Object-Oriented Programming
(in C++)

Pointers and References

Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

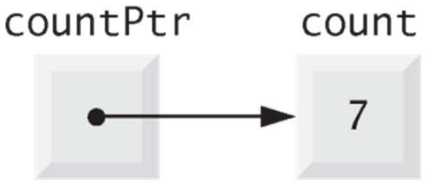
Outline

- Introduction to Pointers
  - Pointer Declaration and Initialization
  - Working with Pointers
  - Problems with Pointers
- Pointers vs References
- `sizeof` Operator
- Using `const` with Pointers
- Pointer Arithmetic and Arrays
- Arrays of Pointers
- Dynamic Memory Management
- Function Pointers

Pointer Variable Declaration

- Pointer variables contain memory addresses as values
  - A variable contains a specific value
    - Recall: a variable is a memory cell
    - A variable name directly references a value
  - A pointer variable contain the address of a variable (or a pointer or a function) that has specific value
    - A pointer indirectly references a value

```
int count;          // an int variable
int *countPtr;      // a pointer to int
countPtr = &count; // &: address operator
```



Pointer `countPtr` indirectly references a variable that contains the value 7

Pointer Variable Initialization

- Initialized to an address
  - Using the address (or “address-of”) operator **&**

```
int *countPtr = &count;
```

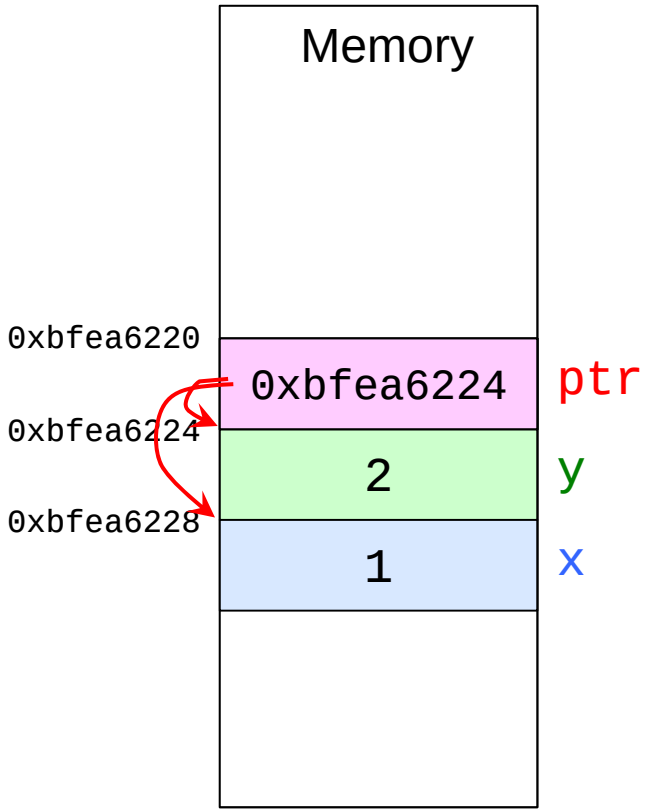
- Initialized to `0`, `NULL`
  - `0` or `NULL` points to nothing (`null pointer`)
  - `0` is the only integer value that can be assigned directly to a pointer variable without casting the integer to a pointer type first
  - Example:

```
int *countPtr = 0;
int *anotherPtr = (int *)0xbfea6220;
```

Pointers: An Example

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

int main() {
   int x=1, y=2; // int variables
   int *ptr=0; // a pointer to int
   ptr = &x;
   cout << ptr << ", ";
   ptr = &y;
   cout << ptr << endl;
   return 0;
}
```

```
0xbfea6228, 0xbfea6224
```

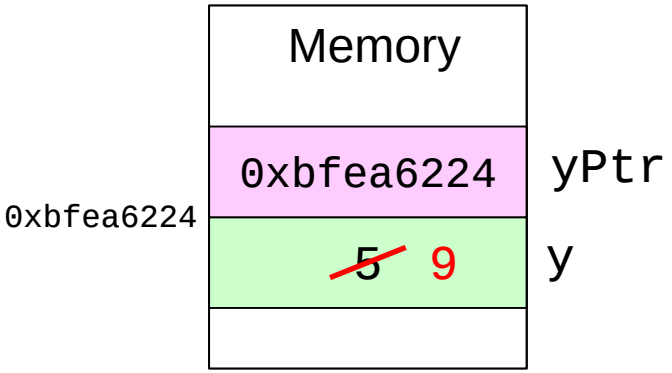


- Assume a byte is the smallest unit of addressable memory

Pointer Operator

- Indirection/Dereferencing operator (*)
  - Returns a synonym (an alias) for the object its operand points to
  - Example:

```
int y = 5, *yPtr;
yPtr = &y;
*yPtr = 9; // y = 9
```



    - *yPtr returns y (because yPtr points to y)
  - An attempt to dereference a variable that is not a pointer is a compilation error
- * and & are inverses of each other
  - *&y -> y
  - &*yPtr -> yPtr

Potential Points of Confusion

- Using * in the declaration of a pointer is NOT an operator
  - It is possible to have a * interpreted as a unary or binary operator, but only when not involved in a declaration

```
int *xPtr, *yPtr;
*xPtr = *yPtr;
```

```
int *xPtr = &x;
```

- Each pointer must be declared with the * prefixed to the name (either with or without spaces in between---compilers ignore the space)

```
int *yPtr;
```
=
```
int * yPtr;
```
=
```
int* yPtr;
```

recommended

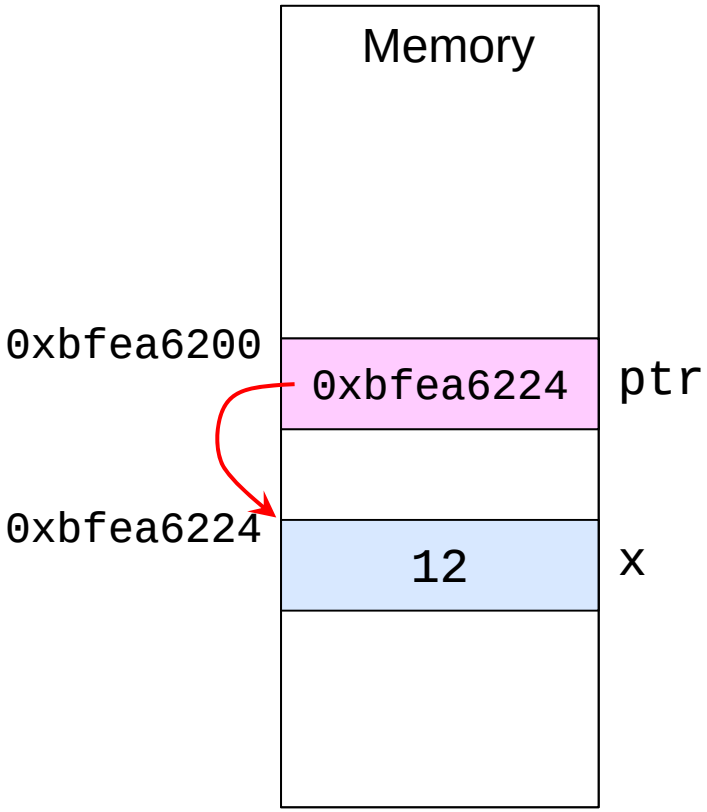
```
int* x, y;
```
=
```
int *x, y;
```
=
```
int *x;
int y;
```

Outline

- Introduction to Pointers
  - Pointer Declaration and Initialization
  - Working with Pointers
  - Problems with Pointers
- Pointers vs References
- `sizeof` Operator
- Using `const` with Pointers
- Pointer Arithmetic and Arrays
- Arrays of Pointers
- Dynamic Memory Management
- Function Pointers

Working With Pointers (1/4)

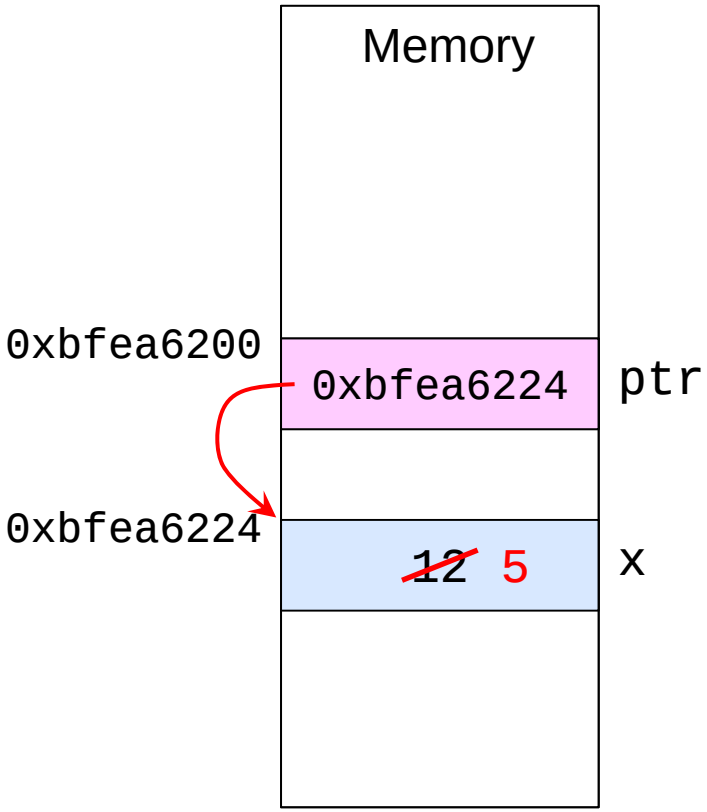
```
int x;
x = 12;
...
int *ptr = 0;
ptr = &x;
std::cout << *ptr;
```



- `*ptr` is the value in the place to which `ptr` points

Working With Pointers (2/4)

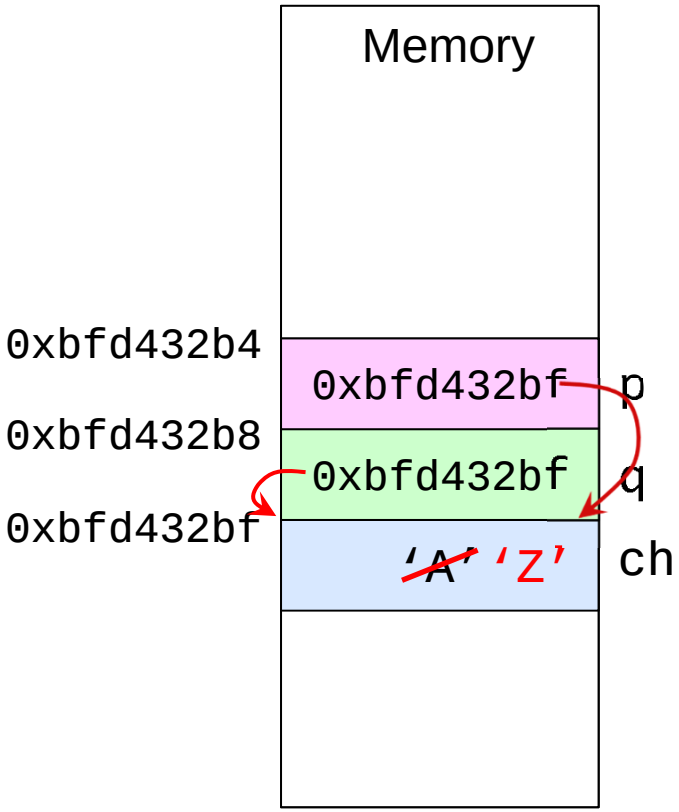
```
int x;
x = 12;
...
int *ptr = 0;
ptr = &x;
*ptr = 5;
```



- Changes the value at the address that `ptr` points to to 5

Working With Pointers (3/4)

```
char ch;
ch = 'A';

char *q = 0;
q = &ch;
*q = 'Z';

char *p = 0;
p = q;
```

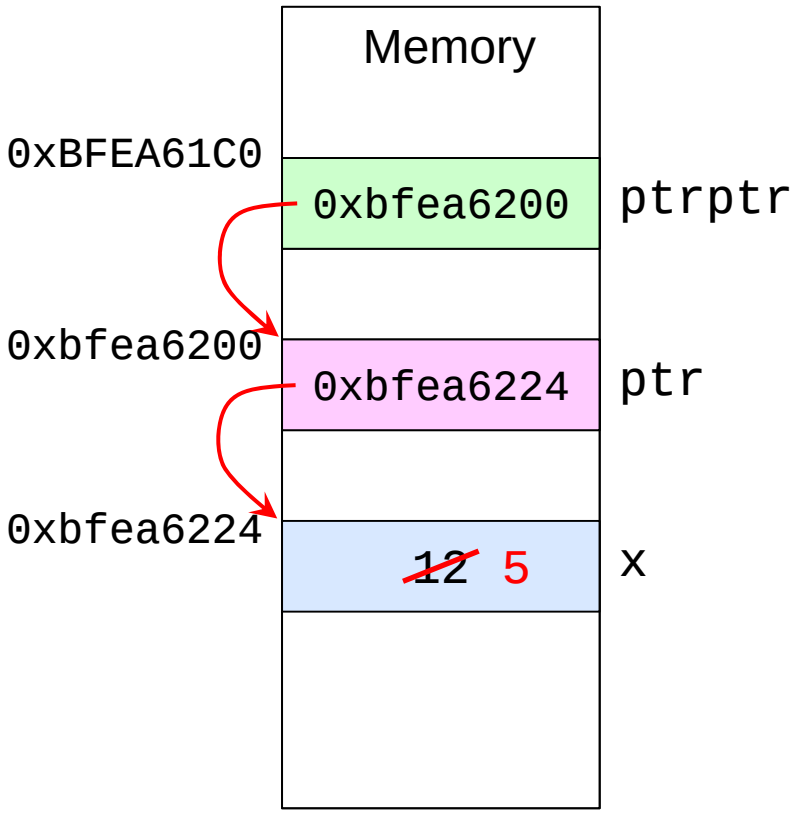


- Now p and q both point to ch

Working With Pointers (4/4)

```
int x;
x = 12;
...
int *ptr = 0;
ptr = &x;
...
int **ptrptr = 0;
ptrptr = &ptr;

**ptrptr = 5;
// *(*ptrptr) = 5;
```

An pointer that points to an `int` pointer



- Changes the value at the address that `ptr`, which is pointed by `ptrptr`, points to 5

Outline

- Introduction to Pointers
  - Pointer Declaration and Initialization
  - Working with Pointers
  - Problems with Pointers
- Pointers vs References
- `sizeof` Operator
- Using `const` with Pointers
- Pointer Arithmetic and Arrays
- Arrays of Pointers
- Dynamic Memory Management
- Function Pointers

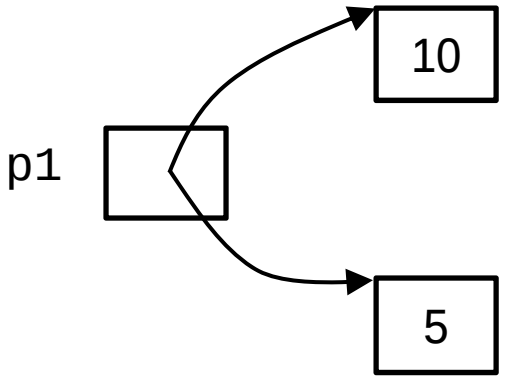
Common Programming Errors

- Dereferencing a null pointer is often a fatal execution-time error
- An uninitialized pointer could point to a random memory location
  - Dereferencing an uninitialized pointer could
    - cause a fatal execution-time error due to dereferencing a null pointer, or
    - accidentally modify the data on which it points to
- An attempt to dereference a variable that is not a pointer is a compilation error

Problems with Pointers (1/2)

- Memory leak (See Common Programming Error 11.5)
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)
    - Pointer p1 is set to point to a newly created heap-dynamic variable
    - Pointer p1 is later set to point to another newly created heap-dynamic variable
    - The process of losing heap-dynamic variables is called *memory leak*
  - Both implicit or explicit deallocation may have the problem of memory leak

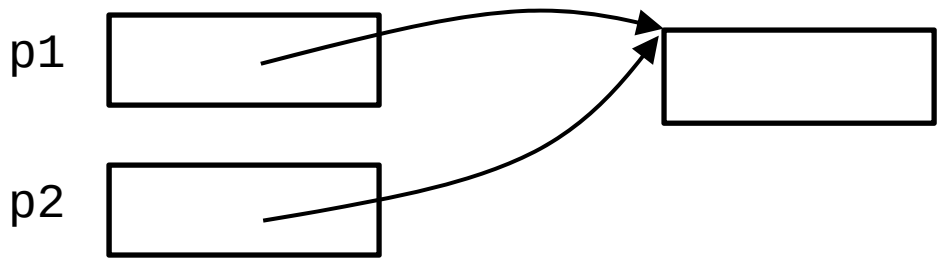
```
int *p1;
p1 = new int(10);
...
p1 = new int(5);
```



Problems with Pointers (2/2)

- Dangling pointers (See Common Programming Error 11.8)
  - A pointer points to a heap-dynamic variable that has been deallocated
  - Caused by explicit deallocation (`free()` or `delete`)
  - E.g.,

```
int *p1, *p2;
p1 = new int;
p2 = p1;
delete p2;
*p1 = 1;
```



Pointers: C vs C++

- C++ is much more strict when dealing with pointer types
  - This is especially apparent in the implementation of `void` pointers
  - A `void` pointer can point to a variable of any type

```
int ivalue = 13;
float fvalue = 8.3;
int *iptr;
float *fptr;
void *vptr;

fptr = &ivalue; // ERROR!
iptr = &ivalue; // CORRECT
vptr = &ivalue; // CORRECT
vptr = &fvalue; // CORRECT
iptr = fptr;    // ERROR!
vptr = fptr;    // CORRECT
vptr = iptr;    // CORRECT
fptr = vptr;    // CORRECT in C, but ERROR in C++
```

- In C, pointer conversions to and from `void*` were always implicit
- In C++, conversions from `T*` to `void*` are implicit, but `void*` to anything else requires a cast

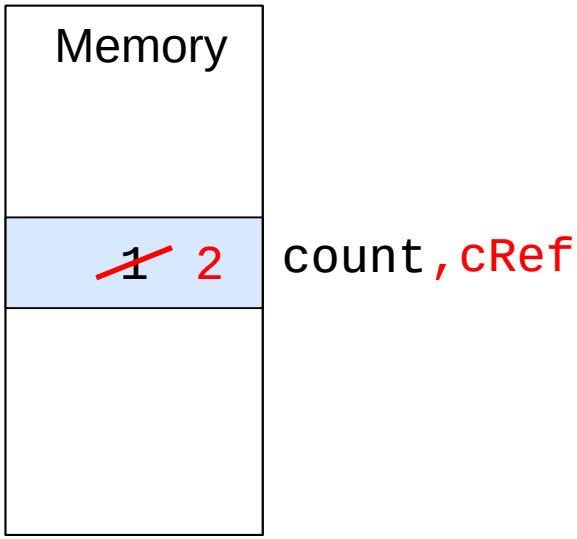
Outline

- Introduction to Pointers
- Pointers vs References
- `sizeof` Operator
- Using `const` with Pointers
- Pointer Arithmetic and Arrays
- Arrays of Pointers
- Dynamic Memory Management
- Function Pointers

References (C++ Only)

- A reference is an implicit pointer that is automatically dereferenced
- A reference also acts as an alternative name (an alias) for another variable (Chapter 5.15)
- A reference has to be initialized when it is declared
- Unlike pointers, no memory location is required

```
// count is an integer variable
int count = 1;

// cRef is an alias for count
int &cRef = count;

// increment count using its alias cRef
cRef++;
```

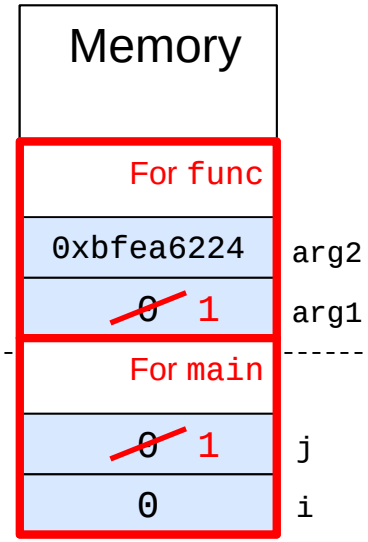


A Tip for Using References

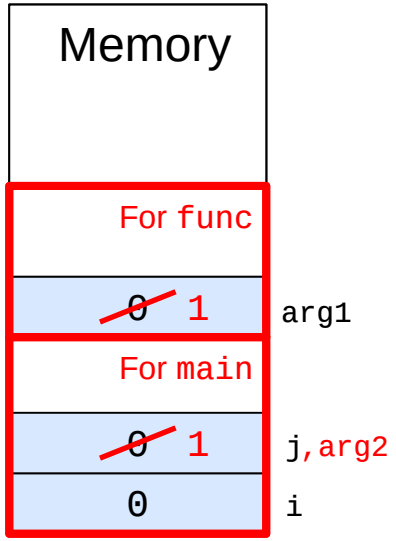
- The code is cleaner, because it is not necessary to use the * deference operator
  - The programmer does not have to remember to pass the address of the function argument
- Unlike passing with a pointer, no memory location is required and no copy of the function argument is made when using a reference
- References have advantages over regular pointers when passed to functions (see Chapter 5)
- TIP: Use references rather than pointers when passing variables to functions by address

Pass-by-Reference: Pointer vs Reference

```
void func(int arg1, int* arg2) {
  // arg1 is passed by value
  // arg2 is passed by reference
  arg1++;
  (*arg2)++;
}

void main() {
  int i,j;
  i = j = 0;
  func(i, &j);

  // i=0, j=1
  return 0;
}
```




```
void func(int arg1, int& arg2) {
  // arg1 is passed by value
  // arg2 is passed by reference
  arg1++;
  arg2++;
}

int main() {
  int i,j;
  i = j = 0;
  func(i, j);

  // i=0, j=1
  return 0;
}
```



Ref: Chapter 5

References vs Pointers

Restrictions	Reference	Pointer
It reserves a space in the memory to store an address that it points to or references	✖	O
It has to be initialized when it is declared	O	✖
It can be initialized at any time	✖	O
Once it is initialized, it can be changed to point to another variable of the same type	✖	O
It has to be dereferenced to get to a value it points to	✖	O
It can be a NULL reference/pointer	✖	O
It can point to another reference/pointer	✖	O
An array of references/pointers can be created	✖	O

Outline

- Introduction to Pointers
- Pointers vs References
- `sizeof` Operator
- Using `const` with Pointers
- Pointer Arithmetic and Arrays
- Arrays of Pointers
- Dynamic Memory Management
- Function Pointers

`sizeof` Operator

- The unary operator `sizeof` determines the size of an array (or of any other data type, variable, constant, or even an expression) in bytes during program compilation

  No negative impact on execution performance

  - Size of a variable

    ```
    sizeof(var)  ≡  sizeof var
    ```

  - Size of a type name

    ```
    sizeof(data_type
    )
    ```

  - Return type: `size_t` (`unsigned int`)

`sizeof` Operator: An Example

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

int main() {
  double i;
  double array[10];

  cout << sizeof(i) << endl;       // sizeof i
  cout << sizeof(array) << endl;  // sizeof array
  cout << sizeof(double) << endl;
  cout << sizeof(double*) << endl;

  return 0;
}
```

```
8
80
8
4
```

- Types may have different sizes based on the platform running the program

`sizeof`: Array Name vs Pointer

- When applied to the name of an array, `sizeof` returns the total number of bytes in the array
- When applied to a pointer, `sizeof` returns the size of the pointer

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

int main() {
  double array[20];
  double *ptr;

  ptr = array;
  cout << "sizeof(array)=" << sizeof array << endl;
  cout << "sizeof(ptr)=" << sizeof(ptr) << endl;

  return 0;
}
```

```
sizeof(array)=160
Sizeof(ptr)=4
```

Outline

- Introduction to Pointers
- Pointers vs References
- `sizeof` Operator
- Using `const` with Pointers
- Pointer Arithmetic and Arrays
- Arrays of Pointers
- Dynamic Memory Management
- Function Pointers

Using const with Pointers

- Four ways to declare a pointer with const
  - A nonconstant pointer to nonconstant data

```
int* ptr;
```

A pointer to int

  - A nonconstant pointer to constant data

```
const int* ptr;
```

```
int const* ptr;
```

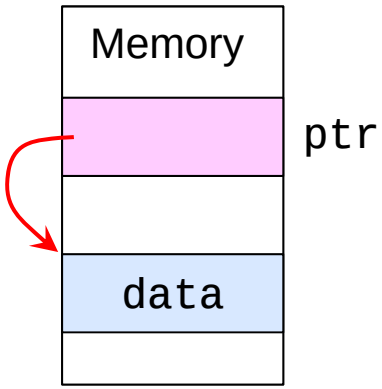
  - A constant pointer to nonconstant data

```
int*const ptr;
```

  - A constant pointer to constant data

```
const int*const ptr;
```

- ptr is constant or not
- *ptr is constant or not



Nonconstant Pointer to Constant Data

- Could point to other memory location, and the data at that location cannot be modified through the pointer
- Could be used to receive an argument that is read only (even though pass-by-reference is used)

```
int g;

void f1(const int* ptr) {
  *ptr = 100; // ERROR: *ptr is const
  ptr = &g;   // ALLOWED: ptr is not const
  *ptr = 100; // ERROR: *ptr is const
}
void f2(const int* array) {
  array[0] = 100; // ERROR: *array is const
}

int main() {
  int x, y[10];

  f1( &x );
  f2( y );
}
```

A pointer to const int

Constant Pointer to Nonconstant Data

- Always points to the same memory location, and the data at that location can be modified through the pointer
- An array name is such a pointer
- A constant pointer must be initialized when it is declared

A const pointer to int

```
int x, y;
int*const ptr = &x;

*ptr = 7; // ALLOWED: *ptr is not const
ptr = &y; // ERROR: ptr is const
```

Constant Pointer to Constant Data

- Always point to the same memory location, and the data at that location cannot be modified through the pointer

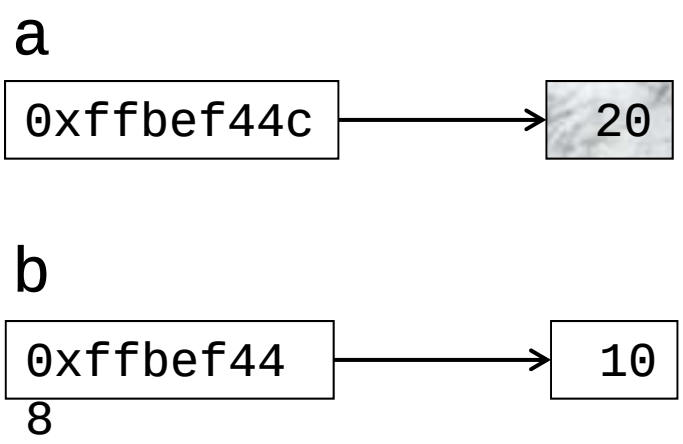
A const pointer to const int

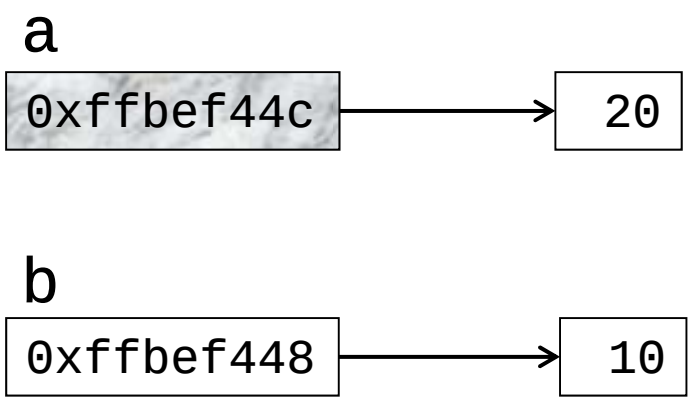
```
int x = 5, y;
const int*const ptr = &x;

*ptr = 7; // ERROR: *ptr is const
ptr = &y; // ERROR: ptr is const
```

Interview Question

```
const char* a = …;
char* b = …;
a = b; // ?
b = a; // ?
```

```
char*const a = …;
char* b = …;
a = b; // ?
b = a; // ?
```





Outline

- Introduction to Pointers
- Pointers vs References
- `sizeof` Operator
- Using `const` with Pointers
- Pointer Arithmetic and Arrays
- Arrays of Pointers
- Dynamic Memory Management
- Function Pointers

Relationship between Pointers and Arrays

- An array name can be thought of as a constant pointer
  - The array name (without a subscript) is a (constant) pointer to the first element of the array
- Pointers can be used to do any operation involving array subscripting

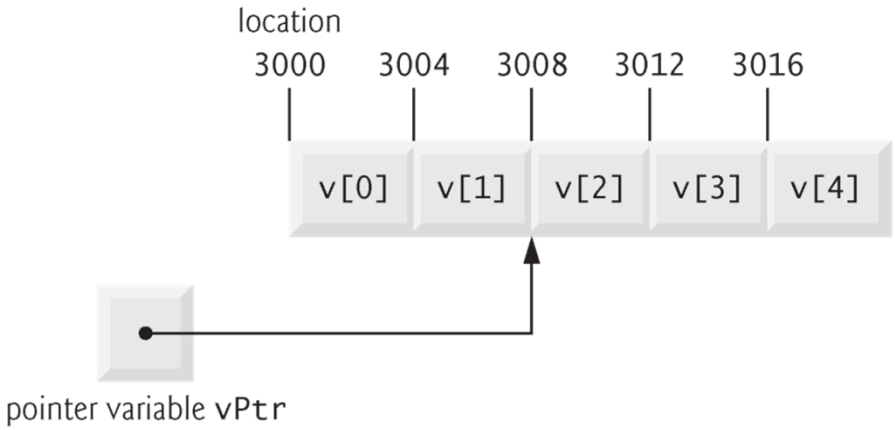
```
int b[ 5 ];
int *bPtr;
bPtr = b;          // b is a const pointer

bPtr = &b[ 0 ]; // equivalent to “bPtr = b”
bPtr[3] = 10;   // equivalent to “b[3] = 10”
```

Pointer Arithmetic

- Pointers are valid operands in arithmetic expressions, assignment expressions, and comparison expressions
- Certain arithmetic operations may be performed on pointers:
  - increment (++)
  - decremented (--)
  - an integer may be added to a pointer (+ or +=)
  - an integer may be subtracted from a pointer (- or -=)
  - one pointer may be subtracted from another of the same type

Pointer Arithmetic: An Example

```
int v[5];
int *vPtr = v;
vptr = &v[0];

vPtr += 2;
v += 2; // ERROR

v[2] = 10;
*(v + 2) = 10;
*(vPtr + 2) = 10;
```



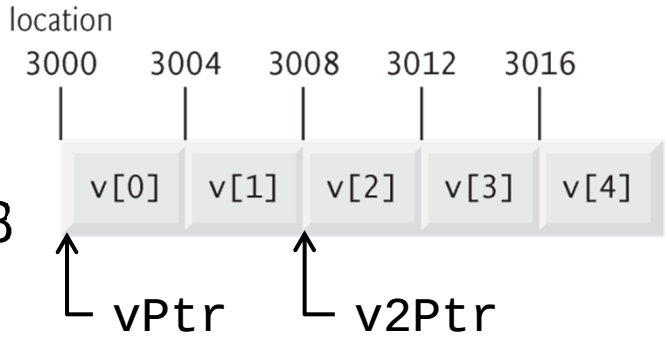
- Integer arithmetic
  - `3000 + 2 = 3002`
- Pointer arithmetic
  - `3000 + 2 * sizeof(int)`

Machine dependent

- Using pointer arithmetic to move a pointer outside the bounds of an array is a logic error

Pointer Arithmetic & Arrays

- Pointer variables pointing to the same array may be subtracted from one another
  - Suppose
    - `vPtr` contains the address 3000
    - `v2Ptr` contains the address 3008
  - Then




```
x = v2Ptr – vPtr;
// x = (3008-3000)/sizeof(int)
// number of elements from vPtr to v2Ptr
```

- Pointer arithmetic is meaningless unless performed on a pointer that points to an array

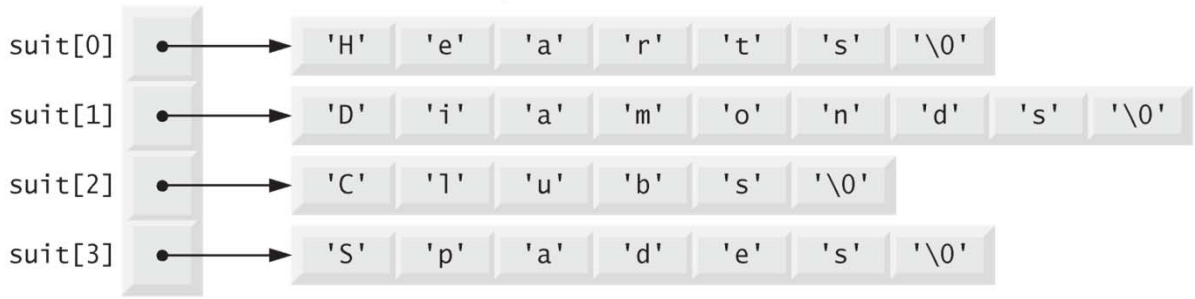
Outline

- Introduction to Pointers
- Pointers vs References
- `sizeof` Operator
- Using `const` with Pointers
- Pointer Arithmetic and Arrays
- Arrays of Pointers
- Dynamic Memory Management
- Function Pointers

Arrays of Pointers

- Arrays may contain pointers
- Most natural motivation: Arrays of C strings

```
const char *suit[4] = {
    "Hearts",
    "Diamonds",
    "Clubs",
    "Spades"
};
```



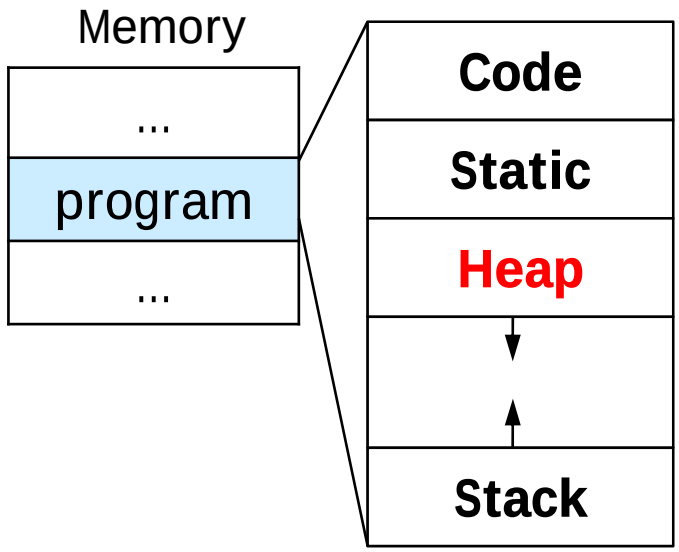
  - An array of four elements
  - Each element is of type "pointer to `const char` data"
- Useful for command line arguments

Outline

- Introduction to Pointers
- Pointers vs References
- `sizeof` Operator
- Using `const` with Pointers
- Pointer Arithmetic and Arrays
- Arrays of Pointers
- Dynamic Memory Management
- Function Pointers

Use of Pointers

- Provide the power of indirect addressing
- Provide a way to manage dynamic memory
  - A pointer can be used to access a location in the area where storage is dynamically created (usually called a *heap*)
  - Chapter 11.8



Dynamic Memory Management

- Run-time allocation/deallocation on heap
  - C: `malloc()` and `free()`
  - C++: `new` and `delete` (Chapter 11.8)

```
pointer_var = new data_type;
pointer_var = new data_type (initial_value);
delete pointer_var;
```

```
float *fpt = new float(0.0);
if ( !fpt ) {
  cout << “Memory allocation error.”;
  exit(1);
}
*fpt=3.45; //Uses pointer to access memory
cout << *fpt;
delete fpt;
```

Dynamic Arrays

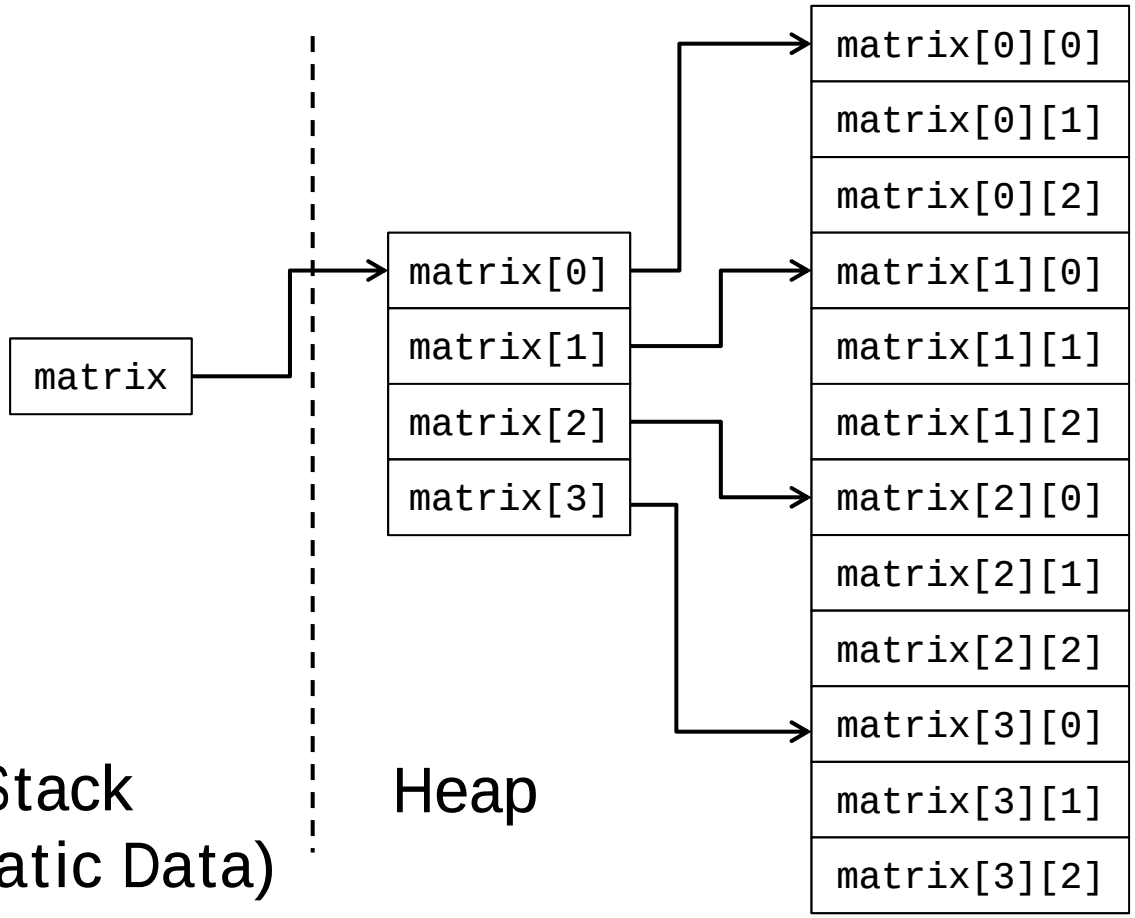
- May have a variable size, run-time allocation
  - Using `new` and `delete`

```
int size;
int *array = new int[size];
delete [] array;
```

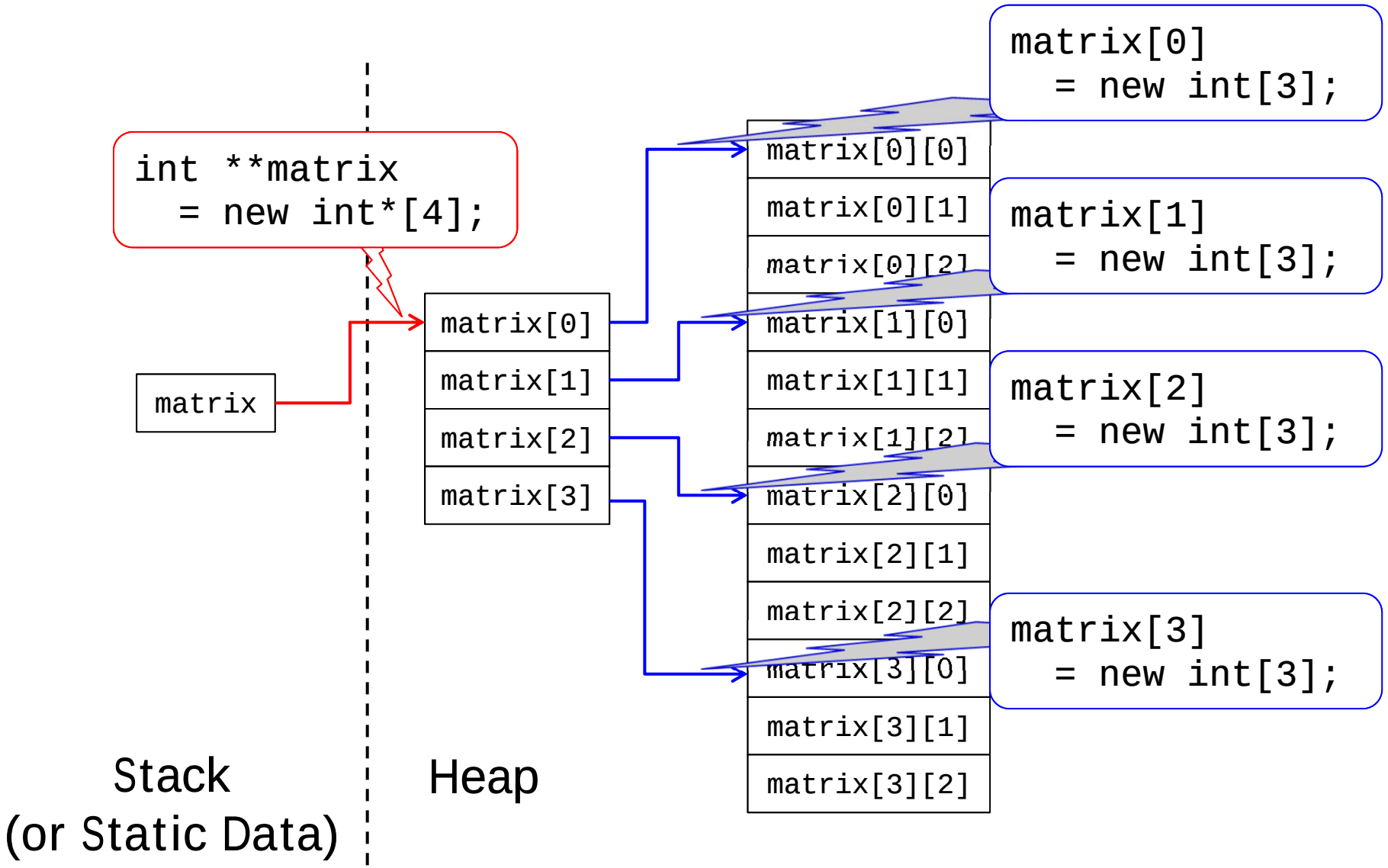
```
int rows, columns;
int **matrix = new int*[rows];
if (!matrix)
  cout << “Memory allocation error!” << endl;
for (int i = 0; i < rows; i++) {
  matrix[i] = new int[columns];
  if (!matrix[i])
    cout << “Memory allocation error!” << endl;
}
```

Multidimensional Array with Dynamic Sizes

- Suppose we want to create an `m` x `n` array. For example, a 4 x 3 array
- `matrix` is a pointer to a 4-pointer array, in which each element is a pointer to a 3-integer array.



Multidimensional Array with Dynamic Sizes

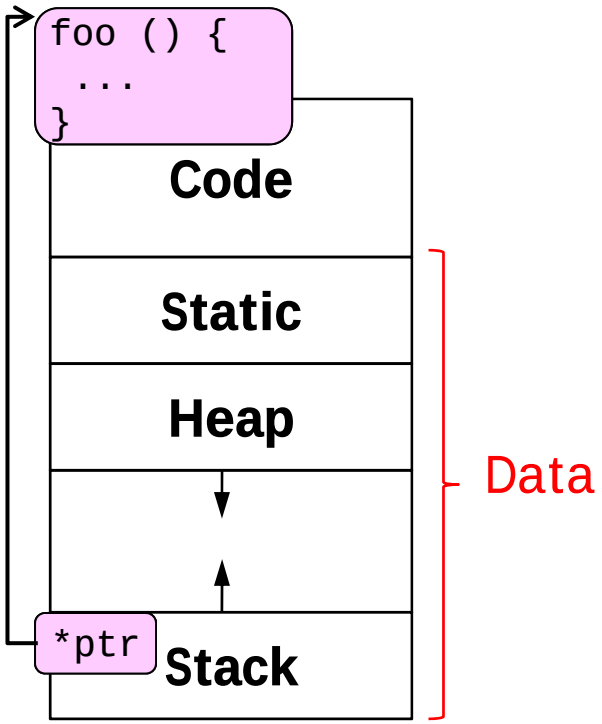


Outline

- Introduction to Pointers
- Pointers vs References
- `sizeof` Operator
- Using `const` with Pointers
- Pointer Arithmetic and Arrays
- Arrays of Pointers
- Dynamic Memory Management
- Function Pointers

Function Pointers

- A pointer to a function contains the function's address in memory
- A function name is actually a constant pointer that points to (the starting address of) the code of the function body
- Pointers to functions can be
  - Passed to functions
  - Returned from functions
  - Stored in arrays
  - Assigned to other function pointers
  - Used to call the underlying function



Function Pointers: An Example

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

void fun1(char *str) {
  cout << "From fun1: " << str << endl;
}
void fun2(char *str) {
  cout << "From fun2: " << str << endl;
}

int main() {
  void (*fn)(char *);
  fn = fun1;
  (*fn)( "Called first" );
  fn = fun2;
  (*fn)( "Called second" );

  return 0;
}
```

`void (*fn)(char *)` declares a pointer, named `fn`, that will reference a function that returns `void` and takes a `char` pointer

Return type and argument types must match

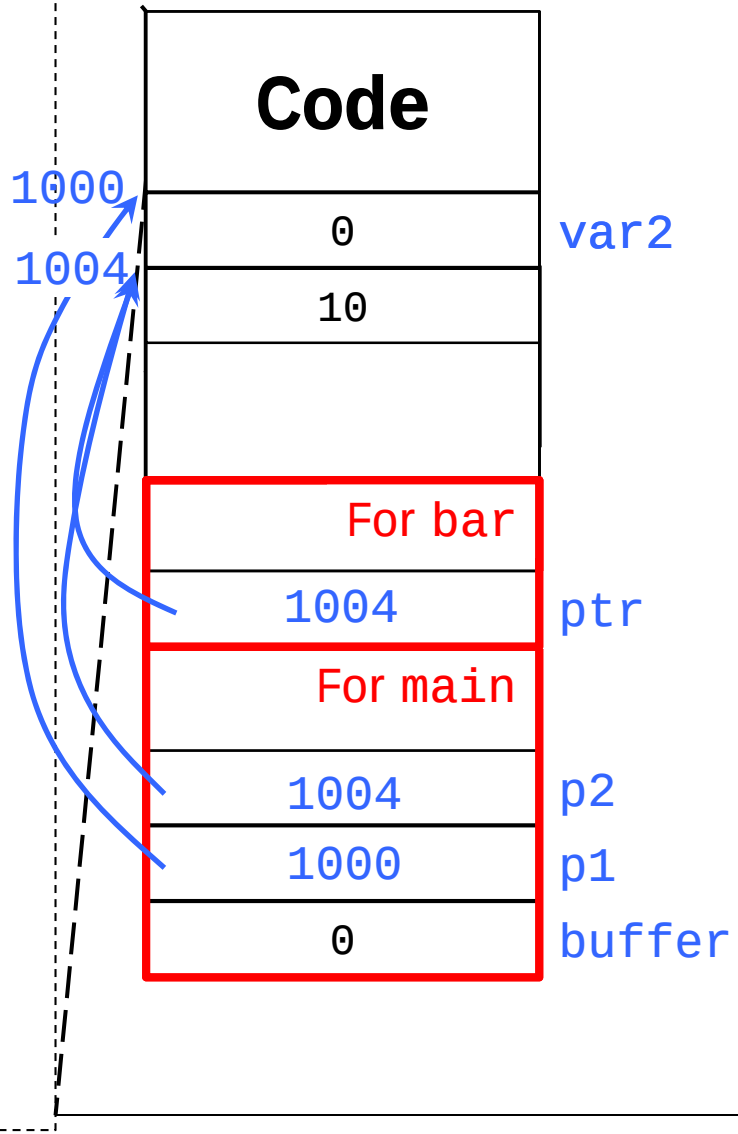
```
From fun1: Called first
From fun2: Called second
```

Review of Pointers

- A pointer variable contains merely a memory address
  - A pointer can point to the address of
    - A variable,
    - A pointer variable, or
    - A function
    - But not a reference (`int& *p`)
      - References have no memory location
- It is your (programmers') job to make sure the address a pointer points to is valid
- You have to know where/how a variable is allocated a block of memory

A Running Example

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

int* foo() {
  int var1 = 1;
  static int var2;
  return &var2;
}

int* bar() {
  int *ptr = new int(10);
  return ptr;
}

int main() {
  char *buffer = 0;
  cin >> buffer; // run-time error
  cout << buffer << endl;

  int *p1 = 0, *p2 = 0;
  p1 = foo();
  p2 = bar();

  return 0;
}
```



Another Running Example

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

void foo(int *ptr) {
  ptr = new int(1);
}

int main() {
  int *p = 0;
  foo(p);
  *p = 10; // run-time error

  return 0;
}
```