# ESc 101: Fundamentals of Computing

Lecture 19

Feb 15, 2010

# Outline

# scanf

- `scanf` is a function that reads values from keyboard.
- The number and type of values to be read is specified by the first argument of `scanf` which is a constant string like in `printf`.
- Since parameters are passed in C by value, how are the values read and stored in parameter variables by `scanf`?
- For this, we need to understand pointers.

# scanf

- `scanf` is a function that reads values from keyboard.
- The number and type of values to be read is specified by the first argument of `scanf` which is a constant string like in `printf`.
- Since parameters are passed in C by value, how are the values read and stored in parameter variables by `scanf`?
- For this, we need to understand pointers.

# scanf

- `scanf` is a function that reads values from keyboard.
- The number and type of values to be read is specified by the first argument of `scanf` which is a constant string like in `printf`.
- Since parameters are passed in C by value, how are the values read and stored in parameter variables by `scanf`?
- For this, we need to understand pointers.

# scanf

- `scanf` is a function that reads values from keyboard.
- The number and type of values to be read is specified by the first argument of `scanf` which is a constant string like in `printf`.
- Since parameters are passed in C by value, how are the values read and stored in parameter variables by `scanf`?
- For this, we need to understand pointers.

# Outline

# Memory Address

- A variable is a name for a memory location.
- Recall that this is done for ease of use, as representing a memory location by its address is tricky.
- However, every memory location does have an address:
  - It is a number uniquely identifying the memory location.

# Memory Address

- A variable is a name for a memory location.
- Recall that this is done for ease of use, as representing a memory location by its address is tricky.
- However, every memory location does have an address:
  - It is a number uniquely identifying the memory location.

# Recall: Call-by-Value

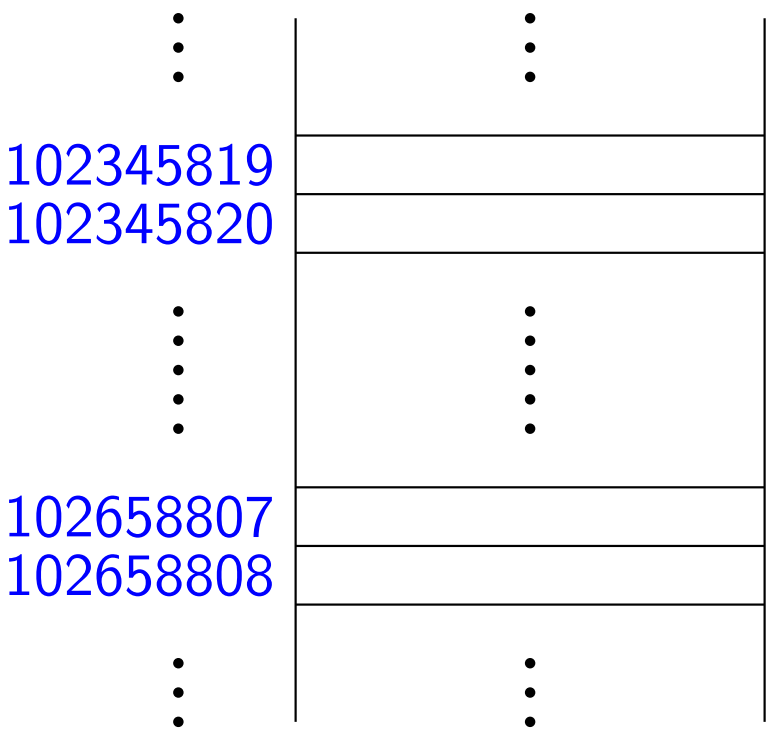
In a function call, only the value stored in an argument is passed to the corresponding parameter of the function. Example:

```
void foo( int y )
{
    y = 20; /* function sets y to 20 */
}

main()
{
    int x;

    x = 10; /* x = 10 here */
    foo(x);
    printf("%d", x); /* x = 10 here too */
}
```

# Recall: Call-by-Value

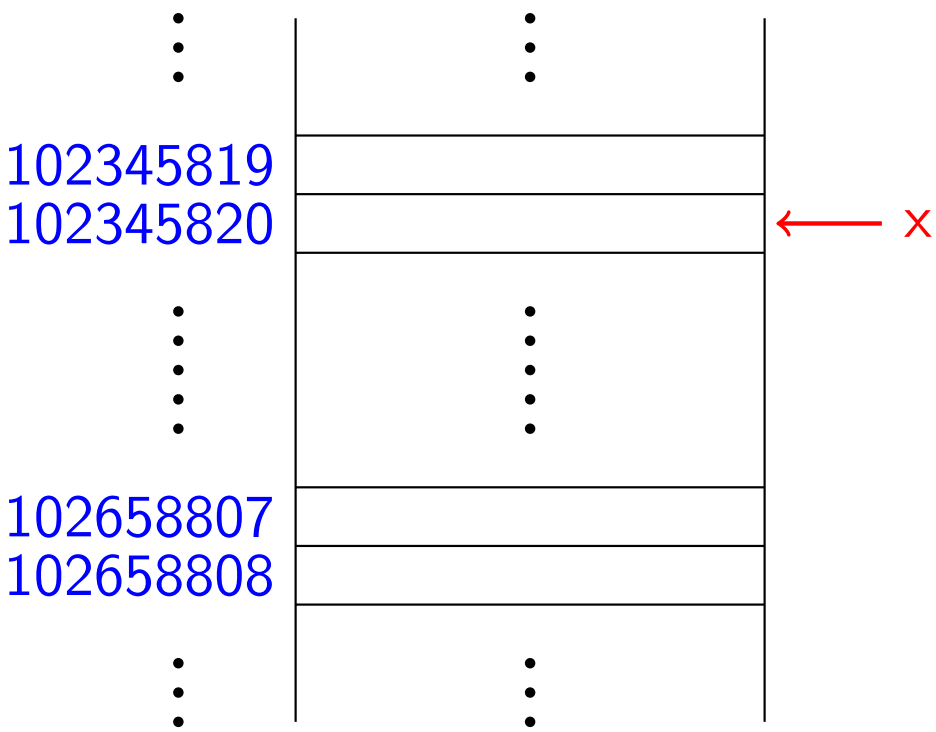



MEMORY

```
void foo( int y )
{
   y = 20;
}
main()
{
   int x;
   x = 10;
   foo(x);
   printf("%d", x);
}
```

PROGRAM

# Recall: Call-by-Value

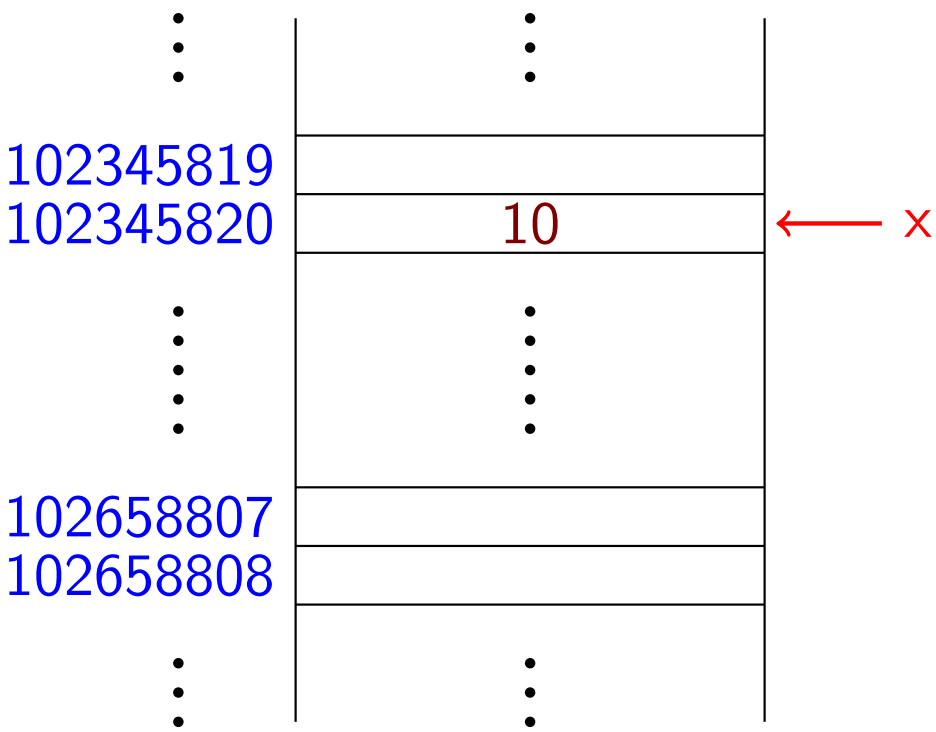



MEMORY

```
void foo( int y )
{
   y = 20;
}
main()
{
   int x;
   x = 10;
   foo(x);
   printf("%d", x);
}
```

PROGRAM

# Recall: Call-by-Value

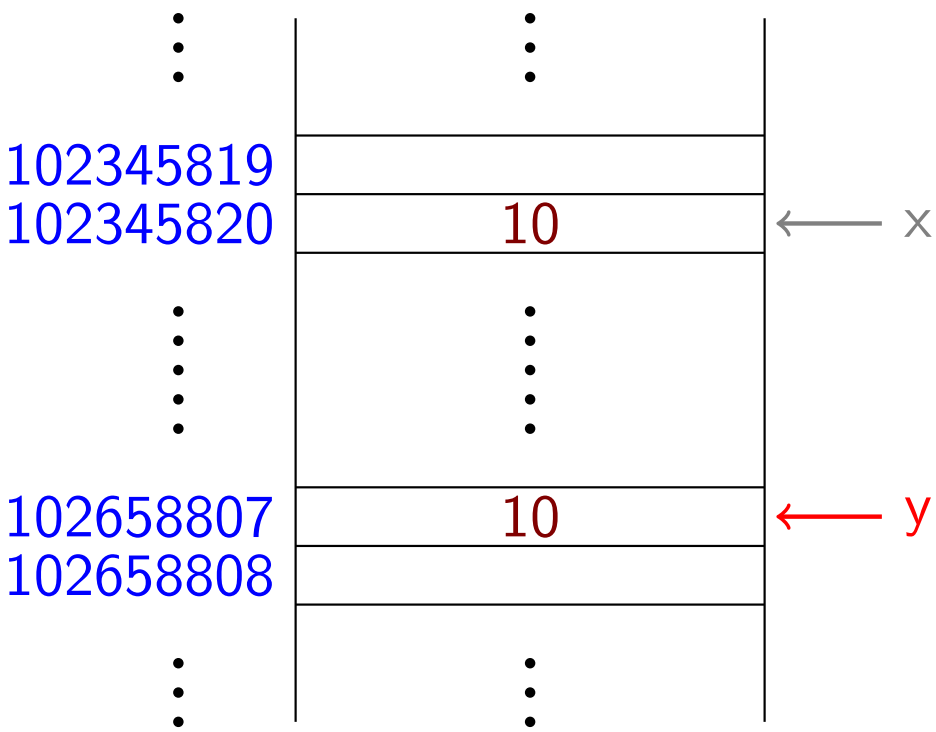



MEMORY

```
void foo( int y )
{
   y = 20;
}
main()
{
   int x;
   x = 10;
   foo(x);
   printf("%d", x);
}
```

PROGRAM

# Recall: Call-by-Value

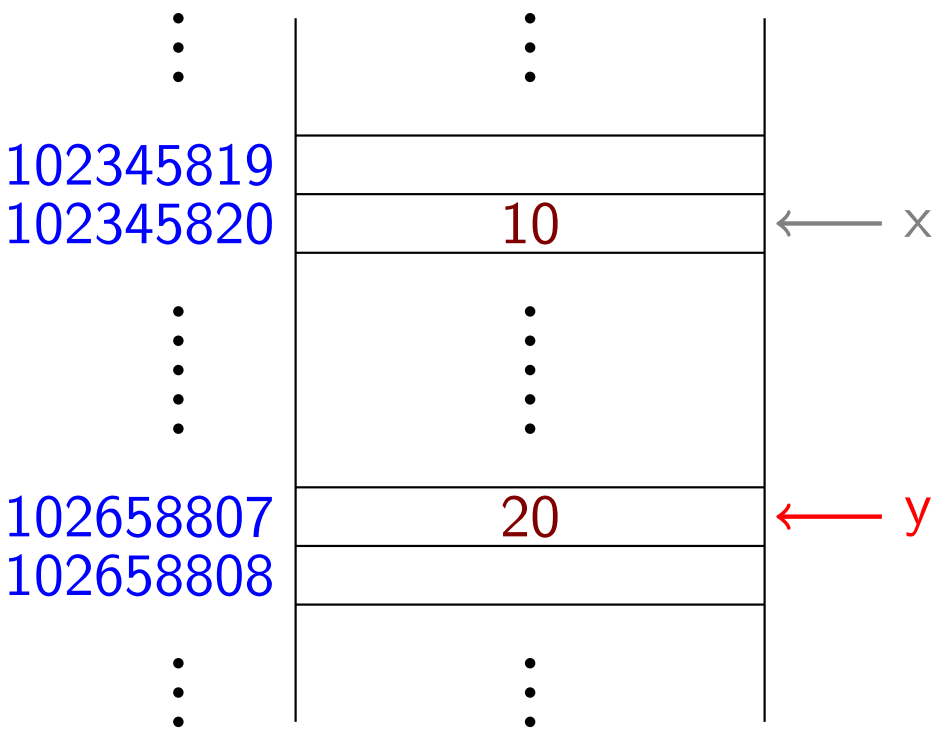



MEMORY

```
void foo( int y )
{
   y = 20;
}
main()
{
   int x;
   x = 10;
   foo(x);
   printf("%d", x);
}
```

PROGRAM

# Recall: Call-by-Value

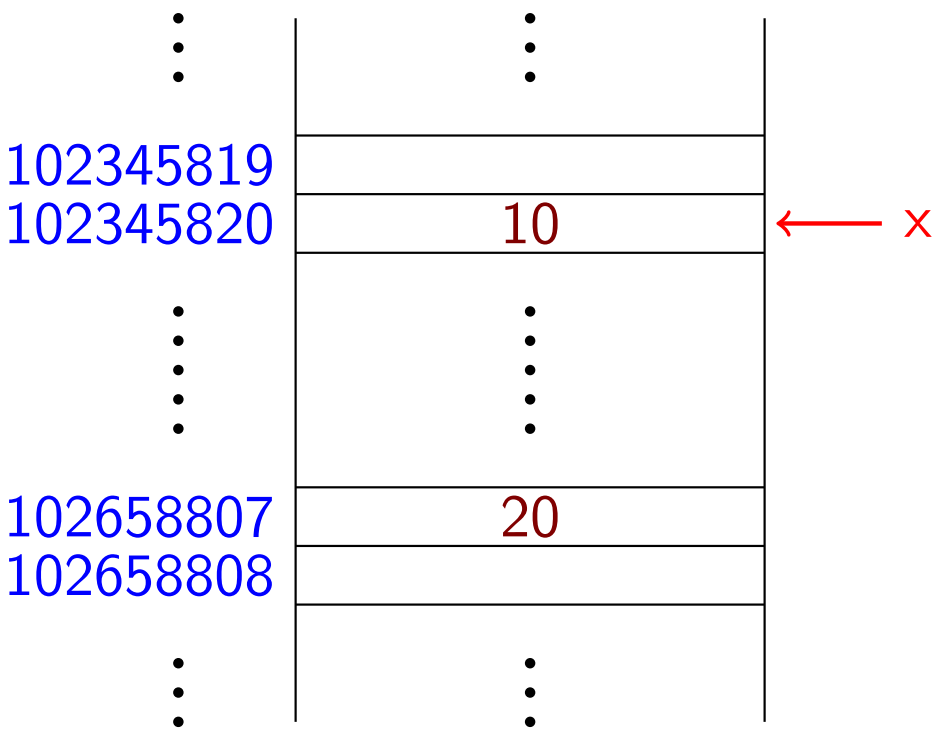



MEMORY

```
void foo( int y )
{
   y = 20;
}
main()
{
   int x;
   x = 10;
   foo(x);
   printf("%d", x);
}
```

PROGRAM

# Recall: Call-by-Value




MEMORY

```
void foo( int y )
{
   y = 20;
}
main()
{
   int x;
   x = 10;
   foo(x);
   printf("%d", x);
}
```

PROGRAM

# Getting Around Call-by-Value

- Suppose that instead of value of `x`, we pass the address of `x` to the function `foo`.
- Then the address of variable `x` of `main` will be available inside `foo` – as the value of variable `y`.
- Suppose also that we can say the following inside `foo`:
  - store 20 in the memory location whose address is stored in variable `y`
- Then the value of `x` will change!

# Getting Around Call-by-Value

- Suppose that instead of value of `x`, we pass the address of `x` to the function `foo`.
- Then the address of variable `x` of `main` will be available inside `foo` – as the value of variable `y`.
- Suppose also that we can say the following inside `foo`:
  - store 20 in the memory location whose address is stored in variable `y`
- Then the value of `x` will change!

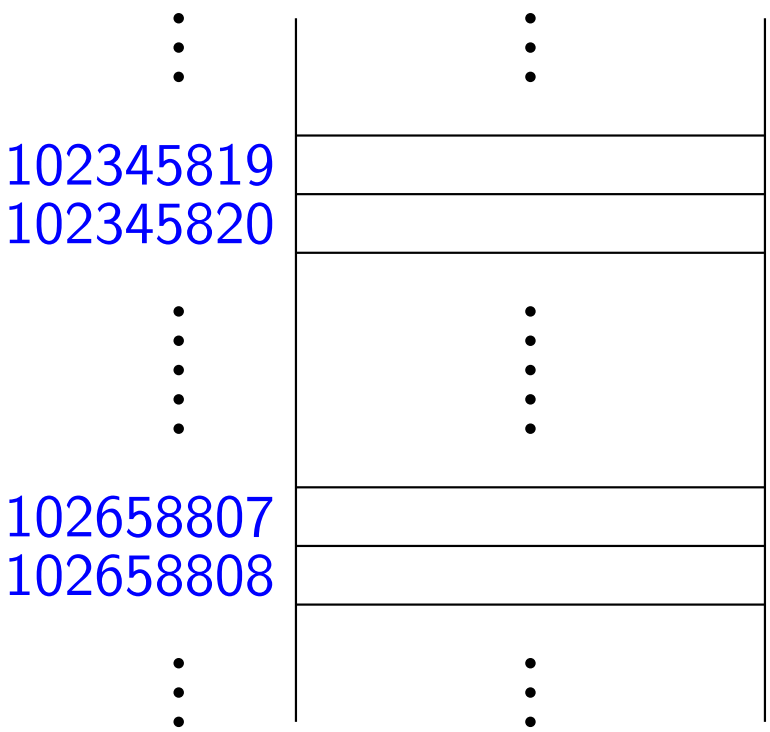
# Getting Around Call-by-Value

- Suppose that instead of value of `x`, we pass the address of `x` to the function `foo`.
- Then the address of variable `x` of `main` will be available inside `foo` – as the value of variable `y`.
- Suppose also that we can say the following inside `foo`:
  - store 20 in the memory location whose address is stored in variable `y`
- Then the value of `x` will change!

# Getting Around Call-by-Value

- Suppose that instead of value of `x`, we pass the address of `x` to the function `foo`.
- Then the address of variable `x` of `main` will be available inside `foo` – as the value of variable `y`.
- Suppose also that we can say the following inside `foo`:
  - store 20 in the memory location whose address is stored in variable `y`
- Then the value of `x` will change!

# Getting Around Call-by-Value

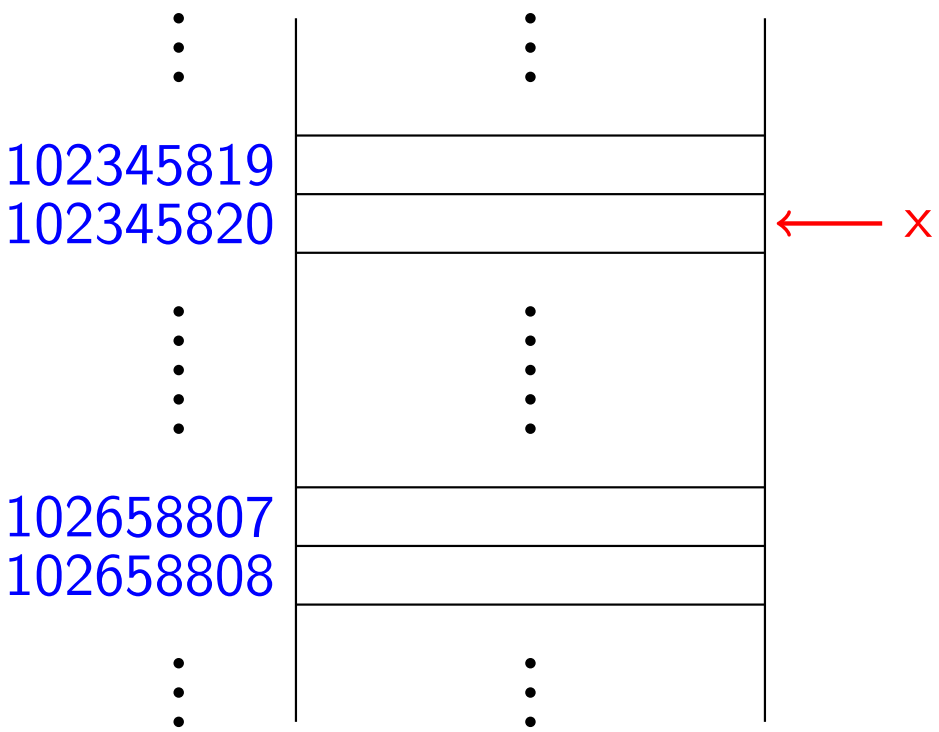



MEMORY

```
void foo( <address> y )
{
   <address in y> = 20;
}
main()
{
   int x;
   x = 10;
   foo(<address of x>);
   printf("%d", x);
}
```

PROGRAM

# Getting Around Call-by-Value

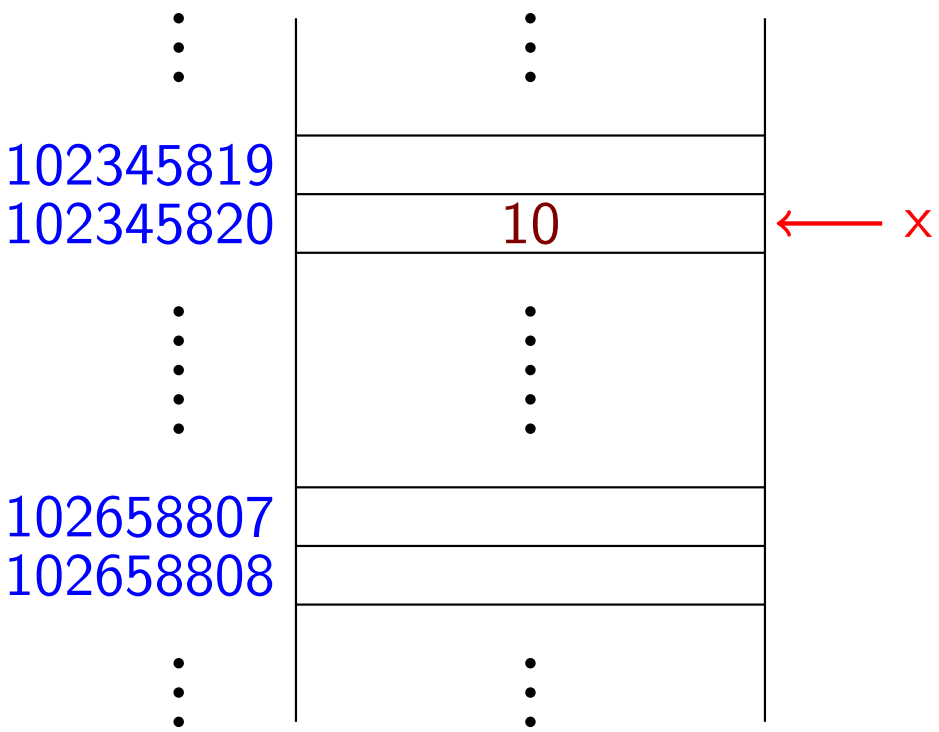



```
void foo( <address> y )
{
   <address in y> = 20;
}
main()
{
   int x;
   x = 10;
   foo(<address of x>);
   printf("%d", x);
}
```

MEMORY

PROGRAM

# Getting Around Call-by-Value

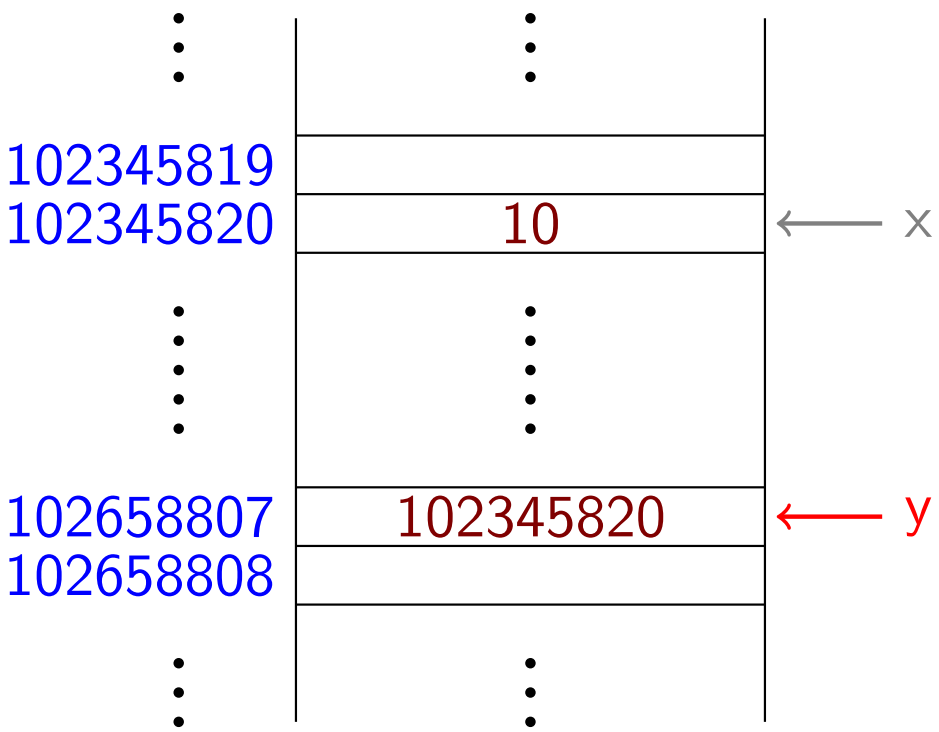



MEMORY

```
void foo( <address> y )
{
   <address in y> = 20;
}
main()
{
   int x;
   x = 10;
   foo(<address of x>);
   printf("%d", x);
}
```

PROGRAM

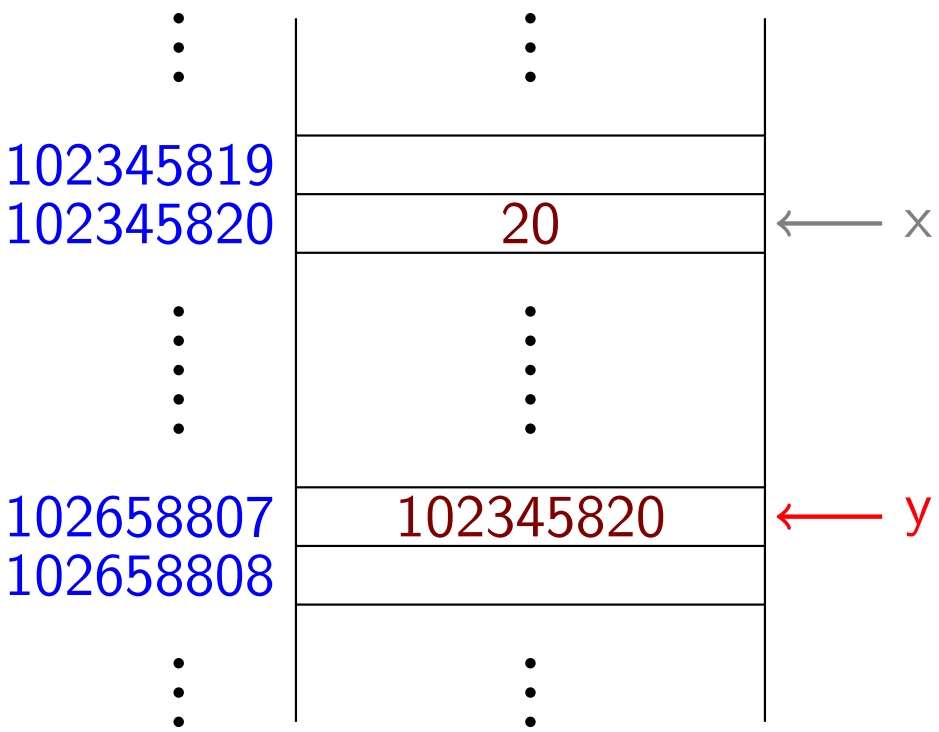
# Getting Around Call-by-Value




```
void foo( <address> y )
{
   <address in y> = 20;
}
main()
{
   int x;
   x = 10;
   foo(<address of x>);
   printf("%d", x);
}
```

**MEMORY** **PROGRAM**

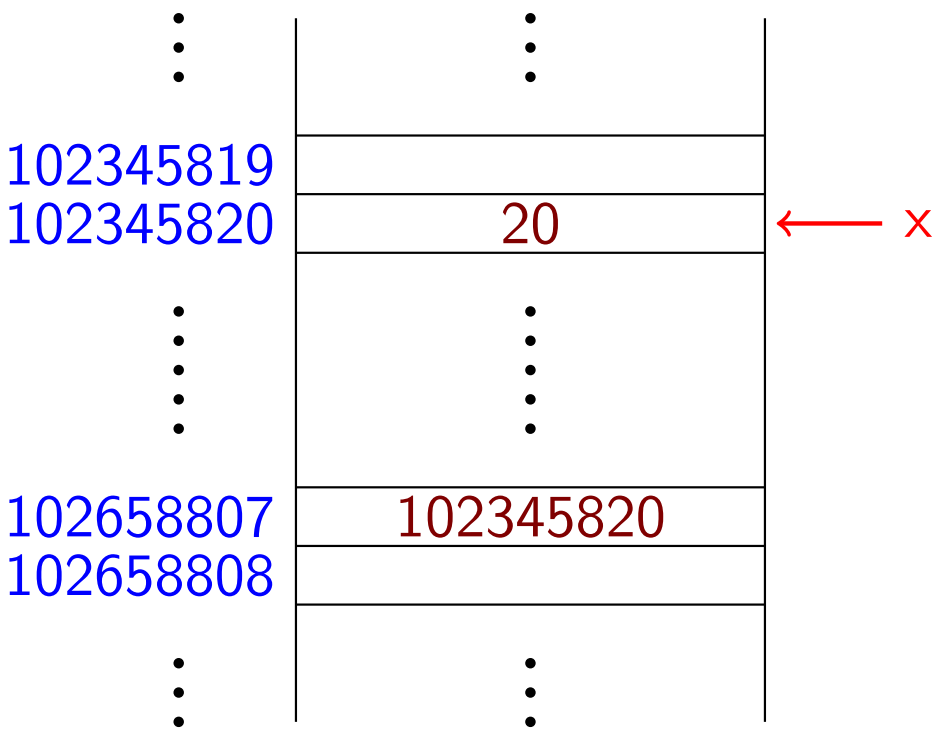
# Getting Around Call-by-Value




```
void foo( <address> y )
{
   <address in y> = 20;
}
main()
{
   int x;
   x = 10;
   foo(<address of x>);
   printf("%d", x);
}
```

**MEMORY** **PROGRAM**

# Getting Around Call-by-Value




```
void foo( <address> y )
{
   <address in y> = 20;
}
main()
{
   int x;
   x = 10;
   foo(<address of x>);
   printf("%d", x);
}
```

MEMORY

PROGRAM

# Getting Around Call-by-Value

C allows this!

<address of x>: &x

<address in y>: *y

Type <address>: Simply the type of *y!

# Getting Around Call-by-Value

C allows this!

<address of x>: &x

<address in y>: *y

Type <address>: Simply the type of *y!

# Getting Around Call-by-Value

C allows this!

`<address of x>: &x`

`<address in y>: *y`

Type `<address>`: Simply the type of `*y`!

# Getting Around Call-by-Value

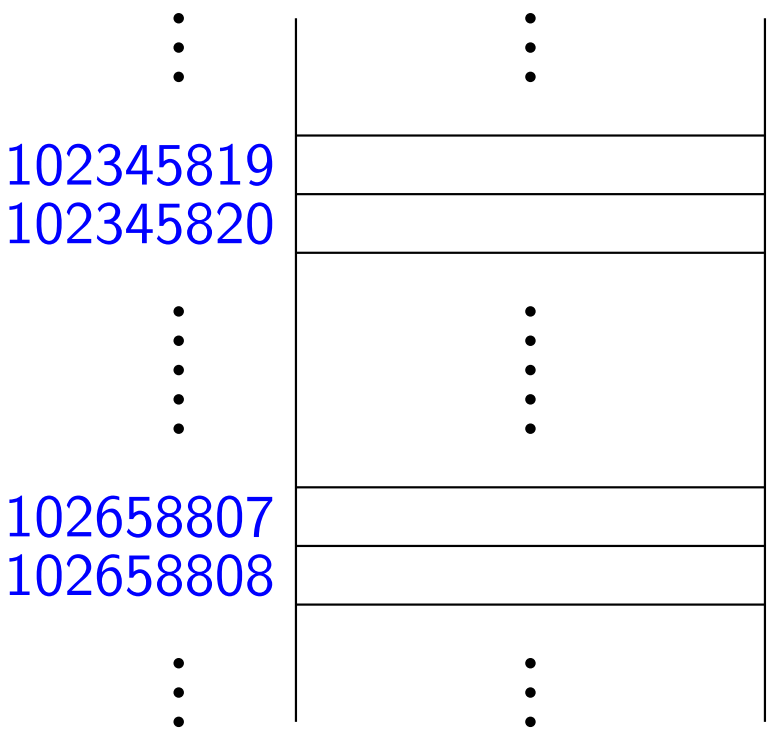
C allows this!

`<address of x>`: `&x`

`<address in y>`: `*y`

Type `<address>`: Simply the type of `*y`!

# Getting Around Call-by-Value

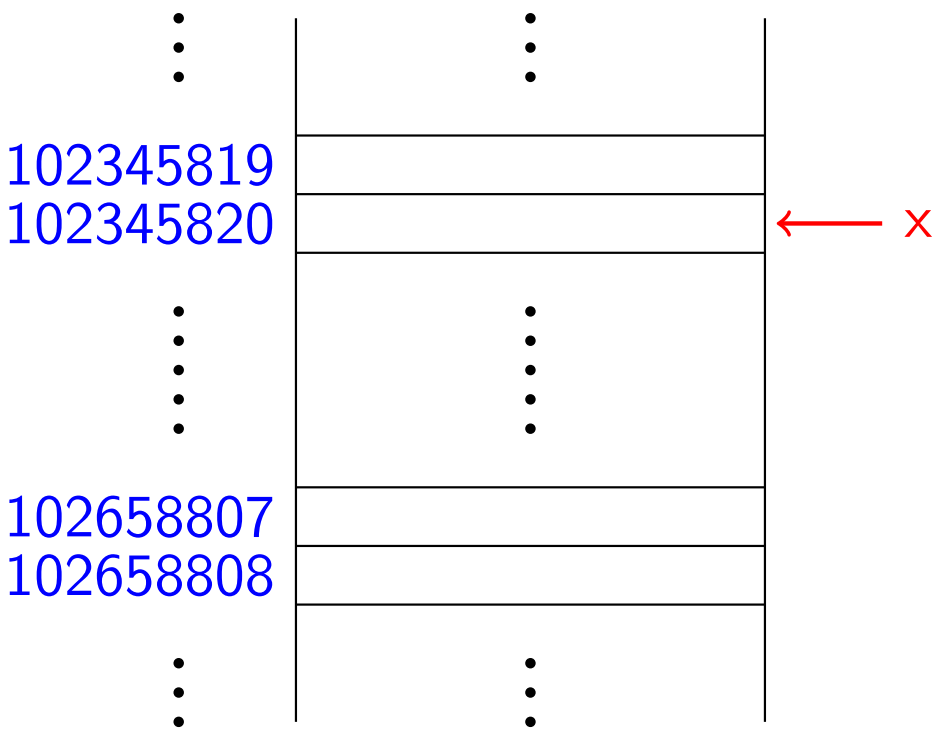



MEMORY

```
void foo(int *y )
{
   *y = 20;
}
main()
{
   int x;
   x = 10;
   foo(&x);
   printf("%d", x);
}
```

PROGRAM

# Getting Around Call-by-Value

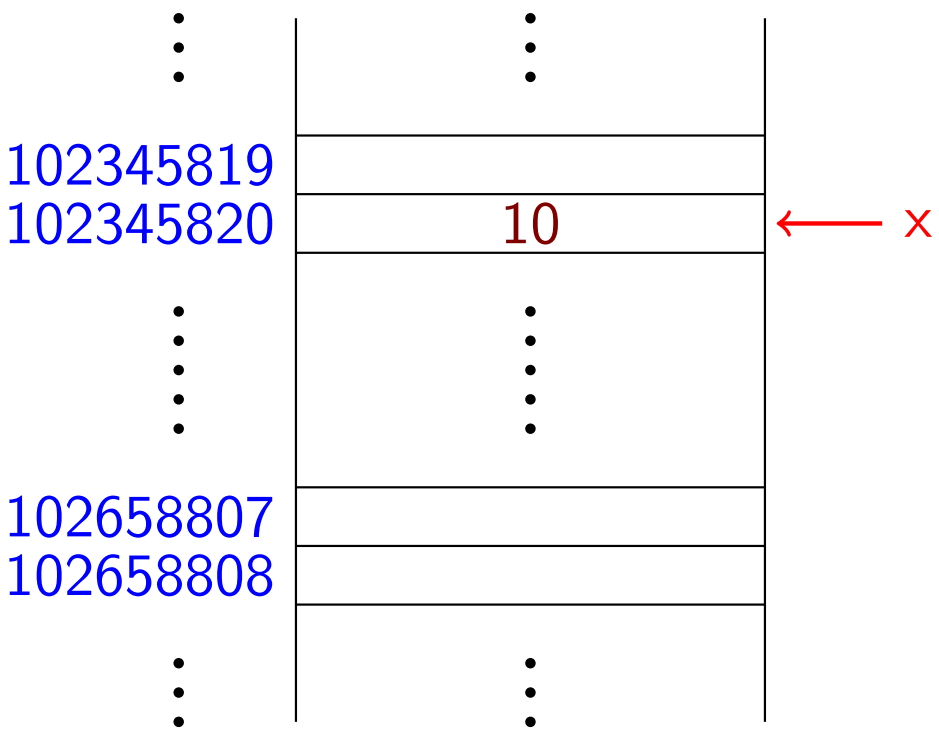



```
void foo(int *y )
{
   *y = 20;
}
main()
{
   int x;
   x = 10;
   foo(&x);
   printf("%d", x);
}
```

MEMORY

PROGRAM

# Getting Around Call-by-Value

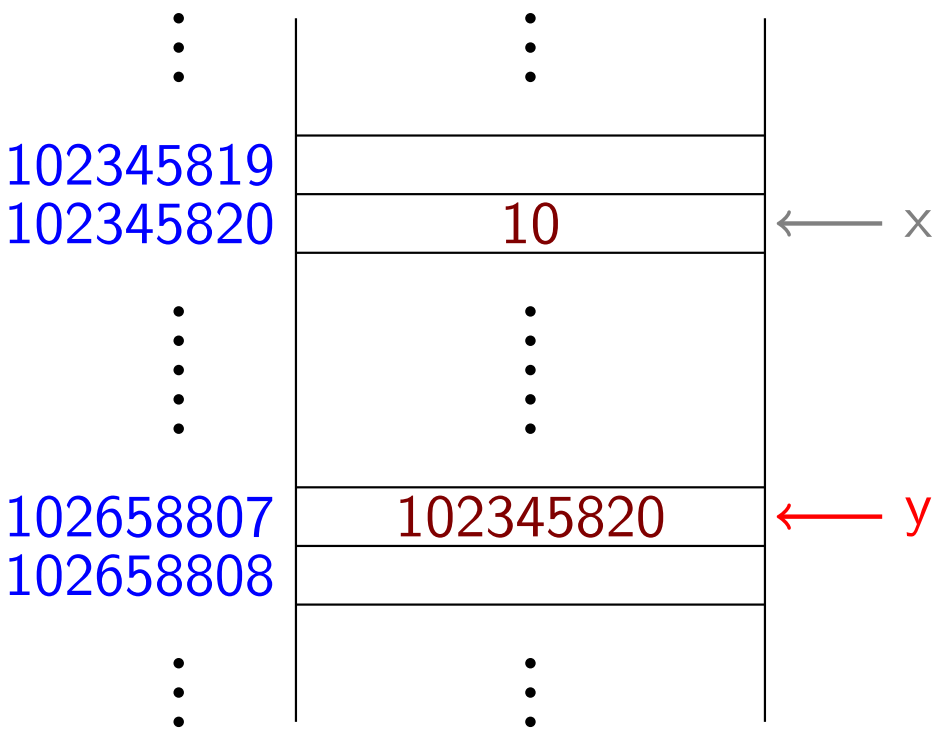



```
void foo(int *y )
{
   *y = 20;
}
main()
{
   int x;
   x = 10;
   foo(&x);
   printf("%d", x);
}
```

MEMORY

PROGRAM

# Getting Around Call-by-Value

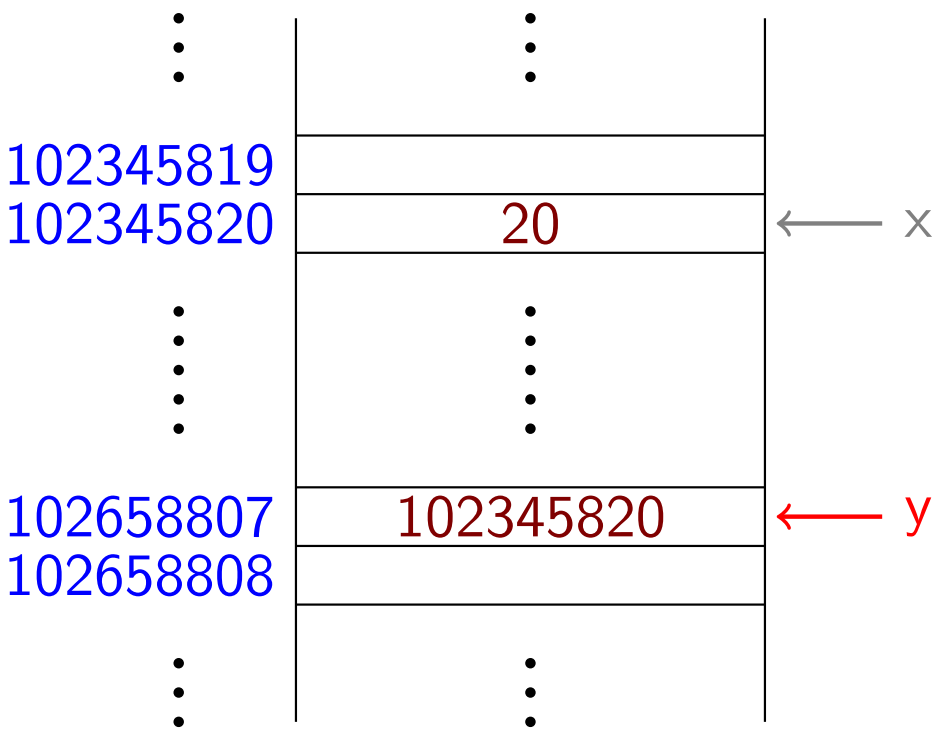



```
void foo(int *y))
{
   *y = 20;
}
main()
{
   int x;
   x = 10;
   foo(&x);
   printf("%d", x);
}
```

MEMORY

PROGRAM

# Getting Around Call-by-Value

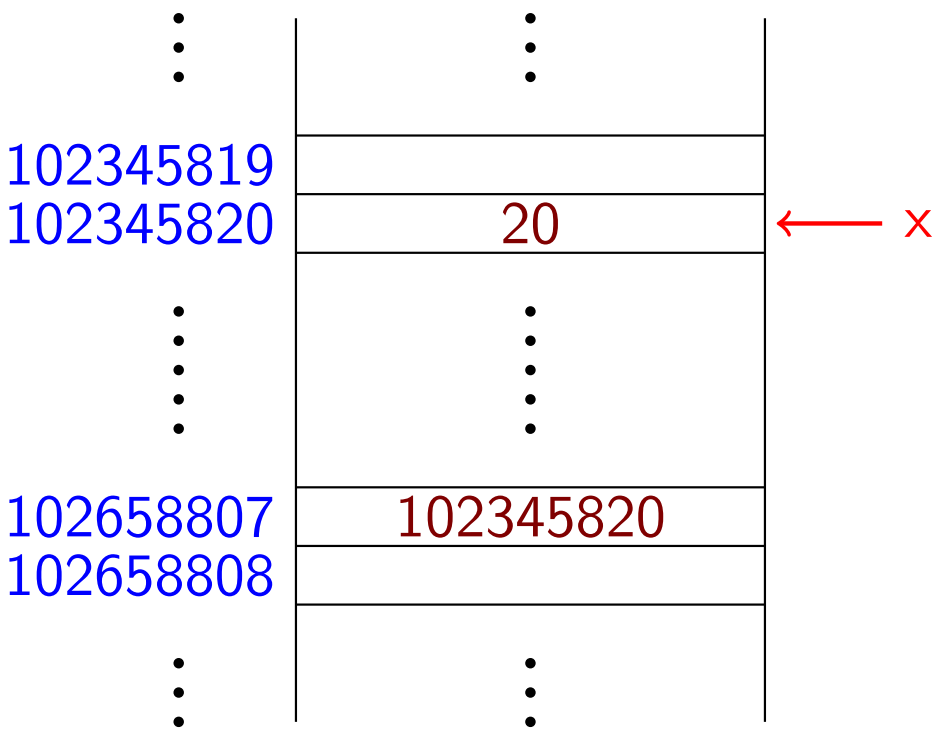



```
void foo(int *y )
{
   *y = 20;
}
main()
{
   int x;
   x = 10;
   foo(&x);
   printf("%d", x);
}
```

MEMORY

PROGRAM

# Getting Around Call-by-Value




```
void foo(int *y )
{
   *y = 20;
}
main()
{
   int x;
   x = 10;
   foo(&x);
   printf("%d", x);
}
```

MEMORY

PROGRAM

# Pointers

- Variable `y` stores a number.
- However, a special meaning is attached to this number: this number is the address of a memory location that stores an `int`.
- The type of `y` is denoted by `(int *)`.
- Variable `y` is also called a pointer:
  - It points to the location of variable `x` of `main`.
  - `*y` represents the variable `x` of `main`!
- `&x` is also a pointer to `x` and equals `y`.

# Pointers

- Variable `y` stores a number.
- However, a special meaning is attached to this number: this number is the address of a memory location that stores an `int`.
- The type of `y` is denoted by `(int *)`.
- Variable `y` is also called a pointer:
  - It points to the location of variable `x` of `main`.
  - `*y` represents the variable `x` of `main`!
- `&x` is also a pointer to `x` and equals `y`.

# Pointers

- Variable `y` stores a number.
- However, a special meaning is attached to this number: this number is the address of a memory location that stores an `int`.
- The type of `y` is denoted by `(int *)`.
- Variable `y` is also called a pointer:
  - It points to the location of variable `x` of `main`.
  - `*y` represents the variable `x` of `main`!
- `&x` is also a pointer to `x` and equals `y`.

# Pointers

- Variable `y` stores a number.
- However, a special meaning is attached to this number: this number is the address of a memory location that stores an `int`.
- The type of `y` is denoted by `(int *)`.
- Variable `y` is also called a pointer:
  - It points to the location of variable `x` of `main`.
  - `*y` represents the variable `x` of `main`!
- `&x` is also a pointer to `x` and equals `y`.

# Pointers

- Variable `y` stores a number.
- However, a special meaning is attached to this number: this number is the address of a memory location that stores an `int`.
- The type of `y` is denoted by `(int *)`.
- Variable `y` is also called a pointer:
  - It points to the location of variable `x` of `main`.
  - `*y` represents the variable `x` of `main`!
- `&x` is also a pointer to `x` and equals `y`.

# Pointers

- Variable `y` stores a number.
- However, a special meaning is attached to this number: this number is the address of a memory location that stores an `int`.
- The type of `y` is denoted by `(int *)`.
- Variable `y` is also called a pointer:
  - It points to the location of variable `x` of `main`.
  - `*y` represents the variable `x` of `main`!
- `&x` is also a pointer to `x` and equals `y`.

# Pointers

- Variable `y` stores a number.
- However, a special meaning is attached to this number: this number is the address of a memory location that stores an `int`.
- The type of `y` is denoted by `(int *)`.
- Variable `y` is also called a pointer:
  - It points to the location of variable `x` of `main`.
  - `*y` represents the variable `x` of `main`!
- `&x` is also a pointer to `x` and equals `y`.

# Resolving the Anomaly for Arrays

- Declaration
  `int z[3]`
  reserves 3 memory locations (each of 4 bytes).
- These are named `z[0]` to `z[2]`.
- In addition to this, another memory location is reserved!
- The name of this location is `z`.
- It stores the pointer to `z[0]`.
- This is why passing name of array as parameter allows us to change its content inside a function.

# Resolving the Anomaly for Arrays

- Declaration
  `int z[3]`
  reserves 3 memory locations (each of 4 bytes).
- These are named `z[0]` to `z[2]`.
- In addition to this, another memory location is reserved!
- The name of this location is `z`.
- It stores the pointer to `z[0]`.
- This is why passing name of array as parameter allows us to change its content inside a function.

# Resolving the Anomaly for Arrays

- Declaration
  `int z[3]`
  reserves 3 memory locations (each of 4 bytes).
- These are named `z[0]` to `z[2]`.
- In addition to this, another memory location is reserved!
- The name of this location is `z`.
- It stores the pointer to `z[0]`.
- This is why passing name of array as parameter allows us to change its content inside a function.

# Resolving the Anomaly for Arrays

- Declaration
  `int z[3]`
  reserves 3 memory locations (each of 4 bytes).
- These are named `z[0]` to `z[2]`.
- In addition to this, another memory location is reserved!
- The name of this location is `z`.
- It stores the pointer to `z[0]`.
- This is why passing name of array as parameter allows us to change its content inside a function.

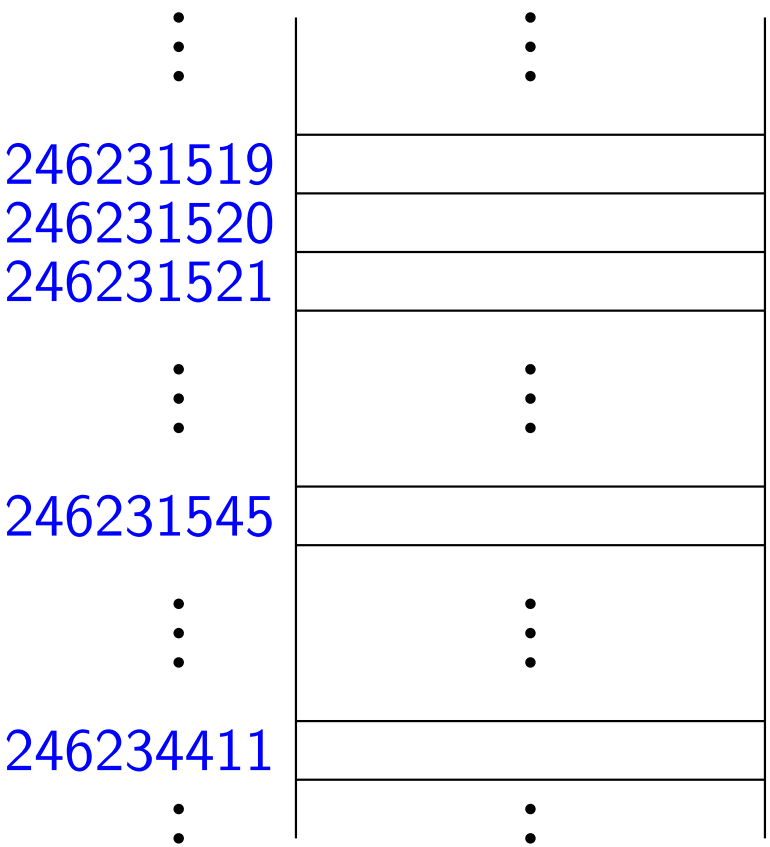
# Resolving the Anomaly for Arrays

- Declaration
  `int z[3]`
  reserves 3 memory locations (each of 4 bytes).
- These are named `z[0]` to `z[2]`.
- In addition to this, another memory location is reserved!
- The name of this location is `z`.
- It stores the pointer to `z[0]`.
- This is why passing name of array as parameter allows us to change its content inside a function.

# Resolving the Anomaly for Arrays

- Declaration
  `int z[3]`
  reserves 3 memory locations (each of 4 bytes).
- These are named `z[0]` to `z[2]`.
- In addition to this, another memory location is reserved!
- The name of this location is `z`.
- It stores the pointer to `z[0]`.
- This is why passing name of array as parameter allows us to change its content inside a function.

# Example

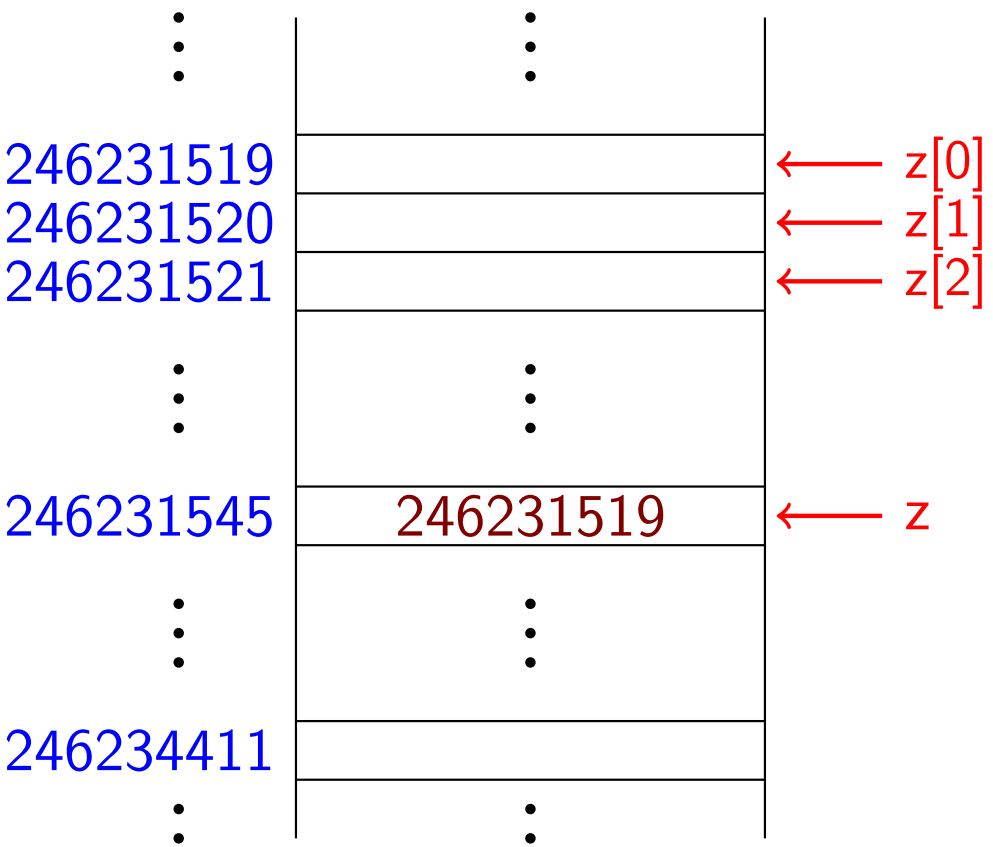



MEMORY

```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

PROGRAM

# Example

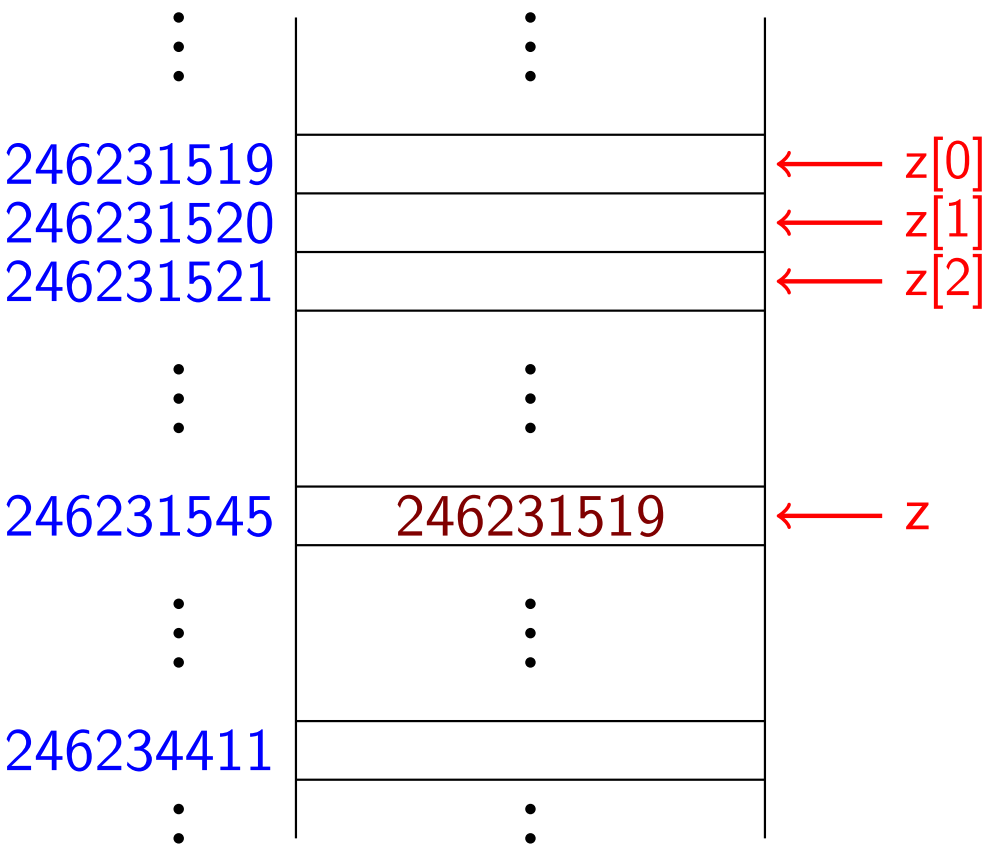



MEMORY

```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

PROGRAM

# Example

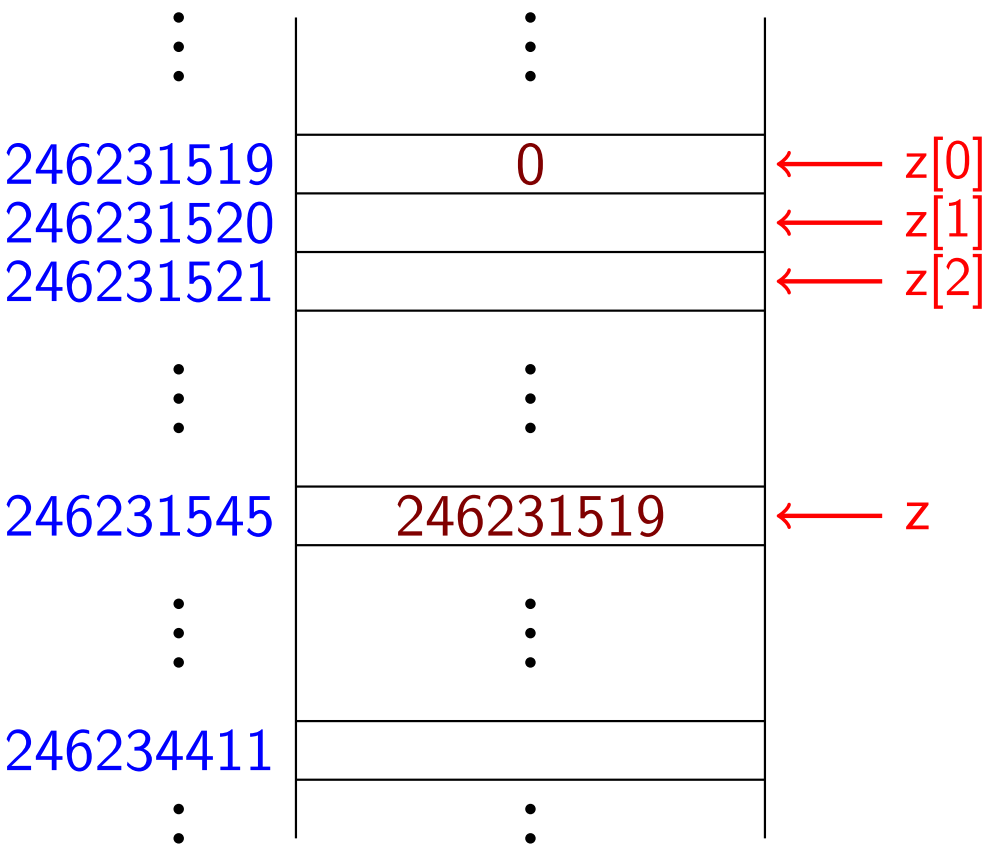



```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

**MEMORY**

**PROGRAM**

# Example

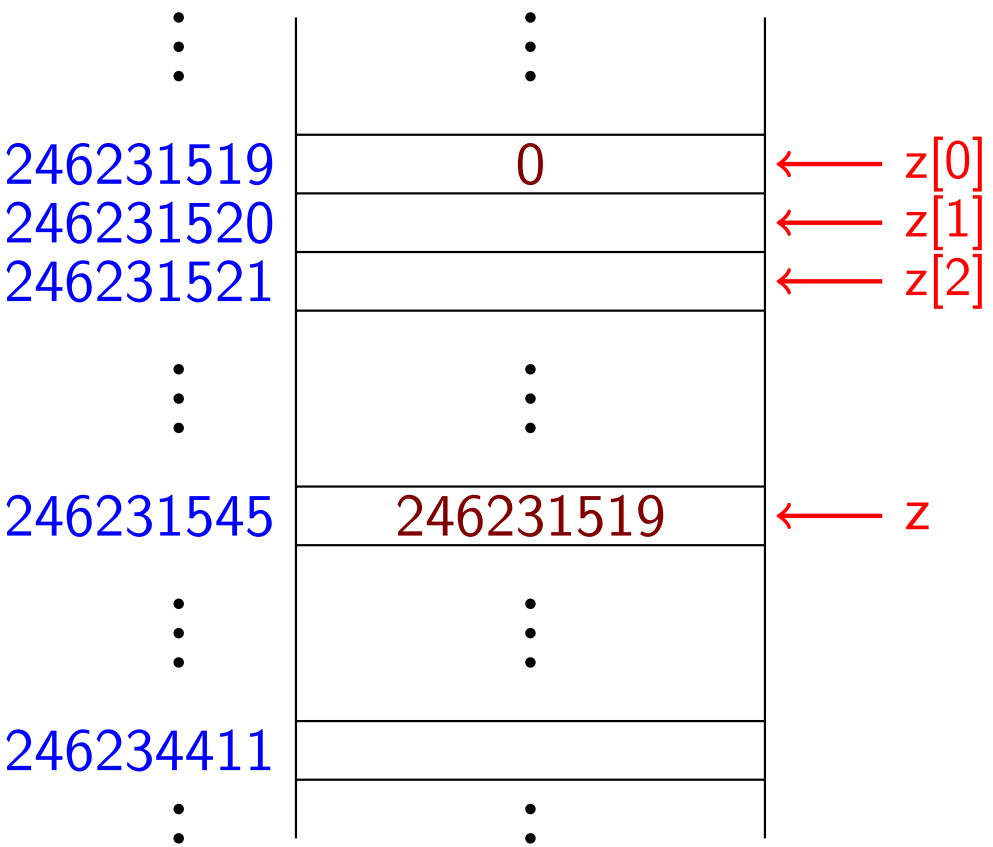



MEMORY

```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

PROGRAM

# Example

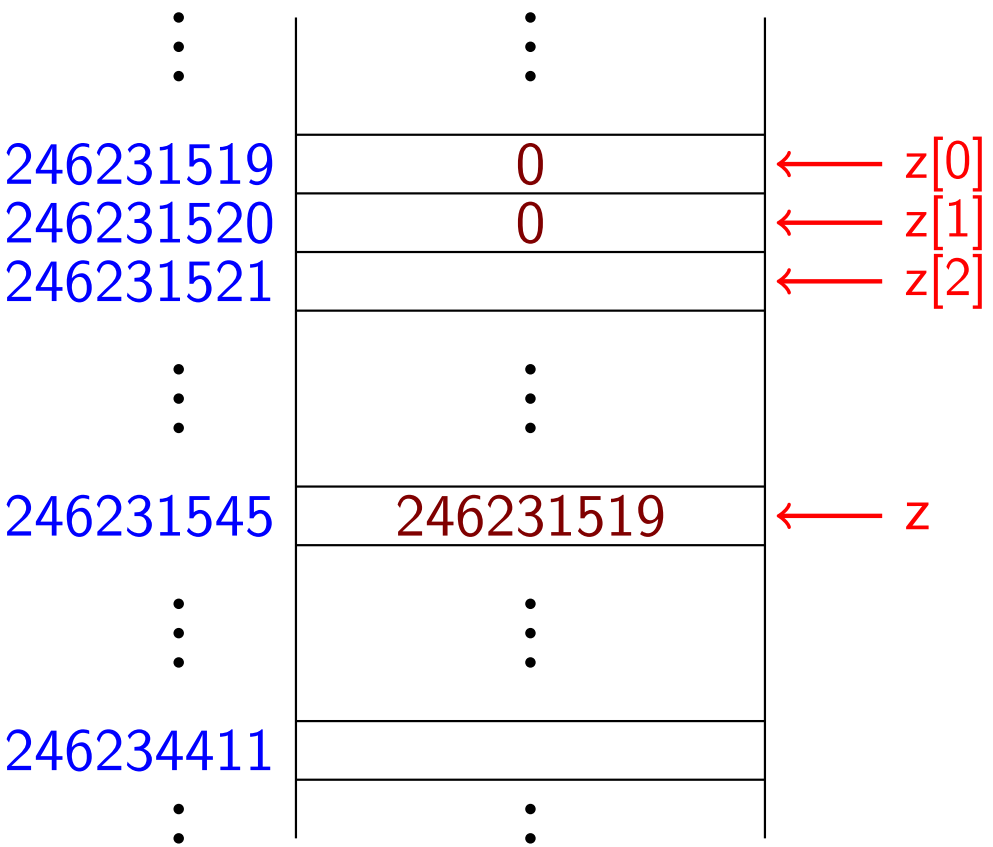



```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

**MEMORY**

**PROGRAM**

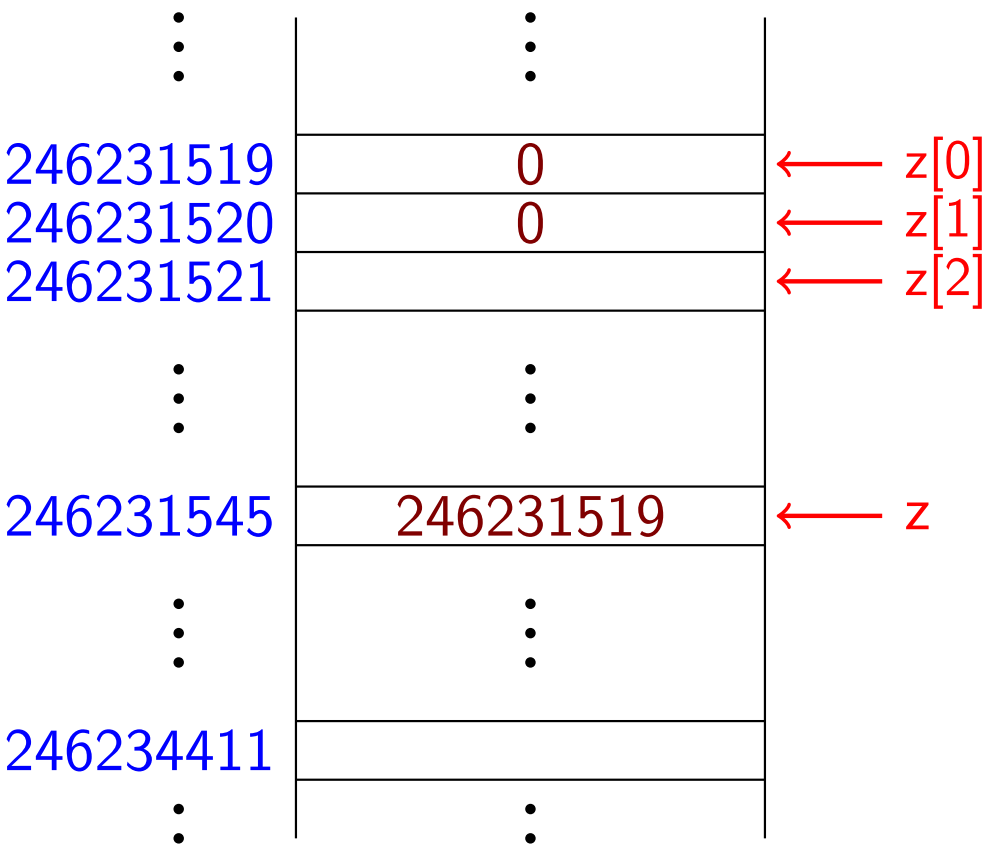
# Example




```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

PROGRAM

# Example

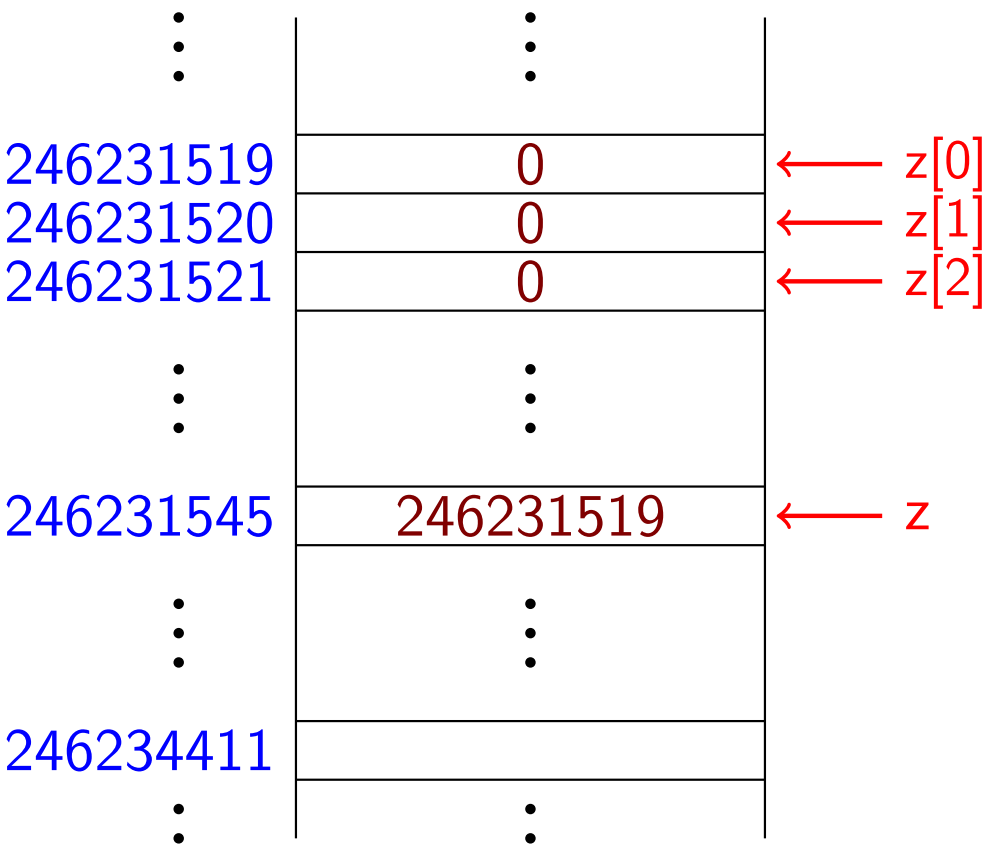



MEMORY

```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

PROGRAM

# Example

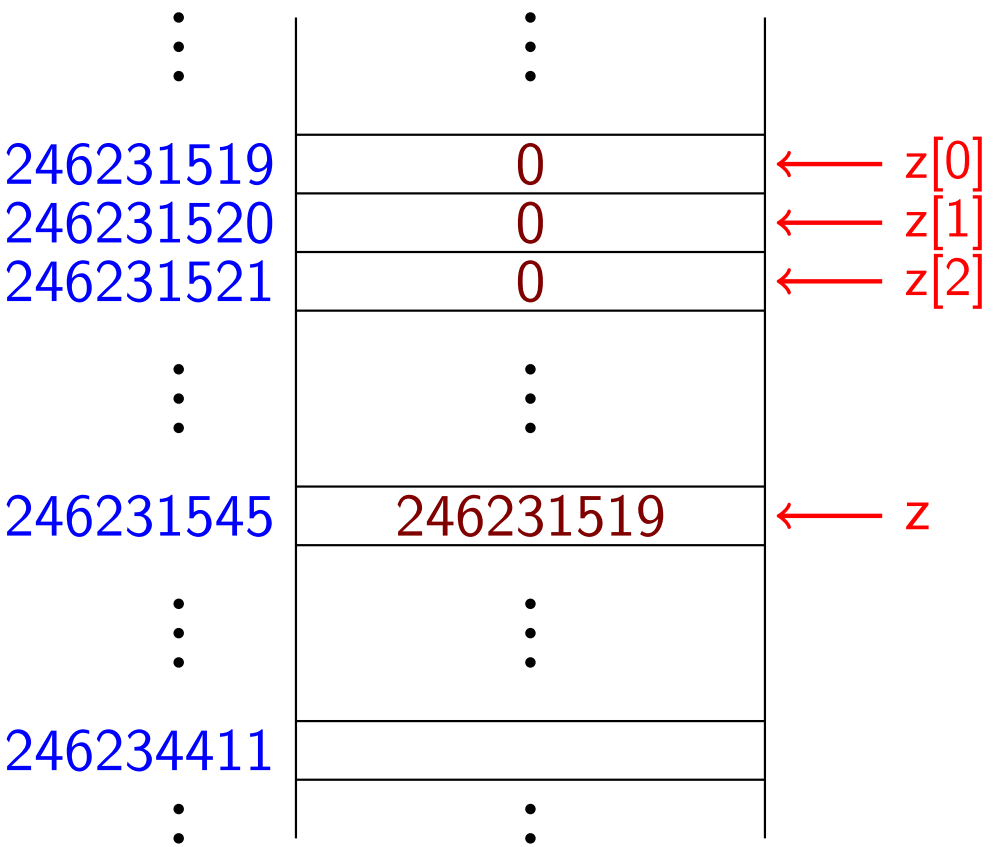



```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

**MEMORY**

**PROGRAM**

# Example

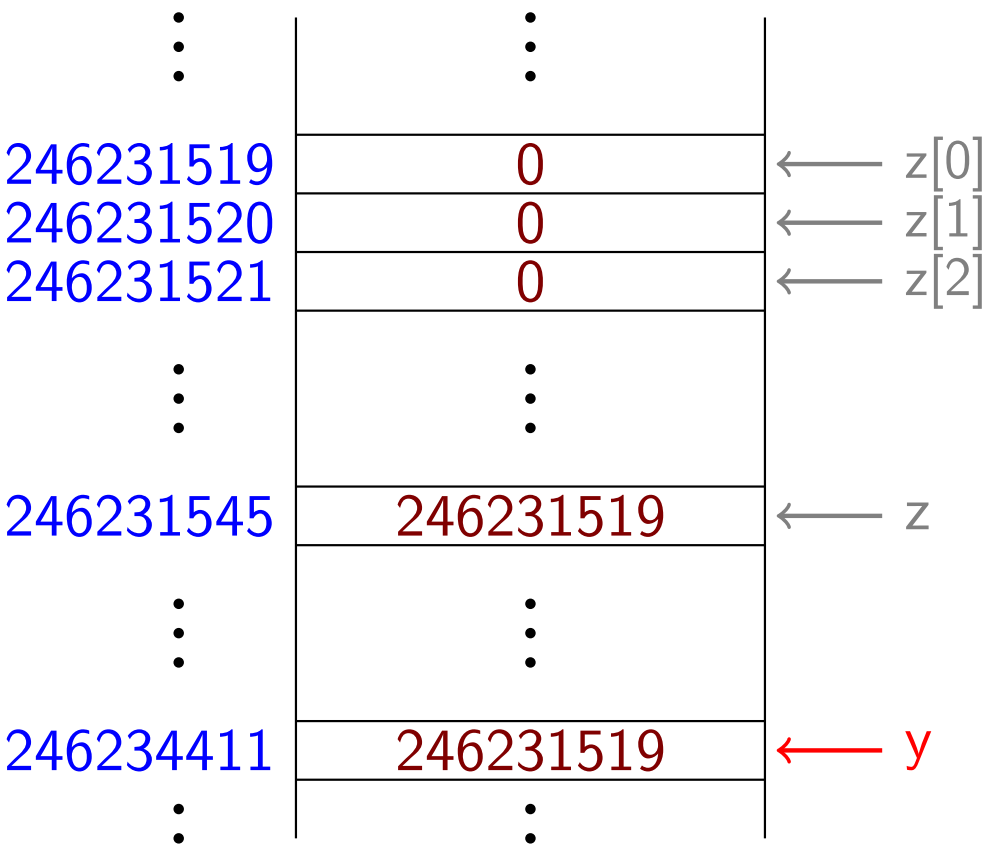



```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

**MEMORY**

**PROGRAM**

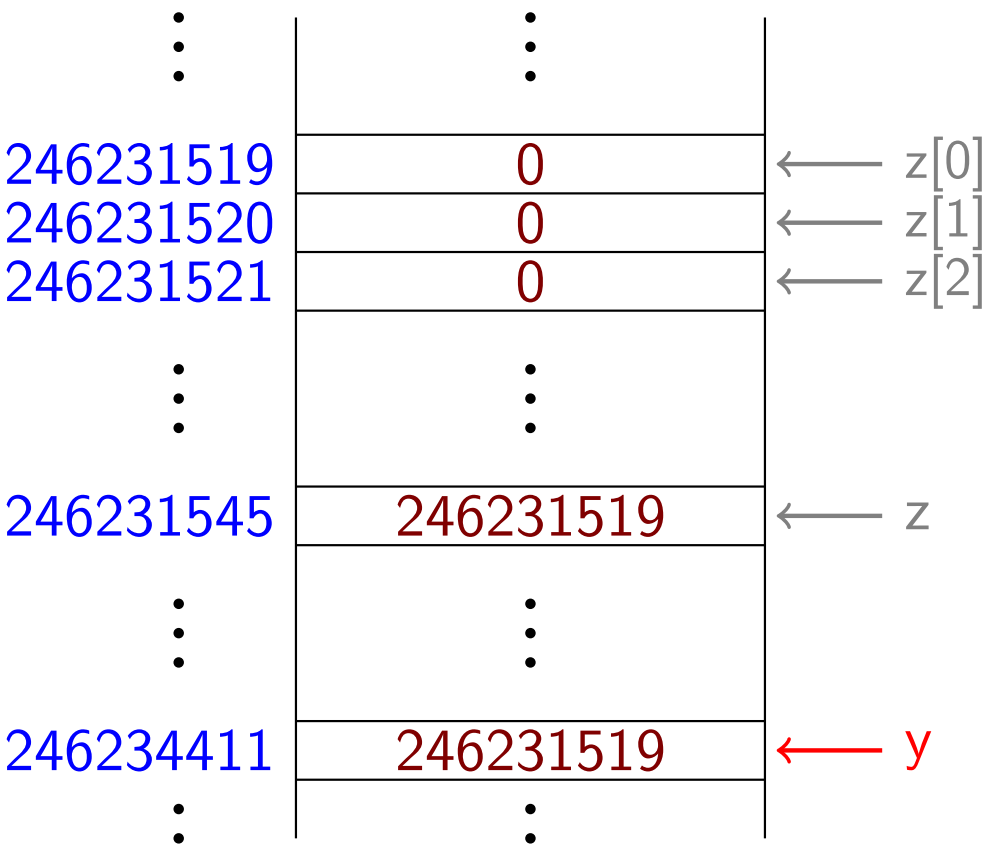
# Example




```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

**MEMORY** **PROGRAM**

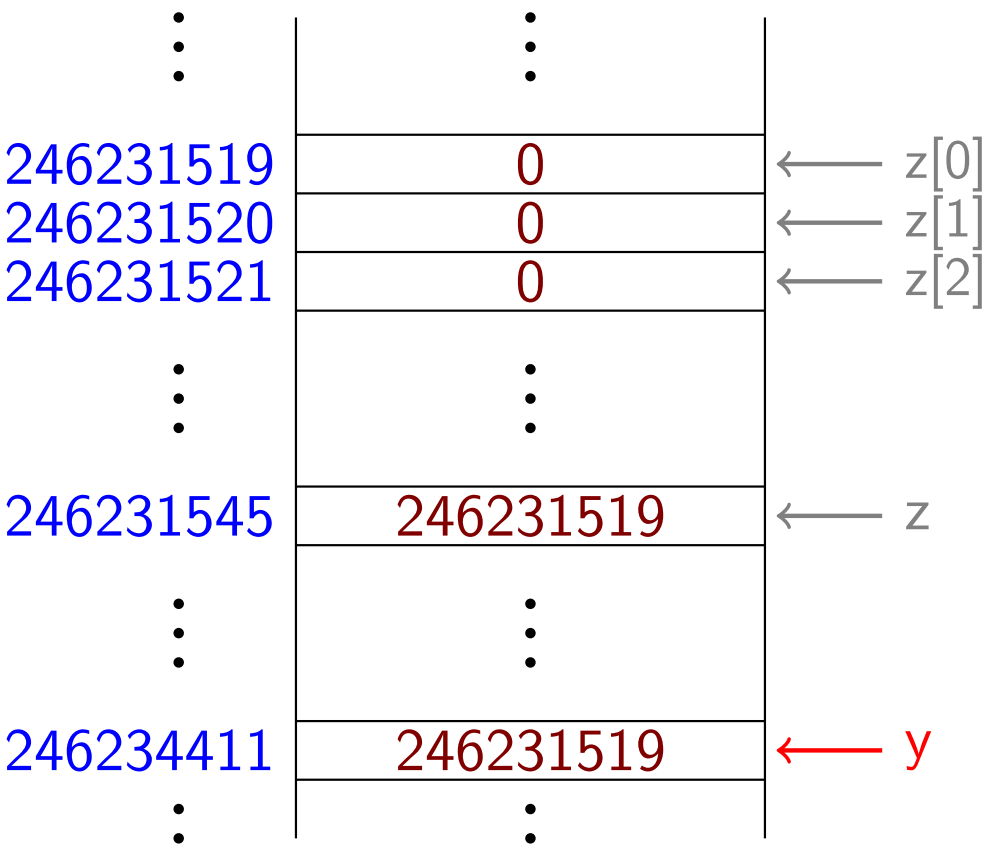
# Example




```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

**MEMORY** **PROGRAM**

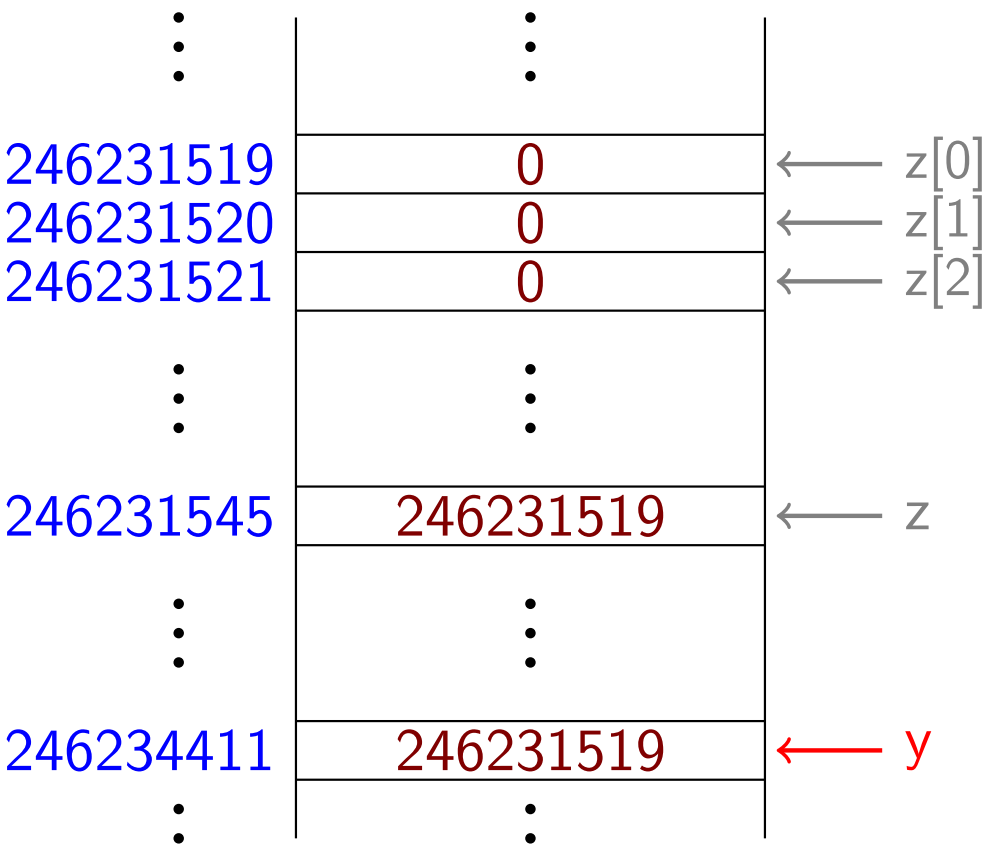
# Example




```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

**MEMORY** **PROGRAM**

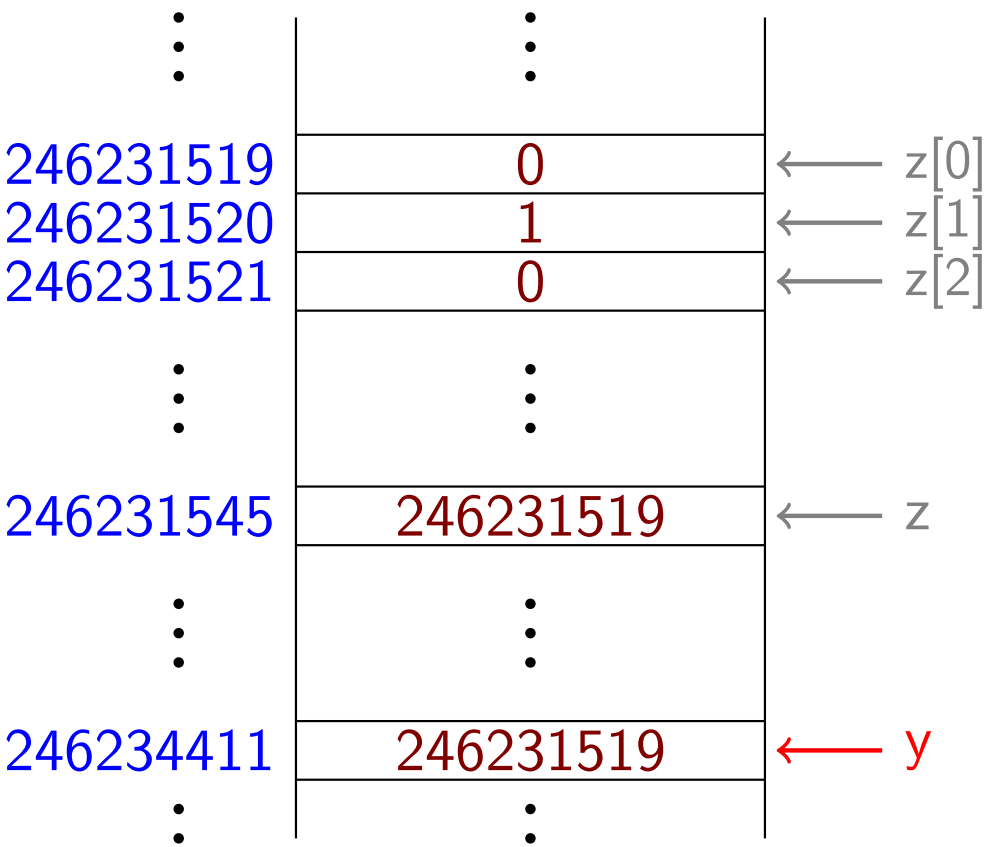
# Example




```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

**MEMORY** **PROGRAM**

# Example

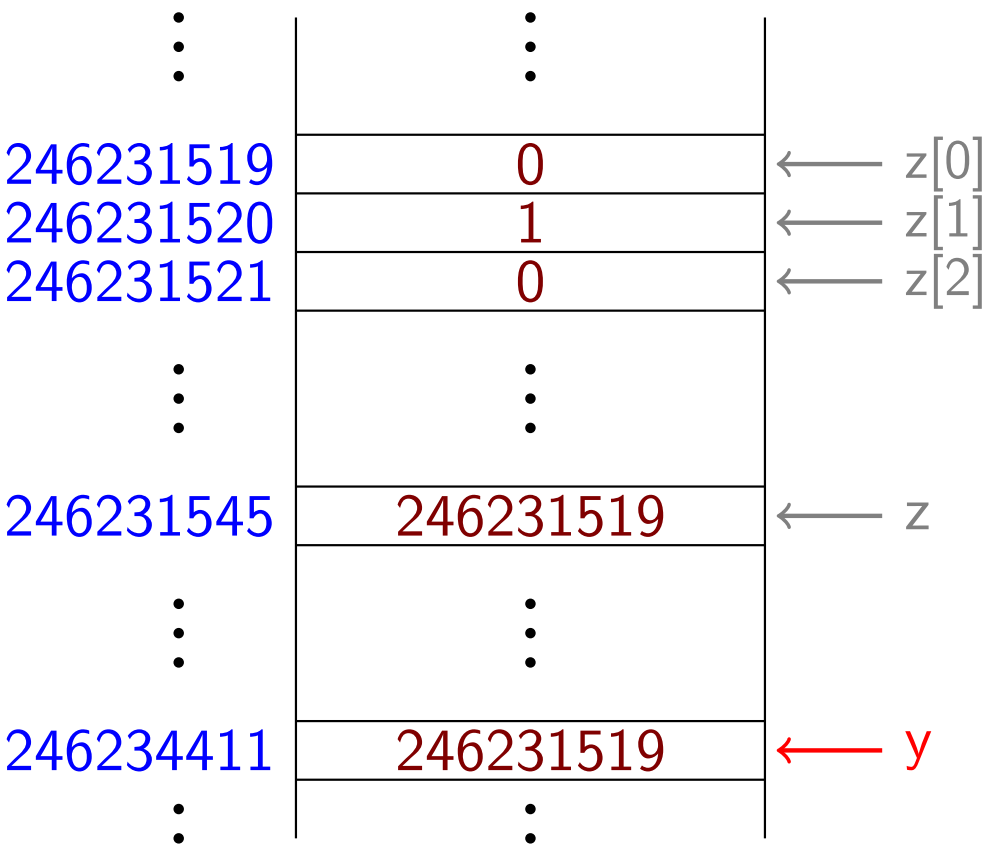



**MEMORY**

```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

**PROGRAM**

# Example

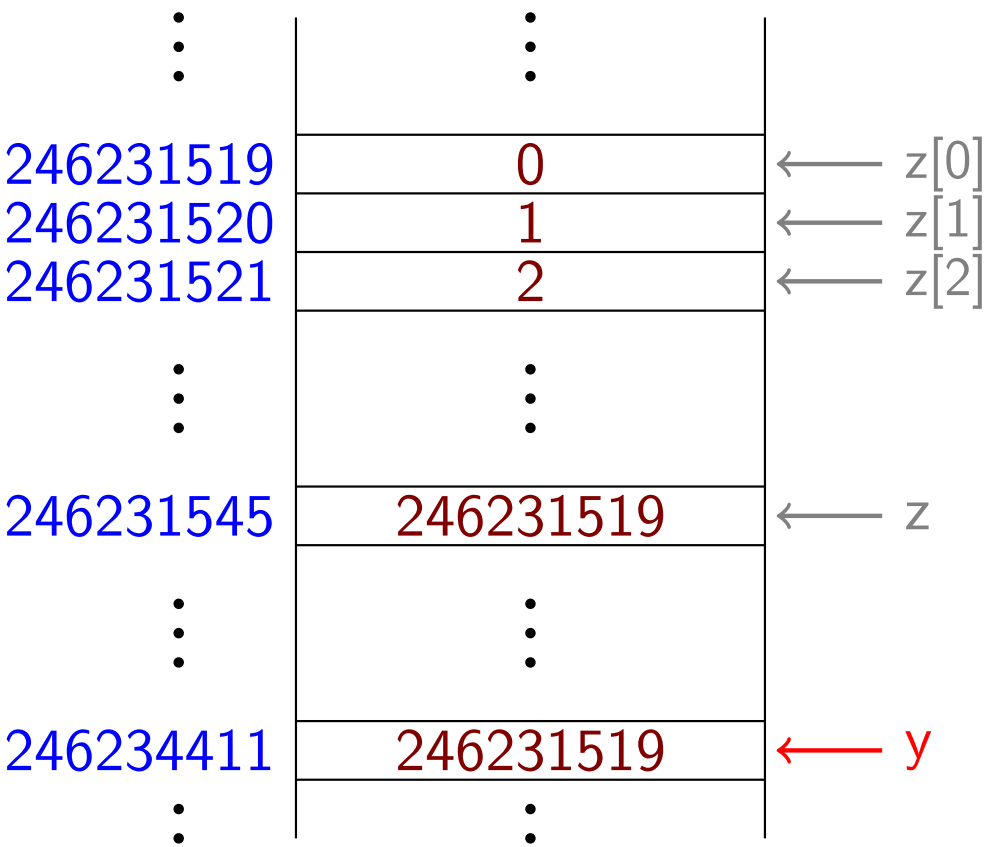



**MEMORY**

```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

**PROGRAM**

# Example

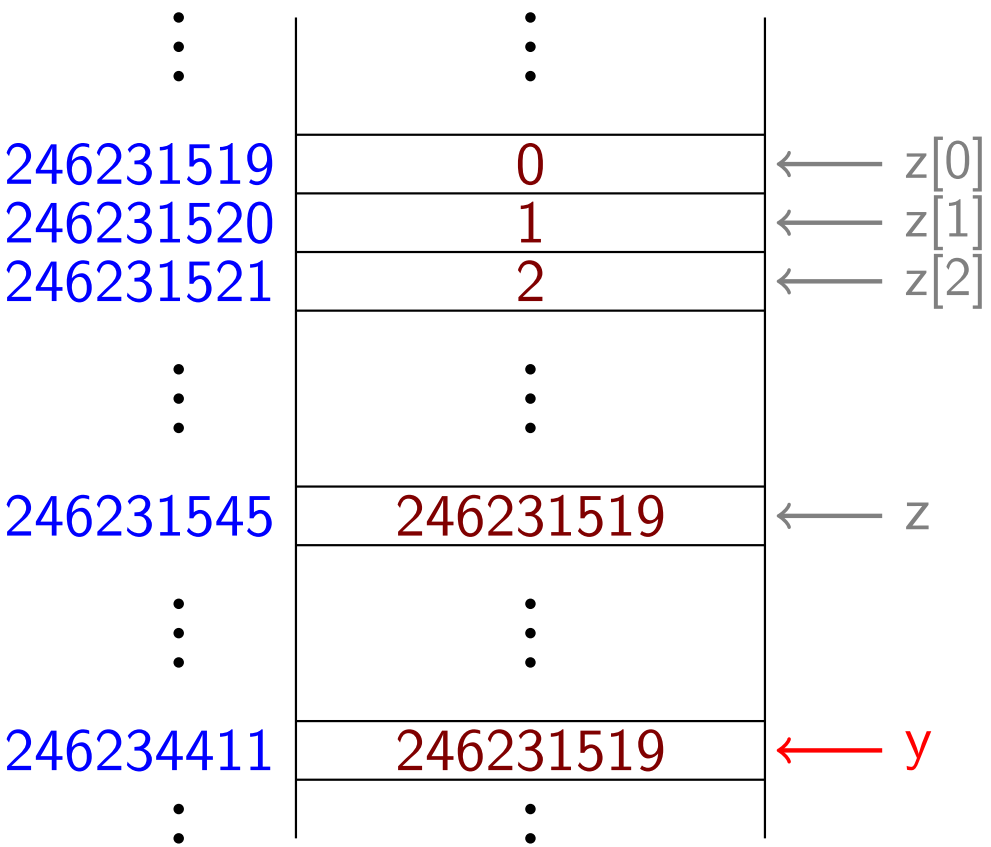



MEMORY

```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

PROGRAM

# Example

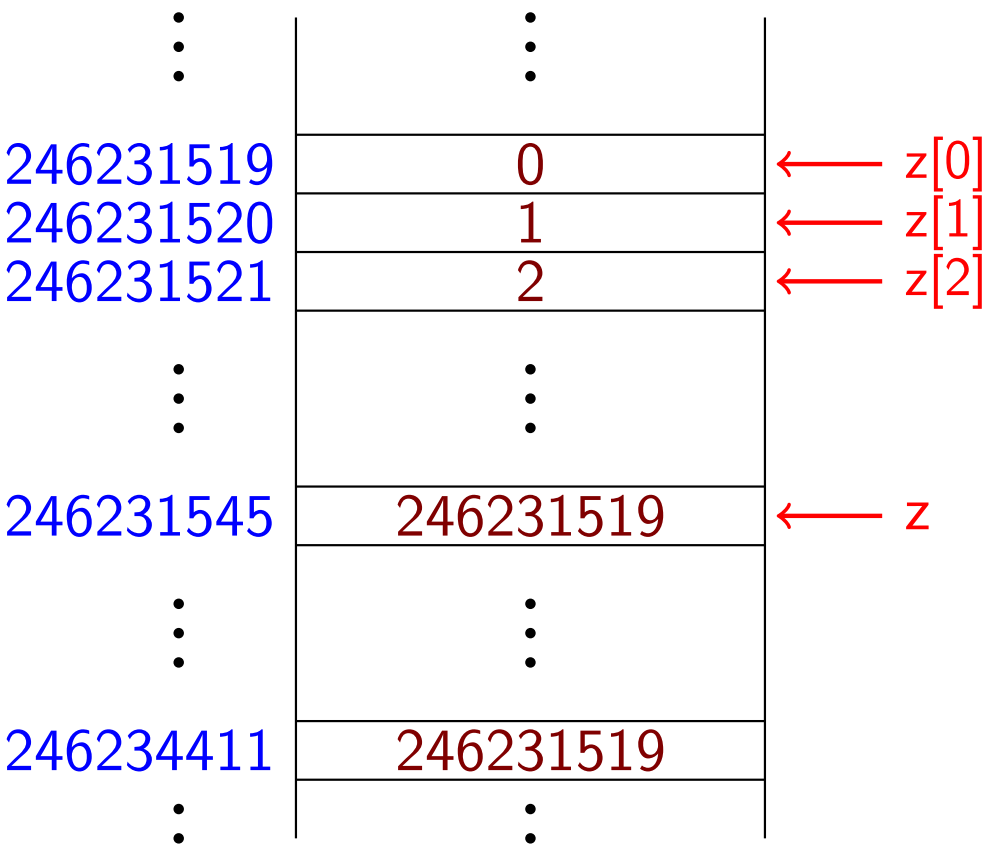



**MEMORY**

```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

**PROGRAM**

# Example




```
void foo( int y[] ) {
   for (int i=0;i<3;i++)
      y[i] = y[i] + i;
}
main() {
   int z[3];
   for (int i=0;i<3;i++)
      z[i] = 0;
   foo(z);
   /* do something */
}
```

**MEMORY** **PROGRAM**