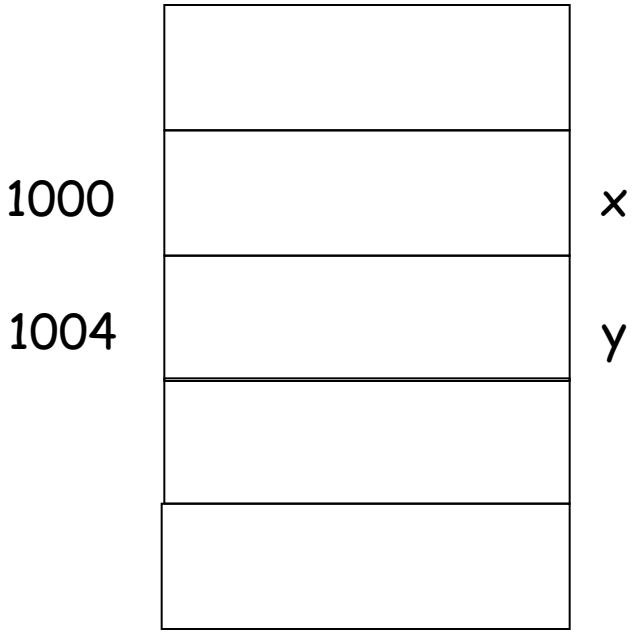
# Pointers and Dynamic Variables

Fall 2018, CS2

# Data, memory

- memory address: every byte is identified by a numeric address in the memory.
- a data value requiring multiple bytes are stored consecutively in memory cells and identified by the address of the first byte
- In program we can:
  - find amount of memory (num. of bytes) assigned to a variable or a data type: sizeof(int), sizeof x
  - find the address of a variable: &x

# Example




int x, y;

int takes 4 bytes

address of x is the address of its first byte…

# Pointers Variables (or Pointers)

- Pointer variables: a variable that stores memory address (of another variable)
  - is used to tell where a variable is stored in memory
  - Pointers **"point" to a variable**
- Memory addresses can be used to access variables
  - Array variable actually stores address of the first element in array
    - int a[10]; cout <<a<<endl; cout <<&(a[0])<<endl;
  - When a variable is used as a call-by-reference argument, its address is passed

# Declaring Pointers

- Pointer variables must be declared to have a pointer type
  - Ex: To declare a pointer variable p that can "point" to a variable of type double:

```
double *p;
```

  - The asterisk identifies p as a pointer variable

# Declaring pointer variables

- **DataType * pointerVariable; //declare a pointerVariable that can be used to point to DataType variable**

```
int * p;
char *cptr;
DayOfYear * pDate;  //pDate is a pointer pointing to DayOfYear obj
 double *q;  //no space between * and variable name
```

- Like other variables, before initialization, p and cptr might contain some arbitrary value
- So, important to initialize:

```
int *p=NULL;  // assign NULL constant to p, a pointer variable to indicate
              // that p does not point to any valid data
             // internally, NULL is value 0.
```

Common pitfall:

```
int *p1, *p2;   //p1,p2 are both pointers that point to int
int *p1, p2;   //p1 is pointer, but p2 is int
//* only applies  to the variable that follows it, p1; not p2
```

# pointer to different types

- **DataType * pointerVariable;  //declare a pointerVariable that can be used to point to DataType variable**

  int * p=NULL;

  char *cptr=NULL;

  DayOfYear * pDate=NULL;  //pDate is a pointer pointing to DayOfYear obj

  double *q=NULL;  //no space between * and variable name

- Pointers to different types
  - have same size, sizeof(int *)==sizeof(double *) //8
  - why differentiate them?
    - int and double, char, … takes different number of bytes, and interpret data differently…

# "address of" Operator

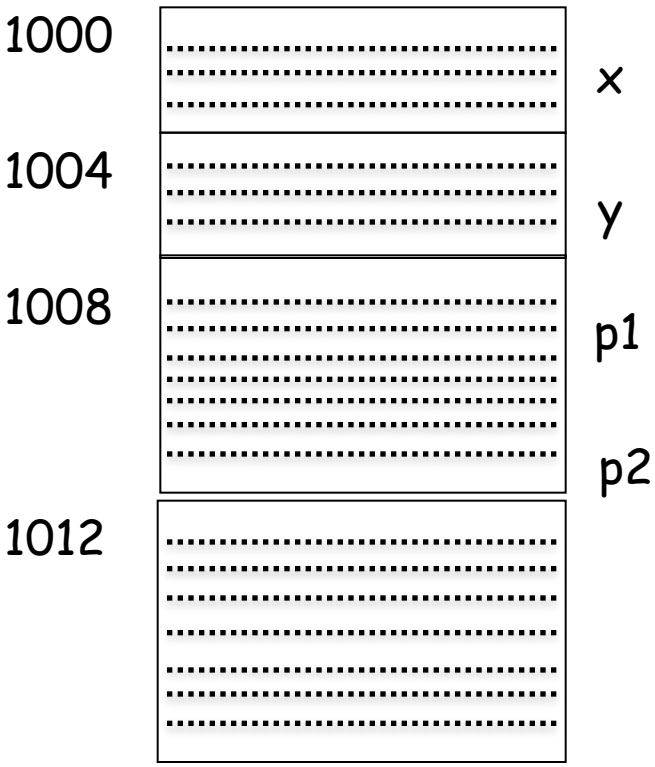
- &variable: yield the address of a variable
  - can then be assigned to a pointer variable

```
int v1;
int * p1;
p1 = &v1;   // assign "address of v1" to p1
            //p1 is now a pointer (pointing) to v1
int a[10];
assert (a==&(a[0])); //array variable itself stores address
```

# Example

```
int x, y;
int *p1, *p2;
```

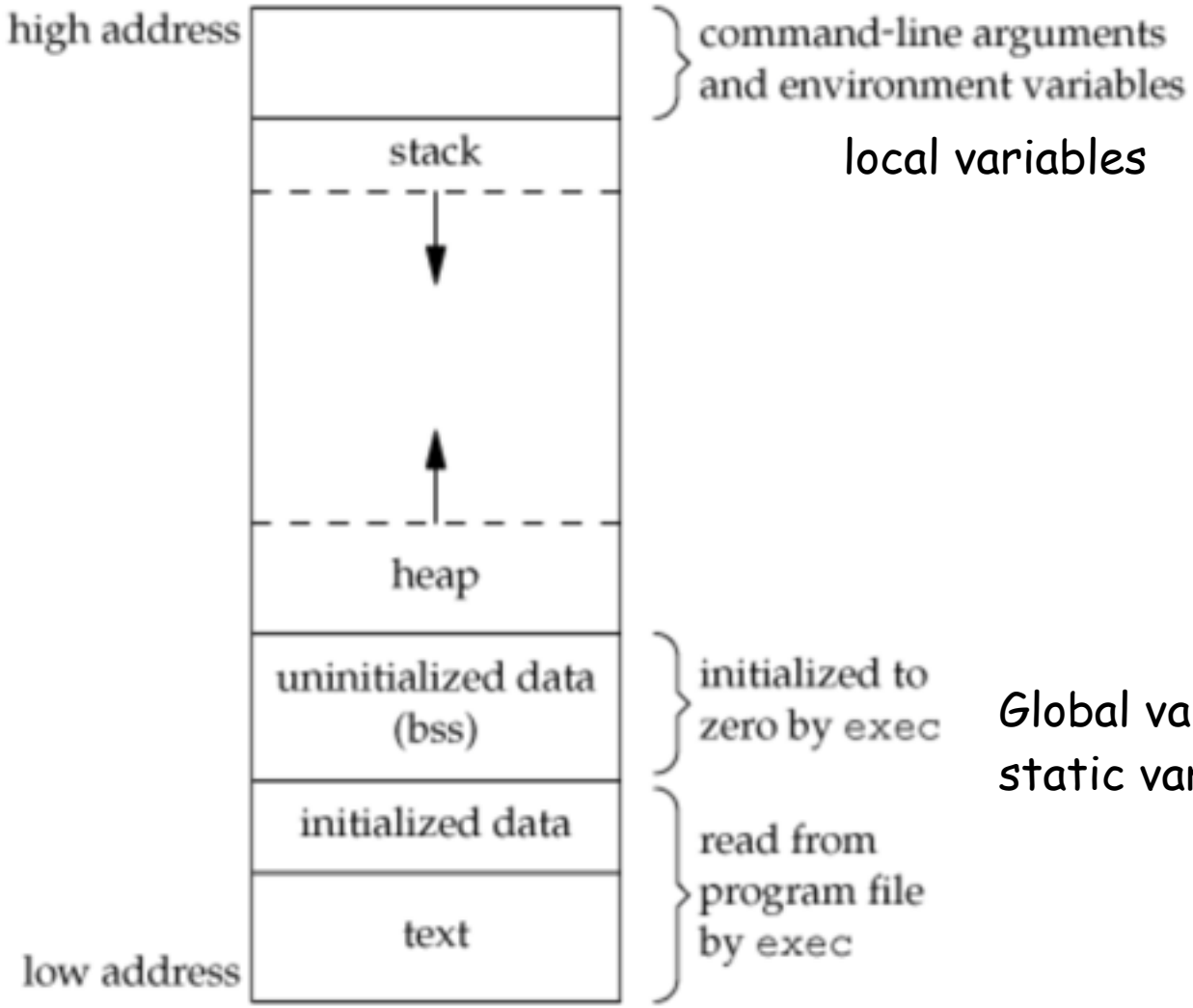
address name




```
address of x: 140724463361388
address of y: 140724463361384
address of p1:140724463361376
address of p2:140724463361368
```

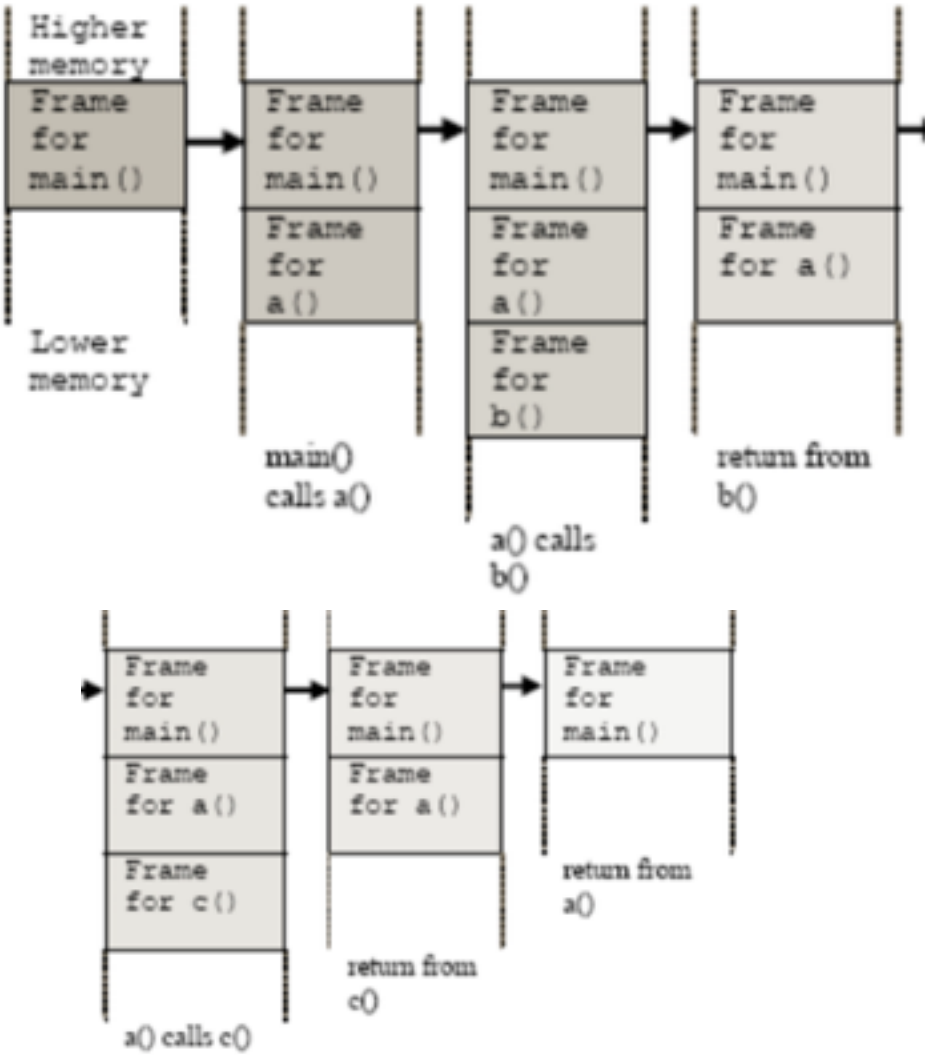
```
address of x0x7ffcbf8f5a8c
address of y0x7ffcbf8f5a88
address of p10x7ffcbf8f5a80
address of p20x7ffcbf8f5a78
```

# Typical layout of a program in memory

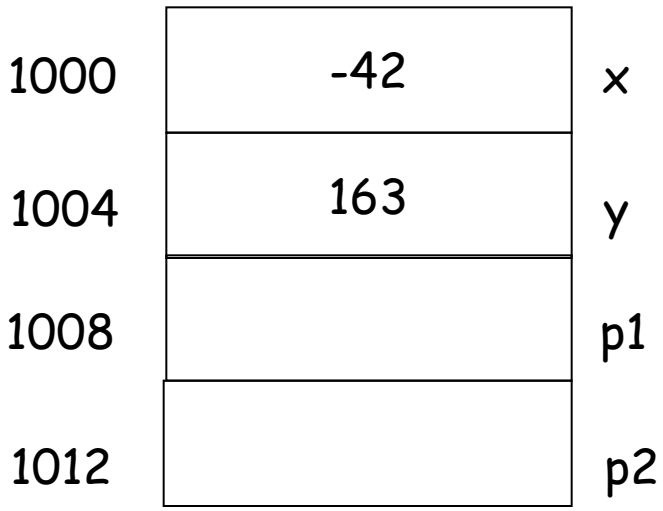


# Stack and StackFrame

```
.....
int a()
{
     b();
     c();
    return 0;
}

int b()
{ return 0; }

int c()
{ return 0; }

int main()
{
   a();
   return 0;
}
```

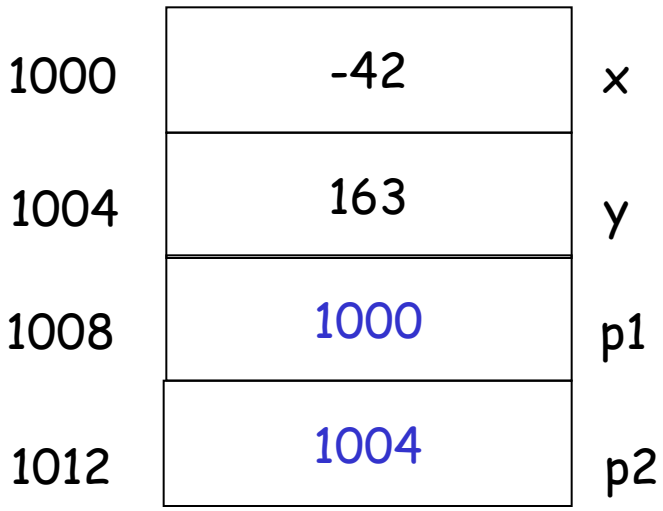


# Example




```
int x, y;
int *p1, *p2;
x=-42;
y=163;
```

# Example




```
int x, y;
int *p1, *p2;
x=-42;
y=163;
p1=&x;
p2=&y;
```

# dereferencing Operator

- *pointerVariable: the variable that pointerVariable points to
  - Here the * is **dereferencing** operator, pointerVariable is said to be **dereferenced**

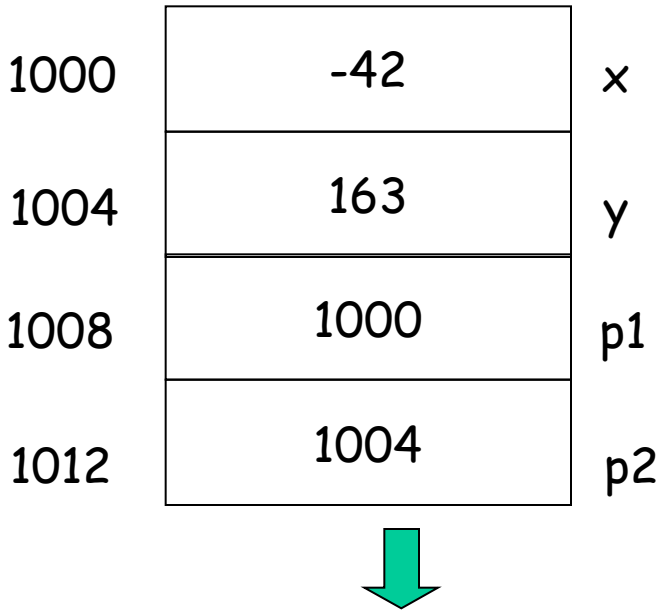
```
int v1;
int *p1;  //this * means p1 is a pointer
p1 = &v1;   // assign "address of v1" to p1
cout << *p1; //display the int that p1 points to, i.e,  v1
```

Pitfall/reminder: the context is important!

* used between type and name

vs. * before a pointer variable

# Example




```
int x, y;
int *p1, *p2;
x=-42;
y=163;
p1=&x;
p2=&y;
```

```
*p1=17;

//*p1 is another name of for x
```

# Fundamental pointer operations

& **address-of** a variable. Its operand is a variable.

example: int *p; int a=10; p=&a;

* variable that a pointer is pointed to. Its operand is a pointer.

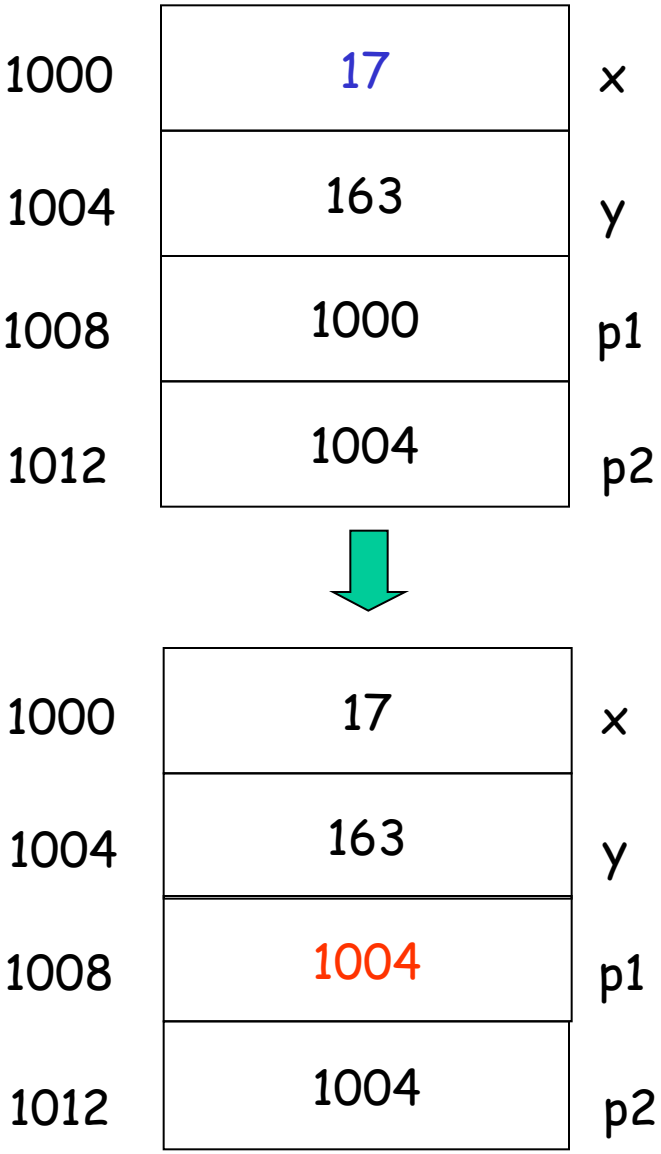
example: *p=5;

they are used to move back and forth between variables and pointers to those variables.

```
int *p;
*p=5; //the variable pointed to by aptr has to be valid
```

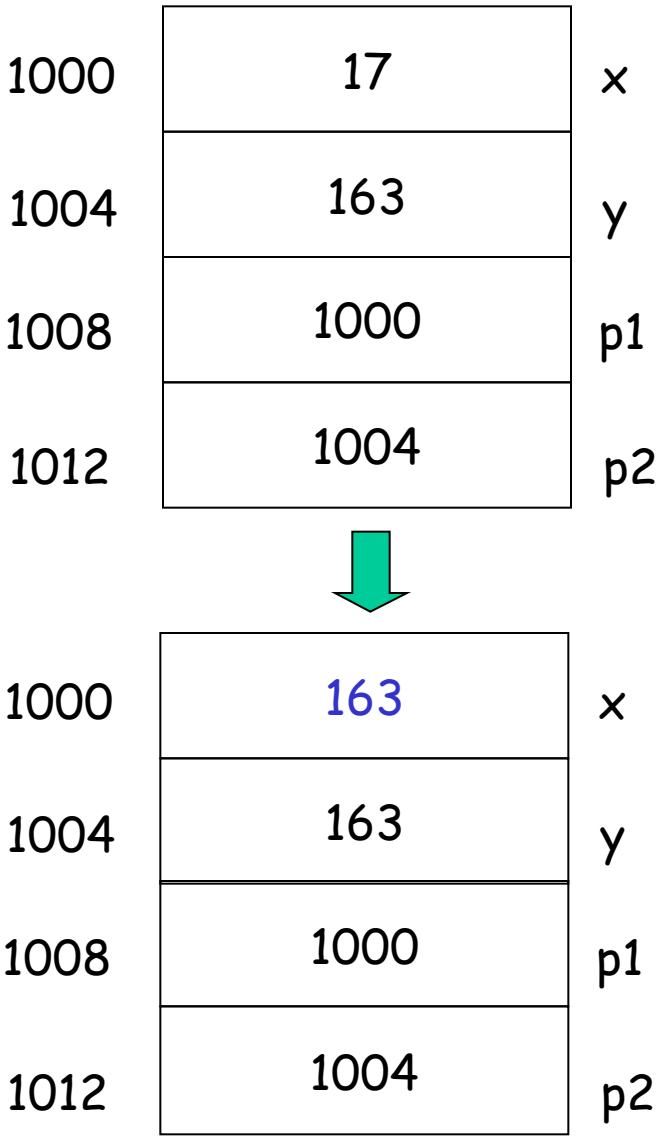
int *p=NULL; <=> int *p; p=NULL;

# example




```
int x, y;
int *p1, *p2;
x=-42;
y=163;
p1=&x;
p2=&y;
*p1=17; /* another name of for x*/

p1=p2; /* pointer assignment, now
   two pointers point to the same
   location*/
```

# example




```
int x, y;
int *p1, *p2;
x=-42;
y=163;
p1=&x;
p2=&y;
*p1=17; /* another name of for x*/
```

```
*p1=*p2; /*value assignment*/
```

```
//think of *p1 as another name of the
    variable p1 points to.
```

# Usage of pointers

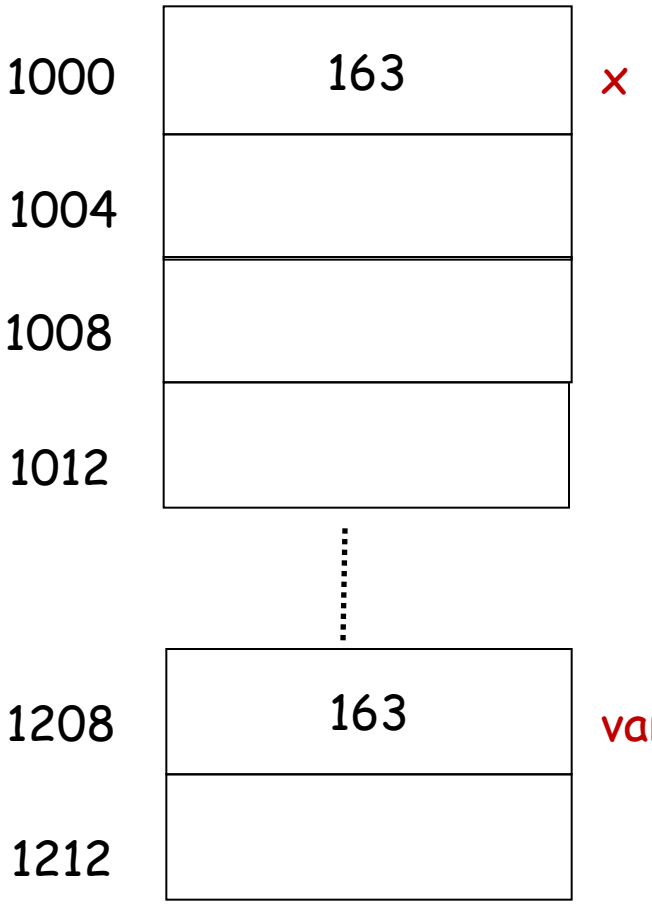
- Allow one to refer to a large data structure in a compact way.
  - Each pointer (or memory address) typically fits in four bytes of memory!
  - Array: static or dynamic arrays
- Different parts of a program can share same data:

  passing parameters by reference (passing address between different functions), or by pointers
- One can reserve new memory in a running program: dynamic memory allocation
- Build complicated data structures by linking different data items

# Passing parameters by reference using pointers

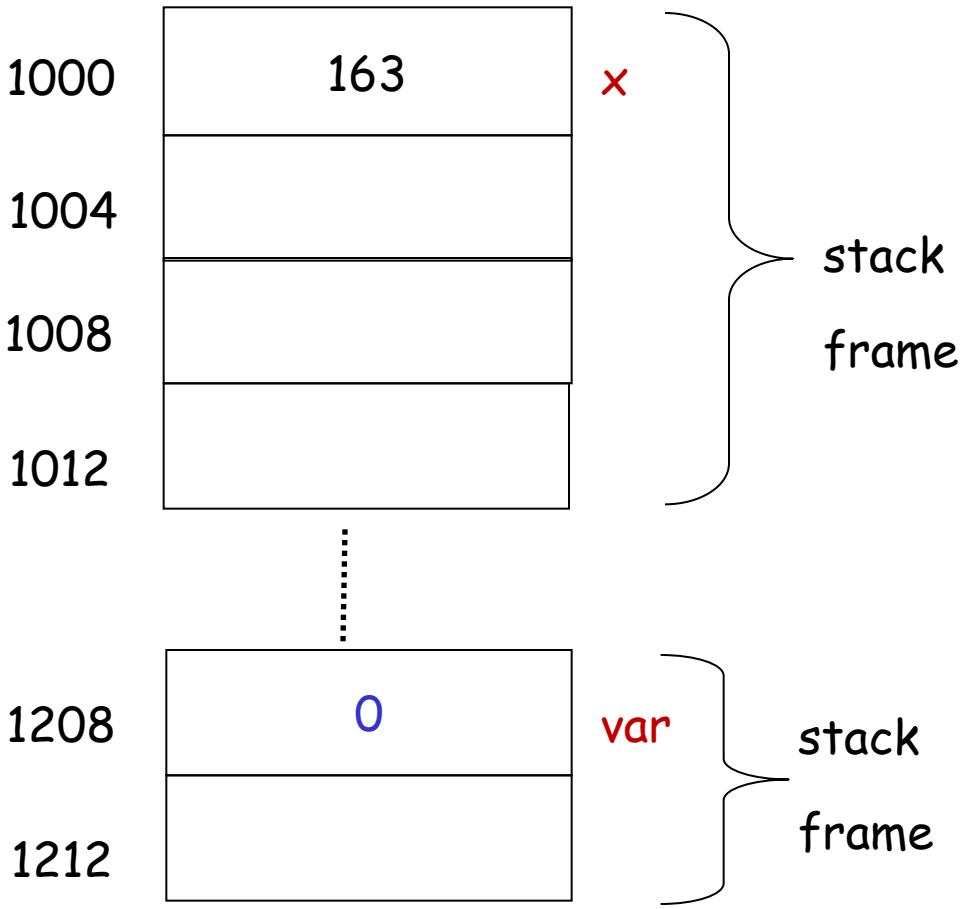
Suppose we want to set X (defined in main() function) to zero, compare the following code:

```
/*pass by value*/
void SetToZero1 (int var) {
     var=0;
}

/*pass by pointer*/
void SetToZero2(int *ip) {
     *ip=0;
}

int main()
{
   int x=163;
   SetToZero1(x)
   SetToZero2 (&x);
}
```



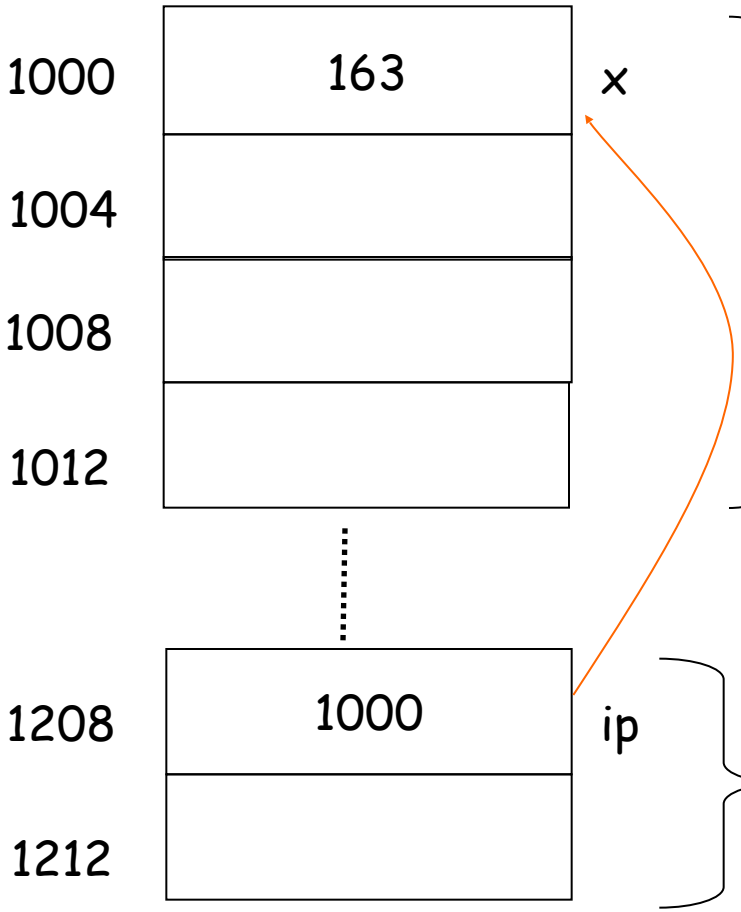

```
int main ()
{
   …
   x=163;
   SetToZero(x);
   …
   …
}
```

```
void SetToZero (int var) {
    var=0;
}
```



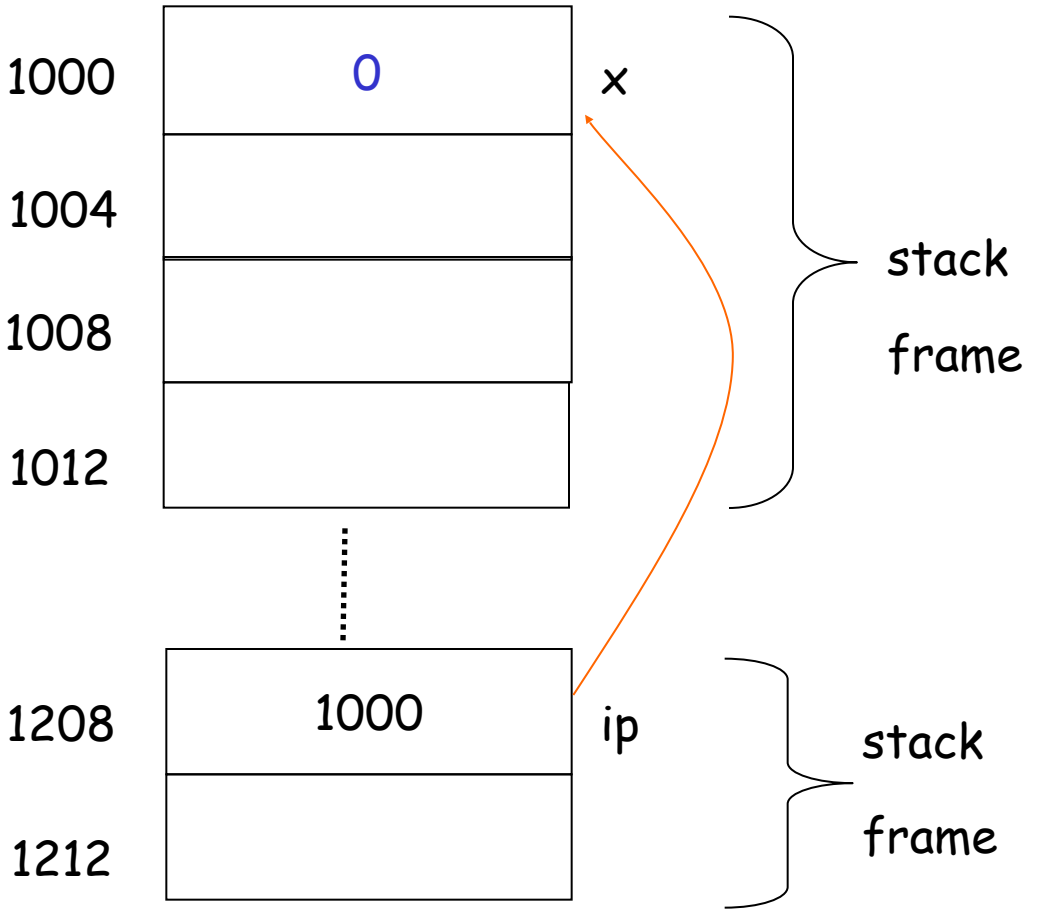

```
int main ()
{
   …
   x=163;
   SetToZero(x);
   …
   …
}
```

```
void SetToZero (int var) {
   var=0;
}
```




```
int main ()
{
    …
    x=163;
    SetToZero2(&x);
    …
    …
}
```

```
void SetToZero2(int *ip)
{
    *ip=0;
}
```

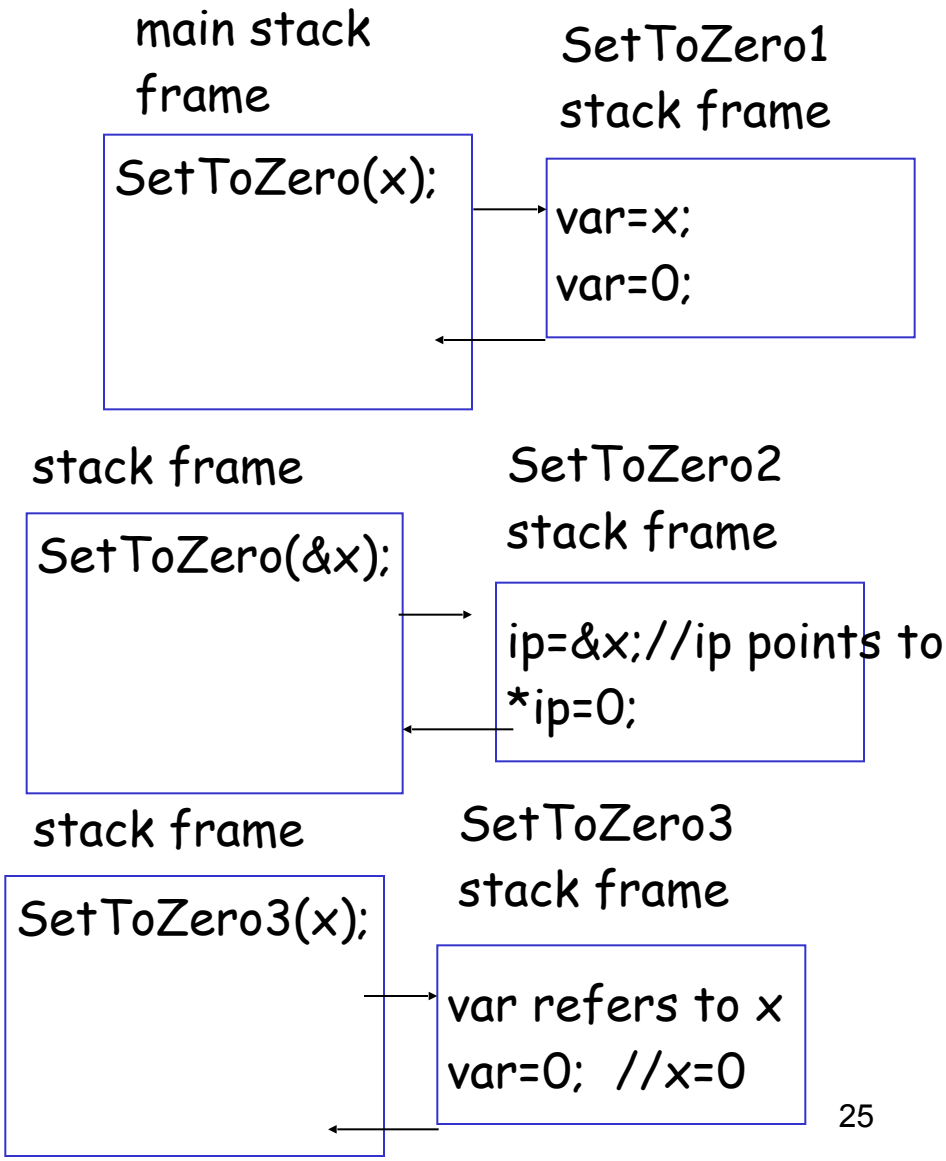



```
int main ()
{
    …
    x=163;
    SetToZero(&x);
    …
    …
}
```

```
void SetToZero(int *ip)
{
    *ip=0;
}
```

# Passing parameters

```
void SetToZero1 (int var) {
      var=0;
}
SetToZero(x);
/* has no effect on x*/


void SetToZero2(int *ip) {
      *ip=0;
}

SetToZero2(&x);

void SetToZero3 (int & var){
     var = 0;
}

SetToZero3 (x);
```



# Example

write a program to solve quadratic equation:

ax^2 + bx + c = 0;

program structure:

input phase: accept values of coefficients from users;

```
void GetCoefficients(double *pa, double *pb, double *pc);
```

computation phase: solve the equation based on those coefficients;

```
void SolveQuadratic(double a, double b, double c, double *px1, double *px2);
```

output phase: display the roots of the equation on the screen

```
void DisplayRoots(double x1, double x2);
```

## Variable Scopes and Lifetimes

—- a bigger picture about memory used by a program

# Global Variables

- Variables declared outside any function are global variables
  - they have "global scope", i.e., they can be accessed by the name from all parts of a program —- unless there is an eclipse!
  - they comes into being when program starts, and disappears when program ends ==> static lifetime
- We discourage the usage of global variables
  - too many cooks in the kitchen: everyone can modify it

# Local Variables

- Variables declared in a function are local variables
  - they have "local scope": they can be accessed using the name from the function/block
  - They are **typically** created when the function is called, and destroyed when the function call ends ==> **automatic lifetime**
- Local variable with static lifetime?

```
void some_func()
{
    static int counter=0;  //created at program starts,
      //destroyed when program ends
    counter++;
    cout <<"called " << counter<<" times\n";
   //…
 }
```

# Dynamic Variables

- Programmer/Code can create variables and then destroy them using operators **new** and **delete**
  - such variables are dynamic variables, their lifetime is dynamic (decided at running time, based upon running time condition). **They have no name.**
- ***e.g.,***

```
int *p1; //declare a pointer variable
p1 = new int; //create a int variable, save its
   address in p1
```

  - This variable can only be referred by address (as it has no name), *p1
  - *p1 can be used any place an integer variable can

```
cin >> *p1;
*p1 = *p1 + 7;
```

# Display 9.2

**Basic Pointer Manipulations**

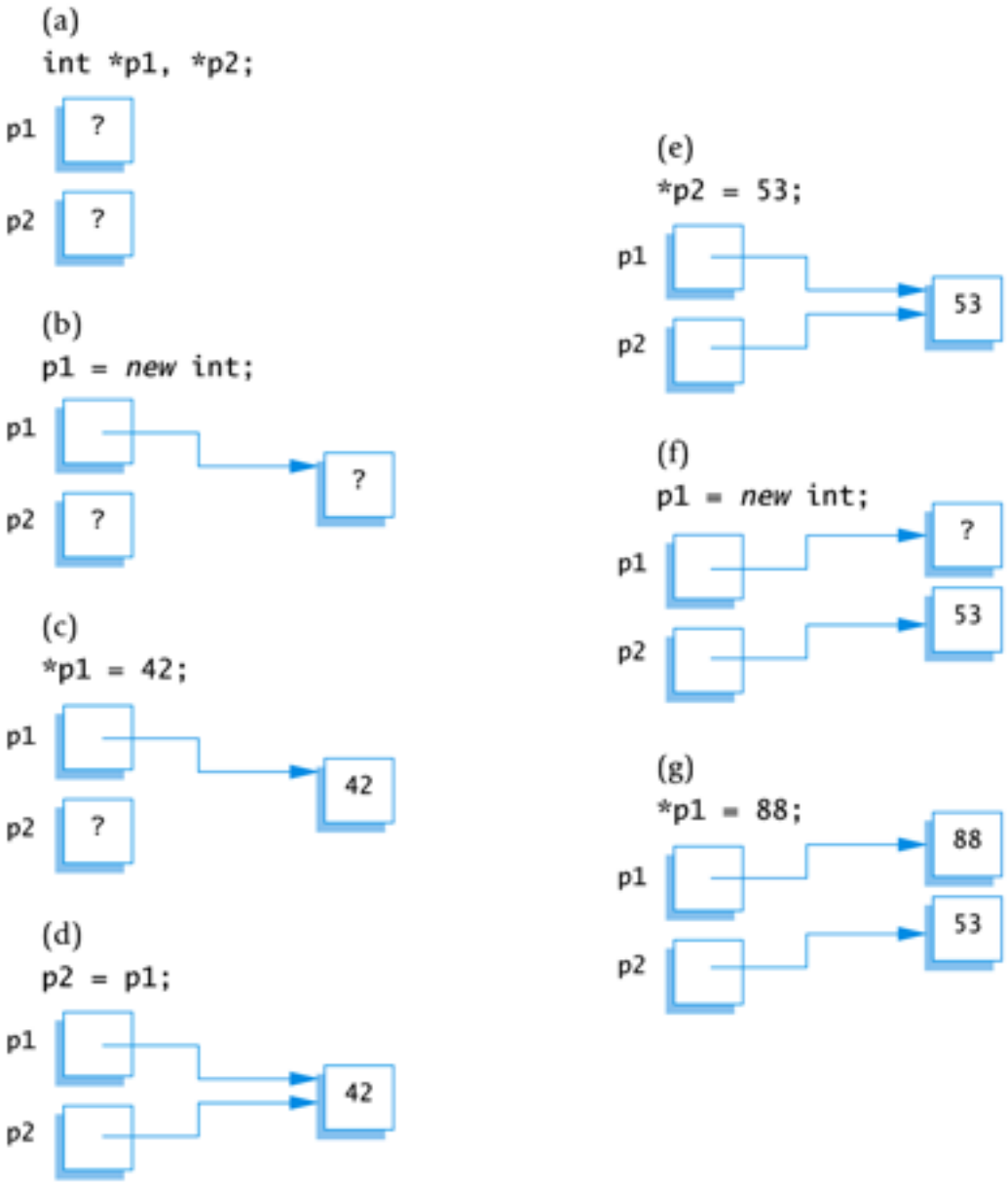
```
//Program to demonstrate pointers and dynamic variables.
#include <iostream>
using namespace std;

int main( )
{
    int *p1, *p2;

    p1 = new int;
    *p1 = 42;
    p2 = p1;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    *p2 = 53;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    p1 = new int;
    *p1 = 88;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    cout << "Hope you got the point of this example!\n";
    return 0;
}
```

**Sample Dialogue**

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

# Display 9.3

**DISPLAY 9.3 Explanation of Display 9.2**



# Caution! Pointer Assignments

- Some care is required making assignments to pointer variables
  - p1= p3; // changes the location that p1 "points" to
  - *p1 = *p3; // changes the value at the location that
    // p1 "points" to

# Basic Memory Management

- An area of memory called the **freestore/heap** is reserved for dynamic variables
  - New dynamic variables use memory in the freestore
  - If all of the freestore is used, calls to new will fail
- Unneeded memory can be recycled
  - When variables are no longer needed, they need to be deleted and the memory they used is returned to the freestore

# delete Operator

- When dynamic variables are no longer needed, delete them to recycle memory to freestore
  - e.g.,

```
delete p;
```

memory used by the variable that p pointed to is back in freestore. p still stores that address.

```
*p=10;    // Disaster!!!
p = NULL;  //value of p is now NULL
```

# Dangling Pointers

- Using delete on a pointer variable destroys the dynamic variable pointed to
- If another pointer variable was pointing to the dynamic variable, that variable is also undefined
- Undefined pointer variables are called dangling pointers
  - Dereferencing a dangling pointer (*p) is usually disasterous

# Type Definitions

- A name can be assigned to a type definition, then used to declare variables
- The keyword `typedef` is used to define new type names
  - Syntax:

```
typedef Known_Type_Definition
  New_Type_Name;
```

    - `Known_Type_Definition` can be any type

# Defining Pointer Types

- To avoid mistakes using pointers, define a pointer type name
  - Example:

```
typedef int* IntPtr;
```

Defines a new type, `IntPtr`, for pointer variables containing pointers to `int` variables

```
IntPtr p;
```

is equivalent to

```
int *p;
```

# Multiple Declarations Again

- Using our new pointer type defined as

```
typedef int* IntPtr;
```

Then, we can prevent this error in pointer declaration:

```
int *P1, P2;//Only P1 is a pointer variable
```

with

```
IntPtr P1, P2;
// P1 and P2 are pointer variables
```

# Pointer Reference Parameters

- A second advantage in using `typedef` to define a pointer type is seen in parameter lists
  - Example:

```
void sample_function(IntPtr&
  pointer_var);
```

is less confusing than

```
void sample_function( int*& pointer_var);
```