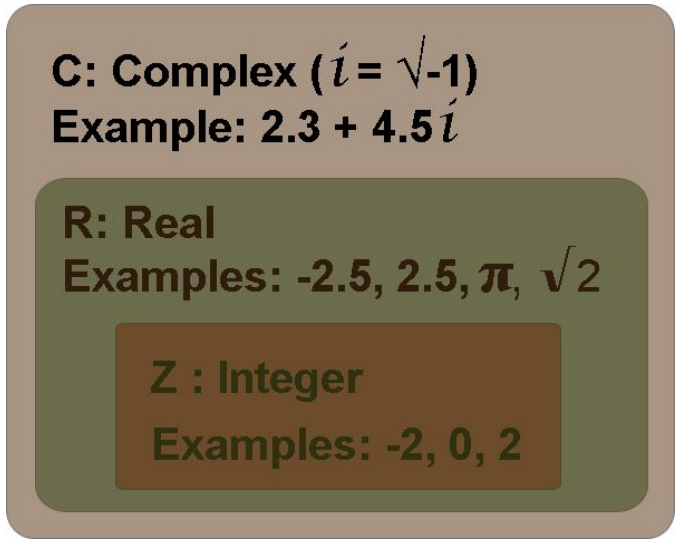
C Programming

Basic Data Types

Basic Data Types in C

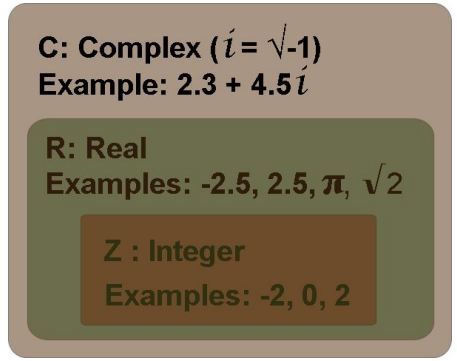
- An object (variable) refers to a location in memory
  - Its content can represent values.
  - Its type specifies the amount of occupied memory and how the values are stored in
    - E.g. in four bytes it is possible to store unsigned integers, float numbers, 4 characters strings and ect.
- Data types in C
  - Basic types
  - Enumerated types
  - Union types
  - Structure types
  - Pointer types
  - Array types
  - Function types



Different types of numbers

Basic Data Types in C

- Basic Data types
  - Integer types
    - Signed and Unsigned
  - Floating-point types (defined in IEEE 754 standard)
    - Single precision (6 decimal digits)
    - Double precision (15 or 19 decimal digits)



Different types of numbers

Data type	Keyword	Size	Range
Character	char	1 byte	-128 to 127
Integer	int	2, 4 or 8 bytes (platform dependent)	For 4 bytes: -2,147,483,648 to 2,147,483,647
Single precision floating-point	float	4 bytes	±3.4E+38
Double precision floating point	double	8 bytes	±1.7E+308
Valueless	void		
Boolean (C99)	_Bool	1 byte	0 or 1 (false or true)

Basic Data Types in C

- C99 introduced the unsigned integer type _Bool to represent boolean values (0 or 1).
  - The boolean value true is coded as 1, and false is coded as 0.
  - To use bool, true and false macros you need to include stdbool.h
- Type modifiers (signed, unsigned, short and long)
  - Using type modifiers we can change size and signedness of the basic types
  - By default the char, int, float and double data types are signed (can cover ± numbers)
    - signed modifier may be used only with char.
    - The char data type is used for plain characters. But signed char and unsigned char are used as a signed/unsigned byte. For the other types it shall be omitted
  - unsigned can only be used with int and char data types
  - long can only be used with int and double data types
  - short can only be used with int data type
  - Type modifiers shall be written before the data types. E.g. *unsigned short int i = 0;*

Fundamental types

Basic Data Types in C

Type	Size	Range
(signed) char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
unsigned int	2, 4 or 8 (platform dependent)	For 4 bytes: 0 to 4294967295
short int	2 bytes	-32,768 to 32,767
unsigned short int	2 bytes	0 to 65,535
long int	4 or 8 bytes (platform dependent)	For 4 bytes: -2,147,483,648 to 2,147,483,647
unsigned long int	4 or 8 bytes (platform dependent)	For 4 bytes: 0 to 4294967295
long long int	8 bytes	-9,223,372,036, 854,775,808 to 9,223,372,036, 854,775,807
unsigned long long int	8 bytes	0 to 18,446,744,073, 709,551,615
long double	10, 12 or 16 (platform dependent)	For 10 bytes ±1.1E+4932

Basic Data Types in C

- Standard C defines only the minimum storage sizes of the other standard types
  - Use sizeof operator to get the actual size of the data types in bytes.
    - sizeof(*type*) or sizeof expression. E.g. sizeof(long int)
  - You can find range of the integer types in *limits.h*
  - You can find range of the floating-point types in *float.h*

- Integer types with exact width (C99)
  - For portability sake C99 has defined integers with exact width in stdint.h
  - int8_t, int16_t, int32_t and int64_t are signed integer types with exact 8, 16, 32 and 64 bits
  - uint8_t, uint16_t, uint32_t and uint64_t are unsigned integer types with exact 8, 16, 32 and 64 bits

Basic Data Types in C

- The void type indicates that there is no value.
  - Can not be used to declare void variables or constants
  - Use void when a function returns nothing. E.g. void func(int a);
  - Use void when a function has no parameter. E.g. int func(void);
  - Use void in cast operations to discard the value of an expression
    - E.g. (void)printf("Hello World!"); or (void)var; and ect.
  - It is possible to declare a pointer to void.
    - A void * represent the address of an object, not its type. E.g. void *ptr;
- C11 supports types for complex numbers in complex.h. But they are optional.
  - float _Complex, double _Complex, long double _Complex are defined.
  - If the __STDC_NO_COMPLEX__ macro is defined, it indicates that these types are not supported

Declaration of Identifiers

- A declaration specifies the properties of one or more identifiers
- The identifiers you declare can be names of variables, functions, and etc. E.g. int var;
  - You know that identifiers have their own scope
  - All identifiers except labels shall be declared before we use them
  - There are some kinds of declaration
    - Declaration of tag names for unions, structures and enumerations or enumeration constants
    - Declarations of one or more variable or function identifiers
    - typedef declarations, which declare an alias for another data type
      - typedef type identifier; E.g. typedef unsigned char byte;
- Declaration of Variables
  - The general form of variable declaration is: ***type variable [, variable [, ...]];*** E.g. int a, b;

Declaration of Variables

- Generally there are two types of variable
  - **Global** variables which are defined outside of all functions and has file scope
    - We should avoid using global variables or else justify their usage
  - **Local** variables which are not global and have non-file scope
    - Before using local variables we shall initialize them. E.g. unsigned int var = 0;
- It is possible to specify a ***storage class*** for a variable.
  - A storage class determines where a variable is stored and the scope of the variable
  - There are four storage classes: ***auto***, ***extern***, ***register***, ***static***
  - If we use a storage class, it shall be written before the type. E.g. static char chr;
- The auto storage class sets the scope of a variable to local
  - It is the default storage class for local variables and it can be omitted.

Declaration of Variables

- The extern storage class is used to declare a reference to a **global** variable defined in another file and in this way we can have access to the variable in all the program files.
  - We can not use extern to get access to a file scope variable whose storage class is static.
- The register storage class with a variable
  - Asks the compiler to use a register of the CPU to store the variable.
    - This is a request and there is no guarantee.
    - The registers of a processor are very faster than the main memory
      - Using a register of the CPU for a variable improves the performance.
      - Normally this optimization is done by the compiler.
- The static storage class with a variable makes the variable like a global variable but with scope of file or function.

Declaration of Variables

- If the scope of a static variable is function, it is alive for the lifetime of the program and the value of the variable is not lost with multiple calls to the function.
- const, volatile, restrict and _Atomic type qualifiers in C
  - It is possible to use a type qualifier to modify properties of a variable. E.g. const int var = 20;
  - The const type qualifier is used to make a constant variable.
    - Such a variable must be initialized and it can not be changed after the initialization
  - The volatile type qualifier is used to make variables which are read always from memory.
    - Unlike register storage class. The compiler does not optimize such variables
    - The volatile type qualifier is used to declare shared variables
      - Which can be modified by multiple processes
      - We always want to have the latest value of a variable

Declaration of Variables

- The restrict type qualifier
  - Is only used with variable pointer types. E.g. int * restrict ptr;
  - Tells the compiler that the pointer is the only way to access the object pointed by it
    - The compiler doesn't need to add any additional checks and it can optimize the program
  - Doesn't add any new functionality.
    - The compiler may ignore it without affecting the result of the program.
  - A pointer can not be restrict and volatile.
- The const and volatile type qualifiers shall be written before types of variables.
  - E.g. const int var = 10; extern volatile int status;
- To get the address of a variable in the memory the & operator is used.
  - E.g. int var = 0; printf("The address of var in the memory is: %p\n", &var);

Declaration of Variables

- The _Atomic type qualifier is used to declare atomic objects.
  - Support for atomic objects is optional.
  - If **__STDC_NO_ATOMICS__** macro is defined, it means that atomic objects are not supported
- An atomic object is a race free object which can be read or modified without getting interrupted by concurrent threads. A declaration example: _Atomic int counter;
- You can use types, macros and functions in ***stdatomic.h*** to work with atomic objects.
  - For example to initialize counter: _Atomic int counter = ATOMIC_VAR_INIT(0);
- Array and function types cannot be atomic.
- An atomic object with a struct or union type should only be read or written as a whole
- An atomic operation is typically a ***read-modify-write*** operation. E.g. The ++, --, or += operators
- In *stdatomic.h* there are functions like ***atomic_store()*** to do operations on an atomic object.

Literals

- A literal is a hard coded constant. E.g. “Hello World!”
- A literal can be an integer, a floating-point number, a character, or a string.
- Type of a literal is determined by its value and its notation.

- Integer Constants
  - Type of an integer literal by default is int. E.g. 2021
  - An unsigned int literal can be expressed using the suffix u or U. E.g 2021U
  - A long int literal can be expressed using the suffix l or L. E.g 2021L
  - An unsigned long int literal can be expressed using the suffix ul or UL. E.g 2021UL
  - A long long int literal can be expressed using the suffix ll or LL. E.g 2021LL
  - An unsigned long long int literal can be expressed using the suffix ull or ULL. E.g 2021ULL

Literals

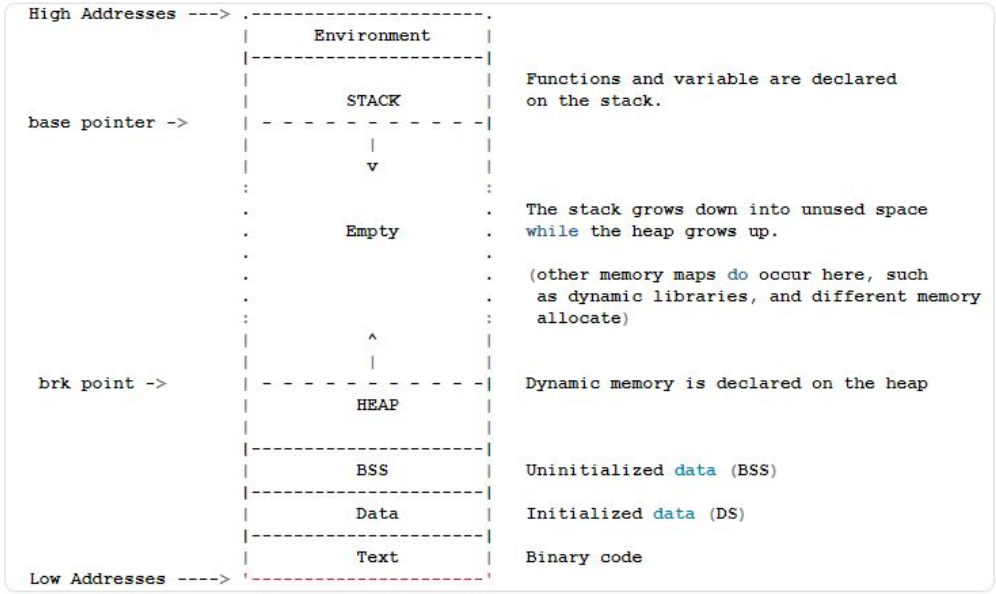
- A character constant is a character enclosed in single quotation marks.
  - E.g. 'A', '\n', '\\', 'a'
- Floating-Point Constants
  - The default type of a real number is double. E.g. 3.1425
  - A float literal can be expressed using the suffix f or F. E.g 3.1425F
  - A long double literal can be expressed using the suffix l or L. E.g 3.1425L
- A String Literal is a sequence of characters enclosed in double quotation marks.
  - E.g. "Hello World!"
- The prefixes 0b, 0 and 0x/0X are used to express numbers in binary, octal and hex forms
  - 0b10101010U is a binary unsigned int
  - 0x2fUL is a hexadecimal unsigned long int

Enumerated Types

- Enumerations are integer type and are defined by the enum keyword
  - enum [identifier] { enumerator-list }; E.g. enum colors { RED, BLUE, GREEN };
  - The identifier is the tag name of the enumeration and it can be omitted
  - The identifiers in the enumerator list are the **enumeration constants** and their type is int
    - We can use the constants in our code
    - Values of the constants start with 0 and incremented automatically by one.
      - The value of RED is 0, BLUE is 1 and GREEN is 2
    - It is possible to change the default values of the constants.
      - E.g. enum colors { RED, BLUE = 5, GREEN }; // RED is 0, BLUE is 5 and GREEN is 6
    - Different constants in an enumeration may have the same value
      - But the value of an implicitly-specified enumeration constant shall be unique. For example
      - enum colors {RED = 3, BLUE, GREEN = 4}; is not OK. enum colors {RED = 3, BLUE = 4, GREEN = 4}; is OK

C Programming - Memory Map

- Memory Map of C Programs
  - **Code segment (text)**: Contains
    - The program code (read only and shared)
  - **Data segment**: Contains
    - Initialized global and static variables (DS)
      - Read only segment. eg. const and strings
      - Read-write segment
    - Uninitialized global and static variables (BSS)
      - They get initialized by the compiler to zero
  - **Heap**: Used for dynamic memory allocation
  - **Stack**: Used for
    - Local and temporary variables (auto)
    - Function arguments and calls



Memory Layout of a C Program

Environment is the used for the arguments passed to the app

BSS (Block Started by Symbol) includes the uninitialized global and static variables. **DS** includes the initialized constants global and static variables and strings.

Run ***size app*** command to get the memory info of **app**