f1;
f1.dia = 8;
f1.mes = 11;
f1.anio = 2022;
f1.mensaje = ":(";

std::cout << f1.mes;

Fecha f2 = {8, 11, 2022, "hola"};

// se inicializan según el orden de declaración en el
// std::cout

Fecha f3 = f2; // se copia cada miembro:
               //(por omisión)

Fecha f4 = {8, 11}; // anio=0, mensaje=""

Fecha f5 = {
    .dia = 8, .anio = 2022, .mes = 11, .mensaje = ":("};
```

Se pueden declarar variables de structs anónimas.

```
int main() {
  struct {
    int x, y;
  } v;
```

El sizeof de un struct mayor o igual que la suma de sus miembros.
Pueden existir bytes de relleno entre sus miembros.

```
struct prueba {
  char c;
  int n;
};
```



C layout

Los miembros de un struct se acomodan en memoria en orden de declaración.

sizeof(T)
alignof(T)
alineación natural

El operador alignof(T) obtiene la alineación natural de t. Variables de tipo T querrán tener direcciones múltiplos de su alineación
Para tipos primitivos, normalmente
sizeof(T) == alignof(T).

El alignof de un struct será el alignof más fuerte de sus miembros

El operador alignas permite cambiar la alineación de un variable o tipo.

alignas(E) char c;

El sizeof de un struct debe ser múltiplo de su alineación
para cumplir se declara un array de structs todos coinciden.

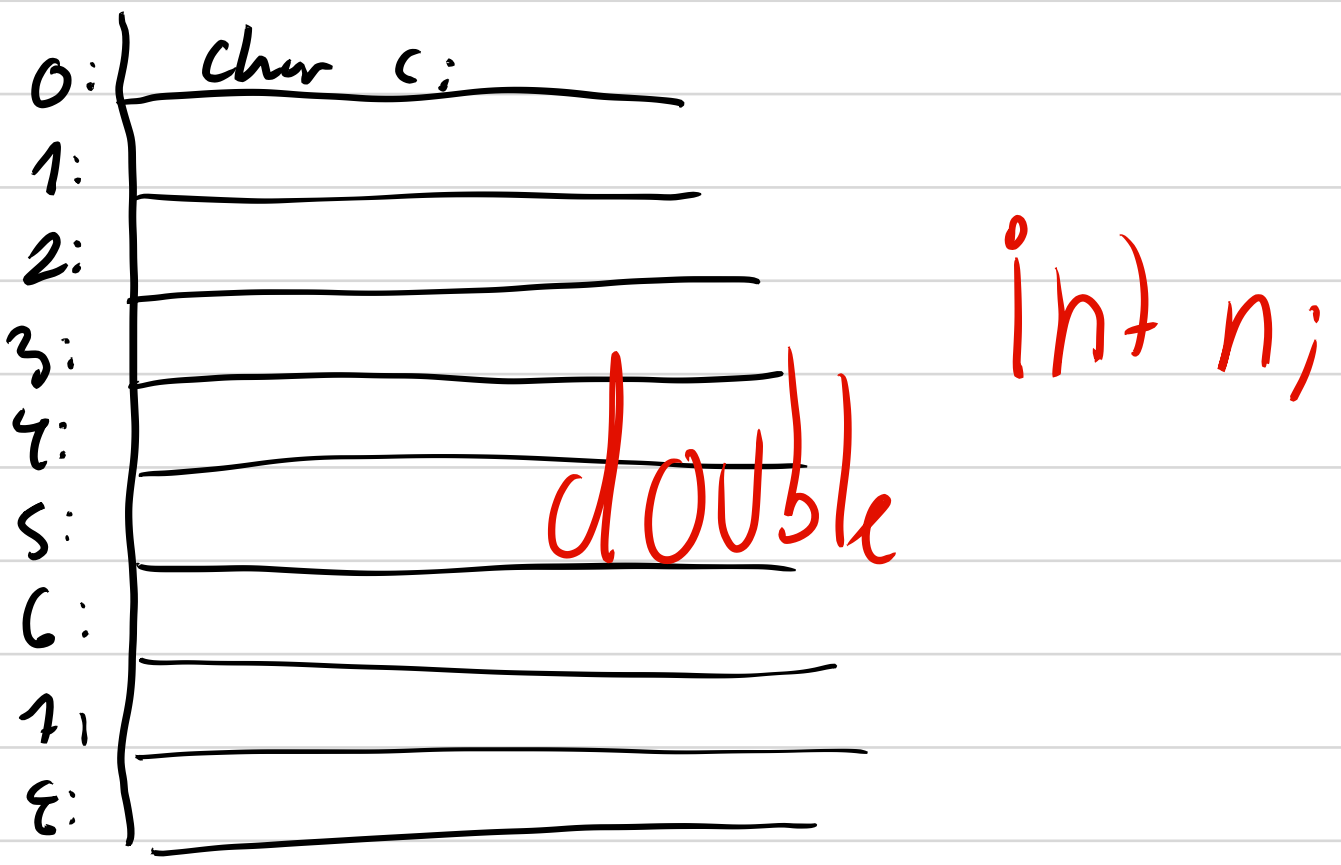
En un struct, se pueden declarar cuántos bits queremos que use un miembro.

bitfield

```
struct name {
  int n: 24;      mascaras de bits
  int m: 8;
};
```

Una unión es similar a un struct excepto que solo se aparta memoria para el más grande de sus miembros. Por dicha razón, todos los miembros emplean la misma memoria

```
union prueba{
    char c;
    double d;
    int n;
};
```




```
prueba v;
v.d = 3.14;
cout << v.c;
```