# Default arguments

A Default argument  is an argument that is passed automatically to a parameter if the argument is missing on the function call.

- Must be a constant, declared in prototype:

```
   void evenOrOdd(int = 0);
```

- Multi-parameter functions may have default arguments for some or all of them:

```
   int getSum(int, int=0, int=0);
```

# Default arguments – cont.

- If not all parameters to a function have default values, the default-less ones are declared first in the parameter list:

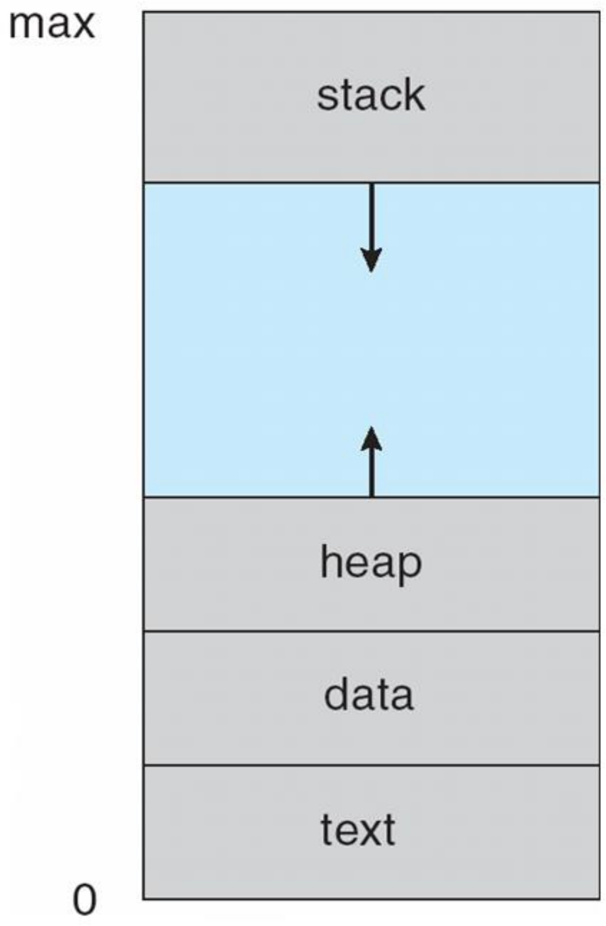
```
int getSum(int, int=0, int=0);// OK
int getSum(int, int=0, int);  // NO
```

- When an argument is omitted from a function call, all arguments after it must also be omitted:

```
 sum = getSum(num1, num2);    // OK
 sum = getSum(num1, , num3);  // NO
```

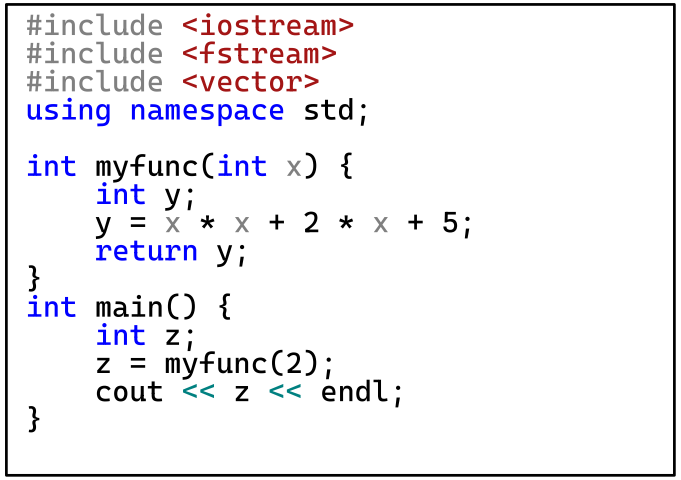
# A Program’s memory

- Generally speaking, every running program occupies an area in the computer’s memory.
- Within that area, the program’s own memory is made of 4 main components:
  - Text – This is where the compiled code (i.e. machine instructions) resides
  - Data – where global variables reside
  - Heap – where dynamically allocated data resides
  - Stack – The call stack is used for passing function parameters and holding local variables

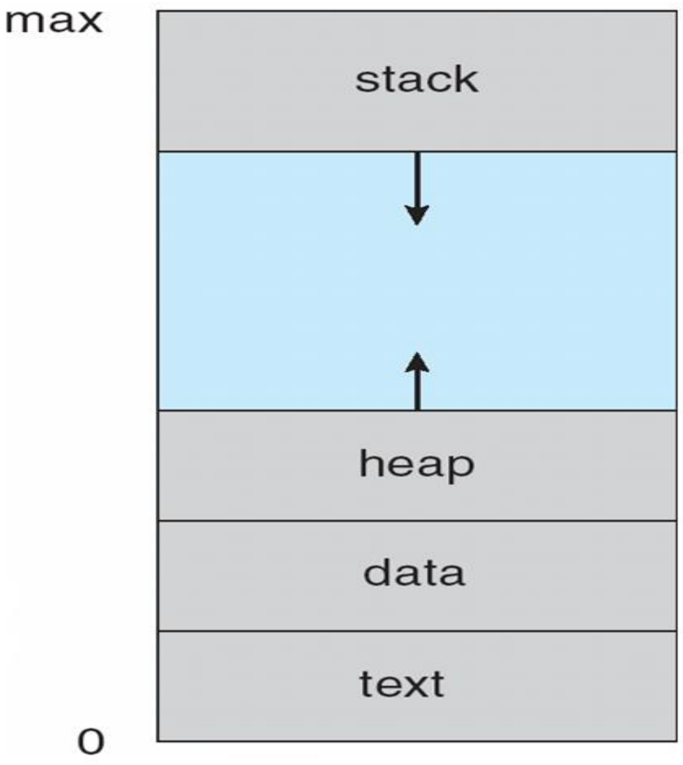


# The Call Stack

- Used for:
  1) Saving CPU registers before a function call (so they can be later restored after the function returns) – Beyond the scope of this course.
  2) Memory storage for a function’s **parameters** (e.g., x in myfunc). The parameter x occupies memory space in the stack.
  3) Memory storage for a function’s local variables (e.g., y in myfunc)



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

int myfunc(int x) {
    int y;
    y = x * x + 2 * x + 5;
    return y;
}
int main() {
    int z;
    z = myfunc(2);
    cout << z << endl;
}
```

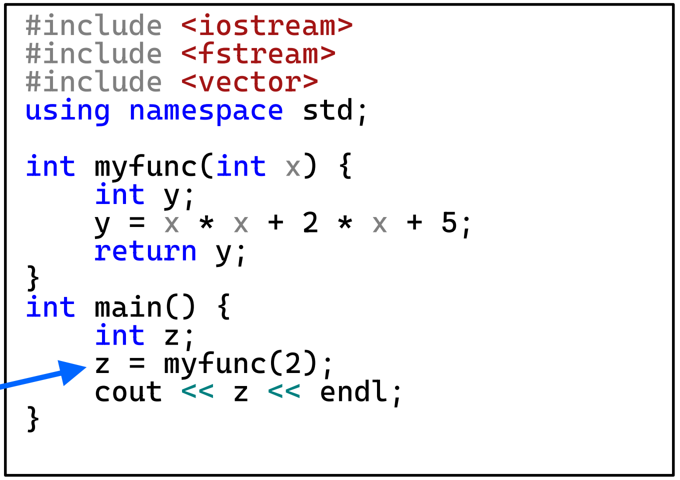


# The Call Stack – cont.

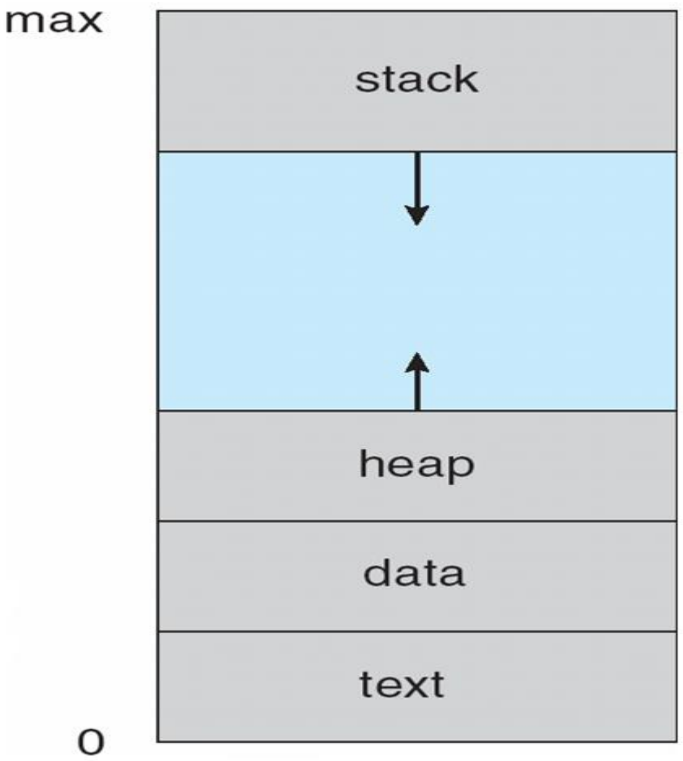
- Implements a Last-In-First-Out (LIFO) – has push() and pop() operations

- Thus, whenever there’s a function call in your code the compiler **IMPLICITLY inserts** machine code that:
  - Pushes the **arguments** into the stack (e.g., 2)
  - Allocates space for local variables inside the stack (e.g., y)

- Thus, a new “**stack frame**” is **implicitly created** whenever you make a function call

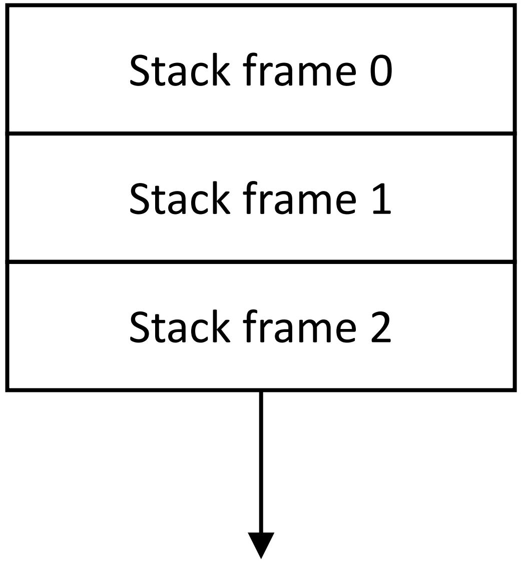


```
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int myfunc(int x) {
    int y;
    y = x * x + 2 * x + 5;
    return y;
}
int main() {
    int z;
    z = myfunc(2);
    cout << z << endl;
}
```

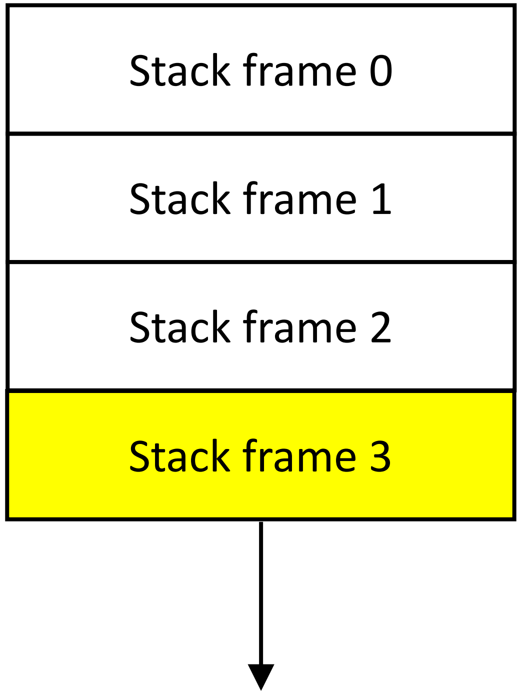


# The Call Stack - example

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

int myfunc(int x) {
    int y;
    y = x * x + 2 * x + 5;
    return y;
}
int main() {
    int z;
    z = myfunc(2);
    cout << z << endl;
}
```

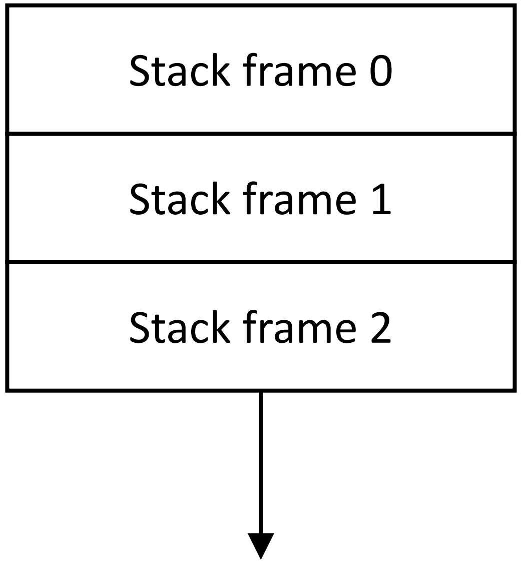


Before calling myfunc()



While myfunc() is executing:

- Parameter x is pushed into stack frame 3
- Local variable y is allocated in frame 3

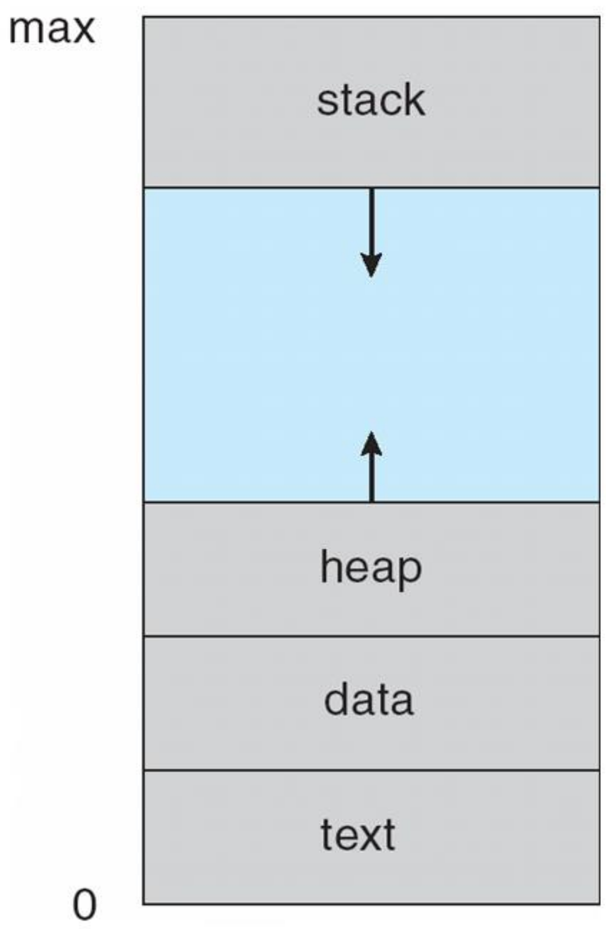


After myfunc() executed:

- Parameter x is deallocated
- Local variable y deallocated
- Stack frame is no more

# The heap and dynamic memory allocation

- Grows in opposite direction to stack
- Used for dynamically allocating variables
- You allocate using **new**
- You deallocate using **delete**



# Dynamic memory allocation – ex.

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

class Person {
    friend ostream& operator<<(ostream& os, const Person& someone) {
        os << "Person: " << someone.name << ", " << someone.age;
        return os;
    }
public:
    Person(const string& name, int age) : name(name), age(age) {}
    const string& getName() const { return name; }
private:
    string name;
    int age;
};

int main() {
    ifstream ifs("stooges.txt");
    vector<Person*> group1; // vector of pointers!
    string name;
    int age;
    while (ifs >> name >> age) {
        // push addresses from the heap!
        group1.push_back(new Person(name, age));
    }
    ifs.close();

    for (Person* p : group1) {
        cout << p->getName() << endl;
    }
    cout << "===========\n";
    for (Person* p : group1) {
        cout << p << ": " << *p << endl;
    }

    for (Person* p : group1) {
        delete p;
    }

}
```

```
Moe
Larry
Curly
Shemp
===========
00000212E56229C0: Person: Moe, 77
00000212E5622AA0: Person: Larry, 72
00000212E5622800: Person: Curly, 48
00000212E5622A30: Person: Shemp, 60
```

Much better!

# Multiple Associations + Dynamic memory alloc.

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

class Person {
    friend ostream& operator<<(ostream& os, const Person& someone) {
        os << "Person: " << someone.name << ", " << someone.age;
        return os;
    }
public:
    Person(const string& name, int age) : name(name), age(age) {}
    const string& getName() const { return name; }
private:
    string name;
    int age;
};

int main() {
    ifstream ifs("C:/Dropbox/CS2124_OOP_24S/Lect_07_stooges.txt");

    // Create vectors of pointers
    vector<Person*> group1;
    vector<Person*> group2;
    vector<Person*> group3;

    // parse file and load into the vector
    string name;
    int age = 0;
    while (ifs >> name >> age) {
        Person* pperson = new Person(name, age);
        group1.push_back(pperson);
        if (age <= 62) {
            group2.push_back(pperson);
        }
        else {
            group3.push_back(pperson);
        }
    }
```

Continued on next slide

# Multiple Associations + Dynamic memory alloc.

```
    // Test by modify a person's age in one group and see it reflect
    // in the other group --> thus same object
    group1[2]->setage(49);

    // print Person's pointer and Person
    cout << "Group 1 (all):" << endl;
    for (const Person* pper : group1) {
        cout << *pper << endl;
    }
    cout << endl << "Group 2 (<= 62):" << endl;
    for (const Person* pper : group2) {
        cout << *pper << endl;
    }

    cout << endl << "Group 3 (> 62):" << endl;
    for (const Person* pper : group3) {
        cout << *pper << endl;
    }

    // print person names (from group1)
    cout << "\nGroup 1 (all - names only)" << endl;
    for (const Person* p : group1) {
        cout << p->getName() << endl;
    }

    // delete all
    for (Person* p : group1) {
        delete p;
    }

    // close the file
    ifs.close();
}
```

```
Group 1 (all):
Person: Moe, 77
Person: Larry, 72
Person: Curly, 49
Person: Shemp, 60

Group 2 (<= 62):
Person: Curly, 49
Person: Shemp, 60

Group 3 (> 62):
Person: Moe, 77
Person: Larry, 72

Group 1 (all - names only)
Moe
Larry
Curly
Shemp
```

# Dynamic allocation - Is there a problem?

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

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); }

private:
    int* p;
};
ostream& operator<<(ostream& os, const Thing& rhs) {
    //return os << "Thing: " << *rhs.p;
    os << "Thing: " << *rhs.p;
    return os;
}

void doSomething() {
    Thing aThing(17);
    cout << aThing << endl;
}

int main() {
    doSomething();
    Thing* ptr = new Thing(100);
    delete ptr;
}
```

```
Thing: 17
```

- Is there anything wrong with this code?

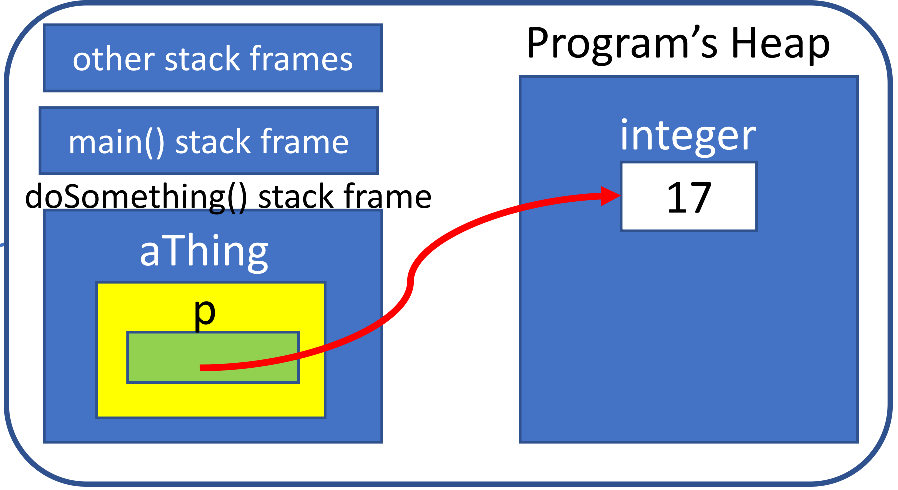
# Deallocation is needed

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

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); }

private:
    int* p;
};
ostream& operator<<(ostream& os, const Thing& rhs) {
    //return os << "Thing: " << *rhs.p;
    os << "Thing: " << *rhs.p;
    return os;
}

void doSomething() {
    Thing aThing(17);
    cout << aThing << endl;
}

int main() {
    doSomething();
    Thing* ptr = new Thing(100);
    delete ptr;
}
```

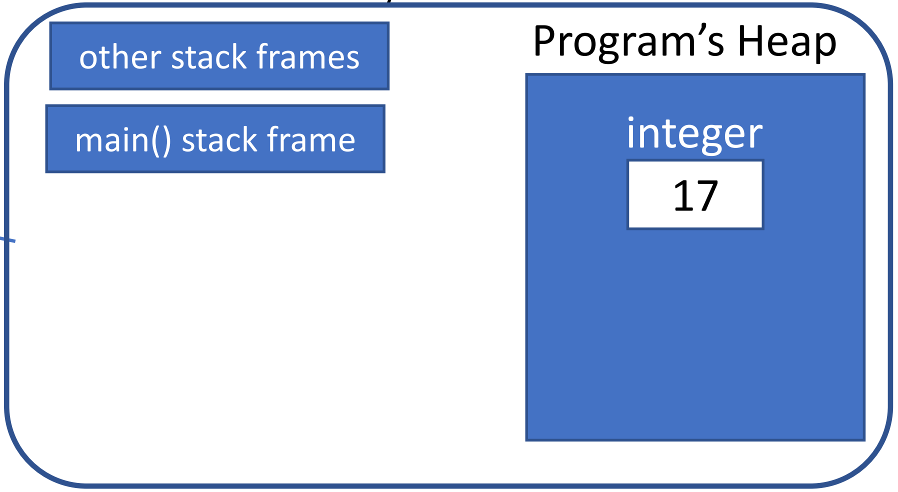
- Is there anything wrong with this code?
  - **Yes,** doSomething() left the dynamically allocated integer behind, with no way of accessing it or de-allocating it later on.

  → leaving garbage behind (**Memory leak**)!
- **How do we fix this?**

Memory View while executing in doSomething()



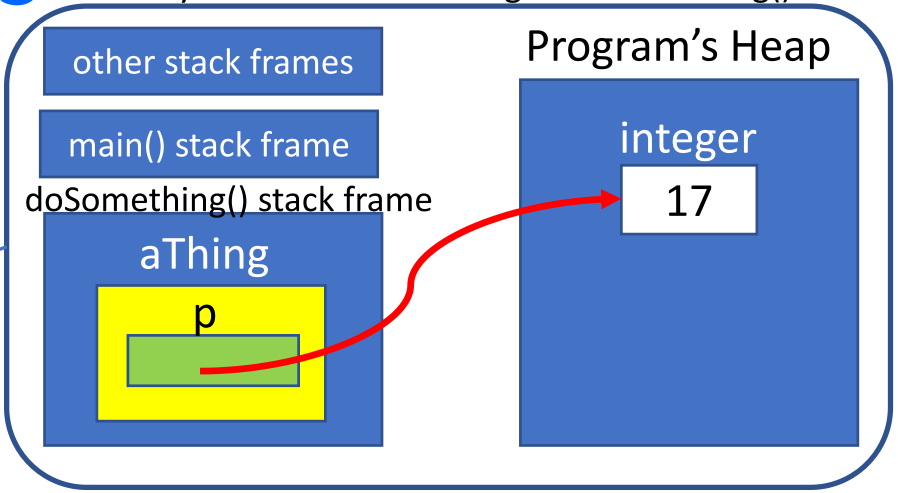
Memory View after doSomething() exits and its stack frame is destroyed



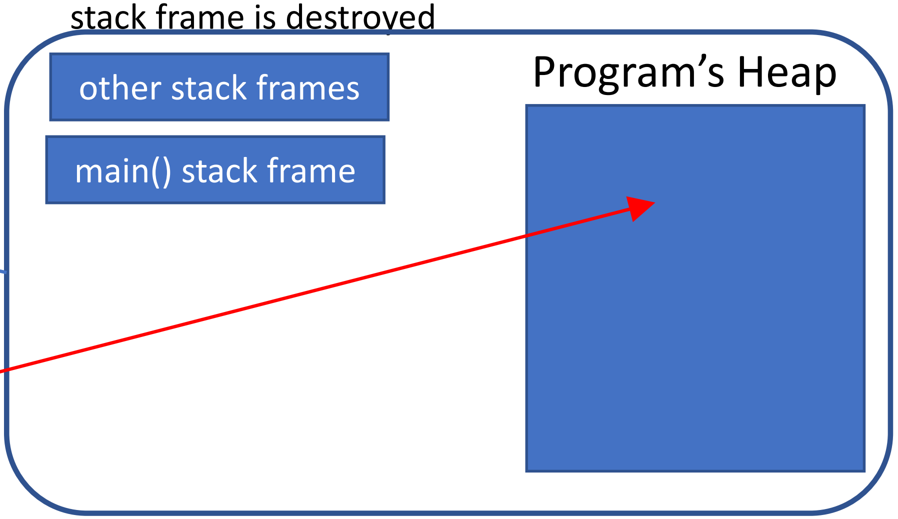
# Deallocation using Destructors

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

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); }
    ~Thing() { delete p; } // destructor
private:
    int* p;
};
ostream& operator<<(ostream& os, const Thing& rhs) {
    //return os << "Thing: " << *rhs.p;
    os << "Thing: " << *rhs.p;
    return os;
}
void doSomething() {
    Thing aThing(17);
    cout << aThing << endl;
}

int main() {
    doSomething();
    Thing* ptr = new Thing(100);
    delete ptr;
}
```

Memory View while executing in doSomething()



Memory View after doSomething() exits and its stack frame is destroyed



- The destructor is called automatically when doSomething exists() and the object is being destroyed
- The destructor de-allocated the dynamic memory

# Destructors – cont.

- The destructor is invoked either
  - When a function returns (i.e. at the closing curly brace)
  - When the `delete` keyword is used.

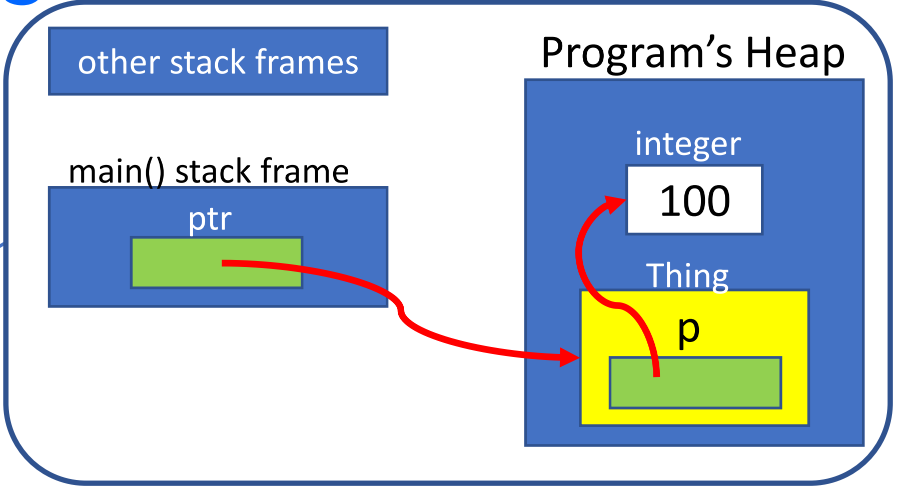
# Deallocation using Destructors

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

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); }
    ~Thing() { delete p; } // destructor
private:
    int* p;
};
ostream& operator<<(ostream& os, const Thing& rhs) {
    //return os << "Thing: " << *rhs.p;
    os << "Thing: " << *rhs.p;
    return os;
}
void doSomething() {
    Thing aThing(17);
    cout << aThing << endl;
}

int main() {
    doSomething();
    Thing* ptr = new Thing(100);
    delete ptr;
}
```

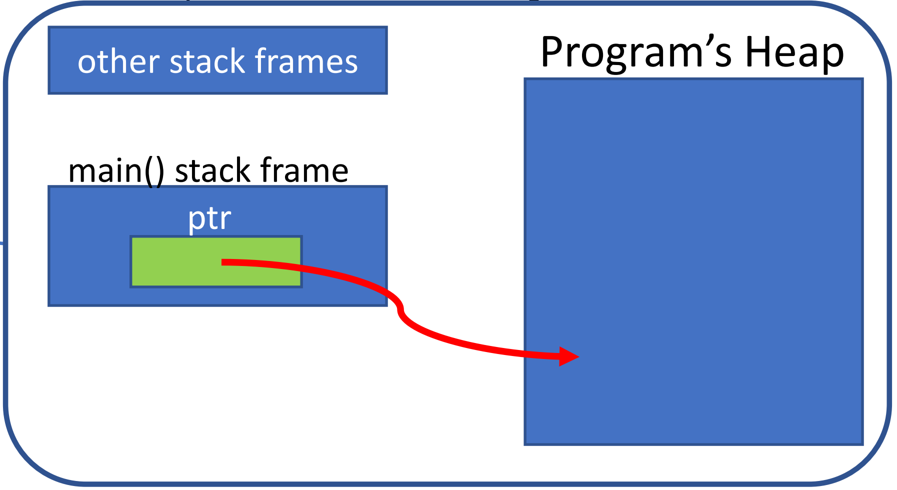
After this statement, ptr is a dangling pointer, but that's okay, we won't use it

- The destructor is called automatically when `delete` is invoked, which
  - Invokes the Thing::~Thing()
  - Deallocates the memory for the Thing object
- The destructor Thing::~Thing() de-allocated the dynamically allocated integer

Memory View after allocating `new Thing(100)`



Memory View after `delete ptr`



other stack frames
Program's Heap
main() stack frame
ptr