this pointer (remaining part)

- static - this pointer does not work with static -> no this pointer -> static is not associated with a object, it is associated with your class
- return current object - return *this
- Used for function chaining

```
class Student{
public:
    int rollNo;

    //error: 'this' is unavailable for static member functions
    static Student& showRollNo(){
        cout << rollNo << endl;
        return *this;
    }
};

int main(){
    Student s1;
    s1.showRollNo();
    return 0;
}
```

```
//function chaining using this pointer
class Student{
public:
    int rollNo;

    Student& setRollNo(int rollNo){
        this->rollNo = rollNo;
        return *this;
    }

    void showRollNo(){
        cout << "Roll No : " << rollNo << endl;
    }
};

int main(){
    Student s1;
    s1.setRollNo(100).showRollNo(); // function chaining
    return 0;
}
```

Initialization vs Assignment

- Initialization - when we create the variable/object and assign the value at the same time
- Assignment - when we create the variable/object first and then assign the value later
- Example of Initialization and Assignment

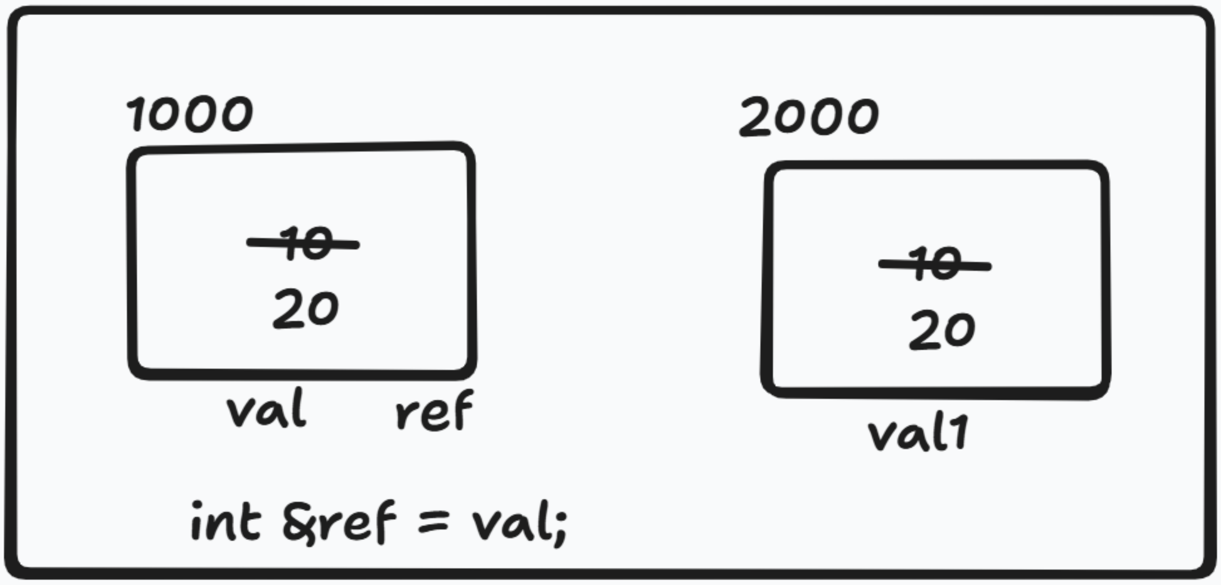
```
#include <iostream>
using namespace std;

class Student{
public:
    int rollNo;
    string name;

    Student(){
        this->name = "";
        this->rollNo = 0;
    }

    Student(int rollNo, string name){ // Initialization
        this->name = name;
        this->rollNo = rollNo;
    }

    void setData(int rollNo, string name){ //Assignment
        this->name = name;
        this->rollNo = rollNo;
    }
};

int main() {
    Student s1;     // Declaration

    Student s2(100,"shil");     //Initialization

    s1.setData(102, "khetesh"); //Assignment

//  Initialization -> Constructor
//  Assignment  -> Setter functions or Constructor
    int val = 10;   //Initialization -> when we create it -> value assigned

    int val2;       // Declaration -> when create it -> value nothing

    val2 = 20;      //Assignment -> already created variable -> value assigned
}
```

Reference variable

- It is an alias name for an existing variable
- It is created using & symbol
- It must be initialized at the time of declaration
- It cannot be null and cannot be changed to refer another variable

- It is used to implement call by reference
- It is used to avoid copying of large data structures
- It is used in operator overloading and function chaining

```cpp
int main(){

    int val = 10;

    int &ref = val;  // this is your reference variable -> just an alias val
variable -> nickename
                    // it doesnt acquire -> memory space
    int val1 = 20;

//  int &ref1;  //error: 'ref1' declared as reference but not initialized

//  int &ref1 = NULL; // error: invalid initialization of
                     // non-const reference of type 'int&' from an rvalue of type
'int'
    ref = val1; // changes -> that will reflected in the main value

    cout << val << endl;
    cout << ref << endl;

    return 0;
}
```



Call by value/address/reference

- Related to function arguments passing
- Call by Value - passing the copy of value to the function

- Call by Address - passing the address of value to the function
- Call by Reference - passing the reference of value to the function

Call by Value

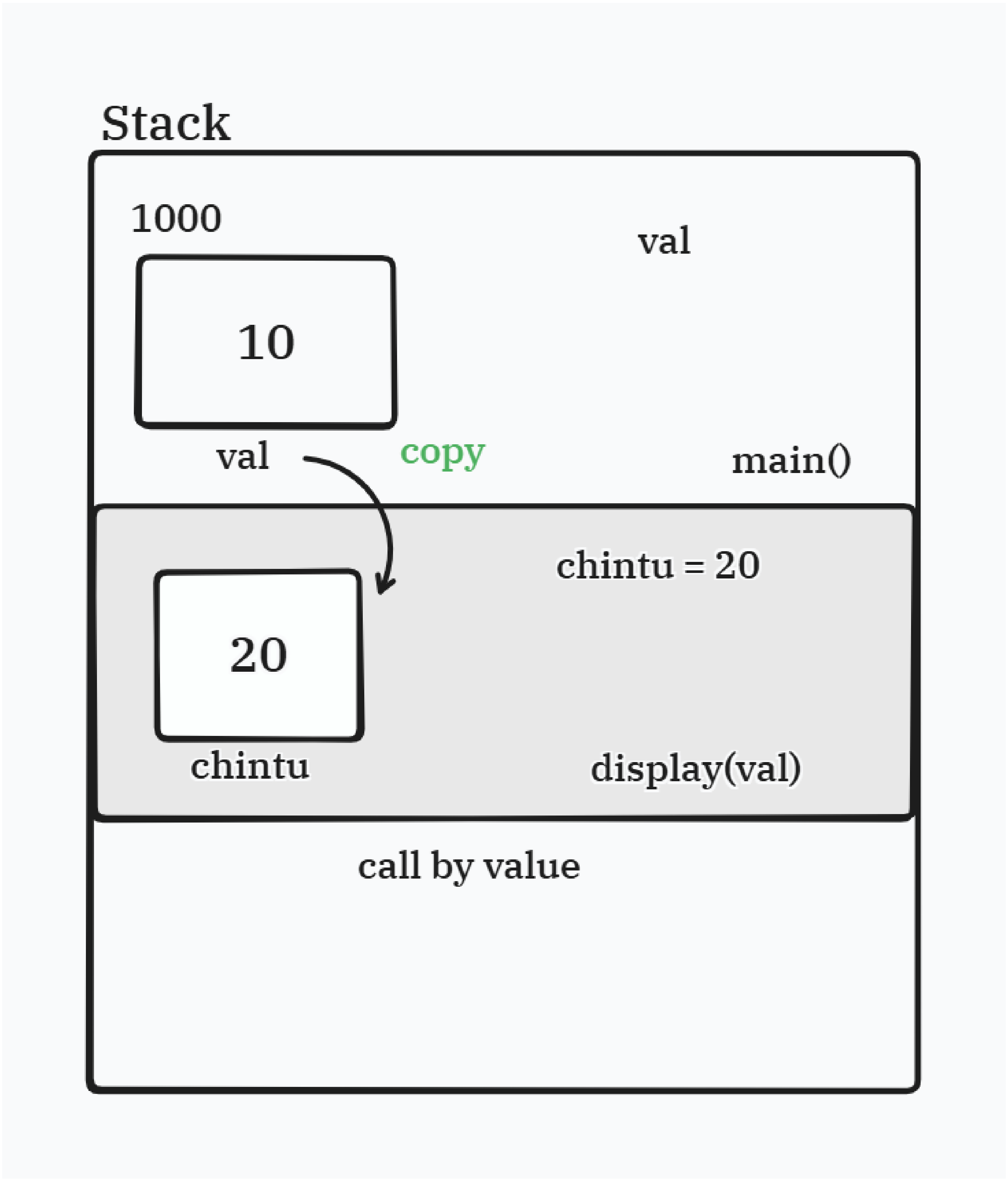
- Copy of value is passed to the function
- Original value remains unchanged
- Used for small data types and less efficient for large data types

> Syntax: data_type variable

```
    void display(int val){ //10
    val = 20;
    cout << "Variable inside function: " << val << endl;
}

int main() {
    int val = 10;

    cout << "Variable before function call: " << val << endl;

    display(val);  //10  // pass by value -> val = copy

    cout << "Variable after function call : " << val << endl;

    return 0;
}
```

- Output

```
Variable before function call: 10
Variable inside function: 20
Variable after function call : 10
```



{width=300 height=150}

Call by Reference

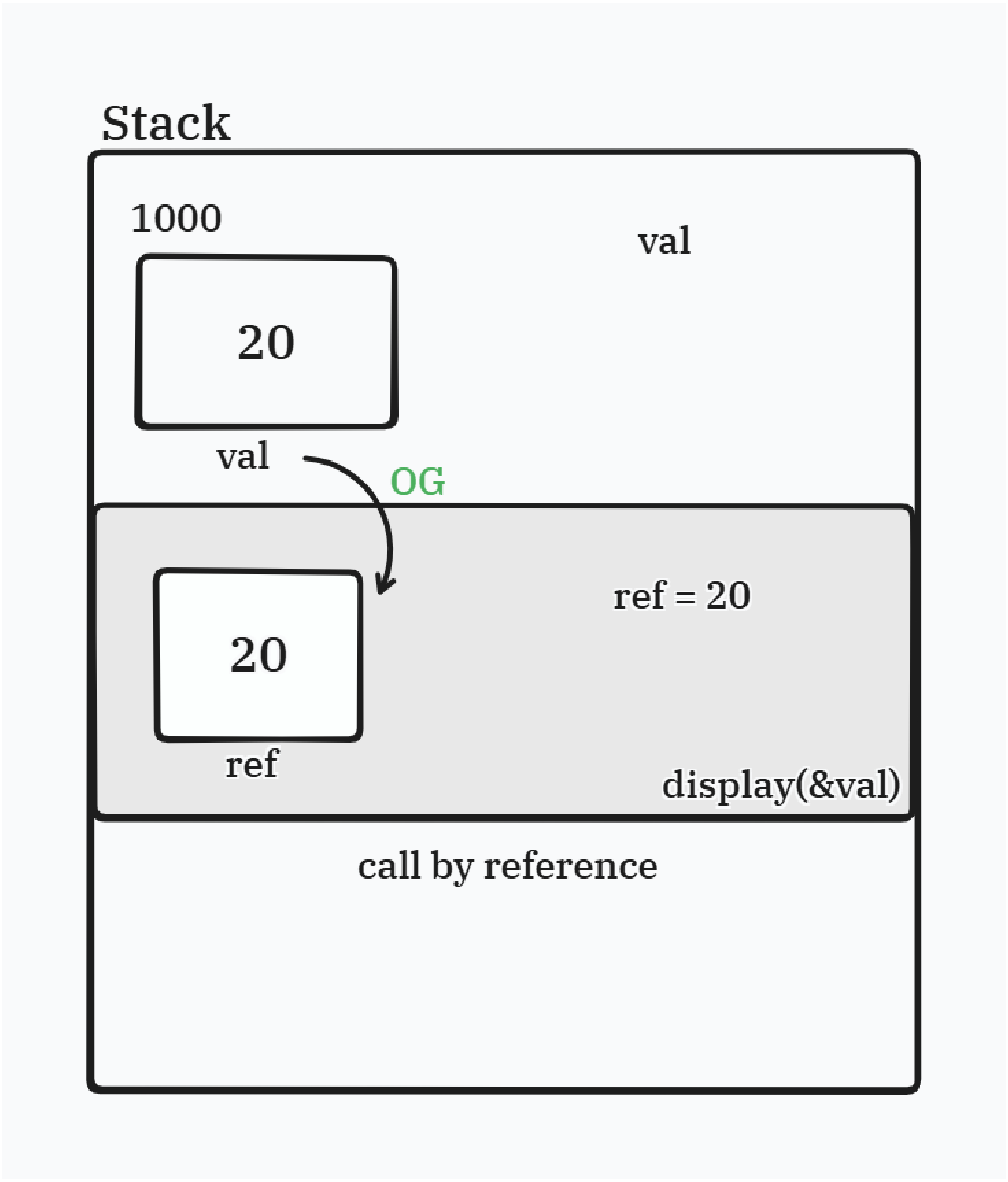
- Reference of value is passed to the function
- Original value can be changed
- Used for large data types and more efficient

> Syntax: data_type &ref_variable

```
void display(int &ref){ // value(copy) -> reference(original)
    ref = 20;
    cout << "Variable inside function: " << ref << endl;
}

int main() {
    int val = 10;

    cout << "Variable before function call: " << val << endl;

    display(val);  //10  // pass by reference -> val = original

    cout << "Variable after function call : " << val << endl;

    return 0;
}
```

- Output

```
Variable before function call: 10
Variable inside function: 20
Variable after function call : 20
```



{width=300 height=150}

Call by Address

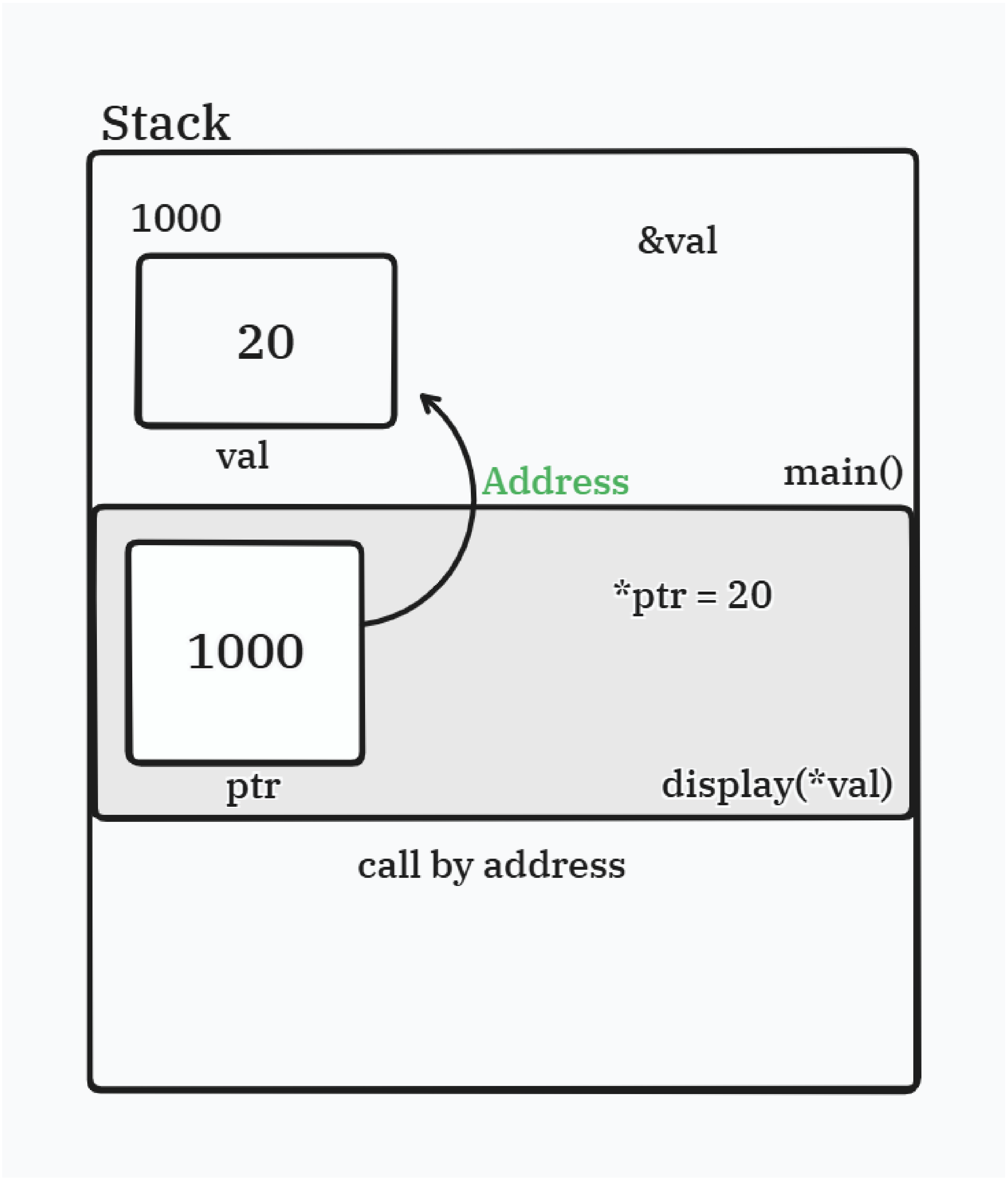
- Address of value is passed to the function
- Original value can be changed
- Used for large data types and more efficient

> Syntax: data_type *ptr_variable

```
void display(int *ptr){
    *ptr = 20;
    cout << "Variable inside pointer function: " << *ptr << endl;
}

int main() {
    int val = 10;

//  int *ptr = &val;  // assigning a address -> ptr

    cout << "Variable before function call: " << val << endl;

    display(val);  //10  // pass by reference -> val = original

    display(&val); // pass by address
    cout << "Variable after function call : " << val << endl;

    return 0;
}
```

- Output

```
Variable before function call: 10
Variable inside function: 20
Variable after function call : 20
```



{width=300 height=150}

Pointer vs Reference

- **Reference** -> It is just an alias for variable/object name (nickname)
    - No separate memory
    - Same variable address
- **Pointer** -> It is a variable which stores the address of another variable.

- Stores address
- Different address

- **Reference** -> It acts like your normal variable -> just syntax symbol &

- **Pointer** -> It is a special variabel -> syntax symbols & * ->

- **Complexity** -> Pointer > Reference

- **Reference**

    - Function call -> display(value)
    - Function definition display(data_type &value)
    - Function body{*ptr}

- **Pointer**

    - Function call -> display(&value)
    - Function definition display(data_type *value)
    - Function body{val}

Dynamic memory (new/delete)

- **Stack** is static memory
    - Stack has fixed size
- **Heap** is dynamic memory
    - Heap expands memory as required

Dynamic Memory Allocation

- **new**
    - creates object on heap -> dynamic memory
    - malloc, calloc, realloc -> it is also present
- **delete**
    - destroy the object and release memory
    - free -> it is also present

Difference: malloc vs new

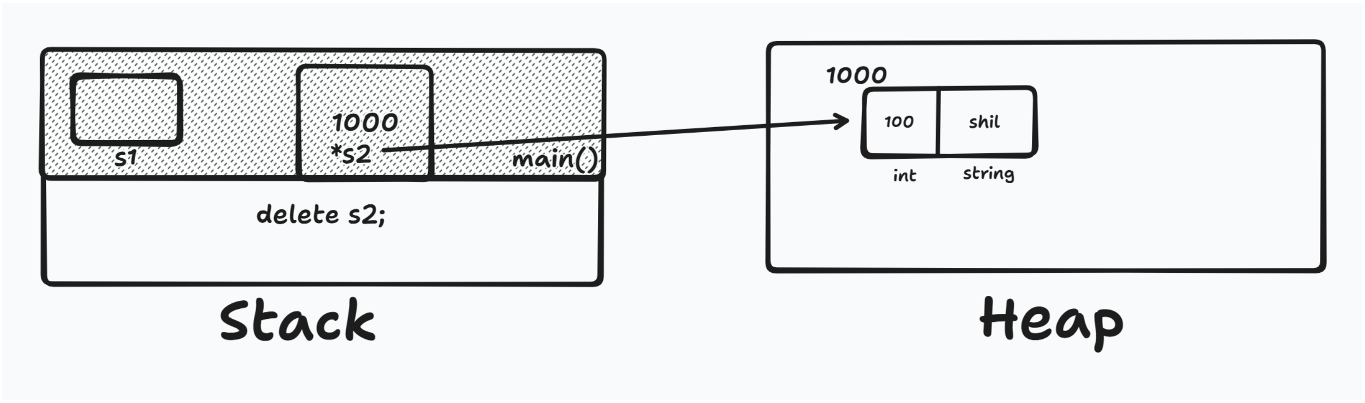
malloc

- It is a function which allocates memory on function call.
- It does not call your constructor
- It returns void pointer which needs to be typecasted.

new

- It is keyword which allocates memory for a object of a class.
- It uses constructor to initialize the values.
- It returns the specific type pointer.

```
class Student{
    int rollNo;
    string name;
//member function this pointer
public:
    Student(){
        this->rollNo = 0;
        this->name = "";
    }

    Student(int rollNo, string name){
        this->rollNo = rollNo;
        this->name = name;
    }
};

int main() {
    Student s1;                     // Object on Stack
    Student *s2 = new Student(101, "shil");     //Object on Heap // c++ way

    Student *s3 = (Student*) malloc(sizeof(Student)); //-> to return void* // c
style

//  new, delete -> both are keywords only
//  malloc(), free() -> both are functions

    free(s3); // c style

    delete s2; // c++ way

    return 0;
}
```



Constructor types

Constructor

- There is no role of constructor in your object creation
- Initialize the object/values

- If you write the parameterized constructor -> you must write default constructor as well.

Constructor Types

- **Default Constructor**
  - It is used to initialize the object with default values
  - It takes no parameters
  - It is called automatically when the object is created
- **Parameterized Constructor**
  - It is used to initialize the object with user-defined values
  - It takes parameters
  - It is called automatically when the object is created
  - It can be overloaded
- **Copy Constructor**
  - It is used to make copy of one to another object
  - It takes reference of same class object as parameter
  - It is called when we create new object from existing object
- **Move Constructor**
  - It takes the values from one object/variable to other object and make the first object null -> steals the values
  - It takes rvalue reference of same class object as parameter
  - It is called when we create new object from temporary object
  - It is used to optimize the performance of the program
- **Delegate Constructor**
  - delegates the task of initialization to another constructor
  - It is used to avoid code duplication
  - It calls another constructor in the same class

```
class Student{
    int rollNo;
    string name;

//member function this pointer
public:
    //default cons + delegate cons
    Student() : Student(rollNo){
        cout << "default cons" << endl;
        this->rollNo = 0;
        this->name = "";
    }

    //parameterized cons
    Student(int rollNo, string name){
        cout << "para cons" << endl;
        this->rollNo = rollNo;
        this->name = name;
    }

    //delegating cons from default
    Student(int rollNo){
        this->rollNo = rollNo;
    }

    //copy cons
    Student(const Student &stud){
        cout << "copy cons" << endl;
        this->rollNo = stud.rollNo;
        this->name = stud.name;
    }

    //move cons
    Student(Student &&stud){
        cout << "move cons" << endl;
        this->rollNo = stud.rollNo;
        this->name = stud.name;
    }
};

int main(){

    Student s1; //calls default cons

    Student *s2 = new Student(100, "shil"); // parameterized cons

    Student s3 = s1; //copy cons = assigment -> operator overload

    Student s4 = move(s3);  // != delete

    return 0;
}
```

Constructor initializer list

- It is used to initialize the data members of the class
- Faster way to initialization
- Used with const data member and reference data member
- It can be used with normal data members as well

> Syntax : constructor_name(): data_member(value), data_member2(value2) {constructor_body}

Object Creation steps

```
1. Memory allocation for object
2. Constructor initializer list
3. Constructor body
4. Object is ready to use
5. Object destruction
```

```
class Student{
    const int rollNo; // const
    string name;

public:
    //default cons
    Student() : rollNo(0), name(""){

    }

    //parameterized cons
    Student(int rollNo, string name) : rollNo(rollNo), name(name){ // initiliaze -
> initializer list
                        //assignment
    }

    void printData(){
        cout << name << endl;
        cout << rollNo << endl;
    }
};

int main(){

    Student s1(20,"shil"); //not initialization

    Student s2(30, "ram");

//  with the help of initializer list we do initialization

    s1.printData();
    s2.printData();

//s1 -> memory allocate -> add values to it
    return 0;
}
```

const data member

- const data member must be initialized using initializer list
- const data member cannot be modified

```
class Student{
    const int MAX_MARKS = 100; // const with initialization
    const int rollNo; // const which can be initialized only using initializer
list
    string name;
public:
    //parameterized cons
    Student(int rollNo, string name) : rollNo(rollNo), name(name){
    }
    void printData(){
        cout << name << endl;
        cout << rollNo << endl;
    }
};
int main(){
    Student s1(20,"shil");
    s1.printData();
    return 0;
}
```

Tomorrow -

- const member function
- mutable member
- Aggregate initialization
- Arrays of objects
- Destructor