RC OF LAST WEEK

VE280

ZHOUHONGKUAN

Very Basic Concepts

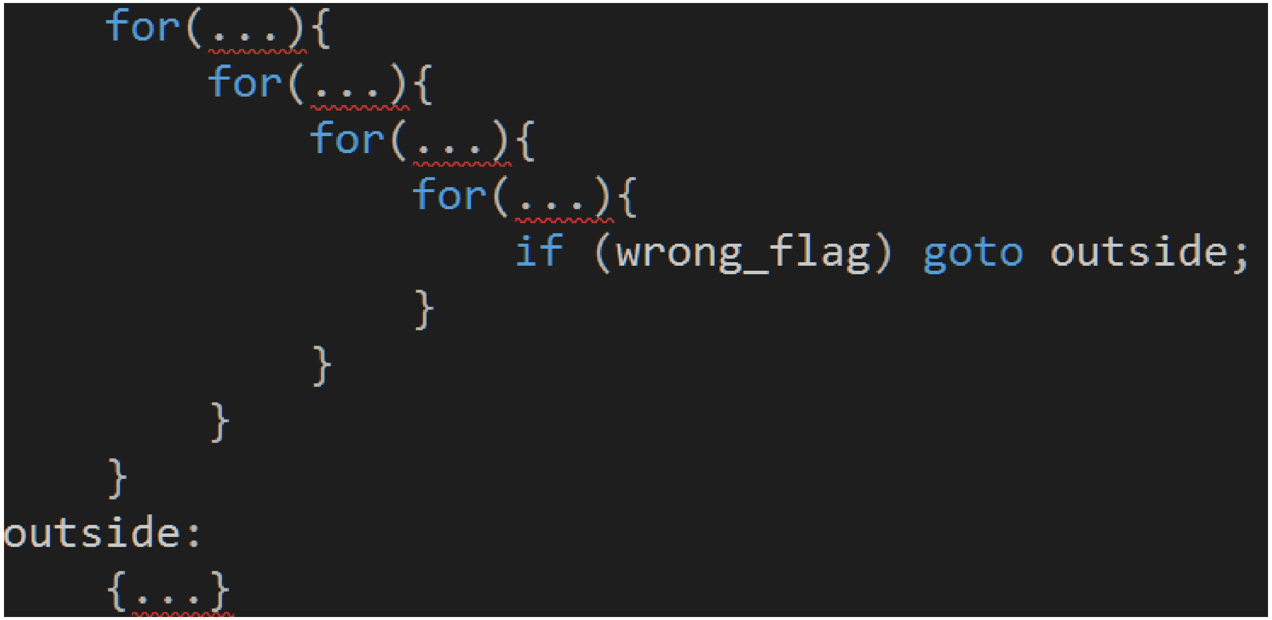
- Variables
- Built-in data types, e.g., `int`, `double`, etc.
- Input and output, e.g., `cin`, `cout`.
- Operators
  - Arithmetic: +, -, *, etc.
  - Comparison: <, >, ==, etc.
  - x++ versus ++x
- Flow of controls
  - Branch: if/else, switch/case
  - Loop: while, for, etc.

About goto

- Be CAREFUL when using goto!!!

  1. Is the function correct?
  2. Will it cause memory leak?
  3. Will it increase readability?

- Good example



```
    for(...){
        for(...){
            for(...){
                for(...){
                    if (wrong_flag) goto outside;
                }
            }
        }
    }
outside:
    {...}
```

Array, Struct and Enum

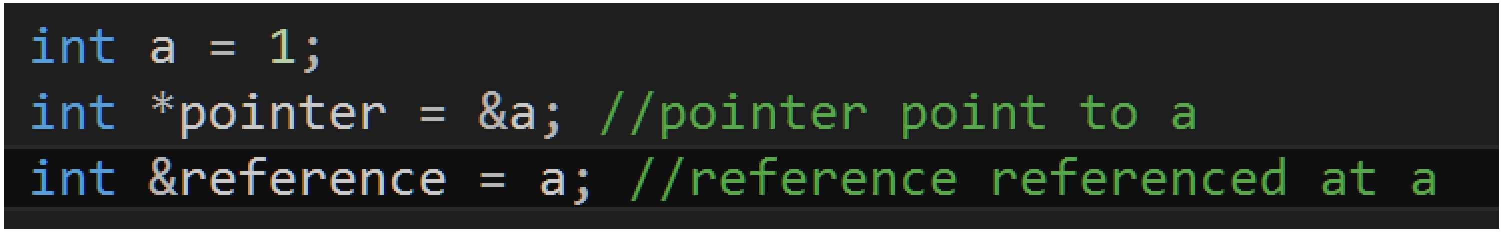
- Array:

  An array is a fixed-sized, indexed data type that stores a collection of items, all of the same type.
- Struct:

  A struct can hold variance variable.
- Enum:

  Used to categorize data
- All can be passed as arguments to a function.

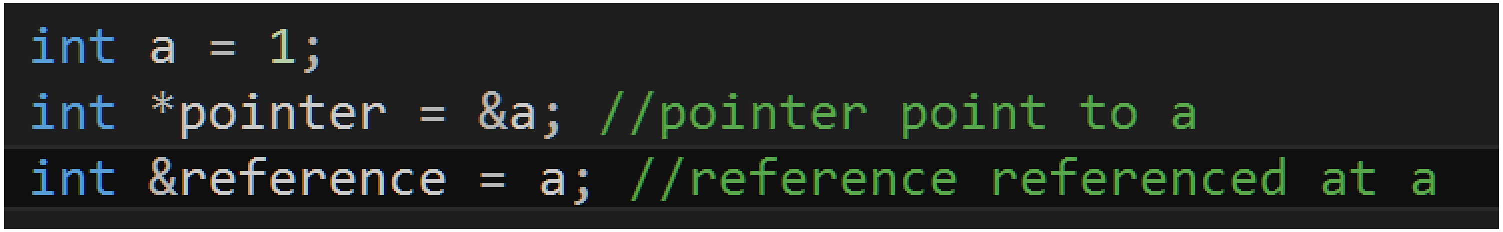
Pointers & Ref



```
int a = 1;
int *pointer = &a; //pointer point to a
int &reference = a; //reference referenced at a
```

- Any differences between pointers and references?
  - Pointers require some extra syntax at calling time (&), in the argument list (*), and with each use (*); references only require extra syntax in the argument list (&).
  - You can change the object to which a pointer points, but you cannot change the object to which a reference refers.
    - In this sense, pointer is **more flexible**

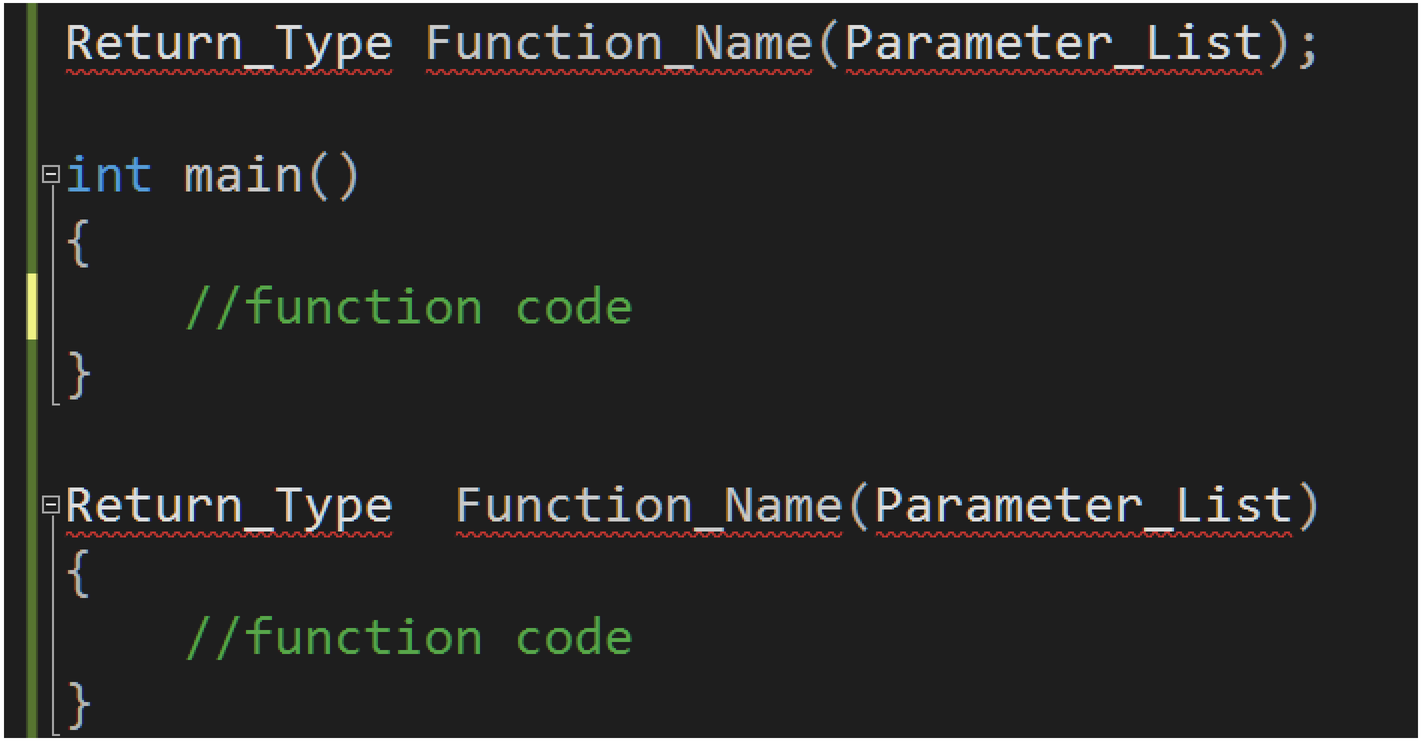
Pointers & Ref



```
int a = 1;
int *pointer = &a; //pointer point to a
int &reference = a; //reference referenced at a
```

- You can regard Ref as const pointers. Both make two variance share a same room in memory.
- **(The only difference is that const pointer can point to NULL while Ref cannot.)**

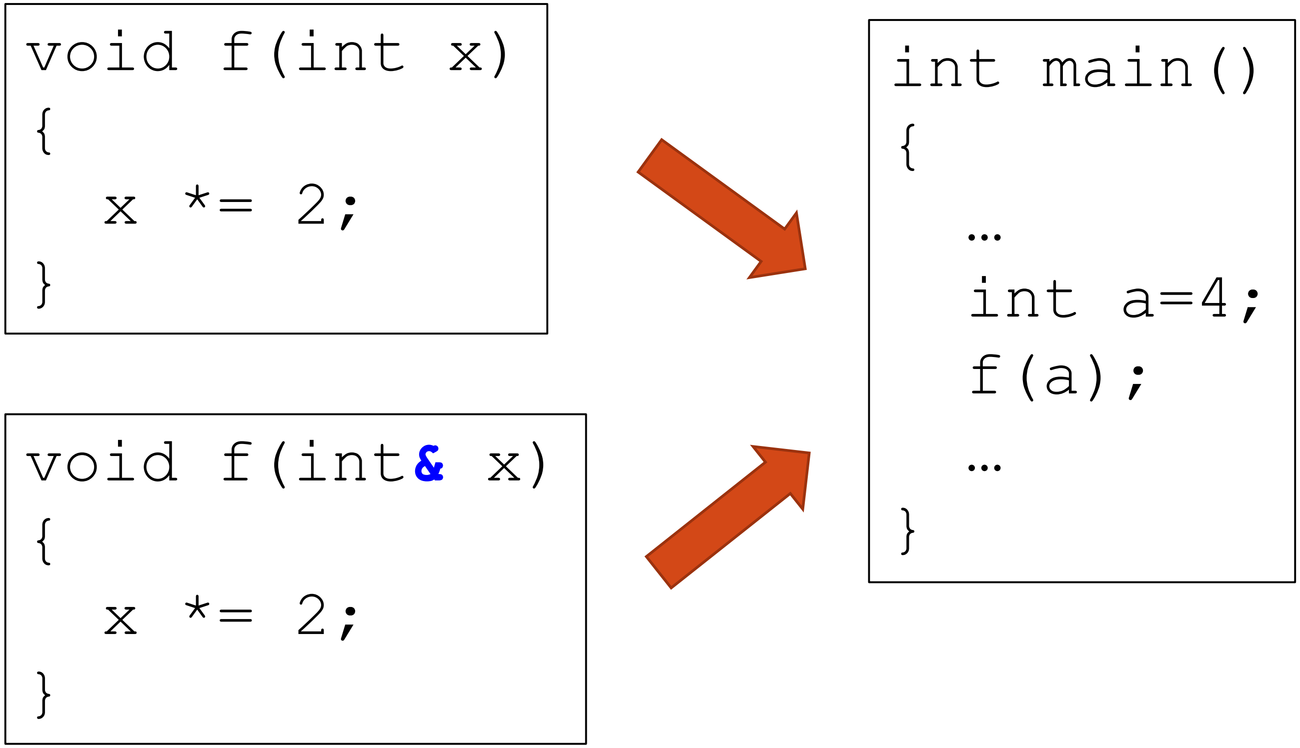
Function Declarations vs. Definitions



```
Return_Type Function_Name(Parameter_List);

int main()
{
    //function code
}

Return_Type  Function_Name(Parameter_List)
{
    //function code
}
```

Function Call Mechanisms

- Two mechanisms:
  - Call-by-Value
  - Call-by-Reference



```
void f(int x)
{
   x *= 2;
}
```

```
void f(int& x)
{
   x *= 2;
}
```

```
int main()
{
   …
   int a=4;
   f(a);
   …
}
```

Function Call Mechanisms

```
void f(int x){ x *= 2;}
```

memory

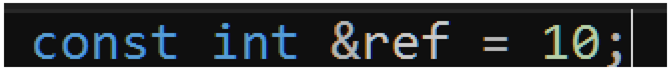
```
void f(int& x){  x *= 2;}
```

memory

Const Qualifer

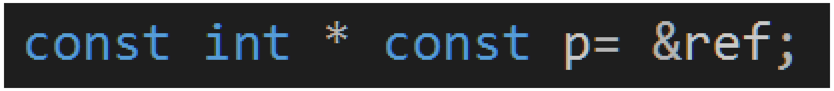
- Usually, constant is defined as a global variable.
- Property
  - Cannot be modified later on
  - Must be initialized when it is defined
  - Const Reference:

    Can be initiated to a rvalue:



```
const int &ref = 10;
```

  - Const Pointer:



```
const int * const p= &ref;
```

Practical Use of Pointer to const

Example

```
void strcpy(char *dest, const char *src)
   // src is a NULL-terminated string.
   // dest is big enough to hold a copy of src.
   // The function place a copy of src in dest.
   // src is not changed.
{ ... }
```

- Strictly speaking, we don’t **need** to include the `const` qualifier here since the comment promises that we won't modify the source string
- So, why include it?

Practical Use of Pointer to const

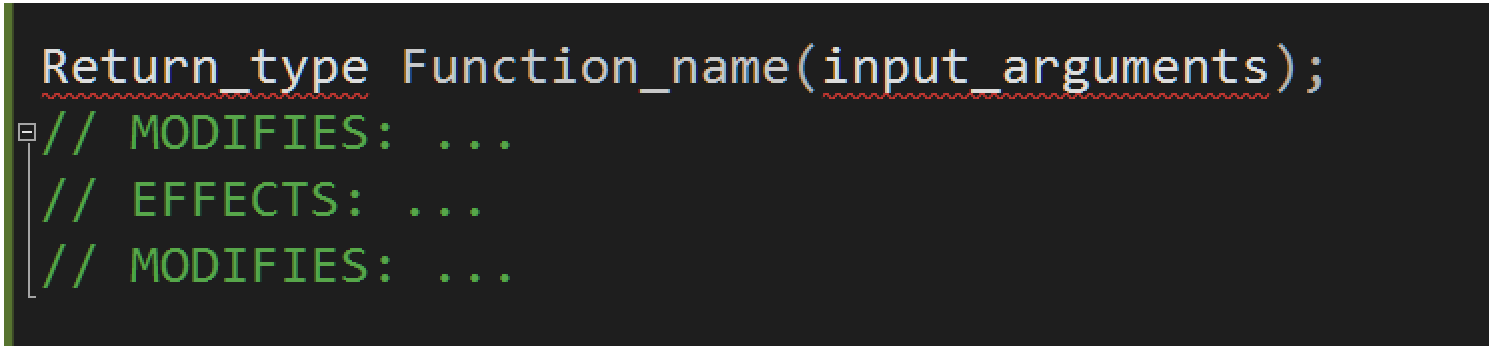
Example

- Why include `const`?

- Because once you add it, you CANNOT change `src`, even if you do so by mistake.
- Such a mistake will be caught by the **compiler**.
  - Bugs that are detected at compile time are among the easiest bugs to fix – those are the kinds of bugs we want.

- **General guideline**: Use `const` for things that are passed by reference, but won't be changed.

Procedural Abstraction



```
Return_type Function_name(input_arguments);
// MODIFIES: ...
// EFFECTS: ...
// MODIFIES: ...
```

- For the convenience of others as well as your self!

Recursion

- Recursion is a nice way to solve problems
  - “Recursive” just means “refers to itself”.
  - There is (at least) one “trivial” base or “stopping” case.
  - All other cases can be solved by first solving one smaller case, and then combining the solution with a simple step.
- Example: calculate factorial $n!$

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

```
int factorial (int n) {
// REQUIRES: n >= 0
// EFFECTS:  computes n!
  if (n == 0) return 1; // base case
  else return n*factorial(n-1); // recursive step
}
```

Recursive Helper Function

- Sometimes it is easier to find a recursive solution to a problem if you change the original problem slightly, and then solve that problem using a **recursive helper function**.

```
soln()
{
   …
   soln_helper();
   …
}
```

```
soln_helper()
{
   …
   soln_helper();
   …
}
```

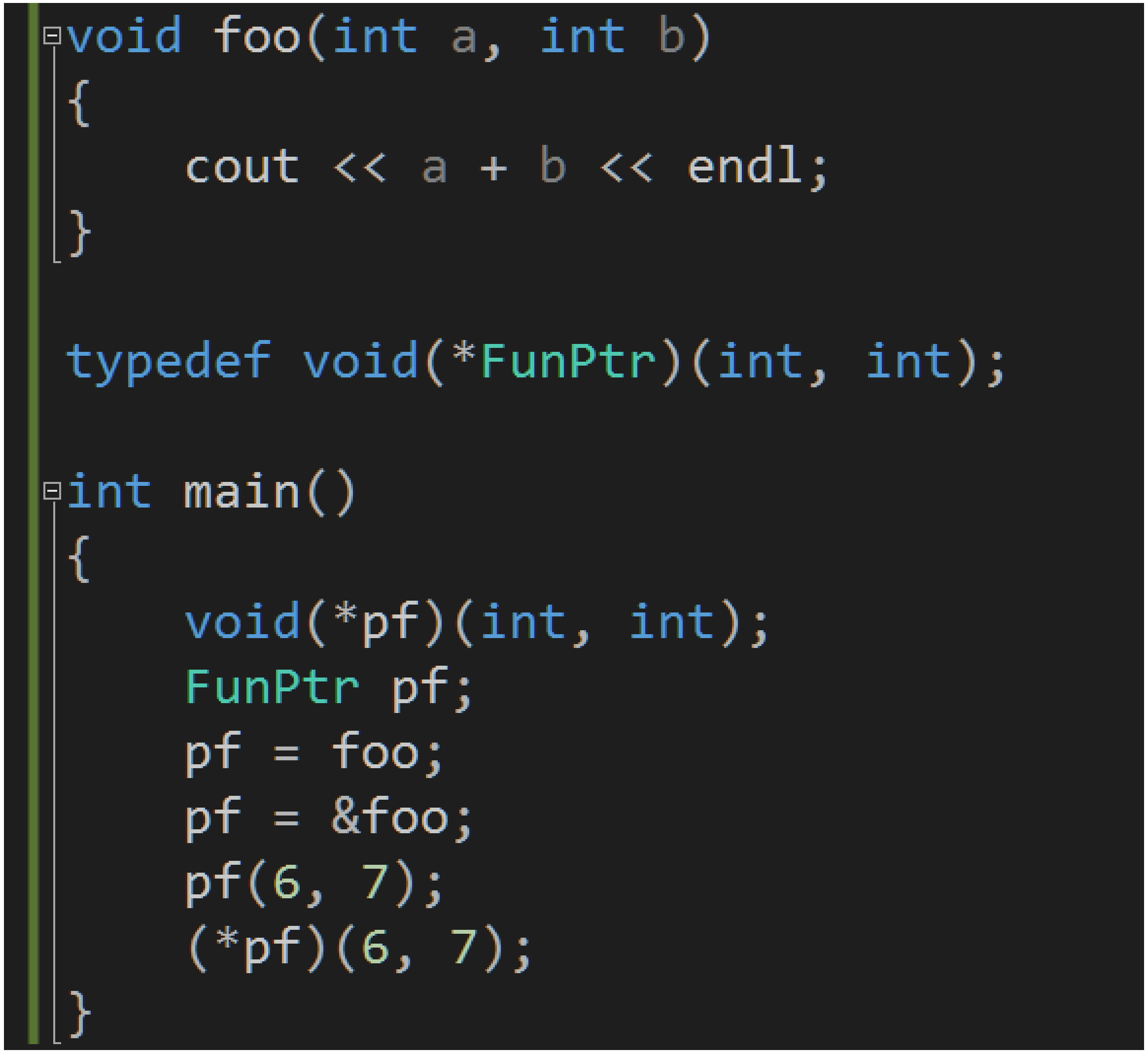
Function Pointers

Motivation

- If you were asked to write a function to add all the elements in a list, and another to multiply all the elements in a list, your functions would be almost exactly **the same**.

- Generally, function pointers are used when a function need another function as input argument or you want to call a function by its pointer instead of the declared function name.

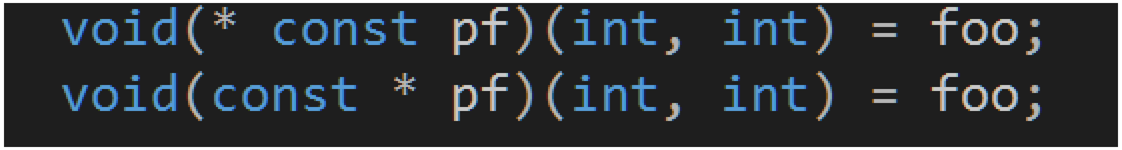
Function Pointers



```cpp
void foo(int a, int b)
{
    cout << a + b << endl;
}

typedef void(*FunPtr)(int, int);

int main()
{
    void(*pf)(int, int);
    FunPtr pf;
    pf = foo;
    pf = &foo;
    pf(6, 7);
    (*pf)(6, 7);
}
```

Function Pointers

- No const function pointer!

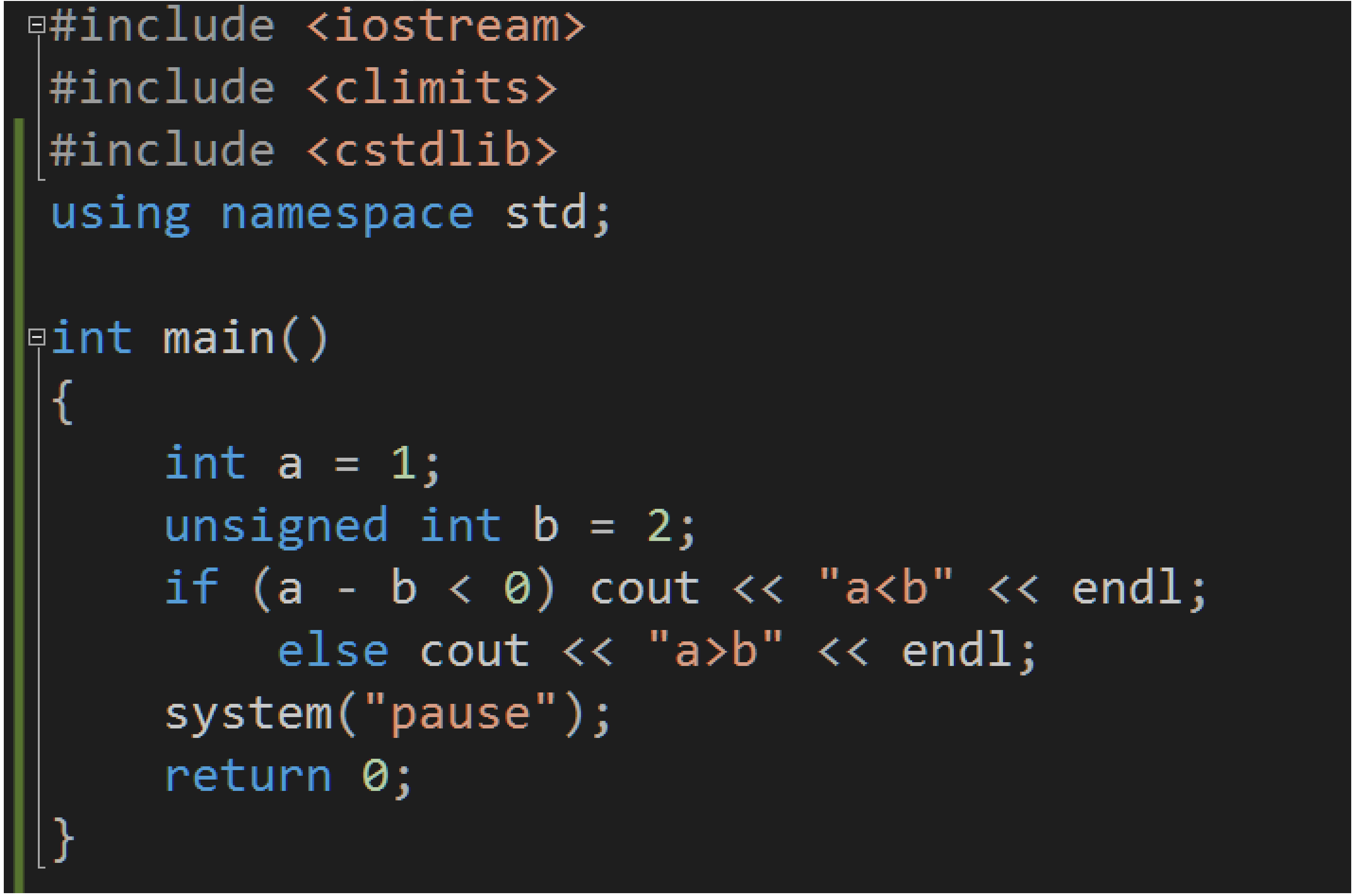


```
void(* const pf)(int, int) = foo;
void(const * pf)(int, int) = foo;
```

- A stand alone function is already const!

Class Exercise

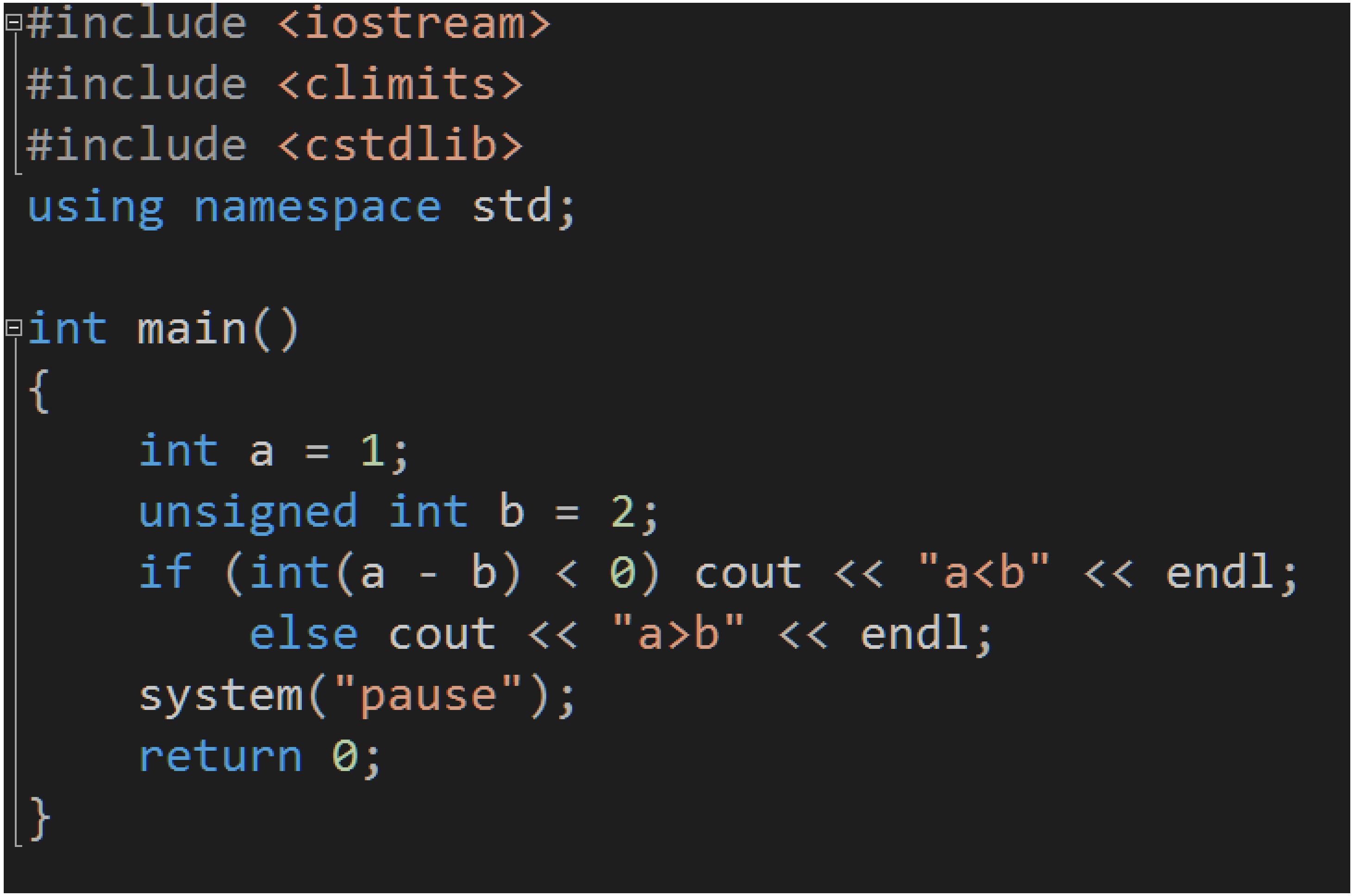
- What is the output?



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

int main()
{
    int a = 1;
    unsigned int b = 2;
    if (a - b < 0) cout << "a<b" << endl;
        else cout << "a>b" << endl;
    system("pause");
    return 0;
}
```

Class Exercise

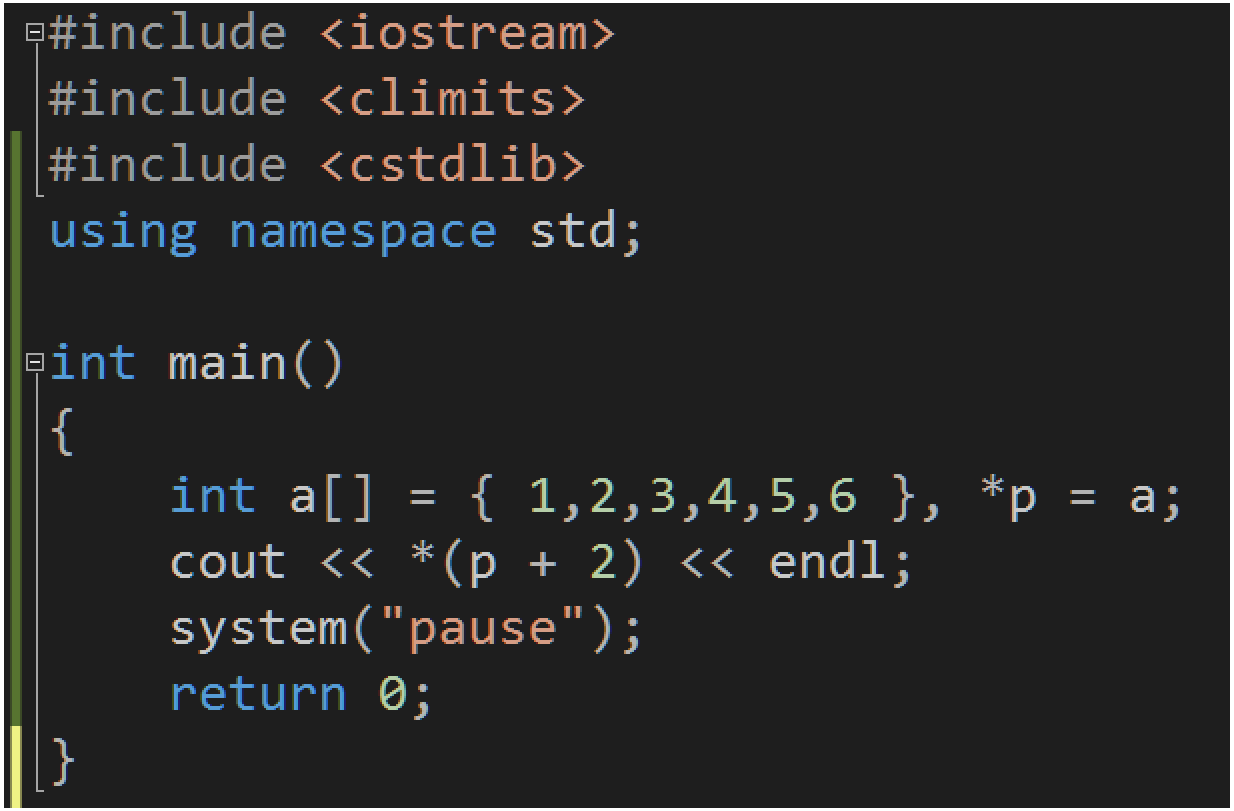
- What is the output?



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

int main()
{
    int a = 1;
    unsigned int b = 2;
    if (int(a - b) < 0) cout << "a<b" << endl;
        else cout << "a>b" << endl;
    system("pause");
    return 0;
}
```

Class Exercise

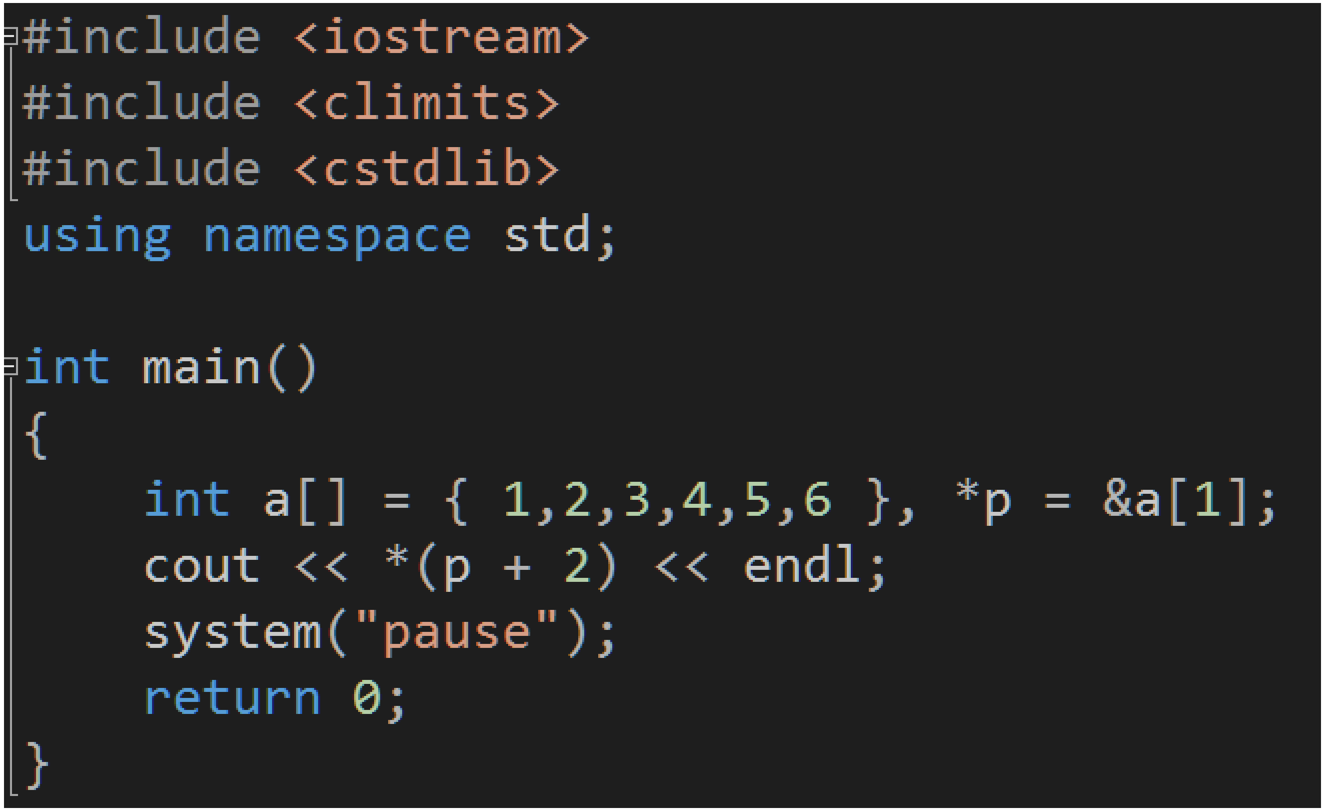
- What is the output?



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

int main()
{
    int a[] = { 1,2,3,4,5,6 }, *p = a;
    cout << *(p + 2) << endl;
    system("pause");
    return 0;
}
```

Class Exercise

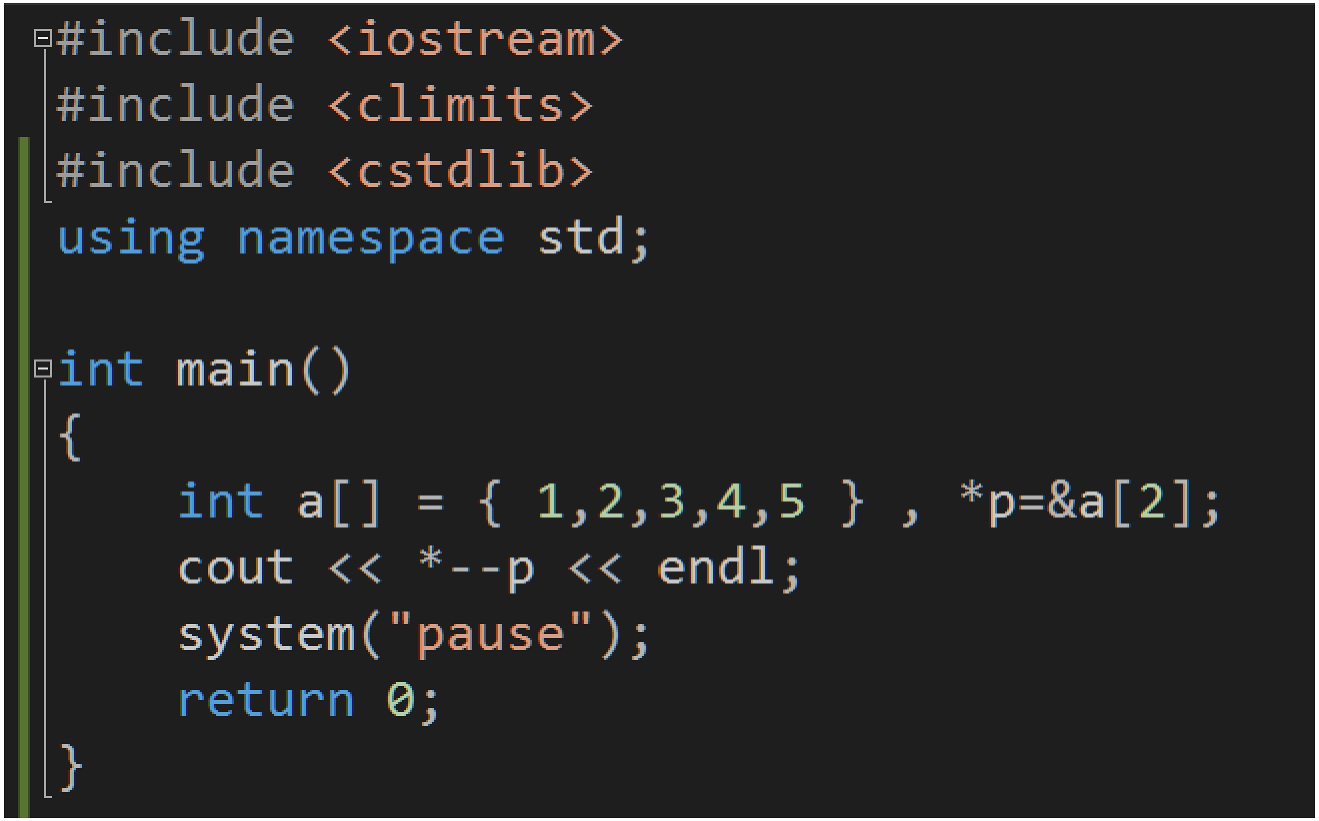
- What is the output?



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

int main()
{
    int a[] = { 1,2,3,4,5,6 }, *p = &a[1];
    cout << *(p + 2) << endl;
    system("pause");
    return 0;
}
```

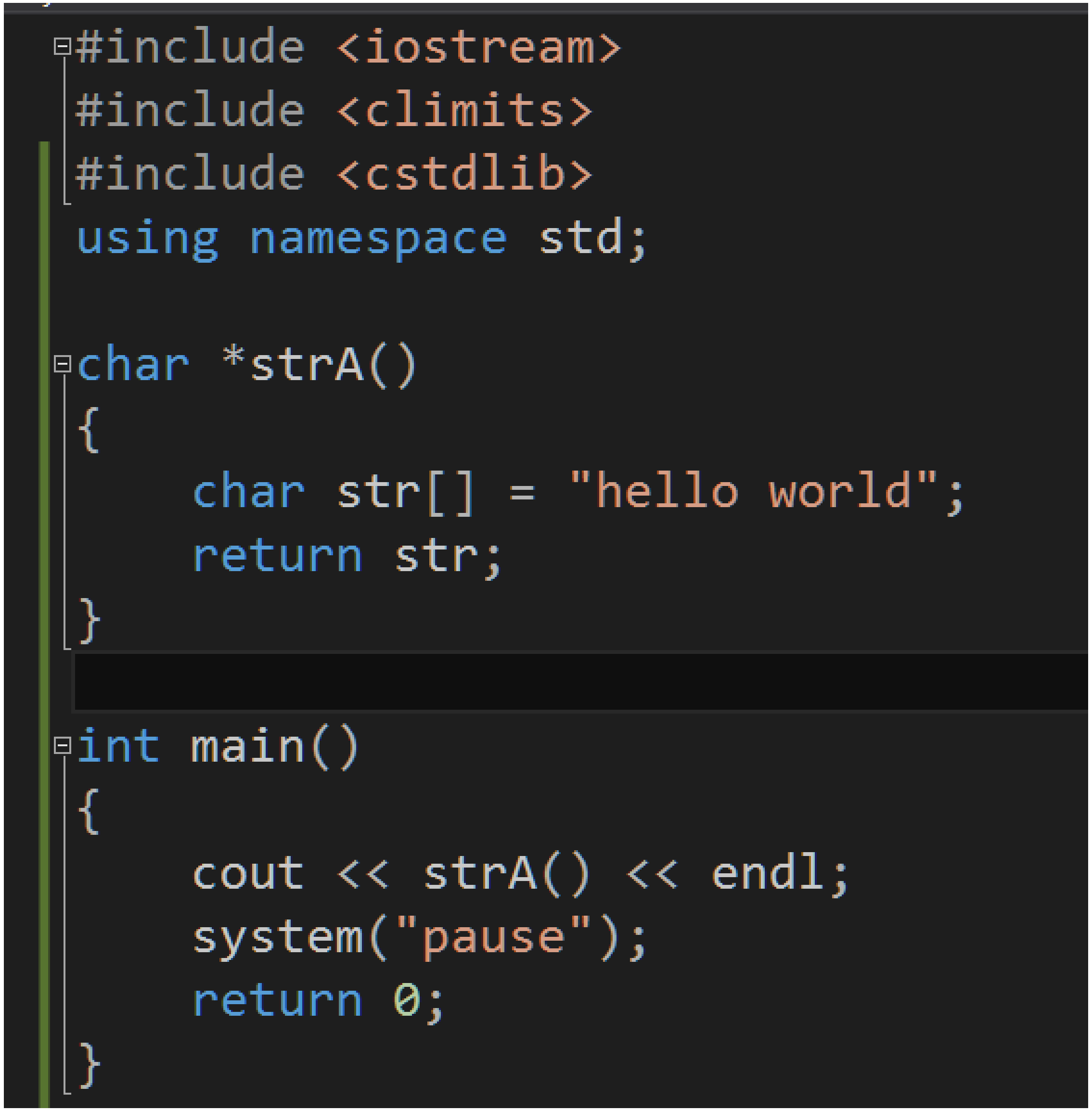
Class Exercise

- What is the output?



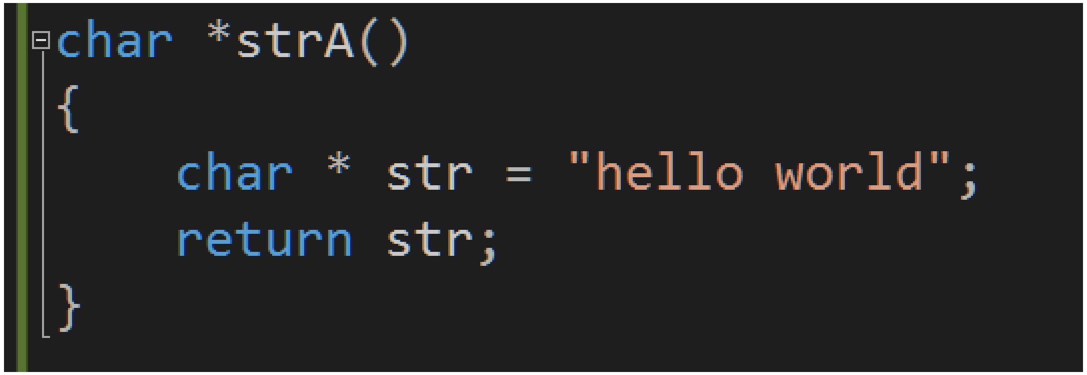
```
#include <iostream>
#include <climits>
#include <cstdlib>
using namespace std;

int main()
{
    int a[] = { 1,2,3,4,5 } , *p=&a[2];
    cout << *--p << endl;
    system("pause");
    return 0;
}
```

- What is the output?

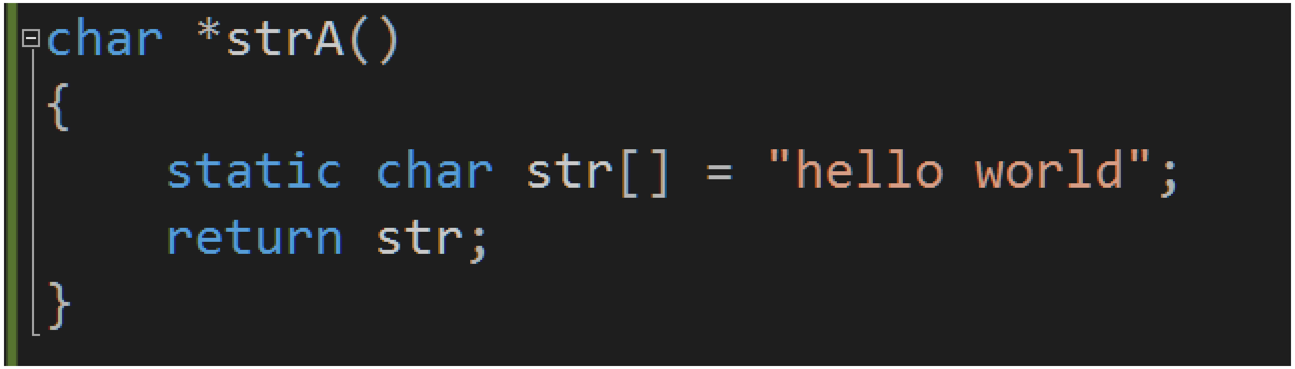


```cpp
#include <iostream>
#include <climits>
#include <cstdlib>
using namespace std;

char *strA()
{
    char str[] = "hello world";
    return str;
}

int main()
{
    cout << strA() << endl;
    system("pause");
    return 0;
}
```

- How to solve the problem?

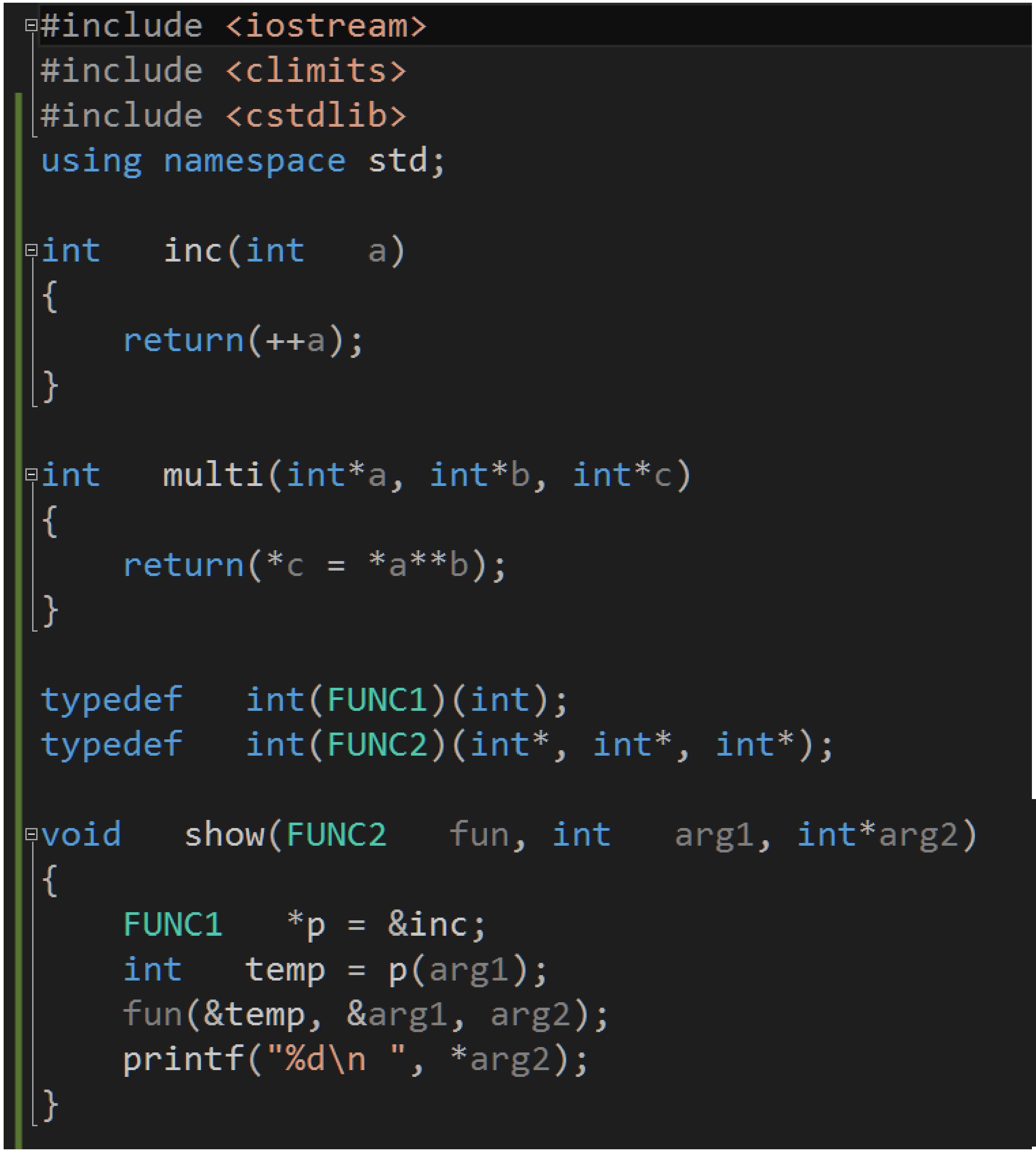


```
char *strA()
{
    char * str = "hello world";
    return str;
}
```

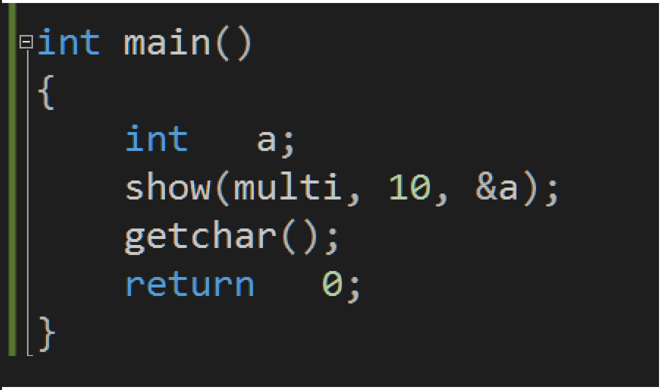


```
char *strA()
{
    static char str[] = "hello world";
    return str;
}
```

- What is the output?



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

int   inc(int   a)
{
    return(++a);
}

int   multi(int*a, int*b, int*c)
{
    return(*c = *a**b);
}

typedef   int(FUNC1)(int);
typedef   int(FUNC2)(int*, int*, int*);

void   show(FUNC2   fun, int   arg1, int*arg2)
{
    FUNC1   *p = &inc;
    int   temp = p(arg1);
    fun(&temp, &arg1, arg2);
    printf("%d\n ", *arg2);
}
```



```
int main()
{
    int   a;
    show(multi, 10, &a);
    getchar();
    return   0;
}
```

- If you don’t know something about C++, first try it by your hand!