C++

پریسا حامد روح بخش

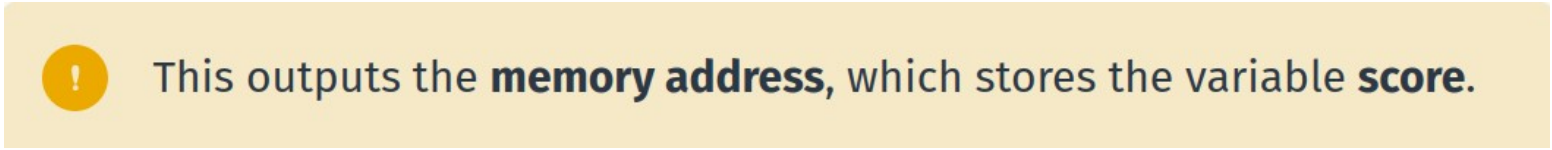
موسسه ی پارس پژوهان

فصل ششم

- اشاره گر ها == Pointers
- Reference
- Dereferencing
- Static & Dynamic Memory
- Dangling Pointers

Pointers

- Every variable is a memory location, which has its address defined.
- That address can be accessed using the ampersand (&) operator (also called the address-of operator), which denotes an address in memory.



Pointers

CPP

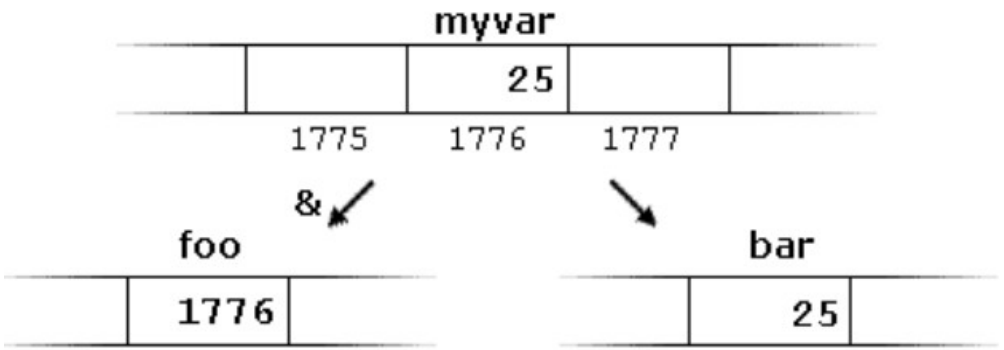
```cpp
#include <iostream>
using namespace std;

int main()
{
    int score = 5;
    cout << &score << endl;

    return 0;
}
```

```
0x7ffe3b4b74cc
```

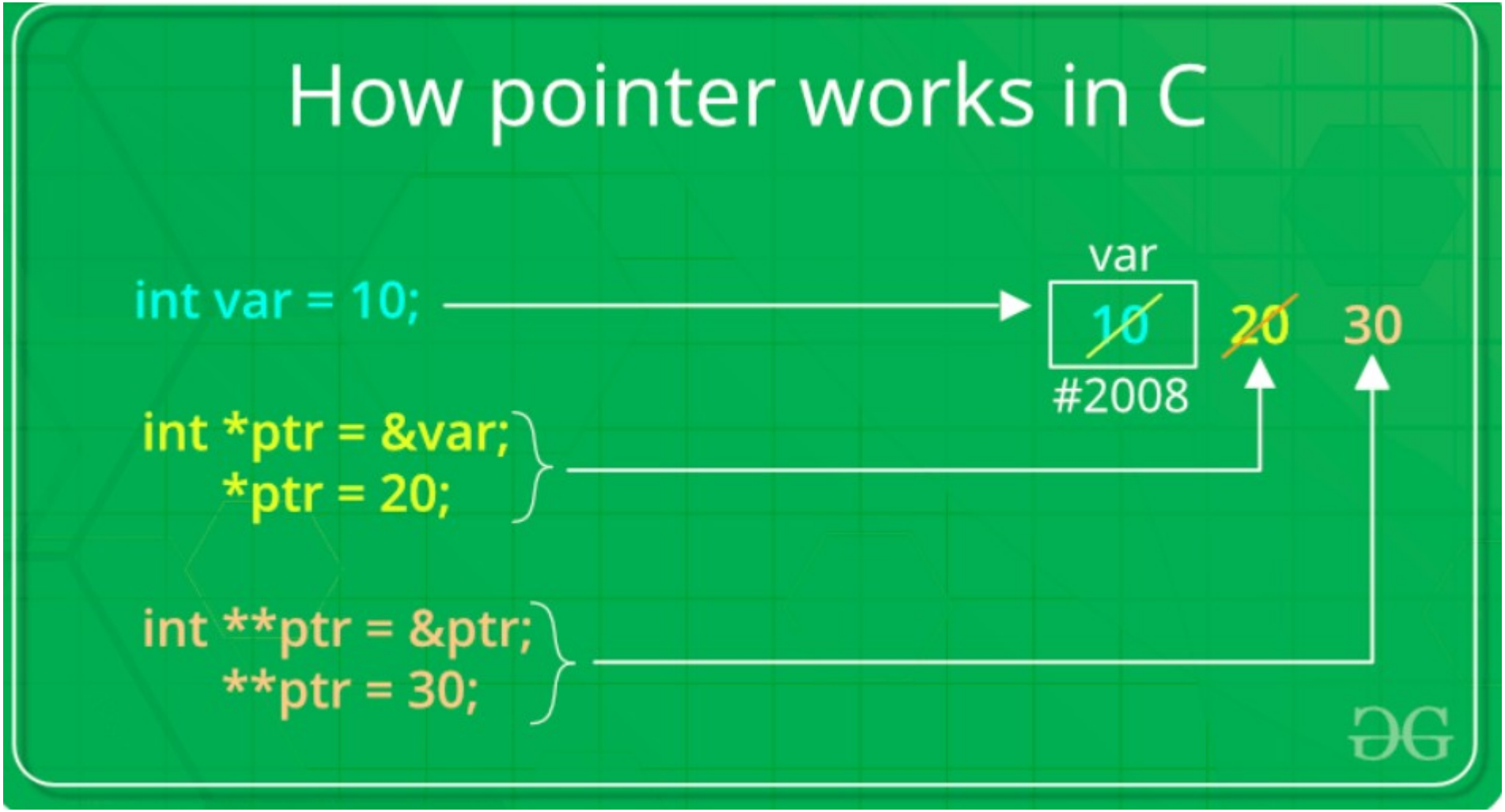
Pointers

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

int main() {
  string food = "Pizza";

  cout << food << "\n";
  cout << &food << "\n";
  return 0;
}
```

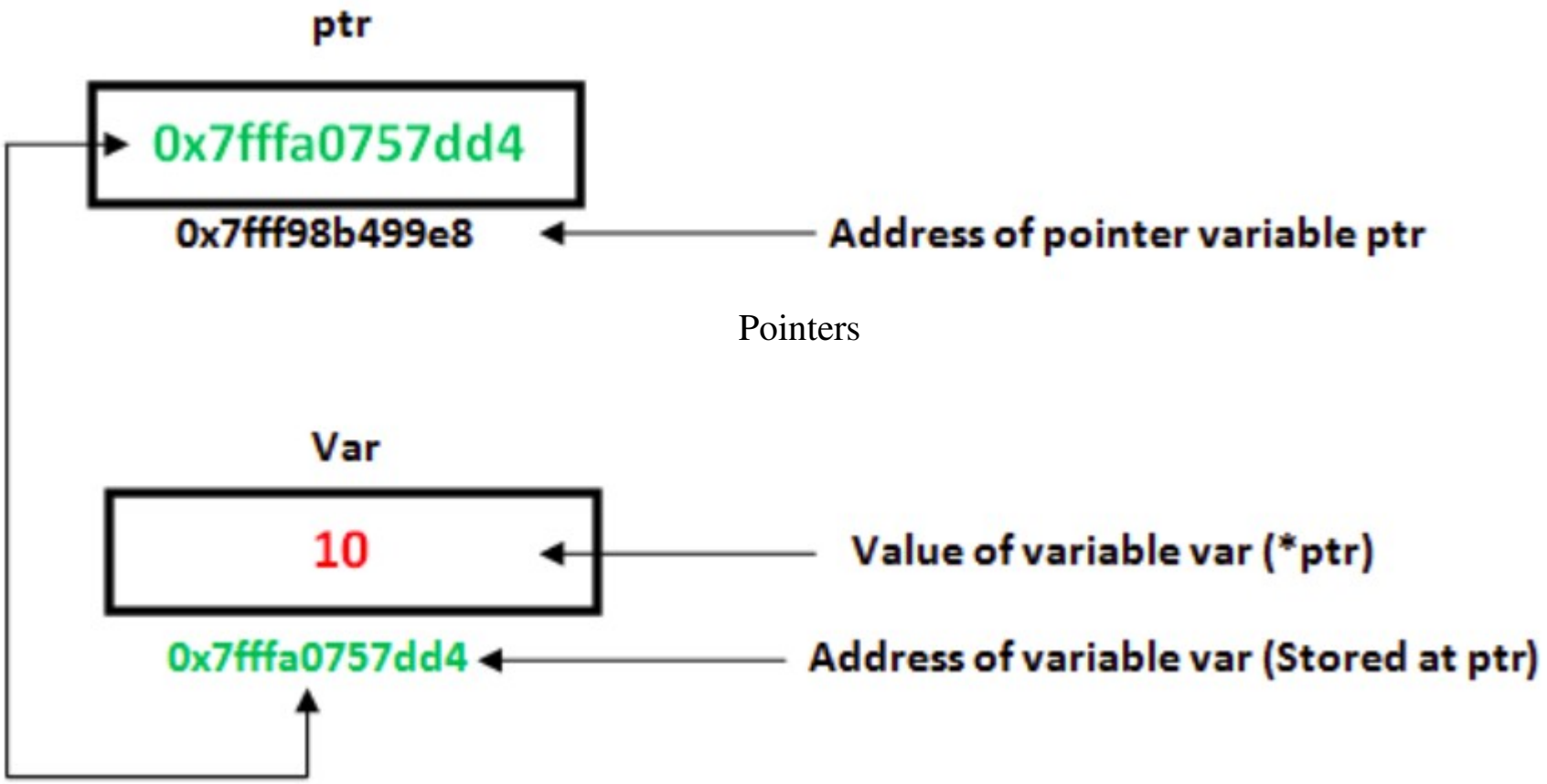
```
Pizza
0x6dfed4
```



Pointers


How pointer works in C
int var = 10;
var
10
20
30
#2008
int *ptr = &var;
*ptr = 20;
int **ptr = &ptr;
**ptr = 30;

Pointers



Pointers

Pointers

- A **pointer** is a variable, with the **address of another variable** as its value.
- In C++, pointers help **make certain tasks easier** to perform. Other tasks, such as dynamic memory allocation, cannot be performed without using pointers.
- **All pointers share the same data type** - a long hexadecimal number that represents a memory address.

> The only difference between pointers of different data types is the data type of the variable that the pointer points to.

Pointers

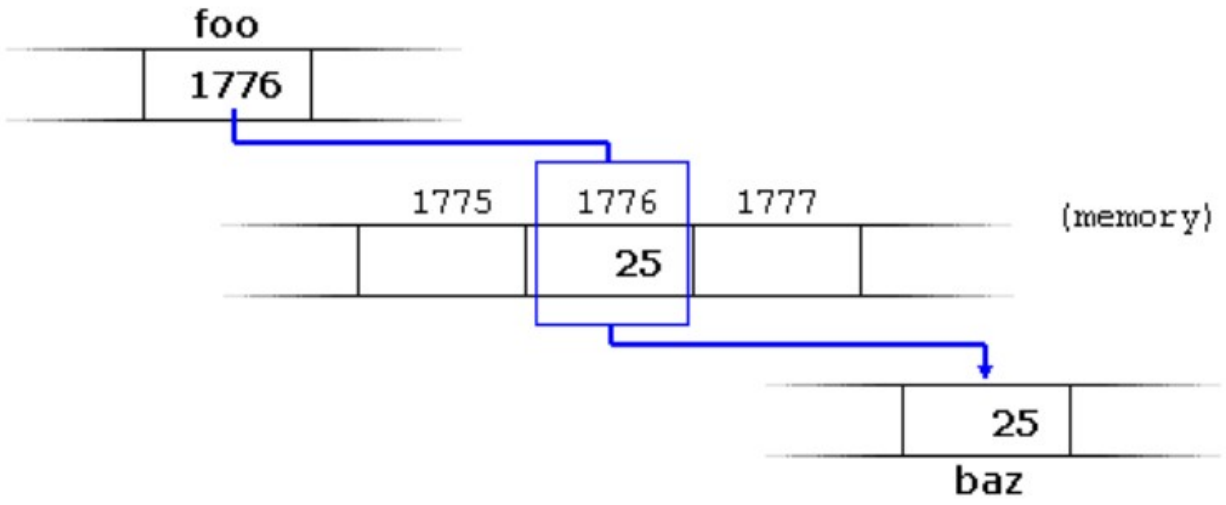
- A pointer is a variable, and like any other variable, it must be declared before you can work with it.
- The asterisk sign is used to declare a pointer (the same asterisk that you use for multiplication), however, in this statement the asterisk is being used to designate a variable as a pointer.

Pointers

```
int *ip;  // pointer to an integer
double *dp;   // pointer to a double
float *fp;  // pointer to a float
char *ch;  // pointer to a character
```

The asterisk sign can be placed next to the data type, or the variable name, or in the middle.

baz = *foo ;



Pointers

CPP

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

int main()
{
    int score = 5;
    int *scorePtr;
    scorePtr = &score;

    cout << scorePtr << endl;

    return 0;
}
```

```
0x7ffe4ee07da4
```

Now, **scorePtr's** value is the memory location of **score**.

Pointers

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

int main() {
  string food = "Pizza";  // A string variable
  string* ptr = &food;  // A pointer variable that stores the
address of food

  // Output the value of food
  cout << food << "\n";

  // Output the memory address of food
  cout << &food << "\n";

  // Output the memory address of food with the pointer
  cout << ptr << "\n";
  return 0;
}
```

Pointers

- The asterisk (*) is used in declaring a pointer for the simple purpose of indicating that it is a pointer (The asterisk is part of its type compound specifier). Don't confuse this with the dereference operator, which is used to obtain the value located at the specified address. They are simply two different things represented with the same sign.

Pointers

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

int main()
{
    int var = 50;
    int  *p;
    p = &var;

    cout << var << endl;
    // Outputs 50 (the value of var)

    cout << p << endl;
    // Outputs 0x29fee8 (var's memory location)

    cout << *p << endl;
    /* Outputs 50 (the value of the variable
     stored in the pointer p) */

    return 0;
}
```

```
50
0x7fff040c0da4
50
```

Dereferencing

- **The dereference operator (*) is basically an alias for the variable the pointer points to.**

```
int x = 5;
int *p = &x;

x = x + 4;
x = *p + 4;
*p = *p + 4;
```

As **p** is pointing to the variable **x**, dereferencing the pointer (***p**) is representing exactly the same as the variable **x**.

Pointers

Note that the `*` sign can be confusing here, as it does two different things in our code:

- When used in declaration (string* ptr), it creates a **pointer variable**.
- When not used in declaration, it act as a **dereference operator**.

The reference and dereference operators are thus complementary:

& is the address-of operator, and can be read simply as "address of"

* is the dereference operator, and can be read as "value pointed to by"

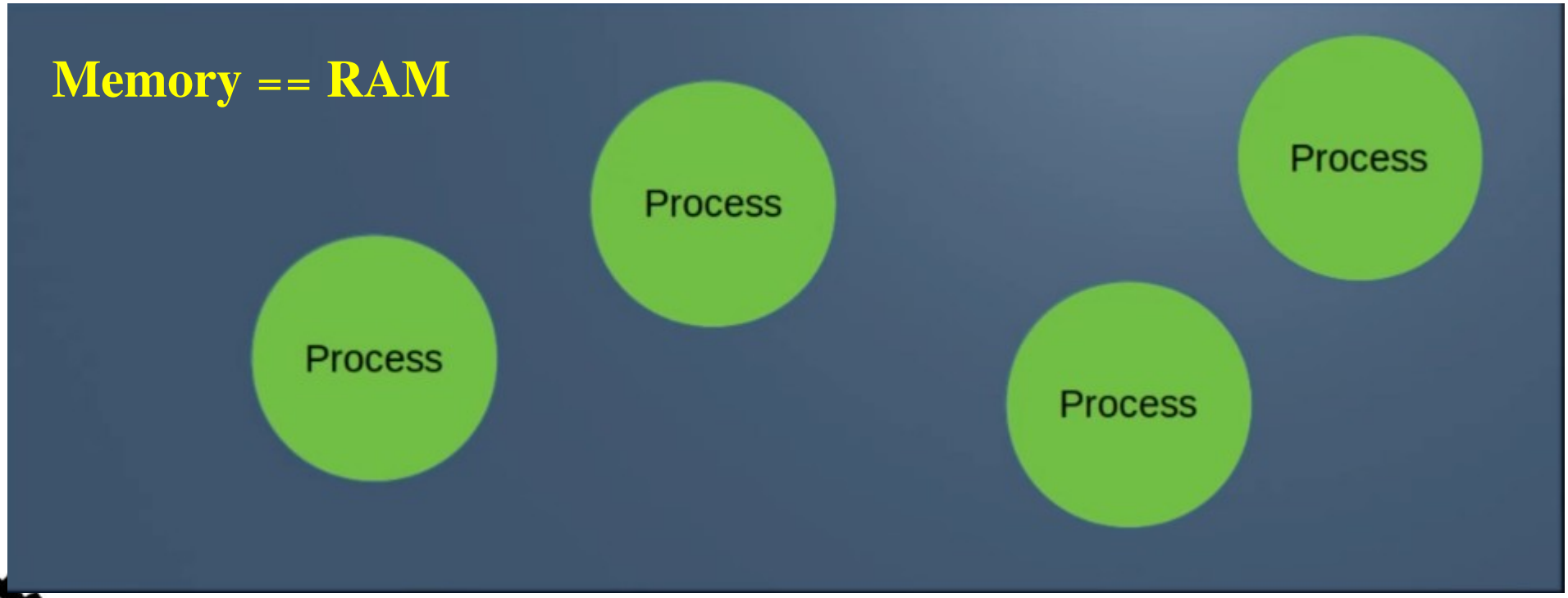
Pointers

```cpp
#include <iostream>
#include <string>
using namespace std;
int  main()
{
    string  food = "Pizza";
    string *ptr  = &food;

    // Output the value of food
    cout << food << "\n";

    // Output the memory address of food
    cout << &food << "\n";

    // Access the memory address of food and output its value
    cout << *ptr << "\n";

    // Change the value of the pointer
    *ptr = "Hamburger";

    // Output the new value of the pointer
    cout << *ptr << "\n";
    // Output the new value of the food variable
    cout << food << "\n";

    return 0;
}
```

Static & Dynamic Memory

- In a C++ program, memory is divided into two parts:
- The stack: All of your local variables take up memory from the stack.
- The heap: Unused program memory that can be used when the program runs to dynamically allocate the memory.
- Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.
- You can allocate memory at run time within the heap for the variable of a given type using the new operator, which returns the address of the space allocated.

Static & Dynamic Memory

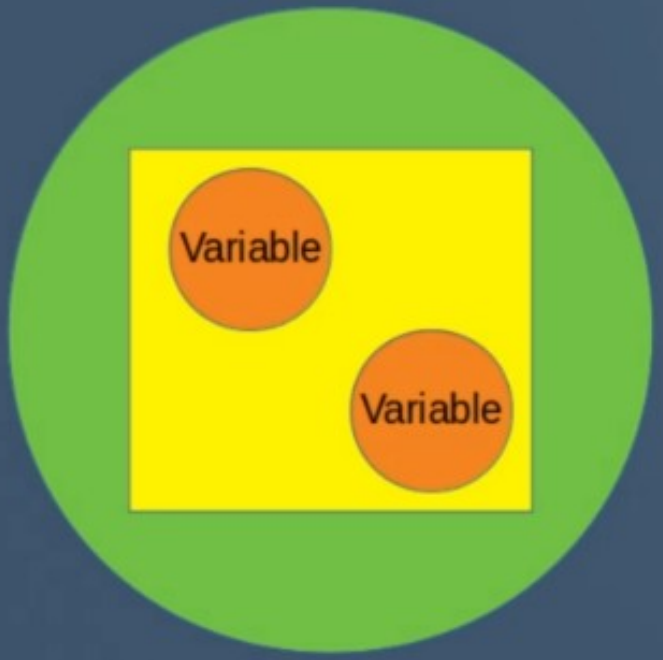


Static & Dynamic Memory

The stack is automatic

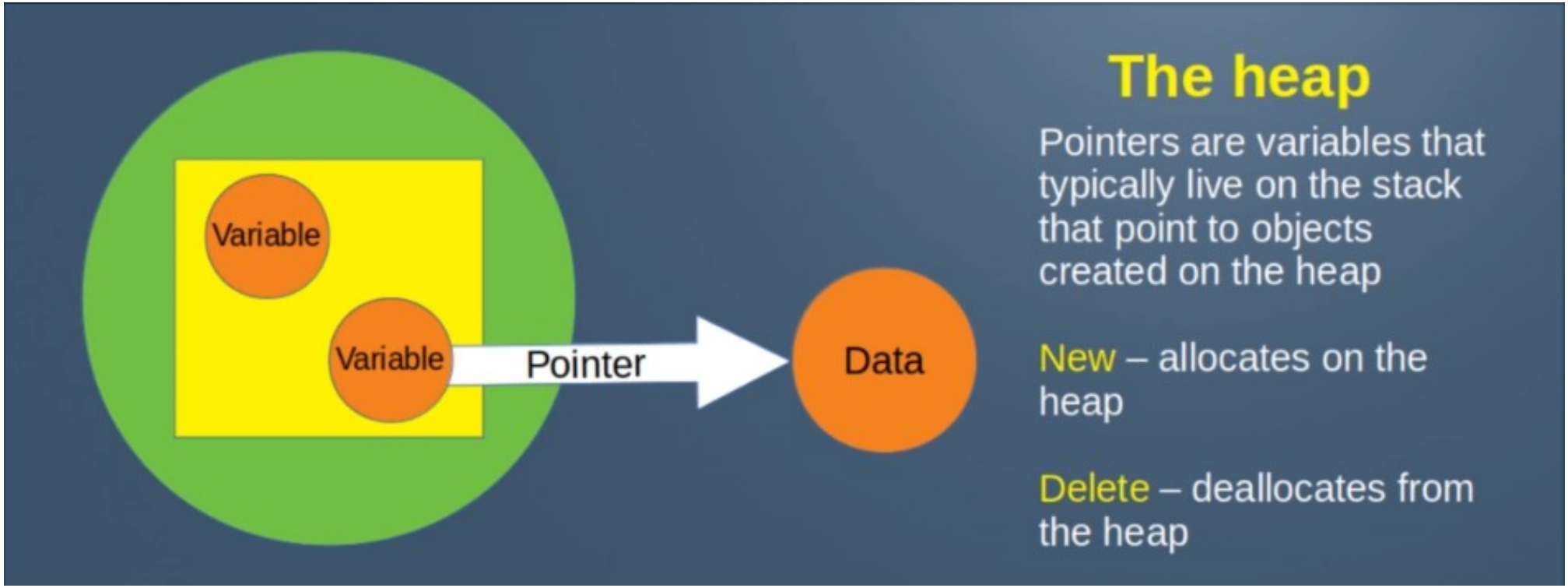
- Appears “inside” the process
- Allocates or deallocates the memory automatically
- Considered a “safe” area
- Limited in space by the operating system

Static & Dynamic Memory

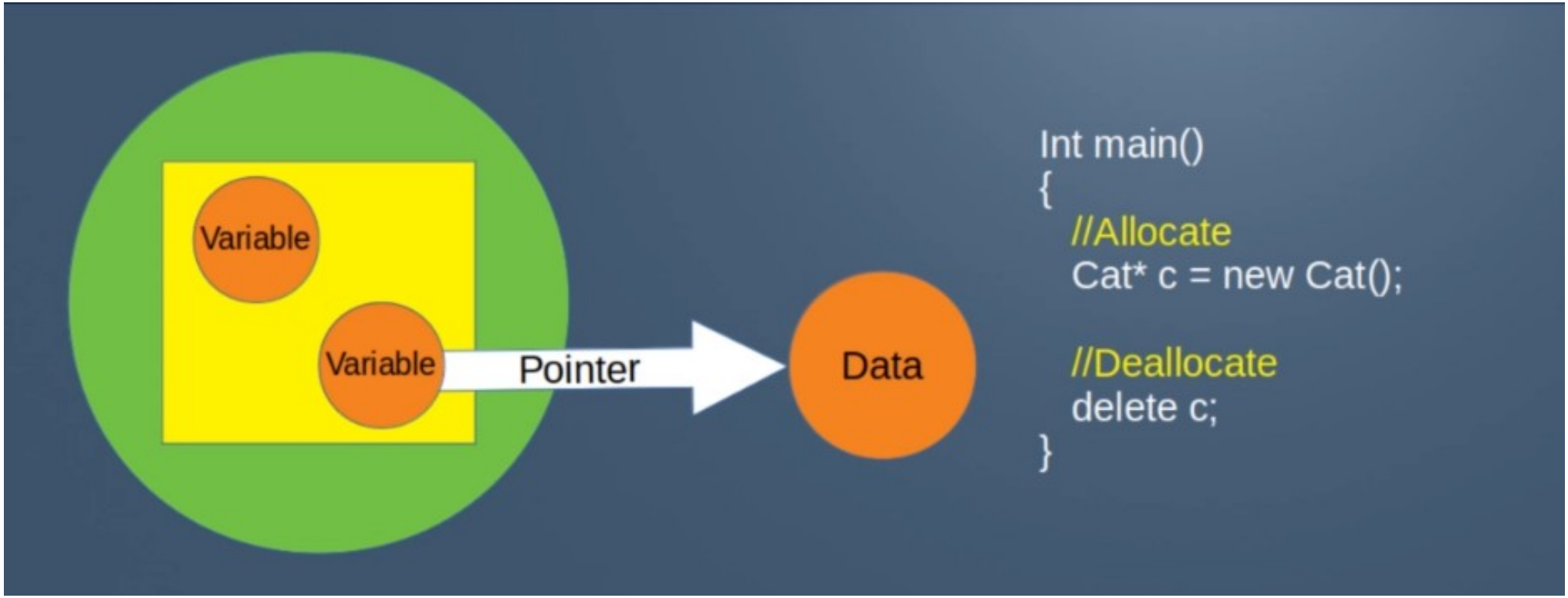



```
int main()
{
   // All these variables get memory
   // allocated on stack
   int a;
   int b[10];
   int n = 20;
   int c[n];
}
```

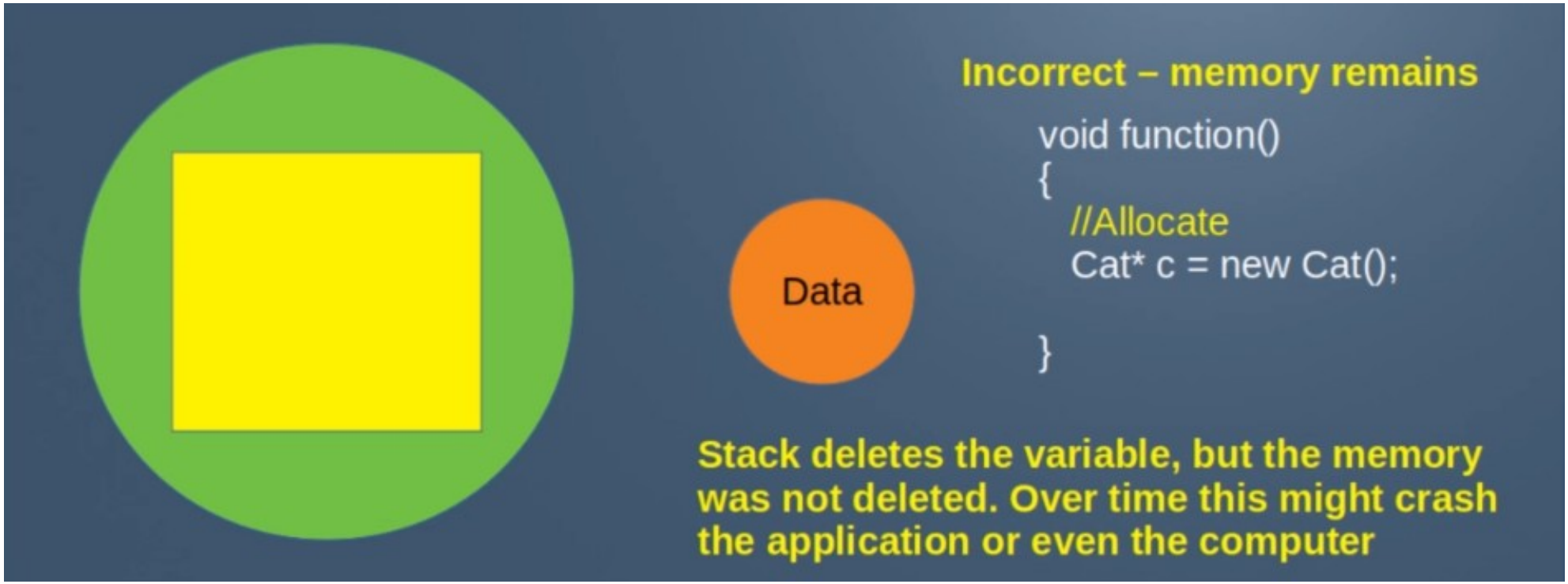
Static & Dynamic Memory



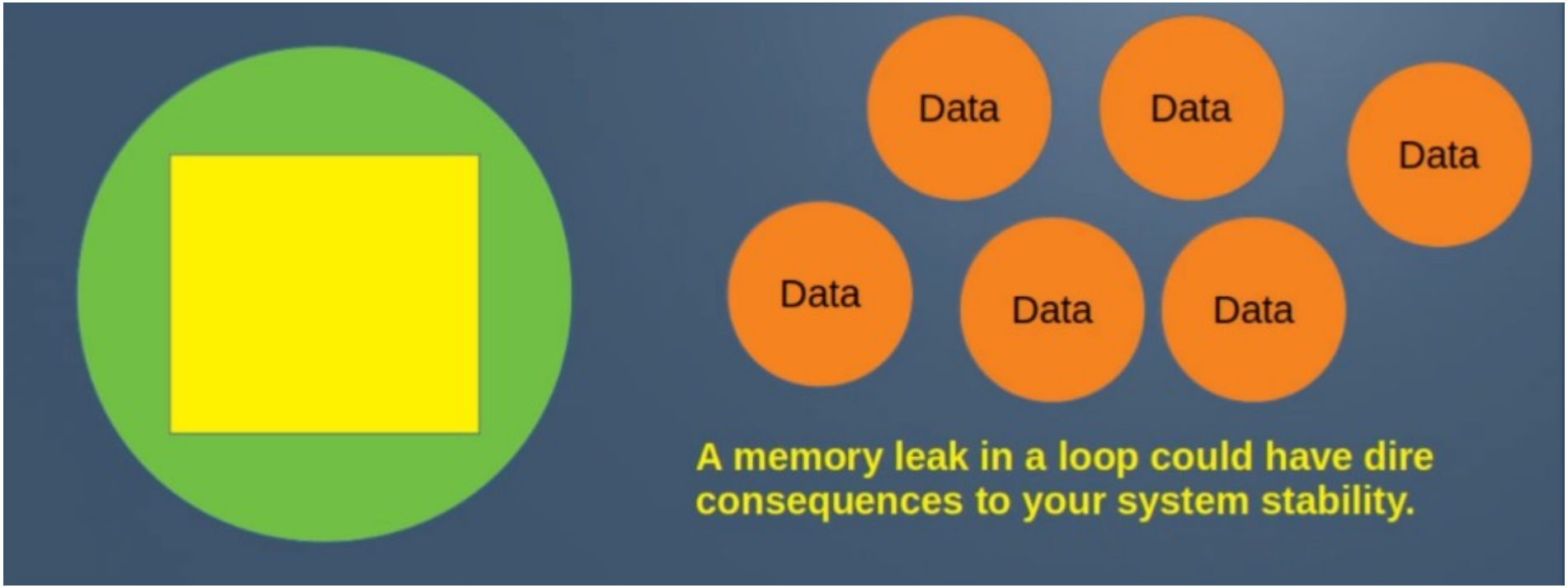
Static & Dynamic Memory



Static & Dynamic Memory



Memory leaks



Dynamic Memory

- **The allocated address can be stored in a pointer, which can then be dereferenced to access the variable**

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

The pointer **p** is stored in the **stack** as a local variable, and holds the **heap**'s allocated address as its value. The value of 5 is stored at that address in the heap.

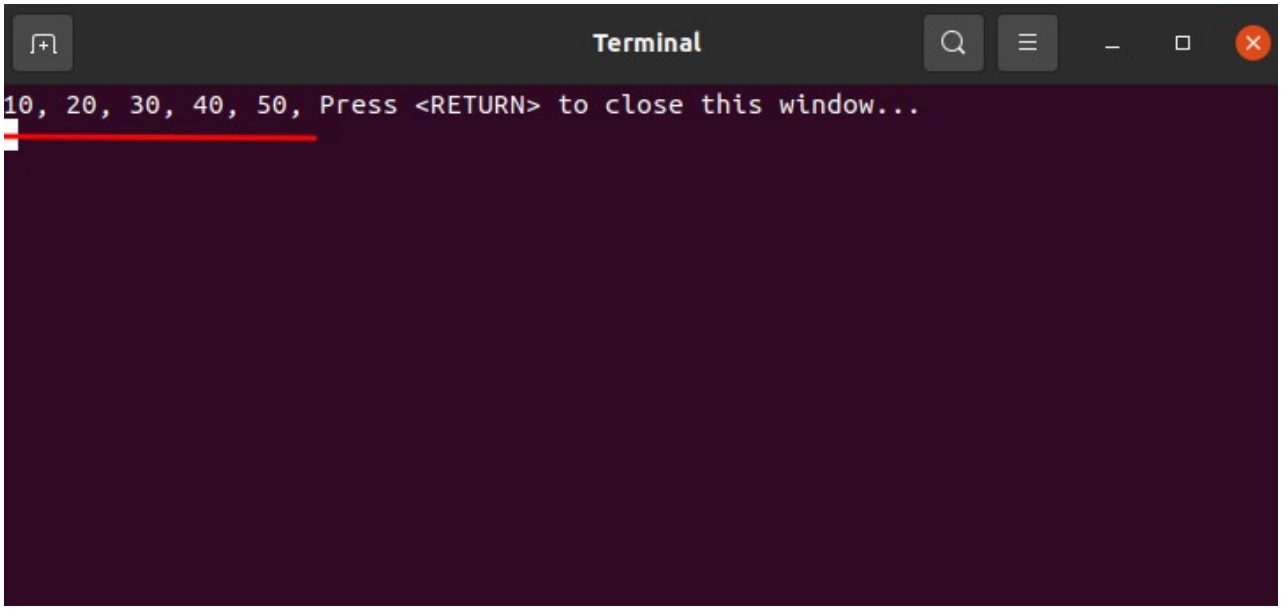
Static & Dynamic Memory

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

int main()
{
    int *p = new int; // request memory
    *p = 5; // store value

    cout << *p << endl; // use value

    delete p; // free up the memory

    return 0;
}
```

Forgetting to free up memory that has been allocated with the **new** keyword will result in memory leaks, because that memory will stay allocated until the program shuts down.

mixes arrays and pointers

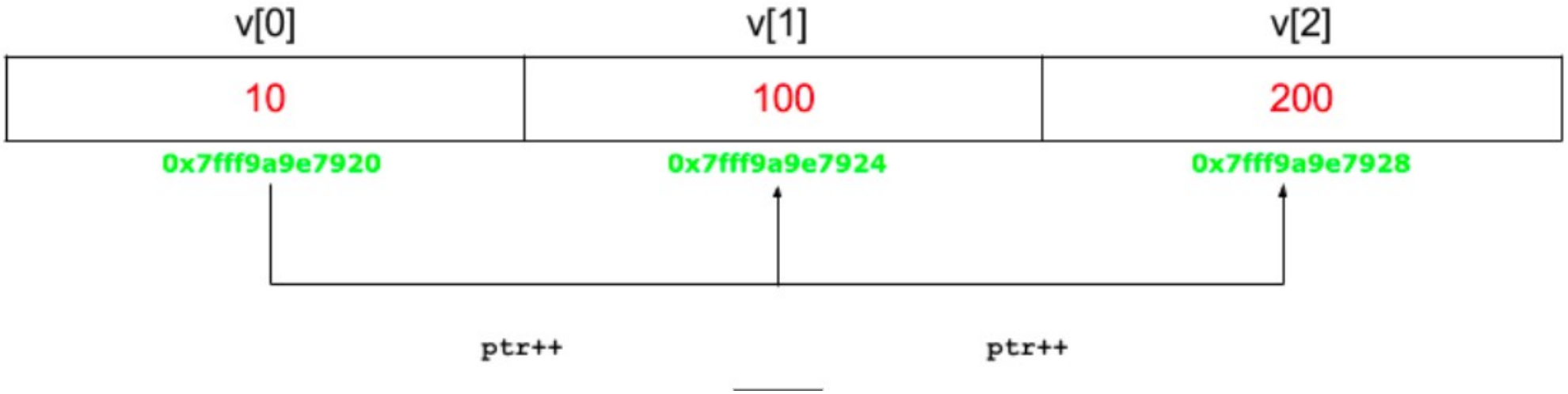
```cpp
#include <iostream>
using namespace std;

int  main()
{
    int  numbers[5];
    int *p;

    p  = numbers;
    *p = 10;
    p++;
    *p       = 20;
    p        = &numbers[2];
    *p       = 30;
    p        = numbers + 3;
    *p       = 40;
    p        = numbers;
    *(p + 4) = 50;

    for (int n = 0; n < 5; n++)
    {
        cout << numbers[n] << ", ";
    }

    return 0;
}
```



arrays and pointers

- **Pointers and arrays support the same set of operations, with the same meaning for both. The main difference being that pointers can be assigned new addresses, while arrays cannot.**
- **In the chapter about arrays, brackets ([]) were explained as specifying the index of an element of the array. Well, in fact these brackets are a dereferencing operator known as offset operator. They dereference the variable they follow just as * does, but they also add the number between brackets to the address being dereferenced.**

arrays and pointers



arrays and pointers

```
int nums[2][3]  =  { { 16, 18, 20 }, { 25, 26, 27 } };
```

In general, nums[i][j] is equivalent to *(*(nums+i)+j)

Pointer Notation	Array Notation	Value
*(*nums)	nums[0][0]	16
*(*nums+1)	nums[0][1]	18
*(*nums+2)	nums[0][2]	20
((nums + 1))	nums[1][0]	25
((nums + 1)+1)	nums[1][1]	26
((nums + 1)+2)	nums[1][2]	27

Dangling Pointers

- The **delete** operator frees up the memory allocated for the variable, but does not delete the pointer itself, as the pointer is stored on the stack.
- Pointers that are left pointing to **non-existent** memory locations are called **dangling pointers**.

Dangling Pointers

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6       int *p = new int; // request memory
7       *p = 5; // store value
8
9       delete p; // free up the memory
10      // now p is a dangling pointer
11
12      p = new int; // reuse for a new address
13
14      return 0;
15  }
```

Dangling Pointers

The **NULL** pointer is a constant with a value of zero that is defined in several of the standard libraries, including iostream.
It's a good practice to assign NULL to a pointer variable when you declare it, in case you do not have exact address to be assigned. A pointer assigned NULL is called a **null pointer**. For example: int *ptr = NULL;

Dynamic Memory

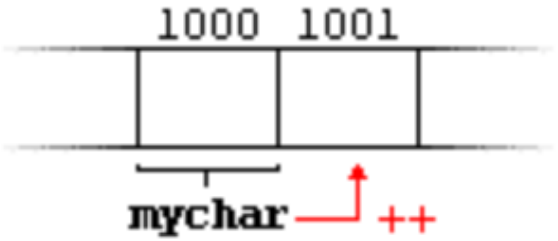
- **Dynamic memory can also be allocated for arrays.**

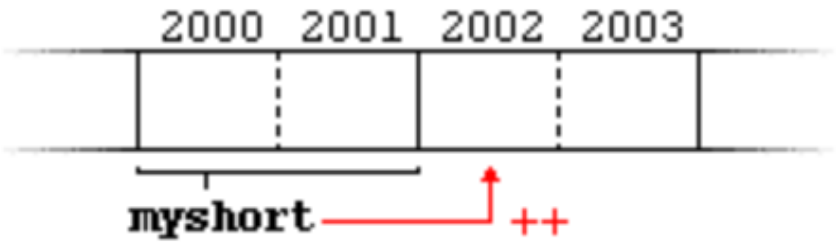
```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6       int *p = NULL; // Pointer initialized with null
7       p = new int[20]; // Request memory
8       delete [] p; // Delete array pointed to by p
9
10      return 0;
11  }
```

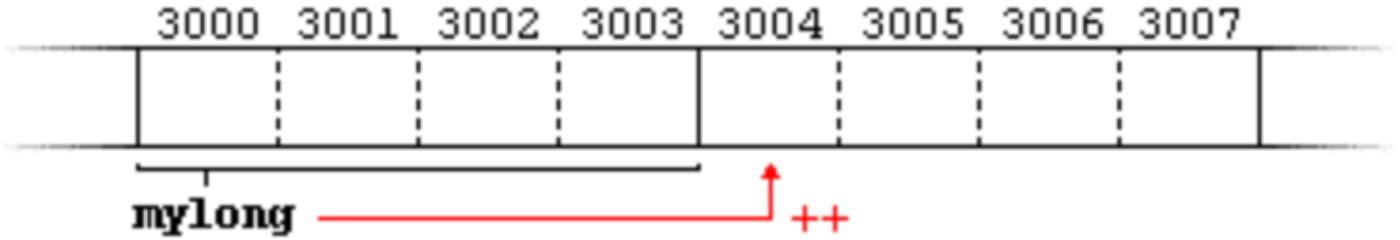
Dynamic Memory

- **Dynamic memory** allocation is useful in many situations, such as when your program **depends on input**. As an example, when your program needs to read an image file, it doesn't know in advance the size of the image file and the memory necessary to store the image.

Pointer arithmetics







Pointer arithmetic

```
1  #include <iostream>
2  using namespace std;
3
4  int  main()
5  {
6      char  *mychar  = nullptr;
7      short *myshort = nullptr;
8      long  *mylong  = nullptr;
9
10     ++mychar;
11     ++myshort;
12     ++mylong;
13
14     mychar  = mychar + 1;
15     myshort = myshort + 1;
16     mylong  = mylong + 1;
17
18     return 0;
19 }
20
```

تمرین های تکمیلی

```cpp
#include <iostream>

using namespace std;

int  main()
{
    int  numbers[2][2];

    *(*(numbers))          = 40;
    *(*(numbers) + 1)      = 50;
    *(*(numbers + 1))      = 60;
    *(*(numbers + 1) + 1) = 70;

    for (int i = 0; i ≤ size(numbers); i++)     △ Comparisc
    {
        for (int j = 0; j ≤ size(numbers); j++)     △ Compa
        {
            cout << " numbers " << i << "," << j << " = "
                 << numbers[i][j] << endl;
        }
    }

    return 0;
}
```

تمرین های تکمیلی

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

int  main()
{
    int  arr[3] = { 10, 20, 30 };

    for (int i = 0; i < 3; i++)
    {
        cout << *(arr + i) << endl;
    }

    return 0;
}
```

تمرین های تکمیلی

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

int  main()
{
    int  arr[3] = { 10, 20, 30 };

    for (int i = 0; i < 3; i++)
    {
        cout << *(&arr[i]) << endl;
    }

    return 0;
}

```

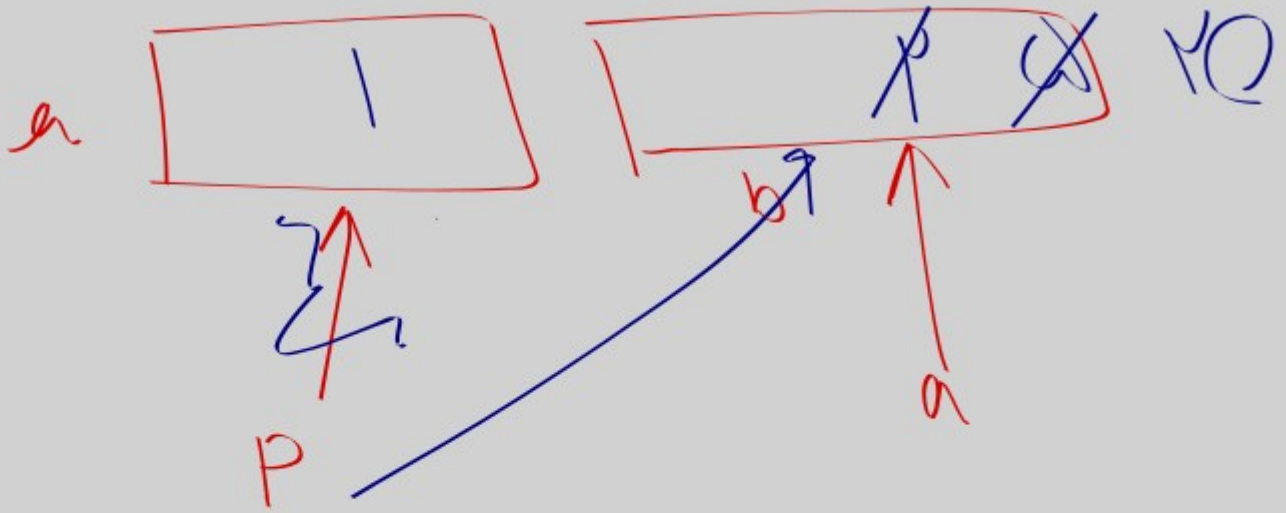
تمرین های تکمیلی

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

int  main()
{
    //! [2]
    int  x = 6, *p1, **p2;

    p2  = &p1;
    *p2 = &x;

    **p2 = *p1 - 1;

    //! [1]
    int  x2 = 6;
    int *p  = &x;

    *p = *p - 1;

    cout << "x = " << x << endl;

    return 0;
}
```

تمرین های تکمیلی

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

int  main()
{
    int  a = 1;
    int  b = 2;
    int *p, *q;

    p = &a;
    q = &b;

    p = q;

    *p = 10;
    *q = 20;

    cout << "a = " << a << "b = " << b << endl;

    return 0;
}

```

تمرین های تکمیلی



$P \neq Q$

$P = Q$

Question ?