Queens' College Cambridge

Programming in C and C++

Alistair O'Brien

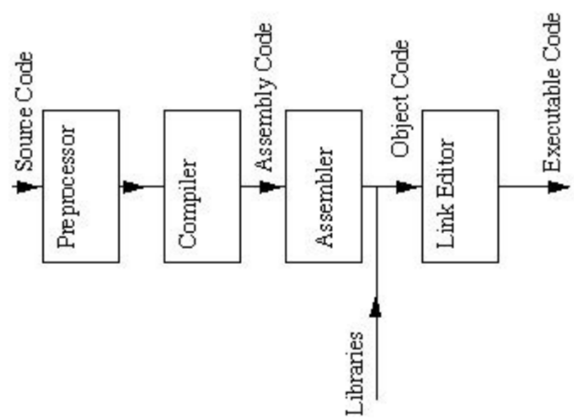
Department of Computer Science

April 21, 2021

Contents

1 The C Language

1.1 The C Compiler



1.1.1 The Pre-Processor

- Performs purely *string-based* operations on C source code using *pre-processor* directives.

- #define directive:

  - Syntax:

    ```
    #define <macro_name> <replacement>
    ```

  - All occurrences of <macro_name> (not in strings) are replaced w/ <replacement>.

- **Note**: Brackets for expressions. `do/while(0)` trick for statements.

- `#undef` directive:
  - Syntax: `#undef <macro_name>`
  - Undefines a macro w/ name `<macro_name>`
  - Used for *local* macros

- `#include` directive:
  - Syntax: `#include location` where `location ::= filepath | <filepath>`
  - `filepath` is a *relative filepath* from *current file*. `<filepath>` is a filepath from an *include directory* predefined via `-I` option (used for `stdlib` + project headers).
  - Copies contents from file w/ `filepath` into current file.

- Conditional directives:
  - `#if` e ... `#endif` directives: e is a constant expression (compile-time evaluated) of integer type. If e evaluates to non-zero integer $\implies$ retain text between `#if` and `#endif` directives.
  - `defined(<macro_name>)` predicate: returns non-zero if `<macro_name>` is defined (in current scope).

```
#ifdef <macro_name> ≜ #if defined(<macro_name>)
#ifndef <macro_name> ≜ #if !defined(<macro_name>)
```

- `#include`-pattern:
  - Define header file:

```
// header.h
#ifndef HEADER_H
#define HEADER_H

...

#endif // HEADER_H
```

  - Prevents double-declaration when using includes.

- Error control:
  - #error "<string>" directive: raises a preprocessor error if executed.
  - __LINE__ and __FILE__ store the current line and filename. Used for debugging. e.g:

    ```
    #define panic(...) kernel_panic(__FILE__, __LINE__,
        __func__, ##__VA_ARGS__)
    ```

- String manipulation:
  - Stringizing operator #: applied to #define-d macro parameters in <replacement> creates string containing parameter source-code e.g.

    ```
    #define PRINT_EXPR(e) printf("\%s = \%d.", #e, eval(e))
    ```

  - Token-pasting operator ##: concatenates tokens together:

    ```
    #define JOIN(a, b) (a ## b)
    ```

  - Consecutive string literals are concatenated:

    ```
    "Hello" "World"
    ```

1.1.2 Object Files

Definition 1.1.1. (**Object File**) An object file is the output of a compiler/assembler containing *position independent assembly/machine code*

- Output of the assembler phase of C compiler.
- Compiled code but contains metadata for linking to an *executable file*.
- *Segmentation*: object files split into sections. Sections are used by *loader* and *linker*.
  - header: Contains descriptive information for object file w/ offsets to other segments.

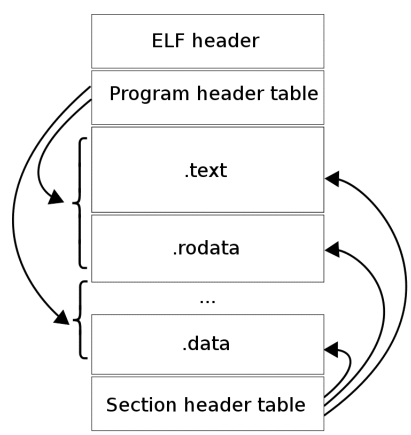
- `.text` segment: Contains the compiled assembly w/ position independent labels
- `.data` segment: Initialized `static` / `global` variables.
- `.rodata` segment: Initialized `const static` variables / constants
- `.bss` segment: Uninitialized `static` variables
- `.strtab` segment: Stores string literals
- `.symtab` segment: Stores visibility of each declaration, declaration name (pointer to `.strtab` section), section index and virtual address (in section).

1.1.3 Linking

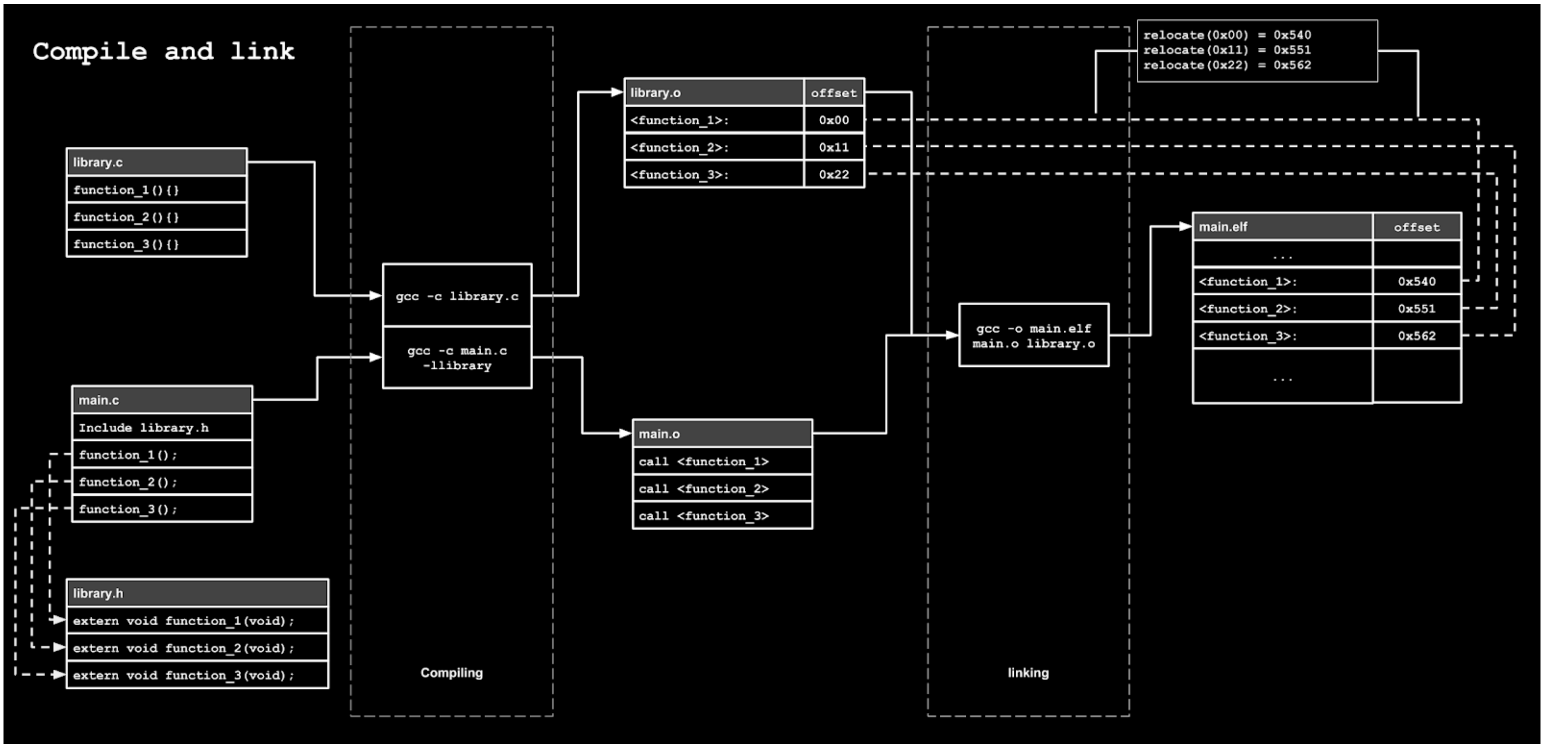
- Compiling multiple files $\implies$ multiple separate *object files*.

Definition 1.1.2. (**Linking**) Linking is the process of combining (*linking*) many object files to form an *executable* (binary).

- ELF (Executable and Linkable Format): standard UNIX representation of C object files (and executables)



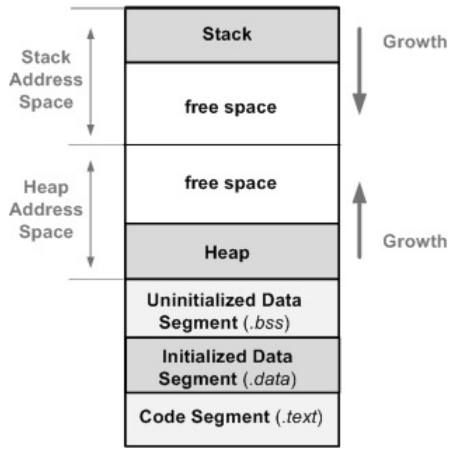
- Segmentation approach: merge individual approach (calculating linked addresses)



- Linking `.text`:
  - Object containing `main` function is placed w/ offset `0` in linked `.text` section
  - Other `.text` sections relocated w/ offsets after `main` `.text` section. `.symtab` is used to determine offsets (and addresses) of external labels.

1.1.4 Runtime Memory Layout

- A *loader* loads an executable into a process address space w/ the following layout:



- **Note**: The stack grows down and the heap grows up.

1.2 Syntax and Semantics

1.2.1 Primitive Types and Modifiers

- C has 4 primitive types:

Primitive Types	Size	Description
`char`	≥ 8 bits	Smallest data type
		Stores *integers* (and characters using ASCII)
`int`	≥ 16 bits	Basic integer type, stores integers in range
		$[-2^{15}, 2^{15} - 1]$
`float`	*unspecified* ($\sim$ 32-bit)	Single precision floating point type
`double`	*unspecified* ($\sim$ 64-bit)	Double precision floating point type

- C has 4 *type* modifiers: `signed`, `unsigned`, `short` and `long`

Type	Size	Description
`signed char`	≥ 8 bits	Stores integers in range $[-127, 127]$
`unsigned char`	≥ 8 bits	Stores integers in range $[0, 255]$

`short`
`short int`
`signed short`
`signed short int`
	≥ 16 bits	Short integer, stores integers in range $[-2^{15}, 2^{15} - 1]$

`unsigned short` `unsigned short int`	≥ 16 bits	Stores integers in range $[0, 2^{16}]$

`long`
`long int`
`signed long`
`signed long int`
	≥ 32 bits	Long signed integers. Stores integers in range $[-2^{31}, 2^{31} - 1]$

`unsigned long` `unsigned long int`	≥ 32 bits	Long unsigned integers. Stores integers in range $[0, 2^{32}]$

`long long`
`long long int`
`signed long long`
`signed long long int`
	≥ 64 bits	Long long signed integers. Stores integers in range $[-2^{63}, 2^{63} - 1]$

`unsigned long long` `unsigned long long int`	≥ 64 bits	Long long unsigned integers. Stores integers in range $[0, 2^{64}]$
`long double`	*unspecified* ($\sim$ 98, or 128 bits)	Extended precision floating point

- `long` widens the types. `short` narrows the types.
- C99 standard defines fixed-width integer types `<stdint.h>`:
  - `intn_t` and `uintn_t` for $n \in \{8, 16, 32, 64\}$
  - `intptr_t` and `uintptr_t`
  - `intmax_t` and `uintmax_t`
- Literals:

Type	Literals	Examples
`char`	See `int`	
`int`	Decimal, ASCII, hexadecimal, binary or octal	`12, 'a', 0x34`
`long`	Add `L` suffix	`12L`
`float`	Uses `.` or `e/E` with `f` suffix	`12.0f`
`double`	Uses `.` or `e/E`	`12.0`

1.2.2 Variables and Modifiers

Definition 1.2.1. (**Declaration and Definition**) A *declaration* provides a variables *type* and *identifier*. A definition reserves a memory location for a *declaration*

- Variables (and functions) are *declared* before use and *defined* **once**.
- C variables are often declared and defined simulatenously:

```
// In global scope: declares and defines x
int64_t x;

// In local scope: declares and defines y
void main() {
    int64_t y;
}
```

- Variables may be optionally *initialized*:

```
int64_t x = 1;
```

- C variables have 4 variable modifiers:
  - 2 scoping modifiers: `extern` and `static`
  - 2 optimization modifiers: `const`, `volatile`

`extern`:

- Declares the variable *but* doesn't define it. Used for code reuse between object files.
- If the linker cannot find a unique definition for declared variable $\implies$ linker error.
- *All declarations in global scope are implicitly* `extern`.

```
// global scope: declares variable x
extern int x;

void main() {
    // declares variable x (points to global definition)
    extern int x;
    ...
}
```

`static`:

- In global scope: `static` limits the variable declaration to the *current* object file. (not placed in symbol table after compilation). Like a `private` modifier.
- In local scope: `static` variable values persist between function invocations (unlike non-static variables whose values are lost after `return`). Like `static` modifier from Java. Similar to global variable, however scope is limited to the function.
- Stored in `.data` segment (or `.bss` if uninitialized)

- `const`: Value of variable cannot change during runtime. Initialized once. Used for optimization (and function parameters that indicate immutability).
- `volatile`: Value of variable can be modified by *non-local* code (or hardward) e.g. MMIO / Kernel. Used to prevent *unsound optimizations*.

1.2.3 Operators

- C has arithmetic, bitwise and boolean (logical) operators.
- Infix notation is used $\implies$ precedence (and associativity)

Precedence	Associativity	Operator	Description
0	Left	`++ --`	Postfix increment and decrement
		`()`	Function call
		`[]`	Array subscripting
		`.`	Structure / union member access
		`->`	Structure / union member access via pointer
1	Right	`++ --`	Prefix increment and decrement
		`+ -`	Unary positive / negation
		`! ~`	Boolean and bitwise `not`
		`(`τ`)` e	Type casting
		`* &`	Dereference and Address-of
		`sizeof`	Size-of type
2	Left	`* / %`	Multiplication, Division and modulo
3	Left	`+ -`	Addition and subtraction
4	Left	`<< >>`	Bitwise left and right shifts
5	Left	`< <=`	$<$ and $\leq$ relations
		`> >=`	$>$ and $\geq$ relations
6	Left	`== !=`	$=$ and $\neq$ relations
7	Left	`&`	Bitwise and
7	Left	`^`	Bitwise xor
8	Left	`|`	Bitwise or
9	Left	`&&`	Logical and
10	Left	`||`	Logical or

- If operators have the same precedence then associativity is used to determine AST w/ left-to-right parsing. (See compilers)
- Closure of operators over variables $V \implies$ set of expressions (denoted e).
- Assignment `=` is an expression: returns value assigned.

- Sequential expressions:

  e_1 `,` ... `,` e_n

  Evaluates e_1 to e_n (left to right order), evaluating to value of e_n.

- Ternary operator:

  e_1 `?` e_2 `:` e_3

  Evaluates e_1, if non-zero then evaluate to e_2 else e_3. e_1 has an integer type.

1.2.4 Statements and Control Flow

- Statements (denoted s) :
  - An *expression statement* is of the form: s `::=` e;
  - An *compound or block statement* is of the form:

    s `::= {` (s `| declaration` d`;)* }`

- Control flow also defines the set of expressions.

- `if`-statements:
  - Syntax:

    `if (`e`)` s_1 `else` s_2

    where s_1 is the *then*-statement and s_2 is the else-statement.
  - e must have an integer type. s_1 is executed if e evaluates to a non-zero integer.

- `switch`-statements:
  - Syntax:

```
switch (e) {
    case e'_1: s_1
    ...
    case e'_n: s_n
    default: s_d
}
```

  e has an integer type. e'_i are *constant expressions* (evaluated at compile time).

  - Evaluate e. Match e w/ case compile-time evaluated e'_i (value v_i), then execute case statement s_i. If no matches, execute s_d (default case).
  - Statement `break;`: breaks out of the enclosing `switch`. Fall-through behavior implemented: No break in $s_i \implies$ other case statements s_j $j > i$ are executed.

- `while`-loops:
  - Syntax:

    `while (`e`)` s

  - e has integer type. Evaluate e, if non-zero execute s and repeat.
  - Statement `break;` breaks out of the enclosing `while`-loop
  - Statement `continue;` Jumps to the end of the loop body.

- `for`-loops:

$$\texttt{for (}e_1\texttt{; }e_2\texttt{; }e_3\texttt{)}\ s \triangleq \begin{array}{l} e_1\texttt{;} \\ \texttt{while (}e_2\texttt{) \{} \\ \quad s\texttt{;} \\ \quad e_3\texttt{;} \\ \texttt{\}} \end{array}$$

- `do-while`-loops:

$$\texttt{do}\ s\ \texttt{while (}e\texttt{);} \triangleq \begin{array}{l} s \\ \texttt{while (}e\texttt{)}\ s \end{array}$$

- `goto`
  - Syntax:

    `goto` l`;`

    l is a label. Label statements: s `::=` ℓ`:`
  - Unconditional jump to statements prefixed by label statement l :.

1.2.5 Arrays and Strings

Definition 1.2.2. (**Array**) An array τ `x[n]` is a block of memory of size `sizeof(τ) * n`

- Array declaration:

  τ `x[n];`

- Array definition:
  - In global scope: allocated to `.data` (if initialized w/ `{ ... } / "..."`) (or `.bss`).
  - In local scope: allocated on the *stack*
- If uninitialized *and* in local scope: *must* zero the array using `memset`.
- Indexing:
  - Indexed at 0. No runtime bounds checking
  - `x[e]` where e has integer type.

Definition 1.2.3. (**String**) A string is a *null terminated character array*

- String syntax: `"Hello World"` (standard escape chars).
- String declaration:

  ```
  char_t str[] = "Hello World"
  char_t str[n] = "Hello"
  ```

  where $n \geq 6 = 5 + 1$.

- String functions defined in `<string.h>`: `strcpy, strcat, strlen, strcmp`.
- Pre-Processor supports additional string manipulation. See section ??.

1.2.6 Pointers

Definition 1.2.4. (**Pointer**) A pointer is an address to memory.

- `sizeof(`τ`*)` = width of the address bus in bytes.
- Address-of operator: `&`x, address of variable / function.
- Dereference operator: `*`e, e has a pointer type.
- Pointer declaration:

  τ `* modifiers` x`;`

  where `modifiers ::= const | volatile`. `*` binds to the variable and modifiers, *not the type* τ.

- **Pointer Arithmetic**:
  - Pointer operators: `++ -- + -`
  - Arithmetic:

    `ptr_`x $\oplus$ e `= (`τ`*)((int)ptr_`x $\oplus$ e `* sizeof(`τ`))`

    Units are known as τ-positions.
  - Pointers may be compared using operators:

    `== != > >= < <=`

- **Function Pointers**:
  - Functions have pointer types w/ declaration:

    τ `(*`x`)(`τ_1`, ..., `τ_n`);`

    Type definitions:

    `typedef` τ `(*function_t)(`τ_1`, ..., `τ_n`);`
  - Invoking function pointer:

    `(*`x`)(`e_1`, ..., `e_n`)`

- **Relation to Arrays**:

- Pointers used to implement array operations:

  τ `x[n];`

  $$x \triangleq \texttt{\&}x\texttt{[0]}$$
  $$x\texttt{[}e\texttt{]} \triangleq \texttt{*(}x \texttt{ + } e\texttt{)} \equiv \texttt{*(\&}x\texttt{[0] + }e\texttt{)}$$

- **Distinction**: Array refers to the block of memory. Identifier of the array x is a *pointer* to the start of the block of memory $\implies$ "pass-by-reference" implementation for arrays (via pointers implicitly).

- Void pointers:
  - All pointers may be (implicitly) converted to `void` pointers.
  - `void*` denotes a pointer to object of *unknown type*. Used to implement "unsafe polymorphic" / allocator functions.

- Null pointers:
  - Pointers without an *initialized* address.
  - Special value `NULL` (typically `0`) (trapped by the OS).

1.2.7 Functions

Definition 1.2.5. (**Function**) A function is a *labelled* block code that returns a value

- Declaration:

  $$\tau\ x\texttt{(}\tau_1\texttt{, ..., }\tau_n\texttt{);}$$

  Optional identifiers in declaration for parameters

- Definition:

  $$\tau\ x\texttt{(}\tau_1\ x_1\texttt{, ..., }\tau_n\ x_n\texttt{)}\ s$$

- Call-by-value ("pass-by-value") *semantics* (w/ the exception of arrays (which implicitly used *pointers*))

- `void` is used to denote:
  - no return type. Used to implement *procedures*
  - no arguments

```
// no return
void foo(int x);

// no arguments
int zero(void);
```

- **Note**: `()` denotes *no type checking* applied to function declaration / definition parameters.
- Variable arguments use ... syntax. See `<stdarg.h>`.

1.2.8 Structures, Unions and Enumerations

Definition 1.2.6. (**Structure**) A structure in C is a collection of one or more members (associated w/ a field) defined as a single block of memory referenced by a single variable.

- Declaration:

```
struct x {
    τ1 x1;
    ...
};
```

  Tag x declares a *new type* `struct` x:

```
// variable y w/ struct x type
struct x y;
```

- Definition: allocated as a contiguous block of memory of size `sizeof(struct x)` bytes.

$$\text{sizeof(struct } x) = \sum_{i=1}^{n} \text{sizeof}(\tau_i)$$

  Struct types may be used *anonymously* (inlined) in declarations:

```
struct {
    τ₁ x₁;
    ...
} y;
```

- Initialization:

```
struct foo { int y };

struct foo x = { 1 }; // compound literals
struct foo x = { .y=1 }; // or

struct foo x;
foo_init(&x, 1); // constructor like procedure
```

- Common to use pointers to structs: `struct foo *`x`;`
- Operators:
  - Member access: e`.`x:

    ```
    struct foo x = { 1 };
    x.y; // evaluates to 1
    ```

  - Member access via pointer: e`->`x. e has a pointer type.

    e`->`x $\triangleq$ `(*`e`).`x

- Structs may be recursive (via pointers):

```
typedef struct tree {
    int label;
    struct tree *left;
    struct tree *right;
} tree_t;
```

Definition 1.2.7. (**Typedef Declaration**) A `typedef` declaration provides a way of declaring a *type-alias*

- Declaration:

```
typedef τ x_t;
```

x`_t` is an alias of τ.

- **Bit Fields**:
  - `struct` fields may have a defined *bit length*:

    ```
    struct {
        ...
        τ x: n
    }
    ```

    n-bit field of type τ.
  - Used for low-level representations. Useful for limited memory.
  - Bit field members have *no address* (since fields are not byte aligned).
  - Padding is often used by the compiler $\implies$ dangerous to use bit fields (use macros w/ shifts)

Definition 1.2.8. (**Union Type**) A union type is a type that may have several types $\{\tau_1, \ldots, \tau_n\}$.

A value of a union type represents a value of type τ_i. This value (and member type τ_i) may change during the runtime of the program.

- Declaration:

  ```
  union x {
      τ₁ x₁; // member
      ...
  }
  ```

  Tag adds `union` x type. (Union types may also be inlined).

- All members of a union are mapped to the same address: `&(`x`.`x_i`) == &(`x`.`x_j`)`

- Definition: All members of the union share the same block of memory of size `sizeof(union` x`)`:

$$\text{sizeof(union } x) = \max\{\text{sizeof}(\tau_i) : 1 \leq i \leq n\}$$

- Address mapping $\implies$ altering value of 1 member alters value of all other members.
- Programs must ensure unions are handled w/ correct type conversion $\implies$ tagged union structure:

```
typedef struct {
    tagged_union_type_t type;

    union {
        struct {
            τ11 x11;
            ...
        }
        ...
    }
} tagged_union_t
```

Definition 1.2.9. (**Enum Type**) An enumeration type is a type whose values have an underlying *integer* representation.

- Declaration:

```
enum x {
    x1=e1, // notice comma
    ...
}
```

  x_i constant identifier. e_i is a *constant expression* (compile type evaluated). e_i are optional $\implies$ x_i is assign value enumeration value $i-1$ (starting at 0).

  x_i must be *distinct* but e_i may be equal.

- `bool` is a enum type (C has no primitive `bool` type):

```
typedef enum {
    false,
    true
} bool;
```

1.3 Unspecified and Undefined Behavior

Definition 1.3.1. (**Unspecified Behavior**) Unspecified behavior is defined as behavior, for a well-formed program w/ correct data, the *depends* on the *implementation* of the C Compiler.

- *or* the C standard provides two or more possibilities, *but* does not impose requirements on which behavior should be implemented.
- Unspecified behavior $\implies$ compiler implementers have more platform specific freedom. e.g. ability to utilize parallelism etc.
- Unspecified behavior makes the C standard more adoptable for different platforms
- **Examples**: Order of evaluation of infix operators e.g. `f() + g()`. Order of evaluation of function arguments.
- C compiler implementation documents the *implementation-defined behavior* (not necessary for *unspecified behavior*).

Definition 1.3.2. (**Undefined Behavior**) Undefined behavior is behavior, for a erroneous program (or data), that depends on the implementation of the C compiler.

- Undefined behavior is due to platform dependent features e.g. arithmetic
- Undefined behavior *is specified* by the C standard $\implies$ compilers may optimize code w/ undefined behavior. e.g. `(INT_MAX + 1) < 0` is optimized to `0`.
- **Examples**: Integer overflow. Dereferencing a null pointer.

2 Assorted C Topics

2.1 Dynamic Memory Allocation

- C Dynamic Memory Allocation (`<stdlib.h>`):

```
// Allocates contiguous block of size bytes. Returning pointer to start.
// Warning: Block is not initialized (use memset)
// Returns null pointer if error
void* malloc(size_t size);

// Frees a previously allocated block.
// Warning: if ptr doesn't point to allocated block ⟹ undefined behavior
// If ptr is NULL ⟹ no effect
void free(void* ptr);
```

  Implemented by OS (see TinyOS)

- Each pointer `ptr` can only be freed *once* (otherwise undefined behavior)
- *Dangling pointers* (pointers to freed memory) must not be used after being freed.
- **Problem**: Memory leaks, ensuring programs free all allocated memory.

2.1.1 The Arena Pattern

- **Solution**: `arena` pattern.
- An *arena* is a block based allocator in which each arena is only freed once.

```
typedef struct arena {
    size_t size;
```

```
    size_t current_size;
    void* block;

} arena_t;


void* arena_init(size_t size) {
    arena_t* arena = malloc(sizeof(arena_t));
    if (!arena) return NULL;

    arena->size = size;
    arena->current_size = 0;

    // Allocate block of managed memory to arena
    arena->block = malloc(size * sizeof(node_type));
    if (!arena->block) { free(arena); return NULL; }

    return arena;
}

void* arena_malloc(arena_t* arena, size_t size) {
    arena_t* prev = arena;
    do {
        if((arena->size - arena->current_size) >= size) {
            arena->current_size += size;

            // Calculate pointer using offset (initial current_size)
            return (void*)((uintptr_t)arena->block
                + (arena->current_size - size));
        }

        // Failed to allocate, repeat
        prev = arena;
    } while ((arena = arena->next) != NULL);

    // Failed to allocate, allocate new arena
    size_t arena_size = max(DEFAULT_ARENA_SIZE, size);
    arena_t* next = arena_init(arena_size);
```

```
    // Add linked list pointer
    prev->next = next;

    // Allocate in new arena
    next->current_size += size;
    return next->block;
}

void arena_free(arena_t* arena) {
    arena_t *next, *prev = arena;
    do {
        next = prev->next;

        // free block and arena header
        free(prev->block);
        free(prev);

        prev = next
    } while (next != NULL);
}
```

- **Problems**:
  - Explicit freeing. Programmer must ensure arena is freed after use (otherwise Memory leak)
  - Increasing working set sizes and memory inefficiency

2.2 Garbage Collection

- **Problem**: Memory leaks
- **Solution**: Automatically managed memory $\implies$ Garbage collectors.

Definition 2.2.1. (**Heap Graph**) A heap graph is a graph $G = (V, E, R)$ where V is the set of vertices modelling nodes on the heap, edges $(u, v) \in E$ denote a pointer from u to v and $R \subseteq V$ is the set of root nodes, the set of nodes w/ pointers stores in registers / on the stack.

- A node $v \in V$ is *garbage* iff there does not exist $u \in R$ s.t $u \rightarrow^* v$.

2.2.1 Reference Counting

- **Idea**: For each heap node $v \in V$, store *reference count* $\rho(v)$, the number of *pointers* to the node. We free $v \in V$ if $\rho(v) = 0$, propagating a new reference count (recursively).
- **Example**: A reference counted binary tree `tree_t`:

```
typedef struct tree {
    size_t ρ;
    int value;
    struct tree *left;
    struct tree *right;
} tree_t;

tree_t* tree_empty = NULL;
tree_t* tree_init(int value, tree_t* left, tree_t* right) {
    tree_t* t = malloc(sizeof(tree_t));
    if (!t) return NULL;

    // Reference count initialized to 1 since
    // t is a pointer to the tree node ( ⟹  a reference)
    t->ρ = 1;
    t->value = value;

    // New reference to left, so increment
    t->left = left;
    inc(left);

    ...

    return t;
}

void inc(tree_t* t) {
    if (t != NULL) t->ρ += 1;
```

```
}

void dec(tree_t* t) {
    if (t != NULL) return;

    if (t->ρ > 1) {
        t->ρ -= 1;
    } else {
        // Decremented count ρ(v) = 0 ⟹ free!
        dec(t->left);
        dec(t->right);
        free(t);
    }
}
```

- **Problem**: Management / responsibility of incrementing reference counts.
- **Solution**: (Partial) Encapsulation via setters and getters manage the reference counts:

  ```
  tree_t* get_left(tree_t* t)
  void set_left(tree_t* t, tree_t* left);
  ```

  Still requires explicit `dec`s after `get`s.
- **Problem**: Reference counting cannot free *garbage cycles* $\implies$ memory leaks.
- **Solutions**: Mark and Sweep, cyclic reference counting, etc

2.2.2 Mark and Sweep

- **Idea**: Traverse the heap graph G from the root set R, marking accessible nodes $v \in V$, denoted $\rho(v) = 1$. Iterate over V, freeing vertices $v \in V$ w/ $\rho(v) = 0$ (inaccessible).
- **Example**:

  ```
  typedef struct {
      ...
  ```

```
    list_node_t list_node;
} tree_t

void mark_tree(tree_t* t) {
    if (t != NULL && !t->ρ) {
        t->ρ = true;
        mark_tree(t->left);
        mark_tree(t->right);
    }
}

void mark(heap_t* h) {
    list_node_t* iterator;
    list_for_each(iterator, &h->roots) {
        mark_tree(LIST_VALUE(tree_t, list_node, iterator));
    }
}

void sweep(heap_t* h) {
    list_node_t *next, *prev = list_head(&h->nodes);
    do {
        next = prev->next;

        if (!(prev->ρ)) {
            list_delete(&h->nodes, prev);
            free(LIST_VALUE(tree_t, list_node, prev));
        } else {
            prev->ρ = false;
        }

        prev = next;
    } while (next != h->nodes.nil);
}

void gc(heap_t* h) {
    mark(h);
    sweep(h);
}
```

- Periodically execute `gc(h)`.
- **Problems**:
  - Requires the program to stop (concurrency issues) to mark and sweep
  - Inefficient (compared to reference counting).

	Reference Counting	Mark and Sweep
Collection	Incremental	Batch
Cost Per Assignment	High	Low
Delays	Short	Long
Collects Cycles	No	Yes

2.3 Tooling and Optimization

2.3.1 Tooling

- Address Sanitizer (*ASan*):
  - Adds additional runtime checks to detect errors:
    * Out-of-bounds array access
    * Double-freeing
    * Using stack stored data once popped
    * Memory leaks
  - Compiler option: `-fsanitize=address`
  - **Problems**: $\times 2$ performance slowdown due to additional checks Doesn't catch all uninitialized memory access (*undefined behavior*).
- Memory Sanitizer (*MSan*):
  - Adds runtime checks for uninitialized memory accesses.
  - Compiler option: `-fsanitize=memory`
  - **Problems**: $\times 3$ performance slowdown

- Undefined Behavior Sanitizer (*UBSan*):
  - Adds runtime checks for undefined behavior. e.g. integer overflow, dereferencing null pointers, etc
  - Compiler option: `-fsanitize=undefined`
  - **Problems**: $\times 1/5$ performance slowdown. Doesn't catch all undefined behavior.
- ASan, MSan and UBSan augment the compiler $\implies$ requires source code to add checks.
- Valgrind:
  - Adds runtime checks to *compiled executables*.
  - `valgrind` procedure is invoked when runtime check fails.
  - **Problems**: Substantial overhead.

2.3.2 Cache Optimizations

- **Problem**: Pointers introduce indirection $\implies$ breaks principle of locality. Increases cache misses.
- Generic list representation:

  ```
  typedef struct node {
      void *head;
      struct node *tail;
  } list_t;
  ```

  `list_t` has 2 pointers $\implies$ increases cache miss rate.
- **Solutions**:
  - Intrusive lists: Add `list_node_t` to list item struct (See TinyOS):

    ```
    typedef struct {
        double x;
        double y;
        list_node_t list_node;
    } point_t
    ```

Removes `void* head` pointer. `tail` pointer still may result in cache miss + overhead of `list_node_t` $\implies$ decreases data density and increases miss rate.

– Arrays of `structs`:

```
typedef point_t* point_array_t;
```

No-longer stores tail pointers. Contiguous block of elements. However, cannot incrementally build lists (since array size must be known).

– `struct` of arrays:

```
typedef struct {
    double* xs;
    double* ys;
} point_vector_t
```

`xs` and `ys` are contiguous arrays that may each have separate cache lines.

Array of `structs` and `struct` of arrays are resizable via a *dynamic array implementation*.

3 The C++ Language

- An extension of C with OOP features

3.1 C++ Features

3.1.1 Lifetime, Storage Modifiers and Temporaries

Definition 3.1.1. (**Lifetime**) All values v in C++ have a *lifetime* $[t_1, t_2]$, the point of execution where the value v is allocated t_1, and where it is freed t_2.

- Storage modifiers dictate the lifetime and allocation of values in C++ (See section ??)
- Extends storage modifiers as C: `static, extern, register, auto` with small semantic differences and `thread_local`:
  - `auto`:
    - Automatic linking storage (on the stack). Only applicable in function body scope.
    - Adds *basic* type inference support e.g. `auto` x `= 'a';`.
  - `thread_local`:
    - Each thread has a local copy of the value during the threads execution.
    - Lifetime defined by thread's lifetime (and other storage modifiers).
  - `register`: Removed by C++.

Definition 3.1.2. (**Temporary**) A temporary is an unnamed value (the result of some types of expressions).

- The temporary's lifetime is defined by the execution times of the expression.

```
string a("Hello"), b("World");
const char *s1 = (a + b).c_str(); // Bad

// s1 is a pointer, however, the pointed too memory location
// has been freed since the lifetime of the temporary (a + b)
// ended after (a + b).c_str() was evaluated.
```

- Temporaries often lead to *bugs* due to implicit lifetimes.
- Temporary value creation (focused on *objects*):
  - Passing an object by value to a function. Uses the *copy constructor* to create a temporary on the stack.
  - Returning a object by value from a function.
  - During the evaluation of an expression in assignment. (See above)

3.1.2 Types, `structs`, `unions` and `enums`

- C++ primitive types extend C's: `char, int, float, double` w/ type modifiers `signed, unsigned, short, long` with the following differences:
  - Character literals `'a'` are of type `char` (not `int`).
  - `bool` primitive type added w/ literals `false, true`:

Boolean Literal	Casting to `int`	Casting from `int`
`false`	0	0
`true`	1	non-zero integers

- `struct`, `enum` and `union` declarations with tag x adds a type x into the current scope (instead of adding type `struct` x, etc).
- `enum`s are a distinct type (instead of an alias for integer constants):
  - Declaration: `enum` x `:` τ `{` x_1 `=` e_1`, ... }` where e_i are constant expressions of type τ, an integer type.

– Bitmap pattern: allocate `enum` constants w/ powers of 2:

$$\texttt{enum } x \texttt{ : } \tau \texttt{ \{ } x_1 \texttt{ = 1, } x_2 \texttt{ = 2, ..., } x_n = 2^{n-1} \texttt{ \}}$$

Defines an `enum` integer range of $[0, 2^n - 1]$.
Integer values $y \in [0, 2^n - 1]$ s.t $y = \sum_{i=0}^{n-1} k_i 2^i$ where $k_i \in [0, 1]$ represent the set of enum constants $\{x_i : k_i \neq 0\}$

```
// access's integer range [0, 7]
enum access : int { r = 1, w = 2, x = 4}

access rwx = (access)7; // represents enum constants r, w and x
```

For unsigned constants: lower bound is 0. For signed: least negative power of 2.

3.1.3 Casts

- Explicit type conversions (casts) have the syntax:

$$(\tau)\ e$$
$$\tau\ (e)$$

$\tau(e)$ is a *functional cast expression* and is syntactic sugar: $\tau(e) \triangleq (\tau)e$.

- C++ casts:

  – `dynamic_cast<`τ`>(`e`)`: e has a pointer / reference type to an *object* type.

    Adds runtime-checks using run-time type information (RTTI) w/ casting within an (up or down) inheritance hierarchy.

    Returns `nullptr` if the cast fails (for pointers) and throws a `bad_cast` exception (for references).

  – `static_cast<`τ`>(`e`)`: C-style cast semantics. May perform upcasts and downcasts on object pointers/references. Also implements `void*` casting and `int, float, double` and `enum` casting.

  – `reinterpret_cast<`τ`>(`e`)`: e has a pointer/reference type. Casts *any* pointer type to any other pointer type. Casts pointers to integers (and vice versa).

  - `const_cast<`τ`>(`e`)`: Modifies the `const` modifiers of a pointer. e.g. pass a `const` pointer to a non-`const` pointer argument.

- C++ compiler typically *attempts* to compile C-style casts into C++ casts.

3.1.4 References

- **Idea**: Safer pointers $\implies$ references

Definition 3.1.3. (**References**) A reference is an *alias* to an already existing value with automatic dereferencing.

- Implemented using `const` pointers (addresses to the value)

	References	Pointers

Initialization	`// Correct`
	`int x = 10;`
	`int& y = x;`
	
	`// Incorrect`
	`int& y;`
	`y = x;`
	
	References *must* be initialized when declared.
		`// Correct`
		`int x = 10;`
		`int *y = &x;`
		
		`// Correct`
		`int *y;`
		`y = &x;`

Null	References cannot be `NULL`	Pointers can be assigned to `NULL`
Dereferencing	References are automatically dereferenced `y++`	Pointers must be dereferenced w/ the `*` operator `(*y)++;`

Reassignment	References cannot be reassigned	Pointers may be reassigned:
		
		`int x = 5, y = 6;`
		`int* z = &x;`
		`z = &y;`

Arithmetic	References cannot be manipulated using arithmetic operators	Pointer arithmetic. See section ??

3.1.5 Functions and Operators

- Functions may be *overloaded*: several functions in the same scope with the same name but different type parameter prototypes (signature).
- Overload resolution: Uses the most specific type parameter prototype. If a unique match cannot be determined, a compiler error is raised.
- Integer type implicit conversion is used.

```
void f(long x);

f(1) // 1 (int) is promoted to long
```

- Operators may also be overloaded:

```
class A {
    bool operator==(const A &b) const { ... }
}
```

- Functions may also have *default parameter values* w/ declaration: $\tau\ x$=e where e is a constant expression.
- **Problem**: Non-default parameters must be *before* default parameters:

```
// Good
int foo(int a, int b=10);

// Bad
int foo(int a=10, int b);
```

3.1.6 Scope resolution `::`, `new` and `delete` Operators

- Scope resolution operator `::` used for qualified variables: `namespace::name` Empty namespace is the *global namespace scope*.

```
int x;
void foo() {
    int x;
    x = 5; // local reference
    ::x = 10; // global reference
}
```

- `::` is also used to lookup *class/struct/union* names.
- C++ runtime replaces `malloc` and `free` w/ the `new` and `delete` operators.
- `new` τ:
  - Syntactic sugar: `new` $\tau \triangleq$ `malloc(sizeof(`τ`))`.
  - Returns a *pointer*. Null pointer if unable to allocate the block (null check required).
- `delete`:
  - Syntactic sugar: `delete` $e \triangleq$ `free(`e`)`.
  - No effect if e is a null pointer.
- Additional operators: `new` τ`[`n`]` and `delete[]` e used for arrays.
- Initializers:
  - Arrays: Compound literal initializer used for array of undefined length. e.g. `new int[]{1,2,3}`
  - Classes/Structs: Constructor arguments used w/ `new` operator. e.g. `new FooBar(1, 2);`.

3.1.7 Namespaces and C Libraries

Definition 3.1.4. (**Namespace**) A namespace is scope prefixed by a *namespace name*.

- Used for larger projects (avoids prefixing all functions) and expressing the structure of the project.
- Declarations:

```
// Defines a namespace in the defining scope w/ namespace name x
namespace x {
    declarations
    ...
}
```

```
// Defines a namespace in defining scope and adds an implicit
// using-directive "using x" in the defining scope
inline namespace x {
    declarations
    ...
}

// Anonymous namespace. Compiler adds a unique namespace name x
// and implicit using-directive "using x" in the defining scope.
// Note: namespace is not accessible via :: operator
namespace {
    declarations
    ...
}

// Namespace alias. x is a alias for the qualified namespace
// name y.
namespace x = y
```

- Namespaces may be *extended* (so-called *extending namespaces*) if the same name for two distinct namespace declarations is used. Compiler compiles into a single namespace (accessible via `using`)

- `using`-directive:

  - Directive: `using namespace` x
  - Implicitly qualifies all available declarations from x in the current scope.
  - **Note**: Doesn't *add* any *declarations* to the current scope. Just provides access.
  - `using`-directive is *transitive*:

    ```
    namespace D {
        int d1;
        void f(char c);
    }
    using namespace D; // Adds D::d1, D::d2, E::e1 to global scope
    ```

```
                    // namespace definition "lifting"


namespace E {
    int e1;
}
namespace D { // extending namespace
    int d2;
    using namespace E; // adds transitivity
}
```

- `using`-declaration:
  - Declaration: `using` x_1, ..., x_n; where x_i are qualified namespace declarations.
  - Adds the declarations for $x_1, \ldots, x_n$ to the current scope.

    ```
    namespace B {
        using A::foo; // adds declaration for foo in ns B
    }
    ```

  - May be used in body local scopes $\implies$ declarations *not* added.
- Linking C code w/ C++ requires a `extern "C"` modifier.
- C library headers (used by C++ code) use the preprocessor pattern:

  ```
  #ifdef __cplusplus
  extern "C" {
  #endif

  ...

  #ifdef __cplusplus
  }
  #endif
  ```

- Required due to "name munging" for overloading functions and differing calling sequence semantics.

3.2 Objects and Classes

3.2.1 Classes and Structures

- C++ extends `struct`s:
  - Declaration:

    ```
    struct x {
        τ1 x1 // member fields
        ...

        τ1^f f1(τ11 x11, ...); // member functions
        ...
    }
    ```

    Defines a type x in the defining scope w/ fields and *methods*.
  - Definition: Struct member fields are allocated storage. Struct member functions (*methods*) are resolved *statically*.
- C++ adds `class`es. Identical to `struct`s. However, the default access modifier for `struct`s is `public` and the default for `class`es is `private`
- Access modifiers: `private`, `protected` and `public`.
- Storage modifiers: `static` fields.
- Member function (methods) definitions may be provided outside the declaration using `::` operator, or inside the declaration.
- Methods may be *overloaded*.
- **Constructor**:
  - A constructor is a special non-static method used to initialize objects w/ declaration:

    ```
    class x {
        ...
        x(τ1 x1, ...);
    }
    ```

and definition:

```
x(τ₁ x₁, ...)
    : initializer_list { body }
```

- All classes have a *default constructor* (performs minimal initialization via `bzero`) if no explicit constructor.
- `private` constructors prevent external instantiation (Singleton pattern)
- Constructors may be overloaded.

- **Destructor**:
  - A destructor is a special non-static method executed at the end of an object's lifetime w/ declaration:

```
class x {
    ...
    ~x();
}
```

  - Destructors may be *virtual* (required with inheritance)
  - Called at end of lifetime. e.g.
    * Out of scope (assuming `auto` storage modifier)
    * `delete` operator
    * Stack unwinding (exceptions)

- `friend` access modifier:
  - **Problem**: Some classes are tightly connected $\implies$ require `public` / `const` getters for internal class state.
  - **Solution**: `friend`s.
  - Declaration:

```
// τ is (class|struct) type. τ has access to private and
// protected members
friend (class|struct) τ;

// f is a qualified function (not a member of current class)
```

```
// f has access to private and protected members.
// Could share w/ specific class method e.g. y::g or
// non-member function e.g. main
friend τ f(τ1 x1, ...);
```

3.2.2 Objects

Definition 3.2.1. (**Object**) An object in C++ is an instance of a `class/struct`

- **Note**: Difference between Java and C++, objects don't have implicit references to heap.
- Declaration:

```
// τ is an object type w/ constructor parameters e1, ...
τ x(e1, ...);
```

- Definition:

```
// Allocated statically (e.g. stack / .data. See storage modifiers)
τ x(e1, ...);

// Allocated at runtime (on heap)
new τ(e1, ...);
```

- Members may be accessed via the `.` operator via a class/struct value (or `->` for pointers).
- `this` pointer: An implicit non-static method parameter of type $x*$ for class x.
- Definitions of non-static methods resolve non-static member fields x of the class (or subclass) w/ transformation $x \triangleq$ `this->`x.

```
struct S {
    int n;
    void f();
}
void S::f() {
    n = 1; // transformed into this->n = 1
}
```

- `const` methods: Adds a `const` qualifier to the implicit `this` pointer. Prevents modifying object members. Only `const` member functions may be used by a `const` object.

```
struct Array {
    int *data;
    Array(size_t size) : data(new int[size]) {}

    // const member function
    int operator[](int i) const {
        return data[i];
    }

    // non-const member function. Overloadable due to const difference
    int& operator[](int i) {
        return data[i];
    }
}

Array x(1); const Array y(1);
x[0] = 0; // Good. type of x[0] is int&
y[0] = 0; // Bad. type of y[0] is int. Not lvalue.
```

3.2.3 Special Member functions

- **Copy constructor**:
  - A copy constructor of class x is a constructor whose first parameter of type x`&`:

    ```
    class x {
        ...
        x(x& src); // typically add const modifier
    }
    ```

  - Executed:
    - Initialization $x\ a$ `=` b or $x\ a(b)$ where $b : x$.
    - Object pass-by-value: $f(a)$ w/ declaration `void` $f(x\ b)$ and $a : x$.

    * Returning an object
  - Default copy constructor: initializes object w/ copies of all non-static member fields of `src`.

- **Assignment operator**:
  - A copy assignment operator of class x is a non-static method w/ name `operator=` whose first parameter is of type x or x`&`:

```
class x {
    ...
    x& operator=(x src);
    // or
    x& operator=(x& src); // typically add const modifier
}
```

  - x`& operator=(`x` src)` refers to a copy-and-swap assignment operator.
  - Executed whenever an object is on the left side of an assignment expression (lvalue)
  - Default assignment operator: overwrites of all non-static member fields of `this` w/ `src`'s fields

3.2.4 Inheritance

- A derived class (or struct) inherit all of the members of it's base class.
- Declaration:

```
class d : b {
    τ_1 x_1;
    ...
}
```

  declares a derived class d from class b.

- Definition: Class d is allocated storage of size:

  `sizeof(`d`) = sizeof(`b`) + `$\sum_{i=1}^{n}$` sizeof(`τ_i`)`

Additional storage is required for `derived` objects, allocated for `derived`'s member fields and `base`'s member fields.
(Not implemented using *pointers*, unlike Java)

- All `protected, public` members of b are accessible inside d. All `public` members of b are accessible outside of d.
- Access modifiers of base class:
  - `public` b: Inherited `public, protected` members of b remain `public, protected` in d
  - `private` b: Inherited `public, protected` members of b are `private` in d
- Default modifiers: `class` uses `private`. `struct` uses `public`.
- Constructors and Destructors:
  - Derivation chain: The base class constructor is executed before the derived class constructor. (vice versa for destructors):

```
struct A {
    A() {
        std::cout << "A's constructor was called" << std::endl;
    }
    ~A(){
        std::cout << "A's destructor was called" << std::endl;
    }
}

struct B : A {
    B() {
        std::cout << "B's constructor was called" << std::endl;
    }
    ~B() {
        std::cout << "B's destructor was called" << std::endl;
    }
}

B b();
```

```
// cout: A's constructor was called
// cout: B's constructor was called
```

- Access modifiers: If the derived class cannot access constructor (`private`) $\implies$ compiler error
- Constructors w/ parameters may be explicitly used via an initializer list:

```
Derived::Derived(τ x)
    : Base(x) { ... } // passes parameter x to constructor of Base
                      // constructor body of Derived is executed on Base
```

- Methods of the derived *and* base class are resolved statically.
- **Problem**: Dynamic polymorphism not possible via static resolution of methods.

3.2.5 Virtual Methods

- **Solution**: Virtual methods.
- Declaration:

```
class x {
    ...
    virtual τ f(τ1 x1, ...);
}
```

declares the non-static method f as *virtual* and supports dynamic polymorphism (dispatch)

- Definition: Allows derived classes to override behavior of virtual methods (`override` modifier). Non-virtual methods are resolved statically.

  Virtual methods are resolved dynamically (at runtime) using a *virtual table* (or *vtable*), stored by each derived class object (including base class) IMAGE

  Dynamic dispatch procedure:

  - Fetch `vtable` pointer

  - Lookup address a of virtual function call
  - Jump to a

- **Problem**: Pointer indirection adds additional runtime overhead.
- `vtable` also stores Runtime Type Information (RTTI) e.g. `typeid` (See `<typeinfo>`)
- `override`-modifier:

  $$\tau\ f(\tau_1\ x_1,\ \ldots)\ \texttt{override;}$$

  Override modifier ensures non-static method is *virtual* and matches the overridden signature.

- Virtual methods allow for *abstract classes*:

  ```
  class b {
      public:
      virtual τ f(τ1 x1, ...) = 0;
  }
  ```

- Classes w/ `default` virtual methods (*an abstract method*) cannot be instantiated.
- Virtual destructors:
  - Required when dynamic polymorphism is used, otherwise the base class destructor is statically resolved.

    ```
    struct Base {
        virtual ~Base() { ... }
    };

    struct Derived : Base {
        ~Derived() { ... }
    }

    Base* d = new Derived;
    delete d; // vtable lookup to Derived::~Derived()
              // releases resources of Derived class
              // then calls Base destructor
    ```

3.2.6 Multiple Inheritance

- C++ classes may inherit multiple base classes.
- Declaration:

```
class d : b_1, ..., b_n {
    ...
}
```

- Explicit field name required if there is a field name clash:

```
d.b_i::x // statically resolves to member x in inherited b_i in d
d.b_i::f() // statically resolves to method f() in inherited b_i in d
```

- Diamond problem: base classes have a common base class:

```
class A { ... }
class B : public A { ... }
class C : public A { ... }
class D : public B, public C { ... }
```

  By default, D will have two instances of A:

```
d.B::A // B's A object
d.C::A // C's A object
```

- `virtual` base classes:
  - Declaration:

```
class d : virtual b { ... }
```

    b is a virtual base class of d.
  - Definition: ensures the derived class d only contains one base class b sub-object. Even if b occurs multiple times in the inheritance hierarchy (provided it is inherited `virtual` every time)

```
struct A { int n; }
struct X : virtual A {};
struct Y : A {};
```

```
struct XY : X, Y {};
XX xx;
xx.X::n = 1; // modifies the virtual A subobject
xx.Y::n = 2; // modifies the non-virtual A subobject
```

3.3 Exceptions and Templates

3.3.1 Exceptions

Definition 3.3.1. (**Exception**) An object that may be thrown and caught. Used to implement improved error handling.

- C++ provides exceptions to communicate errors (instead of error codes)
- `throw` expressions:
  - Declaration:

    `throw` e

  - Definition:
    1. Evaluates e of type τ to v, uses value v to initialize an *exception object*
    2. Passes exception object to *exception handler* w/ matching type τ.
  - Exception handler chaining may be used via `throw;`. Rethrows the currently handled exception
  - Stack is unwound until we reach the matching exception handler. Destructors for `auto` objects are invoked w/ stack frames being popped.
- `try-blocks`:
  - Declaration:

    ```
    try { ... } catch (τ e) { ... }
    try { ... } catch (...) { ... }
    ```

    A catch-clause that matches any exception w/ type τ, binding to parameter e, and a catch-clause that matches *any exception*

- Multiple catch blocks may be used to catch different errors (searched in order).
- **Note**: implicit reference conversion on matching types:

```
try { throw 1; }
catch (const int& e) {
    std::cout << e << std::endl;
}
```

3.3.2 Template Metaprogramming

- Templates support *metaprogramming* (compile time evaluation) and *generic programming* (parametric polymorphism).
- Templates are *turing complete*.
- Template specification:

  ```
  templete <p_1, ..., p_n> d
  ```

  where parameters p_i are either:

  - Non-type parameters of the form: τ x.
  - Type parameters of the `typename` x or `class` x.

  ```
  template <typename T>
  struct List { ... }
  ```

- Template parameters may be dependent:

  ```
  template <typename T, T min> ...
  ```

- Templates may be specialized. Specialized templated must appear after the non-specialized template declaration. Specialized templates provide the same declaration w/ a reduced parameter specification:

  ```
  template<typename T> struct A { ... }
  template<> struct A<int> { ... } // specialization
  ```

- **Advantages**:

- Improves type safety (generic programming, avoiding `void*`)

- **Disadvantage**:
  - Template expansion results in larger executables