C and C++ II

Spring 2014

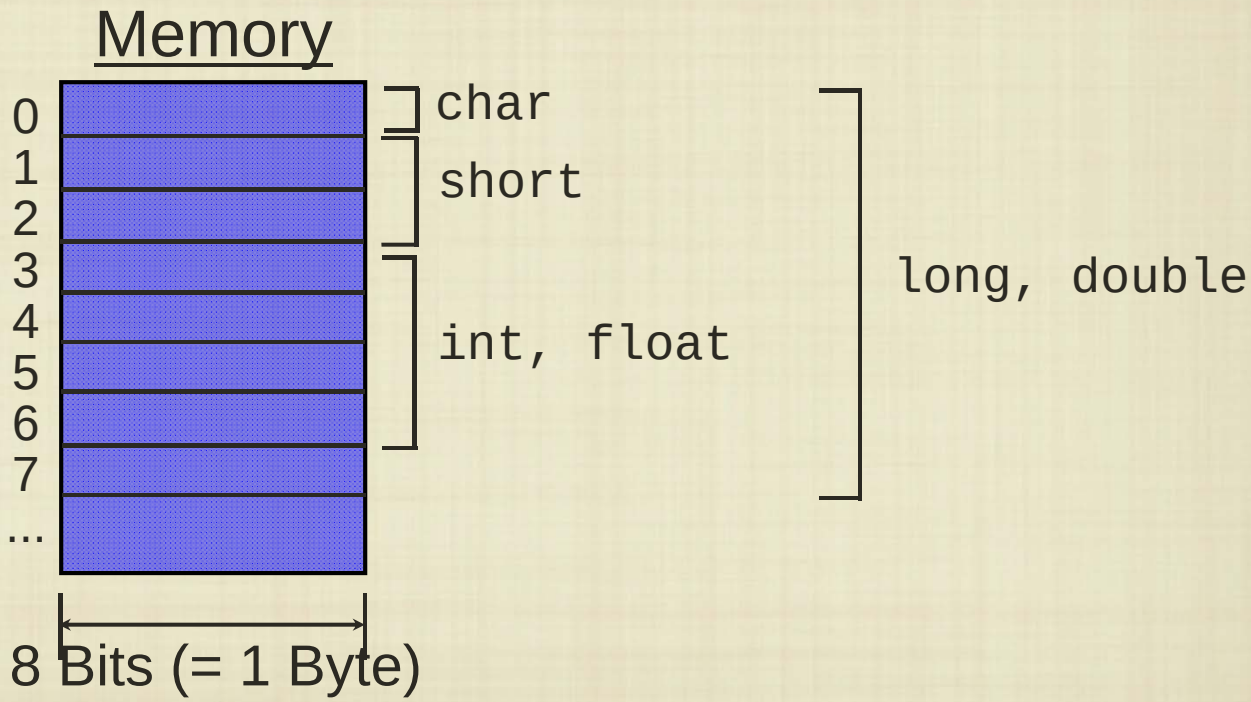
Carola Wenk

C and C++

- The C language was originally developed in the 1970s to assist in the implementation of the UNIX operating system. It was designed to be one step above machine language.
- C++ is a superset of C introduced in the early 1980s to add objected-oriented features to C.
- C is said to be “weakly typed” because there are virtually no compatibility rules between variables of different types.
- What does a “weakly typed” language look like? How do we manage variables and storage?

Types in C

- Primitive types (`int, long, float, double, char`) can be declared in C; the compiler provides some default compatibility.



Types in C

- Variables can be “packaged” using `struct`:

```
#include <stdio.h>

struct my_stuff {
  int a;
  int b;
  int c;
};

int main() {
  struct my_stuff m;
  m.a = 1; m.b = 2; m.c = 3;
  return 0;
}
```

Types in C

- Arrays of a primitive type can be declared, as well as arrays of structs.

```
#include <stdio.h>

struct my_stuff {
  int a;
  int b;
  int c;
};

int main() {
  int x[5] = {1, 5, 10, 11, 12};

  struct my_stuff m[25];
  m[0].a = 1; m[0].b = 2; m[0].c = 3;

  return 0;
}
```

Types in C

- While strings in Java and Python were abstract types, strings in C are implemented as character arrays:

```
#include <stdio.h>


int main() {
  char s[25];
  printf(“Enter a string:”);
  fgets(s, 25, stdin);
  printf(“\nYou entered: %s\n”, s);

  return 0;
}
```

Types in C

- In general, arrays in C are much like Java, except that the `length` field does not exist.

- The syntax for declaring arrays is also slightly different since the brackets are placed after the variable name.

```
#include <stdio.h>

void foo(int A[], int n) {
  ...
}

int main() {
  int X[25];

  foo(X, 25);
  ...
}
```

Pointers and Addresses

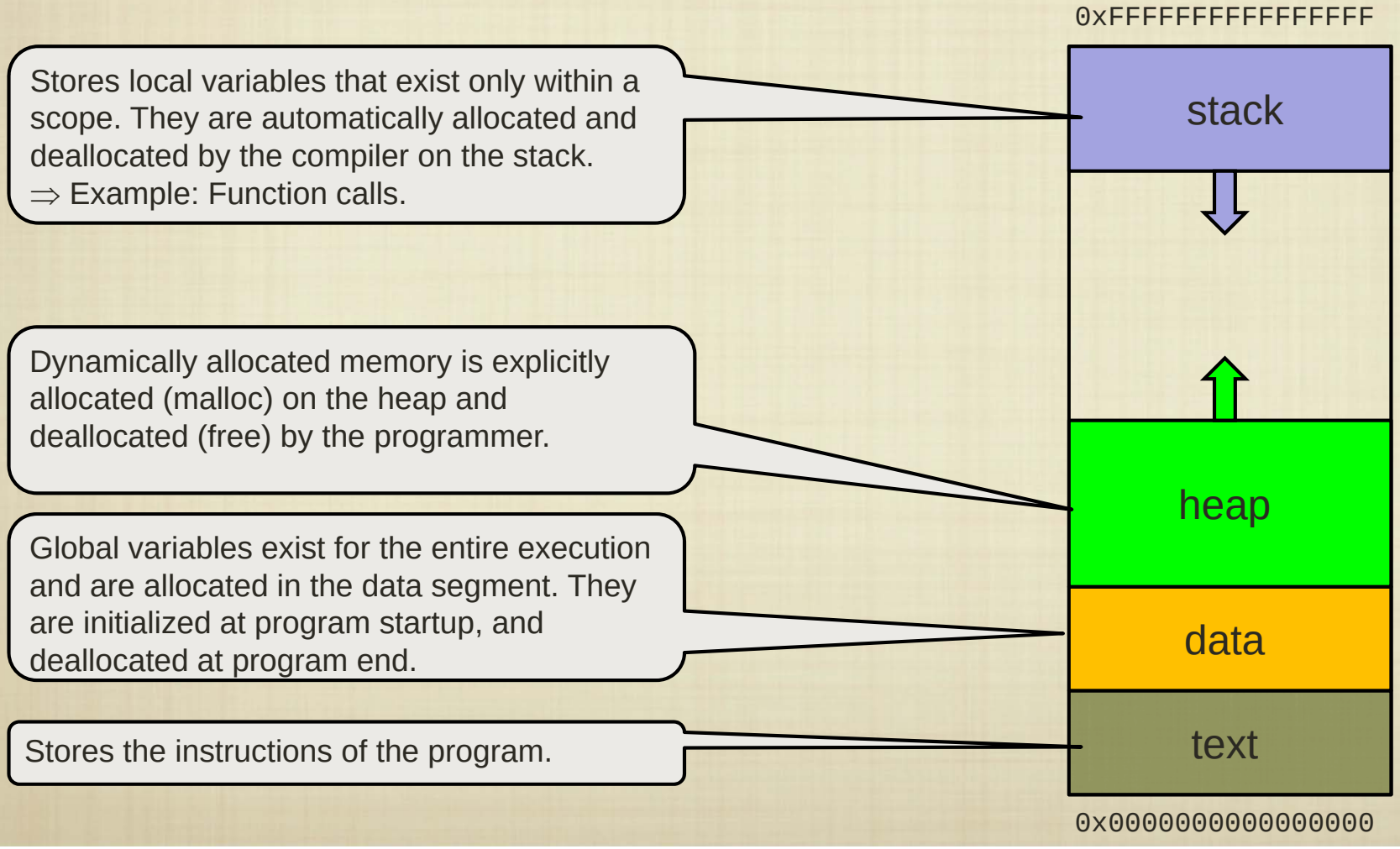
- So far we talked about primitive types (char, int, double) and arrays of them.
- Does C have a concept similar to Java references?
- What would we want to use those for?

$\Rightarrow$ Linked structures; pass changeable parameters into functions

Program Stack

Each program (process) has a full virtual address space.
A typical organization of the stack is as follows:

- The heap begins at low addresses and grows upward
- The stack begins at high addresses and grows downward



Pointers and Addresses

- A pointer is a new kind of variable that, instead of storing a normal value, stores an address in memory.
  - A pointer is often initialized with the address operator (&)
  - A pointer points to (or refers to) the storage location of another variable.
  - A pointer is declared using * .
  - The value a pointer points to can be changed using the dereference operator (*)

```
int a=20;
printf("%d\n",&a);


int* x = &a;
*x = 33;
```

Indexing/Offsetting Arrays and Pointers

- An array is a way to associate multiple items with the same name. It represents a block of variables of the same type.
- Individual variables are accessed using an index (also called offset). E.g., A[2]
- Note that in C, the syntax for declaring an array variable and for indexing an array are decievingly similar.

```
int A[4]={0,1,2,3};
A[2]=57;
/* A is now {0,1,57,3}*/
printf("A[2]=%d\n",A[2]);
```

- Pointers can be offset just like arrays can.

```
int A[4]={0,1,2,3};
int* p = &A[0];
p=p+2;
*p = 57;
/* A is now {0,1,57,3}*/
printf("A[2]=%d\n",A[2]);
printf("*(p+3)=%d\n",*(p+3);
```

Doing Crazy Things with Pointers

- Pointers can point to any place in memory
- One can do a lot of crazy things with offsetting pointers
  - Side-effects that are hard to predict
  - Buffer overflow attacks…

```
int i=0;
int j=1;
int k=2;
int* jp=&j;
jp[1]=57;
*(jp+2)=-32;
```

Dynamic Memory Allocation

- In C we have to declare arrays with a fixed size:

```
int A[4];
```

- But what if we want to have dynamic arrays that might change their size during the course of the program?
  - ➢ In Java we just declared an array without a size:

```
int[] A;
```

- In C we can use pointers instead:
- But then we have to allocate the memory by hand:
  - This allocates dynamic memory on the heap
- In the end, dynamic memory needs to be released back to the system.

```
int *A;

A = malloc(4*sizeof(int));
/* Allocated A[0..3] */
A[2]=57;
*(A+1)=42;
free(A);
```

⇒ Arrays are simply pointers that have space already allocated for us by the compiler, and that can't change what they point to.

Strings

- Strings are character arrays that are null-terminated
  - The last character is a ‘\0’ which really is 0
  - This indicates the end of the string, which is necessary for printing strings…

```
char *word = “hello”;
char anotherWord[8]={‘h’,’e’,’l’,’l’,’o’,’\0’,0,0};
printf(“word=%s\n”,word);
printf(anotherWord=%s\n”,anotherWord);
```