CSC230

Intro to C++ Lecture 8

Outline

- Polymorphism
- Memory Management
- Passing Parameters

Polymorphism

Polymorphism:

- What does the polymorphism mean?
- Many forms
    - Overriding is an example

- Usually used in inheritance
    - Call a function, which has different implementations

Polymorphism

- Binding
  - Association between two things;
    - Name (such as function or variable name) and the thing that it names

- <u>Binding time</u> is the time at which a binding takes place

Binding Times

- Possible binding times:
  1. Language design time
     - e.g., bind operator symbols to operations
  2. Language implementation time
     - e.g., bind floating point type to a representation
  3. Compile time
     - e.g., bind a variable to a type in C or Java
  4. Load time
     - e.g., bind a C static variable
  5. Runtime
     - e.g., bind a nonstatic local variable to a memory cell

Static vs Dynamic Binding

- A binding is static(early) if it first occurs at the compile time and remains unchanged throughout program execution.

- A binding is dynamic(late) if it first occurs at the run time or can change during execution of the program.

- Let's take a look at the example

Polymorphism


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

class Polygon{
protected:
  int numVertices;
  float *xCoord, *yCoord;
public:
  void set(){
    cout<<"From Polygon"<< endl;
  }

};

class Rectangle : public Polygon{
public:
  void set(){
    cout<<"From Rectangle"<< endl;
  }
};
```

```
class Triangle : public Polygon{
public:
  void set(){
    cout<<"From Triangle"<< endl;
  }
};

int main(){
  Polygon *poly;
  Rectangle rec;
  Triangle tri;

  poly = &rec;
  poly->set();
  poly = &tri;
  poly->set();
}
```

Polymorphism

The output of the previous file will be:

```
From Polygon
From Polygon
```

By default, C++ checks the type of the variable (poly), call the function in the corresponding type. This is called static resolution/linage.

If we want C++ to check the **contents** of the pointer instead of it's type. We can add "**virtual**" to the function in the base class.

```
class Polygon{
protected:
  int numVertices;
  float *xCoord, *yCoord;
public:
  virtual void set(){
    cout<<"From Polygon"<< endl;
  }
};
```

Virtual Function


```
class Polygon{
protected:
  int numVertices;
  float *xCoord, *yCoord;
public:
  virtual void set(){
    cout<<"From Polygon"<< endl;
  }
};
```

set() is a virtual function

When we call a virtual function, such as set(), C++ uses dynamic linkage/late binding.
The selected function is based on the kind of the object.
After adding "virtual" to the previous program, the result will be:

```
From Rectangle
From Triangle
```

Examples: binding-1.cpp overriding 1-2

Pure Virtual Function


```
class Polygon{
protected:
  int numVertices;
  float *xCoord, *yCoord;
public:
  virtual void set(){
    cout<<"From Polygon"<< endl;
  }
};
```

set() is a virtual function with implementation. It is polygon's child /derived class's option to implement it's own version or not.

```
class Polygon{
protected:
  int numVertices;
  float *xCoord, *yCoord;
public:
  virtual void set()=0;
};
```

set() is a pure virtual function without implementation. Any child/derived class of polygon must implement it.

Interface (Abstract class)

Any class with at least one **pure** virtual function is a abstract class (interface). Abstract class provides an appropriate base class from which other classes can inherit.

- Abstract class cannot instantiate objects
- If a child class of an abstract class does not implement **all** pure virtual functions, itself is an abstract class.
- If a child class of an abstract class does not have any pure virtual function, it is called **concrete class**, which can instantiate objects.

Abstract class vs. concrete class


```
class Box{
  public:
    virtual double getVolume()=0;
  protected:
    double length;
    double breadth;
    double height;
};
```

Abstract class

Box obj;

✗

```
class rectangle : public Box{
  public:
    double getVolume(){
      return length*breadth*height;
    }
};
```

Concrete class

Rectangle obj;

✔

```
class square : public Box{
  public:
    void info(){}
};
```

Abstract class

Square obj;

✗

Outline

- Polymorphism
- Memory Management
- Passing Parameters

Why memory is a big deal?

- Both instructions (code) and data are stored inside memory
- Memory size is limited
  - When you need some data, which should be inside the memory
  - When you do not need the data, reassign the memory
- Data must be accessed in an efficient way (a major topic of this course)

Fixed address memory

- Executable code
- Global variables
- Constant structures that don't fit inside a machine instruction. (constant arrays, strings, floating points, long integers etc.)
- Static variables

Stack memory

- Local variables for functions, whose size can be determined at call time.
- Information saved at function call and restored at function return:
  - Values of callee arguments
  - Register values:
    - Return address
    - Frame pointer
    - Other registers

Heap memory

- Structures whose size varies dynamically (e.g. variable length arrays or strings).
- Structures that are allocated dynamically (e.g. records in a linked list).
- Structures created by a function call that must survive after the call returns.

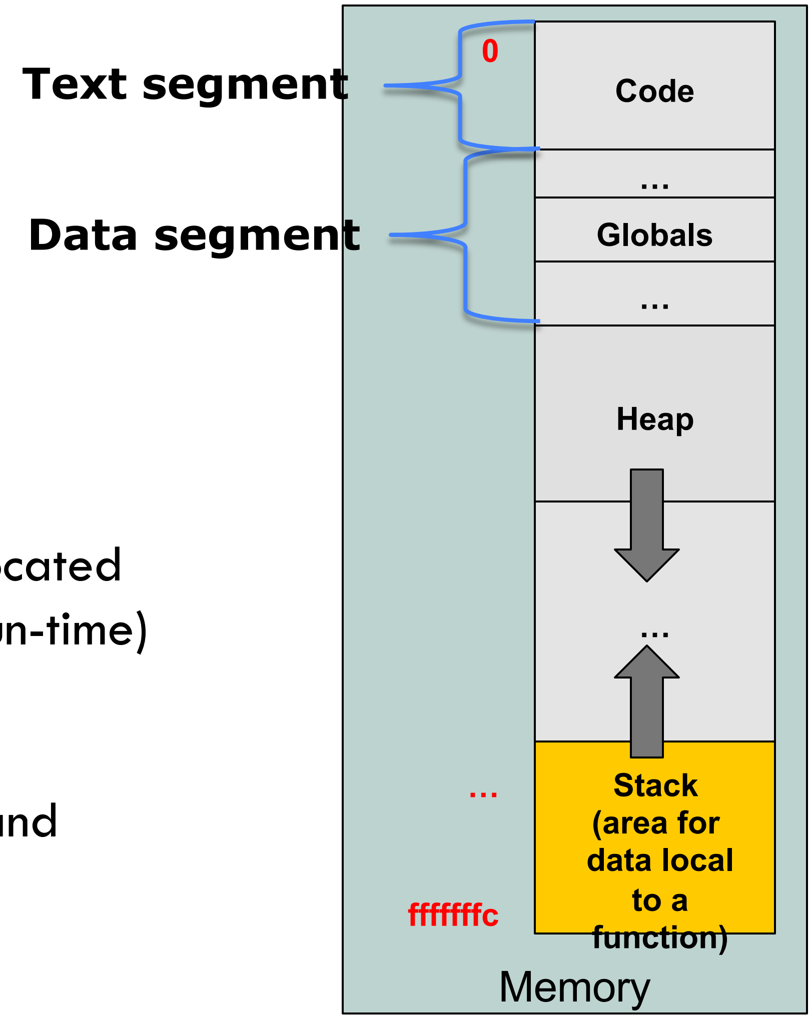
Issues:

- Allocation and free space management
- Deallocation / garbage collection

- Example: Stack 1-3

How a program views memory

- Memory locations have address
- Code is located at low addresses
- Global variables are located after code
- System stack: memory for each function call
  - Parameters, local variables
  - Return address (where to return)
  - etc.
  - **Usually**, NO CODE on the **stack**
- Heap: Memory can be allocated and de-allocated during program execution (dynamically at run-time) based on the needs of the program.
- Heap grows downward, stack grows upward
  - If your program uses up memory, heap and stack collide, program aborts.



Variables and static allocation

Each variable/object in a computer has a:

- Name (such x, used by programmer)
- Address (used by computer to reference it)
- Value
- Scope (the lifetime and visibility to other code)

Automatic/local scope

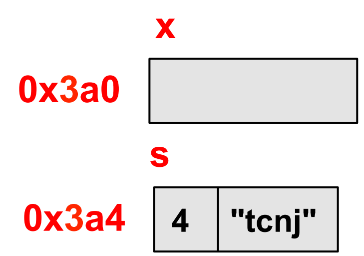
- {…} of a function, loop, or if
- Stored on the stack
- Dies/deallocated when '}' is reached

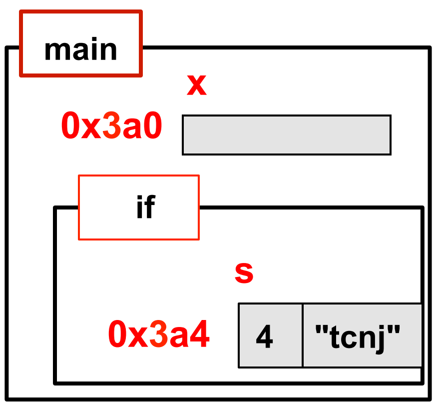
Code

```
int x;
string s("tcnj");
```

```
int main()
{
int x; if( x ){
String s("tcnj");
}
}
```

Computer





Here we use **container box** to represent scope

Local variables

Variables declared inside function or block, {…}, are stored on the stack

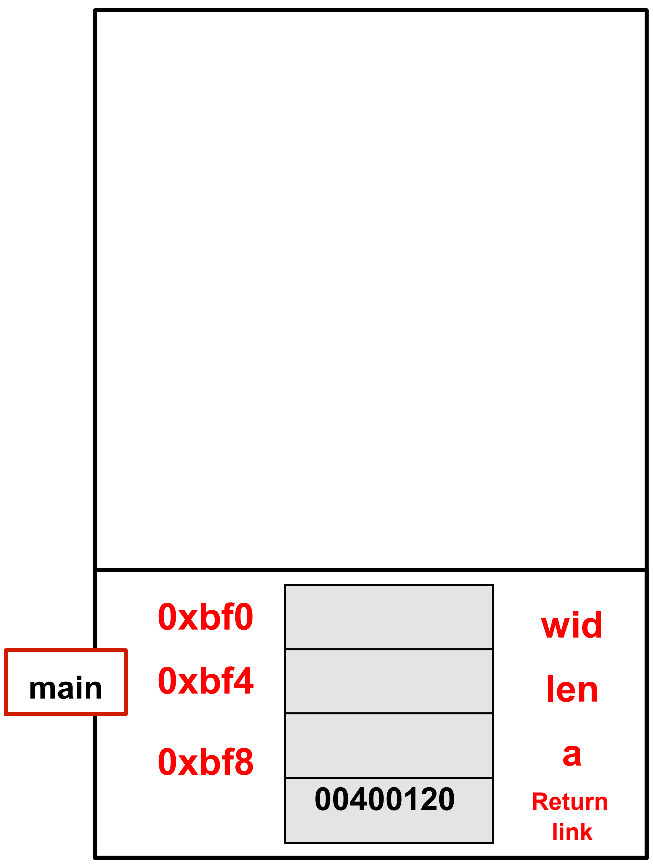
Stack

```
// Computes rectangle area,
//        prints it, & returns it int
 area(int, int);
void print(int);

int main()
{
int wid = 8, len = 5, a;

a = area(wid,len);
}

int area(int w, int l)
{
int ans = w * l; print(ans); return
 ans;
}


void print(int area)
{
cout << "Area is " << area;
}
```

Local variables

- Variables declared inside function or block, {…}, are stored on the stack

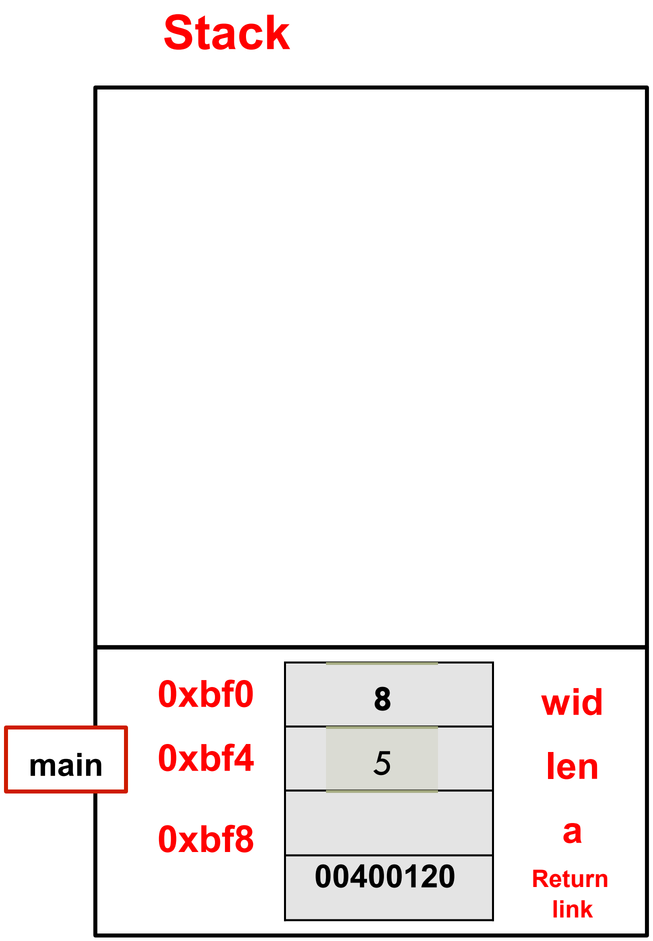
Stack




```
// Computes rectangle area,
//        prints it, & returns it int
 area(int, int);
void print(int);

int main()
{
int wid = 8, len = 5, a;

a = area(wid,len);
}

int area(int w, int l)
{
int ans = w * l; print(ans); return
 ans;
}


void print(int area)
{
cout << "Area is " << area;
}
```

Local variables

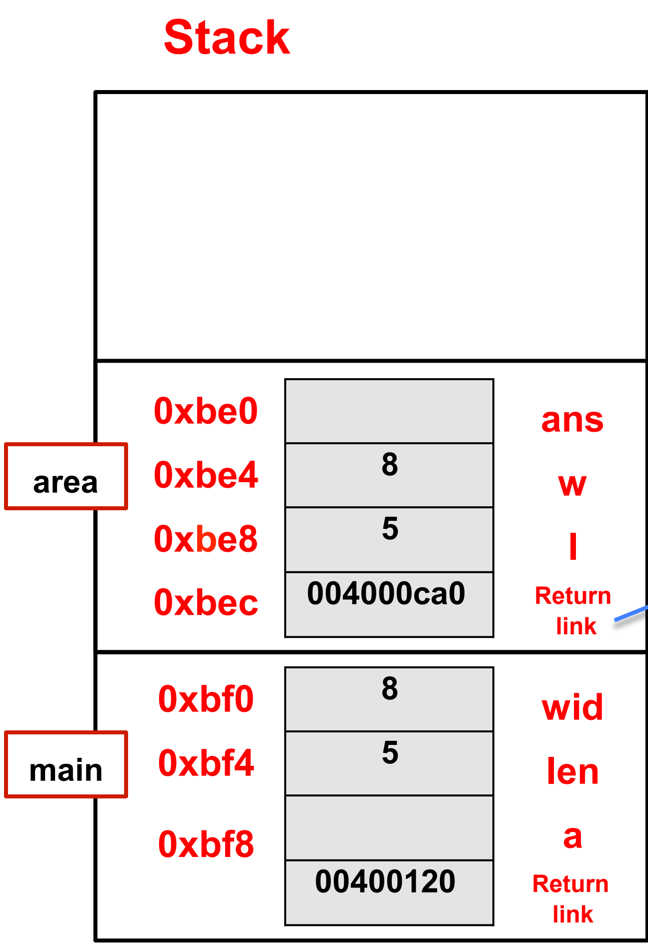
Variables declared inside function or block, {…}, are stored on the stack




```
// Computes rectangle area,
//        prints it, & returns it int
 area(int, int);
void print(int);

int main()
{
int wid = 8, len = 5, a;

a = area(wid,len);
}

int area(int w, int l)
{
int ans = w * l; print(ans); return
 ans;
}


void print(int area)
{
cout << "Area is " << area;
}
```

Local variables

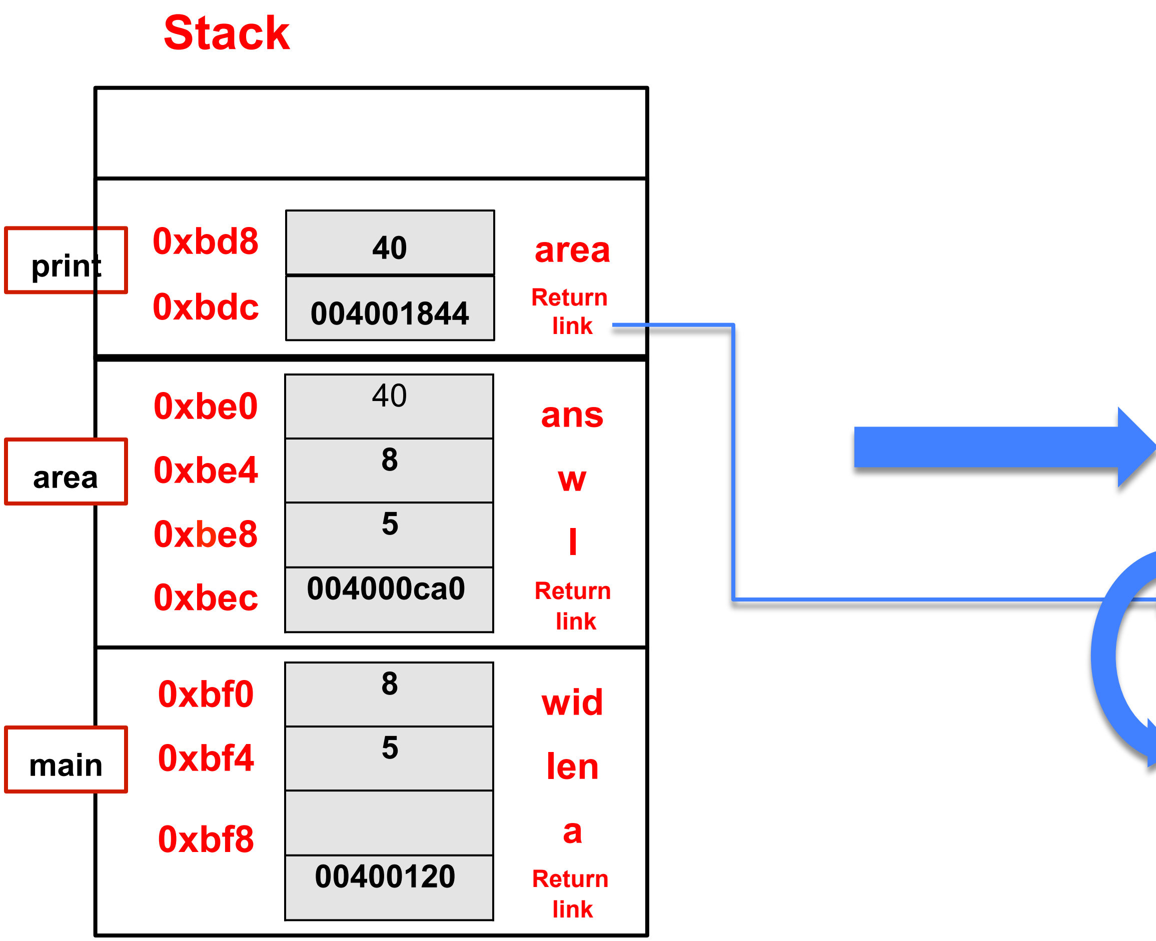
Variables declared inside function or block, {…}, are stored on the stack




```
// Computes rectangle area,
//        prints it, & returns it int
 area(int, int);
void print(int);

int main()
{
int wid = 8, len = 5, a;

a = area(wid,len);
}

int area(int w, int l)
{
int ans = w * l; print(ans); return
 ans;
}


void print(int area)
{
cout << "Area is " << area;
}
```

Local variables

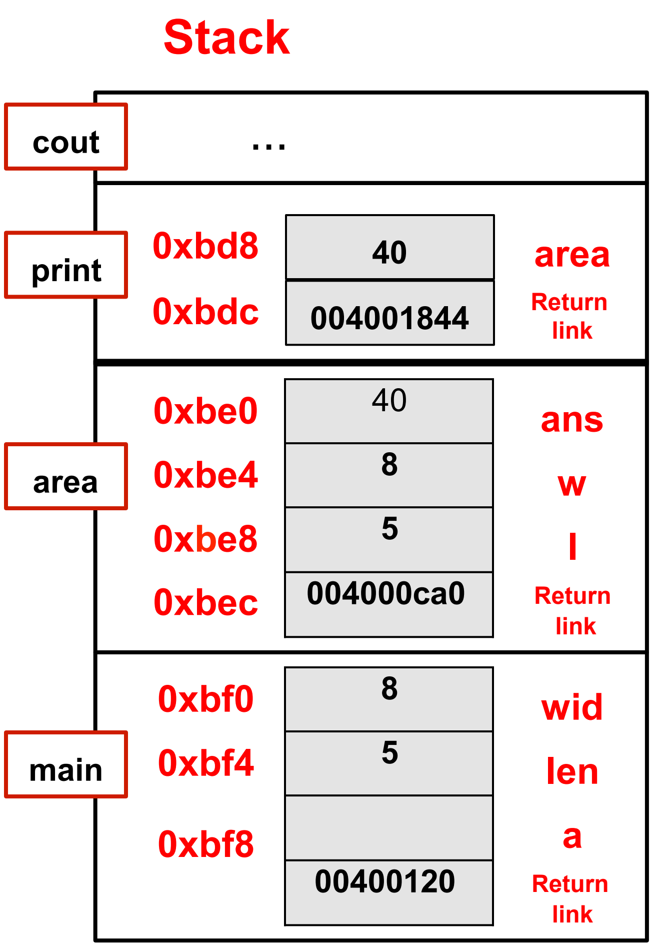
Variables declared inside function or block, {…}, are stored on the stack




```
// Computes rectangle area,
//        prints it, & returns it int
 area(int, int);
void print(int);

int main()
{
int wid = 8, len = 5, a;

a = area(wid,len);
}

int area(int w, int l)
{
int ans = w * l; print(ans); return
 ans;
}


void print(int area)
{
cout << "Area is " << area;
}
```

Local variables

Variables declared inside function or block, {…}, are stored on the stack



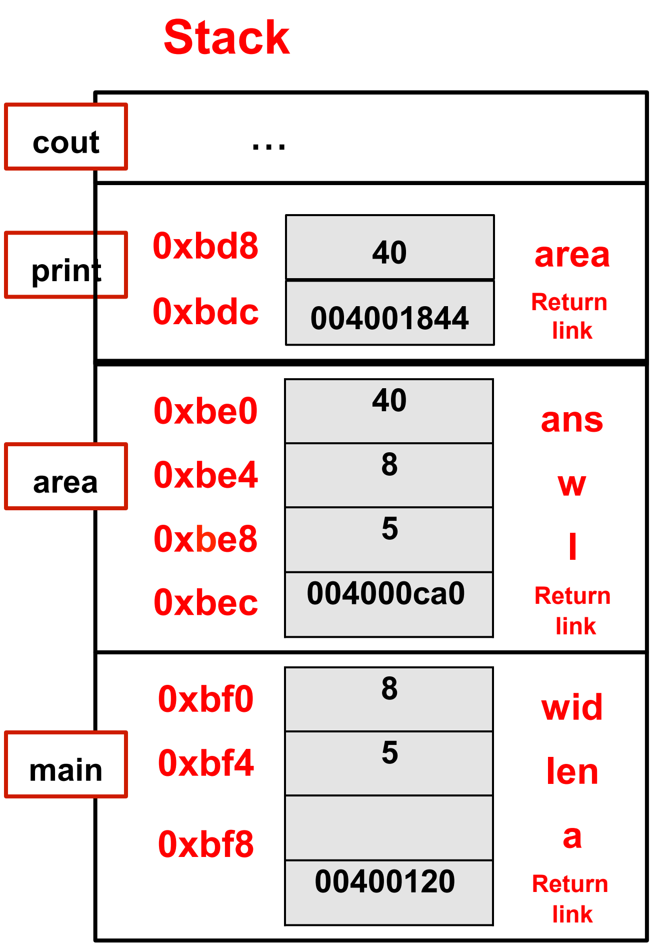

```
// Computes rectangle area,
//        prints it, & returns it int
 area(int, int);
void print(int);

int main()
{
int wid = 8, len = 5, a;

a = area(wid,len);
}

int area(int w, int l)
{
int ans = w * l; print(ans); return
 ans;
}


void print(int area)
{
cout << "Area is " << area;
}
```

004000ca0

004001844

Local variables

Variables declared inside function or block, {…}, are stored on the stack

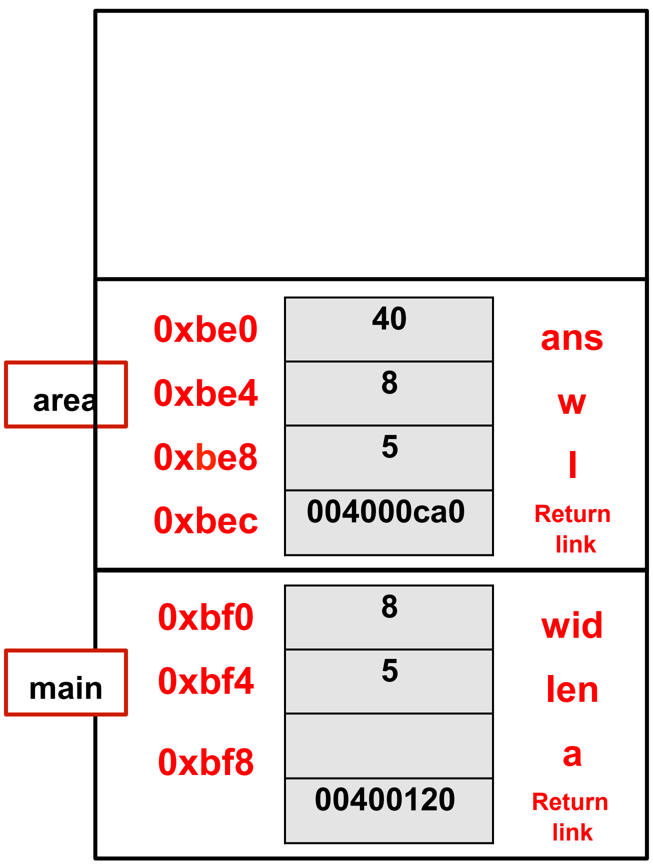



```
// Computes rectangle area,
//        prints it, & returns it int
 area(int, int);
void print(int);

int main()
{
int wid = 8, len = 5, a;

a = area(wid,len);
}

int area(int w, int l)
{
int ans = w * l; print(ans); return
 ans;
}


void print(int area)
{
cout << "Area is " << area;
}
```

Local variables

Variables declared inside function or block, {…}, are stored on the stack

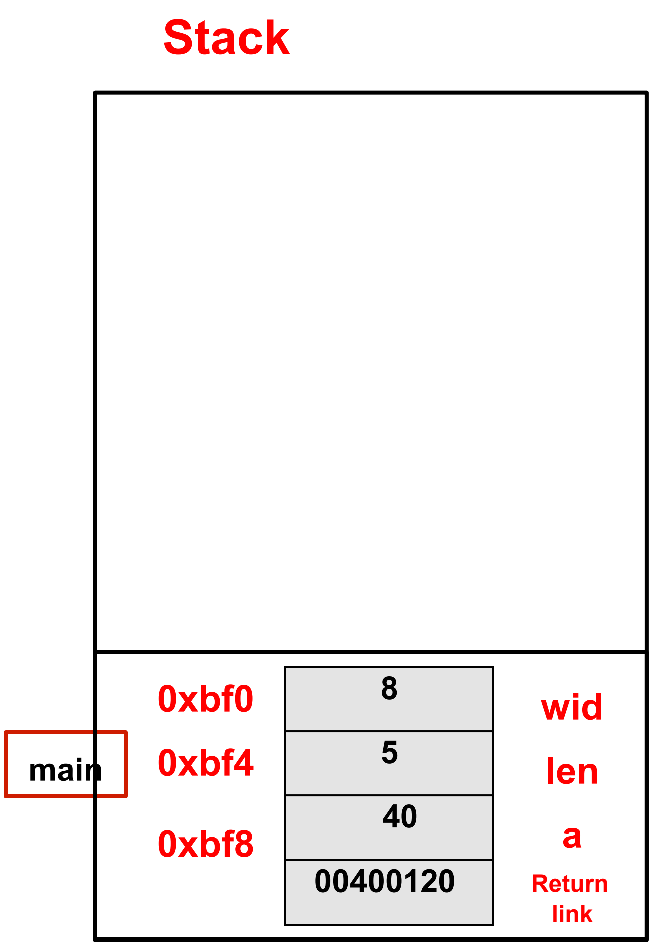
Stack




```
// Computes rectangle area,
//        prints it, & returns it int
 area(int, int);
void print(int);

int main()
{
int wid = 8, len = 5, a;

a = area(wid,len);
}

int area(int w, int l)
{
int ans = w * l; print(ans); return
 ans;
}


void print(int area)
{
cout << "Area is " << area;
}
```

Local variables

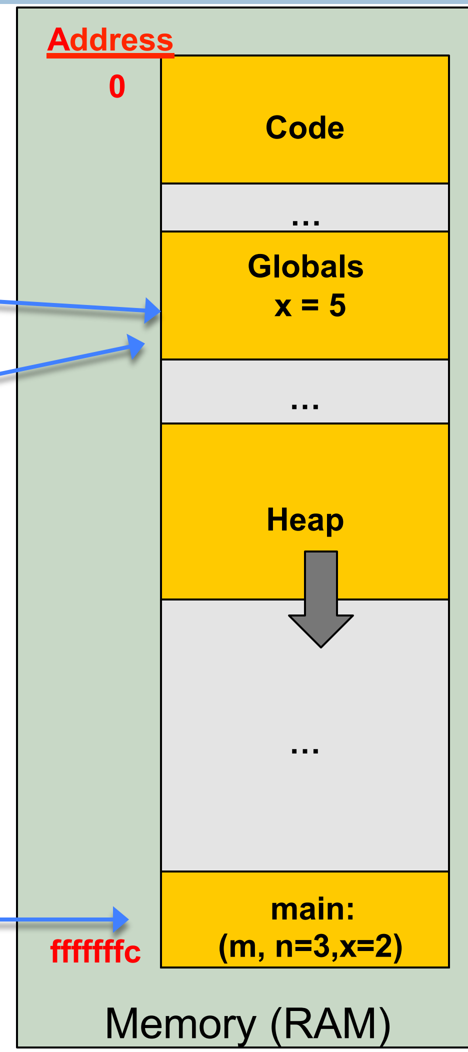
Variables declared inside function or block, {…}, are stored on the stack




```
// Computes rectangle area,
//        prints it, & returns it int
 area(int, int);
void print(int);

int main()
{
int wid = 8, len = 5, a;

a = area(wid,len);
}

int area(int w, int l)
{
int ans = w * l; print(ans); return
 ans;
}


void print(int area)
{
cout << "Area is " << area;
}
```

Scope example


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

int x = 5;

int main() {
  int m, n = 3, x = 2;

  cout << n << "\t" << x << endl;
  cout << ::x << endl;
}
```



- Local variables
  - Defined inside a function or block {…}
  - Used inside the same function or block
- Global variables
  - Defined outside any function
  - Used by **all** functions
- When variables share the same name, the closest declaration will be used.

Outline

- Polymorphism
- Memory Management
- Passing Parameters

Review: Pointer & Reference

Pointer

- Memory address of a variable
- &obj returns the address of obj
- *ptr returns the object at address given by ptr
- *(&obj) returns obj

NULL

- Pointer value points nowhere
- Is 0. So, we can have
    - int * p = NULL;
    - if(p)
- Defined in <cstdlib>

Pointer & Reference

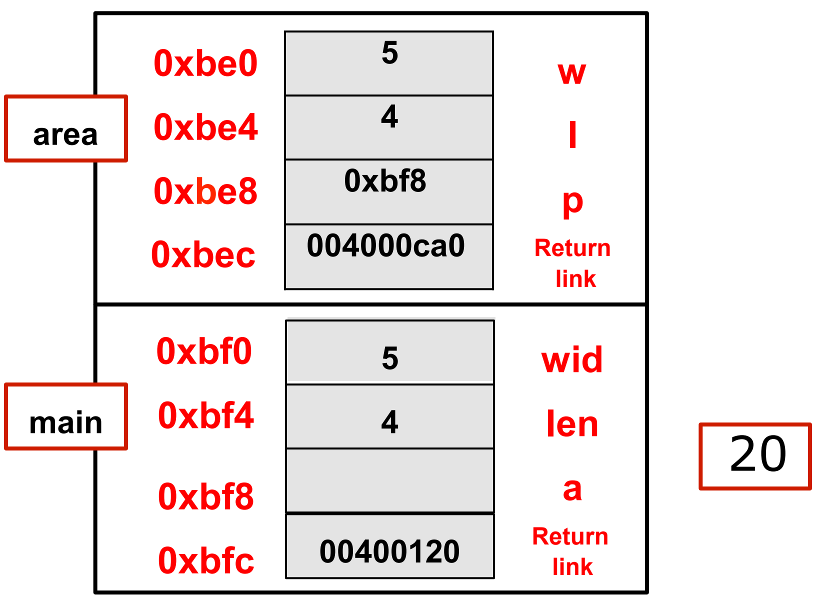
Reference

- Is an **alias** to an existing variable
- Must be **initialized** when it is declared
  - Logically, reference does not consume memory, it is just another name (alias) of some variable.
  - Physically, it may be implemented by pointer.

Use pointers correctly

One function can use pointer to modify the variable in a different function.

Stack Area of RAM

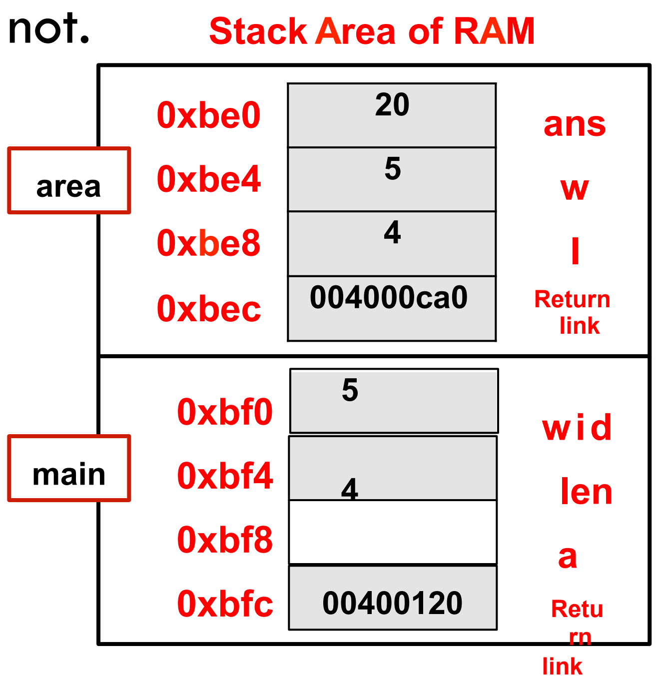



```
int area(int, int, int*);


int main()
{
int wid = 5, len = 4, a;
 area(wid,len,&a);
}


void area(int w, int l, int* p)
{
*p = w * l;
}
```

Misuse of pointers

Make sure you do not let the pointer pointing to a **wrong address** or a **dead variable.**

You may be lucky to get the value, maybe not.




```
int * area(int, int);

int main()
{
  int wid = 5, len = 4, *a;
   a = area(wid,len);
  cout << *a << endl;
}

int* area(int w, int l)
{
  int ans = w * l;

  return &ans;
}
```

Use Reference, as a variable

Reference is an alias for an existing variable.

Variable “r” is alias for variable “x”.

- Here “x” and “r” are **labels** used by human being
- “x” and “r” themselves do **not** take any **memory**
- Compiler and linker will **map** “x” and “r” to a memory address

```
int x = 10;
int &r = x;
```

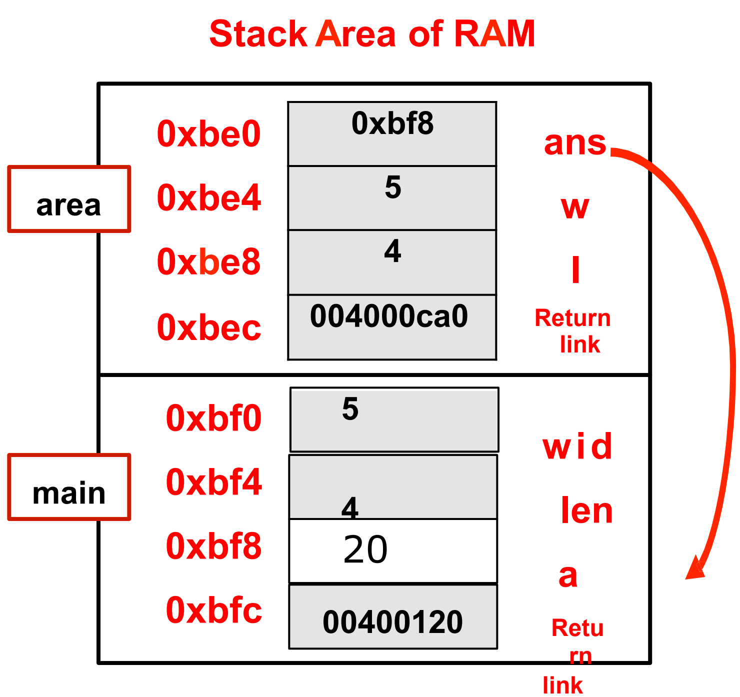
x:
r: 10

```
x == r
&x == &r
```

Use Reference, as a parameter

Reference is an alias for an existing variable.

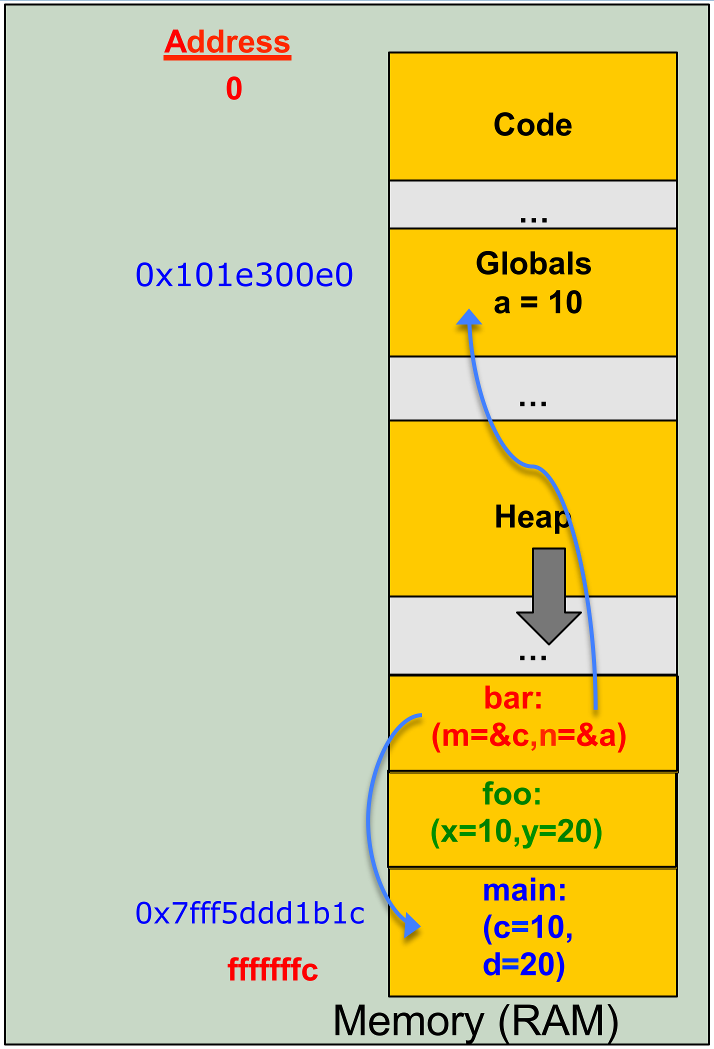
Variable “ans” is an alias for variable “a” in the main function.




```
int  area(int, int, int&);

int main()
{
  int wid = 5, len = 4, a;
   a = area(wid,len, a);
  cout << a << endl;
}

void area(int w, int l, int &ans)
{
  ans = w * l;
}
```

Parameters


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

int a=10;

void foo(int x, int y){
  cout << x << "\t" << y << endl;
}

void bar(int *m, int &n){
  *m = 100;
  n = 200;
  cout << &n << endl;
}

int main(){
  int c = 10, d =20;

  foo(c, d);
  bar(&c, a);
}
```



Parameters


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

int a=10;

void foo(int x, int y){
  cout << x << "\t" << y << endl;
}

void bar(int *m, int &n){
  *m = 100;
  n = 200;
  cout << &n << endl;
}

int main(){
  int c = 10, d =20;

  foo(c, d);
  bar(&c, a);
}
```

Pass by value:

By default, arguments in C++ are passed by value. When an argument is passed by value, the argument's value is **copied** into the function's parameter. When foo() is called, variable x and y are created, and values of c and d are copied to x and y.

Parameters


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

int a=10;

void foo(int x, int y){
  cout << x << "\t" << y << endl;
}

void bar(int *m, int &n){
  *m = 100;
  n = 200;
  cout << &n << endl;
}

int main(){
  int c = 10, d =20;

  foo(c, d);
  bar(&c, a);
}
```

Pass by pointer:
When bar() is called, variable m is created, and the **address** of c is copied to m.

Parameters


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

int a=10;

void foo(int x, int y){
  cout << x << "\t" << y << endl;
}

void bar(int *m, int &n){
  *m = 100;
  n = 200;
  cout << &n << endl;
}

int main(){
  int c = 10, d =20;

  foo(c, d);
  bar(&c, a);
}
```

Examples : pass-1-3

Pass by reference:
When bar() is called, variable n is created, and n becomes **alias** to a. In fact, the **address** of a is copied to n.

When to use pass by value, pass by pointers, pass by reference?

Pass by value:

- Do not want to modify the parameter.
- It is easy to copy (int, double, std::string, std::vector, etc.)

Pass by pointer:

- Expensive to copy the data
- NULL can be the address value (optional parameters)

Pass by reference:

- Expensive to copy the data
- NULL cannot be the address value