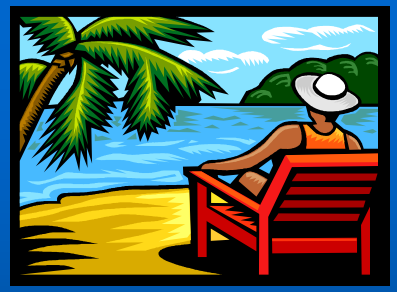
CS-034
Getting Started in C



Thomas Doeppner
Pascal Van Hentenryck

A C Program

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

- C Programs always starts in a main function
- main always return an error code
  - think how many things went wrong
- C programs consist of functions

How to use it?

```
FlyingComet /Users/pvh/courses/CS034 [51] -> ls
main.c
FlyingComet /Users/pvh/courses/CS034 [52] -> cat main.c
int main() {
   return 0;
}

FlyingComet /Users/pvh/courses/CS034 [53] -> cc main.c
FlyingComet /Users/pvh/courses/CS034 [54] -> ls
a.out*  main.c
```

How to use it?

```
FlyingComet /Users/pvh/courses/CS034 [57] -> ls
main.c
FlyingComet /Users/pvh/courses/CS034 [58] -> cc main.c -o main.o
FlyingComet /Users/pvh/courses/CS034 [59] -> ls
main.c  main.o*
FlyingComet /Users/pvh/courses/CS034 [60] -> main.o
FlyingComet /Users/pvh/courses/CS034 [60] ->
```

A C Program

```
int main() {
 printf("Welcome to CS-034\n");
 return 0;
}
```

- `printf` has a lot more functionality

How to use it?

```
FlyingComet /Users/pvh/courses/CS034 [62] -> cc main.c -o main.o
FlyingComet /Users/pvh/courses/CS034 [63] -> main.o
Welcome to CS-034
FlyingComet /Users/pvh/courses/CS034 [64] ->
```

Declarations in C

```
int main() {
 int i;
 float f;
 char c;
 return 0;
}
```

- Types are promises
  - promises can be broken
- Types specify memory sizes
  - that cannot be broken
- Types versus Types
  - strong typing (Java, Pascal (almost))
  - abstract types
  - concrete types
    - representations

Declarations in C

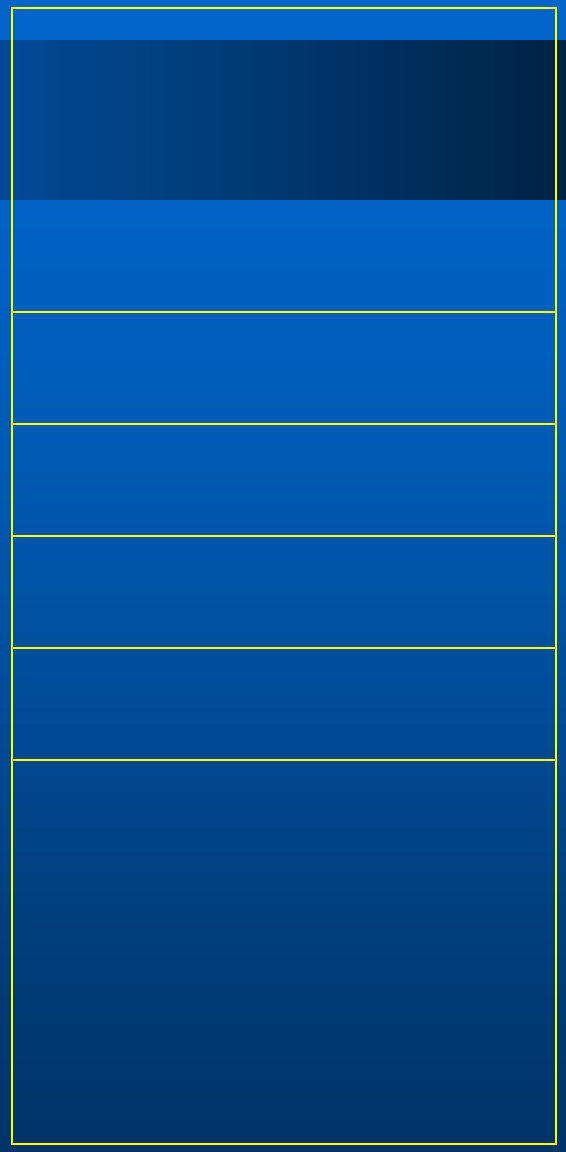
```
int main() {
 int i;
 float f;
 char c;
 return 0;
}
```

Declarations reserve memory space
- where?

The variables are uninitialized
- junk
- whatever was there before

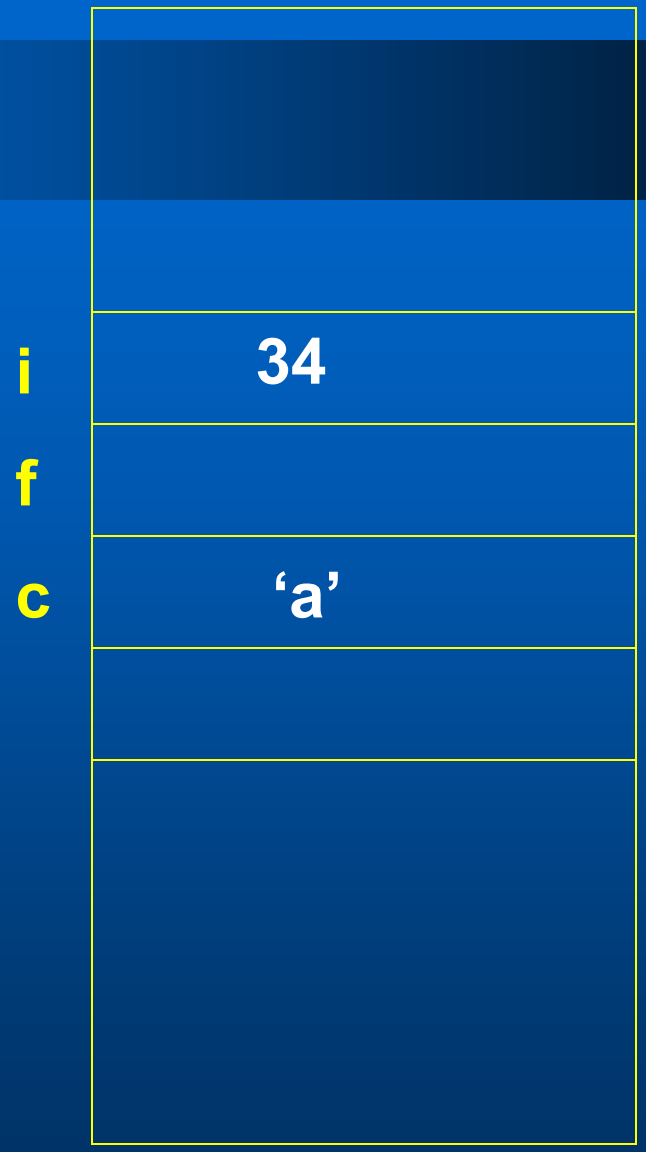
Declarations in C

```
int main() {
 int i;
 float f;
 char c;
 return 0;
}
```



Using Variables

```
int main() {
    int i;
    float f;
    char c;
    i = 34;
    c = 'a';
}
```



printf again

```
int main() {
    int i;
    float f;
    char c;
    i = 34;
    c = 'a';
    printf("%d\n",i);
    printf("%d\t%c\n",i,c);
}
```

```
FlyingComet  [39] -> a.out
34
34      a
```

printf again

```
int main() {
    …
    printf("%d\t%c\n",i,c);
}
```

```
FlyingComet  [39] -> a.out
34
34      a
```

Two parts

- formating instructions
- arguments

printf again

```
int main() {
    …
    printf("%d\t%c\n",i,c);
}
```

```
FlyingComet  [39] -> a.out
34
34      a
```

Formating instructions

- Special characters
  - \n : newline
  - \t : tab
  - \b : backspace
  - \" : double quote
  - \\ : backslash

printf again

```
int main() {
    …
    printf("%d\t%c",i,c);
}
```

```
FlyingComet  [39] -> a.out
34
34      a
```

Formating instructions

- Types of arguments
  - %d: integers
  - %f: floating-point numbers
  - %c: for characters

printf again

```
int main() {
    …
    printf("%6d%3c",i,c);
}
```

```
FlyingComet  [39] -> a.out
   34  a
```

Formating instructions

- %6d: decimal integers at least 6 characters wide
- %6f: floating point at least 6 characters wide
- %6.2f: floating point at least 6 wide, 2 after the decimal point

printf again

```
int main() {
   int i;
   float celsius;
   for(i=30;i<34;i++) {
     celsius = (5.0/9.0)*(i-32.0);
     printf("%3d %6.1f\n",i,celsius);
   }
}
```

```
[56] -> a.out
30   -1.1
31   -0.6
32    0.0
33    0.6
```

Some Primitive Data Types

int
- integer: 16 bits or 32 bits (implementation dependent)

long
- integer: 32 bits

char
- a single byte

float
- single-precision floating point

double
- double-precision floating point

What is the size of my int?

```
int main() {
   int i;
   printf("%d\n",sizeof(i));
}
```

```
FlyingComet /Users/pvh/courses/cs034 [98] -> a.out
4
```

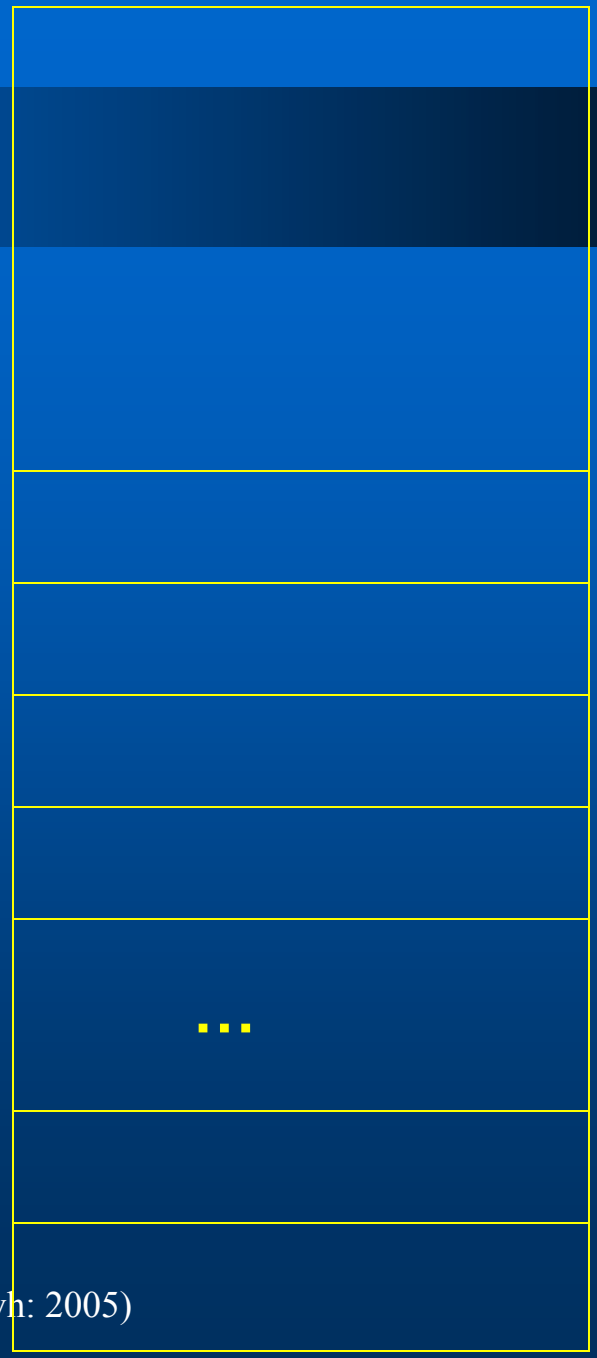
sizeof

- return the size of a variable in bytes
- Very very very very very very important function in C

Arrays

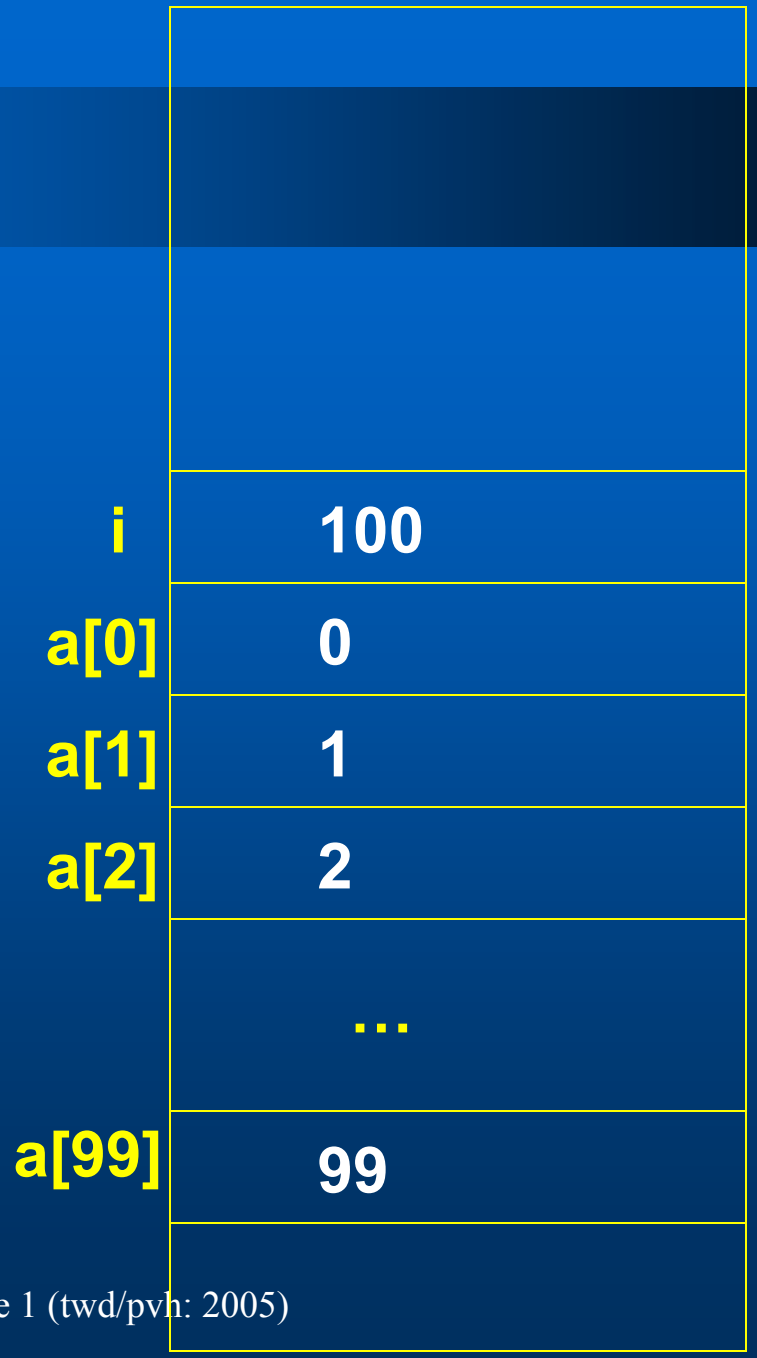
```
int main() {
    int i;
    int a[100];
}
```

The size of the array must be a constant!



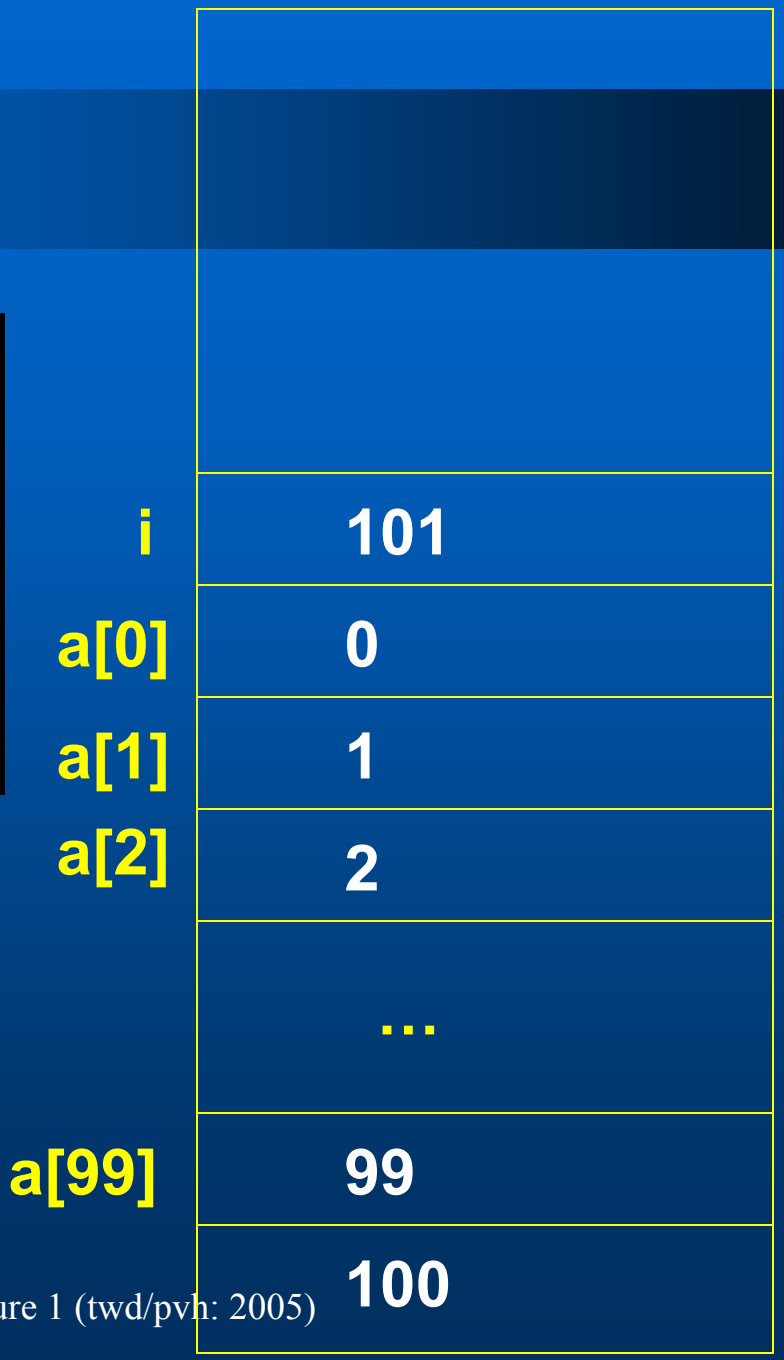
Arrays

```
int main() {
    int i;
    int a[100];
    for(i=0;i<100;i++)
      a[i] = i;
}
```



Array Bounds

```
int main() {
    int i;
    int a[100];
    for(i=0;i<=100;i++)
      a[i] = i;
}
```



Arrays in C

C Arrays = Storage + Indexing

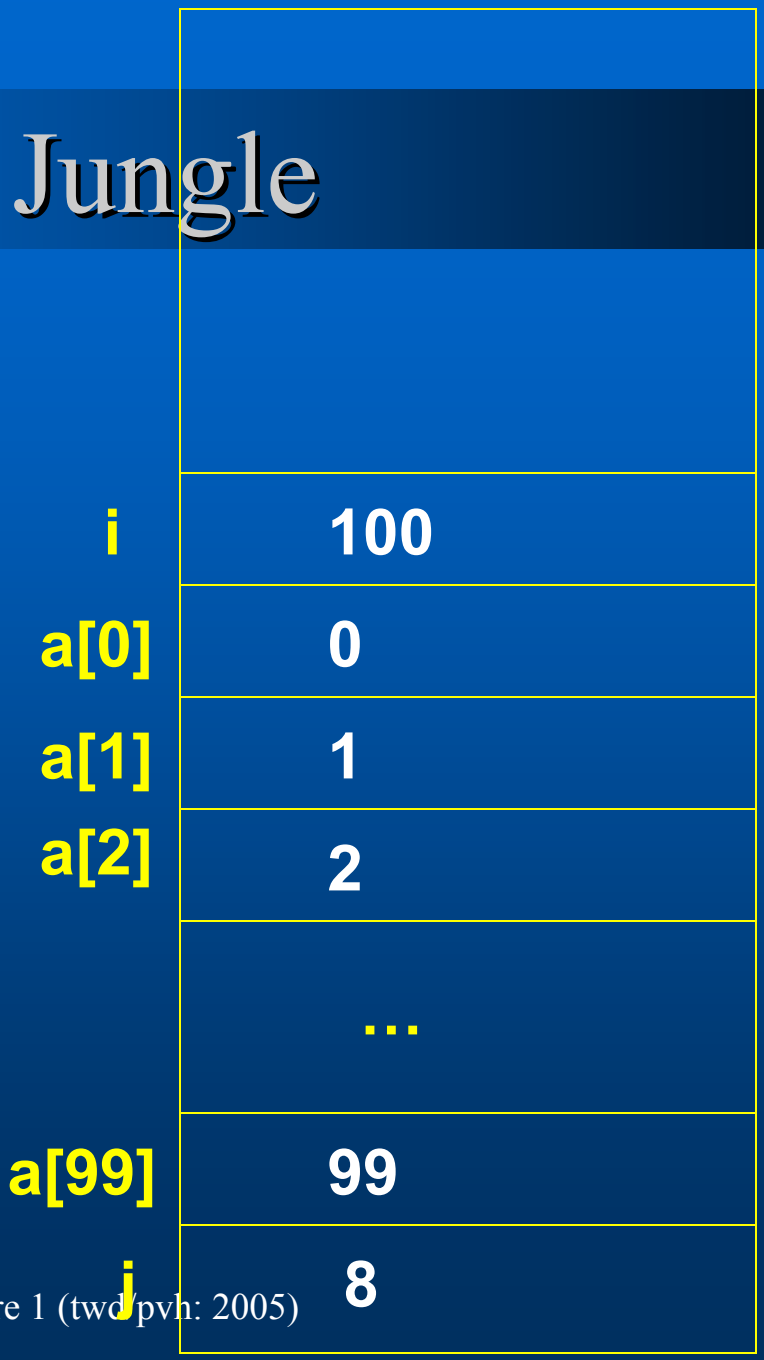
- no bound checking
- no size
- no initialization

Welcome to the jungle

Welcome to the Jungle

```
int main() {
    int i;
    int a[100];
    int j;
    j = 8;
    for(i=0;i<100;i++)
      a[i] = i;
    printf("%d\n",j);
}
```

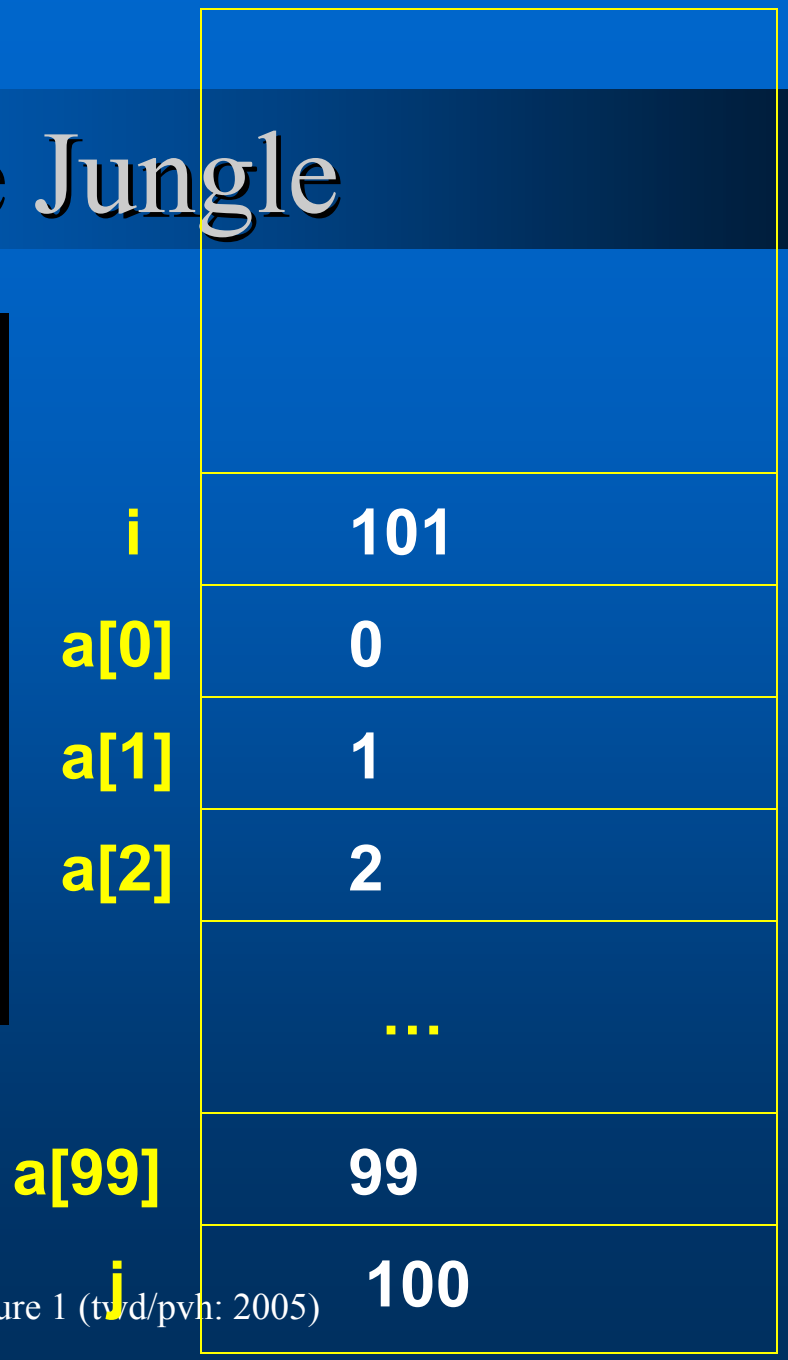
```
[61] -> a.out
8
```



Welcome to the Jungle

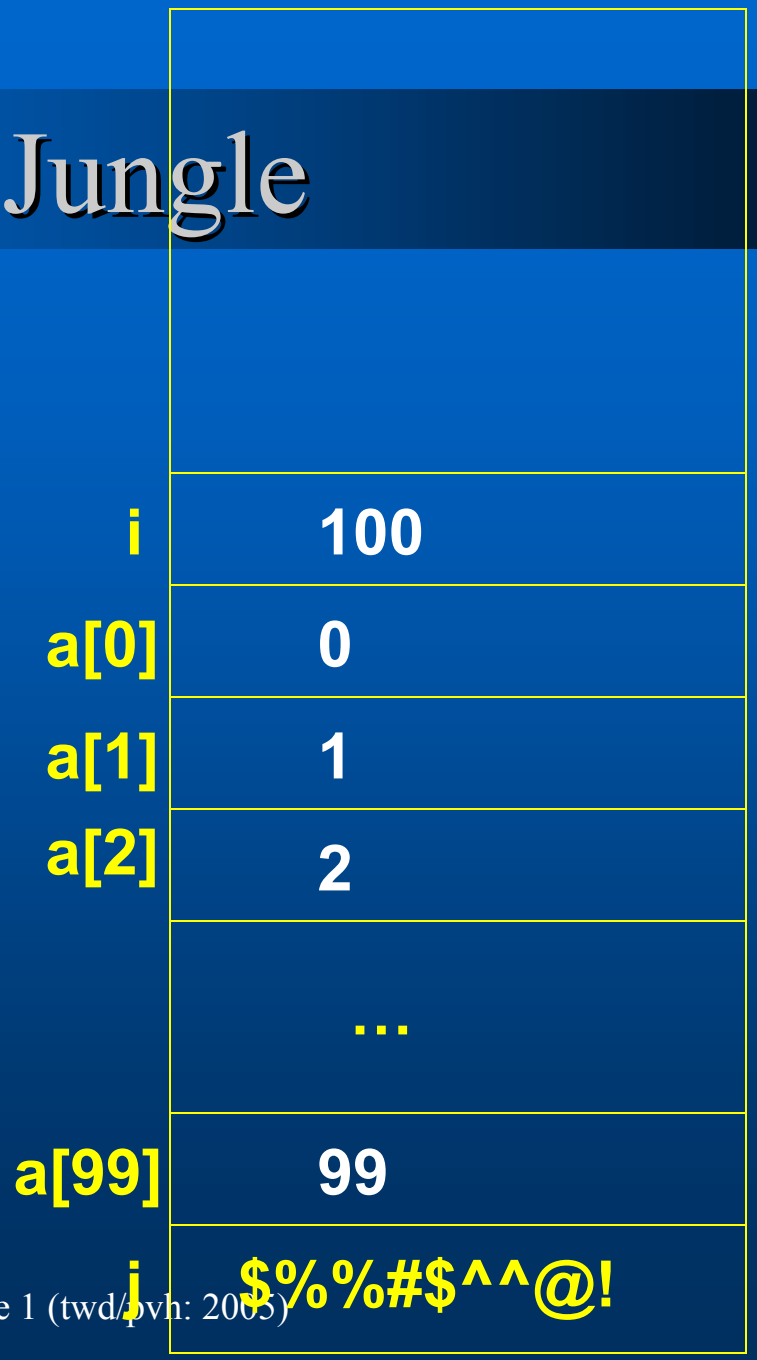
```
int main() {
    int i;
    int a[100];
    int j;
    j = 8;
    for(i=0;i<=100;i++)
      a[i] = i;
    printf("%d\n",j);
}
```

```
[61] -> a.out
100
```



Welcome to the Jungle

```
int main() {
    int i;
    int j;
    int a[100];
    for(i=0;i<100;i++)
      a[i] = i;
    printf("%d\n",j);
}
```




```
[61] -> a.out
-1880816380
```

Welcome to the Jungle

```
int main() {
   int i;
   int a[100];
   a[-3] = 25;
   printf("%d\n",a[-3]);
}
```

```
 [56] -> a.out
25
```

Welcome to the Jungle

```
int main() {
   int i;
   int a[100];
   a[-3] = 25;
   a[11111111] = 6;
   printf("%d\n",a[-3]);
}
```

```
[56] -> a.out
Segmentation fault
```

What is a segmentation fault?

- **access to a memory location outside its address space**

For loops

Before the loop

```
int main() {
   int i;
   float celsius;
   for( i=30 ; i<34 ; i = i+1) {
     celsius = (5.0/9.0)*(i-32.0);
     printf("%3d %6.1f\n",i,celsius);
   }
}
```

after each iteration

Function Definitions

```
int fact(int i) {
   int k;
   int res;
   for(res=1,k=1; k<=i; k++)
      res = res * k;
   return res;
}

int main() {
   printf("%d\n",fact(5));
   return 0;
}
```

main

- is just another function
- starts the program

All functions

- have a return type

Function Definitions

```
float fact(int i) {
   int k;
   float res;
   for(res=1,k=1; k<=i; k++)
     res = res * k;
   return res;
}

int main() {
   printf("%f\n",fact(5));
   return 0;
}
```

All functions

- **have a return type**

Function Definitions

```
int main() {
  printf("%d\n",fact(5));
  return 0;
}
float fact(int i) {
  int k;
  float res;
  for(res=1,k=1; k<=i; k++)
    res = res * k;
  return res;
}
```

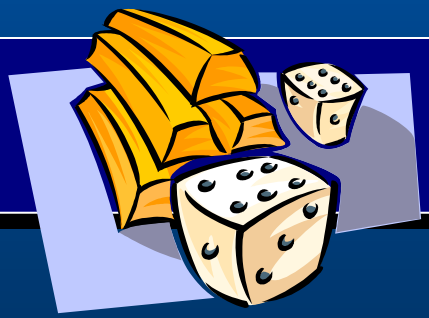
All functions

- **have a return type**

Function Definitions

```
FlyingComet /Users/pvh/courses/cs034 [88] -> cc main.c
main.c:27: warning: type mismatch with previous implicit declaration
main.c:23: warning: previous implicit declaration of `fact'
main.c:27: warning: `fact' was previously implicitly declared to return
`int'
```

```
FlyingComet /Users/pvh/courses/cs034 [88] -> a.out
1079902208
```



Function Declarations

```
float fact(int i);

int main() {
  printf("%f\n",fact(5));
  return 0;
}
float fact(int i) {
```

Declares the function

```
FlyingComet /Users/pvh/courses/cs034 [91] -> a.out
120.000000
```

```
  for(res=0,k=1; k<=i; k++)
    res = res * k;
  return res;
}
```

Function Declarations

fact.h

```
float fact(int i);
```

main.h

```
#include “fact.h”
int main() {
  printf("%f\n",fact(5));
  return 0;
}
float fact(int i) {
  int k; float res;
  for(res=1,k=1; k<=i; k++)
    res = res * k;
  return res;
}
```

My other Precious: assert

```
#include <assert.h>
float fact(int i) {
  assert(i >= 0);
  int k; float res;
  for(res=1,k=1; k<=i; k++)
    res = res * k;
  return res;
  return 0;
}
```

assert

- **verify if the assertion holds**
- **terminates otherwise**

```
FlyingComet /Users/pvh/courses/cs034 [96] -> a.out
main.c:30: failed assertion `i >= 0'
Abort
```

The Preprocessor

`#include`

- calls the preprocessor to include the file

What do you include?

- your own file: **`#include “fact.h”`**
  - **`~ start in the current directory`**
- standard .h file: **`#include <assert.h>`**
  - **`~ start the search in a standard place`**

#define

```
#define SIZE 100
int main() {
    int i;
    int a[SIZE];
}
```

#define

- do not take memory space
- are replaced in the program by the preprocessor
- remember MIPS?

#define

```
#define forever for(;;)
int main() {
    int i;
    forever {
      printf("hello world\n");
   }
}
```

Parameter passing

Passing arrays to function

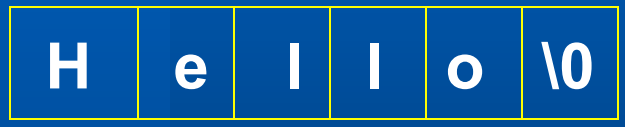
```
int average(int a[],int s) {
   int i;int sum;
   for(i=0,sum=0;i<s;i++)
     sum += a[i];
   return sum/s;
}
int main() {
   int a[100];
   …
   printf("%d\n",average(a,100));
}
```

- **Note that I need to pass the size of the array**
- **This array has no idea how large "he" is**
- **In fact, it does not know that "he" is an array**

Strings (or something like it)

Strings are array of characters terminated by '\0'

- the '\0' is added to string constants
  - "Hello"



Strings

```
int main() {
    printf("%s\n","Hello");
    return 0;
}
```

```
FlyingComet /Users/pvh/courses/cs034 [107] -> a.out
Hello
```

Strings

```
void printString(char s[]) {
   int i;
   for(i=0;s[i]!='\0';i++)
      printf("%c",s[i]);
}
int main() {
   printString("Hello");
   printf("\n");
   return 0;
}
```

Swapping

Write a function to swap two entries of an array

```
void swap(int a[],int i,int j) {
    int tmp;
    tmp = a[j];
    a[j] = a[i];
    a[i] = tmp;
}
```

Tell C that this function does not return a value

Selection Sort

```
void selectsort (int array[], int length){
  int i, j, min;
  for (i = 0; i < length; ++i){
    /* find the index of the smallest item from i onward */
    min = i;
    for (j = i; j< length; ++j)
      if (array[j] < array[min])
        min = j;
    /*  put the smallest item in the i-th position, preserve i-th  */
    swap (array, i, min);
  }
  /*  at the end of each iteration, the first i slots have the i smallest items.  */
}
```

Swapping

Write a function to swap two int

```
void swap(int i,int j) {
```

```
}
int main() {
    int a; a = 4;
    int b; b = 8;
    swap(a,b);
    printf("a:%d  b:%d",a,b);
}
```

Parameters are passed by value

Swapping

Write a function to swap two int

```
void swap(int i,int j) {
    int tmp;
    tmp = j; j = i; i = tmp;
}
int main() {
    int a; a = 4;
    int b; b = 8;
    swap(a,b);
    printf("a:%d  b:%d\n",a,b);
}
```

FlyingComet [114] -> a.out
a:4 b:8

Swapping

Write a function to swap two int

```
void swap(int i,int j) {
    int tmp;
    tmp = j; j = i; i = tmp;
}
int main() {
    int a; a = 4;
    int b; b = 8;
    swap(a,b);
    printf("a:%d  b:%d\n",a,b);
}
```

```
FlyingComet  [114] -> a.out
a:4  b:8
```

Memory addresses

In C

- you can take the address of every object
- just use the magical operator &

```
int main() {
    int a; a = 4;
    printf("%u\n",&a);
}
```

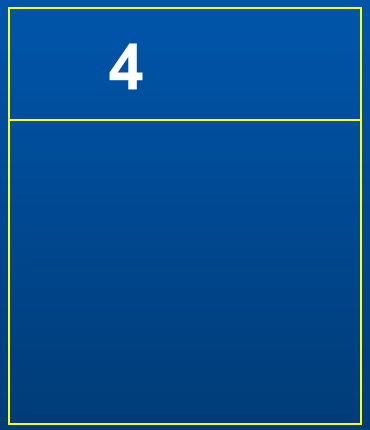
```
/Users/pvh/courses/cs034 [118] -> a.out
3221224352
```

Memory addresses

In C

- you can take the address of every object

```
int main() {
   int a; a = 4;
   printf("%u\n",&a);
}
```




```
/Users/pvh/courses/cs034 [118] -> a.out
3221224352
```

C Pointers

- What is a C pointer?
  - a variable that holds an address
- Pointers in C are “typed” (remember the promises)
  - pointer to an int
  - pointer to a char
  - pointer to a float
  - pointer to <whatever you can define>
- C has a syntax to declare pointer types
  - some people are “religious” about this syntax

C Pointers

p is a pointer to an int

if you follow p, you find an int

```
int main() {
    int *p;
    int a; a = 4;
    p = &a;
    printf("%u\n",p);
}
```

p takes the address of a

```
/Users/pvh/courses/cs034 [125] -> a.out
3221224352
```

C Pointers

p is a pointer to an int

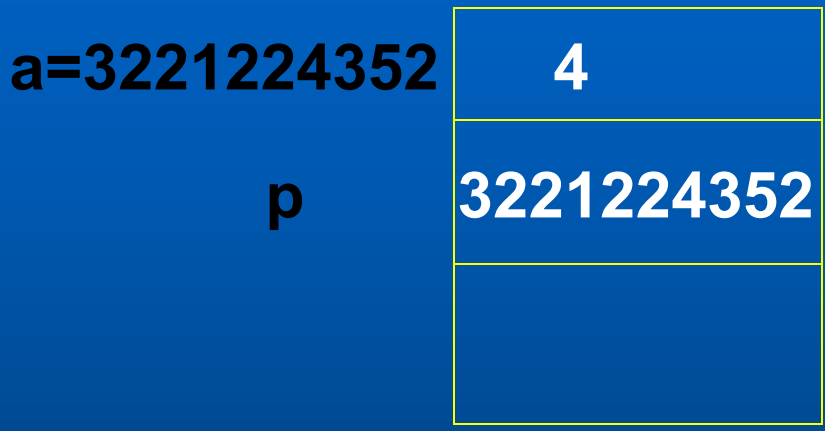
```
int main() {
    int* p;
    int a; a = 4;
    p = &a;
    printf("%u\n",p);
}
```

p takes the address of a

```
/Users/pvh/courses/cs034 [125] -> a.out
3221224352
```

C Pointers

```
int main() {
    int *p;
    int a; a = 4;
    p = &a;
    printf("%u\n",p);
}
```




```
/Users/pvh/courses/cs034 [125] -> a.out
3221224352
```

Can you guess the address of p?

C Pointers

- Pointers are typed
  - the type of the objects they point to is known
  - there is one exception (see later)
- Pointers are first-class citizens
  - they can be passed to functions
  - they can be stored in arrays and other data structures
  - they can be returned by functions

Swapping

What does this do?

```
void swap(int *i,int *j) {
    int *tmp;
    tmp = j; j = i; i = tmp;
}
int main() {
    int a; a = 4;
    int b; b = 8;
    swap(&a,&b);
    printf("a:%d  b:%d\n",a,b);
}
```

FlyingComet [114] -> a.out
a:4 b:8

C Pointers

- Dereferencing pointers
  - accessing/modifying the value pointed to by a pointer

```
int main() {
   int *p;
   int a; a = 4;
   p = &a;
   printf("%d\n",*p);
   *p = *p + 1;
   printf("%d\n",*p);
}
```

```
/Users/pvh/courses/cs034 [125] -> a.out
4
5
```

Dereferencing C Pointers

```
int main() {
    int *p;
    int a; a = 4;
    p = &a;
    printf("%d\n",*p);
    *p = *p + 1;
    *p += 3;
    printf("%d\n",*p);
}
```

Look Ma, I can swap (ints)

```
void swap(int *i,int *j) {
    int tmp;
    tmp = *j; *j = *i; *i = tmp;
}
int main() {
    int a; a = 4;
    int b; b = 8;
    swap(&a,&b);
    printf("a:%d  b:%d\n",a,b);
}
```

```
FlyingComet  [142] -> a.out
a:8  b:4
```

C Pointers

p is a pointer to an int

The expression *p is an int

```
int main() {
    int *p;
    int a; a = 4;
    p = &a;
    printf("%u\n",p);
}
```

p takes the address of a

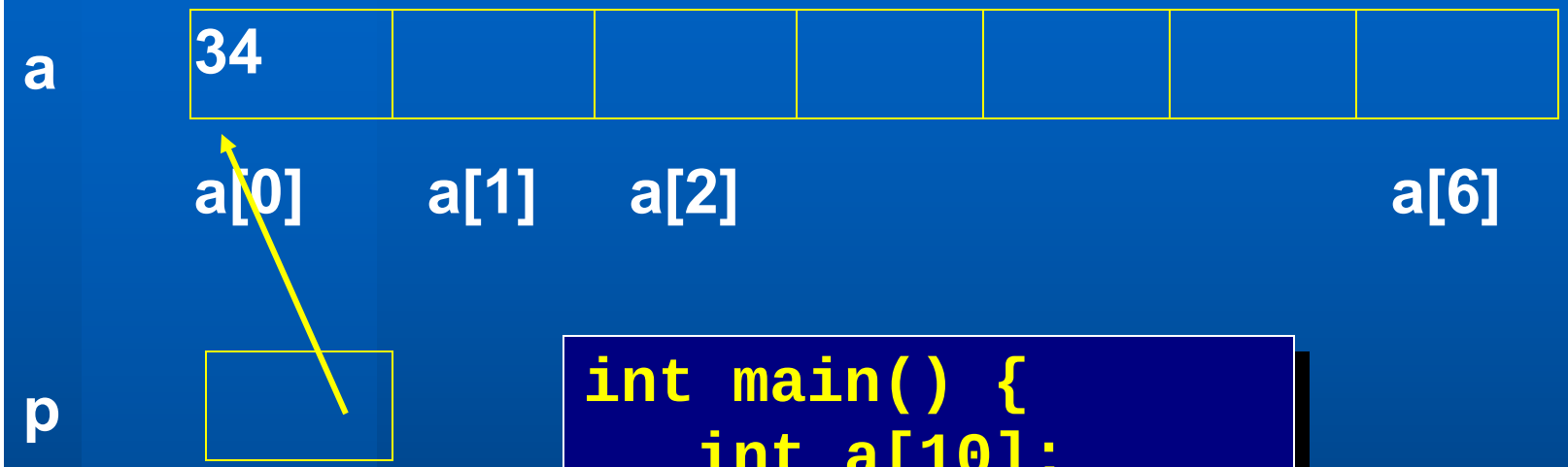
```
/Users/pvh/courses/cs034 [125] -> a.out
3221224352
```

Pointers and Array

- Strong relationships between them

```
int average(int *a,int s) {
   int i;int sum;
   for(i=0,sum=0;i<s;i++)
     sum += a[i];
   return sum/s;
}
int main() {
   int a[100];
   …
   printf("%d\n",average(a,100));
}
```

What is an array?

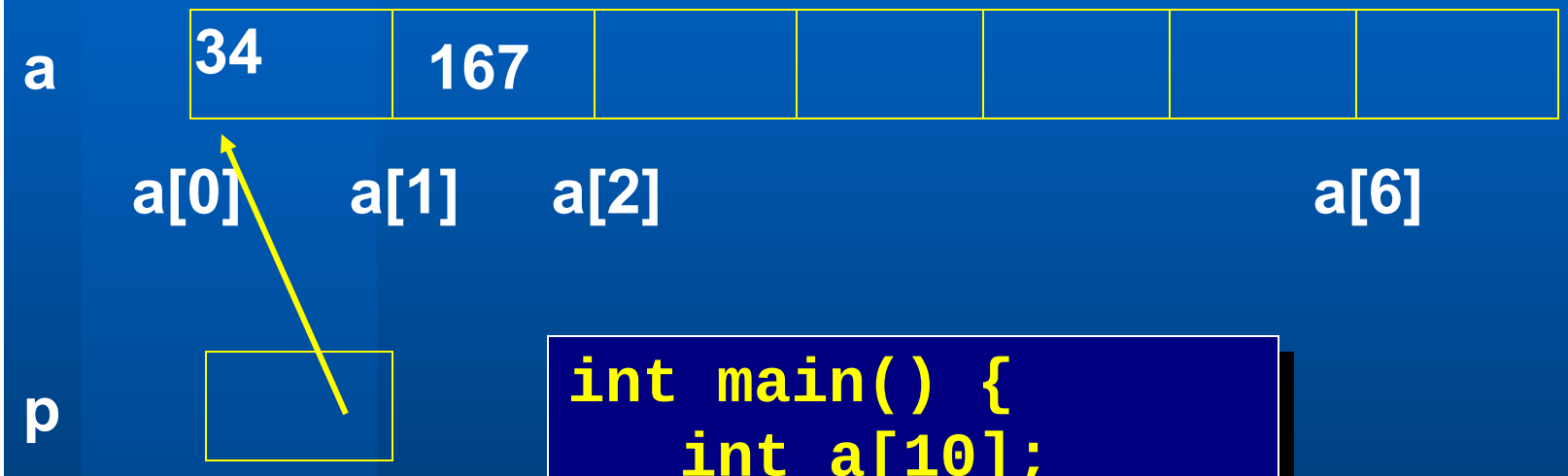



```
int main() {
    int a[10];
    int *p;
    p = &a[0];
    *p = 34;
}
```

Pointer Arithmetic

Pointer can be incremented/decremented

– what this does to the pointer depends on its type

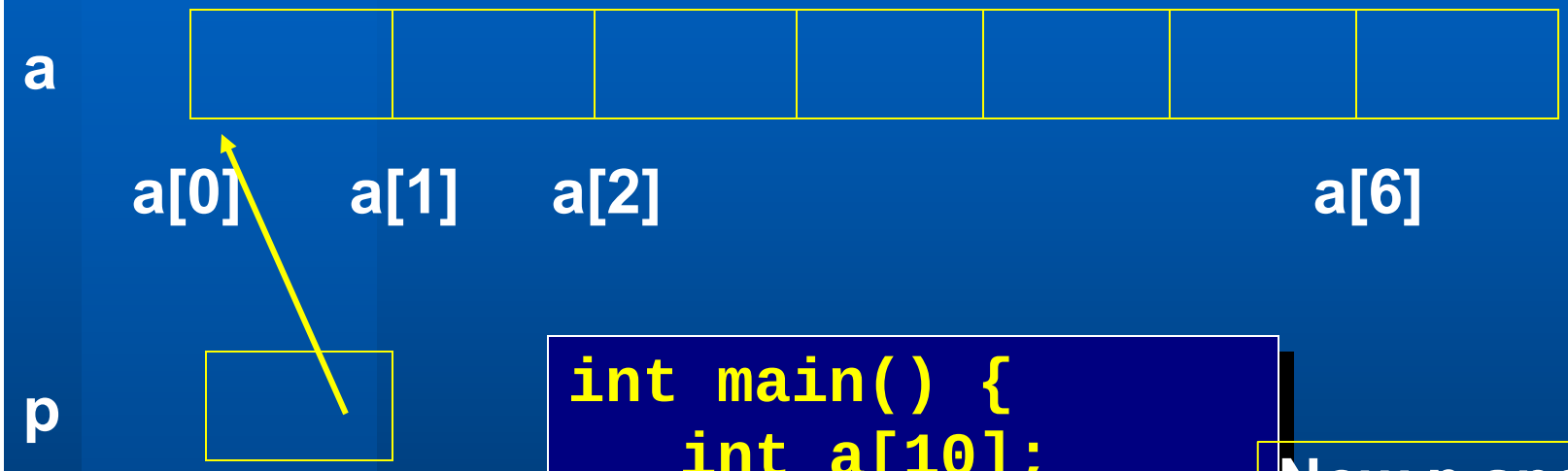



```
int main() {
    int a[10];
    int* p;
    p = &a[0];
    *p = 34;
    *(p+1) = 167;
}
```

Pointer Arithmetic

Pointer can be incremented/decremented

- what this does to the pointer depends on its type



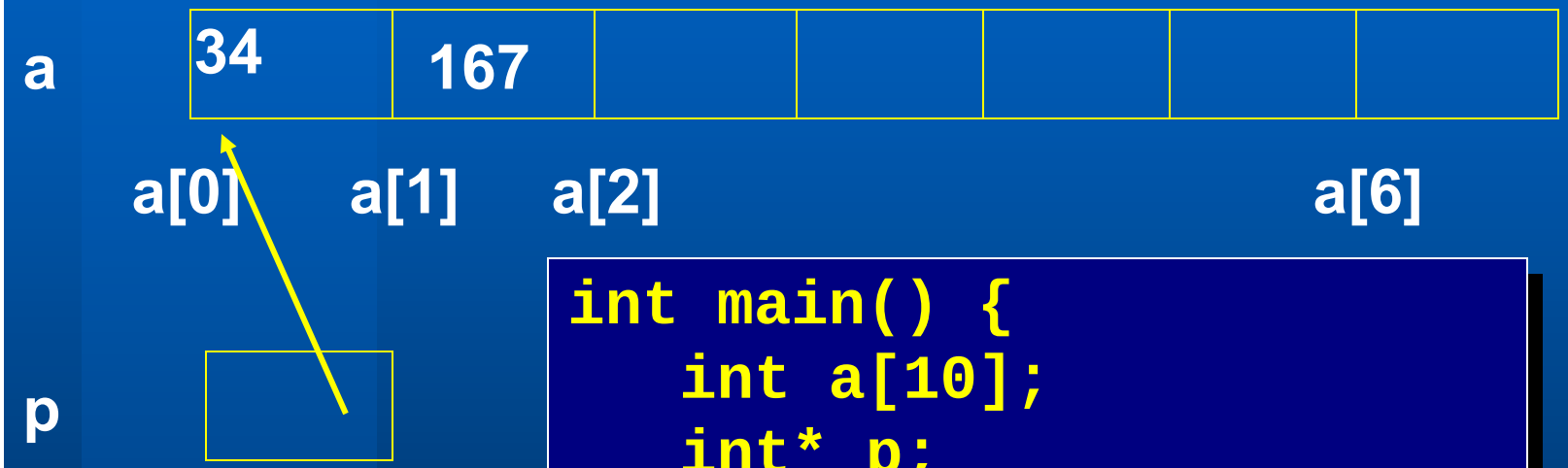

```
int main() {
    int a[10];
    int* p;
    p = &a[0];
}
```

Now p and a have the same value

Pointer Arithmetic

Pointer can be incremented/decremented

- what this does to the pointer depends on its type




```
int main() {
    int a[10];
    int* p;
    p = &a[0];
    *p = 34;
    p[1] = 167;
}
```

Pointer Arithmetic

`p = &a[0];` can also be written as `p = a;`

`a[i];` really is `*(a+i)`

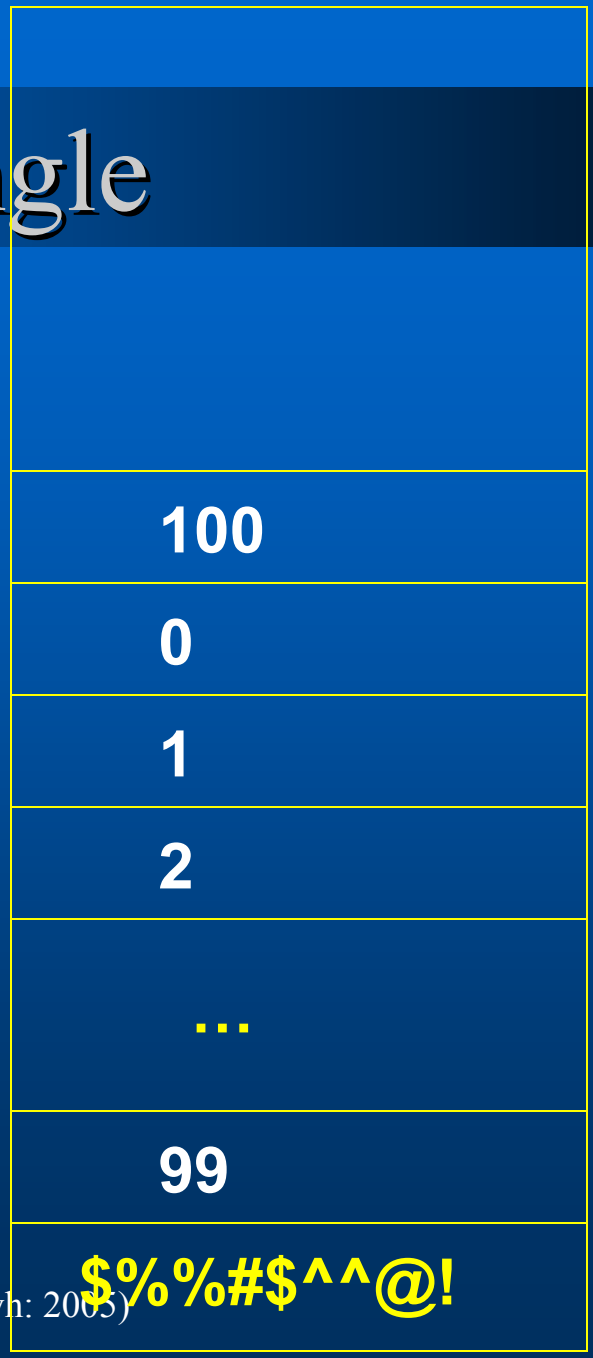
What you cannot do

```
a++;
a = p;
```

Why? Why? Why? Why?

Welcome to the Jungle

```
int main() {
    int i;
    int j;
    int a[100];
    for(i=0;i<100;i++)
      a[i] = i;
    printf("%d\n",j);
}
```




```
[61] -> a.out
-1880816380
```

Pointer Arithmetic

`int a[];` can also be written as `int *a;`

which also explains why

- arrays do not know they are arrays
- they do not know their sizes

C Arrays = Storage + Indexing

Dereferencing C Pointers

```
int main() {
    int *p; int a; a = 4;
    p = &a;
    *p++;
    printf("%d %u\n",*p,p);
}
```

```
/Users/pvh/courses/cs034 [136] -> a.out
-1073742732 3221224360
```

Dereferencing C Pointers

```
int main() {
    int *p; int a; a = 4;
    p = &a;
    (*p)++;
    printf("%d %u\n",*p,p);
}
```

```
/Users/pvh/courses/cs034 [136] -> a.out
5 3221224356
```

Dereferencing C Pointers

```
int main() {
    int *p; int a; a = 4;
    p = &a;
    ++*p;
    printf("%d %u\n",*p,p);
}
```

```
/Users/pvh/courses/cs034 [136] -> a.out
5 3221224356
```

2-D Arrays

```
#define NUM_ROWS 3
#define NUM_COLS 4
…
int main() {
    int row,
    int m[NUM
    for(row=0
      for(col=0;col<NUM_COLS;col++)
          m[row][col] = row*NUM_COLS+col;
    printMatrix(m);
    return 0;
}
```

```
/Users/pvh/courses/cs034 [34] -> a.out
   0   1   2   3
   4   5   6   7
   8   9  10  11
```

2-D Arrays

C must know the number of columns

```
#define NUM_ROWS 3
#define NUM_COLS 4

void printMatrix(int m[][NUM_COLS]) {
   int row, col;
   for(row=0;row<NUM_ROWS;row++) {
     for(col=0;col<NUM_COLS;col++)
        printf("%6d",m[row][col]);
        printf("\n");
   }
}
```

2-D Arrays

```
void printMatrix(int* m) {
   int i;
   for(i=0;i<NUM_ROWS*NUM_COLS;i++) {
      printf("%6d",m[i]);
      if (i%NUM_COLS==NUM_COLS-1)
          printf("\n");
   }
}
```

What does that mean on the memory layout?

Life time of an object

Never never do this

```
int *getArray() {
    int a[10]
    return a;
}
```

The array is allocated on the stack

- **it "disappears" after the function call**

Life time of an object

- **All objects declared in functions**
  - **arguments**
  - **local variables**

  are allocated on the stack
- **They are "deallocated" after the function return;**
- **Never return/use an address to any of them after the function call**
  - **although C lets you do it.**

Global Variables

The scope is global
m can be used
by all functions

```
#define NUM_ROWS 3
#define NUM_COLS 4
int m[NUM_ROWS][NUM_COLS];

int main() {
   int row, col;
   for(row=0;row<NUM_ROWS;row++)
     for(col=0;col<NUM_COLS;col++)
        m[row][col] = row*NUM_COLS+col;
   return 0;
}
```

Global Variables

not allocated on the stack

```
#define NUM_ROWS 3
#define NUM_COLS 4
int m[NUM_ROWS][NUM_COLS];

int main() {
   int row, col;
   printf("%u\n",m);
   printf("%u\n",&row);
   return 0;
}
```

```
/Users/pvh/courses/cs034 [51] -> a.out
8384
3221224352
```

scanf: reading data

```
int main() {
    int i, j;
    scanf("%d%d",&i,&j);
}
```

Two parts

- formating instructions
- arguments: must be addresses

Questions…