CENG 112 – Data Structures

Pointers and Arrays

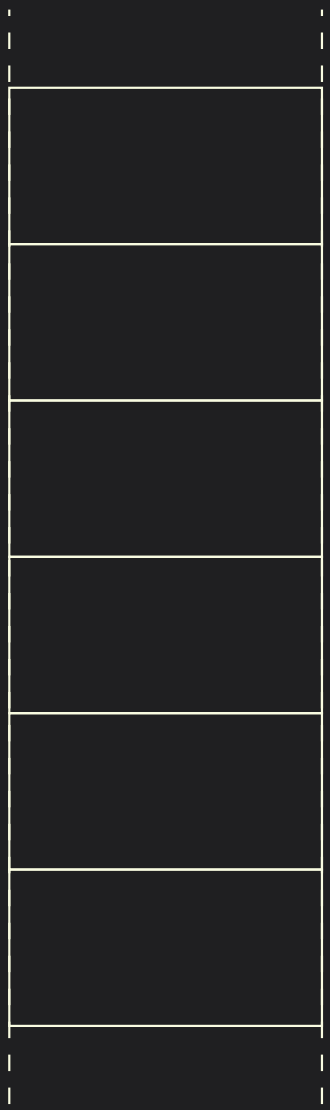
Mustafa Özuysal
mustafaozuysal@iyte.edu.tr

March 2, 2017

İzmir Institute of Technology

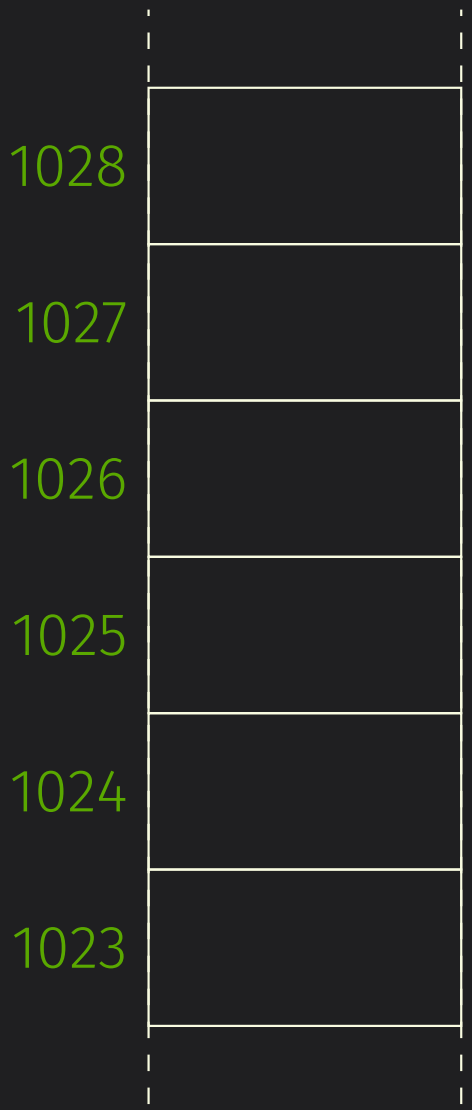
A Flat View of Computer Memory

The Memory



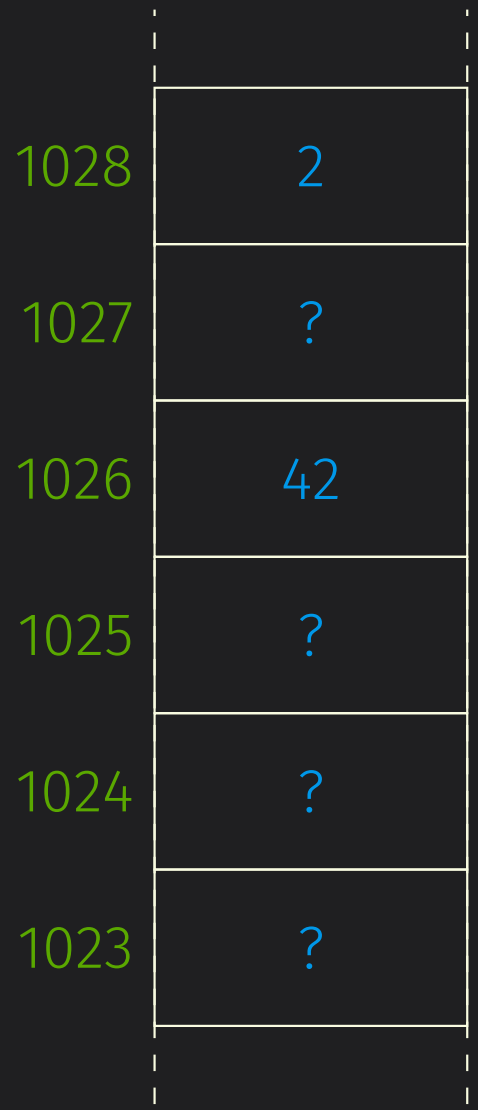
- Computer’s memory can be viewed as a set of boxes, each one of a fixed size.

The Memory



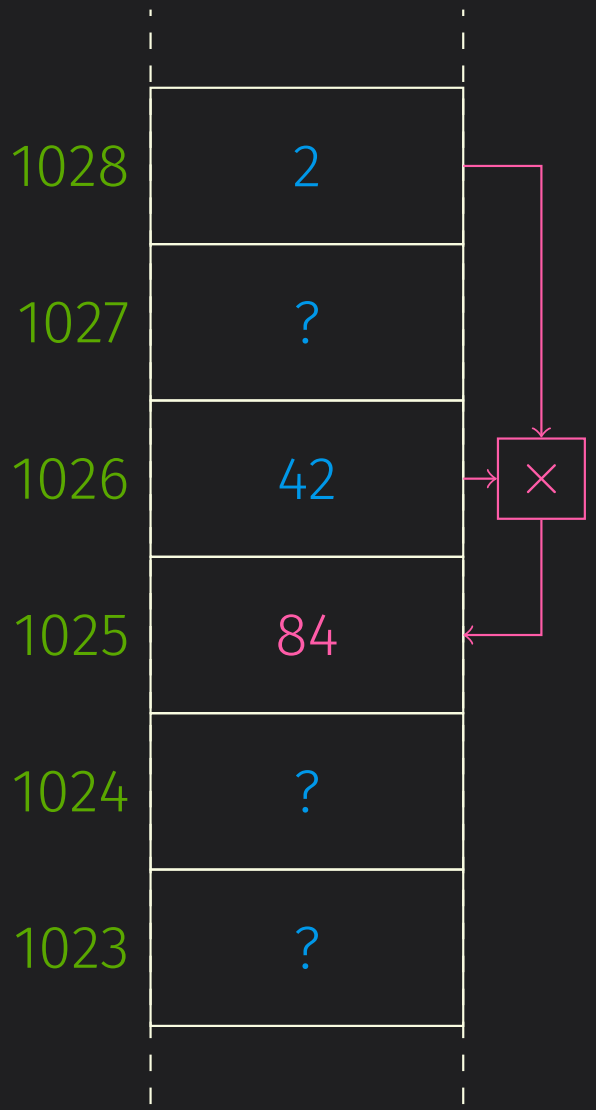
- Computer's memory can be viewed as a set of boxes, each one of a fixed size.
- Each box is referred to by a number which is called its address.

The Memory



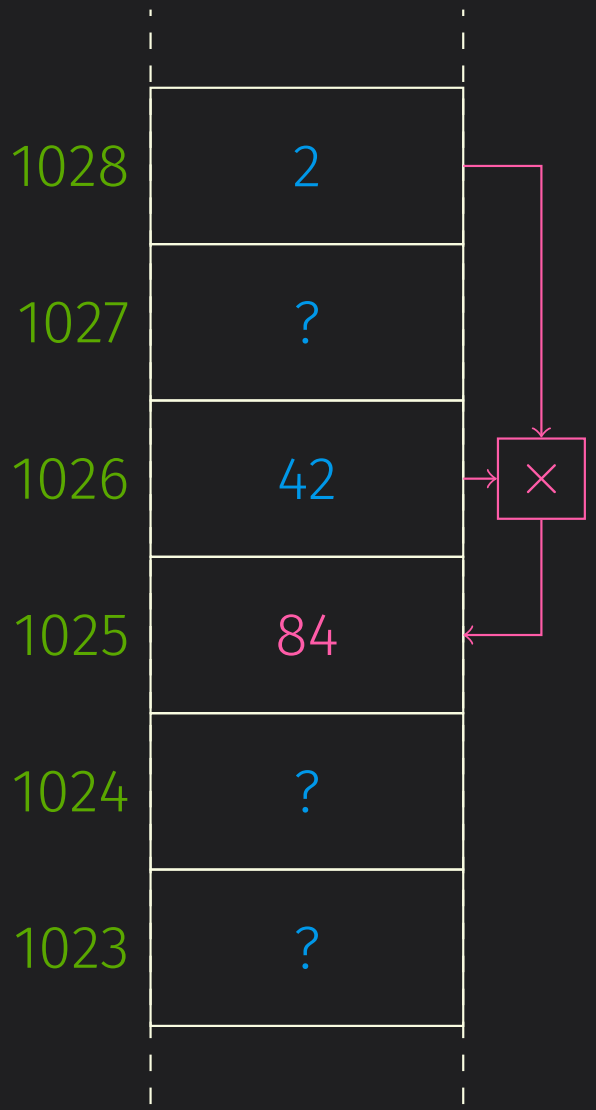
- Computer's memory can be viewed as a set of boxes, each one of a fixed size.
- Each box is referred to by a number which is called its address.
- Your program's data lives in these boxes.

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.

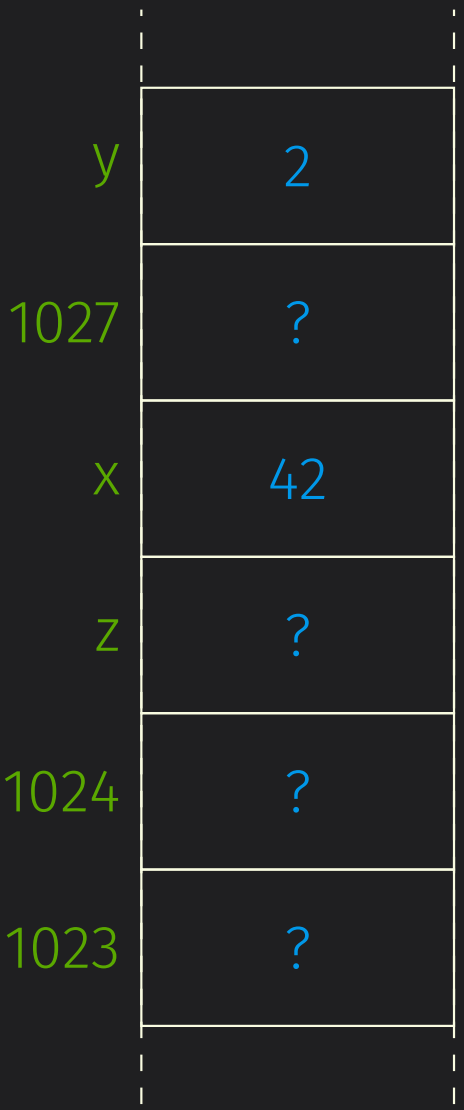
The Memory



- 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
```

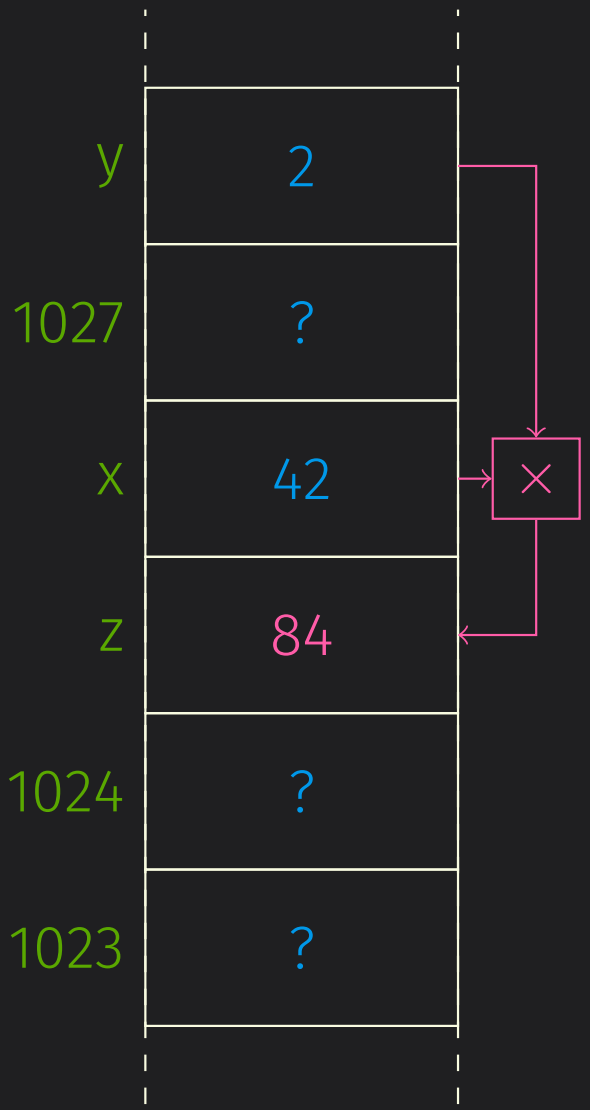
Variables



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

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

Variables



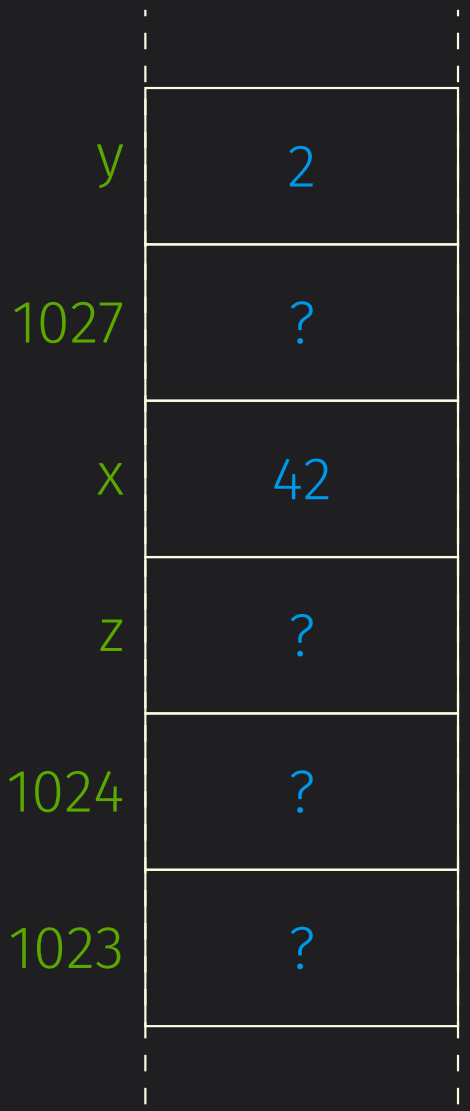
- Modern languages let us create variables that gets assigned to boxes by the compiler:

```
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;
```

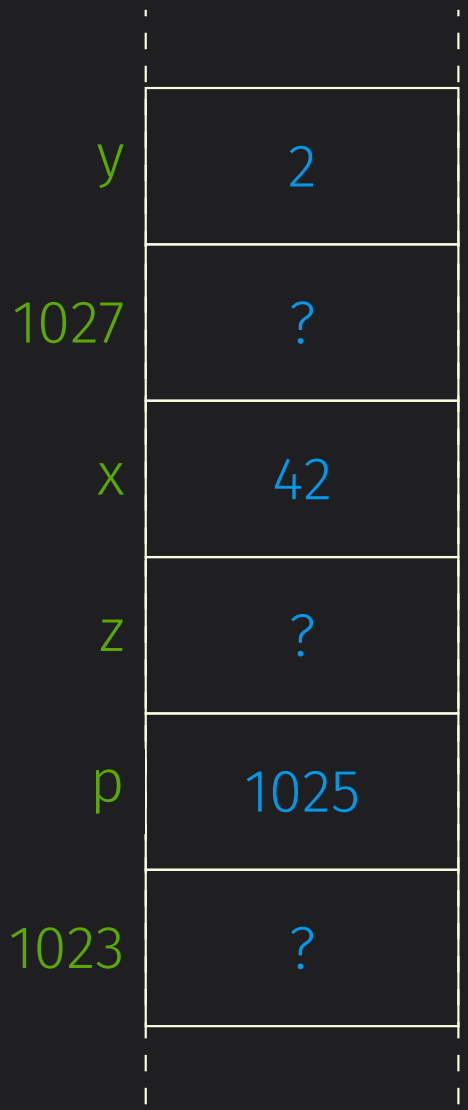
Pointers



- 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
3  cout << &z << endl;
```

Pointers



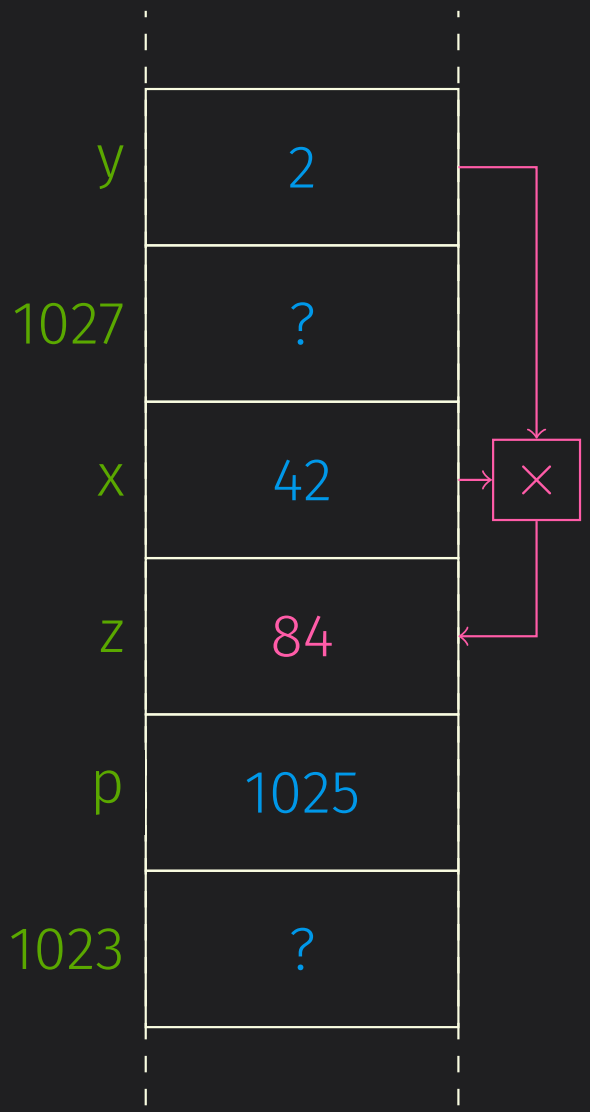
- We can get the address of a variable with the & operator.:

```
1  int x = 42; int y = 2; int z;
2  // This should print 1025
3  cout << &z << endl;
```

- 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
3  cout << &z << endl;
```

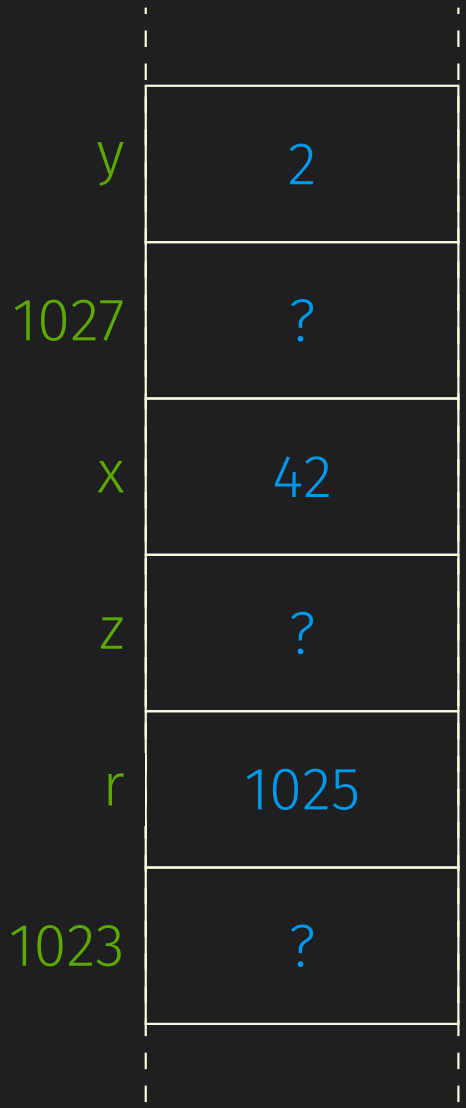
- A pointer is a variable that can store addresses:

```
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;
```

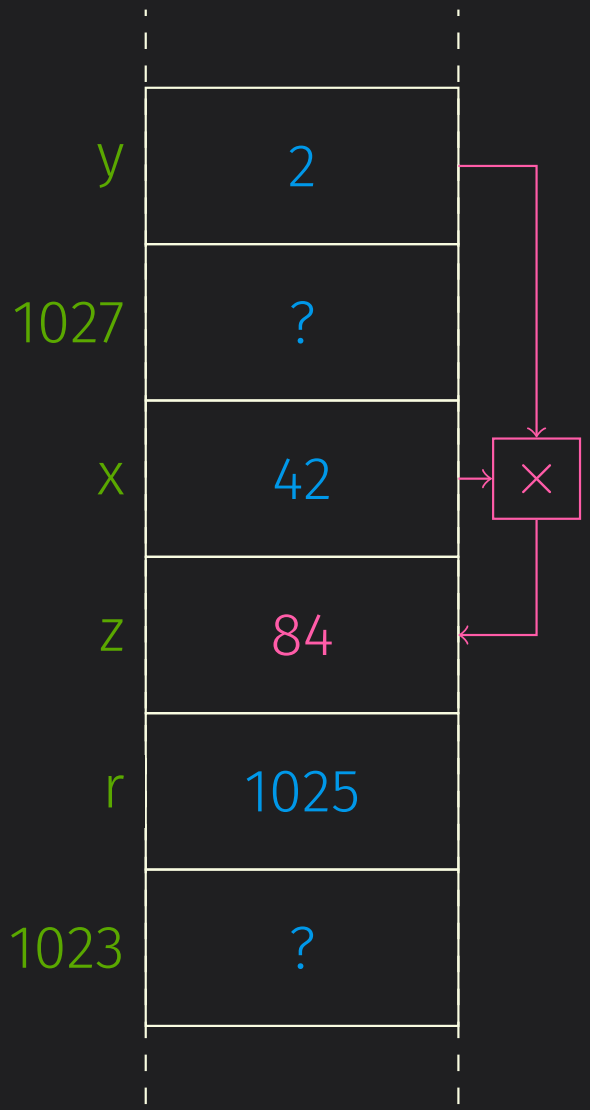
References (C++ only)



- A **reference** is a special variable that refers to another variable. This is just a short-hand notation.

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

References (C++ only)



- A **reference** is a special variable that refers to another variable. This is just a short-hand notation.

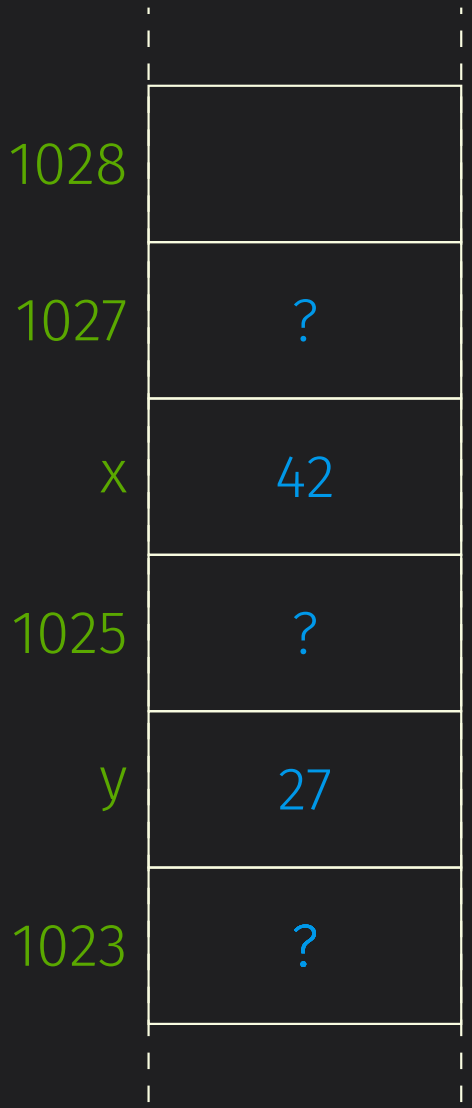
```
1  int x = 42; int y = 2; int z;
2  int &r = z;
```

- References lets us use simpler syntax for the same effect:

```
1  r = x * y;
```

Pointers and Functions

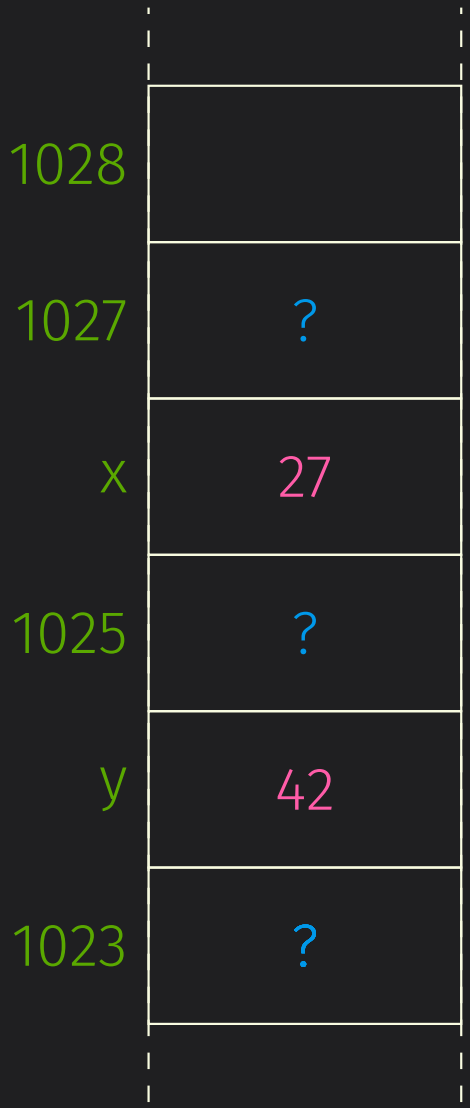
Example: The swap Function



- 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



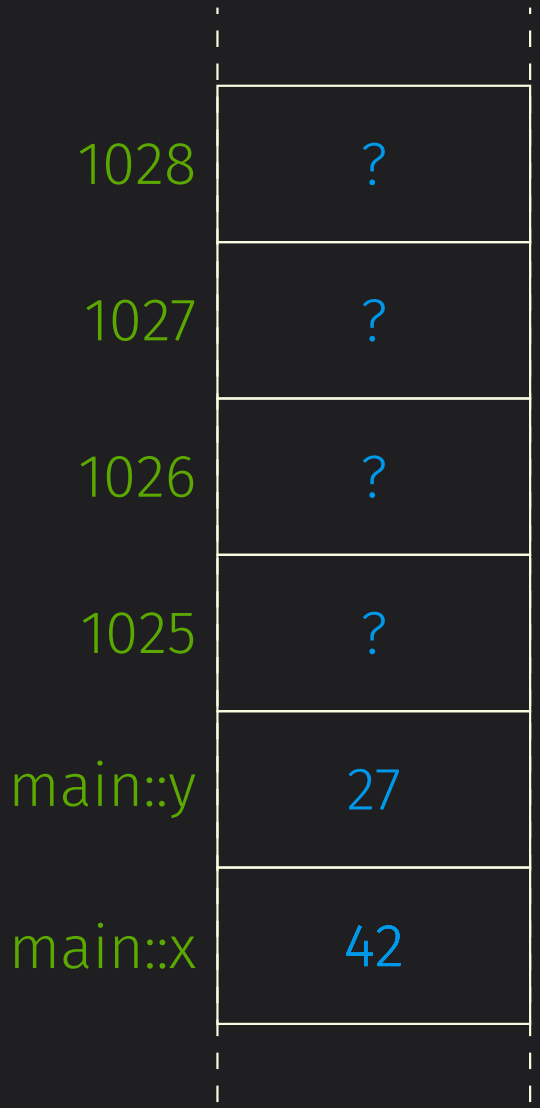
- We want to write a function **swap** that exchanges the values of two variables:

```
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);
```

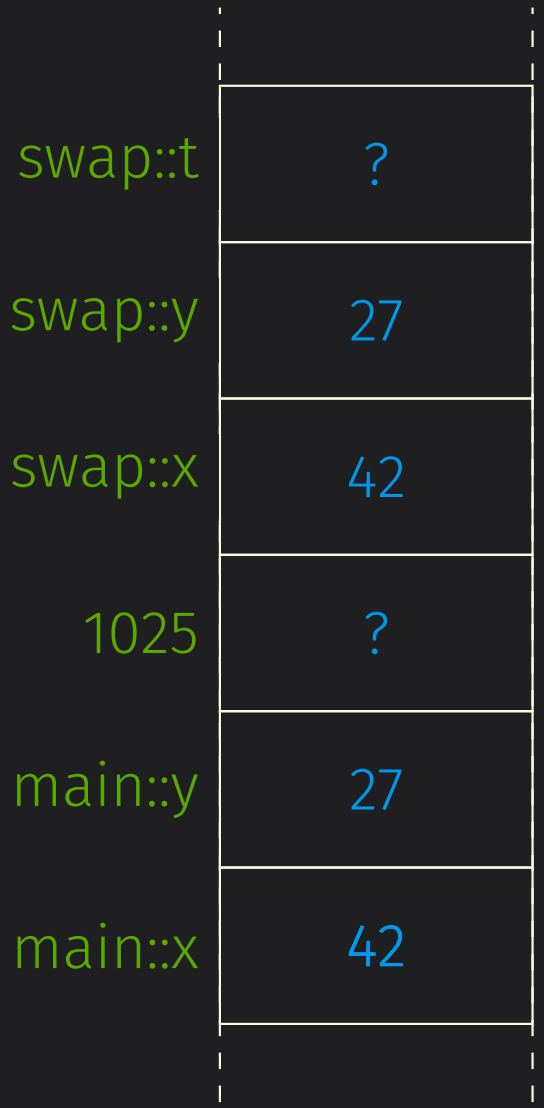
Example: swap, A First Try



- 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  }
```

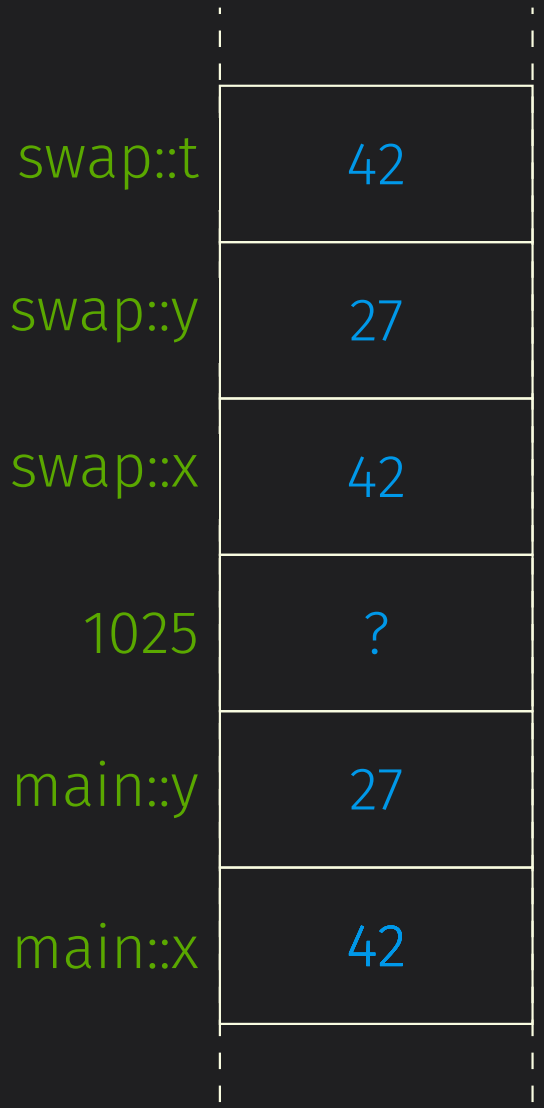
Example: swap, A First Try



- 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  }
```

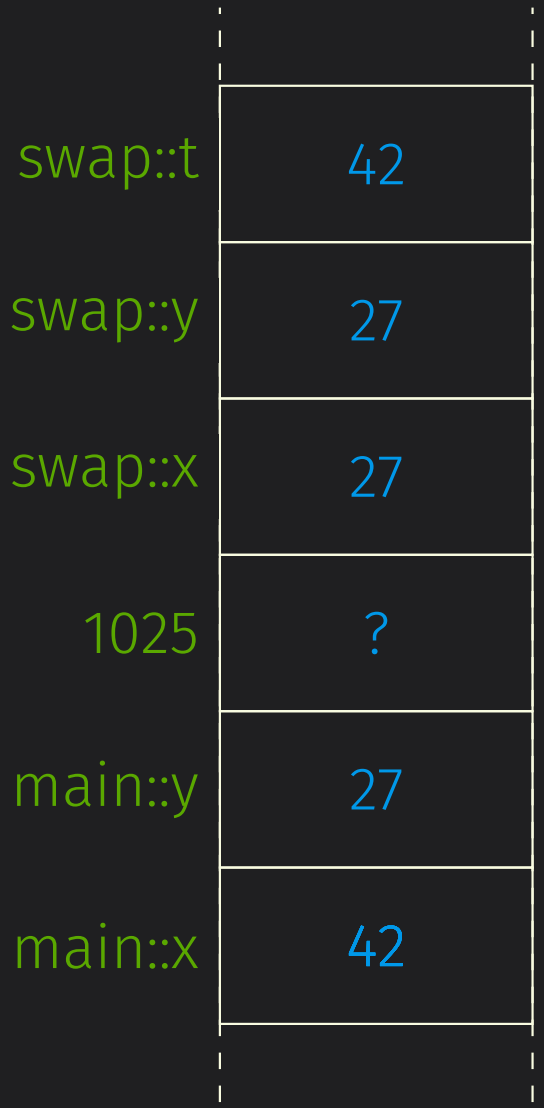
Example: swap, A First Try



- 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  }
```

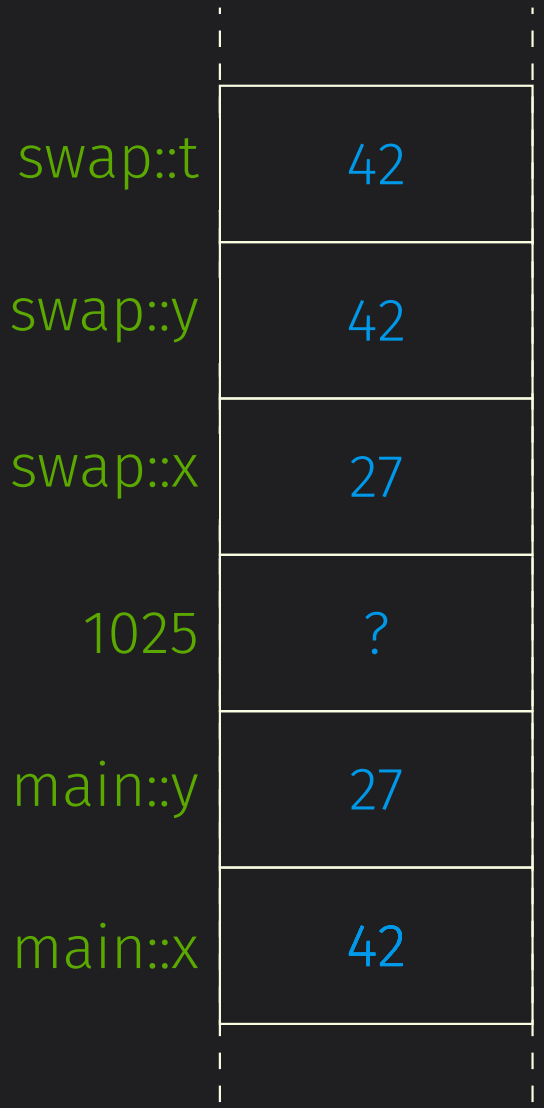
Example: swap, A First Try



- 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

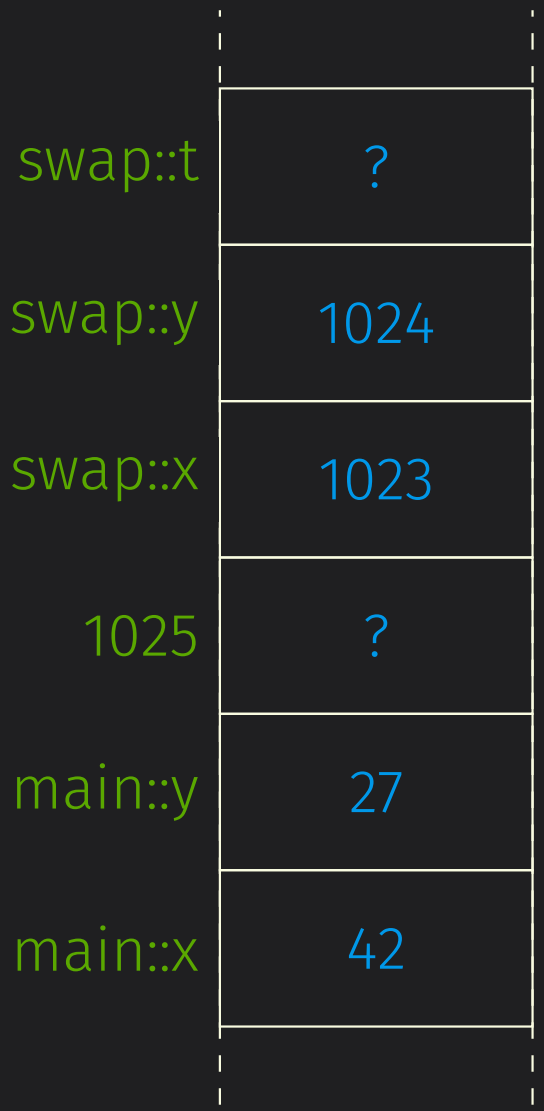
- The following with pointer works since `x` and `y` inside the function are pointers to the `x` and `y` in `main`:

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

- 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



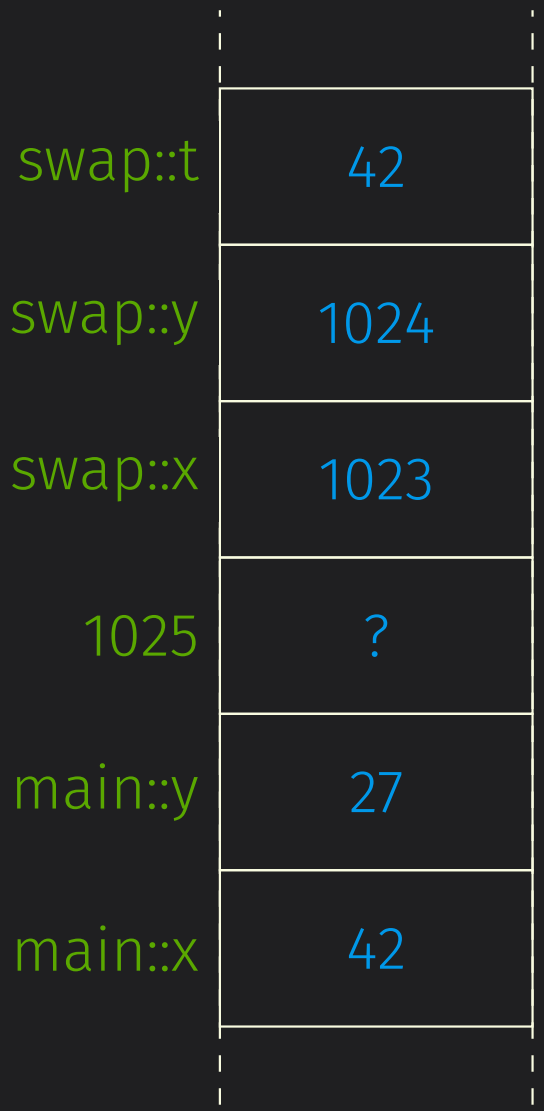
- The following with pointer works since `x` and `y` inside the function are pointers to the `x` and `y` in `main`:

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

- 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



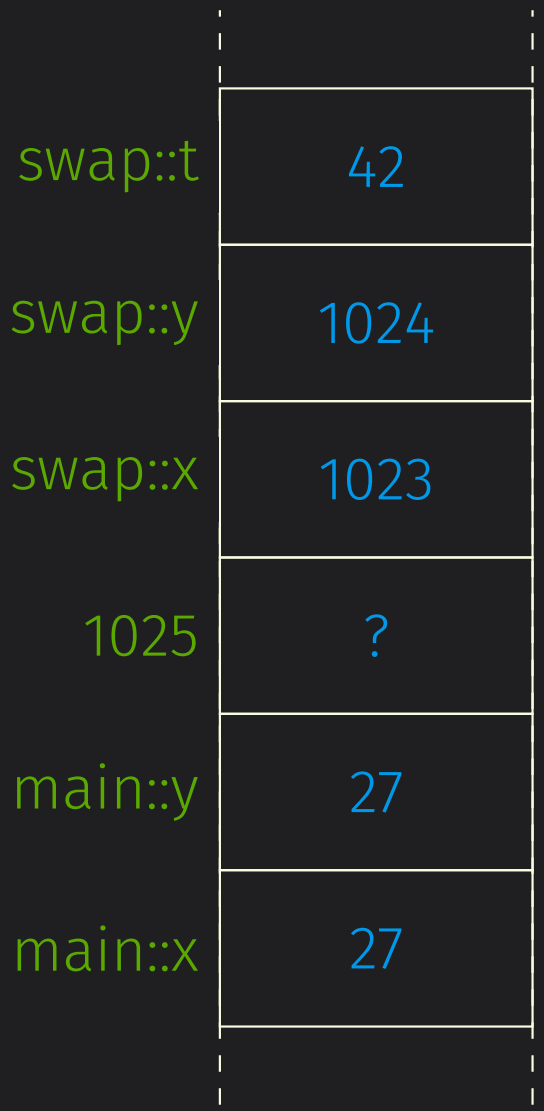
- The following with pointer works since `x` and `y` inside the function are pointers to the `x` and `y` in `main`:

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

- 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



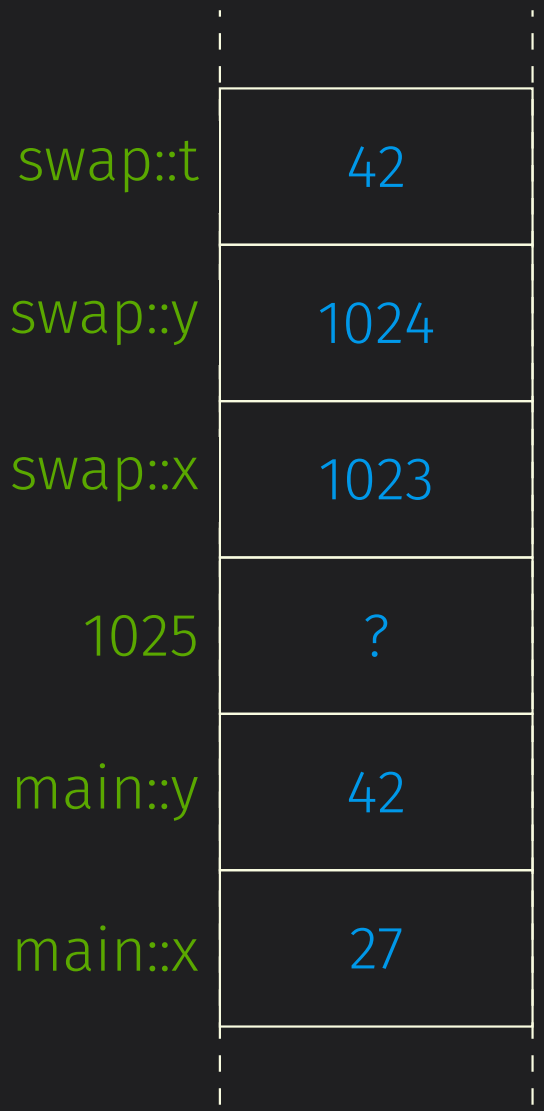
- The following with pointer works since `x` and `y` inside the function are pointers to the `x` and `y` in `main`:

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

- 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`:

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

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

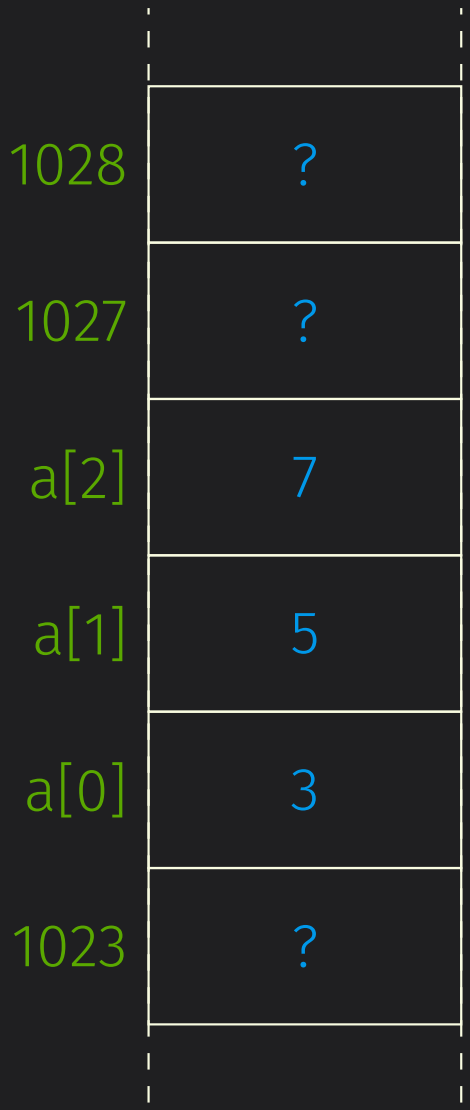
```
1  int main(int argc, char *argv[]) {
2    int x = 42;
3    int y = 27;
4    swap(&x, &y);
5  }
```

Example: swap with References

```
1  void swap(int &x, int &y) {
2    int t = x;
3    x = y;
4    y = t;
5  }
6
7  int main(int argc, char *argv[]) {
8    int x = 42;
9    int y = 27;
10   swap(x, y);
11 }
```

Arrays and Pointers

Fixed Size Arrays

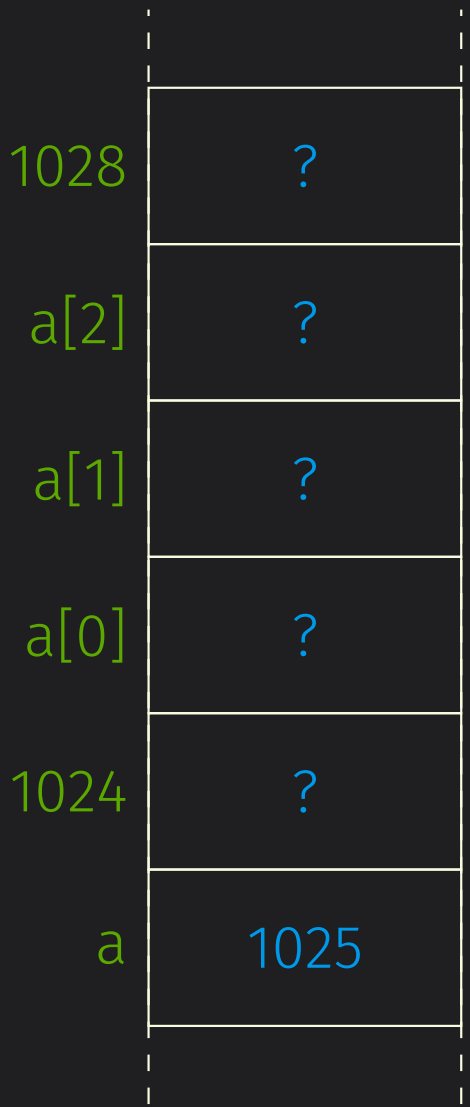


- When you need to store several items of the same type, you can declare an array variable:

```
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.

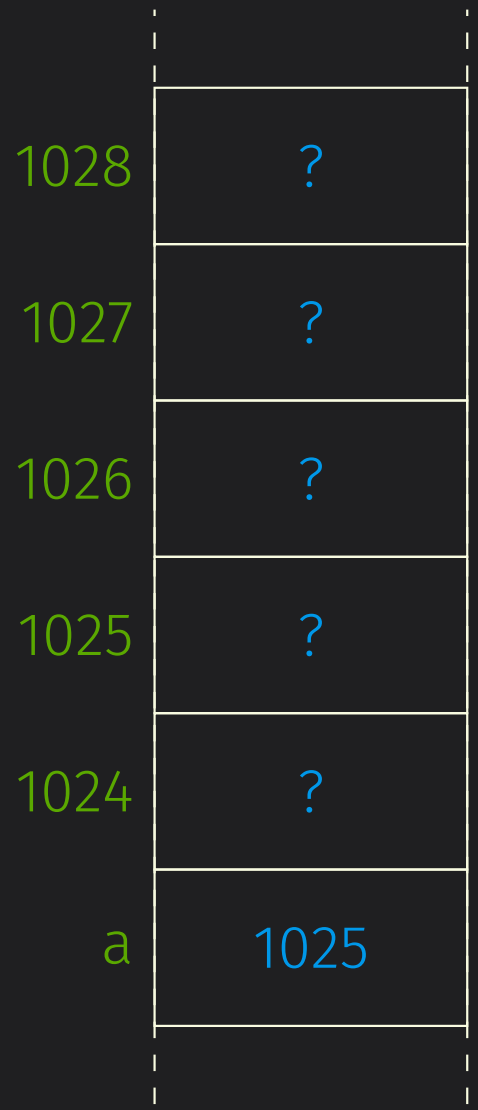
Dynamically Allocated Arrays



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

```
1   int *a = new int[3];
```

Dynamically Allocated Arrays



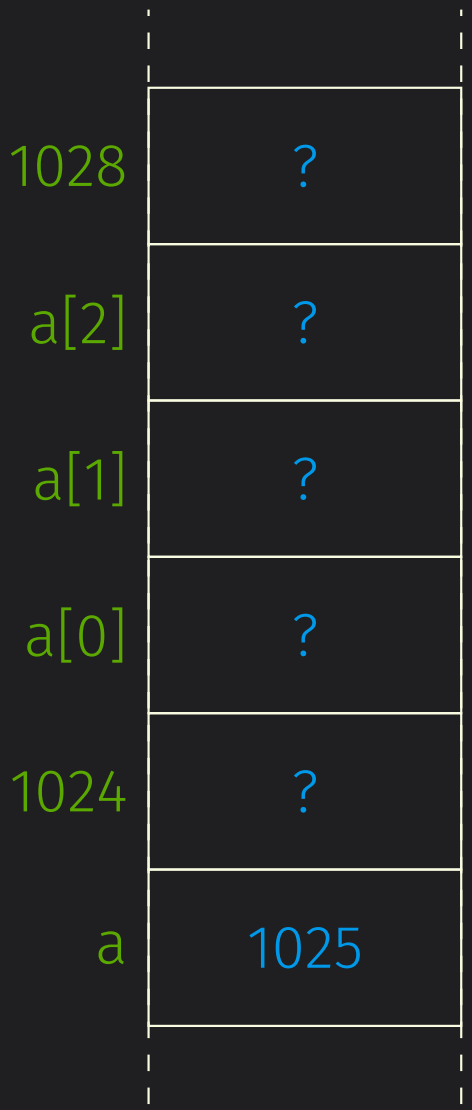
- When the array size is a variable quantity, you need to allocate the necessary memory yourself with the **new** operator:

```
1   int *a = new int[3];
```

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

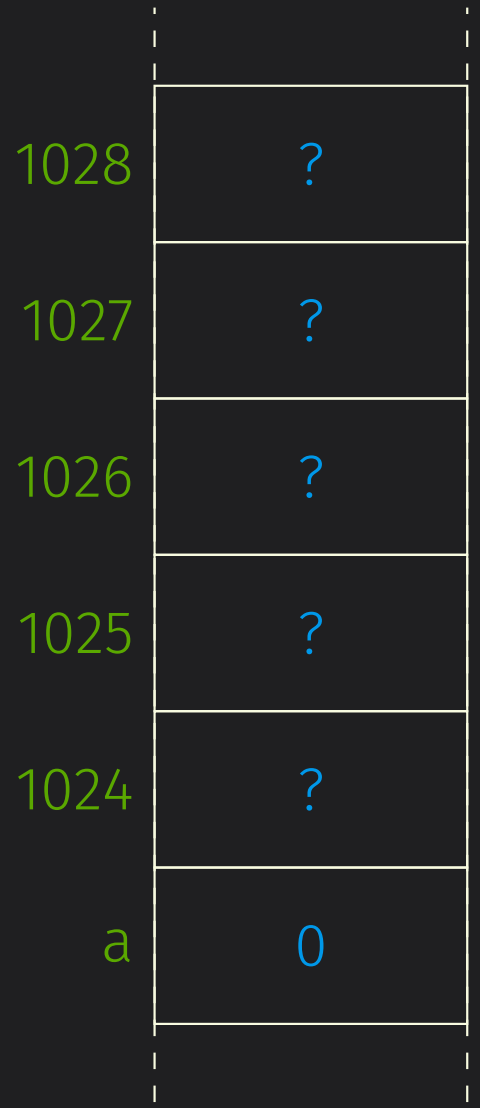
```
1   delete [] a;
```

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 zero, which is called a null pointer.

```
1  int *a = new int[3];
2  delete [] a;
3  a = 0;
```

`std::vector` (C++ only)

Standard Template Library

- C++ comes with a lot of data structures and algorithms inside the Standard Template Library (STL).

Standard Template Library

- C++ comes with a lot of data structures and algorithms inside the Standard Template Library (STL).
- Since working with C like arrays is difficult and manual memory management is error-prone, we will instead use the `std::vector` data structure from STL.

Standard Template Library

- C++ comes with a lot of data structures and algorithms inside the Standard Template Library (STL).
- Since working with C like arrays is difficult and manual memory management is error-prone, we will instead use the `std::vector` data structure from STL.
- Vectors are resizable arrays of a single data type. To use them you need to include the `<vector>` header file.

Standard Template Library

- C++ comes with a lot of data structures and algorithms inside the Standard Template Library (STL).
- Since working with C like arrays is difficult and manual memory management is error-prone, we will instead use the **`std::vector`** data structure from STL.
- Vectors are resizable arrays of a single data type. To use them you need to include the **`<vector>`** header file.
- The elements of the vector can be of any C++ data type and you need to fix it during variable declaration:

```
1  // An empty vector of integers.
2  vector<int> vi;
3  // A vector of floats initially containing 10 items.
4  vector<float> vf(10);
```

Vector Example

```
 9 vector<int> v(2);
10 cout << "Initial number of elements is " << v.size() << endl;
11
12 for (int i = 0; i < 5; ++i)
13         v.push_back(i);
14 cout << "Size after insertions is " << v.size() << endl;
15
16 cout << "Elements are ";
17 for (size_t i = 0; i < v.size(); ++i)
18         cout << v[i] << " ";
19 cout << endl;
```

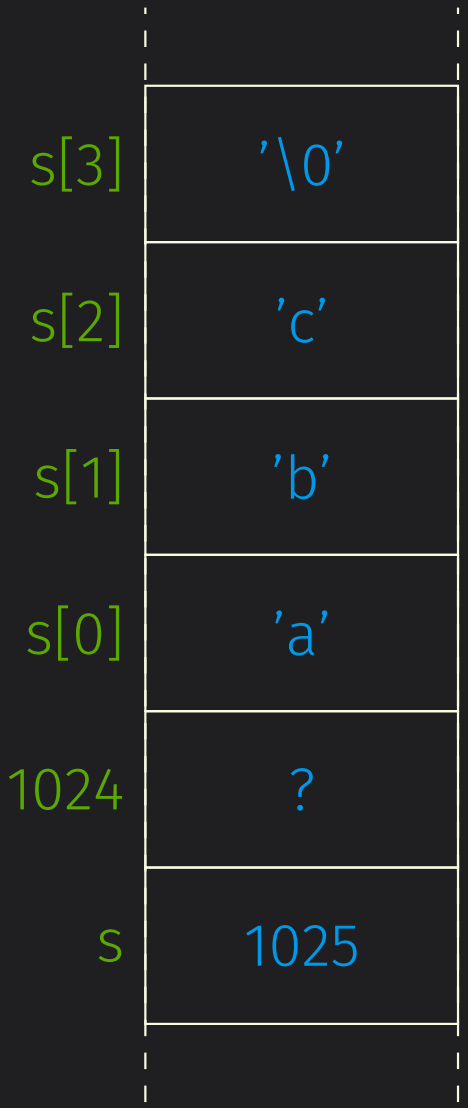
Vector Example

```
9  vector<int> v(2);
10 cout << "Initial number of elements is " << v.size() << endl;
11
12 for (int i = 0; i < 5; ++i)
13         v.push_back(i);
14 cout << "Size after insertions is " << v.size() << endl;
15
16 cout << "Elements are ";
17 for (size_t i = 0; i < v.size(); ++i)
18         cout << v[i] << " ";
19 cout << endl;
```

```
$ ./vectors
Initial number of elements is 2
Size after insertions is 7
Elements are 0 0 0 1 2 3 4
```

Character Arrays (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 const char *s1 = "Hello ";
10 const char *s2 = "World!";
11
12 char *s = new char[strlen(s1)+strlen(s2)+1];
13 strcpy(s, s1);
14 strcat(s, s2);
15 cout << s << endl;
16 delete [] s;
```

```
$ ./cstrings
Hello World!
```

std::string (C++ Only)

Standard Template Library

- Since working with C strings is difficult and manual memory management is error-prone, we will instead use `std::string` defined in the header `<string>`.

Standard Template Library

- Since working with C strings is difficult and manual memory management is error-prone, we will instead use `std::string` defined in the header `<string>`.
- C++ string objects internally store a C string but they manage the memory themselves.

Standard Template Library

- Since working with C strings is difficult and manual memory management is error-prone, we will instead use `std::string` defined in the header `<string>`.
- C++ string objects internally store a C string but they manage the memory themselves.
- You can get the size of a C++ string object by calling its `size()` method:

```
1  string s = "abc";
2  // This prints 3
3  cout << s.size() << endl;
```

Standard Template Library

- Since working with C strings is difficult and manual memory management is error-prone, we will instead use **`std::string`** defined in the header **`<string>`**.
- C++ string objects internally store a C string but they manage the memory themselves.
- You can get the size of a C++ string object by calling its `size()` method:

```
1  string s = "abc";
2  // This prints 3
3  cout << s.size() << endl;
```

- You can concatanate C++ strings with the `+` operator:

```
1  string s = "Hello ";
2  s += "World!";
```