Intro to C Programming

Subil Abraham

Oak Ridge Leadership Computing Facility (OLCF)
Oak Ridge National Laboratory (ORNL)

July 6, 2023

ORNL is managed by UT-Battelle LLC for the US Department of Energy

Associated Materials for this Talk

The slides, code examples, and exercises can be found in the `intro_to_c` directory of the repository for this course:

https://github.com/olcf/foundational_hpc_skills

If you have not yet cloned the repository, you can do so as follows:

```
$ pwd
/ccsopen/home/<username>

$ git clone https://github.com/olcf/foundational_hpc_skills.git

$ cd foundational_hpc_skills
```

C Programming Language

- General-purpose programming language initially developed by Dennis Ritchie at Bell Laboratories in 1972-1973
- Compiled Language
  - A compiler is a program used to convert high-level code (like C) into machine code
- Many operating systems, as well as Perl, PHP, Python, and Ruby, are written in C.
- One of the common programming languages used in HPC.

A simple example

- Show the very basics

A Simple C Program (01_simple_c_program/simple.c)

```
#include <stdio.h>

int main(){

    int a = 3;
    printf("The value of this integer is %d\n", a);

    return 0;

}
```

A Simple C Program

C preprocessor directive telling the compiler to include contents of the header file in angle brackets.

```
#include <stdio.h>

int main(){

    int a = 3;

    printf("The value of this integer is %d\n", a);

    return 0;

}
```

A Simple C Program

```c
#include <stdio.h>

int main(){

    int a = 3;
    printf("The value of this integer is %d\n", a);

    return 0;

}
```

Declaration of a function called main, which is where execution of the program begins. The "int" indicates that the function will return an integer value.

More on functions later…

A Simple C Program

```
#include <stdio.h>

int main(){

    int a = 3;

    printf("The value of this integer is %d\n", a);

    return 0;

}
```

These curly braces indicate the beginning and end of the main function.

A Simple C Program

```
#include <stdio.h>

int main(){

    int a = 3;
    printf("The value of this integer is %d\n", a);

    return 0;

}
```

Defines an integer called "a" and assigns it a value of 3.

More on data types soon…

A Simple C Program

A semicolon is used to indicate the end of each statement.

```
#include <stdio.h>

int main(){

    int a = 3;
    printf("The value of this integer is %d\n", a);

    return 0;

}
```

A Simple C Program

```c
#include <stdio.h>

int main(){

    int a = 3;
    printf("The value of this integer is %d\n", a);

    return 0;

}
```

A function, called printf, that sends formatted output to stdout (typically the terminal from which the program was run).

This is one of the functions defined in the stdio.h header file.

More on printf soon…

A Simple C Program

And, of course, a semicolon to indicate the end of the statement.

```
#include <stdio.h>

int main(){

    int a = 3;
    printf("The value of this integer is %d\n", a);

    return 0;

}
```

A Simple C Program

```
#include <stdio.h>

int main(){
    int a = 3;
    printf("The value of this integer is %d\n", a);

    return 0;

}
```

Return value "returned" to the run-time environment.

Typically, a value of 0 indicates a normal/successful exit.

A Simple C Program – Ok, let's compile and run

```
$ gcc simple.c
```

Compile C code into executable

- Using gcc compiler

```
$ ls
a.out simple.c
```

Executable is named a.out by default

```
$ ./a.out
The value of this integer is 3
```

Run program

A Simple C Program – Ok, let's compile and run

```
$ gcc -o simple simple.c
```

Compile C code into executable

- Using gcc compiler

```
$ ls
simple simple.c
```

-o is a compiler flag that allows you to name the executable

```
$ ./simple
The value of this integer is 3
```

Run program

Data Types & Formatted Output

- Basic data types
- `printf` function
- Arrays

Variables and Basic C Data Types

Variables are named storage areas

- For example, `int a = 5` creates a variable (storage area in memory) named “a” and saves the value of 5 in that memory location.
  - Variables of different data types occupy different amounts of memory and can store different ranges of values
- Must be declared before use.

Basic C Data Types

Name	Type	Range of Values	Size (B)
char	Character	ASCII characters	1
int	Integer	-2,147,483,648 to 2,147,483,647	4
float	Decimal (precision to 6 places)	1.2e-38 to 3.4e38	4
double	Decimal (precision to 15 places)	2.3e-308 to 1.7e308	8

Formatted Output with `printf` Function

Example 1:

```
printf("Hello World");
```

```
$ ./a.out
Hello World$
```

The Result of Example 1 would be: `Hello World`

Example 2:

```
printf("Hello World\n");
```

```
$ ./a.out
Hello World
$
```

The Result of Example 2 would be: `Hello World` (with a new line)

Formatted Output with `printf` Function

Example 3:

```
int i = 2;
printf("The value of the integer is %d\n", i);
```

format tag

String to print, with format tags

Variable whose value is used in format tag

The Result of Example 3 would be: `The value of the integer is 2`

Example 4:

```
float x = 3.14159;
printf("The value of the float is %.2f\n", x);
```

format tag

String to print, with format tags

Variable whose value is used in format tag

The result of Example 4 would be: `The value of the float is 3.14`

Formatted Output with `printf` Function

Name	Type	Range of Values	Format Specifier
char	Character	ASCII characters	%c
int	Integer	-32,768 to 32,767 <or> -2,147,483,648 to 2,147,483,647	%d
float	Decimal (precision to 6 places)	1.2e-38 to 3.4e38	%f
double	Decimal (precision to 15 places)	2.3e-308 to 1.7e308	%f

There are many options to format output using the `printf` function. Feel free to Google :)

E.g., `02_data_types/data_types_table/data_types_table.c`

- This example shows the use of variables, arrays, and the `printf()` function, as well as how to find the size of data types using the `sizeof()` function.

Operators

Although we've been using them already, let's take a closer look at operators…

Arithmetic Operators

```
int A = 10;
int B = 2;
```

A op B

`+` Add	`A + B; // would give 12`	
`-` Subtract	`A - B; // would give 8`	
`*` Multiply	`A * B; // would give 20`	
`/` Divide	`A / B; // would give 5`	
`%` Modulus	`A % B; // would give 0`	Remainder after division of B into A

`A++` Increment (same as `A = A + 1`) `// would give 11`

`B--` Decrement (same as `B = B - 1`) `// would give 1`

Relational Operators

Tests relationship between two operands

- If true, returns 1
- If false, returns 0

```
int A = 10;
int B = 2;
```

A op B

==	Equal to	`A == B; // would give 0 (false)`
!=	Not equal to	`A != B; // would give 1 (true)`
>	Greater than	`A > B;  // would give 1 (true)`
<	Less than	`A < B;  // would give 0 (false)`
>=	Greater than or equal to	`A >= B; // would give 1 (true)`
<=	Less than or equal to	`A <= B; // would give 0 (false)`

Assignment Operators

```
int A = 10;
int B = 2;
```

=	`A = B; // would assign a value of 2 to A`	
+=	`A += B; // would assign a value of 12 to A`	(Same as `A = A + B`)
-=	`A -= B; // would assign a value of 8 to A`	(Same as `A = A - B`)
*=	`A *= B; // would assign a value of 20 to A`	(Same as `A = A * B`)
/=	`A /= B; // would assign a value of 5 to A`	(Same as `A = A / B`)
%=	`A %= B; // would assign a value of 0 to A`	(Same as `A = A % B`)

Logical Operators

Used in conjunction with relational operations for decision making

```
int A = 10;
int B = 2;
int C = 5;
```

`&&`	And (true if both true)	`((A > B) && (B == C)); // would give 0 (false)`
`||`	Or (true if at least 1 is true)	`((A > B) || (B == C)); // would give 1 (true)`
`!`	Not (returns the opposite)	`!(B == C);              // would give 1 (true)`

If statements

Let’s take a look at if statements …

If Statements

```
if(condition_1){
   // Execute these statements if condition_1 is met
}
else if(condition_2){
   // Execute these statements if condition_2 is met
}
else{
   // Execute these statements if other conditions are not met
}
```

Once a condition is met, the statements associated with that section are executed and all other sections are ignored.

04_if_statements/if_statement/if_statements.c

```c
#include <stdio.h>

int main(){

  int i = 1;


  if(i < 1){
    printf("i = %d (i < 1)\n", i);
  }
  else if(i == 1){
    printf("i is equal to 1\n");
  }
  else{
    printf("i = %d (i > 1)\n", i);
  }


  return 0;
}
```

```
$ gcc -o if_statement if_statement.c

$ ./if_statement
i is equal to 1
```

Loops

- `while` Loop
- `do-while` Loop
- `for` Loop

`while` Loops

```
while(expression){
   // Execute loop statements until expression evaluates to 0
}
```

`expression:` **Evaluated before each iteration**

03_loops/while_loop/while_loop.c

```c
#include <stdio.h>

int main(){

  float x   = 1000.0;

  while(x > 1.0){
    printf("x = %f\n", x);
    x = x / 2.0;
  }

  return 0;
}
```

```
$ gcc -o while_loop while_loop.c

$ ./while_loop
x = 1000.000000
x = 500.000000
x = 250.000000
x = 125.000000
x = 62.500000
x = 31.250000
x = 15.625000
x = 7.812500
x = 3.906250
x = 1.953125
```

`do-while` Loops

```
do{
   // Execute loop statements until expression evaluates to 0
}while(expression)
```

`expression`: Evaluated after each iteration

The implications of using a do-while loop instead of a while loop can be explored in `03_loops/do_while_loop/do_while_loop.c`

`for` Loops

```
for(initialization; conditional_expression; iteration){
  // loop statements
}
```

`conditional_expression:`	**Evaluated before body of loop**
`iteration:`	**Evaluated after body of loop**

03_loops/for_loop/for_loop.c

```c
#include <stdio.h>

int main(){

  int N   = 10;
  int sum = 0;

  for(int i=0; i<N; i++){

    sum = sum + i;
    printf("Iteration: %d, sum = %d\n", i, sum);

  }

  return 0;

}
```

i++ is same as i = i + 1

```
$ gcc -o for_loop for_loop.c

$ ./for_loop
Iteration: 0, sum = 0
Iteration: 1, sum = 1
Iteration: 2, sum = 3
Iteration: 3, sum = 6
Iteration: 4, sum = 10
Iteration: 5, sum = 15
Iteration: 6, sum = 21
Iteration: 7, sum = 28
Iteration: 8, sum = 36
Iteration: 9, sum = 45
```

Continue Statement (03_loops/continue/continue.c)

When a `continue` statement is encountered within a loop, the remaining statements in the loop body (after the continue) are skipped and the next iteration of the loop begins.

```c
#include <stdio.h>

int main(){

  for(int i=0; i<10; i++){

    if(i == 7){
      continue;
    }

    printf("Loop iteration: %d\n", i);
  }

  return 0;
}
```

```
$ gcc -o continue continue.c

$ ./continue
Loop iteration: 0
Loop iteration: 1
Loop iteration: 2
Loop iteration: 3
Loop iteration: 4
Loop iteration: 5
Loop iteration: 6
Loop iteration: 8
Loop iteration: 9
```

Break Statement (03_loops/break/break.c)

When a `break` statement is encountered within a loop, the loop is terminated.

```
#include <stdio.h>

int main(){

  for(int i=0; i<10; i++){

    if(i == 7){
      break;
    }

    printf("Loop iteration: %d\n", i);
  }

  return 0;
}
```

```
$ gcc -o break break.c

$ ./break
Loop iteration: 0
Loop iteration: 1
Loop iteration: 2
Loop iteration: 3
Loop iteration: 4
Loop iteration: 5
Loop iteration: 6
```

Pause for Exercise

- Compile and run the code in 01_simple_c_program, 02_data_types, 03_loops, 04_if_statements
- In 08_exercises/1_datatypes_loops_if, complete the fibonacci exercise

Functions

A reusable block of code that performs a specific task

- Standard Library Functions
- User-Defined Functions

Standard Library Functions

C built-in functions that can be accessed with appropriate `#include` statements

We have already encountered the `printf` function, which can be used by including the `stdio.h` header file

There are many other C standard library functions defined in other header files

- `math.h, stdlib.h, string.h,` etc.

These functions should be used whenever possible in order to save time (why re-invent the wheel) and because they are well-tested and portable.

User Defined Functions

```
return_type function_name(type1 arg1, type2 arg2, ...){
  // Function Body
}
```

Let's see some examples …

05_functions/add_two_numbers/add_two_numbers.c

```c
#include <stdio.h>

// Function Definition
int add_numbers(int i, int j){

  int result;
  result = i + j;

  return result;
}

// Main Function
int main(){

  int num1 = 3;
  int num2 = 7;

  int sum = add_numbers(num1, num2);
  printf("The sum of num1 and num2 is %d\n", sum);

  return 0;
}
```

```
$ gcc -o add_two_numbers add_two_numbers.c

$ ./add_two_numbers
The sum of num1 and num2 is 10
```

05_functions/add_two_numbers/add_two_numbers.c

```c
#include <stdio.h>

// Function Definition
int add_numbers(int i, int j){

  int result;
  result = i + j;

  return result;
}

// Main Function
int main(){

  int num1 = 3;
  int num2 = 7;

  int sum = add_numbers(num1, num2);
  printf("The sum of num1 and num2 is %d\n", sum);

  return 0;
}
```

```
$ gcc -o add_two_numbers add_two_numbers.c

$ ./add_two_numbers
The sum of num1 and num2 is 10
```

Formal parameters/arguments

Actual parameters/arguments

05_functions/change_value/change_value.c

```
#include <stdio.h>

// Function Definition
void change_number(int i){
  i = 2;
  printf("Inside the function, the number's value is %d\n", i);
}

// Main Function
int main(){

  int number = 1;
  printf("\nBefore calling the function, number = %d\n", number);


  change_number(number);


  printf("After calling the function, number = %d\n\n", number);


  return 0;
}
```

```
$ gcc -o change_value change_value.c

$ ./change_value
Before calling the function, number = 1
Inside the function, the number's value is 2
After calling the function, number = 1
```

Wait.
What’s going on here?

The values of the actual arguments are copied to the formal arguments.

- So changes to the formal arguments do not affect the actual arguments.
- This is called “call by value”

ASIDE: Variable Addresses and Pointers

Variable Addresses

The memory address of a variable can be referenced using the reference operator, &

```
#include <stdio.h>

int main(){

  int i = 1;

  printf("The value of i:   %d\n", i);
  printf("The address of i: %p\n", &i);

  return 0;
}
```

%p – format tag to print address

& (reference operator) – gives the address of the variable

```
$ gcc -o variable_addresses variable_addresses.c

$ ./variable_addresses
The value of i:   1
The address of i: 0x7fff3e720c2c
```

(this address will vary)

Pointer Variables

06_addresses_and_pointers/pointers_1/pointers_1.c

```
#include <stdio.h>

int main(){

  float x = 2.713;
  float *p_x;

  p_x = &x;

  printf("The value of x:   %f\n", x);
  printf("The address of x: %p\n", &x);
  printf("The value of p_x: %p\n", p_x);
  printf("The value stored at the memory address held in p_x: %f\n", *p_x);

  return 0;
}
```

There are special variables in C to store memory addresses: pointers

* used to declare pointer

The pointer is assigned the value of the memory address of x

* (dereference operator) – gives the value stored at a memory address

This is different use of * than above!

```
$ gcc -o pointers_1 pointers_1.c

$ ./pointers_1
The value of x:   2.713000
The address of x: 0x7fff5ce8aa68
The value of p_x: 0x7fff5ce8aa68
The value stored at the memory address held in p_x: 2.713000
```

Pointer Variables

06_addresses_and_pointers/pointers_2/pointers_2.c

```
$ gcc -o pointers_2 pointers_2.c

$ ./pointers_2
The value of x:   2.713000
The address of x: 0x7fff5ce8aa68
The value of p_x: 0x7fff5ce8aa68
The value stored at the memory address held in p_x: 2.713000

The value of x:   3.141000
```

```
#include <stdio.h>

int main(){

  float x = 2.713;
  float *p_x;

  p_x = &x;

  printf("The value of x:   %f\n", x);
  printf("The address of x: %p\n", &x);
  printf("The value of p_x: %p\n", p_x);
  printf("The value stored at the memory address held in p_x: %f\n", *p_x);

  *p_x = 3.141;

  printf("\nThe value of x:   %f\n", x);

  return 0;
}
```

* (dereference operator) – gives the value stored at a memory address

* (dereference operator) – also allows you to change the value stored at that memory address

Ok, back to functions …

05_functions/change_value/change_value.c

```
$ gcc -o change_value change_value.c

$ ./change_value
Before calling the function, number = 1
Inside the function, the number's value is 2
After calling the function, number = 1
```

```c
#include <stdio.h>

// Function Definition
void change_number(int i){
  i = 2;
  printf("Inside the function, the number's value is %d\n", i);
}

// Main Function
int main(){

  int number = 1;
  printf("\nBefore calling the function, number = %d\n", number);


  change_number(number);


  printf("After calling the function, number = %d\n\n", number);


  return 0;
}
```

In order to change the value of an actual argument, we must pass its memory address, not just its value.

(call by reference)

05_functions/change_value_correct/change_value_correct.c

```
$ gcc -o change_value_correct change_value_correct.c

$ ./change_value_correct
Before calling the function, number = 1
Inside the function, the number's value is 2
After calling the function, number = 2
```

```c
#include <stdio.h>

// Function Definition
void change_number(int *i){
  *i = 2;
  printf("Inside the function, the number's value is %d\n", *i);
}

// Main Function
int main(){

  int number = 1;
  printf("\nBefore calling the function, number = %d\n", number);


  change_number(&number);


  printf("After calling the function, number = %d\n\n", number);


  return 0;
}
```

Remember, the * used declare the pointer variable, i, in the function argument is different than the * used within the body of the function. To be clear,

`int *i`

- The * here is simply because this is how you declare a pointer to an integer.

`*i = 2`
`printf(" … %d\n", *i)`

- The * in these statements is the dereference operator, which allows you to access the value of the variable associated with the memory address.

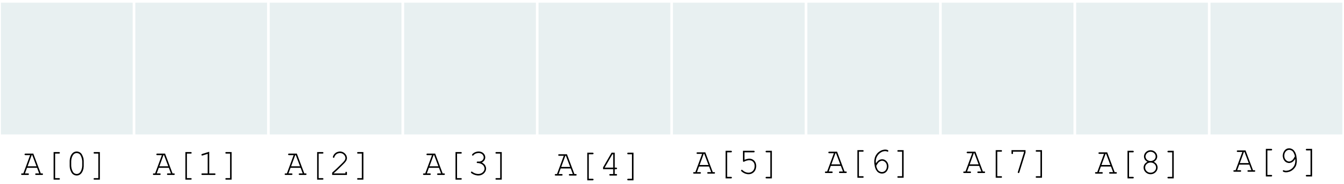
"Call by reference"

Pause for Exercise

- Compile and run the code in 05_functions, 06_addresses_and_pointers
- Complete the exercises in 08_exercises/02_pointers and 08_exercises/03_functions

C Arrays

Data structure that holds a fixed number of data elements of a specific type




```
int A[10];     // declares an array of 10 integers
```

C Arrays

Data structure that holds a fixed number of data elements of a specific type

Each element is 4 bytes for int

7	32	256	17	-20	22	1	0	59	-2
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

```
int A[10];      // declares an array of 10 integers

A[0] = 7;       // assigns values to the array elements
A[1] = 32;
A[2] = 256;
A[3] = 17;
A[4] = -20;
A[5] = 22;
A[6] = 1;
A[7] = 0;
A[8] = 59;
A[9] = -2;
```

```
printf("The value of A[3] = %d\n", A[3]);
```

The result would be:

```
The value of A[3] = 17
```

Memory Allocation

- Stack
  - Region of computer memory that stores temporary variables
    - When a new function is called the variables are created on stack
    - When the function returns, the memory is returned to the stack (LIFO)
  - Memory managed for you
  - Variables can only be accessed locally
  - Variable size must be known at compile time

- Heap
  - Region of compute memory for dynamic allocation
    - No pattern to allocation/deallocation (user can do this any time)
  - Memory managed by user
    - E.g. using malloc(), free(), etc.
  - Variables can be accessed globally
  - Variable size can be determined at run time

07_memory_allocation/static.c

```c
#include <stdio.h>

int main(){

// Statically-allocated array of floats
  int N = 5;
  float f_array[N];

  for(int i=0; i<N; i++){
    f_array[i] = 0.25*i;
  }

  for(int i=0; i<N; i++){
    printf("f_array[%d] = %f\n", i, f_array[i]);
  }

  return 0;
}
```

```
$ gcc -o static static.c

$ ./static
f_array[0] = 0.000000
f_array[1] = 0.250000
f_array[2] = 0.500000
f_array[3] = 0.750000
f_array[4] = 1.000000
```

07_memory_allocation/dynamic.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(){

// Dynamically-allocated array of floats
  int N = 5;
  float *f_array_dyn = malloc(N*sizeof(float));

  for(int i=0; i<N; i++){
    f_array_dyn[i] = 0.25*i;
  }

  for(int i=0; i<N; i++){
    printf("f_array_dyn[%d] = %f\n", i, f_array_dyn[i]);
  }

  free(f_array_dyn);

  return 0;
}
```

```
$ gcc -o dynamic dynamic.c

$ ./dynamic
f_array_dyn[0] = 0.000000
f_array_dyn[1] = 0.250000
f_array_dyn[2] = 0.500000
f_array_dyn[3] = 0.750000
f_array_dyn[4] = 1.000000
```

Allocates N*sizeof(float) bytes of memory and returns pointer to the block of memory

Releases block of memory associated with f_array_dyn

Pause for Exercise

- Compile and run code in 07_memory_allocation
- Complete the exercise in 08_exercises/04_allocation

Additional Resources

- Exercises that go with these slides (as well as some examples to work through)
  - https://github.com/olcf/foundational_hpc_skills

- Other sites
  - https://en.cppreference.com/w/c/language
  - https://en.wikibooks.org/wiki/C_Programming
  - https://stackoverflow.com/questions/tagged/c
  - Many other tutorials can be found by googling “c programming language”

- Website with many practice problems
  - https://projecteuler.net/

Examples Used in These Slides

The slides and code examples used in these slides (as well as the exercises) can be obtained from OLCF's GitHub…

On the OpenDTN node, you should have cloned the repository to the following location:

```
/ccsopen/home/<username>/foundational_hpc_skills
```

From there

```
$ cd intro_to_c
```

The exercises to try on your own can be found in `intro_to_c/08_exercises`


Questions?