CENG216 – Tutorial 02

C Basics

Mustafa Özuysal
mustafaozuysal@iyte.edu.tr

February 27, 2022

İzmir Institute of Technology

Primitive Types and Operators

Variables

In C, we have to define all variables before their first use.

```
<type> <identifier>; // or
<type> <identifier> = <initial value>;
```

Integral Types	Floating-Point Types
`char`, `short`, `int`, `long`	`float`, `double`

Note `char`, `short`, `int`, `long` also have `unsigned` versions.

Variables

```
1 int i = -123;
2 long l = 123L;
3 double d = 123.0;
4 float f = 123.0f;
5 double d2 = 1e-5;
6 unsigned int u = 12345;
7 char c = 'A';
8 const char s[] = "a string";
```

Operators

- Arithmetic: `+, -, *, \, %`

Operators

- Arithmetic: `+`, `-`, `*`, `\`, `%`
- Logical: `<`, `<=`, `>`, `>=`, `!=`, `==`, `||`, `&&`, `!`

Operators

- Arithmetic: `+, -, *, \, %`
- Logical: `<, <=, >, >=, !=, ==, ||, &&, !`
- Bitwise: `&, |, ^, ~, <<, >>`

Operators

- Arithmetic: `+`, `-`, `*`, `\`, `%`
- Logical: `<`, `<=`, `>`, `>=`, `!=`, `==`, `||`, `&&`, `!`
- Bitwise: `&`, `|`, `^`, `~`, `<<`, `>>`
- Assignment: `=`, `+=`, `-=`, `*=`, `...`

Operators

- Arithmetic: `+, -, *, \, %`
- Logical: `<, <=, >, >=, !=, ==, ||, &&, !`
- Bitwise: `&, |, ^, ~, <<, >>`
- Assignment: `=, +=, -=, *=, ...`
- Pre/post increment/decrement: `++,--`

Operators

- Arithmetic: `+, -, *, \, %`
- Logical: `<, <=, >, >=, !=, ==, ||, &&, !`
- Bitwise: `&, |, ^, ~, <<, >>`
- Assignment: `=, +=, -=, *=, ...`
- Pre/post increment/decrement: `++,--`
- Conditional: `<condition> ? <true-part> : <false-part>`

Control Flow and Functions

`if` Statement

```
if (<cond>)
    <statement>; // Executed only if <cond> is true
```

if Statement

```
if (<cond>)
    <statement>; // Executed only if <cond> is true
```

```
if (<cond>)
    <statement>; // Executed only if <cond> is true
else
    <statement>; // Executed only if <cond> is false
```

if Statement

```
if (<cond>)
    <statement>; // Executed only if <cond> is true
```

```
if (<cond>)
    <statement>; // Executed only if <cond> is true
else
    <statement>; // Executed only if <cond> is false
```

```
if (<cond1>)
    <statement>; // <cond1> true
else if (<cond2>)
    <statement>; // <cond1> false && <cond2> true
else
    <statement>; // <cond1> && <cond2> both false
```

`while` and `for` Statements

```
while (<cond>)
    <statement>; // Executed as long as <cond> is true
```

`while` and `for` Statements

```
while (<cond>)
    <statement>; // Executed as long as <cond> is true
```

```
for (<expr1>; <expr2>; <expr3>)
    <statement>; // Executed as long as <expr2> is true
```

`while` and `for` Statements

```
while (<cond>)
    <statement>; // Executed as long as <cond> is true
```

```
for (<expr1>; <expr2>; <expr3>)
    <statement>; // Executed as long as <expr2> is true
```

↓

```
<expr1>;
while (<expr2>) {
    <statement>;
    <expr3>;
}
```

Functions

```
<return-type>  <function-name>(<parameter-list>|<void>)
{
    <function-body>;
}
```

Example

Square-roots with Newton's method:

- Set initial `guess = 1.0` for the square root of x
- Improve guess with $\texttt{guess} \leftarrow \frac{\texttt{guess} + \frac{x}{\texttt{guess}}}{2}$
- Stop when $|\texttt{guess} * \texttt{guess} - x| < \tau$

Example

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "sqrt_newton.h"

int main(int argc, char** argv)
{
        double tol = 1.0e-2;
        if (argc >= 2) {
                tol = atof(argv[1]);
        }
        printf("Tolerance = %.2g\n", tol);

        for (double x = 0.0; x < 10.0; x += 1.0) {
                double c_sqrt = sqrt(x);
                double n_sqrt = sqrt_newton(x, tol);
                printf("x= %.1f, sqrt(x)= %.5f, sqrtn(x)= %.5f\n",
                        x, c_sqrt, n_sqrt);
        }

        return 0;
}
```

Example

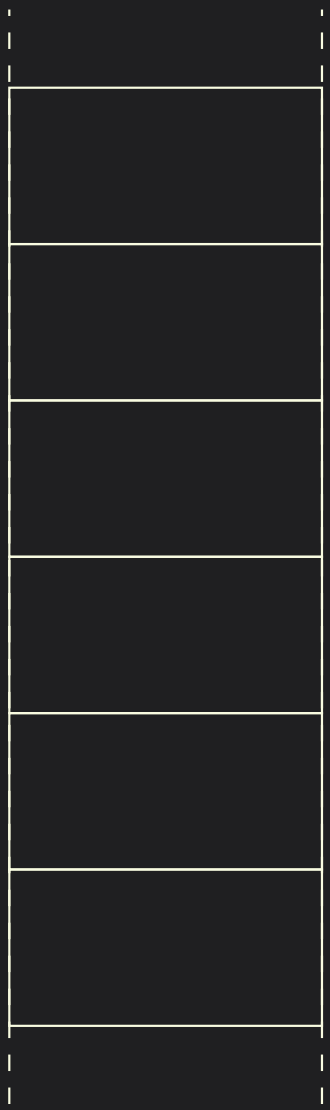
```
#ifndef SQRT_NEWTON_H
#define SQRT_NEWTON_H

double sqrt_newton(double y, double tol);

#endif
```

Example

```c
#include "sqrt_newton.h"

#include <math.h>
#include <stdbool.h>

bool sqrt_good_enough(double guess, double y, double tol)
{
        return fabs(guess * guess - y) < tol;
}

double sqrt_improve(double guess, double y)
{
        return (guess + y / guess) / 2.0;
}

double sqrt_newton(double y, double tol)
{
        double guess = 1.0;
        while (!sqrt_good_enough(guess, y, tol))
                guess = sqrt_improve(guess, y);
        return guess;
}
```

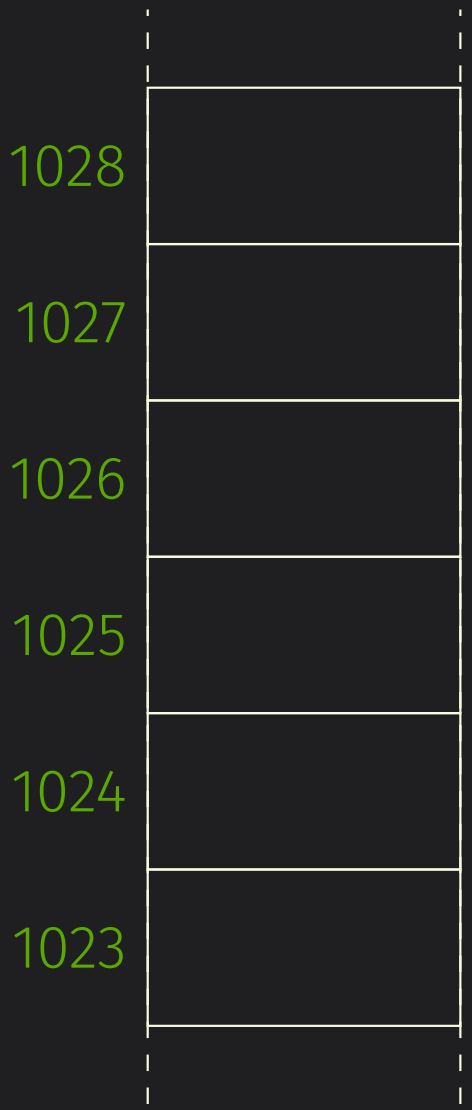
A Flat View of Computer Memory

The Memory



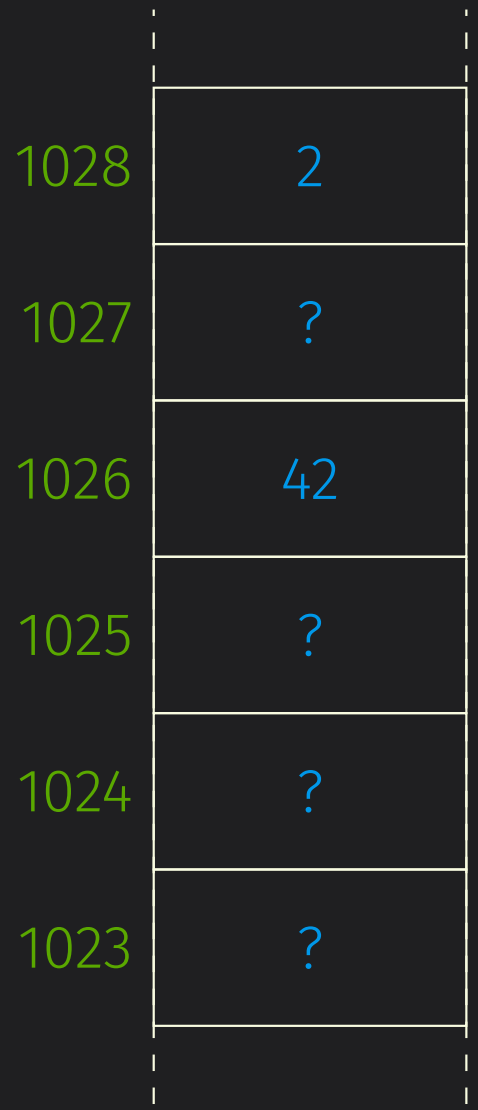
- Computer’s memory can be viewed as a set of boxes, each one of a fixed size.

The Memory



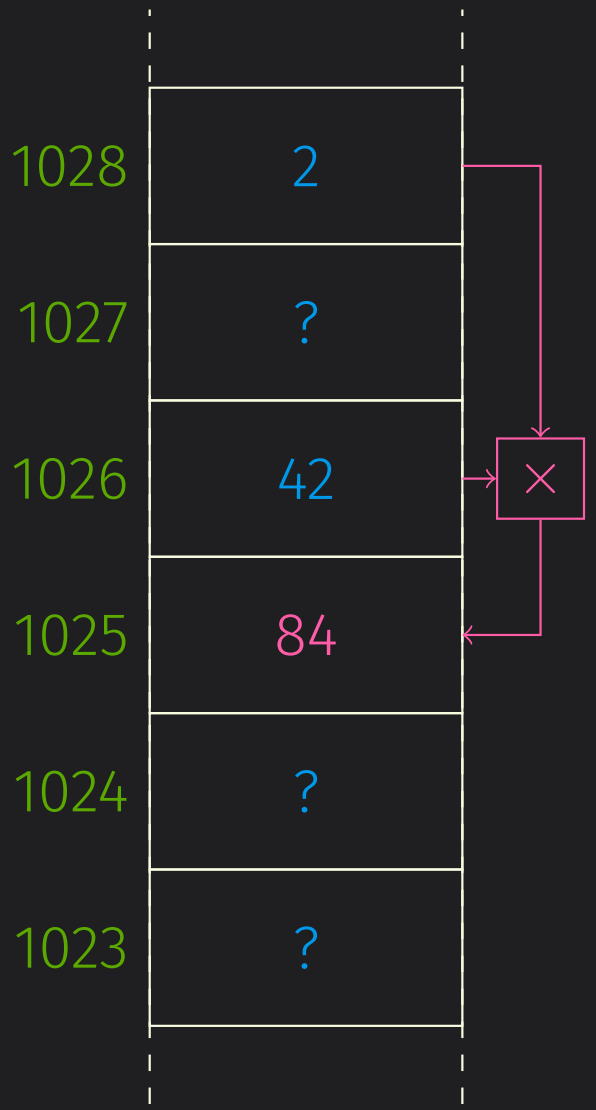
- Computer's memory can be viewed as a set of boxes, each one of a fixed size.
- Each box is referred to by a number which is called its address.

The Memory



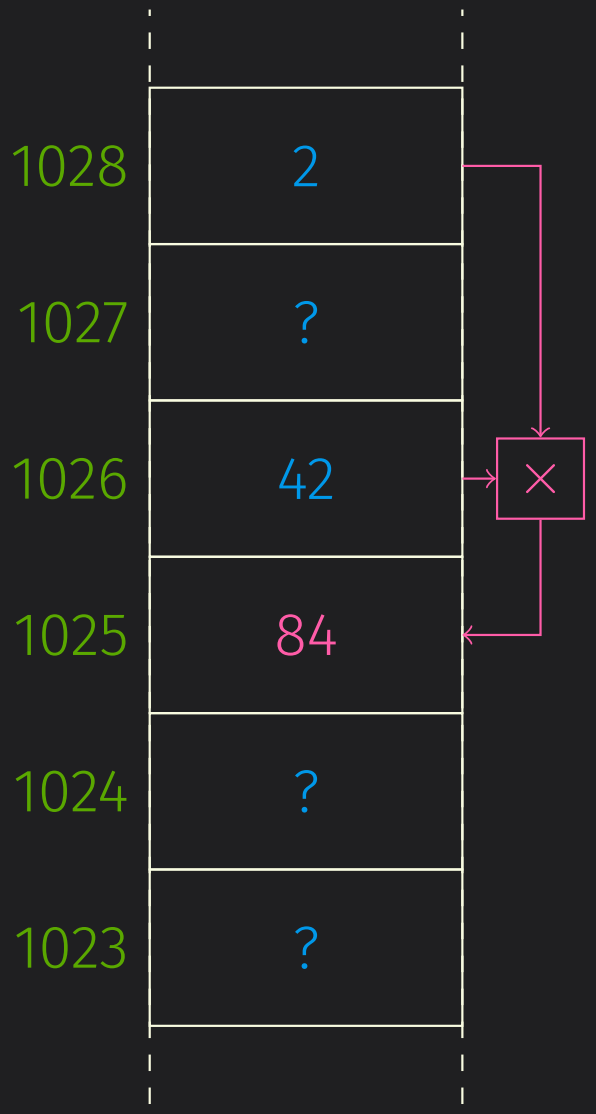
- Computer's memory can be viewed as a set of boxes, each one of a fixed size.
- Each box is referred to by a number which is called its address.
- Your program's data lives in these boxes.

The Memory



- Computer's memory can be viewed as a set of boxes, each one of a fixed size.
- Each box is referred to by a number which is called its address.
- Your program's data lives in these boxes.
- The Central Processing Unit (CPU) knows how to move and process the data in these boxes.

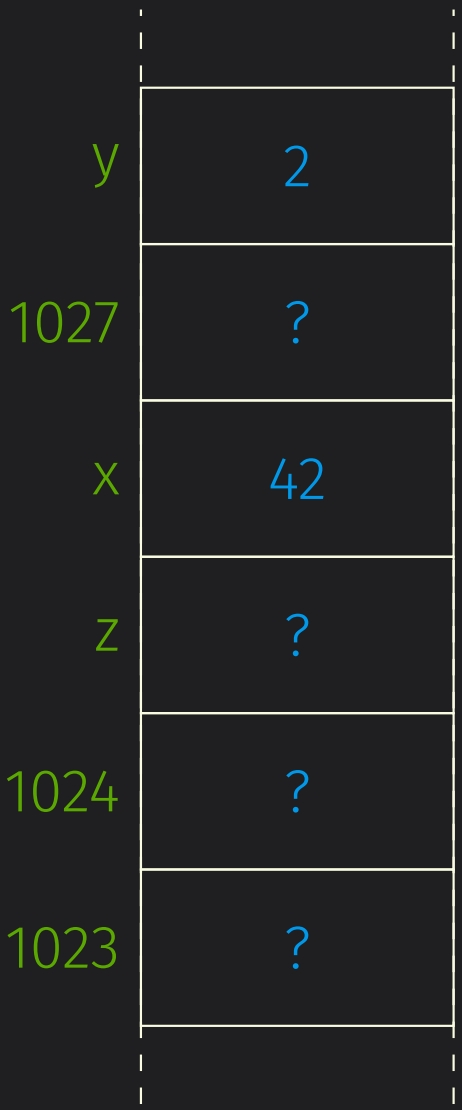
The Memory



- Computer's memory can be viewed as a set of boxes, each one of a fixed size.
- Each box is referred to by a number which is called its address.
- Your program's data lives in these boxes.
- The Central Processing Unit (CPU) knows how to move and process the data in these boxes.
- Back in the old days, people had to program using addresses:

```
mul $1026,$1028,$1025
```

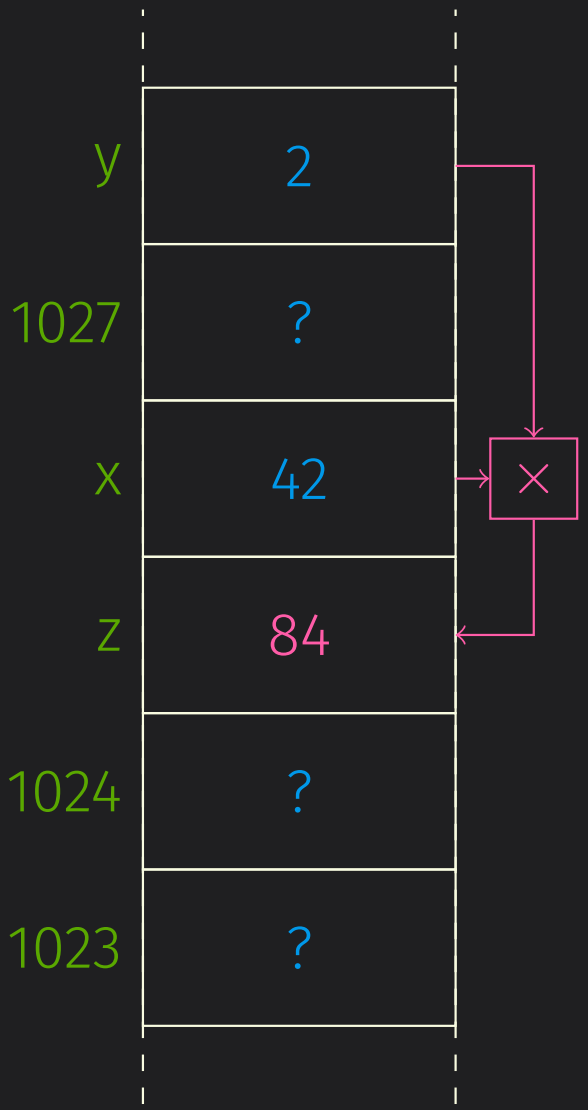
Variables



- Modern languages let us create variables that gets assigned to boxes by the compiler:

```
1  int x = 42; int y = 2; int z;
```

Variables



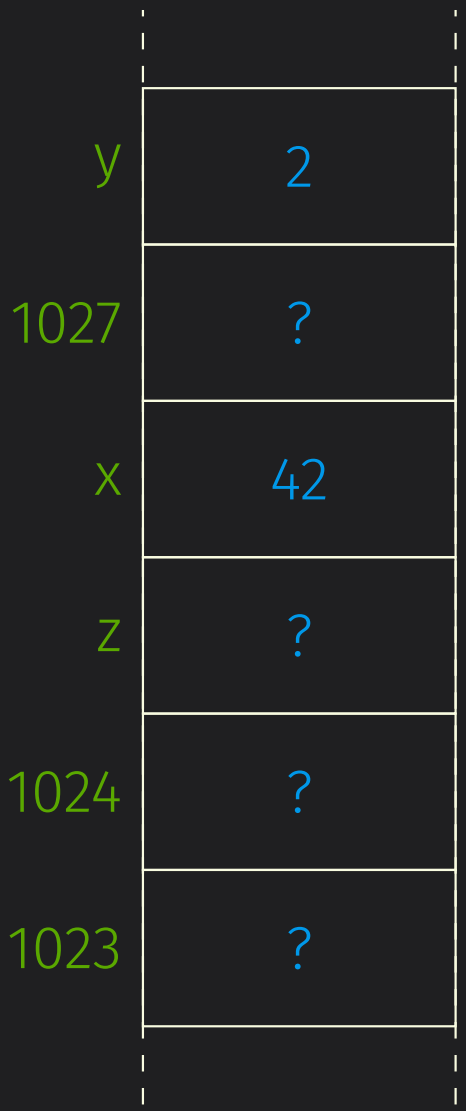
- Modern languages let us create variables that gets assigned to boxes by the compiler:

```
1  int x = 42; int y = 2; int z;
```

- When we write code operating on these variables, compiler and linker convert the variables to addresses automatically:

```
1  z = x * y;
```

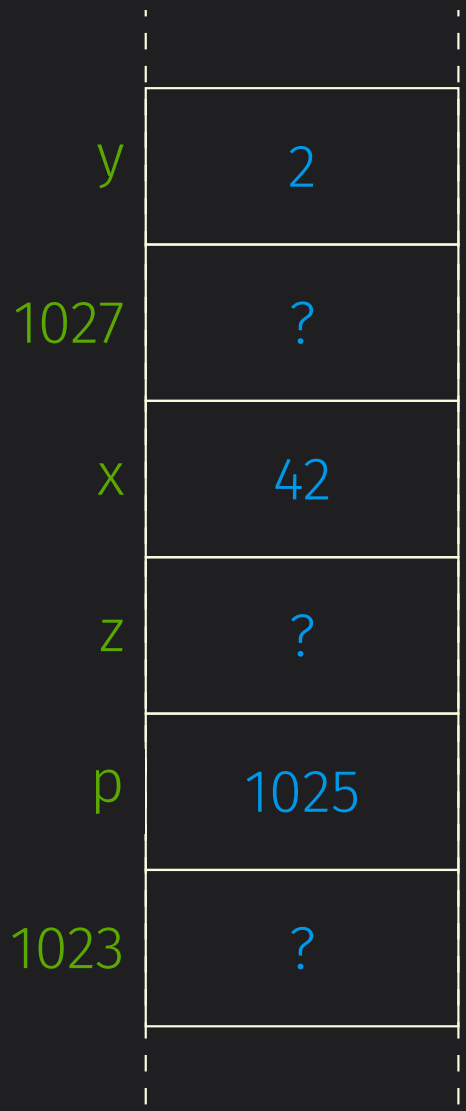
Pointers



- We can get the address of a variable with the & operator.:

```
1  int x = 42; int y = 2; int z;
2  // This should print 1025 in base-16
3  printf("%p\n", &z);
```

Pointers



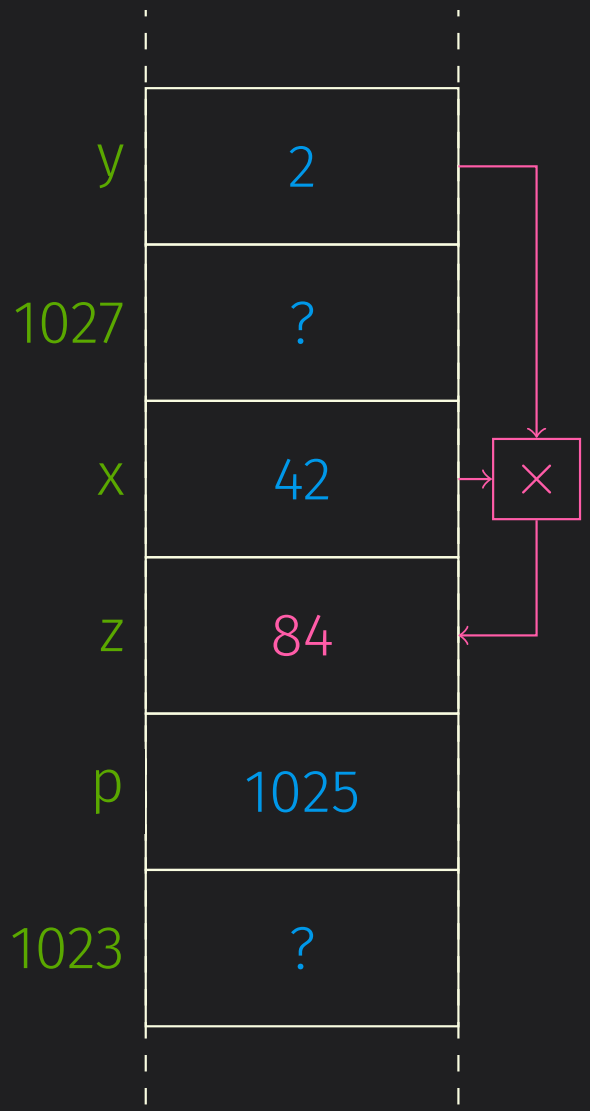
- We can get the address of a variable with the & operator.:

```
1  int x = 42; int y = 2; int z;
2  // This should print 1025 in base-16
3  printf("%p\n", &z);
```

- A pointer is a variable that can store addresses:

```
1  int *p = &z;
```

Pointers



- We can get the address of a variable with the & operator.:

```
1  int x = 42; int y = 2; int z;
2  // This should print 1025 in base-16
3  printf("%p\n", &z);
```

- A pointer is a variable that can store addresses:

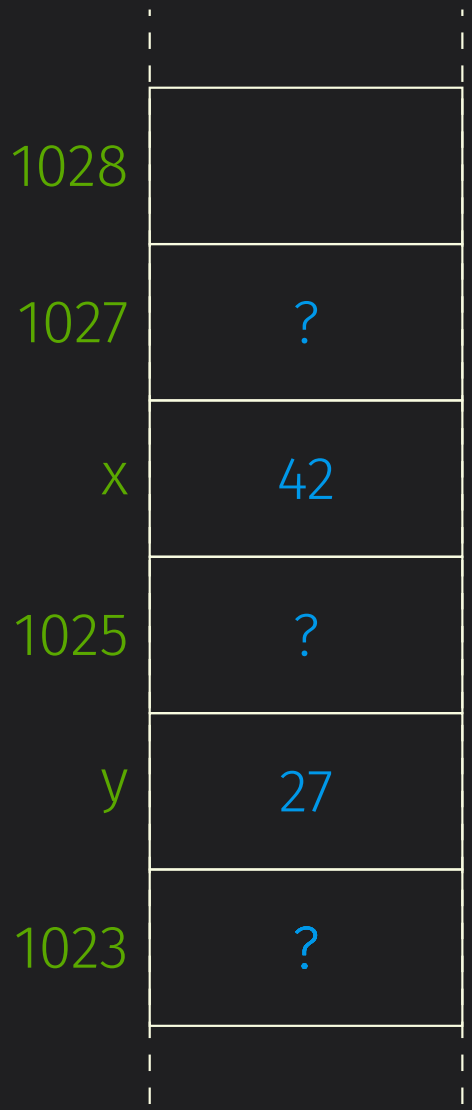
```
1  int *p = &z;
```

- The contents of the box whose address is stored in a pointer can be reached with the ∗ operator:

```
1  *p = x * y;
```

Pointers and Functions

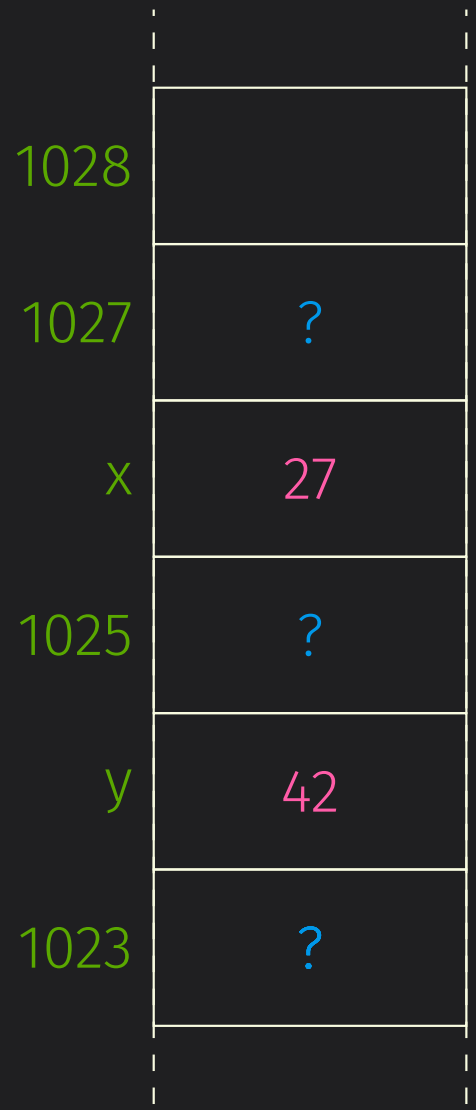
Example: The `swap` Function



- We want to write a function **`swap`** that exchanges the values of two variables:

```
1  int main(int argc, char **argv) {
2    int x = 42;
3    int y = 27;
4  }
```

Example: The `swap` Function



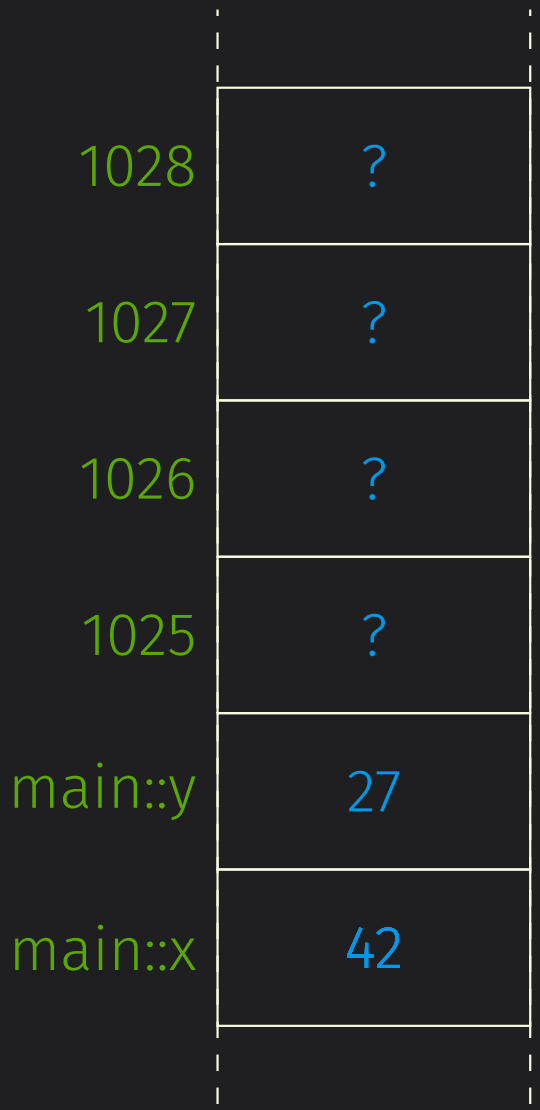
- We want to write a function **`swap`** that exchanges the values of two variables:

```
1  int main(int argc, char **argv) {
2    int x = 42;
3    int y = 27;
4  }
```

- After the **`swap`** call the variables should have the previous values of each other.

```
1  swap(x, y);
```

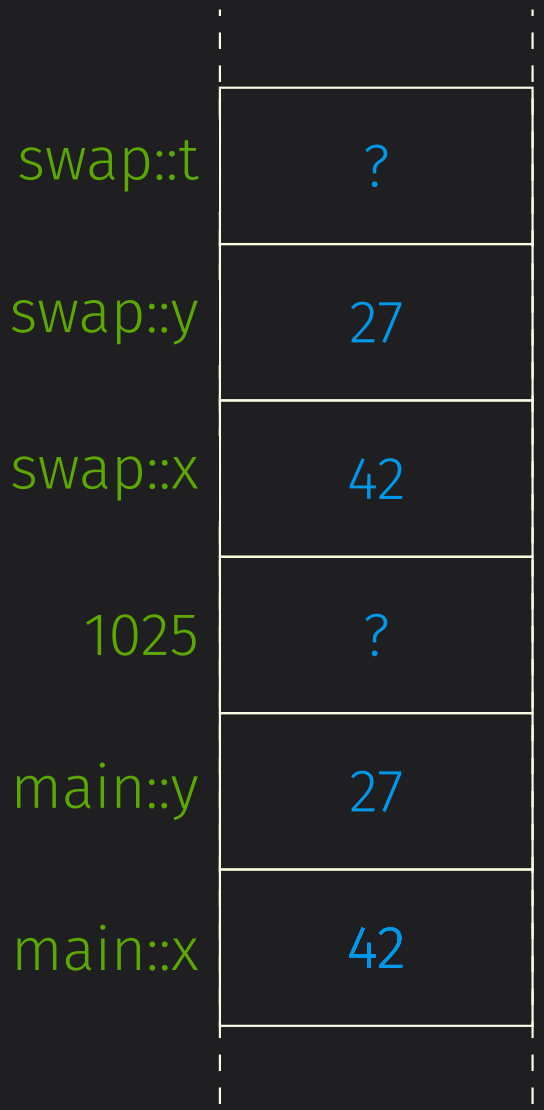
Example: `swap`, A First Try



- The following DOES NOT work since `x` and `y` inside the function are local variables:

```
1  void swap(int x, int y) {
2    int t = x;
3    x = y;
4    y = t;
5  }
```

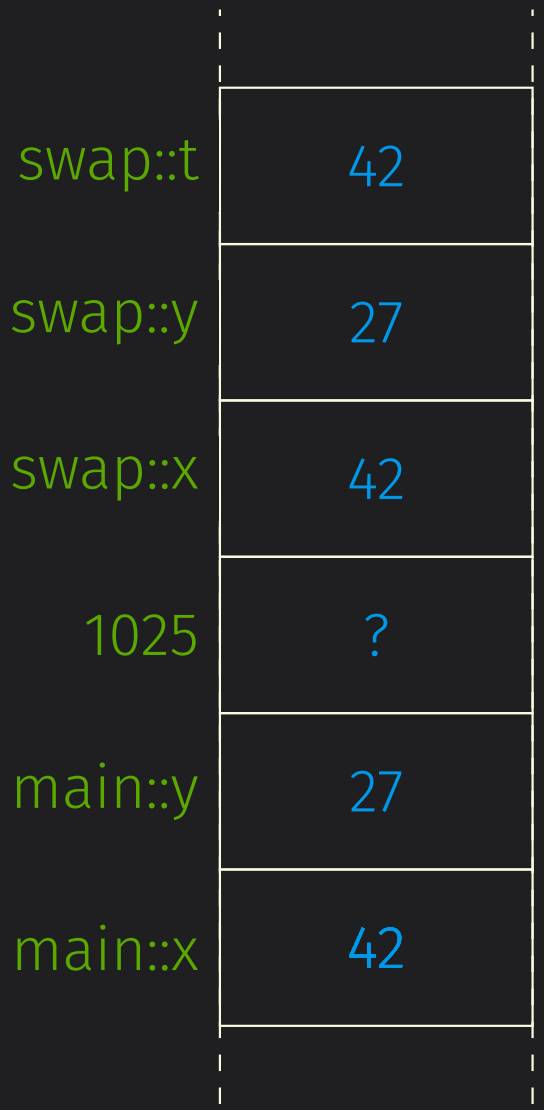
Example: `swap`, A First Try



- The following DOES NOT work since `x` and `y` inside the function are local variables:

```
1  void swap(int x, int y) {
2    int t = x;
3    x = y;
4    y = t;
5  }
```

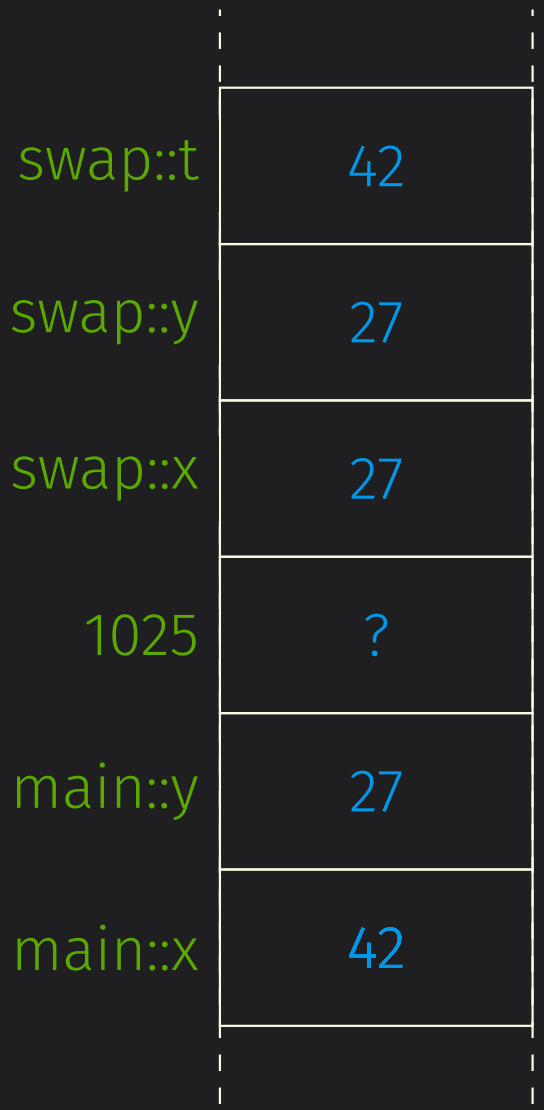
Example: `swap`, A First Try



- The following DOES NOT work since `x` and `y` inside the function are local variables:

```
1  void swap(int x, int y) {
2    int t = x;
3    x = y;
4    y = t;
5  }
```

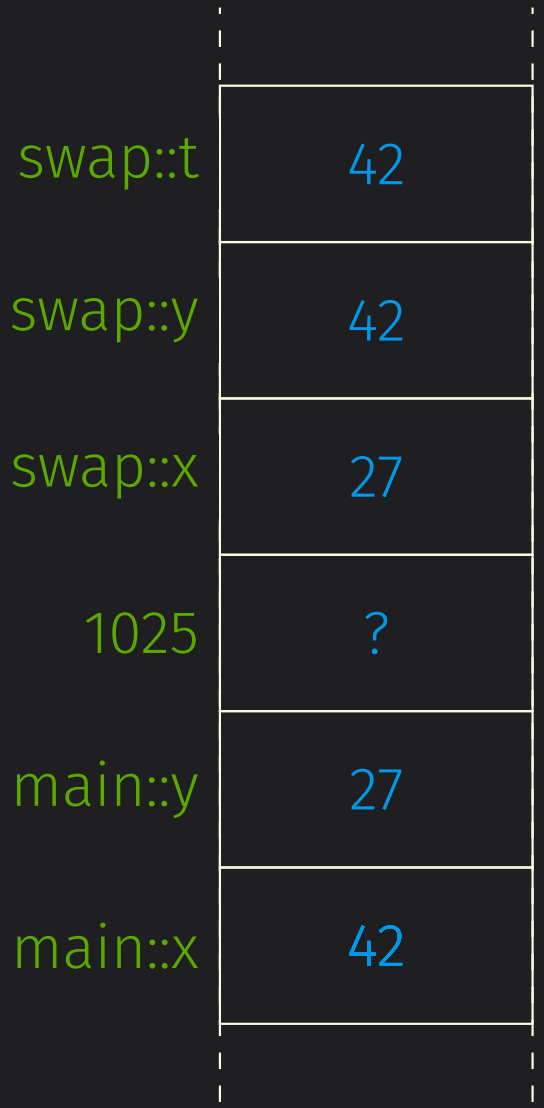
Example: `swap`, A First Try



- The following DOES NOT work since `x` and `y` inside the function are local variables:

```
1  void swap(int x, int y) {
2    int t = x;
3    x = y;
4    y = t;
5  }
```

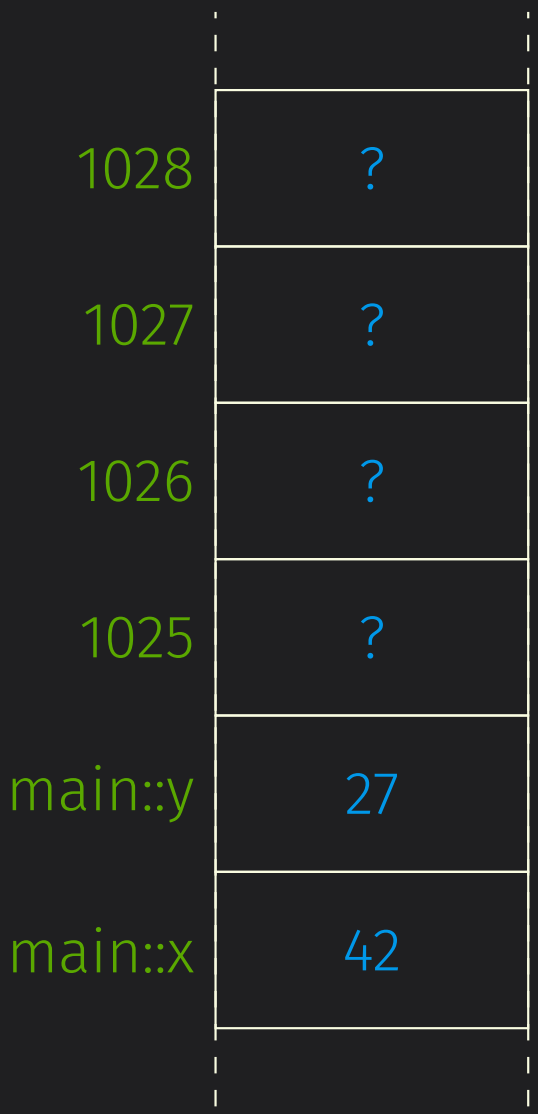
Example: `swap`, A First Try



- The following DOES NOT work since `x` and `y` inside the function are local variables:

```
1  void swap(int x, int y) {
2    int t = x;
3    x = y;
4    y = t;
5  }
```

Example: `swap` with Pointers



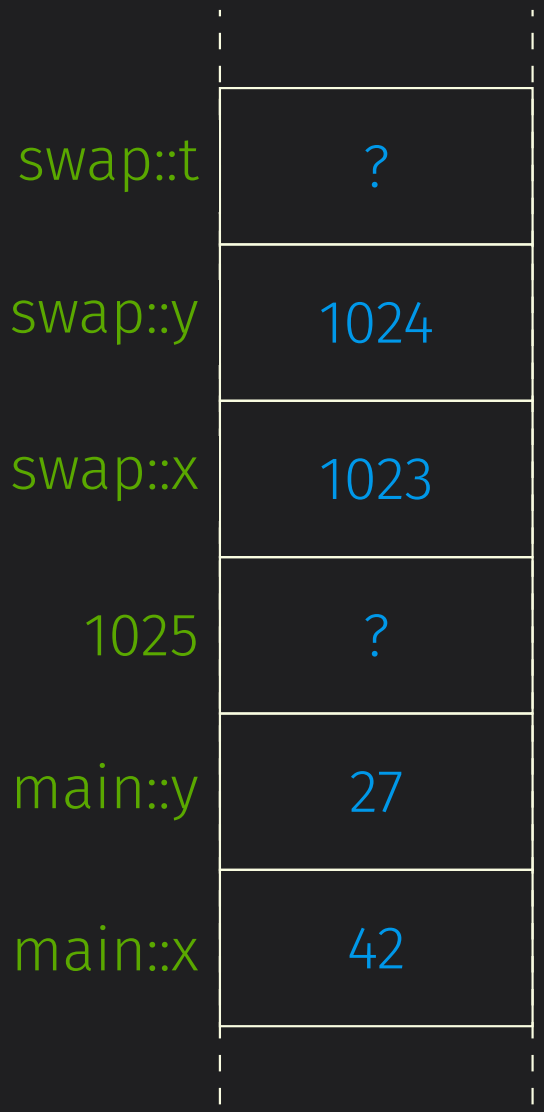
- The following with pointer works since `x` and `y` inside the function are pointers to the `x` and `y` in `main`:

```
void swap(int *x, int *y) {
  int t = *x;
  *x = *y;
  *y = t;
}
```

- We need to call this version of swap with addresses of the variables:

```
1  int main(int argc, char **argv) {
2    int x = 42;
3    int y = 27;
4    swap(&x, &y);
5  }
```

Example: `swap` with Pointers



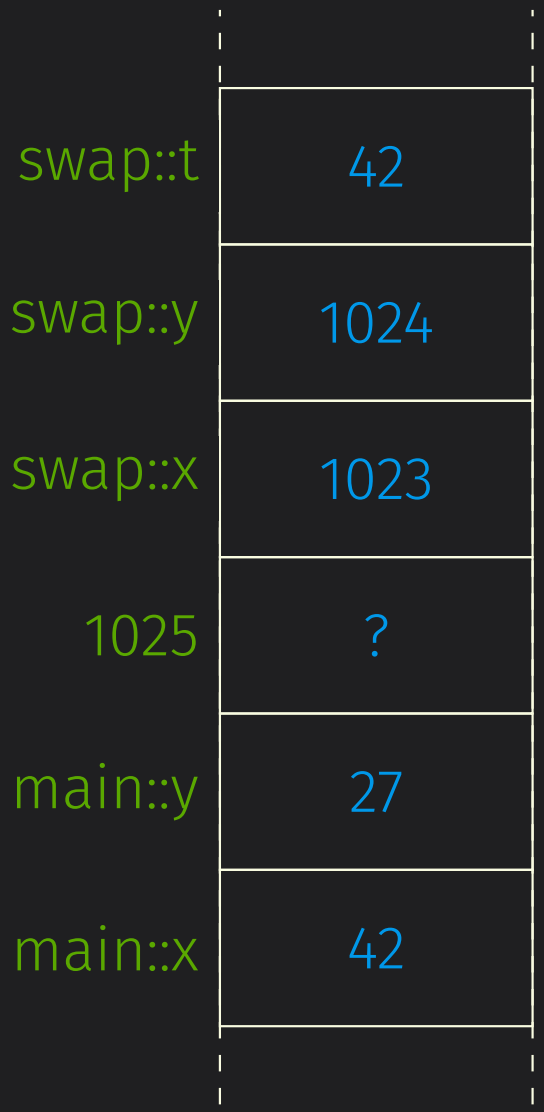
- The following with pointer works since `x` and `y` inside the function are pointers to the `x` and `y` in `main`:

```
void swap(int *x, int *y) {
  int t = *x;
  *x = *y;
  *y = t;
}
```

- We need to call this version of swap with addresses of the variables:

```
1  int main(int argc, char **argv) {
2    int x = 42;
3    int y = 27;
4    swap(&x, &y);
5  }
```

Example: `swap` with Pointers



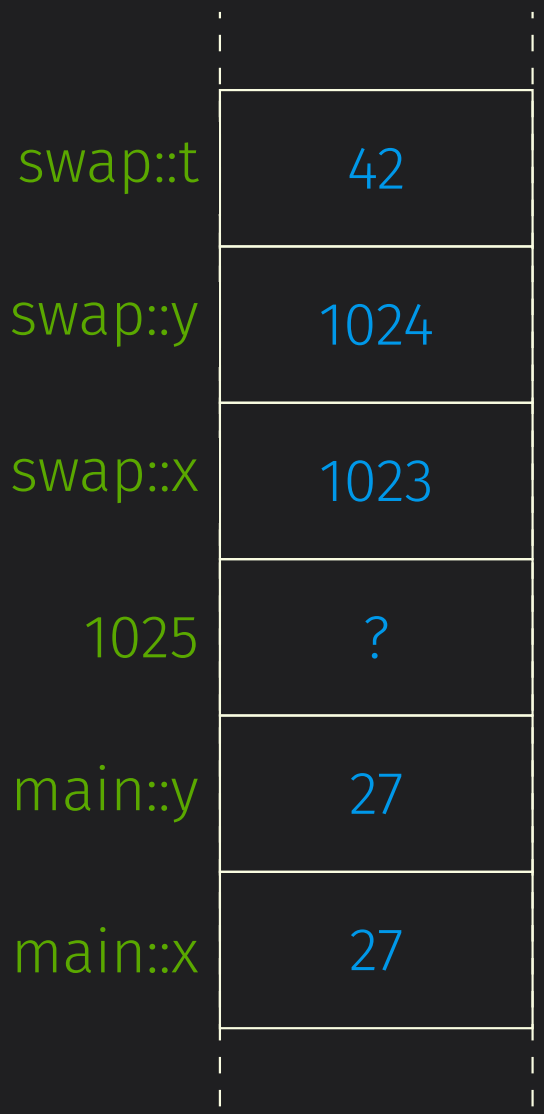
- The following with pointer works since `x` and `y` inside the function are pointers to the `x` and `y` in `main`:

```
void swap(int *x, int *y) {
  int t = *x;
  *x = *y;
  *y = t;
}
```

- We need to call this version of swap with addresses of the variables:

```
1  int main(int argc, char **argv) {
2    int x = 42;
3    int y = 27;
4    swap(&x, &y);
5  }
```

Example: `swap` with Pointers



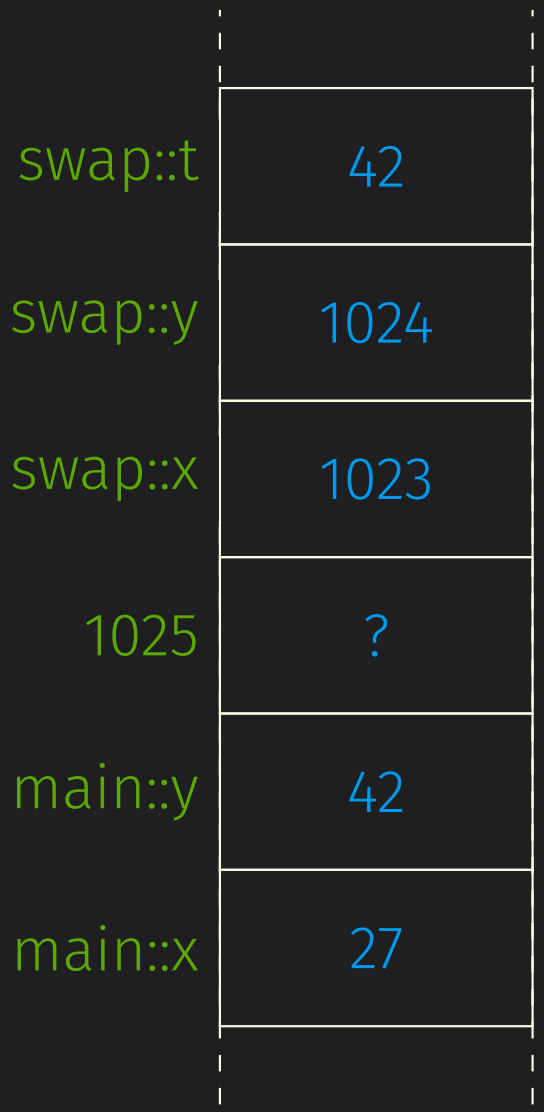
- The following with pointer works since `x` and `y` inside the function are pointers to the `x` and `y` in `main`:

```
void swap(int *x, int *y) {
  int t = *x;
  *x = *y;
  *y = t;
}
```

- We need to call this version of swap with addresses of the variables:

```
1  int main(int argc, char **argv) {
2    int x = 42;
3    int y = 27;
4    swap(&x, &y);
5  }
```

Example: `swap` with Pointers



- The following with pointer works since `x` and `y` inside the function are pointers to the `x` and `y` in `main`:

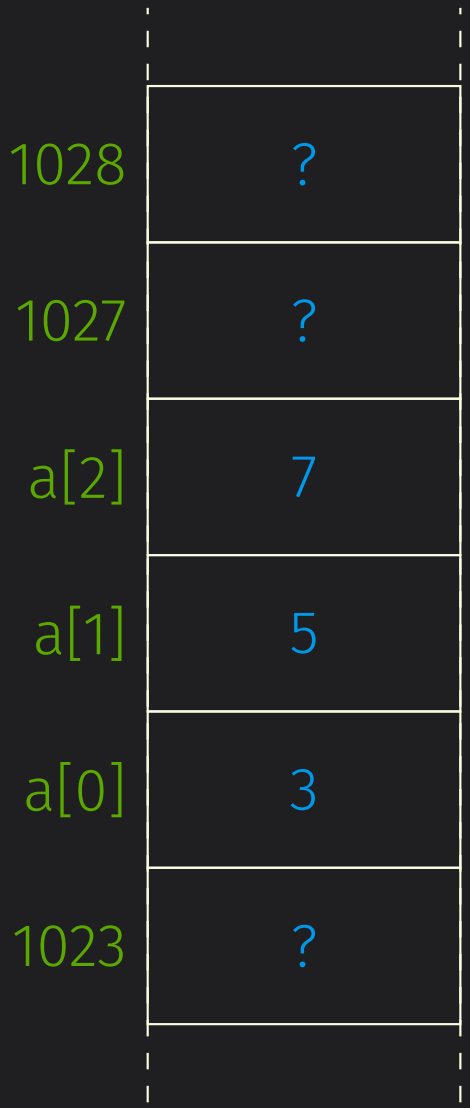
```
void swap(int *x, int *y) {
  int t = *x;
  *x = *y;
  *y = t;
}
```

- We need to call this version of swap with addresses of the variables:

```
1  int main(int argc, char **argv) {
2    int x = 42;
3    int y = 27;
4    swap(&x, &y);
5  }
```

Arrays and Pointers

Fixed Size Arrays

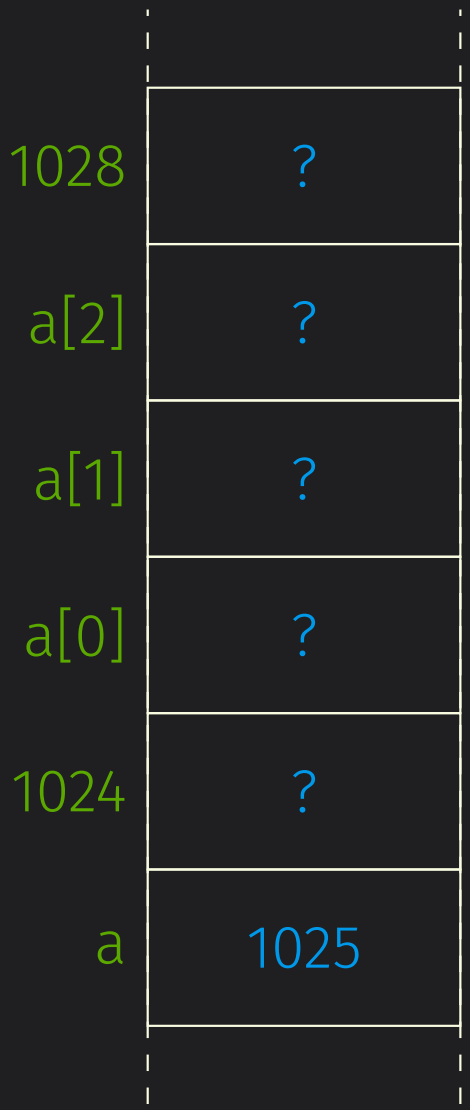


- When you need to store several items of the same type, you can declare an array variable:

```
1  int a[3] = {3,5,7};
```

- The size of the array needs to be a constant.
- The valid indices range from zero to size minus one.
- Accessing an item with a negative index or an index above or equal to size may lead to a crash of your program or it might just corrupt your data.

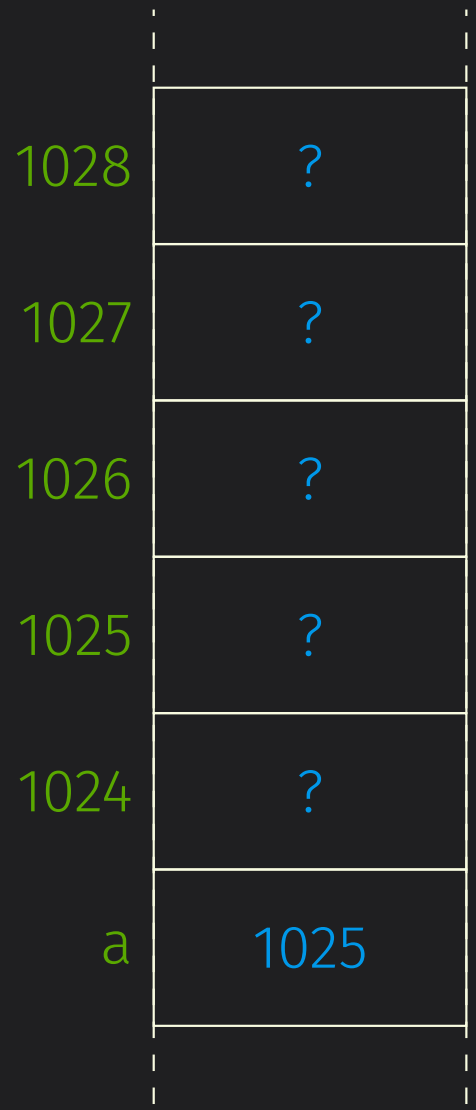
Dynamically Allocated Arrays



- When the array size is a variable quantity, you need to allocate the necessary memory yourself with the **`new`** operator:

```
1   int *a = (int *)malloc(3 * sizeof(int));
```

Dynamically Allocated Arrays



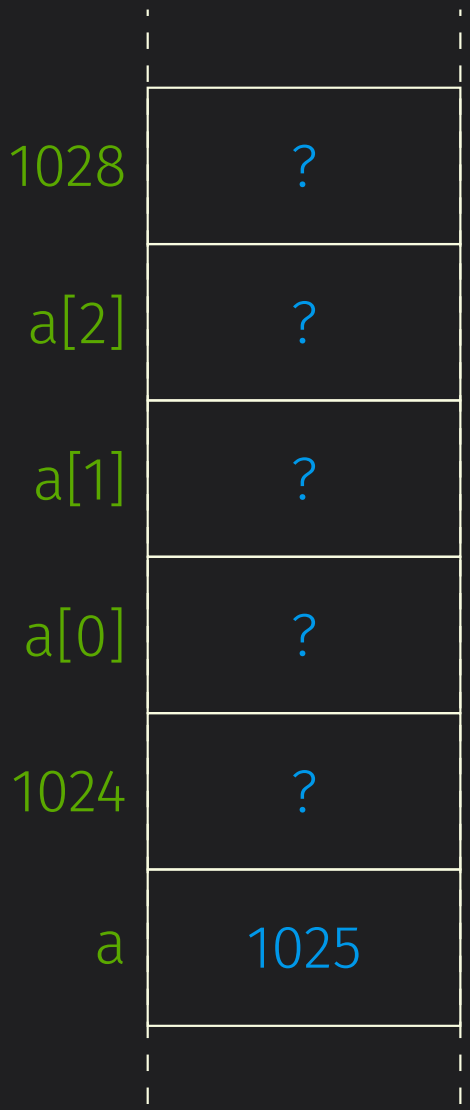
- When the array size is a variable quantity, you need to allocate the necessary memory yourself with the `new` operator:

```
1  int *a = (int *)malloc(3 * sizeof(int));
```

- Once you do not need the array, you need to deallocate the memory or a memory leak will occur:

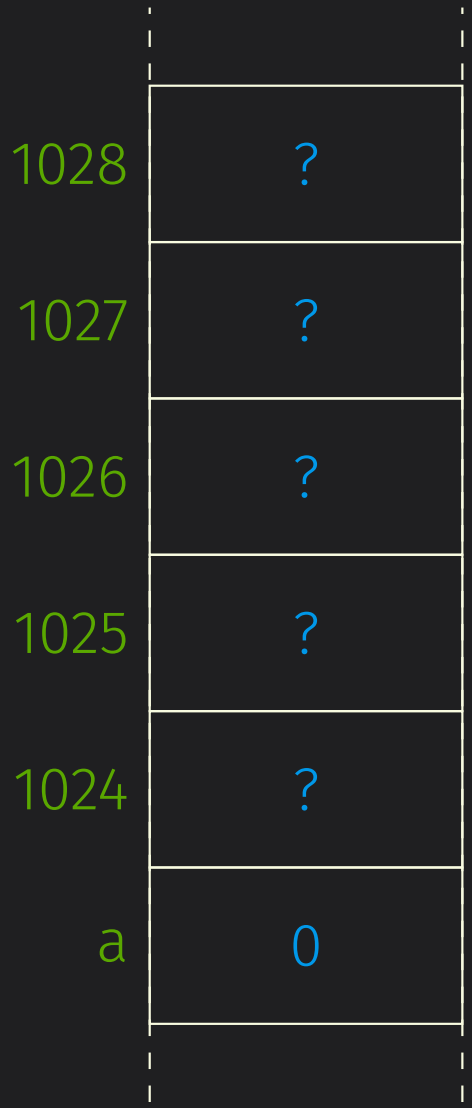
```
1  free(a);
```

Deallocation of Arrays



- It is an error to read/write to the array after deallocation.
- It is also an error to deallocate the same memory more than once.

Deallocation of Arrays

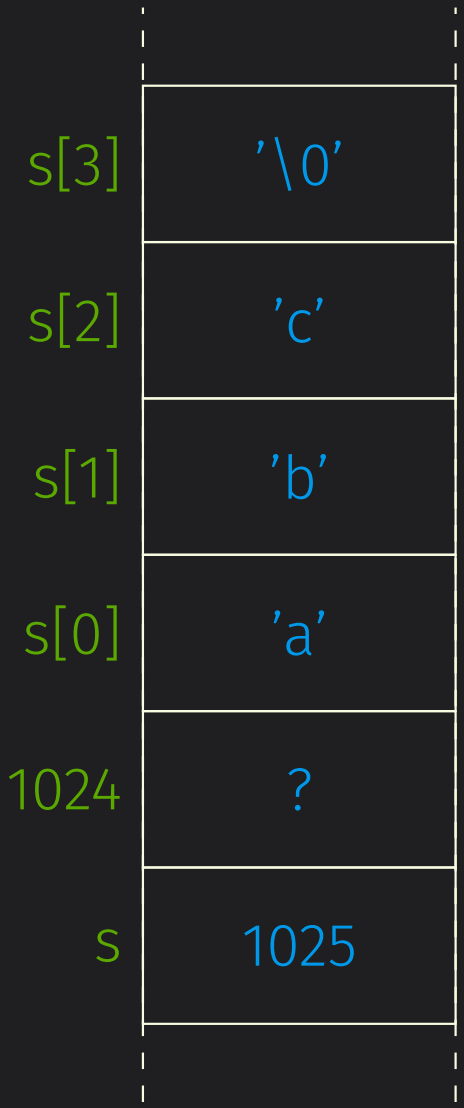


- It is an error to read/write to the array after deallocation.
- It is also an error to deallocate the same memory more than once.
- When you deallocate the memory, it is a good idea to the pointer to `NULL`, which is called a null pointer.

```
1  int *a = (int *)malloc(3 * sizeof(int));
2  free(a);
3  a = NULL;
```

C Strings

Null-Terminated Strings



- In C, the strings are simply arrays of characters with a null (zero) chracter at the end:

```
1  const char s[] = "abc";
```

- Equivalently, you can write:

```
1  const char *s = "abc";
```

- You can use the **`strlen`** function to get the length of a string.

```
1  strlen("abc") == 3 -> true
```

C String Example

```
9
10      int length = strlen(s1) + strlen(s2) + 1;
11      char *s = (char *)malloc(length * sizeof(char));
12      strcpy(s, s1);
13      strcat(s, s2);
14      printf("Combined String: %s\n", s);
15      free(s);
16
```

```
$ ./string-test
Hello World!
```