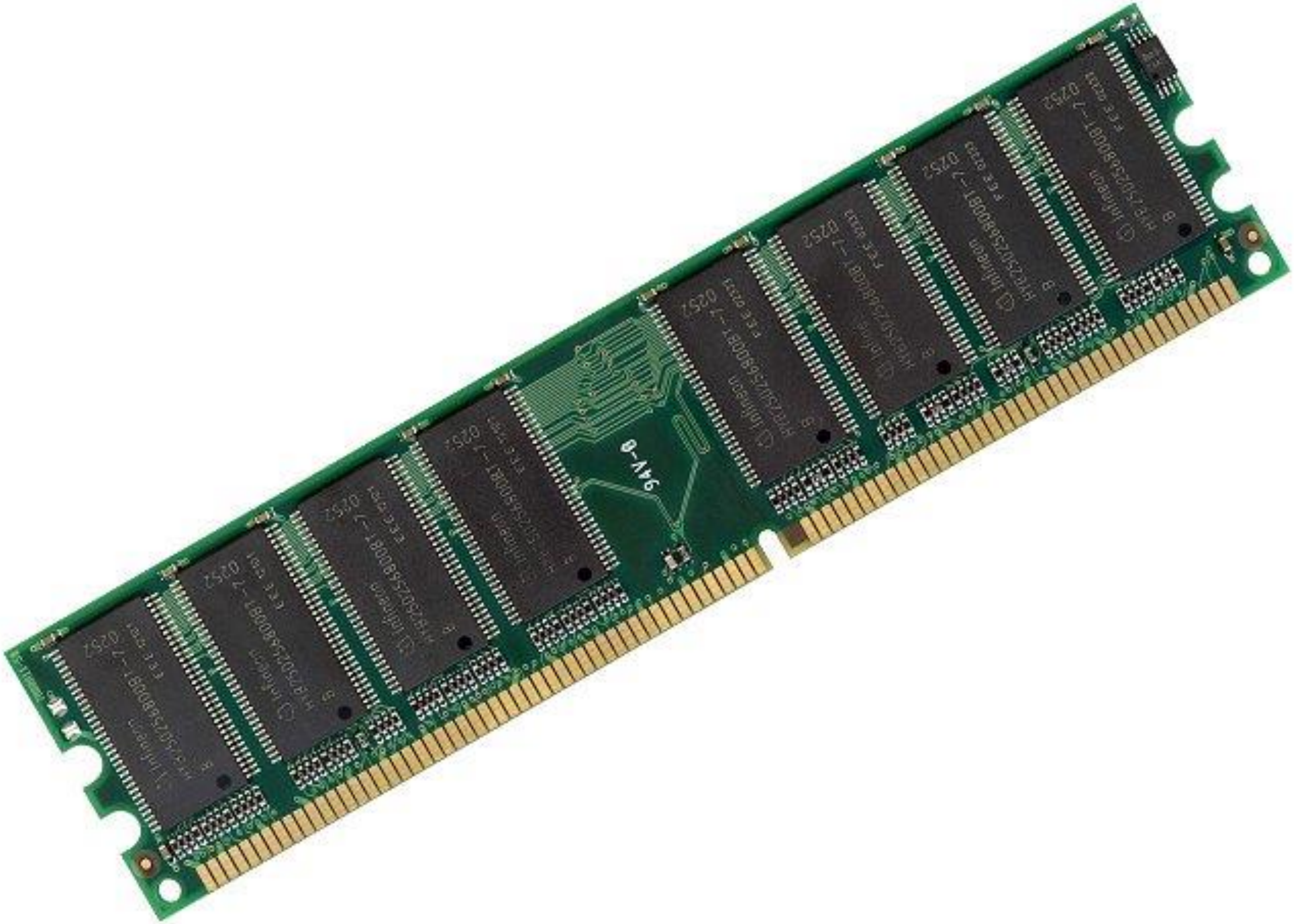
Chapter 07
POINTER

Dr. Le Thanh Sach

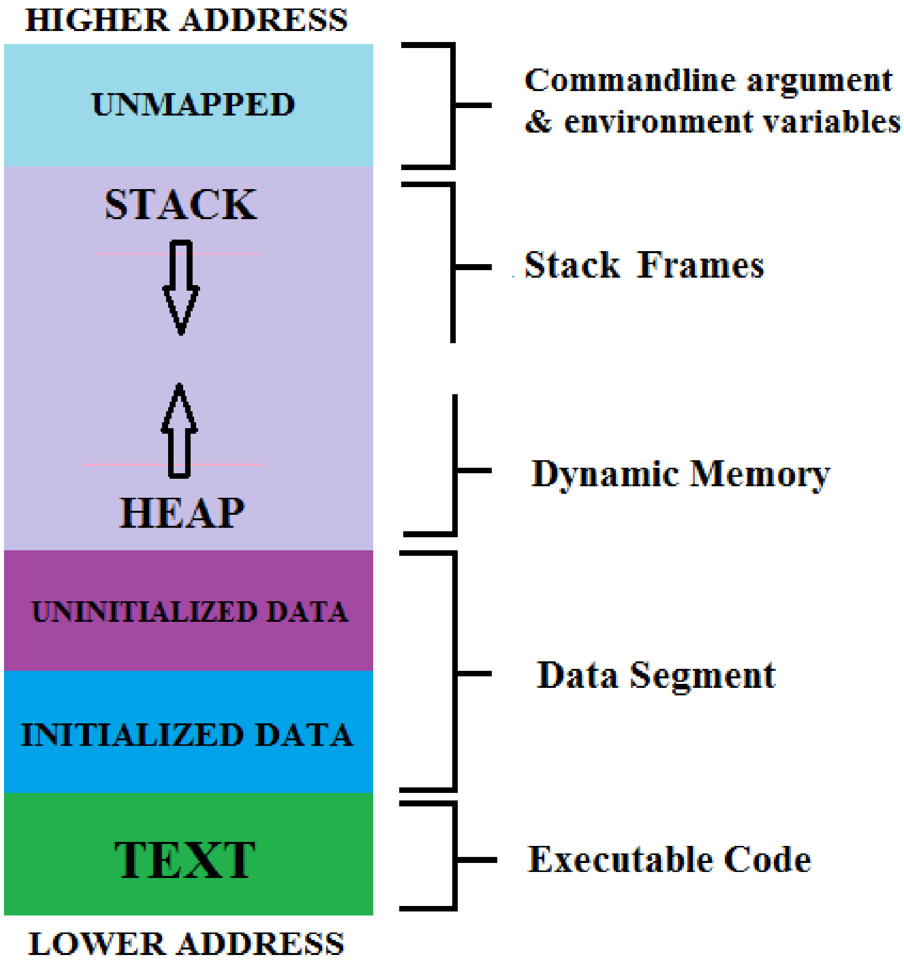


Content

- Memory layout
- Uses of pointer
- Model of pointer
- & Operator
- Declare a pointer
- * Operator
- Operations on pointer
- Pointer and array
- Dynamic memory allocation
- Pointers and structures, `->` operator
- Advanced topics with pointers
  - Order of evaluation `* và ++, --`
  - Pointer and const
  - Pointer to pointer
  - Void pointer

Memory layout

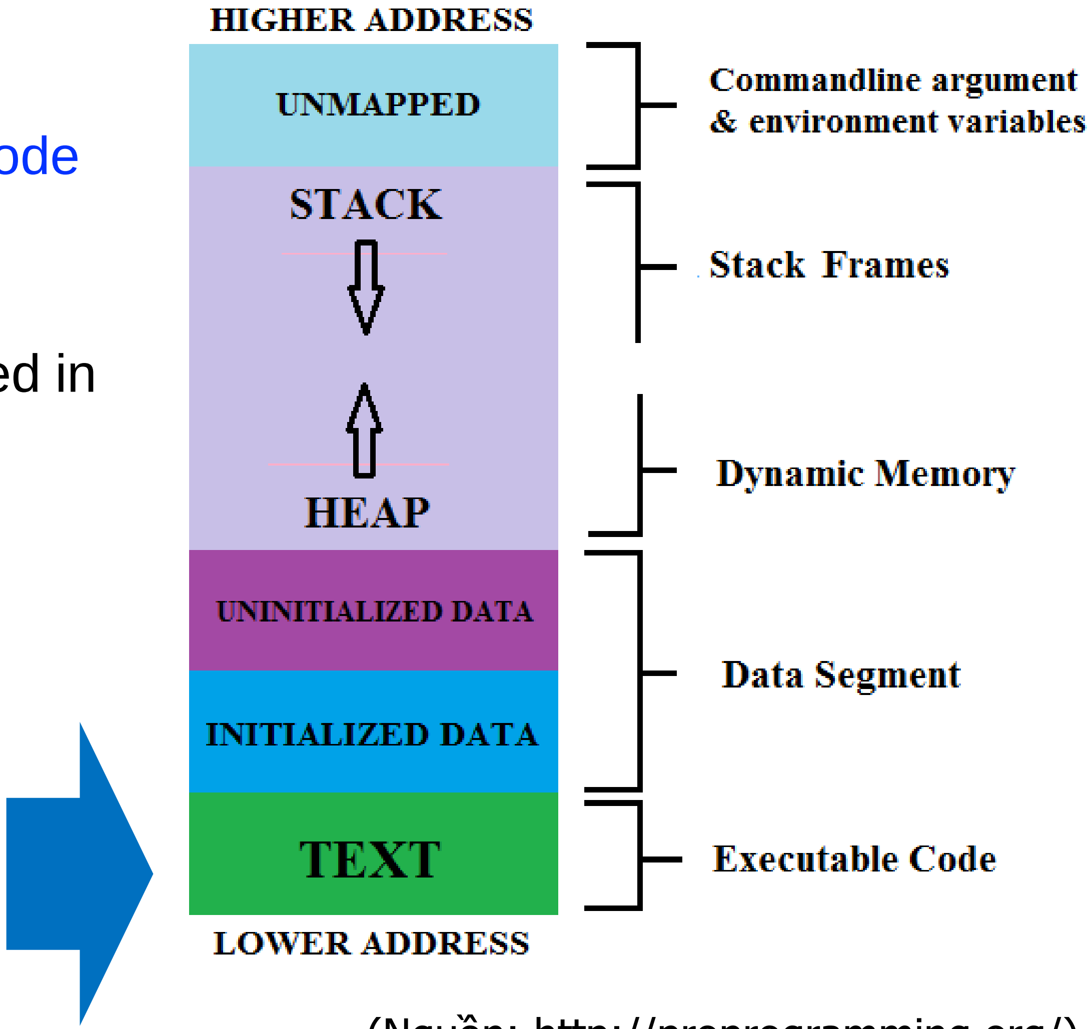
- When the program is put into the memory to execute, the system organizes the memory like the next diagram



(Source: http://proprogramming.org/)

Memory layout

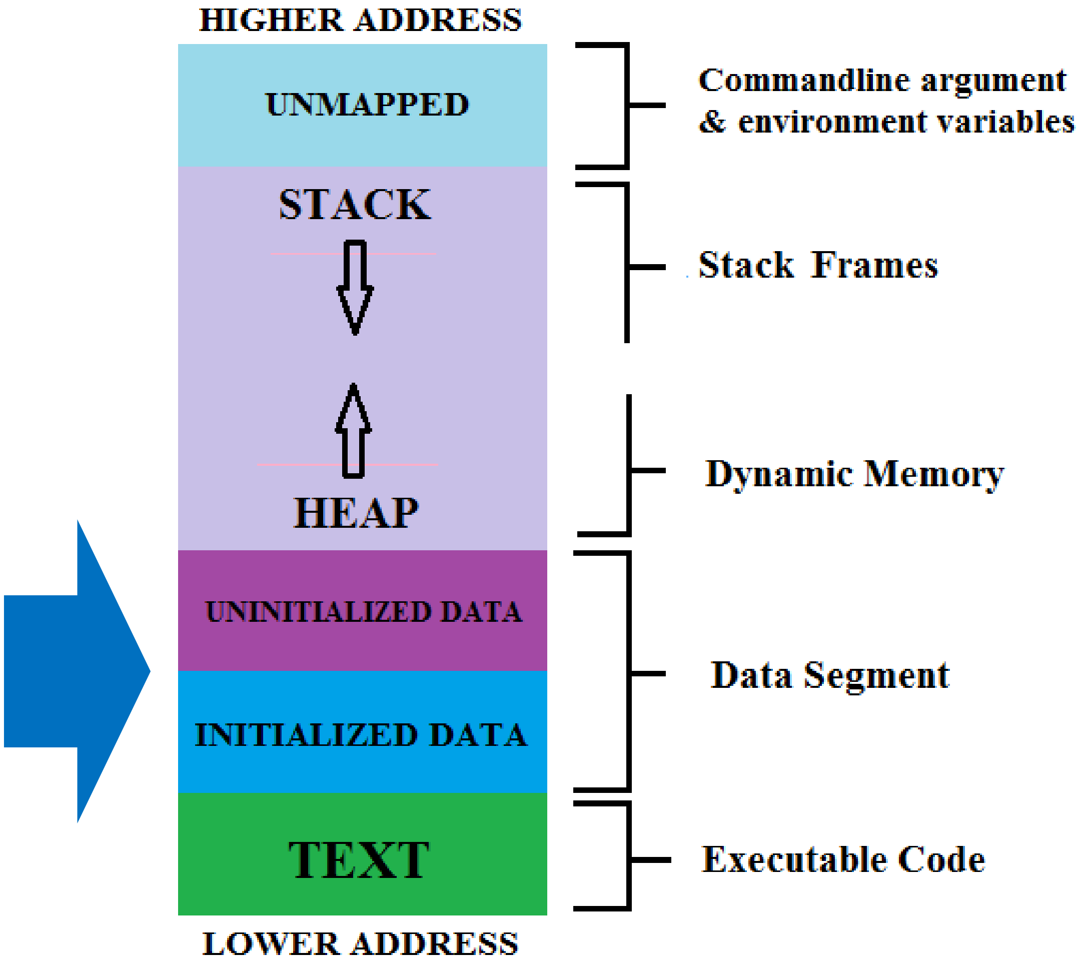
- “text” area
  - Contains executable code of the program
  - This area is read-only
  - This area can be shared in case the program is executed frequently



(Nguồn: http://proprogramming.org/)

Memory layout

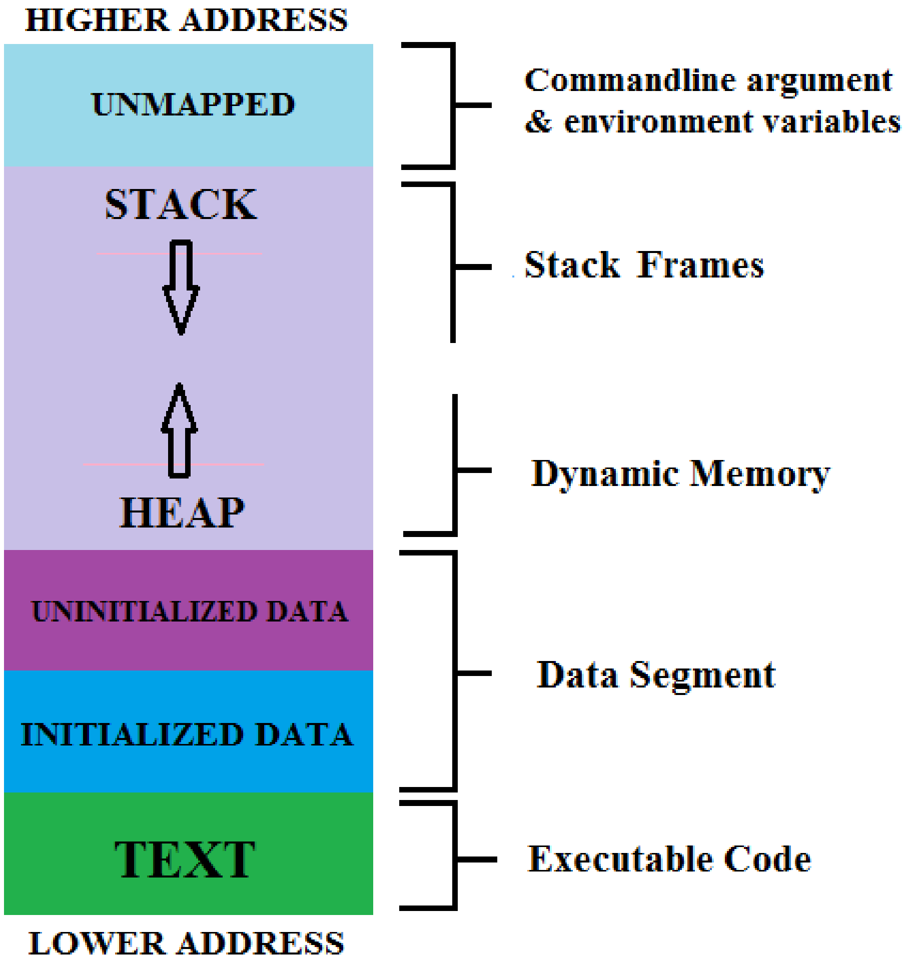
- “Data” Area
  - include:
    - Initialized data (by programmer)
    - Uninitialized data (by programmer)



(Source: http://proprogramming.org/)

Memory layout

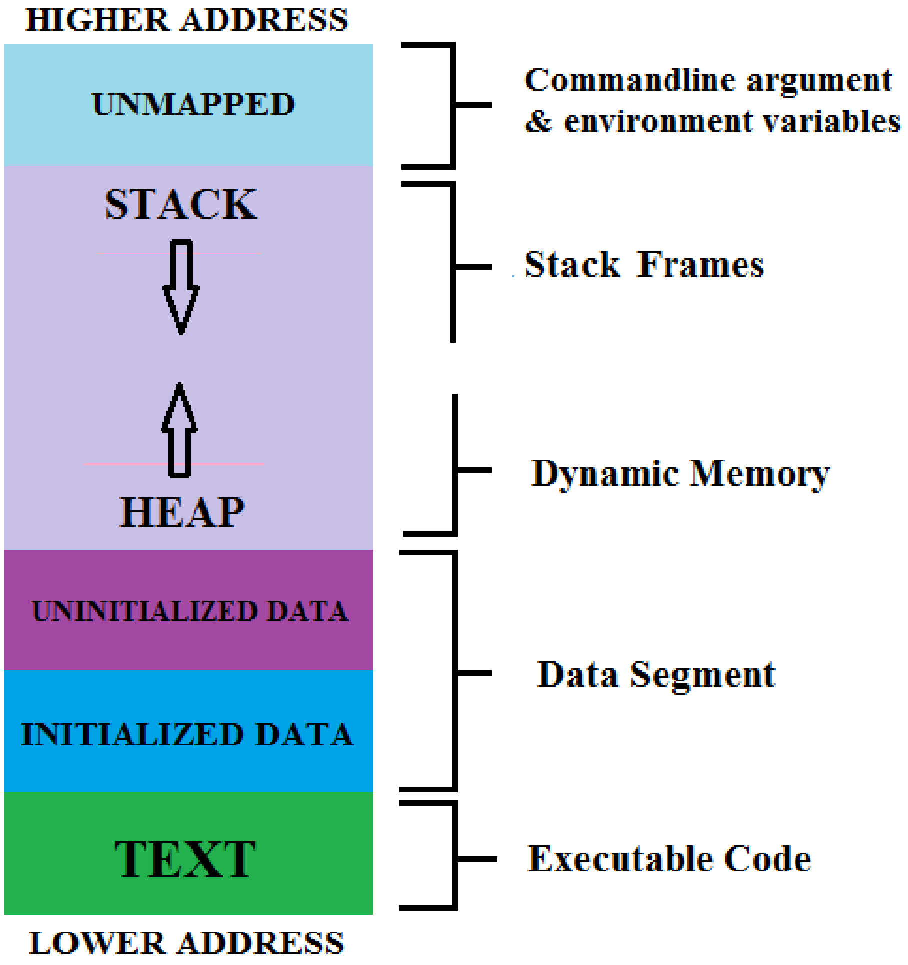
- "Data" area
  - Include:
    - Initialized data (by programmer)
      - Global variable
      - Static variable
    - This area consists of two sub-areas:
      - Read-only
        - Example: String constant
      - Read/Write
        - Non-constant static and global variables



(Source: http://proprogramming.org/)

Memory layout

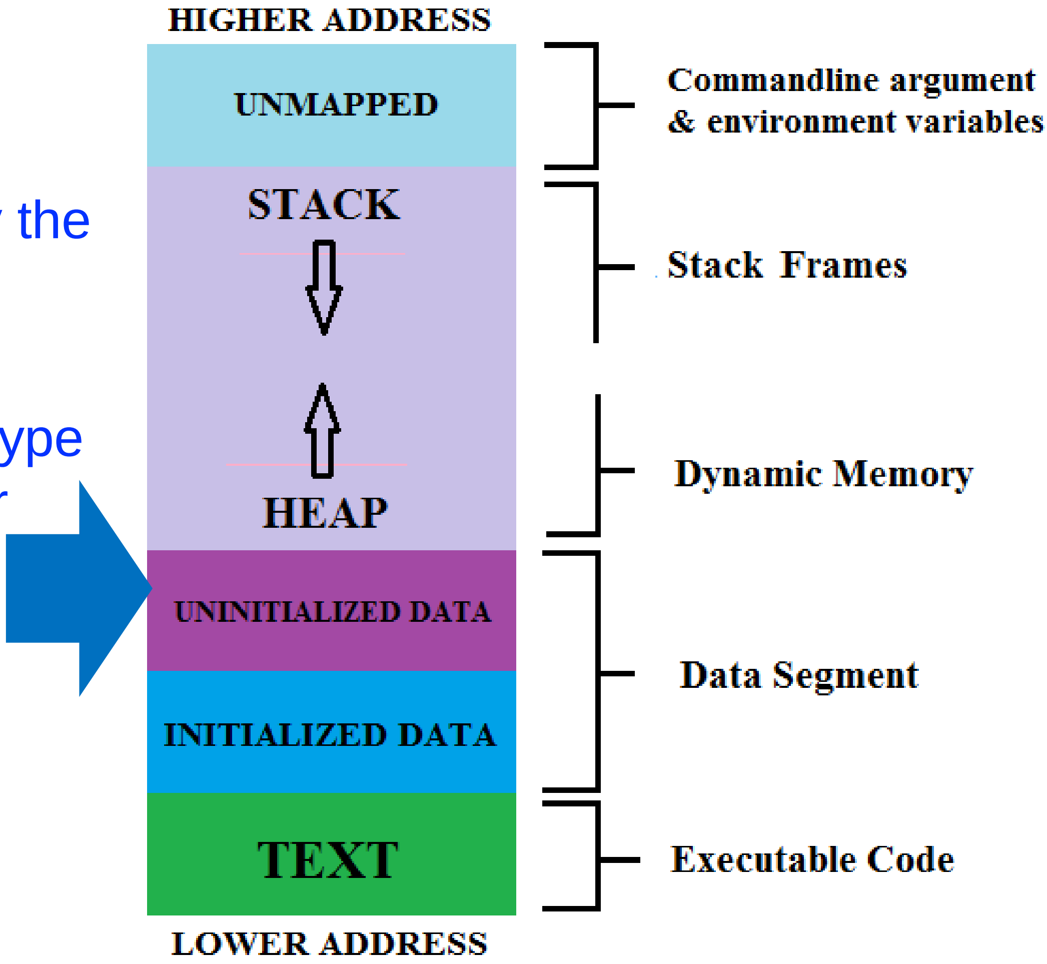
- "Data" area
  - Include:
    - Initialized data
    - Uninitialized data by programmer
      - Global variable
      - Static variable
      - The system assign 0 (number) to variables that were not explicitly initialized by programmer



(Nguồn: http://proprogramming.org/)

Memory layout

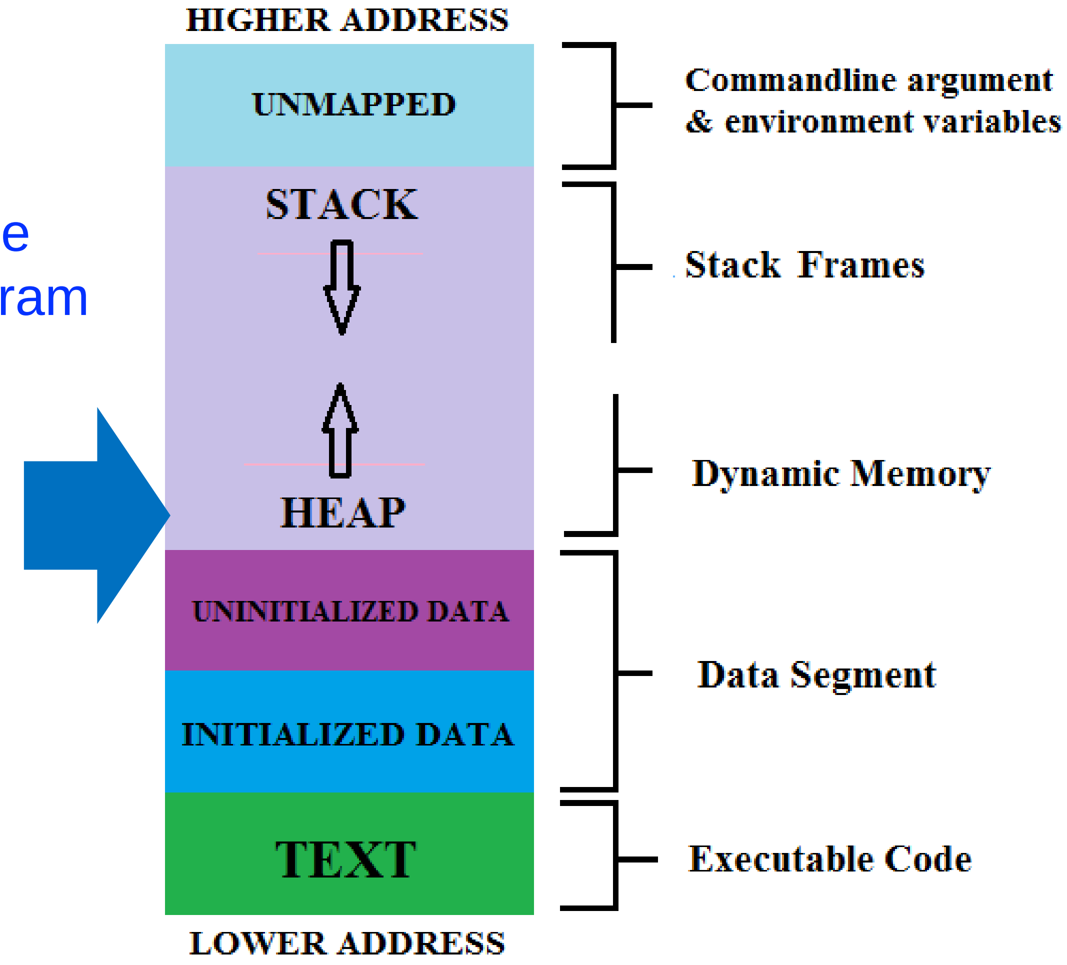
- "HEAP" area
  - Contains memory that is allocated dynamically by the programmer
  - Is related to the pointer type discussed in this chapter



(source: http://proprogramming.org/)

Memory layout

- “STACK” area
  - Include:
    - Variables which were declared in the program
    - Information of each function call



HIGHER ADDRESS

UNMAPPED

STACK

HEAP

UNINITIALIZED DATA

INITIALIZED DATA

TEXT

LOWER ADDRESS

Commandline argument & environment variables

Stack Frames

Dynamic Memory

Data Segment

Executable Code

(Nguồn: http://phortlegrasmeidg/)org/)

Uses of pointer

- Array in C++
  - Must know the number of elements at the time of writing the code
  - Therefore, it is necessary to declare a large number of memory cells. However, at some point, the program may use a lot less → wasteful
  - Question: Is it possible to use an array with the number of elements that can only be known when the program is running?
  - => We need pointer

Uses of pointer

- Array in C++
  - When adding and removing elements on the array
  - => Need to move many elements to the right and to the left → Time consuming
  - Question: Is there any way to organize the data that helps to manage the above elements quickly?
  - => Use linked list
  - => We need pointer

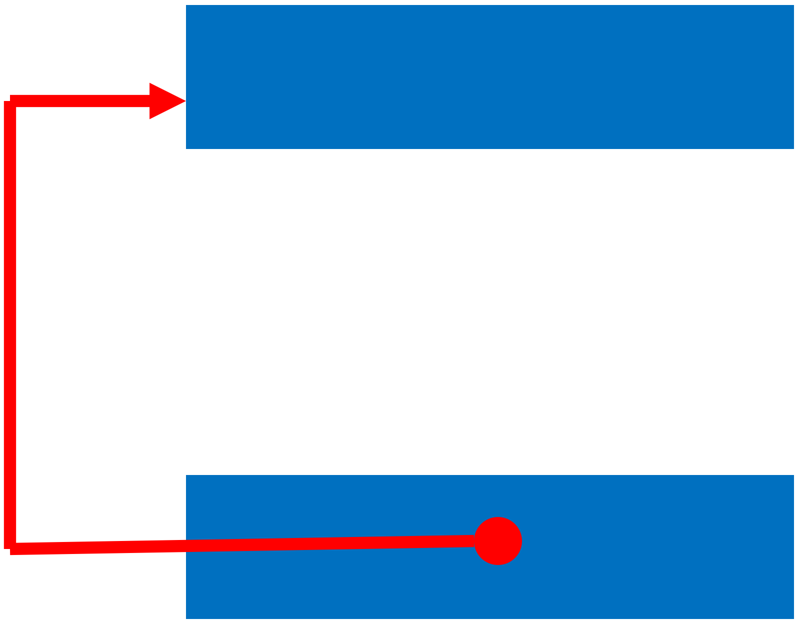
Model of pointer

0x1234 FFFF

Variable a: Is a variable of any kind
Started memory cell of a has the address:
(example) 0x1234 FFFF

Variable p: Is a pointer
Contains the address of the variable x,
It means that the value of p is 0x1234 FFFF

Model of pointer



Variable a: Is a variable of any kind
Started memory cell of a has the address:
(example) 0x1234 FFFF

Variable p: Is a pointer
Contains the address of the variable x,
It means that the value of p is 0x1234 FFFF

The pointer is illustrated by an arrow from variable p Point to the memory cell of variable x

& Operator

- The pointer stores the address of another memory cell (variable) ➔ How to get the address of a variable or memory cell to assign a pointer variable

& Operator

- The pointer stores the address of another memory cell (variable) ➔ How to get the address of a variable or memory cell to assign to a pointer variable
  - Method 1: Use the & operator to get the address of an existing variable
  - Method 2: Request dynamic memory allocation (later)

& Operator

- The operator & returns the address of a variable
- Example

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

int main(){
        int a = 100;
        cout << a << endl;

        cout << &a << endl;

        system("pause");
        return 0;
}
```

Print out the value of a

Print out the address of a

& Operator

- The operator & returns the address of a variable
- Example

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

typedef struct sPoint3D{float x, y, z;} Point3D;
int main(){
        Point3D p1 = {1.0f, 2.0f, 3.0f};
        cout << p1.x << endl;
        cout <<  &p1 << endl;
        cout <<  &p1.x << endl;
        cout <<  &p1.y << endl;
        cout <<  &p1.z << endl;
        system("pause");
        return 0;
}
```

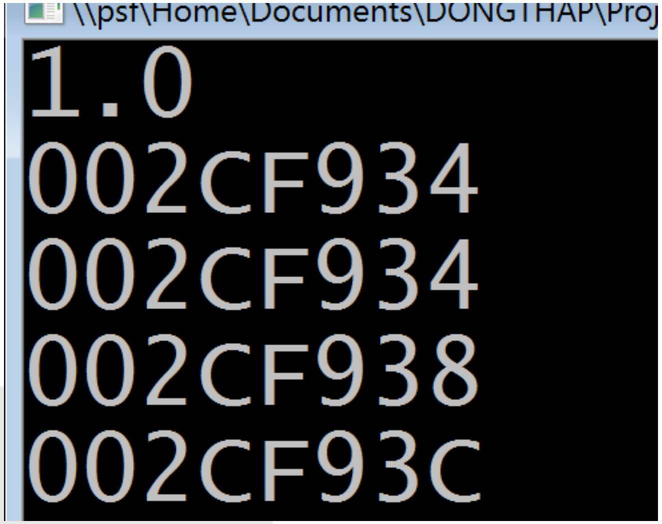
Print out the value of p1.x

Print out the address of p1

Print out the address of p1.x

Print out the address of p1.y

Print out the address of p1.z



```
1.0
002CF934
002CF934
002CF938
002CF93C
```

Pointer declaration

Syntax

```
<Type name> * <Variable name>;
<Type name> * <Name of variable a> = 0;
<Type name> * <Name of variable a> =
                    &<Name of variable b>;
```

<Name of variable b>: Muse have type <Type name>, or have type that can be casted to <Type name>

0: Constant, is called NULL

Pointer declaration

Syntax

```
int a;
int *p1;
int *p2 = 0;
int *p3 = &a;
```

a: is an Integer
p1: pointer to Integer, unknown value
p2: pointer to Integer, the value is NULL
p3: pointer to Integer, the value is the address of the memory cell of a

```
double d;
double *pd1;
double *pd2 = 0;
double *pd3 = &d;
```

```
float f;
float *pf1;
float *pf2 = 0;
float *pf3 = &d;
```

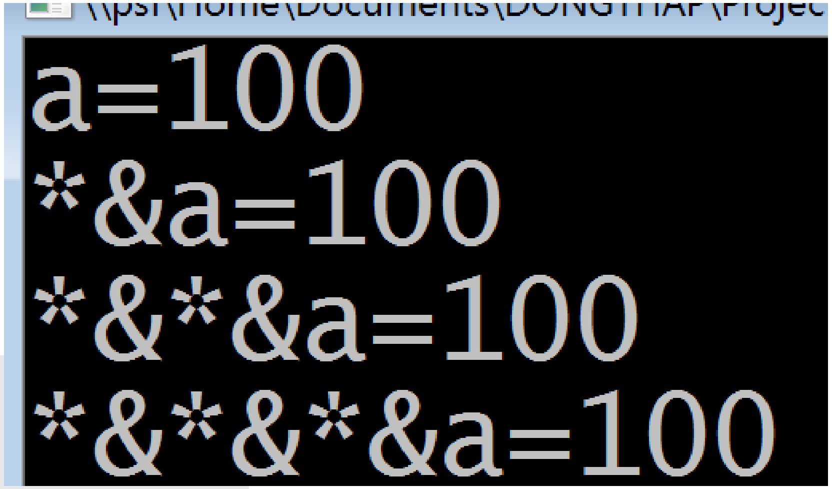
f: is float
pf1: pointer to float, unknown value
pf2: pointer to float, the value is NULL
pf3: pointer to float, the value is the address of the memory cell of f

```
Point3D p1 = {1.0f, 2.0f, 3.0f};
Point3D *pp1;
Point3D *pp2 = 0;
Point3D *pp3 = &p1;
```

* Operator

- The operator * get the reference value at an address
- Operator & get address of variable

* Operator



```
a=100
*&a=100
*&*&a=100
*&*&*&a=100
```

```
#include <iostream>
using namespace std;
int main(){
	int a = 100;

	cout << "a=" << a << endl;
	cout << "*&a=" << *&a << endl;
	cout << "*&*&a=", *&*&a << endl;
	cout << "*&*&*&a=%d\n" <<*&*&*&a
<< endl;

	system("pause");
	return 0;
}
```

Operations on the pointer

- Increase and Decrease : ++, --
- Addition and subtraction : +, -
- Addition and subtraction combined with assignment : +=, -=
- Comparation: ==, !=

Let p be a pointer of type T: **T *p;**
Addition and Subtraction: Make pointer p increase or decrease a multiple of size of type T.

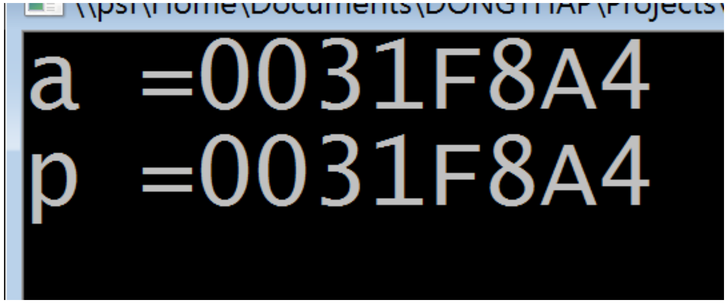
Pointer and Array

- Pointer and array have many similarities
  - Pointer and array : Keep address of memory cell
    - Pointer: Keeps the address of a certain memory cell (of another variable, of dynamic memory)
    - Array: Keep the address of the first element
    - => Can assign array to pointer
    - However, assigning pointer to array is not allowed

Pointer and Array

```cpp
#include <iostream>
using namespace std;
int main(){
        int a[5];
        int *p = a;

        cout << "a =" << a << endl;
        cout << "p ="<< p << endl;

        system("pause");
        return 0;
}
```

Array pointer can be assigned to pointer
=>
A and p have the same address: The address of the first cell on the array



```
a =0031F8A4
p =0031F8A4
```

Pointer and array

- Pointers and arrays have many similarities
  - Same way to access memory cells
    - Use [ ] operator
    - Use * and + operator

```
int a[5];
int *p = a;
int id = 2;

a[id] = 100;
p[id] = 100;

*(a + id) = 100;
*(p + id) = 100;
```

Same

Pointer and array

- Pointers and arrays also have differences
  - Array: The elements of the array are on the STACK of the program
  - Pointer: The elements that the pointer points to can be stored on STACK or HEAP

Dynamic memory allocation

- Help programmers create dynamic arrays. No need to determine the number of elements of dynamic array at compile time as static array.

- Dynamic arrays will be allocated on HEAP

When you use dynamic memory allocation
You **MUST** release memory after usage

Dynamic memory allocation

- Allocate dynamic memory
  - `new`
- Release dynamic memory
  - `delete`

Dynamic memory allocation

new

```
#include <iostream>

typedef struct sPoint3D{float x, y, z;} Point3D;
int main(){
       int *p1;
       float *p2;
       Point3D *p3;
       int num = 100;

       p1 = new int[num];
       p2 = new float[num];
       p3 = new Point3D();

       delete []p1; delete []p2; delete []p3;
       return 0;
}
```

Pointer variables

Allocate memory

Release memory

Dynamic memory allocation

new

```
p1 = new int[num];
```

num: number of elements

int: data type of each element

➔ int[num]: allocate num elements with int data type

Dynamic memory allocation

new

```
#include <iostream>
using namespace std;
int main(){
        int num = 100;
        int *p1 = new int[num];
        if (p1 == NULL){
                cout << "Can not allocate!" << endl;
                exit(1);
        }
        else{
                // Your work goes here…
                delete(p1);
        }
        return 0;
}
```

Dynamic memory allocation

new

```
int *p1 = new int[num];

If (p1 == NULL){ ...} else { ...}
```

Operator new returns a NULL pointer when it fails.
At that time, memory cannot be used!
Therefore, ALWAYS check if the new operator returns NULL

Dynamic memory allocation

Example

```
typedef struct{
        float x, y, z;
} Point3D;
int main(){
        int num = 20;
        int *int_ptr = new int[num];
        char *str = new char[num];
        double *double_ptr = new double[num];
        Point3D *p_ptr = new Point3D();
        // Usage
        delete(int_ptr);
        delete(str);
        delete(double_ptr);
        delete(p_ptr);

        return 0;
}
```

Declare pointer variables with data types + allocate dynamic memory

Release memory after usage

Pointer and struct

Declaration

```
typedef struct{
      float x, y, z;
} Point3D;
```

(1) Define struct data type: `Point3D`

```
Point3D *p_ptr = new Point3D();

// (4) Usage

delete(p_ptr);
```

(2) Declare a pointer points to an array

(3) Allocate dynamic memory on HEAP, `p_ptr`: store the address points to the first cell in the allocated block.

(5) Release memory

Pointer and struct

Access members of struct via pointer

Example: assign values to members of struct Point3D

```
(*p_ptr).x = 4.5f; (*p_ptr).y = 5.5f; (*p_ptr).z = 6.5f;

p_ptr->x = 7.5f; p_ptr->y = 8.5f; p_ptr->z = 9.5f;
```

p_ptr: memory cell (variable) contains address of struct Point3D

(*p_ptr): memory cell contains object with struct Point3D

(*p_ptr).x: memory cell contains member (variable x) of object

p_ptr->x: memory area contains variable x of struct Point3D accessed via the operator -> from the pointer **p_ptr**

Pointer and struct

Access members of struct via pointer

Example: assign values to members of struct Point3D

```
(*p_ptr).x = 4.5f; (*p_ptr).y = 5.5f; (*p_ptr).z = 6.5f;

p_ptr->x = 7.5f; p_ptr->y = 8.5f; p_ptr->z = 9.5f;
```

General:

```
<pointer> -> <member of struct>
```

Example:

```
p_ptr->x
```

Pointer and struct

Example

```
# include <iostream>
using namespace std;
typedef struct{
        float x, y, z;
} Point3D;
int main(){
        Point3D p = {1.5f, 2.5f, 3.5f};
        Point3D *p_ptr = new Point3D();
        (*p_ptr).x = 4.5f; (*p_ptr).y = 5.5f; (*p_ptr).z = 6.5f;
        p_ptr->x = 7.5f; p_ptr->y = 8.5f; p_ptr->z = 9.5f;

        cout << "p=[" << p.x << ", " << p.y << ", " << p.z << "]";
        cout << "*p_ptr = [" << (*p_ptr).x << ", " << (*p_ptr).y <<
", " << (*p_ptr).z << "]";
        cout << "*p_ptr = [" << p_ptr->x << ", " << p_ptr->y << ",
" << p_ptr->z << "]";
        delete(p_ptr);
        system("pause");
        return 0;
}
```

Order of operations `*, ++ và --`

- `*p++   // *(p++)`
- `*++p   // *(++p)`
- `++*p   // ++(*p)`
- `(*p)++ // Increases the memory area pointed to by the pointer p`

When in doubt, or do not remember … use the operator () to resolve the priority

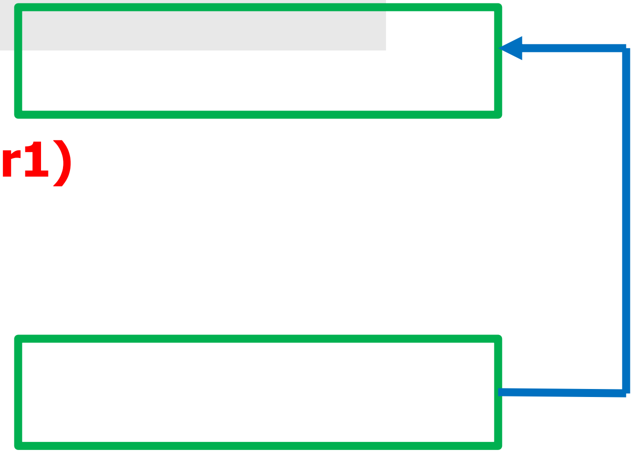
Pointer and const

```
int a = 20, b = 30, c = 40;

const int * ptr1 = &a;
//int const * ptr1 = &a;

int* const ptr2 = &b;
```

`ptr1`: can be changed.

The value that `ptr1` points to cannot be changed through `*ptr1`

Memory ptr1 points to



(Can not be changed via ptr1)

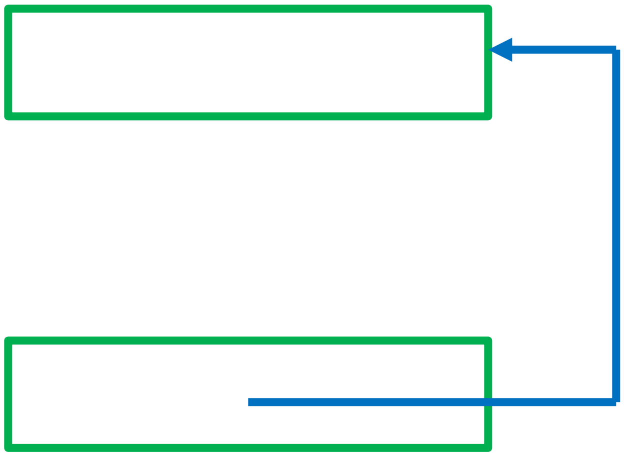
ptr1:

Pointer and const

```
int a = 20, b = 30, c = 40;

const int * ptr1 = &a;

int* const ptr2 = &b;
```

ptr2: Cannot change the value of ptr2 = cannot make ptr2 point to a different address after this line.

The value that **ptr2** points to can be changed via ***ptr2**.



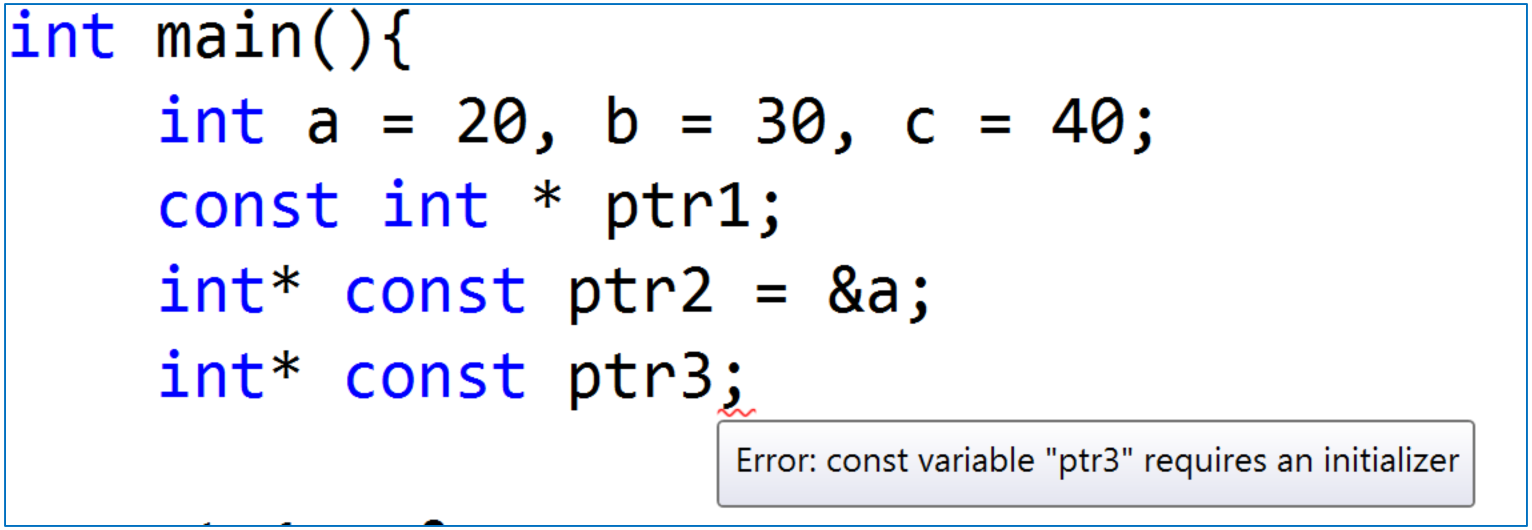
Memory ptr2 points to

ptr2:

(ptr2 can not be changed)

Pointer and const

Common mistakes



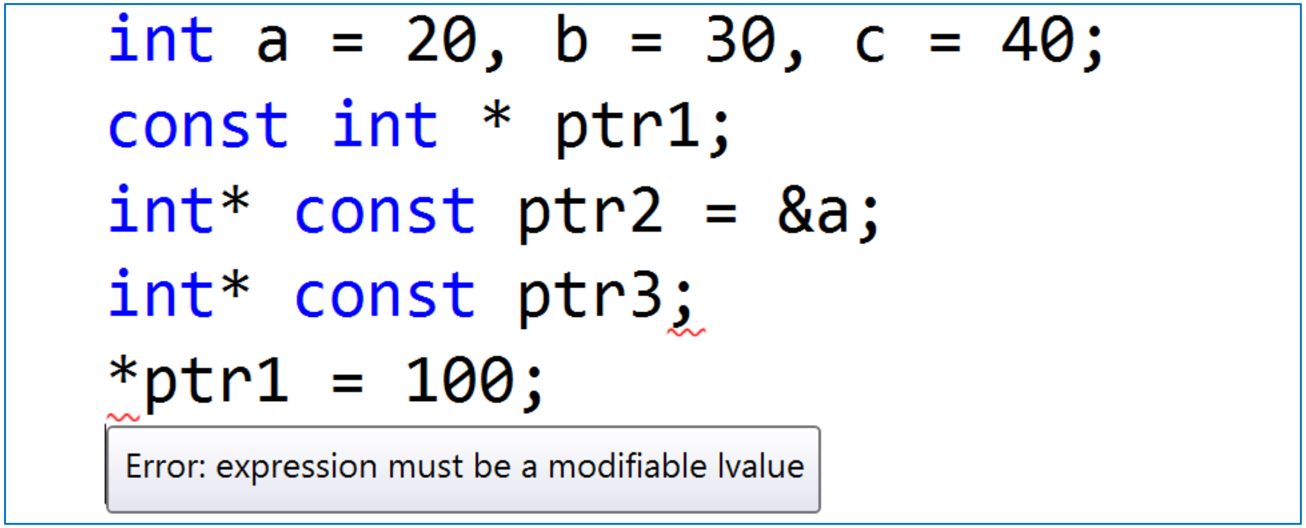
```
int main(){
    int a = 20, b = 30, c = 40;
    const int * ptr1;
    int* const ptr2 = &a;
    int* const ptr3;
```

Error: const variable "ptr3" requires an initializer

Ptr3: is a constant pointer but it is not initialized likes pointer ptr2

Pointer and const

Common mistakes



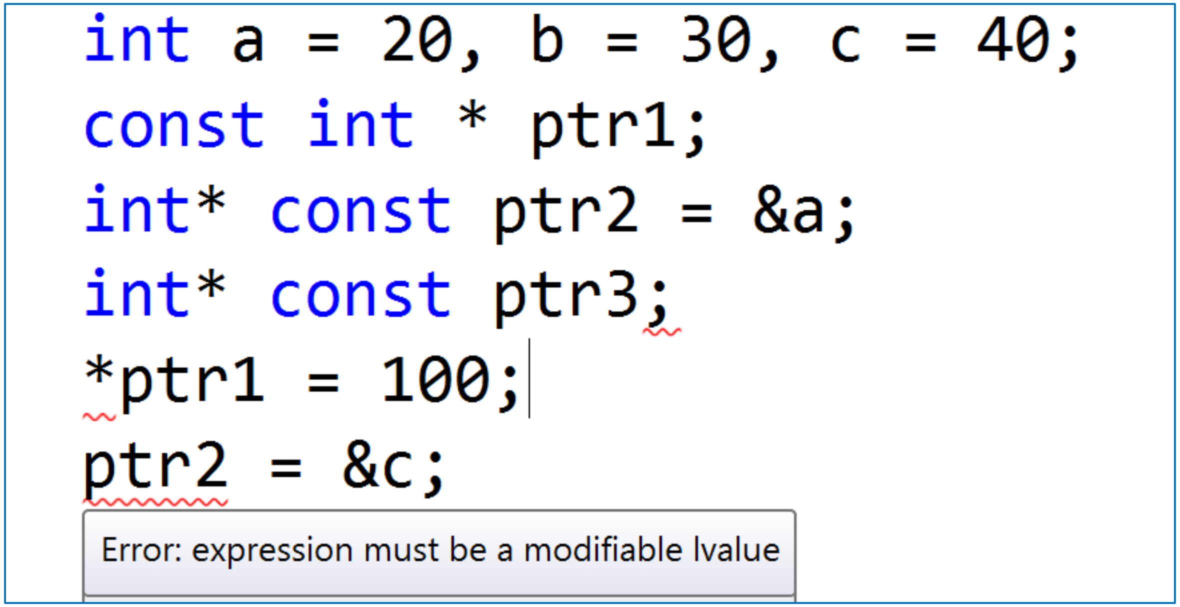
```
int a = 20, b = 30, c = 40;
const int * ptr1;
int* const ptr2 = &a;
int* const ptr3;
*ptr1 = 100;
```

Error: expression must be a modifiable lvalue

The value that ptr1 points to cannot be changed via *ptr1. Therefore, it cannot be on the left of the assignment expression

Pointer and const

Common mistakes



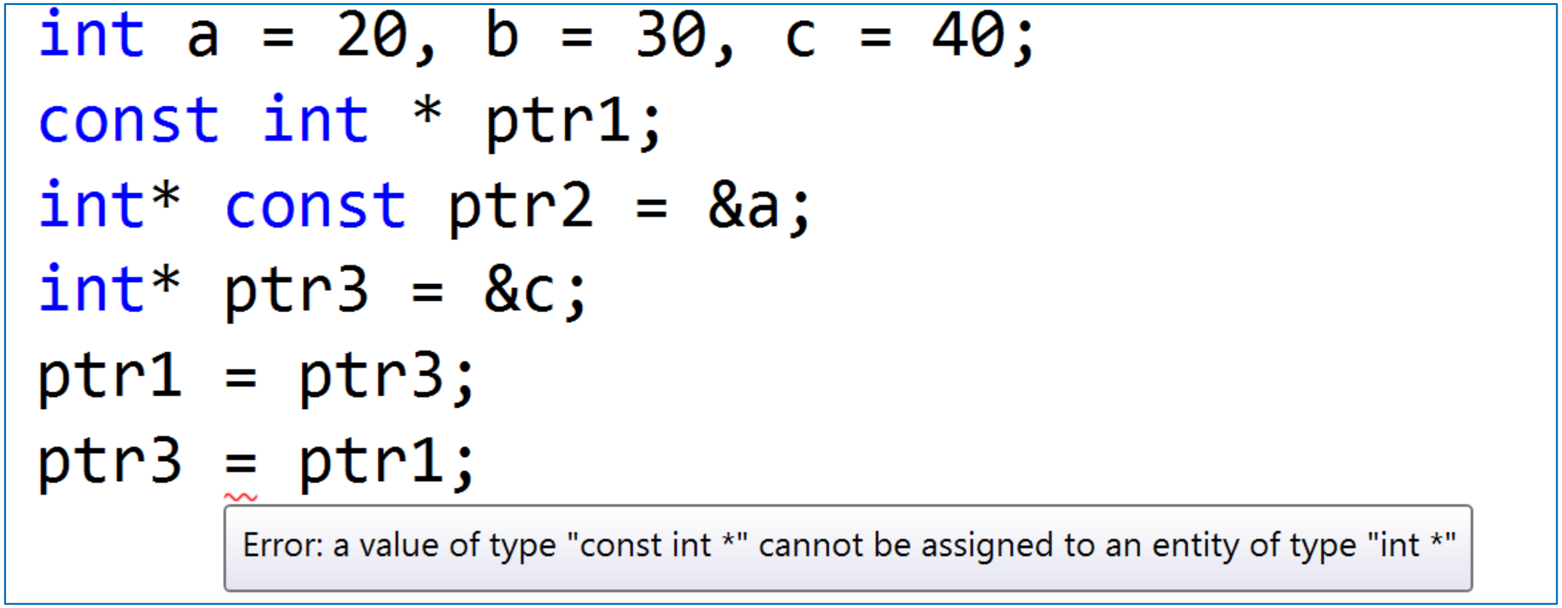
```
int a = 20, b = 30, c = 40;
const int * ptr1;
int* const ptr2 = &a;
int* const ptr3;
*ptr1 = 100;
ptr2 = &c;
```

Error: expression must be a modifiable lvalue

Pointer ptr2 is a constant and is initialized when it is created
Then, it cannot make ptr2 point to any other object

Pointer and const

Common mistakes



```
int a = 20, b = 30, c = 40;
const int * ptr1;
int* const ptr2 = &a;
int* ptr3 = &c;
ptr1 = ptr3;
ptr3 = ptr1;
```

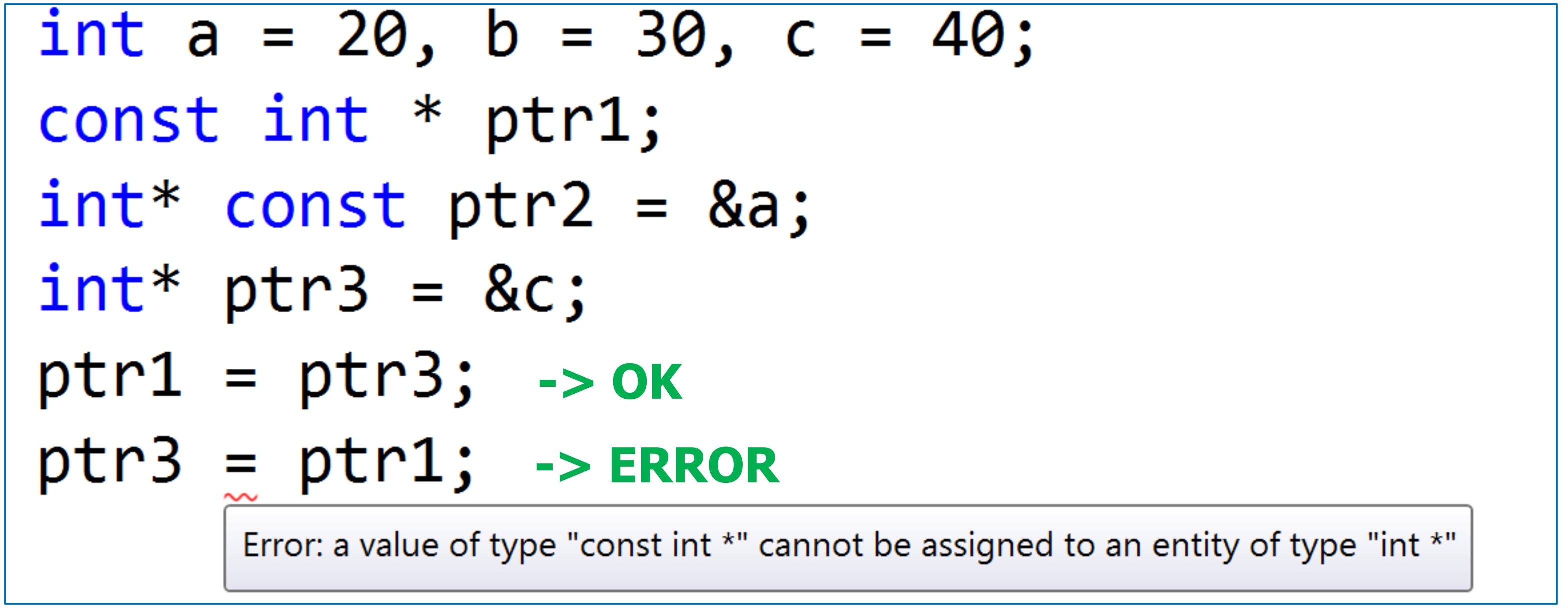
Error: a value of type "const int *" cannot be assigned to an entity of type "int *"

ptr3: is a pointer, can be changed or change the value it points to

Assign pointer ptr1 to ptr3: makes the value that ptr1 points to changeable
→ This is not allowed by compiler
Because if it allows, the ptr1 is meaningless.
Programmers can always change the content that ptr1 points to by using temporary pointer.

Pointer and const

Common mistakes



```
int a = 20, b = 30, c = 40;
const int * ptr1;
int* const ptr2 = &a;
int* ptr3 = &c;
ptr1 = ptr3;  -> OK
ptr3 = ptr1;  -> ERROR
```

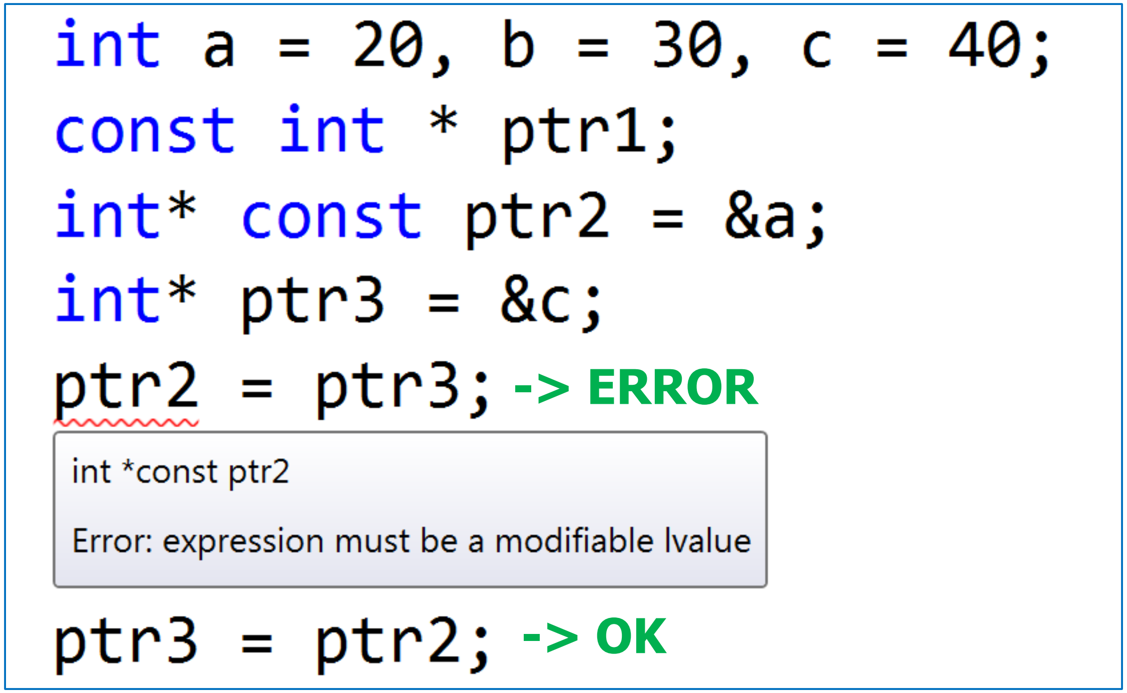
Error: a value of type "const int *" cannot be assigned to an entity of type "int *"

ptr3: is a pointer, can be changed or change the value it points to

Assign pointer ptr1 to ptr3: makes the value that ptr1 points to changeable
→ This is not allowed by compiler
Because if it allows, the ptr1 is meaningless.
Programmers can always change the content that ptr1 points to by using temporary pointer.

Pointer and const

Common mistakes



```
int a = 20, b = 30, c = 40;
const int * ptr1;
int* const ptr2 = &a;
int* ptr3 = &c;
ptr2 = ptr3; -> ERROR
```

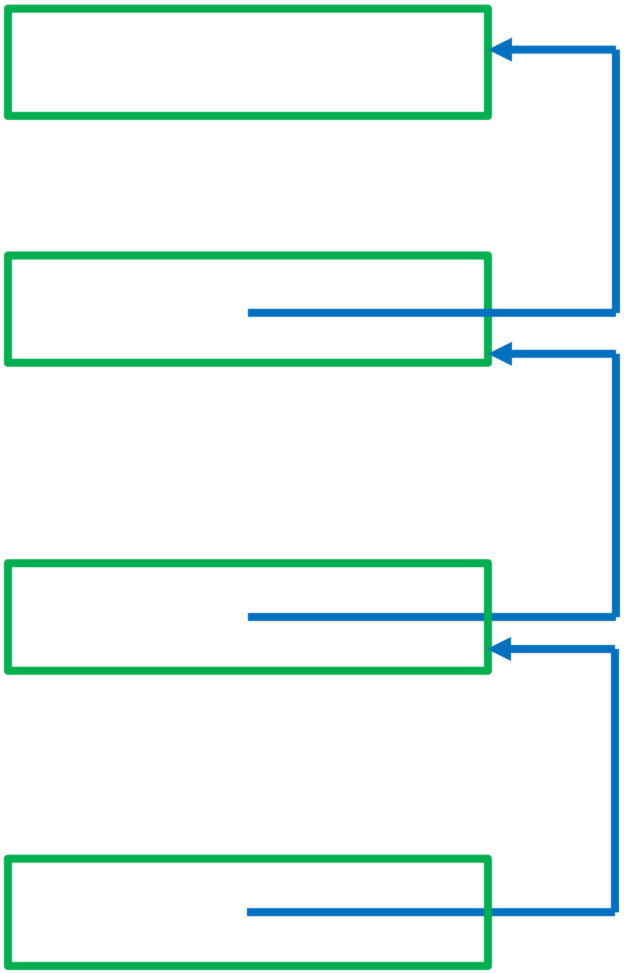
int *const ptr2

Error: expression must be a modifiable lvalue

```
ptr3 = ptr2; -> OK
```

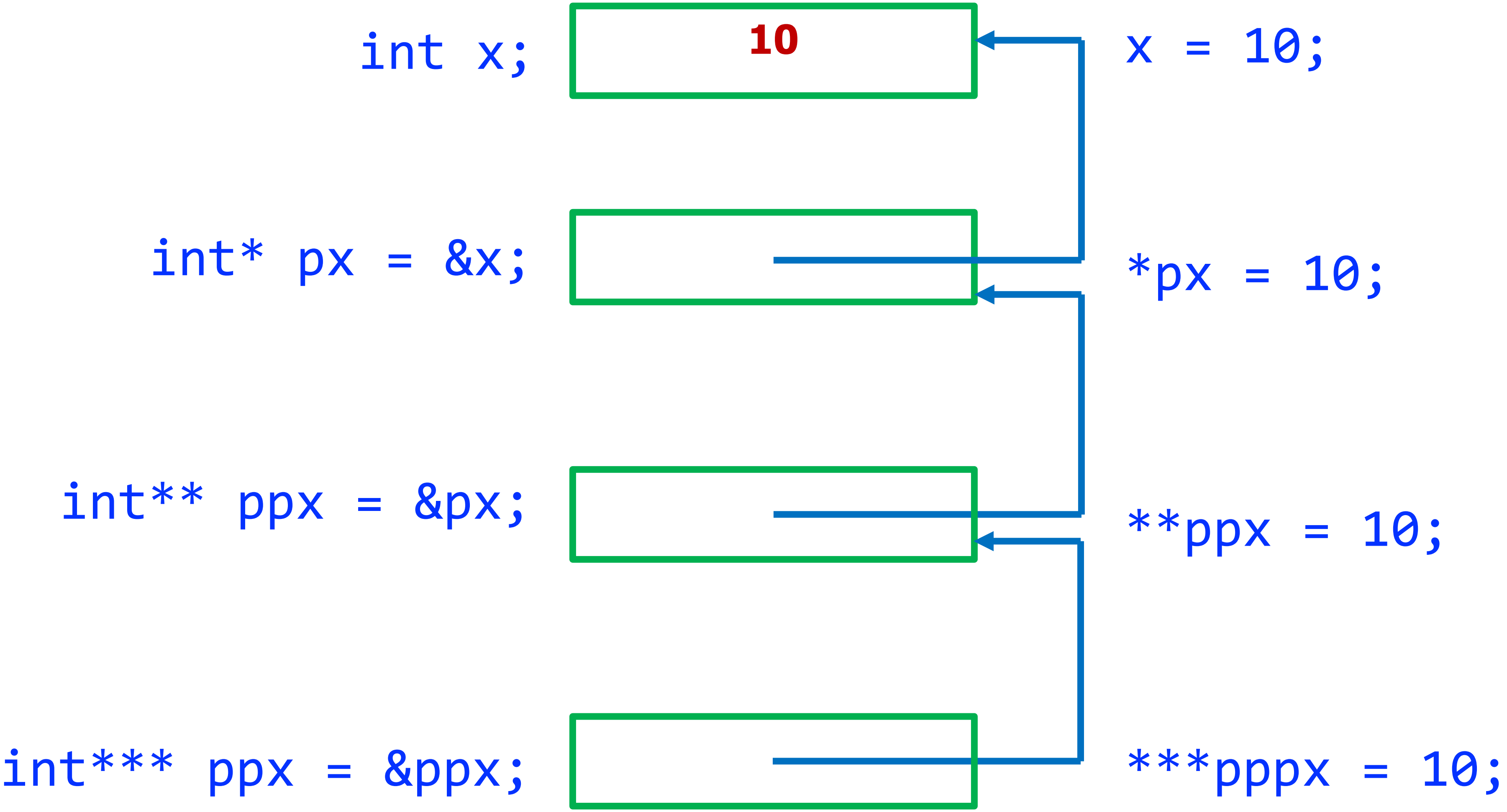
ptr2: can not be changed

Pointer to pointer



```
         int x;

    int* px = &x;

  int** ppx = &px;

int*** ppx = &ppx;
```

Pointer to pointer



```
int x;

int* px = &x;

int** ppx = &px;

int*** ppx = &ppx;
```

```
x = 10;

*px = 10;

**ppx = 10;

***pppx = 10;
```

Void pointer

- void *ptr: is pointer with undefined data type
  - Can be cast to any desired data type
  - Gives programs a great flexibility
  - But it has risks: the compiler cannot check data type compatibility at compile time

Exercise

- Re-implement all exercises related to array, but all data must be stored in HEAP memory