deep se
dependable evolvable pervasive software engineering group

Pointers, References and Functions Parameters

Danilo Ardagna

Politecnico di Milano
danilo.ardagna@polimi.it

POLITECNICO DI MILANO

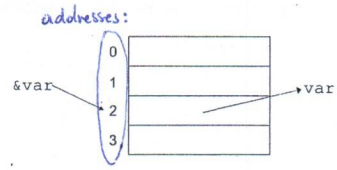
Pointers

- Declaring a variable means reserving a memory area including several locations
- The number of locations depends on the type of data (e.g., integer 2 bytes, float 4 bytes, ...)
- Each memory location has a physical address and:
  - The name of the variable indicates the contents of the memory location
  - The operator `&` allows obtaining the memory address of the location associated with the variable to which the operator is applied

Variables and Memory

addresses:

```
int var;
```



- Suppose the declaration reserves the memory area at address 2
- `var` indicates the content of the memory location
- `&var` indicates the memory address

Pointers

- Pointer variables store memory addresses
  - A pointer `p` can hold the address of a memory location
  - Think of them as a kind of integer values
- When declaring a pointer you must also specify what type of object the pointer points to
- Syntax:

```
double *p;
```

- A pointer variable usually requires 2 bytes or 4 bytes depending on the architecture

Dereferencing a pointer

- You can apply the dereferencing * operator to a pointer variable
- `*p` indicates the content of the location pointed by `p`
- If `p` is a pointer to an integer then `*p` is a simple integer variable

```
int * p;        = pointer to an integer
// some code

*p = 5;  // OK. * p is an integer
p = 5;   // error, p is a pointer   (and 5 is not an address)
```

- Warning! The symbol * is used both in the declaration and in the dereferencing

!

*p ⇒ we're accessing the memory location whose address is stored in p

p = address
*p = integer (memory location (of an integer) whose address is stored in p)

on compiler:

```
int x;
int *p;
x = 5;
p = &x;
```

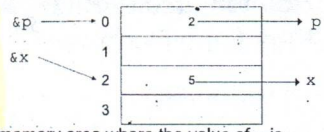
```
x = 5
p = 0x6ffe0c
*p = 5
&x = 0x6ffe0c
&p = 0x6ffe00
```


Operations

- We can store in a pointer variable the address of another variable

```
int x;
int *p;

x=5;
p=&x; // *p is 5
```



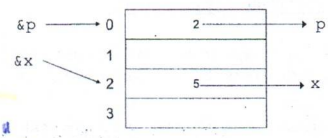
- p will point to the memory area where the value of x is stored
- A pointer's type determines how the memory referred to by the pointer's value is used
  - e.g. what a `int*` points to can be added but not, say, concatenated

Operations

- What happens if we perform the assignment?

```
*p=7;
```

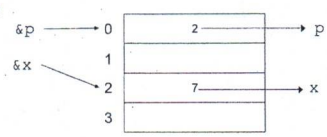
in this moment we're saying: "go where p is pointing and assign '7', but also x changes!"



Operations

- What happens if we perform the assignment?

```
*p=7; // x get 7!
```



This is a little bit different when we declare or define

Example:

```
int main(){
  int x=3;
  int y=5;
  int *p=&x;
  *p=7;
  p=&y;
  *p+=3;
  cout<<x<<y<<*p<<endl;
  return 0;
}
```

(same)

print: 7 8 8

Function Parameters

Recap: Why functions?

- Chop a program into manageable pieces
  - "divide and conquer"
- Match our understanding of the problem domain
  - Name logical operations
  - A function should do one thing well
- Functions make the program easier to read
- A function can be useful in many places in a program
- Ease testing, distribution of labor, and maintenance
- Keep functions small
  - Easier to understand, specify, and debug

Functions

- General form:
  - return_type name (formal arguments); // a declaration
  - return_type name (formal arguments) body // a definition
  - For example:

```
double f(int a, double d);
double f(int a, double d) { return a*d; }
```

- Formal arguments are often called parameters
- If you don't want to return a value give void as the return type

```
void increase_power_to(int level);
```

Here, void means "doesn't return a value"

- A body is a block
  - { /* code */ } // a block

Function Parameters

```
double circ(double radius) {
  float res;
  res = radius * 3.14 * 2;
  radius = 7; // No sense instruction, let's see what happens to radius
  return res;
}

// somewhere in the main
double c;
double r=5;

c= circ(r);
```


Formal and Actual parameters

- In a function definition we use **formal parameters representing a symbolic reference** (identifiers) to objects used within the function
  - `radius` is a formal parameter
  - They are used by the function as if they were local variables
- The initial value of formal parameters is defined when the function is called using the **actual parameters** specified by the caller
  - `r` in our running example

Parameters Passing

- In a function call, parameters passing consists in associating the actual parameters with the formal parameters
- If the function prototype is:

```
double circ (double radius);
```

- If we write:

```
c = circ (5.0);
```

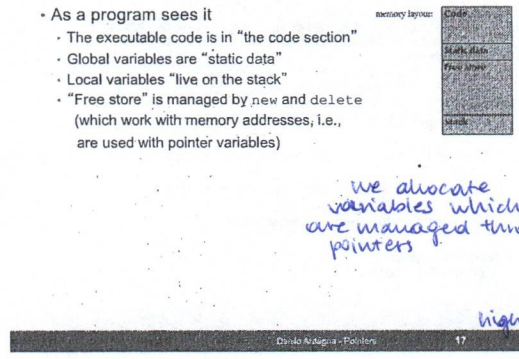
- `radius` (the formal parameter) will assume the value 5.0 (the actual parameter) for that particular invocation
- The exchange of information with the passing of parameters between caller and callee can take place in two ways:
  - Pass by value
  - Pass by reference

1. Pass by value

- At the time of the function call, the value of the actual parameter is **copied** into the memory location of the corresponding formal parameter. In other words, the formal parameter and the actual parameter refer to two different memory locations
- The function works in its **environment** and therefore on formal parameters
- The actual parameters are not changed

The computer' s memory

- As a program sees it
  - The executable code is in "the code section"
  - Global variables are "static data"
  - Local variables "live on the stack"
  - "Free store" is managed by `new` and `delete` (which work with memory addresses, i.e., are used with pointer variables)



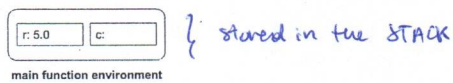
Pass by value example

```
double circ(double radius) {
  float res;
  res = radius * 3.14 *2;
  radius = 7; /* No sense instruction, let's
    see what happens to radius */
  return res;
}

// somewhere in the main
double c;
double r=5;

c= circ(r);
// r is still 5.0
```

- Functions environment are stored in the Stack
- The same mechanism is used to store block variables



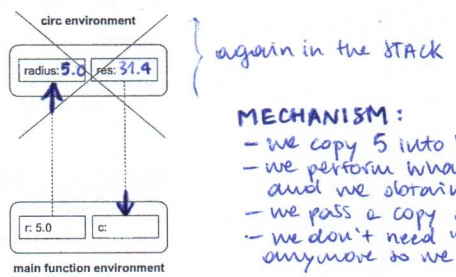
Pass by value example

```
double circ(double radius) {
  float res;
  res = radius * 3.14 *2;
  radius = 7; /* No sense instruction, let's
    see what happens to radius */
  return res;
}

// somewhere in the main
double c;
double r=5;

c= circ(r);
// r is still 5.0
```

- Functions environment are stored in the Stack
- The same mechanism is used to store block variables



MECHANISM:
- we copy 5 into "circ environment"
- we perform what we have to and we obtain 31.4
- we pass a copy of 31.4
- we don't need "circ environment" anymore so we delete it

2. Pass by reference

- At the time of the call the address of an actual parameters is associated with the formal parameters
  - In other words, the actual parameter and the formal parameter **share the same memory location**
- The running function works in its environment on the formal parameters (and consequently also on the actual parameters) and **each change** on the **formal parameter** is **reflected** to the corresponding **actual parameter**
- The function execution effects the caller with modifications to its the caller environment
  - In this way we can return multiple results!

a change in the function environment will reflect the variables outside the function environment

Parameters passing in C

- In C there is no syntax mechanism to distinguish between parameters passing by value and by address
- Parameters passing is always by value:

```
double circ(double radius);
                /*pass by value*/
```

- To implement pass by reference we need to rely on pointers

```
double circ(double *radius );
                /*pass by reference*/
```

Pass by reference

- We need:
  - A pointer for each formal parameter
  - The dereference operator in the function body to access the actual parameter
  - In the function call the address of the actual parameter is used
- Warning! Arrays are always passed by reference
  - The name of an array variable, is an address, i.e., it is a pointer!
  - Why?
    - Efficiency!

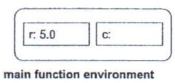
Pass by reference example

```
double circ(double *radius) {
  double res;
  res = *radius * 3.14 *2;
  *radius = 7; /* No sense instruction, let's see
    what happens to radius */
  return res;
}

// somewhere in the main
double c;
double r=5;

c=circ(&r);
//Warning! Now r is 7.0
```

this means we've expecting a pointer, (= an address)



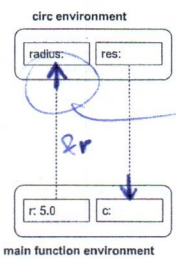
main function environment

Pass by reference example

```
double circ(double *radius) {
  double res;
  res = *radius * 3.14 *2;
  *radius = 7; /* No sense instruction, let's see
    what happens to radius */
  return res;
}

// somewhere in the main
double c;
double r=5;

c=circ(&r);
//Warning! Now r is 7.0
```



main function environment

here we're copying the address of r

Now, inside the function we've modifying the variables outside the function environment !!

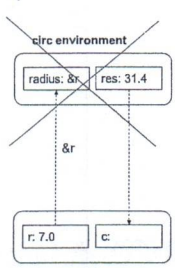
Pass by reference example

```
double circ(double *radius) {
  double res;
  res = *radius * 3.14 *2;
  *radius = 7; /* No sense instruction, let's see
    what happens to radius */
  return res;
}

// somewhere in the main
double c;
double r=5;

c=circ(&r);
//Warning! Now r is 7.0
```

What about circ(22.0);?



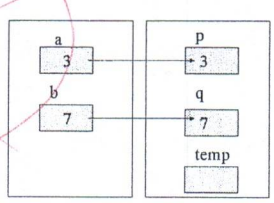
main function environment

What if we write circ(22.0)? syntax error because we expect a pointer (an address) and we get a float !

Example: swap of 2 ints

```
void swap (int p, int q) {
    int temp;
    temp = p;
    p = q;
    q = temp;
}
```

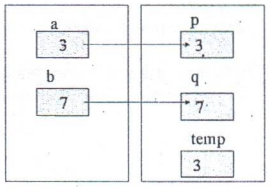
In main: `swap(a,b)`



WHY (↓)

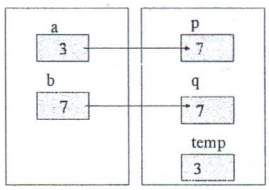
Example: swap of 2 ints

```
void swap (int p, int q) {
     int temp;
     temp = p;
     p = q;
     q = temp;
}
```
In main: `swap(a,b)`



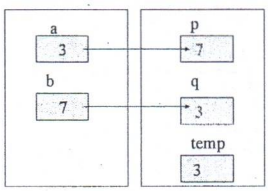
Example: swap of 2 ints

```
void swap (int p, int q) {
     int temp;
     temp = p;
     p = q;
     q = temp;
}
```
In main: `swap(a,b)`



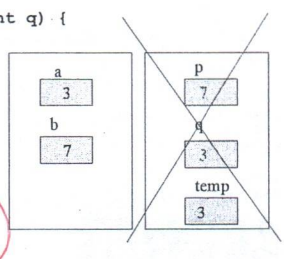
Example: swap of 2 ints

```
void swap (int p, int q) {
     int temp;
     temp = p;
     p = q;
     q = temp;
}
```
In main: `swap(a,b)`



Example: swap of 2 ints

```
void swap (int p, int q) {
     int temp;
     temp = p;
     p = q;
     q = temp;
}
```
In main: `swap(a,b)`

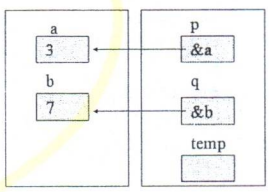


! At the end of swap main variables are **unchanged**!

The pass-by-value mechanism destroyes the swapping (we swapped p and q and then we deleted them)

Example: swap of 2 ints

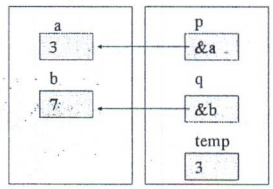
```
void swap (int *p, int *q){
     int temp;
     temp = *p;
     *p = *q;
     *q = temp;
}
```
In main: `swap(&a, &b)`



Example: swap of 2 ints

```
void swap (int *p, int *q){
      int temp;
      temp = *p;
      *p = *q;
      *q = temp;
}
```

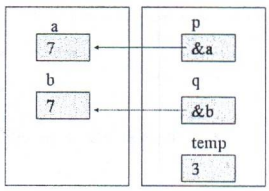
In main: `swap(&a, &b)`



Example: swap of 2 ints

```
void swap (int *p, int *q){
      int temp;
      temp = *p;
      *p = *q;
      *q = temp;
}
```

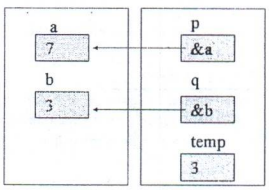
In main: `swap(&a, &b)`



Example: swap of 2 ints

```
void swap (int *p, int *q){
      int temp;
      temp = *p;
      *p = *q;
      *q = temp;
}
```

In main: `swap(&a, &b)`

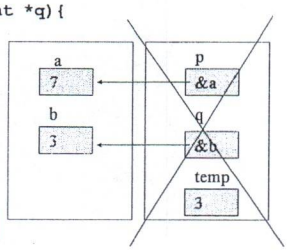


Example: swap of 2 ints

```
void swap (int *p, int *q){
      int temp;
      temp = *p;
      *p = *q;
      *q = temp;
}
```

In main: `swap(&a, &b)`

At the end of swap main variables are **changed!**




```
1
2
3
4
5  #include <iostream>
6
7  void cube( int * nPtr );   /* prototype */
8
9  void main()
10 {
11    int number = 5;
12
13    std::cout << "The original value of number is " << number;
14    cube( &number );
15    printf( "\nThe new value of number is %d\n", number );
16
17
18 }
19
20 void cube( int *nPtr )
21 {
22    *nPtr = *nPtr * *nPtr * *nPtr;
23 }
```

cube requires a pointer!!

In cube, *nPtr is used (*nPtr is number!)

```
The original value of number is 5
The new value of number is 125
```

Void functions and functions returning a value interchange

→ function:

```
int f(int par1)
{
… (compute integer result") …
return result;
}
```

call:

```
y=f(x);
```

→ void function:

```
void f(int par1,int *par2)
{
… (compute integer result) …
*par2=result;
}
```

call:

```
f(x,&y);
```

y gets the proper result value

If we need to return multiple values

```
void f(int in_par1, …, int in_parn, int *out_par1, int *out_parn.)
{
… (compute integer results) …
*out_par1=result1;
…
*out_parn=resultn;
}
```

Call (one input parameter, two output parameters):

```
f(x,&y,&z);
```

y, z get the proper result values

Pass by value and pass by reference comparison

- Pass by value:
  - Requires a lot of time to perform the copy if the parameter is large
  - Actual parameter and formal parameter are different
  - Cannot return a value to the caller
- Pass by reference:
  - An address is copied ⇒ fixed size ⇒ fast!
  - Actual parameter and formal parameter are the same
  - Can return a value to the caller

Summary

Property \ Pass	by value	by reference
Time and Space	Large	Small
Side effects risk	No	Yes
Return value to the caller	No	Yes

If a function erroneously modify a variable, the modification remains also outside the function environment

References

How to make parameter passing a bit easier

If we introduce a reference for an object we're introducing another name for the same object

References

- An automatically dereferenced pointer:
  - Or as "an alternative name for an object"
  - A reference is introduced though the & modifier in a variable declaration
  - A reference **must be** initialized
  - The value of a reference **cannot be changed after initialization**
    - I.e., you cannot make a reference refer to another object after initialization

```
int x = 9;
int y = 8;

int &r = x;
r = 10; // x = 10
r = &y;    // error (and so is all other attempts to change what r
           // refers to)
```

in this way r is not an integer, is a reference to an integer. In this moment r and x ARE THE SAME THING

WHENEVER WE DEFINE A REFERENCE WE DON'T CHANGE !

References

- An automatically dereferenced pointer:
  - Or as "an alternative name for an object"
  - A reference is introduced though the & modifier in a variable declaration
  - A reference **must be** initialized
  - The value of a reference **cannot be changed after initialization**
    - I.e., you cannot make a reference refer to another object after initialization

```
int x = 9;
int y = 8;

int &r = x;
r = 10; // x = 10!
r = &y;    // error (and so is all other attempts to change what r
           // refers to)
```

References

- When we initialize a variable, the value of the initializer is copied into the object we are creating
- When we define a reference, instead of copying the initializer value, we **bind the reference to its initializer**
- Once initialized, a reference remains bound to its initial object
  - **There is no way to rebind a reference to refer to a different object**
  - Because of this, references **must be initialized**
- When we fetch the value of a reference:
  - We are really fetching the value of the object to which the reference is bound
- When we use a reference as an initializer:
  - We are really using the object to which the reference is bound

```
int i = 7;
int& r = i;
// ok: r3 is bound to the object to which r is bound, i.e., to i
int &r3 = r;
// initializes j from the value in the object to which r is bound
int j = r;  // ok: initializes j to the same value as i
```


Pointers and references

- A pointer is a compound type that "points to" another type
- Like references, pointers are used for indirect access to other objects
- Unlike a reference, a **pointer is an object** in its own right
  - **Pointers can be assigned and copied; a single pointer can point to several different objects over its lifetime**
  - **Unlike a reference, a pointer does not need to be initialized at the time it is defined**
- Like other built-in types, pointers defined at block scope have undefined value if they are not initialized. **Be very careful !!!**

Pointers and references

- & and *, are used as both an operator in an expression and as part of a declaration
- The **context** in which a symbol is used **determines** what the symbol **means**

```
int i = 42;
int &r = i;   //& follows a type and is part of a declaration; r is a
              // reference
int *p;       // * follows a type and is part of a declaration; p is a
              // pointer
p = &i;       // & is used in an expression as the address-of operator

*p = i;       // * is used in an expression as the dereference
              // operator

int &r2 = *p; // & is part of the declaration; * is the dereference
              // operator
```

`const` Qualifier

`const` Qualifier

- We might want to define a variable whose value we know cannot be changed
  - For example, to refer to the size of a buffer size
- We can make a variable unchangeable by defining the variable's type as **const**:

```
const int bufSize = 512; // input buffer size
                         //same as constexpr int bufsize=512
bufSize = 512;           // error: attempt to write to const object
```

- **Because we can't change the value of a const object after created, it must be initialized**

```
const int j = 42; // ok: initialized at compile time
const int i = get_size(); // ok: initialized at run time
const int k; // error: k is uninitialized const

j= 47; // error: we try to change a const variable
```


`const` Qualifier

- By default, const objects are local to a file
- When a const object is initialized from a compile-time constant, our compiler will usually replace uses of the variable with its corresponding value during compilation
  - The compiler will generate code using the value 512 in the places that our code uses bufSize

References to `const`

- We can bind a reference to an object of a const type
- To do so we use a **reference to const**, which is a reference that refers to a const type
- Unlike an ordinary reference, a reference to const cannot be used to change the object to which the reference is bound

```
const int ci = 1024;
const int &r1 = ci;// ok: both reference and underlying
                   // object are const
r1 = 42;           // error: r1 is a reference to const

int &r2 = ci;      // error: non const reference to
                   // a const object (same error as trying to
                   //change j in the previous slide!)
```

Note: by doing:

```
int x = 2;
const int &r = x;
```

we have a constant reference to an integer not constant.
if we do:

```
r = 4;
```

ERROR

because r is constant,
if we do;

```
x = 4;
```

OK

because x is not a constant.
Moreover, after x = 4 r becomes 4.

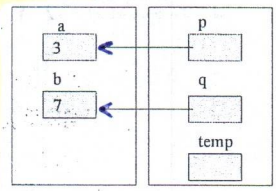
C++ pass by reference

- C++ relies on references to implement pass by reference parameters passing mechanism
- This simplifies a lot syntax and notation
- **Const references** can be used to pass **large objects** in **read only** (obtaining the same benefits of C arrays passing **+ read only protection**, i.e., **no side effects**)

Example: swap of 2 ints

```
void swap (int &p, int &q){
      int temp;
      temp = p;
      p = q;
      q = temp;
}
```

In main: `swap(a, b)`

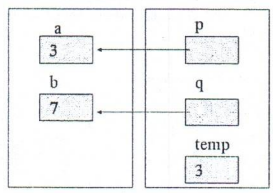


We're now considering references! It's actually easier than pointers. Now p and a are the same object (since p is the reference of a)

Example: swap of 2 ints

```
void swap (int &p, int &q){
      int temp;
➡     temp = p;
      p = q;
      q = temp;
}
```

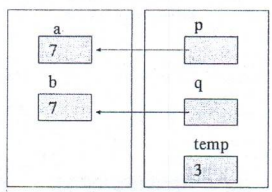
In main: `swap(a, b)`



Example: swap of 2 ints

```
void swap (int &p, int &q){
      int temp;
      temp = p;
➡     p = q;
      q = temp;
}
```

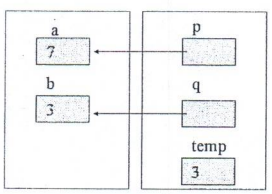
In main: `swap(a, b)`



Example: swap of 2 ints

```
void swap (int &p, int &q){
      int temp;
      temp = p;
      p = q;
➡     q = temp;
}
```

In main: `swap(a, b)`

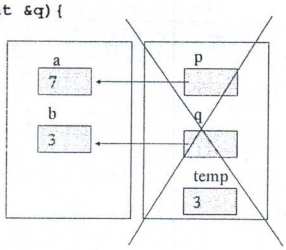


Example: swap of 2 ints

```
void swap (int &p, int &q){
      int temp;
      temp = p;
      p = q;
➡     q = temp;
}
```

In main: `swap(a, b)`

! At the end of swap main variables are **changed**!



Pass by reference example

```
double circ(double *radius) {
  double res;
  res = *radius * 3.14 *2;
  *radius = 7; /* No sense instruction,
        let's see what happens to radius */
  return res;
}

// somewhere in the main
double c;
double r=5;

c=circ(&r);
//Warning! Now r is 7.0
```

pointers

```
double circ(const double &radius) {
  double res;
  res = radius * 3.14 *2;
  radius = 7; /* No sense instruction,
        let's see what happens to radius */
  return res;
}

// somewhere in the main
double c;
double r=5;

c=circ(r);
//r is 5.0
```

references

we're introducing a constant reference (which means that we cannot modify what we're bringing in the function)

ERROR!

Pass by reference example

```
double circ(double *radius) {
  double res;
  res = *radius * 3.14 *2;
  *radius = 7; /* No sense instruction,
        let's see what happens to radius */
  return res;
}

// somewhere in the main
double c;
double r=5;

c=circ(&r);
//Warning! Now r is 7.0
```

```
double circ(const double &radius) {
  double res;
  res = radius * 3.14 *2;
  radius = 7; /* No sense instruction,
        let's see what happens to radius */
  return res;
}

// somewh
double c;
double r=

c=circ(r);
//r is 5.0
```

We get a compiler error here and we need to delete this line of code

Pass by reference example

```
double circ(double *radius) {
  double res;
  res = *radius * 3.14 *2;
  *radius = 7; /* No sense instruction,
        let's see what happens to radius */
  return res;
}

// somewhere in the main
double c;
double r=5;

c=circ(&r);
//Warning! Now r is 7.0
```

```
double circ(const double &radius) {
  double res;
  res = radius * 3.14 *2;
  return res;
}

// somewhere in the main
double c;
double r=5;

c=circ(r);
//r is 5.0
```

What about circ(22.0);?

A temporary const object is created, initialized with 22.0 and the function can be executed

✓

Call by Value vs. by Call Reference

Play video

Guidance for passing variables

- Use call-by-value for very small objects (base types!)
- Use call-by-const-reference for large objects
- Use call-by-reference only when you must return a result rather than modify an object through a reference argument

- For example:

```
class Image { /* objects are potentially huge */ };
void f(Image i);… f(my_image);// oops: this could be s-l-o-o-o-w
void f(Image& i);… f(my_image);// no copy, but f() can modify
                                // my_image
void f(const Image& i); … f(my_image);// f() won't mess with
                                // my_image
```

(pass-by-value)

here the image will be an INPUT

here the image will be an INPUT & OUTPUT

Variables scope

Scope

- The scope of an identifier is the portion of the program in which the identifier can be referenced
  - Some identifiers can be referenced throughout the program
  - Others can be referenced from only portions of a program
- **Example:**
  When we declare a local variable in a block (e.g., in a for loop), it can be referenced only in that block or in blocks nested within that block

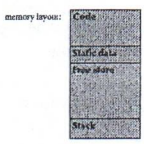
Methods of variable creation

- Global variable
  - declared very early stage (even before the "main()")
  - available always and from anywhere
  - created at the start of the program, and lasts until the end, stored in the **static data**
  - difficult to debug

Never use global variables!
-5 points at exams!

Methods of variable creation

- Local on the fly variables
  - simply created when they are needed
  - only available from within the routine/block in which they were created, stored in the **stack**
  - easy to debug



Methods of variable creation

- Local defined variable
  - created before they are needed
  - only available in the routine in which they were created
  - easy to debug, stored in the **stack**
  - the most usually favored method

A program written in C++

```
// This program calculates the number calories in a cheese sandwich
#include <iostream>
const int BREAD = 63;                  // global constant
const int CHEESE = 106;                // global constant
const int MAYONNAISE = 49;             // global constant
const int PICKLES = 25;                // global constant
int main()
{
    int totalCalories;                 // local variable
    totalCalories = 2 * BREAD + CHEESE + MAYONNAISE + PICKLES;
    cout << "There were " << totalCalories;
    cout << " calories in my lunch yesterday."<< endl;
    return 0;
}
```

Global constant are ok!

global constants are OKAY
global variables are NOT okay

Function definition and function prototype

```
#include <iostream>

int square (int);    // Function prototype

int main()
{
    for (int x = 0; x < 10; x++) //local on the fly variable
        cout << square(x) << ' ';
        cout << endl;
    return 0;
}
int square (int y)   // Function definition
{
    return y * y;
}
```

Local variable

this cout is outside the for loop, here we cannot access x anymore (x exists only in the for loop)

Global and Local declarations

```
#include <iostream>
void func( float );
const int a = 17;      // global constant
int b;                 // global variable
int c;                 // global variable
int main()
{
      b = 4;           // assignment to global b
      c = 6;           // assignment to global c
      func(42.8);
      return 0;
}
void func( float c)    // prevents access to global c
{
      float b;         // prevent access to global b
      b = 2.3;         // assignment to local b
      cout << " a = " << a;        // output global a (17)
      cout << " b = " << b;        // output local b (2.3)
      cout << " c = " << c;        // output local c (42.8)
}
```

Output:
a = 17 b = 2.3 c = 42.8

The local redefinition is stronger than the global/outside

Explanation

- In this example, function func accesses global constant a
- However, func declares its own local variable b and parameter c
- Local variable b takes precedence over global variable b, effectively hiding global variable b from the statements in function func
- Function parameter acts like local variable

Lifetime of a variable

- Local variables: variables declared inside a function or variables declared as function parameter
- When you will call the function, memory will be allocated for all local variables defined inside the function
- Finally memory will be deallocated when the function exits
- The period of time a variable will be "alive" while the function is executing is called "lifetime" of this variable

Another example - Scopes nest

```
int x;  // global variable – avoid those where you can
int y;  // another global variable

int f();

int main(){
      x =8; y = 3;
      f();
      cout << x << ' ' << y << '\n';
}

int f()
{
  int x;                  // local variable (Note – now there are two
                          // x's)
  x = 6;                  // local x, not the global x
  {
      int x = y;          // another local x, initialized by the global y
                          // (Now there are three x's)
      ++x;                // increment the local x in this scope
  }
  // what is the vale of x here?   6
  y++;
}
```

→ 8 4

DEMO

// avoid such complicated nesting and hiding: keep it simple!