C-STYLE STRUCTS AND ARRAYS

LECTURE 03-1

JIM FIX, REED COLLEGE CS2-F20

HOMEWORK 01 FEEDBACK; HOMEWORK 02 EXTENSION

I'm working to finish grading everyone's **Homework 01**

- Look for a branch named **`feedback`** under your **`hw01-...`** repo.
- I've put comments within your code, and also a **`FEEDBACK.md`** file.
- People's C++ work generally looks good, some "style" issues...

- I will be posting my solutions to their puzzles.
  - *(show solutions; Makefile)*

- **Homework 02** my is due Wednesday at 3pm, instead.
- We'll meet to work on **Lab 02** tomorrow.

A WORD ON STYLE

- Use meaningful, readable names.
  - Avoid single-letter names; only use them judiciously.
    - ➡ Common exceptions are use of `i`, `j`, `k`, `n`, `x`, `y`
  - Use `camelCase` or `snake_case` for names.
- Indent code despite the braces.
  - Structure it as if you were writing Python code.
  - Use curly braces even around single-line conditional and loop blocks.
- Typically I only want you to use constructs I've shown in class!
  - ➡ Mostly I don't want you to avoid the puzzle of the problem.

MORE ON STYLE

- I'd prefer that you don't use **`namespace`**.
- Start using **`std::endl`** instead of **`"\n"`**.
  - ➡ You will still see me use **`\n`** in examples to fit code on slide.
- Work to comment your code!
  - Put them near or within tricky code.
  - When using terse variable names, comment how they are being used.
  - Make a top comment with your name and the assignment being solved.
  - Comment each **`struct`** (today's topic), function, and procedure definition.

TODAY: C++ STRUCTS AND ARRAYS

We look at the primitive data structures that were introduced with C

- C++ ***arrays*** are like primitive Python lists
  - ➡ sequences of data, all of same type
- C++ ***structs*** are like primitive Python objects
  - ➡ conglomeration of data, of mixed type

‣ There will be some important differences; some are subtle.

NOTE: for now, these will be *stack-allocated*

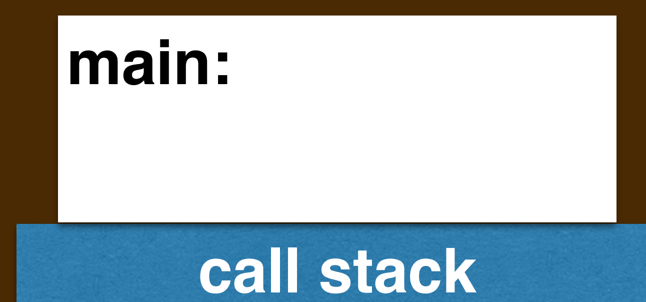
‣ Soon we will look at *pointers*, and also *heap-allocated* arrays and structs.

RECALL CALL STACK STUFF: stack.cc

```
double g(double a, double z0) {
  double z = 0.5 * (z0 + a/z0);
  return z;
}
double f(int i, double x) {
  double y = x/2.0;
  for (int j=0; j<i; j++) {
    y = g(x,y);
  }
  return y*y;
}
void P(int a, int b) {
  std::cout << f(a,2.0) << std::endl;
  std::cout << f(b,2.0) << std::endl;
}
int main() {
  P(2,5);
}
```

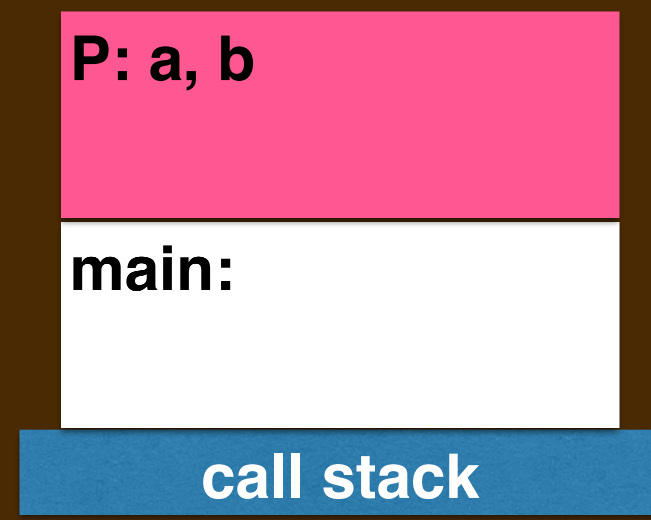
RECALL CALL STACK STUFF: stack.cc

```
double g(double a, double z0) {
  // uses z
  ...
}
double f(int i, double x) {
  // uses x, j, y; calls g
  ...
}
void P(int a, int b) {
  // calls f twice
  ...
}
int main() {
  P(2,5);
}
```



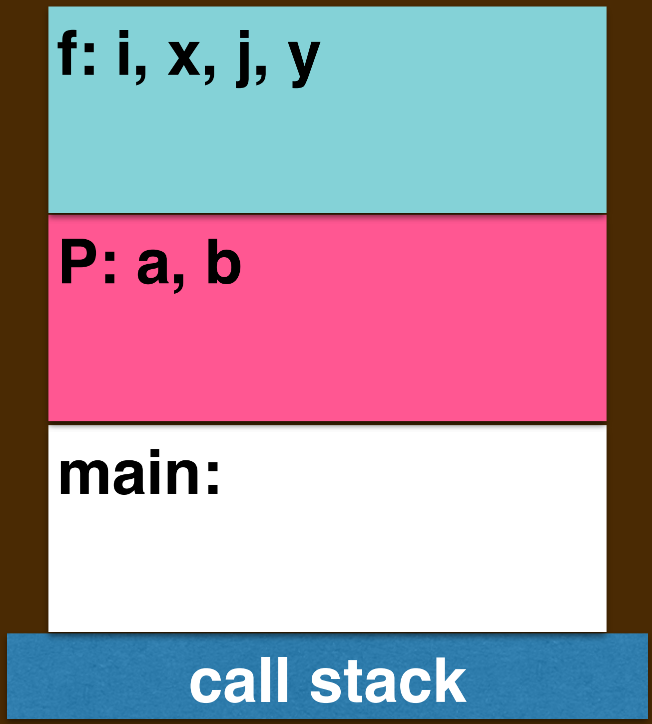
RECALL CALL STACK STUFF: stack.cc

```
double g(double a, double z0) {
   // uses z
   ...
}
double f(int i, double x) {
   // uses x, j, y; calls g
   ...
}
void P(int a, int b) {
   // calls f twice
   ...
}
int main() {
   P(2,5);
}
```



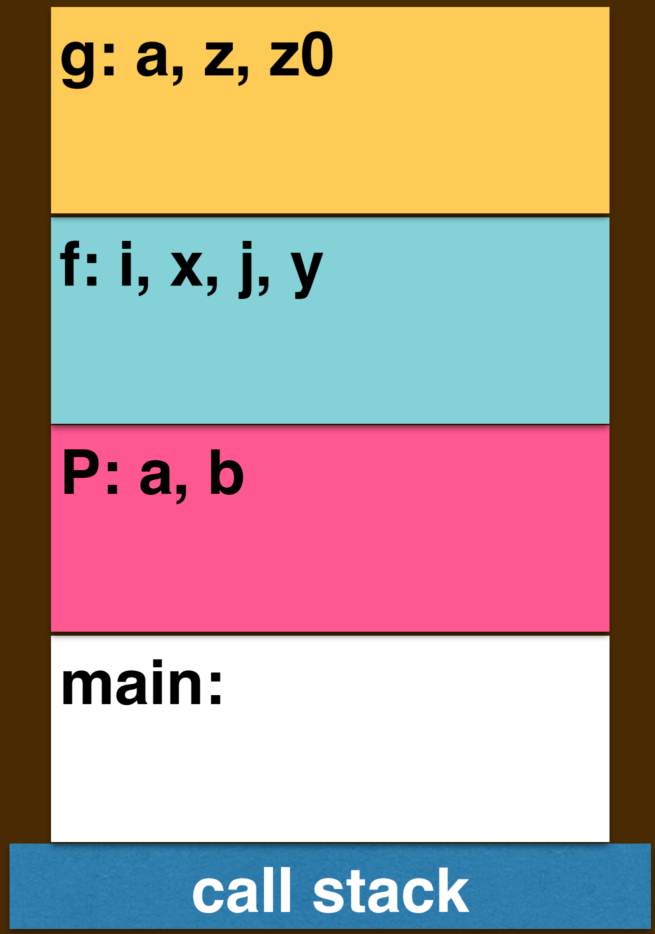
RECALL CALL STACK STUFF: stack.cc

```
double g(double a, double z0) {
   // uses z
   ...
}
double f(int i, double x) {
   // uses x, j, y; calls g
   ...
}
void P(int a, int b) {
   // calls f twice
   ...
}
int main() {
   P(2,5);
}
```



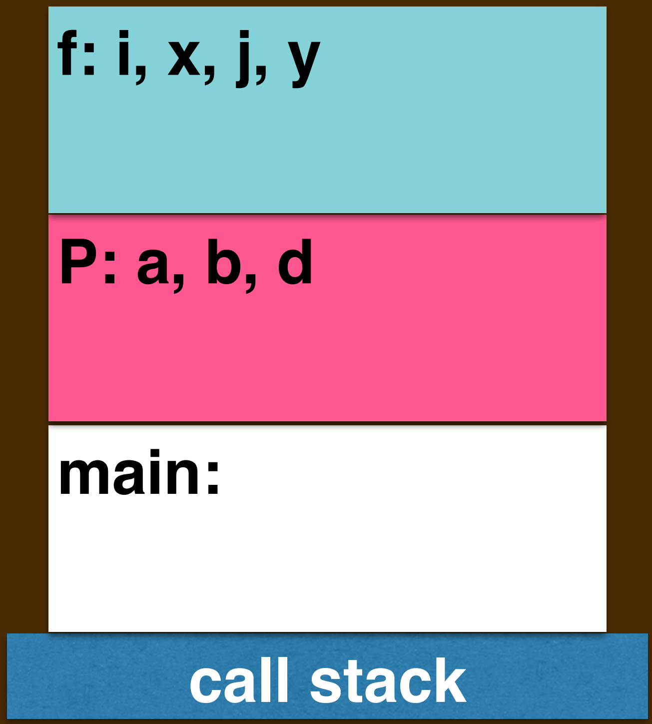
RECALL CALL STACK STUFF: stack.cc

```
double g(double a, double z0) {
   // uses z
   ...
}
double f(int i, double x) {
   // uses x, j, y; calls g
   ...
}
void P(int a, int b) {
   // calls f twice
   ...
}
int main() {
   P(2,5);
}
```



RECALL CALL STACK STUFF: stack.cc

```
double g(double a, double z0) {
   // uses z
   ...
}
double f(int i, double x) {
   // uses x, j, y; calls g
   ...
}
void P(int a, int b) {
   // calls f twice
   ...
}
int main() {
   P(2,5);
}
```



CALL STACK FRAME SUMMARY

- Every function and procedure has a collection of *local variables*.
  - NOTE: These include its *formal parameter* variables.
  - Each variable's bytes are stored in memory on a *stack frame*.
    - ➡ This means they each (temporarily) live at some *address* in memory.
- When a program first runs, a stack frame is built for **`main`**.
  - ➡ This allocates storage for values of each of **`main`**'s local variables.
- When a function is called...
  - ➡ a stack frame is built for its local variables. A new frame is "*pushed on top*."
- When a function returns...
  - ➡ its stack frame is "taken down"; storage is reclaimed. Frame is "*popped off*."

ADDRESS-OF OPERATOR &

- We can put **`&`** in front of an expression that accesses locations in memory.
  - ➡ This tells us the start address of those locations.

- **RECALL:** For **`fib.cc`** we...
  - ➡ output **`&n`**, to inspect the memory address where each frame lived.
- We saw that the stack "grew downward" from higher to lower addresses.

- Let's do a similar thing with **`stack.cc`** from the animation...

(DEMO in TERMINAL)

TODAY: C++ STRUCTS AND ARRAYS

We look at the primitive data structures that were introduced with C

- C++ ***arrays*** are like primitive Python lists
  - ➡ sequences of data, all of same type
- C++ ***structs*** are like primitive Python objects
  - ➡ conglomeration of data, of mixed type
- There will be some important differences; some are subtle.

NOTE: for now, these will be *stack-allocated*

- Soon we will look at *pointers*, and also *heap-allocated* arrays and structs.

STACK-ALLOCATED ARRAYS

- An ***array*** is a sequence of values, all of the same type.
  - It is named with a single variable.
  - Arrays are allocated on the stack with declarations like these:

```
int values[] = {8, 1, 8, 7, 5};
double stuff[3];
```

- Each item of that sequence is accessible by an integer *index*

```
...values[index]...
```

  - The index starts at **0**, runs up to one less than its length

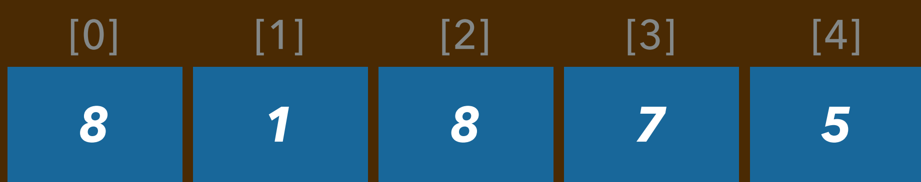
```
for (int i=0; i < 5; i++) {
  std::cout << values[i] << std::endl;
}
```

STACK-ALLOCATED ARRAYS

- An array is a sequence of *memory locations*, storing values of the same type

```
int values[] = {8, 1, 8, 7, 5};
```

- Picture:



- The expression **`values[3]`** refers to the storage of the 4th element, so

```
std::cout << values[3]; // prints 7
```

  - ➡ accesses and prints its integer value, and

```
values[3] = 47;
```

  - ➡ modifies that element within the array.

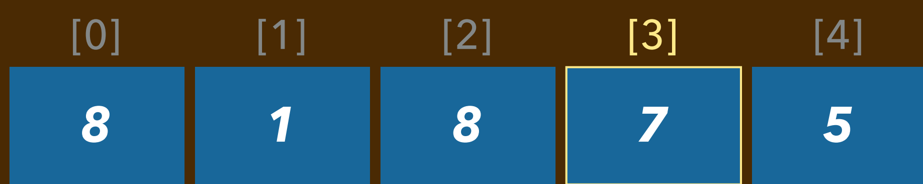
```
std::cout << values[3]; // now prints 47
```

STACK-ALLOCATED ARRAYS

- An array is a sequence of *memory locations*, storing values of the same type

```
int values[] = {8, 1, 8, 7, 5};
```

- Picture:



- The expression **`values[3]`** refers to the storage of the 4th element, so

```
std::cout << values[3]; // prints 7
```

  - ➡ accesses and prints its integer value, and

```
values[3] = 47;
```

  - ➡ modifies that element within the array.

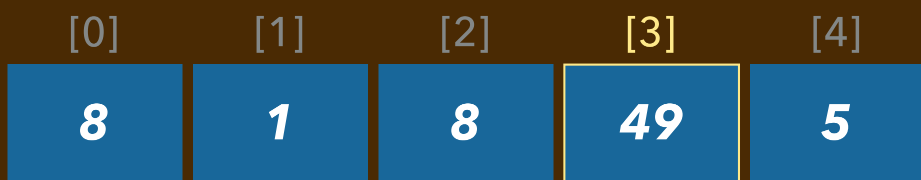
```
std::cout << values[3]; // now prints 47
```

STACK-ALLOCATED ARRAYS

- An array is a sequence of *memory locations*, storing values of the same type

```
int values[] = {8, 1, 8, 7, 5};
```

- Picture:



- The expression **`values[3]`** refers to the storage of the 4th element, so

```
std::cout << values[3]; // prints 7
```

  ➡ accesses and prints its integer value, and

```
values[3] = 49;
```

  ➡ modifies that element within the array.

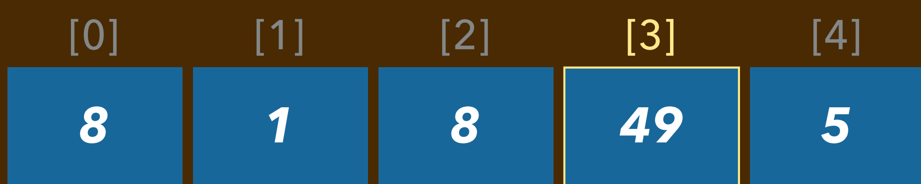
```
std::cout << values[3]; // now prints 49
```

STACK-ALLOCATED ARRAYS

- An array is a sequence of *memory locations*, storing values of the same type

```
int values[] = {8, 1, 8, 7, 5};
```

- Picture:



- The expression `values[3]` refers to the storage of the 4th element, so

```
std::cout << values[3]; // prints 7
```

  - ➡ accesses and prints its integer value, and

```
values[3] = 49;
```

  - ➡ modifies that element within the array.

```
std::cout << values[3]; // now prints 49
```

EXAMPLE: array1.cc

```
#include <iostream>

int main(void) {
  int a[5] = {8,1,8,7,5};
  for (int i=0; i<5; i++) {
    std::cout << a[i] << std::endl;
  }
  return 0;
}
```

```
% ./array1
8
1
8
7
5
```

EXAMPLE: array2.cc

```
#include <iostream>
int main(void) {
  int a[5] = {8,1,8,7,5};
  int sum = 0;
  for (int i=0; i<5; i++) {
    sum += a[i];
  }
  std::cout << sum << std::endl;
  return 0;
}
```

```
% ./array2
29
```

EXAMPLE: array3.cc

```
#include <iostream>
int main(void) {
  int a[5];

  for (int i=0; i<5; i++) {
    a[i] = (6-i)*10 + i;
  }
  for (int i=0; i<5; i++) {
    std::cout << a[i] << std::endl;
  }
  std::cout << std::endl;

  a[2] = a[2] + 100;
  for (int i=0; i<5; i++) {
    std::cout << a[i] << std::endl;
  }
  return 0;
}
```

```
% ./array3
60
51
42
33
24

60
51
142
33
24
```

ARRAY SYNTAX

- To declare and allocate storage for a **stack-allocated array**

  type-name variable-name **[ ] = {** ***initializer-list*** **} ;**

  type-name variable-name **[** ***integer-literal*** **] ;**

- To access the contents of an array item (this is an "*R-value reference*"):

  ... variable-name **[** ***integer-expression*** **] *...***

- To modify the contents of an array item (LHS is an "*L-value reference*"):

  variable-name **[** ***integer-expression*** **] =** ***expression*** **;**

- This means that you can think of each item as a variable in memory.

  &*variable-name* **[** ***integer-expression*** **]** gives the address where the item lives in memory

- NOTE: The array variable's value itself is a *pointer*. More on this Wednesday...

  variable-name on its own (no index/brackets) also gives the address of the 0-th item

NOTES ON (STACK-ALLOCATED) ARRAYS

- Notation is similar to a Python list, ***but:***
  - ➡ allocated on the function's stack frame with a declaration
    - ✦ they are of fixed size set by the declaration
  - ➡ elements all have to be the same type
  - ➡ cannot resize the storage (can't change array to have fewer/more items)
  - ➡ they "don't know" their length; can accidentally access at a bad index
  - ➡ deallocated when the function returns (storage is reclaimed)
    - ✦ shouldn't return a stack-allocated array!
  - ➡ array variable's value is an address or *pointer*
    - ✦ *passing arrays to functions as parameters requires a bit of explanation*

EXAMPLE: array4.cc

```
#include <iostream>

int main(void) {
  int i;
  int a[5] = {8,1,8,7,5};
  int j;

  std::cout << "   &i is " << &i << std::endl;
  std::cout << "    a is " << a  << std::endl;
  for (i=0; i<5; i++) {
    std::cout << "&a[" << i << "] is ";
    std::cout << &a[i] << std::endl;
  }
  std::cout << "   &j is " << &j << std::endl;

  std::cout << "    j is " << j << std::endl;
  a[-3] = 345;
  std::cout << "    j is " << j << std::endl;
}
```

EXAMPLE: array4.cc

```
#include <iostream>

int main(void) {
  int i;
  int a[5] = {8,1,8,7,5};
  int j;

  std::cout << "   &i is " << &i << std::endl;
  std::cout << "    a is " << a  << std::endl;
  for (i=0; i<5; i++) {
    std::cout << "&a[" << i << "] is ";
    std::cout << &a[i] << std::endl;
  }
  std::cout << "   &j is " << &j << std::endl;

  std::cout << "    j is " << j << std::endl;
  a[-3] = 345;
  std::cout << "    j is " << j << std::endl;
}
```

```
% ./array4
   &i is 0x7ffee0ded9f8
    a is 0x7ffee0deda00
&a[0] is 0x7ffee0deda00
&a[1] is 0x7ffee0deda04
&a[2] is 0x7ffee0deda08
&a[3] is 0x7ffee0deda0c
&a[4] is 0x7ffee0deda10
   &j is 0x7ffee0ded9f4
    j is 111
    j is 345
```

STACK-ALLOCATED STRUCTS

- An ***struct*** is a grouping of stored values; a collection of storage *components*
  - Each component has a name. Also called a *field* or "instance variable".
  - Each component can be of a different type.
- You have to declare the struct as a new type before you use it.
  - Arrays are allocated on the stack with declarations like these:

```
struct record {
  int value;
  std::string text;
  double amount;
};
```

- You can use that type in a struct variable declaration, for example:

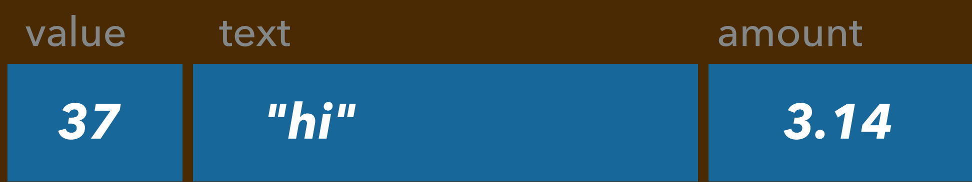
```
record r = {37, "hi", 3.14}; // creates a new record instance
```

STACK-ALLOCATED STRUCTS

- Like arrays, a struct is also a sequence of bytes in memory, chunked as fields

```
record r = {37, "hi", 3.14};
```

- Picture:



- The expression `r.amount` refers to the storage of the 3rd field, so

```
std::cout << r.amount; // prints 3.14
```

  - ➡ accesses and prints its double-precision floating point value, and

```
r.amount = 2.78;
```

  - ➡ modifies that element within the array.

```
std::cout << r.amount; // now prints 2.78
```

STACK-ALLOCATED STRUCTS

- Like arrays, a struct is also a sequence of bytes in memory, chunked as fields

```
record r = {37, "hi", 3.14};
```

- Picture:

- The expression `r.amount` refers to the storage of the 3rd field, so

```
std::cout << r.amount; // prints 3.14
```

  - ➡ accesses and prints its double-precision floating point value, and

```
r.amount = 2.78;
```

  - ➡ modifies that element within the array.

```
std::cout << r.amount; // now prints 2.78
```

STACK-ALLOCATED STRUCTS

- Like arrays, a struct is also a sequence of bytes in memory, chunked as fields

```
record r = {37, "hi", 3.14};
```

- Picture:

- The expression **`r.amount`** refers to the storage of the 3rd field, so

```
std::cout << r.amount; // prints 3.14
```

  ➡ accesses and prints its double-precision floating point value, and

```
r.amount = 2.78;
```

  ➡ modifies that element within the array.

```
std::cout << r.amount; // now prints 2.78
```

STACK-ALLOCATED STRUCTS

- Like arrays, a struct is also a sequence of bytes in memory, chunked as fields

```
record r = {37, "hi", 3.14};
```

- Picture:

- The expression **`r.amount`** refers to the storage of the 3rd field, so

```
std::cout << r.amount; // prints 3.14
```

  - ➡ accesses and prints its double-precision floating point value, and

```
r.amount = 2.78;
```

  - ➡ modifies that element within the array.

```
std::cout << r.amount; // now prints 2.78
```

EXAMPLE: struct1.cc

```
#include <iostream>

struct CS2Student {
    std::string name;
    int year;
    bool isTA;
};

void outputCS2Student(CS2Student student) { ... }

int main(void) {
  CS2Student s = {"Rory Gluthy", 1, false};
  CS2Student t;
  t.name = "Dom Kalemba";
  t.year = 2;
  t.isTA = true;
  outputCS2Student(s);
  outputCS2Student(t);
  ...
}
```

EXAMPLE: struct1.cc

```
#include <iostream>
struct CS2Student {
    std::string name;
    int year;
    bool isTA;
};
void outputCS2Student(CS2Student student) { ... }
int main(void) {
  CS2Student s = {"Rory Gluthy", 1, false};
  CS2Student t;
  t.name = "Dom Kalemba";
  t.year = 2;
  t.isTA = true;
  outputCS2Student(s);
  outputCS2Student(t);
  ...
}
```

```
% ./struct1
A student named Rory Gluthy is a first year student.
A student named Dom Kalemba is a sophomore and is TAing CS2.
```

EXAMPLE: struct1.cc

```
void outputCS2Student(CS2Student student) {
  std::cout << "A student named " << student.name << " is a ";

  if (student.year == 1) {
    std::cout << "first year student";
  } else if (student.year == 2) {
    std::cout << "sophomore";
  } else if (student.year == 3) {
    std::cout << "junior";
  } else if (student.year == 4) {
    std::cout << "senior";
  } else {
    std::cout << "graduate";
  }

  if (student.isTA) {
    std::cout << " and is TAing CS2";
  }
  std::cout << "." << std::endl;
}
```

EXAMPLE: struct1.cc

```
void outputCS2Student(CS2Student student) {
  std::cout << "A student named " << student.name << " is a ";

  if (student.year == 1) {
    std::cout << "first year student";
  } else if (student.year == 2) {
    std::cout << "sophomore";
  } else if (student.year == 3) {
    std::cout << "junior";
  } else if (student.year == 4) {
    std::cout << "senior";
  } else {
    std::cout << "graduate";
  }

  if (student.isTA) {
    std::cout << " and is TAing CS2";
  }
  std::cout << "." << std::endl;
}
```

EXAMPLE: struct1.cc

```
void outputCS2Student(CS2Student student) {
  std::cout << "A student named " << student.name << " is a ";

  if (student.year == 1) {
    std::cout << "first year student";
  } else if (student.year == 2) {
    std::cout << "sophomore";
  } else if (student.year == 3) {
    std::cout << "junior";
  } else if (student.year == 4) {
    std::cout << "senior";
  } else {
    std::cout << "graduate";
  }

  if (student.isTA) {
    std::cout << " and is TAing CS2";
  }
  std::cout << "." << std::endl;
}
```

EXAMPLE: struct1.cc

```
#include <iostream>
struct CS2Student { ... }
void outputCS2Student(CS2Student student) { ... }
int main(void) {
  ... // Declaration and init of s and t.
  outputCS2Student(s);
  outputCS2Student(t);
  std::cout << "A year goes by... " << std::endl;
  s.year++;
  s.isTA = !s.isTA;
  t.year++;
  t.isTA = !s.isTA;
  outputCS2Student(s);
  outputCS2Student(t);
}
```

```
% ./struct1
A student named Rory Gluthy is a first year student.
A student named Dom Kalemba is a sophomore and is TAing CS2.
A year goes by...
A student named Rory Gluthy is a sophomore and is TAing CS2.
A student named Dom Kalemba is a junior.
```

STRUCT SYNTAX

- To declare and allocate storage for a stack-allocated struct

  struct-type-name *variable-name* = { *initializer-list* };

  struct-type-name *variable-name*;

- To access a component of a struct use the "dot notation":

  ... *variable-name*.*component-name* ...

- To modify the contents of an array item (LHS is an "L-value reference"):

  variable-name.*component-name* = *expression*;

- This means that you can think of each component as a variable in memory.

  & *variable-name*.*component-name* gives the address where the field's storage lives in memory

- The struct variable itself is a collection of values that can be passed/returned by value (much like a Python tuple).

NOTES ON (STACK-ALLOCATED) STRUCTS

- Notation is similar to a Python object, ***but:***
  - Allocated on the function's stack frame by a variable declaration statement.
  - Their layout is fixed; based on the struct's *type declaration*
    - Cannot add fields at run-time
  - (For now, they do not have *methods.)*
  - Deallocated when the function returns (storage is reclaimed).
  - Not passed or returned *by reference,* but passed/returned *by value*

NOTES ON (STACK-ALLOCATED) STRUCTS

- Notation is similar to a Python object, ***but:***
  - Allocated on the function's stack frame by a variable declaration statement.
  - Their layout is fixed; based on the struct's *type declaration*
    - Cannot add fields at run-time
  - (For now, they do not have *methods.)*
  - Deallocated when the function returns (storage is reclaimed).
  - Not passed or returned *by reference,* but passed/returned *by value*
    - Component values are copied into the formal parameter's struct.
    - Component values are copied back from the returned struct.

EXAMPLE: struct2.cc

```
#include <iostream>
struct CS2Student { ... }
void outputCS2Student(CS2Student student) { ... }
void yearGoesByWith(CS2Student student) {

  ...

}
int main(void) {
  CS2Student s = {"Rory Gluthy", 1, false};
  outputCS2Student(s);
  yearGoesByWith(s);
  outputCS2Student(s);
}
```

ATTEMPT #1: yearGoesBy

```
#include <iostream>
struct CS2Student { ... }
void outputCS2Student(CS2Student student) { ... }
void yearGoesByWith(CS2Student student) {
  if (student.year <= 4) {
    student.year++;
  }
}
int main(void) {
  CS2Student s = {"Rory Gluthy", 1, false};
  outputCS2Student(s);
  yearGoesByWith(s);
  outputCS2Student(s);
}
```

ATTEMPT #1: yearGoesBy

```
#include <iostream>
struct CS2Student { ... }
void outputCS2Student(CS2Student student) { ... }
void yearGoesByWith(CS2Student student) {
  if (student.year <= 4) {
    student.year++;
  }
}
int main(void) {
  CS2Student s = {"Rory Gluthy", 1, false};
  outputCS2Student(s);
  yearGoesByWith(s);
  outputCS2Student(s);
}
```

```
% ./struct2
A student named Rory Gluthy is a first year student.
A student named Rory Gluthy is a first year student.
```

ATTEMPT #1: yearGoesBy

```
#include <iostream>
struct CS2Student { ... }
void outputCS2Student(CS2Student student) { ... }
void yearGoesByWith(CS2Student student) {
  if (s.year <= 4) {
    s.year++;
  }
}
int main(void) {
  CS2Student s = {"Rory Gluthy", 1, false};
  outputCS2Student(s);
  yearGoesByWith(s);
  outputCS2Student(s);
}
```

ONLY ACCESSES AND CHANGES A COPY OF MAIN'S STRUCT.

```
% ./struct2
A student named Rory Gluthy is a first year student.
A student named Rory Gluthy is a first year student.
```

ATTEMPT #2: yearGoesBy

```
#include <iostream>
struct CS2Student { ... }
void outputCS2Student(CS2Student student) { ... }
CS2Student yearGoesByWith(CS2Student student) {
  if (student.year <= 4) {
    student.year++;
  }
  return student;
}
int main(void) {
  CS2Student s = {"Rory Gluthy", 1, false};
  outputCS2Student(s);
  s = yearGoesByWith(s);
  outputCS2Student(s);
}
```

ATTEMPT #2: yearGoesBy

```
#include <iostream>
struct CS2Student { ... }
void outputCS2Student(CS2Student student) { ... }
CS2Student yearGoesByWith(CS2Student student) {
  if (student.year <= 4) {
    student.year++;
  }
  return student;
}
int main(void) {
  CS2Student s = {"Rory Gluthy", 1, false};
  outputCS2Student(s);
  s = yearGoesByWith(s);
  outputCS2Student(s);
}
```

ACCESSES AND CHANGES A COPY, RETURNS COPY BACK.

ATTEMPT #2: yearGoesBy

```
#include <iostream>
struct CS2Student { ... }
void outputCS2Student(CS2Student student) { ... }
CS2Student yearGoesByWith(CS2Student student) {
  if (student.year <= 4) {
    student.year++;
  }
  return student;
}
int main(void) {
  CS2Student s = {"Rory Gluthy", 1, false};
  outputCS2Student(s);
  s = yearGoesByWith(s);
  outputCS2Student(s);
}
```

ACCESSES AND CHANGES A COPY, RETURNS COPY BACK.

REASSIGNS BASED ON RETURNED COPY.

ATTEMPT #2: yearGoesBy

```
#include <iostream>
struct CS2Student { ... }
void outputCS2Student(CS2Student student) { ... }
CS2Student yearGoesByWith(CS2Student student) {
  if (student.year <= 4) {
    student.year++;
  }
  return student;
}
int main(void) {
  CS2Student s = {"Rory Gluthy", 1, false};
  outputCS2Student(s);
  s = yearGoesByWith(s);
  outputCS2Student(s);
}
```

ACCESSES AND CHANGES A COPY, RETURNS COPY BACK.

REASSIGNS BASED ON RETURNED COPY.

```
% ./struct2
A student named Rory Gluthy is a first year student.
A student named Rory Gluthy is a sophomore student.
```

EXAMPLE: struct4.cc

```
#include <iostream>

struct record {
  int value;
  std::string text;
  double amount;
}

int main(void) {
  int i;
  record r = {37, "hi", 3.14};
  int j;

  std::cout << "&i is " << &i << std::endl;
  std::cout << "r is " << r  << std::endl;
  std::cout << "&r.value is " << &r.value << std::endl;
  std::cout << "&r.text is " << &r.text << std::endl;
  std::cout << "&r.amount is " << &r.amount << std::endl;
  std::cout << "&j is " << &j << std::endl;
}
```

EXAMPLE: struct4.cc

```
#include <iostream>

struct record {
  int value;
  std::string text;
  double amount;
}

int main(void) {
  int i;
  record r = {37, "hi", 3.14};
  int j;

  std::cout << "&i is " << &i << std::endl;
  // std::cout << "r is " << r  << std::endl; // error!
  std::cout << "&r.value is " << &r.value << std::endl;
  std::cout << "&r.text is " << &r.text << std::endl;
  std::cout << "&r.amount is " << &r.amount << std::endl;
  std::cout << "&j is " << &j << std::endl;
}
```

EXAMPLE: struct4.cc

```
#include <iostream>

struct record {
  int value;
  std::string text;
  double amount;
}

int main(void) {
  int i;
  record r = {37, "hi", 3.14};
  int j;

  std::cout << "&i is " << &i << std::endl;
  // std::cout << "r is " << r  << std::endl; // error!
  std::cout << "&r.value is " << &r.value << std::endl;
  std::cout << "&r.text is " << &r.text << std::endl;
  std::cout << "&r.amount is " << &r.amount << std::endl;
  std::cout << "&j is " << &j << std::endl;
}
```

EXAMPLE: struct4.cc

```
#include <iostream>

struct record {
  int value;
  std::string text;
  double amount;
}

int main(void) {
  int i;
  record r = {37, "hi", 3.14};
  int j;

  std::cout << "&i is " << &i << std::endl;
  // std::cout << "r is " << r  << std::endl; // error!
  std::cout << "&r.value is " << &r.value << std::endl;
  std::cout << "&r.text is " << &r.text << std::endl;
  std::cout << "&r.amount is " << &r.amount << std::endl;
  std::cout << "&j is " << &j << std::endl;
}
```

```
% ./struct4
&i is 0x7ffee9498a1c
&r.value is 0x7ffee94989f0
&r.text is 0x7ffee94989f8
&r.amount is 0x7ffee9498a10
&j is 0x7ffee94989ec
```

EXAMPLE: struct4.cc

```
#include <iostream>
struct record { ... };
int main(void) {
  int i;
  record r = {37, "hi", 3.14};
  int j;

  std::cout << "&i is " << &i << std::endl;
  std::cout << "&r.value is " << &r.value << std::endl;
  std::cout << "&r.text is " << &r.text << std::endl;
  std::cout << "&r.amount is " << &r.amount << std::endl;
  std::cout << "&j is " << &j << std::endl;

  std::cout << "j is " << j << std::endl;
  (&r.value)[-1] = 345;
  std::cout << "j is " << j << std::endl;
}
```

```
% ./struct4
&i is 0x7ffee9498a1c
&r.value is 0x7ffee94989f0
&r.text is 0x7ffee94989f8
&r.amount is 0x7ffee9498a10
&j is 0x7ffee94989ec
j is 111
j is 345
```

TOMORROW IN TUESDAY LAB

We'll write simple code to get acquainted with struct/array syntax.

- ➡ Needn't worry too much about call stack, addresses, etc.

WEDNESDAY IN LECTURE

We'll look more at memory address stuff, and also:

- We'll define *pointer types*.
- We'll look at passing arrays as parameters.
- We'll look at *allocating* arrays and structs "*dynamically*" *on the heap*.