Machine Program: Data

Jinyang Li

Last lecture

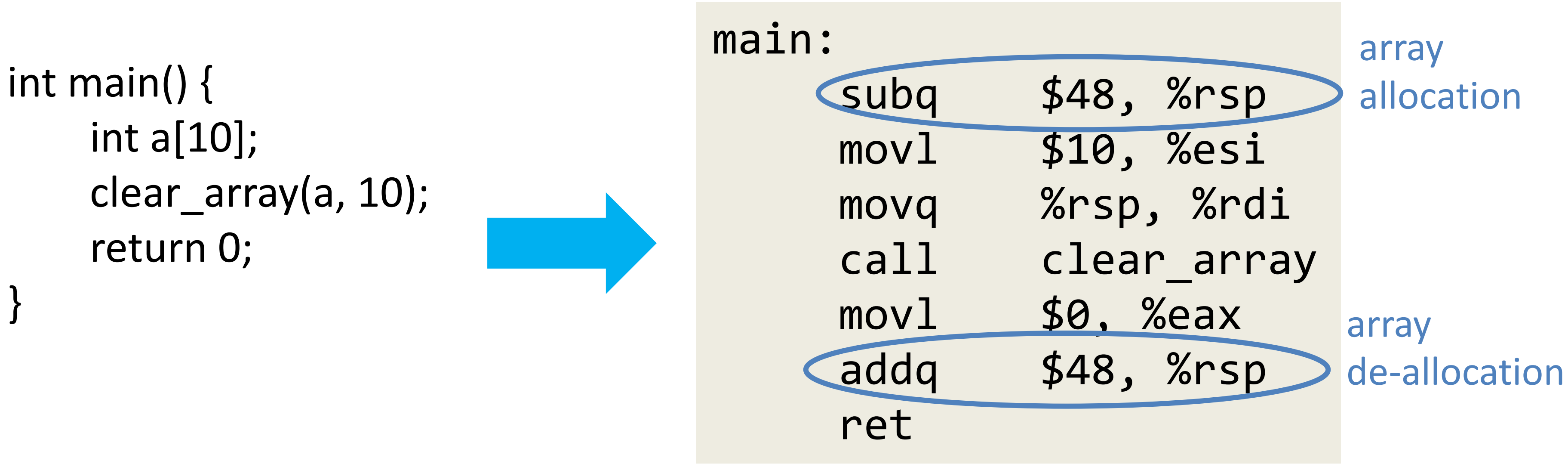
- How x86 supports procedure calls
  - call (pushes return address on stack; jump to function)
  - ret (pops return address from stack; jump to return address)
- C/UNIX calling convention (location of args/return val)
  - First 6 args are stored in regs: %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - Rest of arguments are stored on the stack
  - Return value (if there’s one) is stored in %rax
  - Caller vs callee save registers

Today's lesson plan

- Program data storage and manipulation
  - Local variable, global variable, dynamically-allocated storage
  - Arrays, 2D arrays, structs

Local variables

- For primitive data types, use registers whenever possible
- Allocate local array/struct variables on the stack



```
int main() {
        int a[10];
        clear_array(a, 10);
        return 0;
}
```

```
main:
        subq      $48, %rsp
        movl      $10, %esi
        movq      %rsp, %rdi
        call      clear_array
        movl      $0, %eax
        addq      $48, %rsp
        ret
```

array allocation

array de-allocation

Global variables

- Allocated in a memory region called “data” segment
  - Statically allocated; compiler determines each global variable’s location in data segment.

```
int count = 0;

void inc() {
  count++;
}

int main() {
    inc();
}
```

```
inc:
    addl $0x1, count(%rip)
    ret
main:
    ...
   call     add
   movl $0, %eax
   ...
```

Dynamically allocated space

- Allocated in a memory region called “heap”
  - Allocated by malloc library using sophisticated algorithms (discussed in later lecture)

```
int main() {
    int *x;
    x=malloc(100*sizeof(int));
    …
}
```

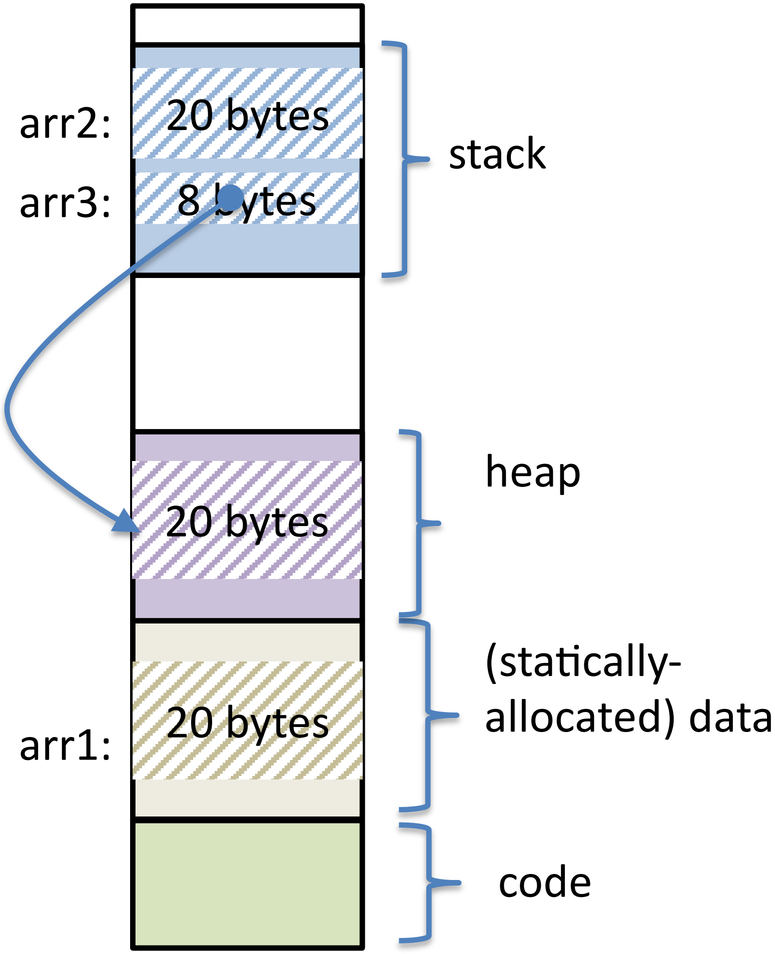
```
main:
    movl   $400 %edi
    call   malloc
    ...
```

Lots of code in this function

A process' memory regions

- A running program (process)'s memory consists of code, data, stack, heap (and code/data of its shared libraries)

```
int arr1[5];

int main() {

   int arr2[5];
   int *arr3;
   arr3 = malloc(sizeof(int)*5);
}
```

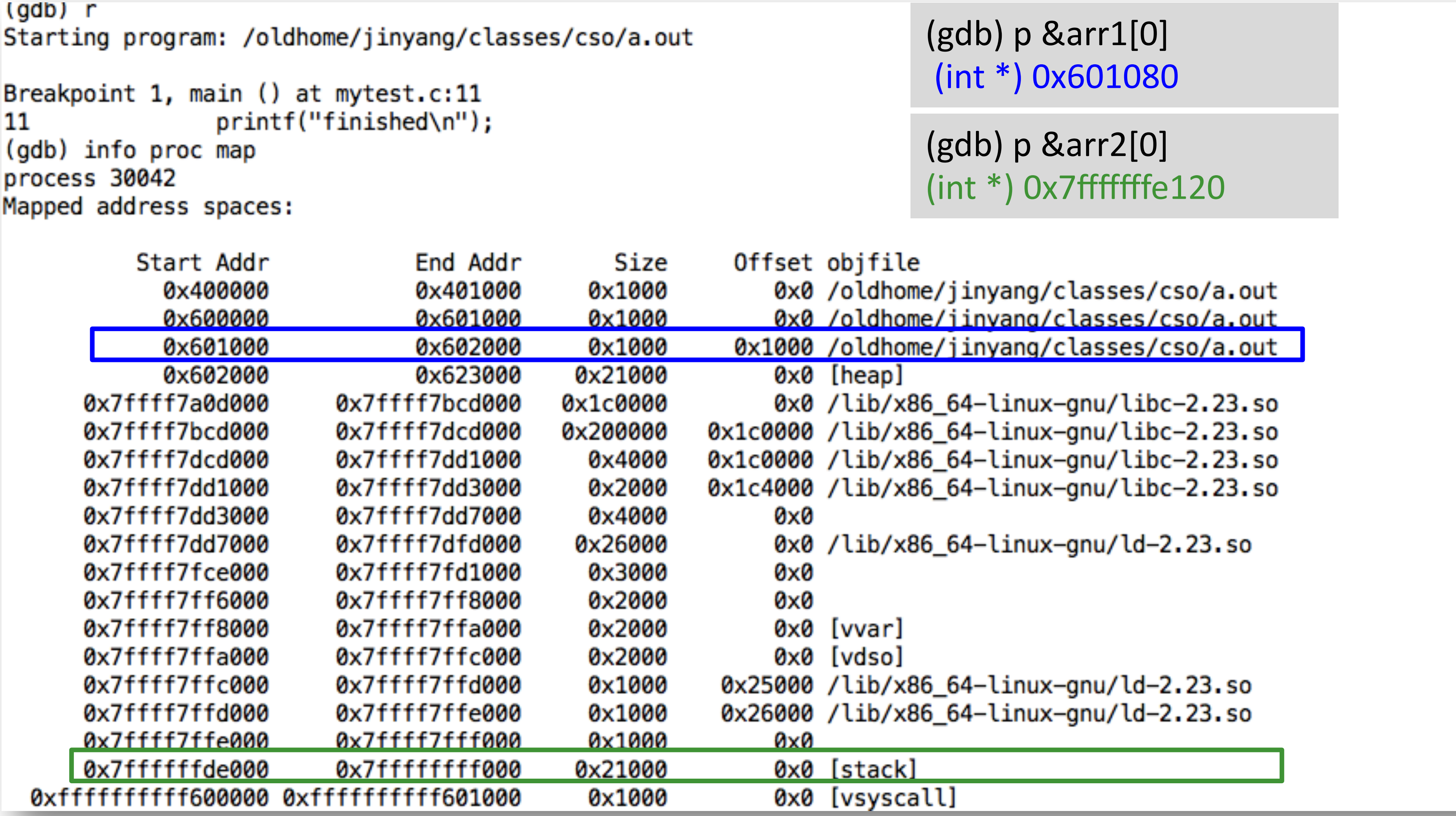


Data allocation

```
int arr1[5];
int main() {
   int arr2[5];
   int *arr3;
   arr3 = malloc(sizeof(int)*5);
}
```

(gdb) p &arr1[0]
 (int *) 0x601080

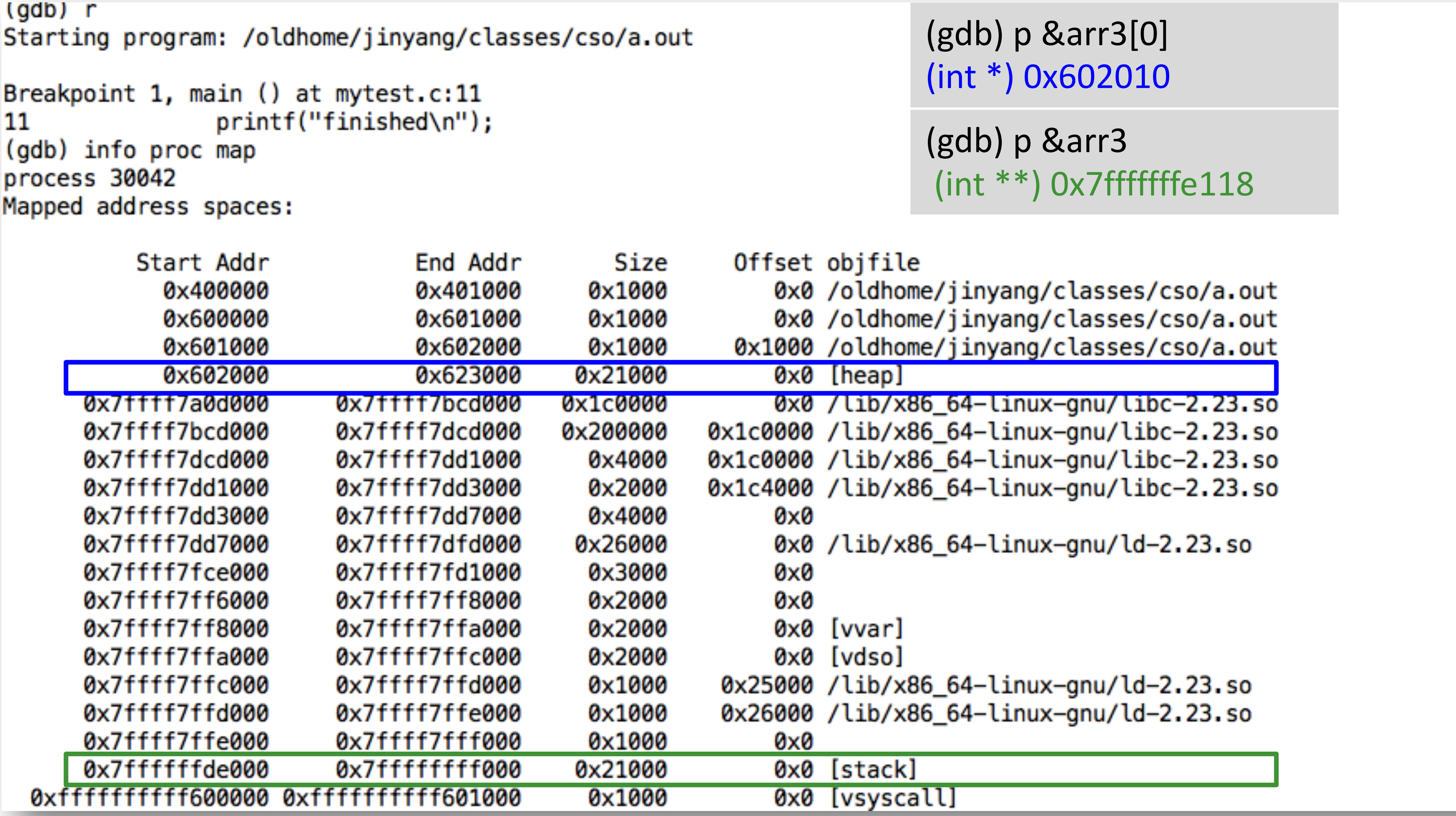
(gdb) p &arr2[0]
(int *) 0x7fffffffe120



```
(gdb) r
Starting program: /oldhome/jinyang/classes/cso/a.out

Breakpoint 1, main () at mytest.c:11
11              printf("finished\n");
(gdb) info proc map
process 30042
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /oldhome/jinyang/classes/cso/a.out
            0x600000           0x601000     0x1000        0x0 /oldhome/jinyang/classes/cso/a.out
            0x601000           0x602000     0x1000     0x1000 /oldhome/jinyang/classes/cso/a.out
            0x602000           0x623000    0x21000        0x0 [heap]
      0x7ffff7a0d000     0x7ffff7bcd000   0x1c0000        0x0 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7bcd000     0x7ffff7dcd000   0x200000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dcd000     0x7ffff7dd1000     0x4000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dd1000     0x7ffff7dd3000     0x2000   0x1c4000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dd3000     0x7ffff7dd7000     0x4000        0x0
      0x7ffff7dd7000     0x7ffff7dfd000    0x26000        0x0 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7fce000     0x7ffff7fd1000     0x3000        0x0
      0x7ffff7ff6000     0x7ffff7ff8000     0x2000        0x0
      0x7ffff7ff8000     0x7ffff7ffa000     0x2000        0x0 [vvar]
      0x7ffff7ffa000     0x7ffff7ffc000     0x2000        0x0 [vdso]
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x25000 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x26000 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
```

Data allocation

```
int arr1[5];
void main() {
   int arr2[5];
   int *arr3;
   arr3 = malloc(sizeof(int)*5);
}
```



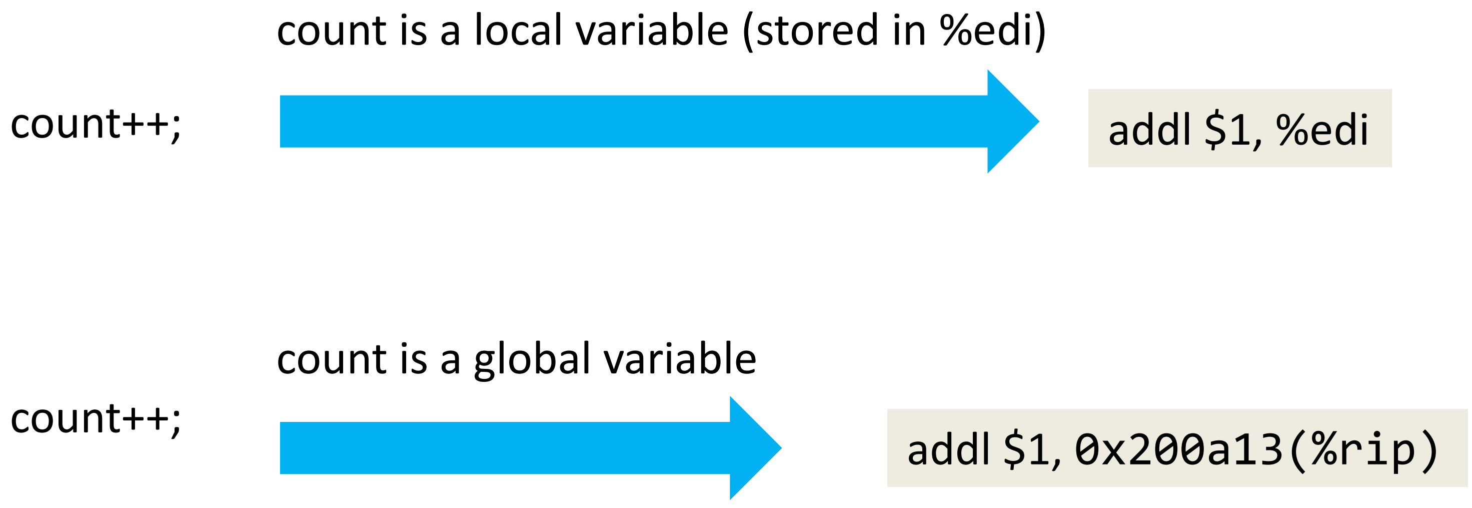
```
(gdb) r
Starting program: /oldhome/jinyang/classes/cso/a.out

Breakpoint 1, main () at mytest.c:11
11              printf("finished\n");
(gdb) info proc map
process 30042
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /oldhome/jinyang/classes/cso/a.out
            0x600000           0x601000     0x1000        0x0 /oldhome/jinyang/classes/cso/a.out
            0x601000           0x602000     0x1000     0x1000 /oldhome/jinyang/classes/cso/a.out
            0x602000           0x623000    0x21000        0x0 [heap]
      0x7ffff7a0d000     0x7ffff7bcd000   0x1c0000        0x0 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7bcd000     0x7ffff7dcd000   0x200000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dcd000     0x7ffff7dd1000     0x4000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dd1000     0x7ffff7dd3000     0x2000   0x1c4000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dd3000     0x7ffff7dd7000     0x4000        0x0
      0x7ffff7dd7000     0x7ffff7dfd000    0x26000        0x0 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7fce000     0x7ffff7fd1000     0x3000        0x0
      0x7ffff7ff6000     0x7ffff7ff8000     0x2000        0x0
      0x7ffff7ff8000     0x7ffff7ffa000     0x2000        0x0 [vvar]
      0x7ffff7ffa000     0x7ffff7ffc000     0x2000        0x0 [vdso]
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x25000 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x26000 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
```

```
(gdb) p &arr3[0]
(int *) 0x602010
```

```
(gdb) p &arr3
 (int **) 0x7fffffffe118
```

Accessing program data: primitive types

- Local variables of primitive data types are commonly stored in regs



count is a local variable (stored in %edi)

```
count++;
```

```
addl $1, %edi
```

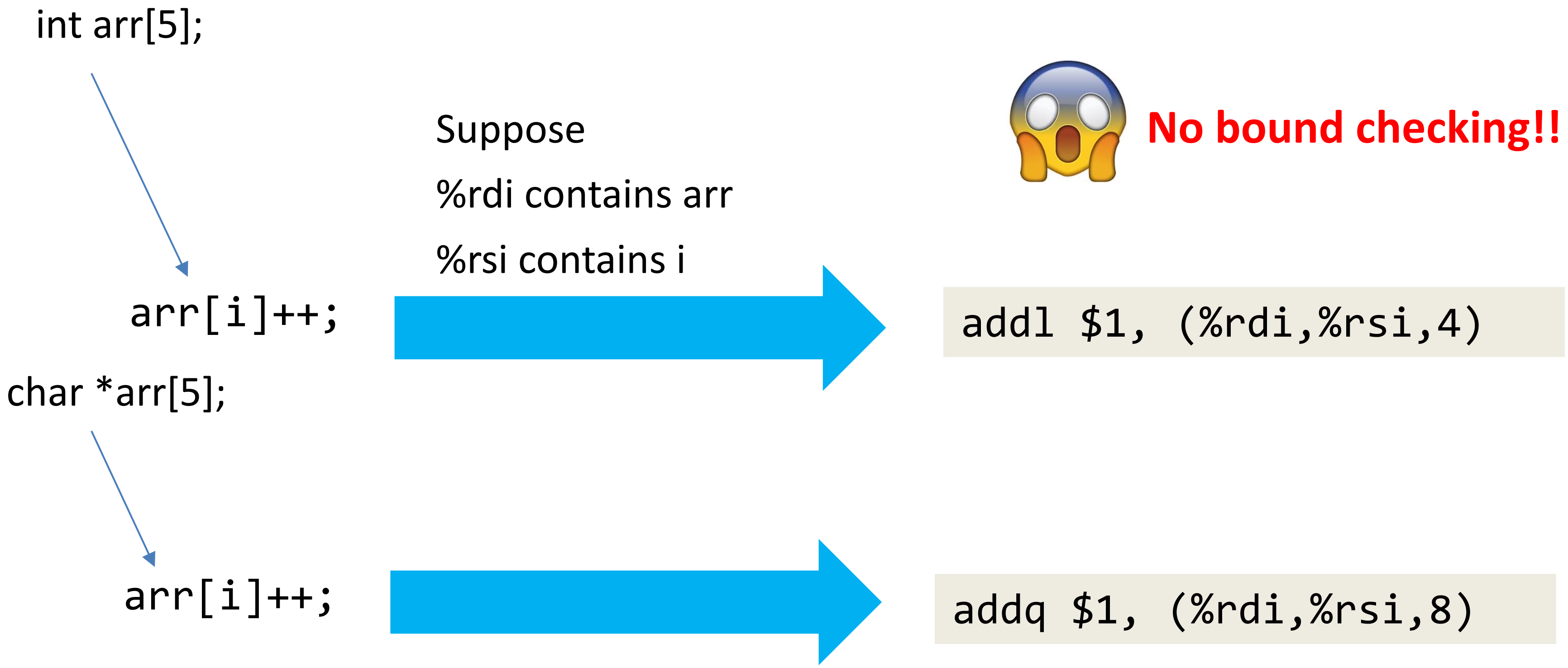
count is a global variable

```
count++;
```

```
addl $1, 0x200a13(%rip)
```

Accessing program data: arrays

- Arrays are always stored in the memory (stack, heap or data)



```
int arr[5];

arr[i]++;
```

Suppose

%rdi contains arr

%rsi contains i

```
addl $1, (%rdi,%rsi,4)
```

😱 **No bound checking!!**

```
char *arr[5];

arr[i]++;
```

```
addq $1, (%rdi,%rsi,8)
```

Binary Puzzle 1

```
void mystery(int *arr, int n) {
   ???
}
```

```
  movl $0, %eax
  jmp  .L3
.L4:
  movslq %eax, %rdx
  addl $1, (%rdi,%rdx,4)
  addl $1, %eax
.L3:
  cmpl %esi, %eax
  jl  .L4
  ret
```

%rdi has the value of arr
%esi has the value of n

Binary Puzzle 1

```
void mystery(int *arr, int n) {
    ???
}
```

```
   movl $0, %eax
   jmp   .L3
.L4:
   movslq %eax, %rdx
   addl $1, (%rdi,%rdx,4)
   addl $1, %eax
.L3:
   cmpl %esi, %eax
   jl   .L4
   ret
```

```
   a = 0;
   goto .L3
```

`%rdi` has the value of arr

`%esi` has the value of n

Binary Puzzle 1

```
void mystery(int *arr, int n) {
    ???
}
```

```
  movl $0, %eax
  jmp  .L3
.L4:
  movslq %eax, %rdx
  addl $1, (%rdi,%rdx,4)
  addl $1, %eax
.L3:
  cmpl %esi, %eax
  jl   .L4
  ret
```

```
  a = 0;
  goto .L3




.L3:
  if a < n
       goto .L4

  return
```

`%rdi` has the value of arr

`%esi` has the value of n

Binary Puzzle 1

```
void mystery(int *arr, int n) {
    ???
}
```

```
   movl $0, %eax
   jmp  .L3
.L4:
   movslq %eax, %rdx
   addl $1, (%rdi,%rdx,4)
   addl $1, %eax
.L3:
   cmpl %esi, %eax
   jl  .L4
   ret
```

```
   a = 0;
   goto .L3
.L4
   arr[a] = arr[a] + 1
   a++

.L3:
   if a < n
      goto .L4
   return
```

%rdi has the value of arr

%esi has the value of n

Binary Puzzle 1

type of a?

```
void mystery(int *arr, int n) {
   for(    int i = 0; i < n; i++)
   {
     arr[i] = arr[i] + 1;
   }
}
```

```
  movl $0, %eax
  jmp  .L3
.L4:
  movslq %eax, %rdx
  addl $1, (%rdi,%rdx,4)
  addl $1, %eax
.L3:
  cmpl %esi, %eax
  jl   .L4
  ret
```

```
  a = 0;
  goto .L3
.L4
  arr[a] = arr[a] + 1
  a++
.L3:
  if a < n
     goto .L4
  return
```

`%rdi` has the value of arr

`%esi` has the value of n

Binary puzzle 2

```
?? mystery(char *s) {

      ???
}
```

%rdi contains s

```
      movl    $0x0,%eax
      jmp    L1.
  L2.
      addl    $0x1,%eax
  L1.
      movslq %eax,%rdx
      cmpb   $0x0,(%rdi,%rdx,1)
     jne    L2.
      ret
```

Binary puzzle 2

```
?? mystery(char *s) {

      ???
}
```

%rdi contains s

```
     movl    $0x0,%eax
     jmp     L1.
 L2.
     addl    $0x1,%eax
 L1.
     movslq %eax,%rdx
     cmpb   $0x0,(%rdi,%rdx,1)
     jne     L2.
     ret
```

```
int a = 0;
goto L1;
```

Binary puzzle 2

```
?? mystery(char *s) {

        ???
}
```

%rdi contains s

```
     movl    $0x0,%eax
     jmp     L1.
L2.
     addl    $0x1,%eax
L1.
     movslq %eax,%rdx
     cmpb   $0x0,(%rdi,%rdx,1)
     jne     L2.
     ret
```

```
    int a = 0;
    goto L1;


L1.
    long d = a;
```

Binary puzzle 2

```
?? mystery(char *s) {

      ???
}
```

%rdi contains s

```
      movl    $0x0,%eax
      jmp     L1.
  L2.
      addl    $0x1,%eax
  L1.
      movslq %eax,%rdx
      cmpb   $0x0,(%rdi,%rdx,1)
      jne     L2.
      ret
```

```
    int a = 0;
    goto L1;


L1.
    long d = a;
    if(0 != s[d])
         goto L2;
```

Binary puzzle 2

```
?? mystery(char *s) {

      ???
}
```

%rdi contains s

```
     movl    $0x0,%eax
     jmp     L1.
 L2.
     addl    $0x1,%eax
 L1.
     movslq %eax,%rdx
     cmpb   $0x0,(%rdi,%rdx,1)
     jne     L2.
     ret
```

```
    int a = 0;
    goto L1;
L2.
    a = a  + 1;
L1.
    long d = a;
    if(0 != s[d])
       goto L2;
```

Binary puzzle 2

```
int mystery(char *s) {

    int a = 0;
    while(s[a]) {
        a = a + 1;
    }
    return a;
}
```

%rdi contains s

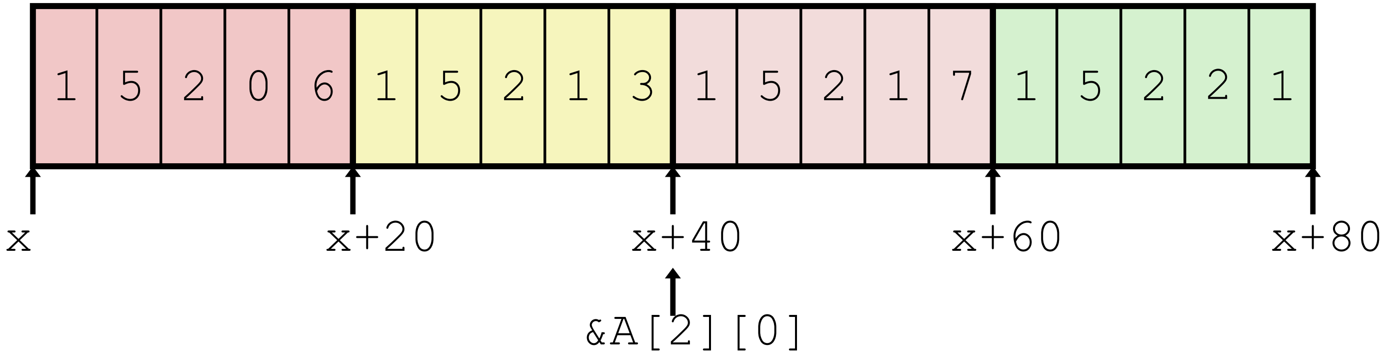
```
        movl    $0x0,%eax
        jmp     L1.
    L2.
        addl    $0x1,%eax
    L1.
        movslq %eax,%rdx
        cmpb   $0x0,(%rdi,%rdx,1)
        jne     L2.
        ret
```

```
    int a = 0;
    goto L1;
L2.
    a = a  + 1;
L1.
    long d = a;
    if(0 != s[d]) {
       goto L2;
    }
    ret;
```

2D arrays

```
int A[4][5] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

- “Row-Major” ordering of all elements in memory



1 5 2 0 6 1 5 2 1 3 1 5 2 1 7 1 5 2 2 1

x x+20 x+40 x+60 x+80

&A[2][0]

x+(i*COL+j)*sizeof(int)

2D Array Element Access

```
int getnum(int A[4][5], long i, long j) {
  return A[i][j];
}
```

%rdi contains A
%rsi contains i
%rdx contains j
%eax is to contain A[i]

```
leaq   (%rsi,%rsi,4), %rcx   # %rcx = 5*i
addq   %rdx, %rcx            # %rcx = 5*i+j
movl   (%rdi,%rcx,4), %eax   # %eax = *(int *)((char *)A+(5*i+j)*4)
```

```
leaq   (%rsi,%rsi,4), %rax   # %rax = 5*i
leaq   (%rdi,%rax,4), %rax   # %rax = (char *)A + 5*i*4
movl   (%rax,%rdx,4), %eax   # %eax = *(int *)(%rax+4*j)
```

Array of pointers

```
int getnum(int **A, long i, long j) {
  return A[i][j];
}
```

```
int main() {
   int a0[3] = {1, 2, 3};
   int a1[3] = {4, 5, 6};
   int *a[2] = {a0, a1};
   int n = getnum(a, 1, 2);
}
```

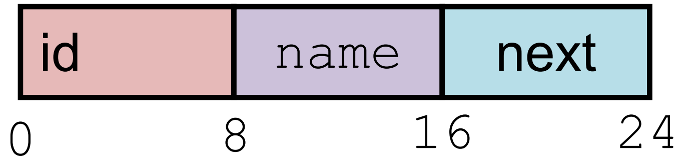
%rdi contains A
%rsi contains i
%rdx contains j
%eax is to contain A[i]

```
movq   (%rdi, %rsi, 8),  %rax      # %rax = *(int **)((char  *)A + i*8)
movl   (%rax, %rdx, 4),  %eax      # %eax = %rax + j*4
```

Accessing Program Data: struct

- Struct is stored in the memory
  - Fields are contiguous in the order they are declared in struct
  - There may be padding (gaps) between fields

```
typedef struct node {
     long id;
     char *name;
     struct node *next;
}node;
```



id
name
next
0
8
16
24

```
n->id = 10;
n->name = NULL;
n->next = n;
```

%rdi contains n

```
movq      $10, (%rdi)
movq      $0, 8(%rdi)
movq      %rdi, 16(%rdi)
```

Binary Puzzle 3

```
?? mystery(node *n, long id) {
   ???
}
```

```
  jmp     .L1
.L3:
  cmpq    %rsi, (%rdi)
  jne     .L2
  movq    8(%rdi), %rax
  ret
.L2:
  movq    16(%rdi), %rdi
.L1:
  testq   %rdi, %rdi
  jne     .L3
  movq    $0, %rax
  ret
```

%rdi has the value of n
%rsi has the value of id
%rax is to contain return value

Binary Puzzle 3

```
?? mystery(node *n, long id) {
   ???
}
```

```
  jmp     .L1
.L3:
  cmpq    %rsi, (%rdi)
  jne     .L2
  movq    8(%rdi), %rax
  ret
.L2:
  movq    16(%rdi), %rdi
.L1:
  testq   %rdi, %rdi
  jne     .L3
  movq    $0, %rax
  ret
```

```
goto .L1
```

%rdi has the value of n
%rsi has the value of id
%rax is to contain return value

Binary Puzzle 3

```
?? mystery(node *n, long id) {
   ???
}
```

```
  jmp     .L1
.L3:
  cmpq    %rsi, (%rdi)
  jne     .L2
  movq    8(%rdi), %rax
  ret
.L2:
  movq    16(%rdi), %rdi
.L1:
  testq   %rdi, %rdi
  jne     .L3
  movq    $0, %rax
  ret
```

```
goto .L1
```

```
.L1:
  if (n != 0)
     goto .L3
```

`%rdi` has the value of n

`%rsi` has the value of id

`%rax` is to contain return value

Binary Puzzle 3

```
?? mystery(node *n, long id) {
   ???
}
```

```
  jmp     .L1
.L3:
  cmpq    %rsi, (%rdi)
  jne     .L2
  movq    8(%rdi), %rax
  ret
.L2:
  movq    16(%rdi), %rdi
.L1:
  testq   %rdi, %rdi
  jne     .L3
  movq    $0, %rax
  ret
```

```
  goto .L1

.L1:
  if (n != 0)
     goto .L3
  return 0;
```

%rdi has the value of n

%rsi has the value of id

%rax is to contain return value

Binary Puzzle 3

```
?? mystery(node *n, long id) {
   ???
}
```

```
  jmp      .L1
.L3:
  cmpq     %rsi, (%rdi)
  jne      .L2
  movq     8(%rdi), %rax
  ret
.L2:
  movq     16(%rdi), %rdi
.L1:
  testq    %rdi, %rdi
  jne      .L3
  movq     $0, %rax
  ret
```

%rdi has the value of n

%rsi has the value of id

%rax is to contain return value

n->id != id

```
  goto .L1
.L3:
  if (~~*((long *)n) != id)~~
     goto .L2




.L1:
  if (n != 0)
     goto .L3
  return 0;
```

Binary Puzzle 3

```
?? mystery(node *n, long id) {
   ???
}
```

```
  jmp     .L1
.L3:
  cmpq    %rsi, (%rdi)
  jne     .L2
  movq    8(%rdi), %rax
  ret
.L2:
  movq    16(%rdi), %rdi
.L1:
  testq   %rdi, %rdi
  jne     .L3
  movq    $0, %rax
  ret
```

```
  goto .L1;
.L3:
  if (n->id != id)
     goto .L2;
  return n->name;



.L1:
  if (n != 0)
     goto .L3;
  return 0;
```

%rdi has the value of n
%rsi has the value of id
%rax is to contain return value

Binary Puzzle 3

```
?? mystery(node *n, long id) {
   ???
}
```

```
  jmp     .L1
.L3:
  cmpq    %rsi, (%rdi)
  jne     .L2
  movq    8(%rdi), %rax
  ret
.L2:
  movq    16(%rdi), %rdi
.L1:
  testq   %rdi, %rdi
  jne     .L3
  movq    $0, %rax
  ret
```

```
  goto .L1;
.L3:
  if (n->id != id)
     goto .L2;

   return n->name;
.L2
    n = n->next;

.L1:
  if (n != 0)
     goto .L3;
  return 0;
```

%rdi has the value of n

%rsi has the value of id

%rax is to contain return value

Binary Puzzle 3

```
char *mystery(node *n, long id) {
   while (n) {
      if (n->id == id)
         return n->name;
      n= n->next;
   }
   return NULL;
}
```

```
  jmp     .L1
.L3:
  cmpq    %rsi, (%rdi)
  jne     .L2
  movq    8(%rdi), %rax
  ret
.L2:
  movq    16(%rdi), %rdi
.L1:
  testq   %rdi, %rdi
  jne     .L3
  movq    $0, %rax
  ret
```

```
  goto .L1;
.L3:
  if (n->id != id)
     goto .L2;

   return n->name;
.L2
   n = n->next;
.L1:
  if (n != 0)
     goto .L3;
  return 0;
```

Summary

- How program data is stored and accessed
  - Primitive data types
  - Arrays
  - Structs
- Separate memory regions for stack, heap, data