Estructuras de datos básicas

AED I

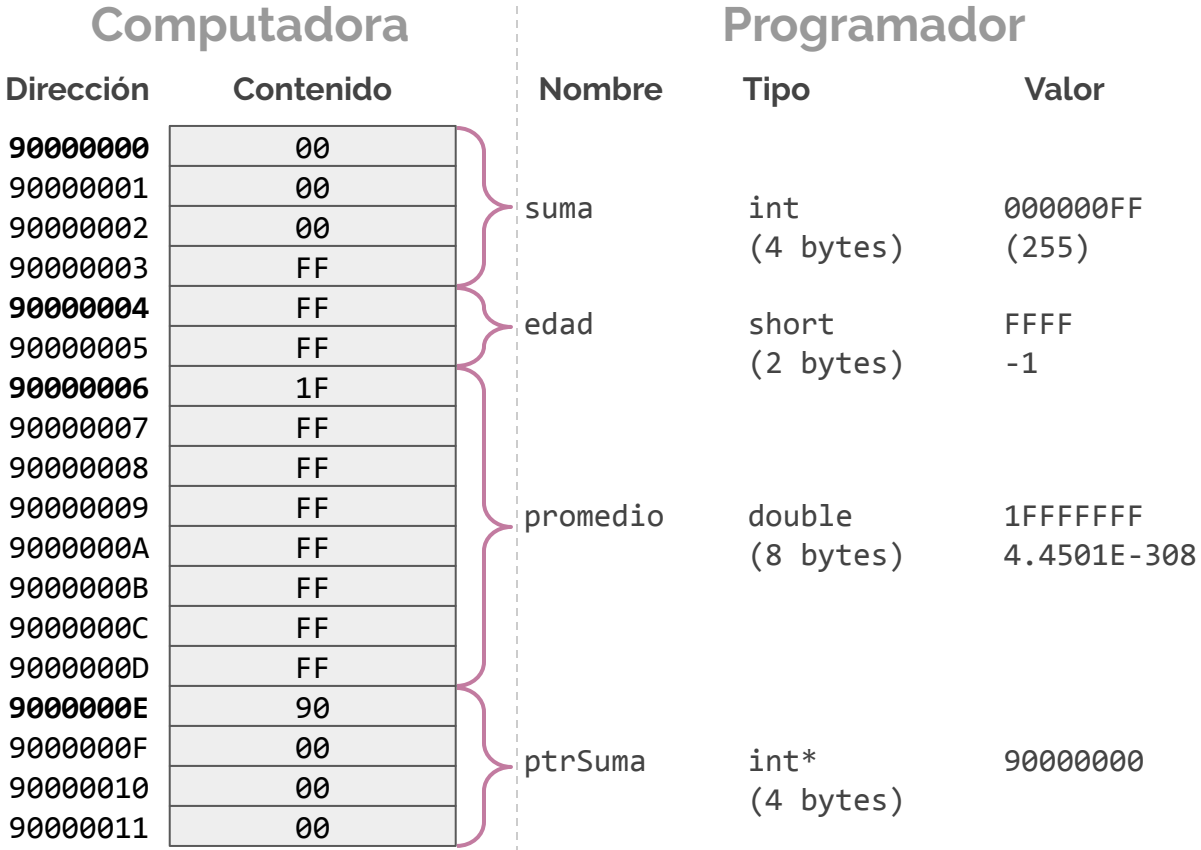
2020

UBP UNIVERSIDAD BLAS PASCAL

Punteros, Referencias y Asignación Dinámica de la Memoria

Punteros

variables y memoria



puntero

Una variable tipo puntero o sencillamente un **puntero** es como cualquier otra variable que puede almacenar un dato.

A diferencia de una variable común que almacena un valor (como un int, un char o un double), **un puntero almacena una dirección de memoria**

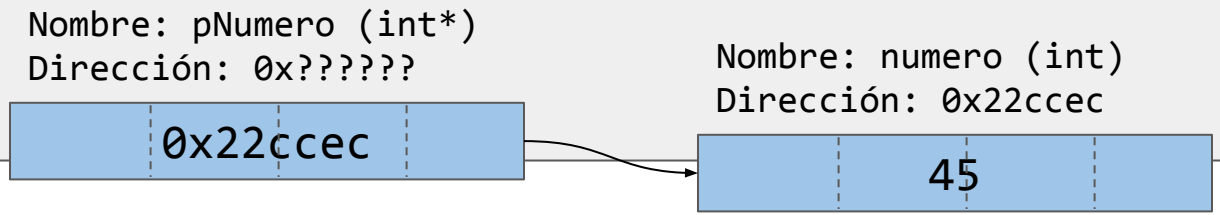
declaración

```
<tipo> *ptr;
<tipo>* ptr;
<tipo> * ptr;

// ejemplos

int* ptrEntero;
double* ptrDouble;
```

inicialización usando el operador de dirección &

El operador de dirección o referencia (&) opera sobre una variable y devuelve la dirección de la variable

```
int numero = 45;
int* pNumero;
pNumero = &numero;

int* pNumero2 = &numero;
```



operación de indirección *

El operador de indirección o derreferencia (*) opera sobre un puntero y devuelve el valor guardado en la dirección almacenada en la variable puntero

```
int numero = 45;
int* pNumero = &numero;
cout << pNumero << endl;     // Imprime el contenido de la variable puntero
cout << *pNumero  << endl;   // Imprime el valor al que apunta el puntero (45)
*pNumero = 90;               // Asigna un valor al lugar apuntado por el puntero
cout << *pNumero << endl;    // Muestra el nuevo valor apuntado por el puntero (90)
cout << numero << endl;      // También cambió el valor de la variable (90)
```

punteros y tipos

Un puntero está asociado al tipo (del valor al que apunta) que se especifica al declararlo. Solo puede tener direcciones del tipo del que ha sido declarado

```
int i= 88;
double d= 50.65;

int* iPtr = &i;
double* dPtr = &d;

iPtr = &d; // Error: no puede mantener una dirección de diferente tipo
dPtr = &i; // Error: no puede mantener una dirección de diferente tipo
iPtr = i;  // Error: a iPtr se le puede asignar una dirección de un entero no un valor
```

punteros genericos o void (void*)

```
void incrementar(void* dato, int psize)
{
  if ( psize == sizeof(char) )
  { char* pchar; pchar=(char*)dato; ++(*pchar); }
  else if (psize == sizeof(int) )
  { int* pint; pint=(int*)dato; ++(*pint); }
}
```

```
int main ()
{
  char a = 'x';
  int b = 1602;
  incrementar(&a,sizeof(a));
  incrementar(&b,sizeof(b));
  cout << a << ", " << b << '\n';
  return 0;
}
```

Un puntero void indica que no tiene un tipo definido. Esto es, puede ser usado para cualquier tipo. Con una limitación importante: el dato al que apunta ese puntero no puede ser derreferenciado directamente. Debemos convertirlo (castearlo) a un tipo específico antes de usarlo.

Un uso típico es en el paso de parámetros genéricos a una función

punteros no inicializados

```
int* iPtr;
*iPtr = 55;
cout << *iPtr << endl;
```

Este programa tiene un error de lógica muy serio. Como el puntero no ha sido inicializado la asignación de la segunda línea va a corromper la memoria en una locación desconocida.

Este programa compilará correctamente aunque con una advertencia.

punteros nulos

```
int* iPtr = NULL;
int* iPtr = nullptr;
cout << *iPtr << endl;

int* iPtr2 = 0;
```

```
// ERROR! STATUS_ACCESS_VIOLATION exception
```

Un puntero se puede inicializar a `NULL` o `0`. En C++11 podemos usar `nullptr`. En este caso no apunta a nada y se llama **Puntero Nulo**
Derreferenciar un puntero nulo provoca una excepción de tipo STATUS_ACCESS_VIOLATION.

Inicializar un puntero a NULL cuando lo declaramos es una buena práctica

Ejercicio 2.01.

Intercambiar valores:

Realizá un programa que permita leer dos valores enteros por consola y asignarlos a las variables **a** y **b**.

A continuación se debe invocar a una función **`void cambiar(int*, int*)`** que asigne a **a** el valor de **b** y a **b** el valor de **a**

Ejercicio 2.01.

```
#include <iostream>
using namespace std;

void cambiar(int *a,int *b)
{
    int t;
     t   = *a;
    *a   = *b;
    *b   =  t;
}
```

```
int main()
{
    int n1,n2;
    cout<<"Ingrese el valor de n1: "<<endl;
    cin>>n1;
    cout<<"Ingrese el valor de n2: "<<endl;
    cin>>n2;
    cout<<"Antes de intercambiar"<<endl;
    cout<<"n1 = "<<n1<<" y n2 = "<<n2<<endl;

    cambiar(&n1,&n2);

    cout<<"Antes de intercambiar"<<endl;
    cout<<"n1 = "<<n1<<" y n2 = "<<n2<<endl;
    return 0;
}
```

Referencia

referencia o alias (&)

Funcionan de forma similar a los punteros salvo por el hecho de que estoy obligado a inicializarlas

```
<tipo> &nuevoNombre = nombreExistente;
<tipo>& nuevoNombre= nombreExistente;
<tipo> & nuevoNombre= nombreExistente;

// ejemplos

int& refNumero = numero;
double& reprecio = precio;
```

paso por valor

En c y c++ por defecto los **argumentos** se pasan a las funciones **por valor** (excepto los arrays que son punteros en sí mismos)

```
void incrementar(int numero){
     cout<<"numero en incrementar() antes: "<<numero<<endl;
     ++numero;
     cout<<"numero en incrementar() despues: "<<numero<<endl;
}

int main() {
     int numero = 10;
     cout<<"numero en main() antes de invocar a incrementar(): "<<numero<<endl;
     incrementar(numero);
     cout<<"numero en main() despues de invocar a incrementar(): "<<numero<<endl;
}
```

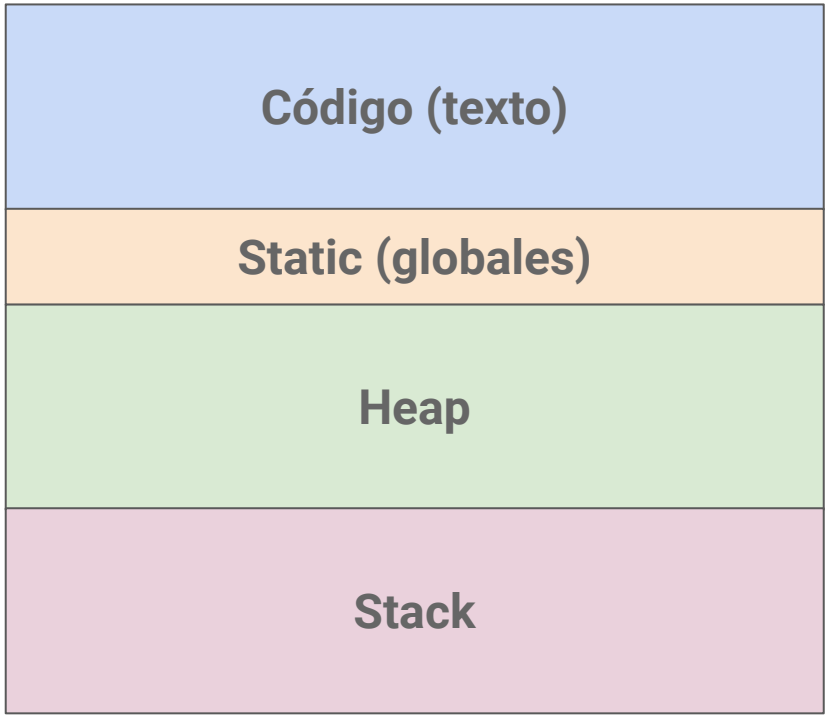
paso por referencia

La única diferencia está en la declaración del parámetro. Ahora estamos pasando una referencia a la variable y no una copia.

```
void incrementar(int& numero){
     cout<<"numero en incrementar() antes: "<<numero<<endl;
     ++numero;
     cout<<"numero en incrementar() despues: "<<numero<<endl;
}

int main() {
     int numero = 10;
     cout<<"numero en main() antes de invocar a incrementar(): "<<numero<<endl;
     incrementar(numero);
     cout<<"numero en main() despues de invocar a incrementar(): "<<numero<<endl;
}
```

Asignación dinámica de la memoria

distribución de la memoria

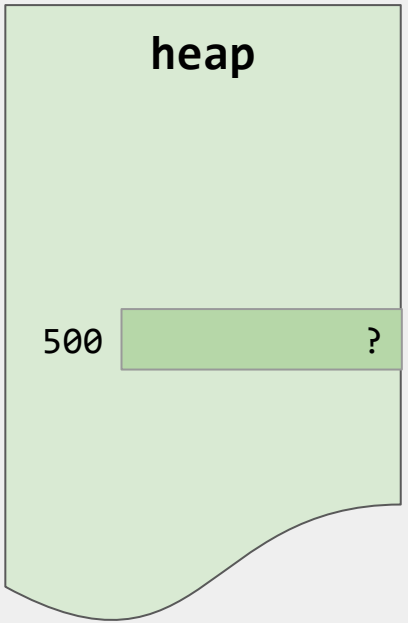


asignación dinámica de la memoria

```
new <tipo>;
```

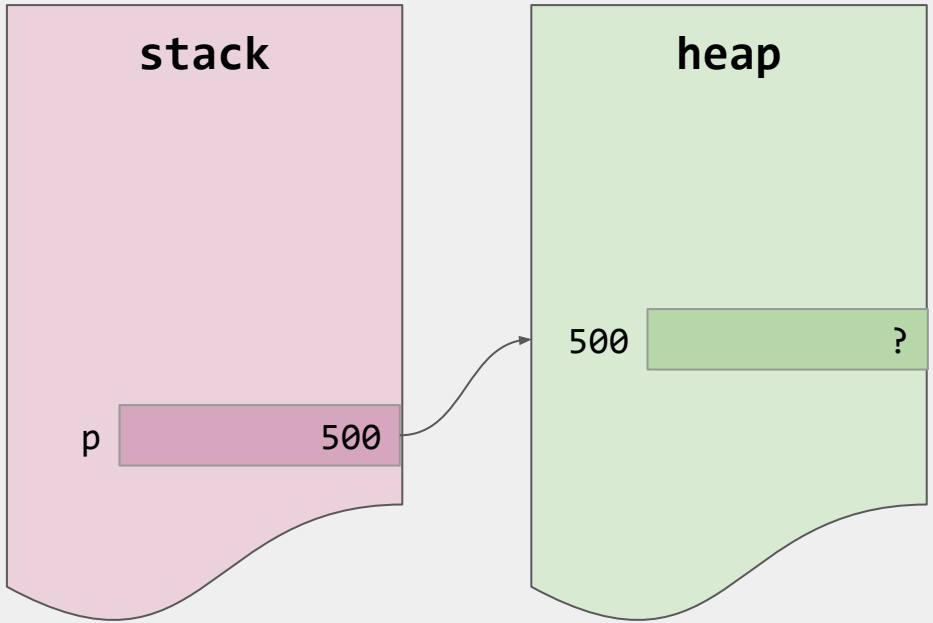
asignación dinámica de la memoria

```
new int;
```



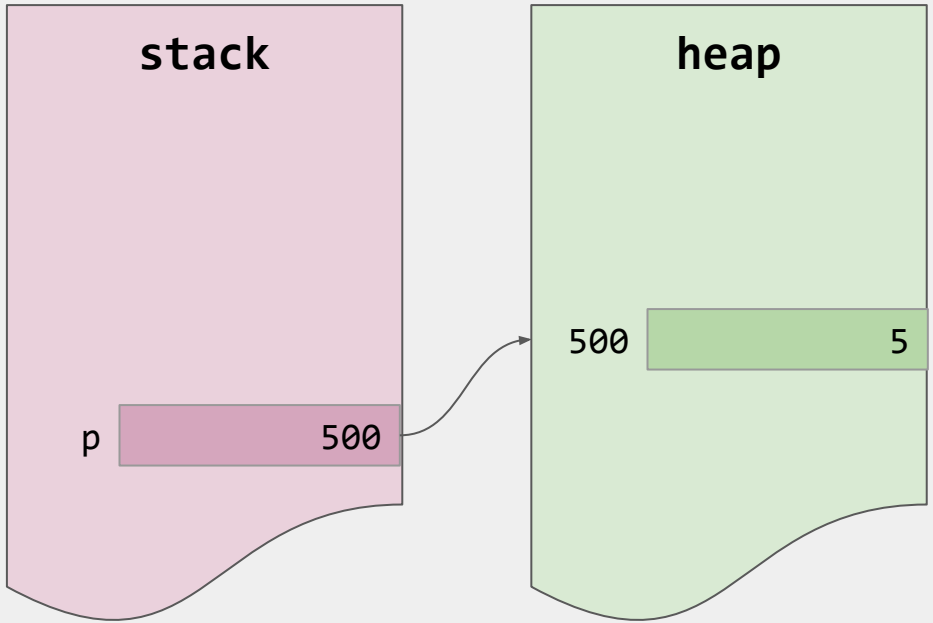
asignación dinámica de la memoria

```
int* p = new int;
```



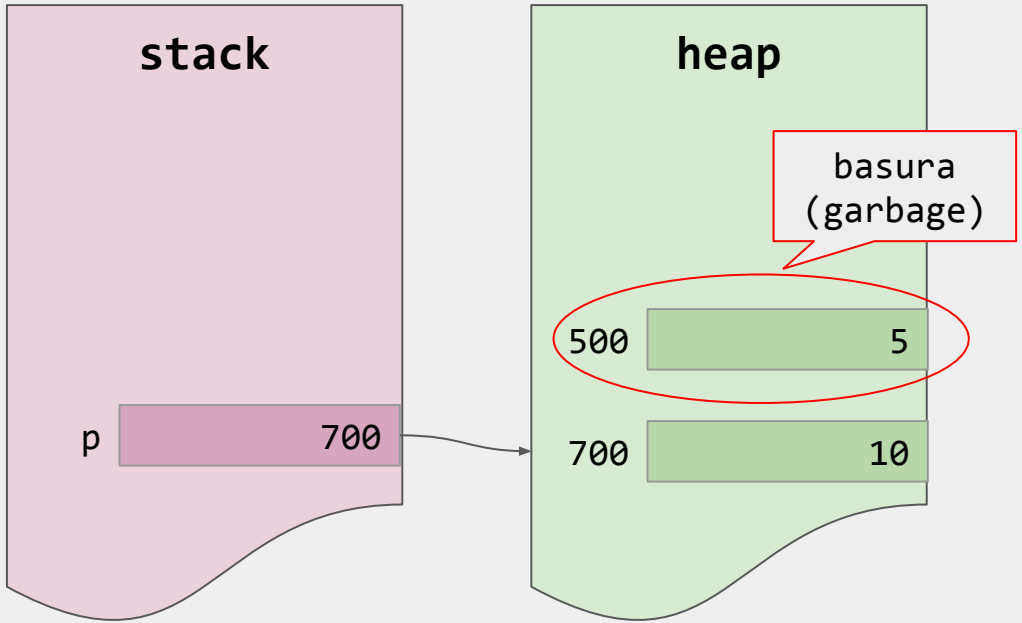
asignación dinámica de la memoria

```
int* p = new int;
*p = 5;
```



asignación dinámica de la memoria

```
int* p = new int;
*p = 5;
p = new int(10);
```

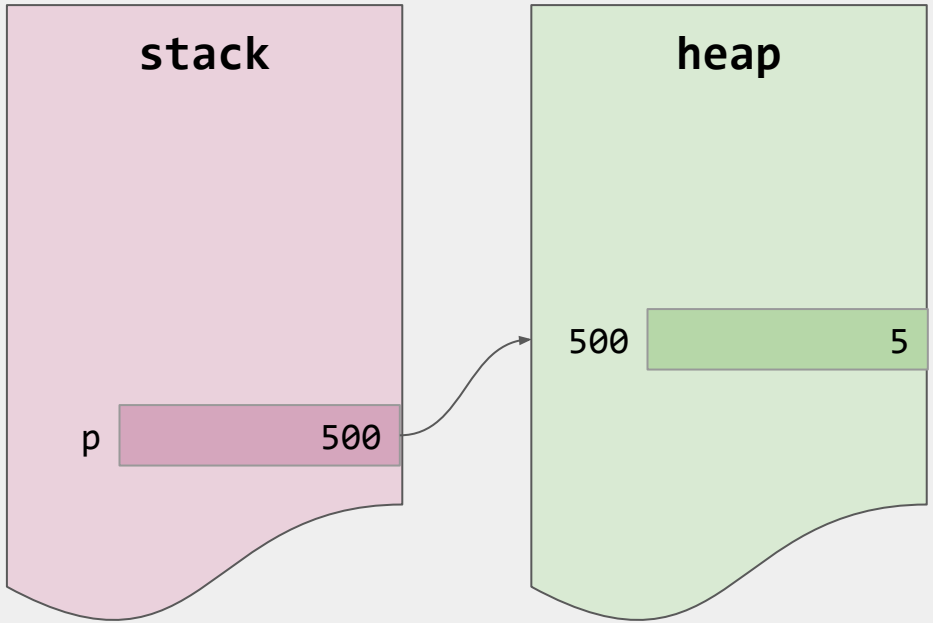


asignación dinámica de la memoria

Volvemos atrás <<

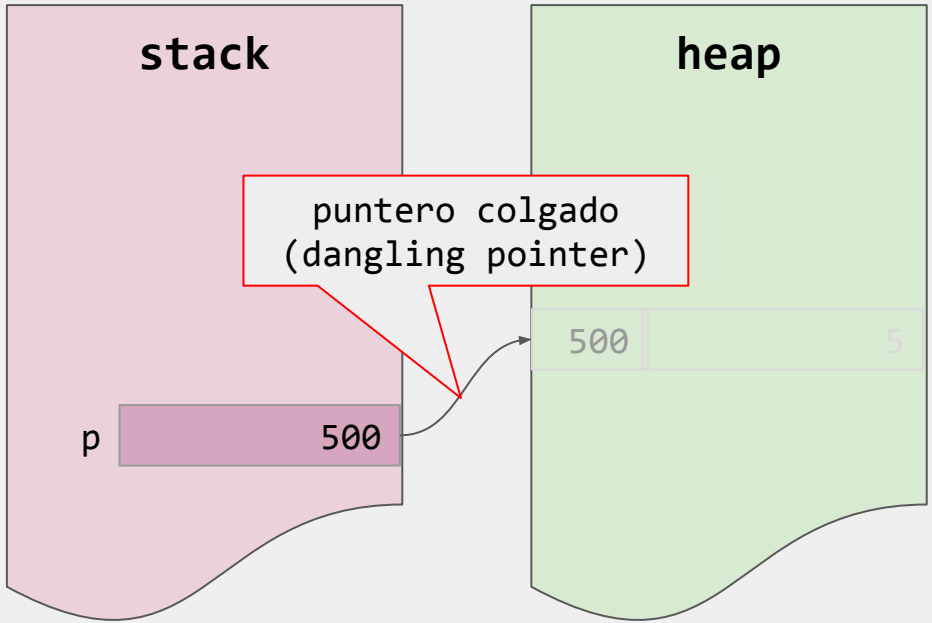
asignación dinámica de la memoria

```
int* p = new int;
*p = 5;
```



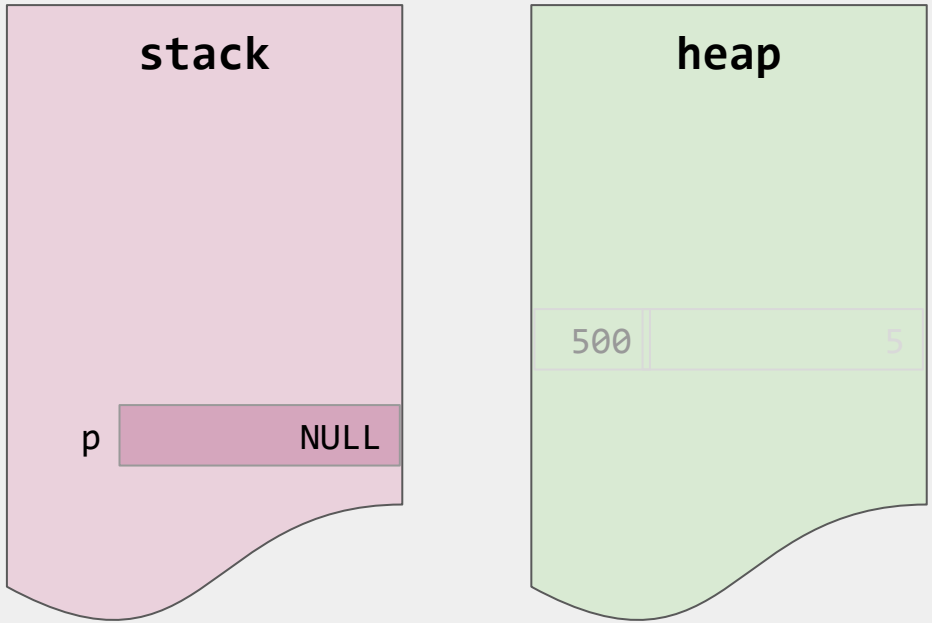
asignación dinámica de la memoria

```
int* p = new int;
*p = 5;
delete p;
```



asignación dinámica de la memoria

```
int* p = new int;
*p = 5;
delete p;
p = NULL;
```

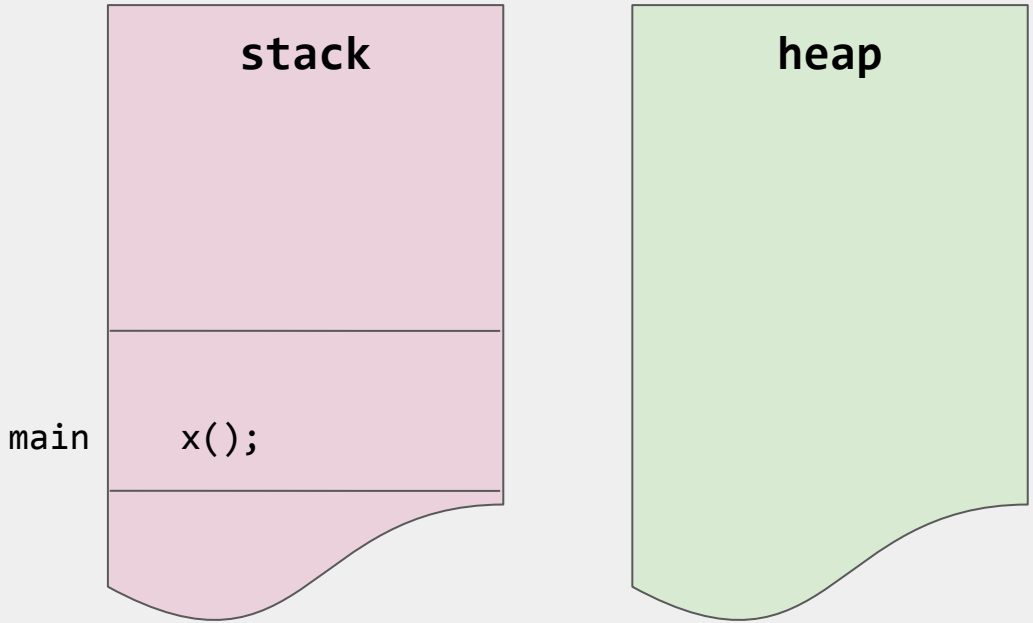


asignación dinámica de la memoria

Otra forma de generar basura

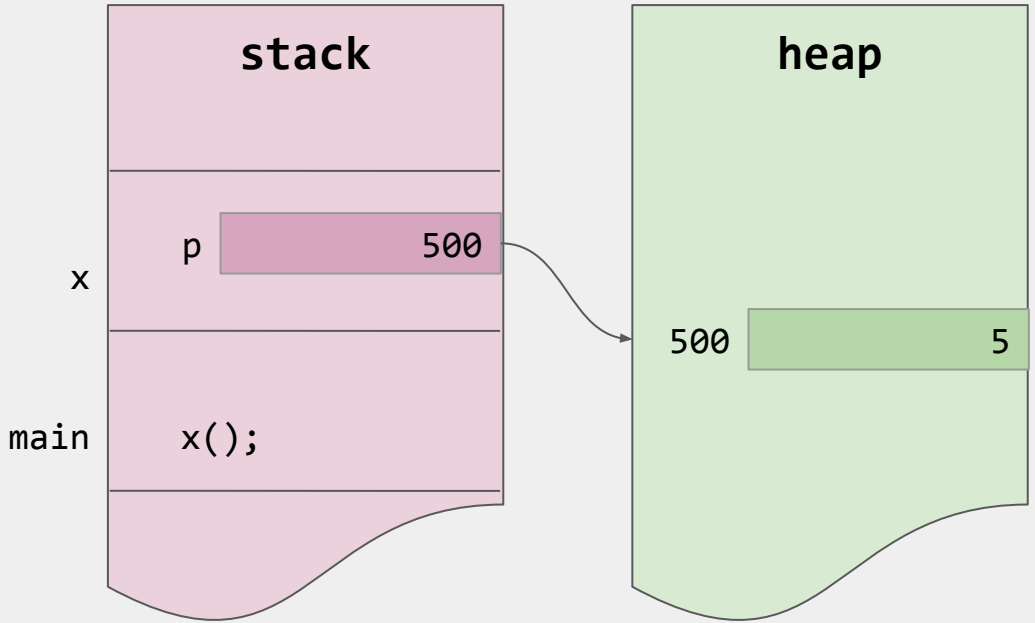
asignación dinámica de la memoria

```
void x(){
    int* p = new int;
    // más código
}

int main(){
    ...
    x();
    ...
}
```



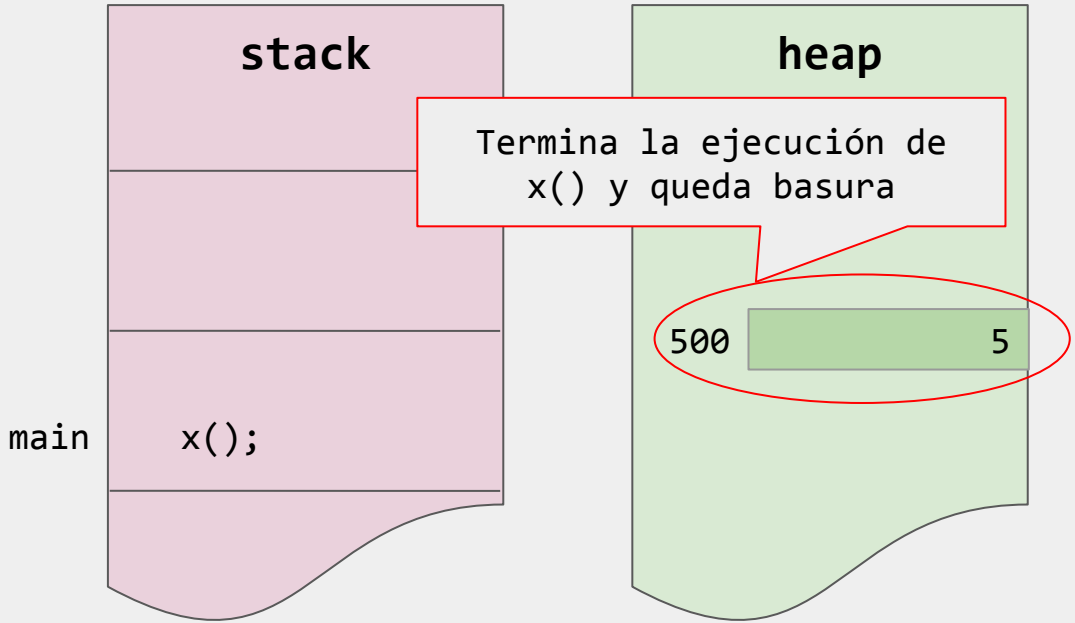
asignación dinámica de la memoria

```
void x(){
    int* p = new int;
    // más código
}

int main(){
    ...
    x();
    ...
}
```



asignación dinámica de la memoria

```
void x(){
    int* p = new int;
    // más código
}

int main(){
    ...
    x();
    ...
}
```



arrays: operadores

```
new[]
delete[]
```

Punteros y arrays

arrays

En C y C++ **un array es un puntero** que apunta al primer elemento del array

```
const int N = 5;
int numeros[N] = {11, 22, 44, 21, 41};

cout << &numeros[0] << endl;
// direccion del primer elemento (0x22fef8)

cout << numeros << endl;
// lo mismo (0x22fef8)
```

aritmética de punteros

Vimos que `numeros` es un puntero al primer elemento del array, `(numeros + 1)` apunta al siguiente entero y no a la dirección contigua.

Asi:

```
int numeros[] = {11, 22, 33};
int * iPtr = numeros;
cout << iPtr << endl;
cout << iPtr + 1 << endl;
cout << *iPtr << endl;
cout << *(iPtr + 1) << endl;
cout << *(iPtr + 2) << endl;
cout << *iPtr + 1 << endl;
```

aritmética de punteros

Vimos que `numeros` es un puntero al primer elemento del array, `(numeros + 1)` apunta al siguiente entero y no a la dirección contigua.

Asi:

```
int numeros[] = {11, 22, 33};
int * iPtr = numeros;
cout << iPtr << endl;          // 0x22cd30
cout << iPtr + 1 << endl;      // 0x22cd34
cout << *iPtr << endl;         // 11
cout << *(iPtr + 1) << endl; // 22
cout << *(iPtr + 2) << endl; // 33
cout << *iPtr + 1 << endl;   // 12
```

tamaño de un array

La operación `sizeof(array)` nos da el tamaño total del array. Si la dividimos por el tamaño de un elemento, podemos obtener la cantidad de elementos.

```
int numeros[100];
cout << sizeof(numeros) << endl;
cout << sizeof(numeros[0]) << endl;
cout << "Tiene " << sizeof(numeros) / sizeof(numeros[0]) <<
" elementos" << endl;
```

pasar un array a una función

Los arrays **se pasan** a una función **como un puntero** al primer elemento del array

```
int max(int numeros[], int nElementos);
int max(int *numeros, int nElementos);
int max(int numeros[50], int nElementos); //el tamaño se ignora
```

El tamaño del array no es parte del parámetro array. El compilador no tiene la capacidad de deducir el tamaño del array del puntero y no realiza verificación de límites

arrays creación dinámica: ejemplo

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
   const int N = 5;
   int * pArray;
   pArray = new int[N];  // Asignación usando el operador new[]
   for (int i = 0; i < N; ++i) {
      *(pArray + i) = rand() % 100;
   }
   // Imprimir el array
   for (int i = 0; i < N; ++i) {
      cout << *(pArray + i) << " ";
   }
   cout << endl;
   delete[] pArray;  // Dealocar usando el operador delete[]
   return 0;
}
```

Ejercicio 2.02.

Imprimir array y sus direcciones:

Permití al usuario ingresar un arreglo y a continuación imprimí el valor de cada elemento con su dirección

Ejercicio 2.02.

```
#include <iostream>
using namespace std;
int main()
{
    int arr[10];
    int *pa;
    int  i;

    pa=&arr[0];

    cout<<"Ingrese los elementos del array:\n"<<endl;
    for(i=0;i < 10; i++){
        cout<<"Ingrese a["<<i+1<<"]: ";
        cin>> *(pa+i);
    }

    cout<<endl<<"Los elementos del array son:\n"<<endl;
    for(i=0;i<10;i++){
        cout<<pa+i<<"\t\t"<<*(pa+i)<<endl;
    }

    return 0;
}
```