GNG1106
Fundamentals of Engineering Computation

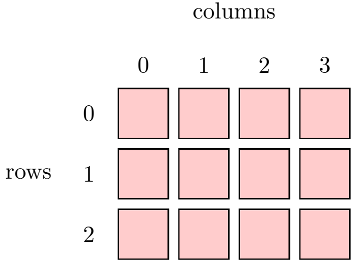
Instructor: **Hitham Jleed**

University of Ottawa

Fall 2023 ~

Outline

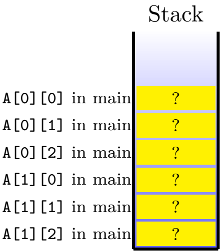
- A multi-dimensional array is used to store a matrix or higher dimensional data of same type.
- We will primarily focus on 2D array, which stores data that are logically organized in a matrix (i.e., row-column) form.



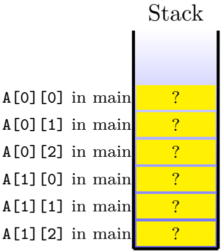
Declaration of 2D Array

```
type arrayName[numOfRows][numOfColumns];
```

- "`int A[2][3];`" declares a 2D array with 2 rows and 3 columns, where each array element is an int-typed variable.
- A 2-D array occupies contiguous block of bytes in stack, just like arrays.



- Each element in a 2D array is identified by the array name followed by its row and column indices (each inserted in a "[ ]").
- First index is the row index.
- Note: the indices start from 0, not 1!
- The declaration "`int A[2][3];`" in `main` gives rise to memory allocation in stack as in the figure.
- Note: The arrangement of the variables in stack is row-by-row!



- Like with arrays, the name of a 2D array is the address of the first variable in stack.
  - The value of 2D array `A` is the address of variable `A[0][0]`.
- Each variable in a 2D array is accessed in precisely the same way as a regular variable. For example:
  - `A[0][1]=10;`
  - `A[0][0]++;`
- Like with arrays, the indices of a 2D array can be an expression that evaluates to an integer in the valid range.

Declaration and Initialization of 2D Array

- A 2D array can be viewed as an “array of array”.
- For example, a 2D array with 3 rows and 4 columns can be regarded as a regular array of size 3, in which each element is an array of size 4.
- With this understanding, initializing a 2D array during declaration is essentially simultaneously initializing the array in a nested manner.

	0	1	2	3
0	1	3	5	7
1	11	13	15	17

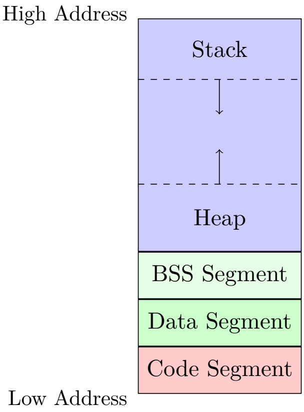
Two ways:

```
int A[2][4]={{1, 3, 5, 7}, {11, 13, 15, 17}};
```

or

```
int A[][4]={{1, 3, 5, 7}, {11, 13, 15, 17}};
```

Outline



- In the execution of a program, the data it uses and the program itself are both stored in the program memory.
- The program memory is logically organized as a large array of bytes.
- Each byte (whether it lives in code segment, data segment, BSS segment, stack or heap) has a unique address that specifies its location in the memory.
- Address is a positive integer.
  - In "64-bit computing": address is represented by 8 bytes.
- Contiguous bytes have contiguous addresses.

- When a variable (whether having a simple type or a structure type) or array is declared, a block of memory is allocated to store the variable or array.
- The number of bytes required to store a variable/array depends on the type of the variable (i.e. int, float, ...), the length of the array, and the computer system on which the program is compiled.
- The operator (not a function) `sizeof` can be used to determine the number of bytes required to store a particular variable or array.
    - The operator `sizeof` requires one argument which can be a type, a variable name, or an array name.
    - Correct syntax:
      "`sizeof variableName`", "`sizeof(variableName)`",
      "`sizeof arrayName`", "`sizeof(arrayName)`",
      or "`sizeof(type)`"
    - Wrong syntax: "`sizeof type`"

- Variables declared together in the same function are not necessarily stored in contiguous memory locations.
- The elements of an array are always stored in a contiguous block of bytes.
- The memory for all local variables inside a function is allocated (in the stack) only when the function is called, and the allocation occurs when the declaration statements are executed.
- The memory allocated for local variables are released when the function (containing the local variables) returns.

- Among all high-level programming languages, C provides the most direct access to the program memory.
- With C, the programmer can make the program to access any byte in the program memory as he desires.
- This is the most powerful feature of C comparing with other programming languages.
- It also presents risks for the programmer, since the programmer can make the program to perform illegal memory access (which the compiler won't detect)!
- Illegal memory accesses can be difficult to debug, since it may or may not cause the program to crash!

- The address of a variable is the address of the first byte (i.e., the byte having the lowest address value) in the memory block storing the variable.
- The address of a variable can be obtained using the address operator which "&". Recall we have used it in `scanf`.
  - "&" is a unary operator, acting on its right-side argument.
  - The argument of "&" must be a variable.
  - The argument cannot be a symbolic constant or an expression.
  - "&" gives the address of the variable.

Highlight (Variable ~ Variable Address ~ Variable Value)

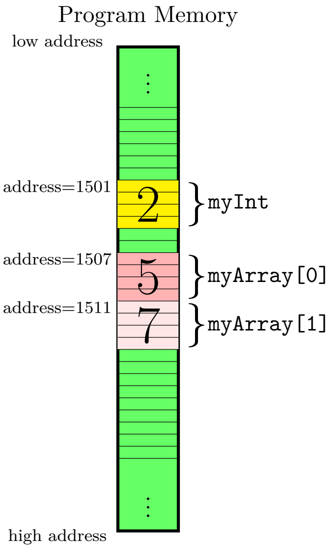
- variable: a block of bytes in the memory
- address of variable: the address of the first byte of the block
- value of variable: the data represented by the block

```
int myInt=2;
int myArray[2]={5, 7};
```

- Suppose that these declarations result in memory allocation shown in the picture (where each slot is a byte).

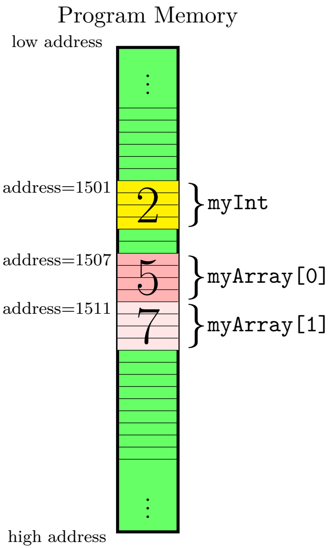
X	value of X
`myInt`	2
`myArray[0]`	5
`myArray[1]`	7
`&myInt`	1501
`&myArray[0]`	1507
`&myArray[1]`	1511
`myArray`	1507

- Recall the name of an array is the address of its first element!



Address Arithmetics

- Suppose that
  - `X` is the address of a byte in the memory,
  - starting from that byte (inclusive), a block of memory has been allocated for storing data of `Y` type, and
  - a `Y`-typed value takes m bytes to store.
- Then for any integer value k, (positive, negative or 0), `X+k` is an address value that is exactly $k \times m$ higher than `X`.
  - That is: one unit of address does not correspond to a byte (unless `Y` is the char type), but corresponds to a unit of `Y`-typed data.



- Consider this previous example where each int-typed value takes 4 bytes to store.
- In the table below "access to X" means "access to memory with address value X".

X	value of X	access to X
`&myInt+1`	1505	illegal
`myArray+1`	1511	legal
`myArray+2`	1515	illegal
`&myArray[0]+1`	1511	legal
`&myArray[1]+1`	1515	illegal
...	...	...

Dereferencing/Indirection Operator "*"

- Suppose that
  - `X` is the address of a byte in the memory,
  - starting from that byte (inclusive), a block of memory has been allocated for storing data of `Y` type, and
  - a `Y`-typed value takes m bytes to store.
- Then `*X` refers to the "variable" or block of m-byte memory starting from address `X`.

```
int myInt=2;
int b[2]={5, 7};
```

- `*(&myInt)` is the same as `myInt`.
- `*b` is the same as `b[0]`.
- `*(b+1)` is the same as `b[1]`.
- `*(b+2)` is the same as `b[2]`, but is an illegal memory access (and yet the compiler won't tell)!

Highlight (“&” and “*” as a Reciprocal Pair of Operators)

- `&X` gives the address for variable `X`.
- `*X` gives the “variable” having address `X`.

Highlight (Variable Revisit)

A variable is nothing more than a block of bytes in the memory, which has been associated with a prescribed method specifying how it should be interpreted as a number or as numbers.

- Suppose that we have the following declarations.

```
int myInt=2;
int b[2]={5, 7};
```

- Are the following assignment legal?

  `&myInt=b;`

  or

  `b=&myInt;`

- No! This is disallowed by the syntax of C!
- The memories allocated for a variable and for an array are fixed and consequently they have the fixed addresses.
- The memory allocated for a variable or for an array can be released but can not be changed!
- That is, the address values `&myInt` and `b` are address constants.

Outline

Pointer: Address-Typed Variable

Declaration

```
someType *x;
```

- `someType` can be any type, including structures.
- `x` is the name of declared variable.
- You are encouraged to think of "`someType *`" as the type of variable `x`.
- This line of declaration says the following:
  - `x` is declared to be a variable that takes an address as its value, and
  - the memory block with starting from address `x`, when referred to using `x`, should be interpreted as storing data of type `someType`.
- We may say that `x` is a "pointer to `someType`". E.g., if we declare "`int *x;`", we say `x` is a "pointer to int".

- The declaration of variables having a given type and the declaration of pointers to the same type can be combined in one statement.
- For example:

```
int myInt;
int *myPointer2int;
```

can be written as

```
int myInt, *myPointer2int;
```

- Like regular variables, when a pointer is declared, a block of memory is allocated for storing the value of the pointer.
  - Note: the value of a pointer is an address!
- If you declare a pointer `x`, you can use `sizeof(x)` to obtain the number of bytes that is required to store of the value of `x`.
  - On a 64-bit computer, it takes 8 bytes to store the value of a pointer.

Assignment of Pointers

- A pointer to a given type, say to type A, can be assigned an address of a memory block that has been designated for storing data of the same type, namely, type A. We then say that the pointer points to the memory block (or the variable).

Example

If we have declared "`int x, a[10], *ptr;`", the following are legal:

```
ptr=&x;
ptr=a;
ptr=a+3;
ptr=&a[5];
```

Example

If we have declared "`int x, a[10];`" and "`float *ptr;`", the following are illegal:

```
ptr=&x;
ptr=a;
ptr=a+3;
ptr=&a[5];
```

Declaring and Initializing a Pointer

- A pointer can be declared and assigned a value at the same time. For example, the following two ways are identical.

```
int x;
int *ptr=&x;
```

```
int x, *ptr=&x;
```

The NULL Pointer

- When a pointer is not assigned, it may have random values or have value `NULL`. This may depend on the computer system and the compiler.
- `NULL` is a symbolic constant defined in `stdio.h`. Its numerical value is 0.
- A pointer having value `NULL` means that it points to nowhere, i.e., that it does not point to any valid memory address.
- When we want to assure a pointer point to nowhere, we should explicitly assign value `NULL` to it.
- It is also encouraged to initialize pointers to NULL when they are declared.

Dereferencing a Pointer

- A pointer can be dereferenced using the operator "*".
- Dereferencing a pointer works in exactly the same way as dereferencing an address.
- A dereferenced pointer is essentially a variable and can be used in exactly the same way as a variable.

```
int x=10, *ptr;
ptr=&x;
printf("%d\n", *ptr); // prints 10
*ptr = *ptr + 10;
printf("%d\n", *ptr); // prints 20
printf("%d\n", x); // prints 20
```

The Confusing Symbol *

- We have now learned three distinct uses of the symbol "*".
  - As the multiplication operator
  - In declaring a pointer (i.e., forming a "pointer type")
  - As the dereferencing operator
- When you see the symbol * in a code, ask yourself what it is meant there.

Dereferencing a Pointer That Points to an Array

- We have seen that a pointer can point to an array, or more precisely, points to the first element of an array.
- Example: in "`int a[3]={2, 4, 6}, *ptr=a;` "
- In this case, we can use pointer arithmetics (namely, address arithmetics) in combination with pointer dereferencing to access any element of the array.
- Following the above example line of code, the dereferenced addresses `*ptr`, `*(ptr+1)`, and `*(ptr+2)` will be precisely variables `a[0]`, `a[1]`, and `a[2]` respectively.
- It is also allowed to use the array notation to write the dereferenced addresses. That is, `*ptr`, `*(ptr+1)`, and `*(ptr+2)` can also be written respectively as `ptr[0]`, `ptr[1]`, and `ptr[2]`.

Allowed Operations on Pointers

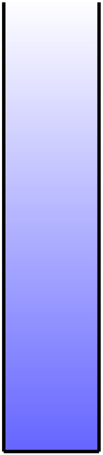
- Assignment
- Dereferencing
- Address arithmetics (addition/subtraction)
- Increment: `ptr++;` (the same as `ptr=ptr+1;`)
- Decrement: `ptr--;` (the same as `ptr=ptr-1;`)
- Comparison (with NULL or with other pointers)

Tracing A Code in Programming Model

Code Segment

```
#include <stdio.h>
int main()
{
  int x=3, y[3]={4, 5, 6};
  int *ptr;
  ptr=&x;
  *ptr=20;
  ptr=y;
  *ptr=-1;
  ptr[1]=-2;
  *(ptr+2)=-3;
  ptr++;
  ptr[0]=ptr[0]-10;
  *(ptr+1)=ptr[0]*ptr[1];
  return 0;
}
```

Stack

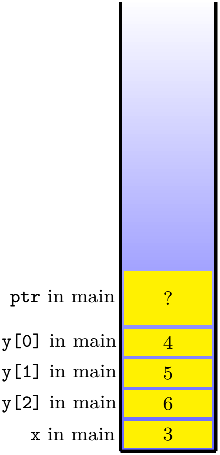


Tracing A Code in Programming Model: Step 1

Code Segment

```c
#include <stdio.h>
int main()
{
   int x=3, y[3]={4, 5, 6};
   int *ptr;
   ptr=&x;
   *ptr=20;
   ptr=y;
   *ptr=-1;
   ptr[1]=-2;
   *(ptr+2)=-3;
   ptr++;
   ptr[0]=ptr[0]-10;
   *(ptr+1)=ptr[0]*ptr[1];
   return 0;
}
```

Stack

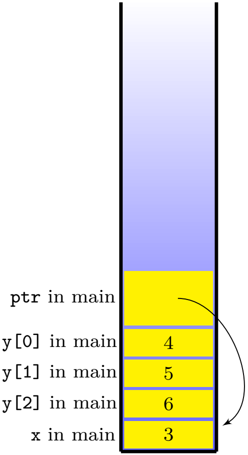


Tracing A Code in Programming Model: Step 2

Code Segment

```
#include <stdio.h>
int main()
{
  int x=3, y[3]={4, 5, 6};
  int *ptr;
  ptr=&x;
  *ptr=20;
  ptr=y;
  *ptr=-1;
  ptr[1]=-2;
  *(ptr+2)=-3;
  ptr++;
  ptr[0]=ptr[0]-10;
  *(ptr+1)=ptr[0]*ptr[1];
  return 0;
}
```

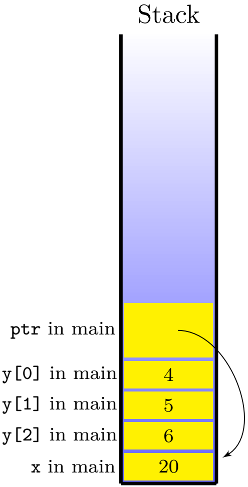
Stack



Tracing A Code in Programming Model: Step 3

Code Segment

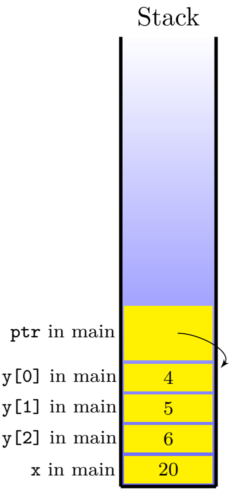
```
#include <stdio.h>
int main()
{
  int x=3, y[3]={4, 5, 6};
  int *ptr;
  ptr=&x;
  *ptr=20;
  ptr=y;
  *ptr=-1;
  ptr[1]=-2;
  *(ptr+2)=-3;
  ptr++;
  ptr[0]=ptr[0]-10;
  *(ptr+1)=ptr[0]*ptr[1];
  return 0;
}
```



Tracing A Code in Programming Model: Step 4

Code Segment

```
#include <stdio.h>
int main()
{
  int x=3, y[3]={4, 5, 6};
  int *ptr;
  ptr=&x;
  *ptr=20;
  ptr=y;
  *ptr=-1;
  ptr[1]=-2;
  *(ptr+2)=-3;
  ptr++;
  ptr[0]=ptr[0]-10;
  *(ptr+1)=ptr[0]*ptr[1];
  return 0;
}
```

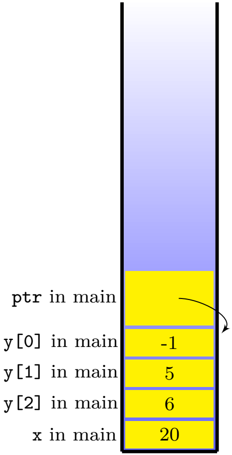


Tracing A Code in Programming Model: Step 5

Code Segment

```
#include <stdio.h>
int main()
{
  int x=3, y[3]={4, 5, 6};
  int *ptr;
  ptr=&x;
  *ptr=20;
  ptr=y;
  *ptr=-1;
  ptr[1]=-2;
  *(ptr+2)=-3;
  ptr++;
  ptr[0]=ptr[0]-10;
  *(ptr+1)=ptr[0]*ptr[1];
  return 0;
}
```

Stack

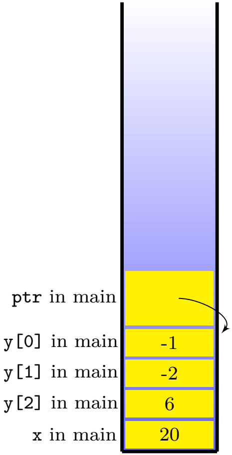


Tracing A Code in Programming Model: Step 6

Code Segment

```
#include <stdio.h>
int main()
{
  int x=3, y[3]={4, 5, 6};
  int *ptr;
  ptr=&x;
  *ptr=20;
  ptr=y;
  *ptr=-1;
  ptr[1]=-2;
  *(ptr+2)=-3;
  ptr++;
  ptr[0]=ptr[0]-10;
  *(ptr+1)=ptr[0]*ptr[1];
  return 0;
}
```

Stack

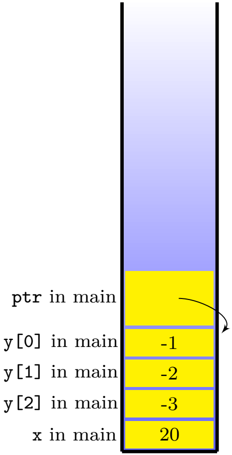


Tracing A Code in Programming Model: Step 7

Code Segment

```
#include <stdio.h>
int main()
{
  int x=3, y[3]={4, 5, 6};
  int *ptr;
  ptr=&x;
  *ptr=20;
  ptr=y;
  *ptr=-1;
  ptr[1]=-2;
  *(ptr+2)=-3;
  ptr++;
  ptr[0]=ptr[0]-10;
  *(ptr+1)=ptr[0]*ptr[1];
  return 0;
}
```

Stack

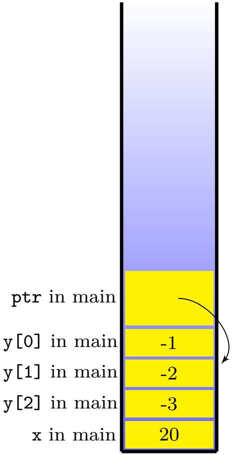


Tracing A Code in Programming Model: Step 8

Code Segment

```
#include <stdio.h>
int main()
{
  int x=3, y[3]={4, 5, 6};
  int *ptr;
  ptr=&x;
  *ptr=20;
  ptr=y;
  *ptr=-1;
  ptr[1]=-2;
  *(ptr+2)=-3;
  ptr++;
  ptr[0]=ptr[0]-10;
  *(ptr+1)=ptr[0]*ptr[1];
  return 0;
}
```

Stack

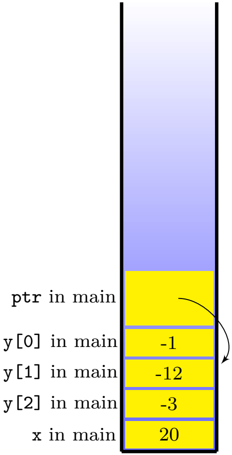


Tracing A Code in Programming Model: Step 9

Code Segment

```
#include <stdio.h>
int main()
{
  int x=3, y[3]={4, 5, 6};
  int *ptr;
  ptr=&x;
  *ptr=20;
  ptr=y;
  *ptr=-1;
  ptr[1]=-2;
  *(ptr+2)=-3;
  ptr++;
  ptr[0]=ptr[0]-10;
  *(ptr+1)=ptr[0]*ptr[1];
  return 0;
}
```

Stack

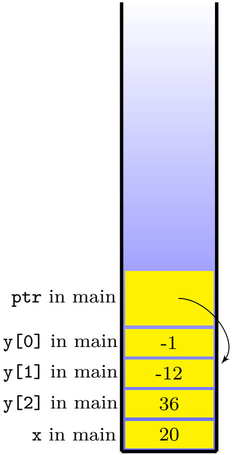


Tracing A Code in Programming Model: Step 10

Code Segment

```
#include <stdio.h>
int main()
{
   int x=3, y[3]={4, 5, 6};
   int *ptr;
   ptr=&x;
   *ptr=20;
   ptr=y;
   *ptr=-1;
   ptr[1]=-2;
   *(ptr+2)=-3;
   ptr++;
   ptr[0]=ptr[0]-10;
   *(ptr+1)=ptr[0]*ptr[1];
   return 0;
}
```

Stack



Tracing A Code in Programming Model: Step 11

Code Segment

```
#include <stdio.h>
int main()
{
  int x=3, y[3]={4, 5, 6};
  int *ptr;
  ptr=&x;
  *ptr=20;
  ptr=y;
  *ptr=-1;
  ptr[1]=-2;
  *(ptr+2)=-3;
  ptr++;
  ptr[0]=ptr[0]-10;
  *(ptr+1)=ptr[0]*ptr[1];
  return 0;
}
```

Stack